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

N° d'ordre : 3621

THESE

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

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

Discipline : Informatique
Spécialité : Intelligence Artificielle

Présentée et soutenue le 23/04/2022 par :

BENAMRANE AMINE

Repair algorithms for Centralized and Distributed Dynamic Constraints Reasoning

JURY

BELLAFKIH Mostafa	**PES, INPT, Rabat**…….....	**Président**
EZZAHIR Redouane	PES,ENSA, Agadir…….....	Rapporteur / Examinateur
BENMENSOUR Rachid	PH, INSEA, Rabat……...	Rapporteur / Examinateur
MAHMOUDI Abdelhak	PH, ENS, Rabat……...	Rapporteur / Examinateur
BOUYAKHF El Houssine	Expert, PES, Faculté des Sciences, Rabat …	Expert
HIMMI Mohammed Majid	PES, Faculté des Sciences, Rabat……..	Directeur
BENELALLAM Imade	PES, INSEA, Rabat…….....	Encadrant

Année Universitaire : 2021/2022

Dedication

Words alone cannot express the thanks I owe to the one above all of us, the omnipresent God, for answering my prayers for giving me the strength to plod on despite my constitution wanting to give up and throw in the towel, thank you so much, Dear Lord.
I would like to thank my parents J.B and Z.F and all my family for believing and loving me. They have always filled my life with generous love, and unconditional support, and prayers. To them, i dedicate this thesis. Last but not the least, A special thanks are addressed to my wife A.Y for his help, support, and patience throughout the preparation of this thesis, with who I began by sharing some ideas, and today I'm sharing with her all my life. To who I dedicate this thesis. Finally, i would like to thank all my friends and everybody important to the successful realization of this thesis, as well as expressing my apology that I could not mention personally one by one.

May 6, 2022

BENAMRANE Amine

Acknowledgements

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

First and foremost, I gratefully acknowledge Pr.**BELLAFKIH Mostafa** (PES, INPT Rabat) for honoring my jury by presiding it. I am thankful that among all his activities, he accepted to preside on the jury of my dissertation.

I would like to thank also Pr.**EZZAHIR Redouane** (PH, ENSA, Agadir) for the examination of the dissertation. Thanks a lot for being a jury and being part of my thesis defense.

In the same way, i wish to thank Pr.**BENMENSOUR Rachid** (PH, INSEA, Rabat) for accepting to be a reporter for this thesis. I am thankful that during all his activities, he accepted to review my thesis.

I would like to thank Pr.**MAHMOUDI Abdelhak** (PH, Ecole Normale Supérieure de Rabat) for accepting to be a reporter for this thesis and the precious comments.

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

And, i would also like to warmly thank Pr.**Mohammed Majid HIMMI** (PES, Faculty of

Sciences, Mohammed V University in Rabat) for accepting to be my second supervisor and for the help, the facilities, the insights, and the comments he gave.

Finally. It is with immense gratitude that I acknowledge all the support, advice, and guidance of my co-supervisors, Pr.**Imade Benelallam** (PH, INSEA, Rabat). It was a real pleasure to work with him. His expertise has exceptionally inspired and enriched my research thesis. I hope to keep up and develop our collaboration in the future. I am also glad that he gave me many teaching opportunities which provided me an excellent environment to develop my pedagogical skills.

May 6, 2022

BENAMRANE Amine

Abstract

Constraint programming (CP) is one of the formalisms of artificial intelligence to deal with complex combinatorial problems qualified as NP-complete. The CP formalism is an elegant formalism that separates the modeling and the solving processes and through its various variants, it is able to solve many real problems. In this thesis, we extend the state of the art of constraint programming by studying some practical cases and proposing several resolution algorithms. We start by modeling a real planning problem, i.e. the planning of hospital internships at the CHU of Casablanca to resolve and optimize planning in terms of service occupancy as well as the preferences of future doctors. Then, because many real applications are dynamic, we are interested in dynamic CSP, we proposed an heuristic called Pdeg (Profond degree) which extends the classic heuristic deg and which exploits the depth of the constraint network when choosing the variable to modify. Then we experimented with this heuristic to improve the Partial-order Dynamic Backtracking PDB repair algorithm. Two algorithms IPDB and LRB have been proposed to repair the pre-disturbed solution of dynamic CSPs, quickly in terms of time, and efficiently in terms of effort and quality of the solution. Also in this thesis, to cover dynamic distributed CSPs, we propose on the one hand a new approach based on multi-Agent systems to model the intelligent autonomous market environment (sale and purchase of goods) to face the problems of e-markets. Then based on the famous ABT algorithm, we proposed the LiveABT algorithm to deal with disturbances in distributed CSPs in real time. Experiments have shown that the LiveABT algorithm outperforms the sequentially repair version performed by the DynABT algorithm.

Keywords: *Artificial Intelligence, Constraint Programming, Constraint Satisfaction Problem (CSP), Multi Agent System, Medical internships, Distributed CSP (DisCSP), Modeling, Optimizing, Intelligent Marketplace, Dynamic Distributed CSP (DDisCSPs).*

Résumé

La programmation par contraintes (PC) est un des formalismes de l'intelligence artificielle pour traiter les problèmes combinatoires complexes qualifiés NP-complet. Le formalisme PC est un formalisme élégant qui sépare les processus de modélisation et celui de la résolution et à travers ses diverses variantes, il est capable de résoudre de nombreux problèmes réels. Dans cette thèse, nous étendons l'état de l'art de la programmation par contraintes en étudiant quelques cas pratiques et en proposant plusieurs algorithmes de résolution. Nous commençons par modéliser un problème réel de planification i.e. la planification des stages hospitaliers du CHU de Casablanca pour résoudre et optimiser la distribution des étudiants en termes d'occupation des services ainsi que de préférences des futurs médecins. Puis, parce que de nombreuses applications réelles sont dynamiques, nous nous intéressons aux CSPs dynamiques, nous avons proposé une heuristique appelée Pdeg (Profond degree) qu'étend l'heuristique classique deg et qui exploite la profondeur du réseau de contraintes lors du choix de la variable à modifier. Puis nous avons expérimenté cette heuristique pour améliorer l'algorithme de réparation Partial-order Dynamic Backtracking PDB. Deux algorithmes IPDB et LRB ont été proposés pour réparer la solution pré-perturbée des CSPs dynamiques, rapidement en terme de temps, et efficacement en termes d'effort et de la qualité de la solution. Toujours dans cette thèse, pour couvrir les CSPs dynamiques distribués, nous proposons d'une part une nouvelle approche basée sur les systèmes multi-Agents pour modéliser l'environnement du marché autonome intelligent (vente et achat de biens) et pouvoir négocier dans les e-marchés. Ensuite basé sur le célèbre algorithme ABT, nous avons proposé l'algorithme LiveABT pour traiter les perturbations dans les CSPs distribués en temps réel. Les expérimentations ont montré que cet algorithme surpasse la version de réparation séquentiellement réalisée par l'algorithme DynABT.

Mots clefs : *Intelligence Artificielle, Programmation par contraintes, Systèmes Multi Agents, problèmes de satisfaction de contraintes (CSP), CSP Distribuées (DisCSP), Marché intelligent, Modélisation, Stages médicaux, Optimisation, CSP Distribuées Dynamique (DDisCSP).*

List of publications

INTERNATIONAL JOURNALS

— Lazy Repairing Backtracking for Dynamic Constraint Satisfaction Problems.
Y Acodad, **A Benamrane** I Benelallam, EH Bouyakhf.
Computing and Informatics journal. Vol.40, 2021, 1056–1079, doi:10.31577/cai202151056

— Constraint programming based techniques for medical resources optimization: Medical internships planning
A Benamrane, I Benelallam, EH Bouyakhf.
Journal of Ambient Intelligence and Humanized Computing (1–10) 2019 - Springer

— LiveABT: A Real-Time Repairing Protocol for Incremental and Dynamic DisCSPs.
Amine, BENAMRANE and Yosra, Acodad and Imade, Benelallam and El Houssine, Bouyakhf.
International Journal of Artificial Intelligence 15 (126-146) 2017.

— On improving the dynamic constraint satisfaction problems repairing.
Yosra, Acodad and **Amine, BENAMRANE** and Imade, Benelallam and El Houssine, Bouyakhf.
International Journal of Artificial Intelligence 15 (79–101) 2017.

INTERNATIONAL CONFERENCES AND WORKSHOPS

— Modeling trainings in Faculty of Medicine and Pharmacy of Casablanca as Constraint Satisfaction Problem.
A Benamrane,Y Acodad, I Benelallam, EH Bouyakhf, Mohammed, Bennani Othmani.
The 14^{th} International Workshop on Constraint Modelling and Reformulation(14–20)(CP 2014).

— Profound Degree: A Conservative Heuristic to Repair Dynamic CSPs
Y Acodad, **A Benamrane**, I Benelallam, EH Bouyakhf.
IFIP International Conference on Artificial Intelligence Applications and Innovations (140–149), 2014 - Springer.

— A Distributed Constraint Reasoning approach towards intelligent marketplace environment.
Z Erraji, A Hakkou, **A Benamrane**, I Benelallam, EH Bouyakhf.
The 16^{th} International Workshop on Constraint Modelling and Reformulation (CP 2016).

Contents

CHAPTER

1 Introduction

Artificial intelligence today is one of the most studied fields and will grow continuously in the future that gathers many various actors such: Academics, industries, societies, governments. This is due to the recent revelations that the IA comes with last years, just to indicate AI publications represented 3.8% of all peer-reviewed scientific publications worldwide in 2019, up from 1.3% in 2011 [Zhang et al., 2021]. In particular reasoning problems using SAT, Operational Research (OR), or also Logic Programming techniques to solve some everyday problems.
Among these formalisms, Constraint Programming (CP) is one of them. CP is an elegant formalism that separates between Modeling and solving processes. Modeling a problem consists of representing the problem in a set of variables connected with some constraints. CP formalism offers various extensions to cover a large problem structure such as Discrete and Continue CSP, static and dynamic CSP, centralized and distributed CSP, Valuated and fuzzy CSP, etc. So, solving a constraint problem is the process of searching for a complete assignment for all variables that do not violate any constraints.
Constraint programming has been growing rapidly in the last two decades due to its simplicity and its flexibility to represent many real everyday problems that can be clustered into classes such as Recommendation Systems, Scheduling, Sensor Network, Smart Grid, and Smart Homes, Supply Chain Management and Traffic Flow Control [Fioretto et al., 2018] are ubiquitous in all sector of activity, and they are qualified NP-hard thus there is no efficient algorithm to solve them.
Indeed. CP community has and continues to develop algorithms to deal with this problem's kind, these approaches can be separated into two categories: complete and incomplete ones. Complete approaches have the advantage to detect soonly inconsistency. Otherwise, their complexity grows exponentially depending on the problem's size. While Incomplete approaches are frequently used to find solutions for large size consistent problems but they are not able to prove problems inconsistency. In this thesis, we are primarily concerned with the first approaches' category.

Most of the popular industries such as Huawei, Cisco, Google, Baidu, Sony, etc. invest in developing CP solvers, they are present annually in the International XCSP3 Competition [Audemard et al., 2020] and International MiniZinc Challenge [MiniZinc, 2021] to assess the performances of constraint solvers. OR-Tools [Perron and Furnon, 2019], Picat [Zhou and Kjellerstrand, 2016] and Choco [Prud'homme et al., 2017] are the best solvers which won the last two editions.

As mentioned previously, CP can model and solve many real-life problems. In chapter 3 we will describe the Hospital internships problem which is a planning/resources allocation problem consisting of allocation of internship students to different hospital services undergoing several logistical and pedagogical constraints during the university calendar. And we will design it as Constraint Satisfaction Problems. Then, we propose an extended model (Constraint Optimization model) to cover the limitations of the CSP model and find the optimal scheduling taking into account students' pref-

erences. Finally, we show that using several What-If scenarios can help to suggest and recommend to decision-makers some adjustments to the problem inner rules to get the optimal solution.

To increase the efficiency of the constraint satisfaction algorithms, many techniques such as Backtracking Methods, Look back Methods, and consistencies propagation Methods using efficient representations and heuristics have been developed. However, these techniques assume that the complete description of the CSP instance is completely known and fixed in advance. This is a strong limitation when dealing with real situations [Verfaillie and Jussien, 2005], where problems may evolve due to changes in the environment nature, spurious actions, incomplete knowledge, etc.
chapter 4 present how Constraints programming designs the dynamic problem and present some popular approaches to deal with this kind of problems. Especially Heuristics-based methods which are the algorithms consisting of choosing which variable assign first and with which value according to the importance of the variable in the network and the domains' size. We distinguish between static and dynamic variable ordering heuristic (VOH) depending on how and when calculating the weights of the variables. For static ones, the weights are calculating once at the beginning of the search process and they don't be changed along with the search. Otherwise, the dynamic ones are calculating after every assignment.

The degree(deg) VOH [Dechter and Meiri, 1989],where variables are decreasingly ordered according to their initial and current degree (the number of neighbors of each variable), has a major advantage, it's that it does not compute every time dynamic information about the current state of research (such as the size of the current domain dom or the degree of the current variable ddeg), but it operates static information. However, in the context of repairing solutions, this heuristic appears limited. No information regarding the position of variables concerning the entire network is operated, whereby the change of a variable can affect not only the neighbors but all of the variables in the network with a given probability. For this, in chapter 5, we propose pdeg heuristic to calculate an estimation of this information for each variable X at the beginning of the research, considering all the network constraints.

The DCSP (Dynamic CSP) is an adapted CSP that deals with problems changes i.e., a restriction, which is an addition of constraints, or a relaxation, that consists of elimination of some existing constraints. To effectively repair Dynamic CSPs, Extended Partial-order Dynamic Backtracking (EPDB) uses ordering heuristics and retroactive data structures, *safety conditions* and nogoods, which are saved during the search process. In chapter 5, we show also that the drawback of EPDB is the exhaustive verification of orders, saved in *safety conditions* and nogoods, between variables. This verification affects remarkably search time, especially since orders are often indirectly deduced.
Therefore, we propose an improved version of EPDB, the Improved Partial-order Dynamic Backtracking (IPDB), which is a fast version of EPDB insofar as it deduces orders directly, by supporting an additive structure of orders, which allows having the information about all successors of each variable. Then we propose also a fast version of EPDB, the Lazy Repairing Backtracking (LRB) approach. This algorithm aims to repair solutions using EPDB advantages while eliminating unnecessary tests of orders.

As said before many real-life problems are naturally distributed or it's more desirable to model and deal with them distributedly. chapter 6 shows another advantage of using CP models to represent a problem that is in distributed setting. Distributed/Cooperative Constraints Satisfaction Problem (DisCSP) is a set of connected agents, each agent has a part of the problem. An agent can be seen as a Centralized CSP. Agents act autonomously to solve the problem i.e., find a solution for each agent without violating a constraint over agents. [Faltings and Yokoo, 2005] has described many reasons and motivations to prefer modeling a problem like a Distributed and not Centralized CSP: Cost of creating a central authority, Knowledge transfer costs, Privacy/Security concerns, Robustness against failure, etc.
With the growing success of e-marketplace adoption, the need for new intelligent approaches to support both buying and selling goods or services becomes a fundamental requirement. In the recent past, several techniques have been proposed, in different research areas, allowing the simulation of a market with a set of sellers proposing products and buyers who have a list of interests. In chapter 7,

we will describe the main problem of the e-marketplace environment, and the important issues to fully automated negotiation. And we address the distributed constraint reasoning model using privacy and relaxation concepts.

In chapter 9, we propose a new approach, called LiveABT, to solve Dynamic Distributed Constraint Satisfaction Problems. This new approach is based on the Asynchronous Backtracking (ABT) algorithm and real-time processing techniques that run the solver immediately against live perturbations.

At the end of this thesis, in chapter 10, we summarize all the performed work and present some perspectives and research paths.

CHAPTER

2 Centralized Constraint Programming

In this chapter we summarize fundamental concepts that we need in the following. Firstly, we present the constraint programming formalism either the satisfaction model and the optimization one, next we present some classical problems designed and solved by CP *approaches. Secondly, we present some popular strategies to find a solution/optimal solution. Finally, we show some famous heuristics used to boost the search process.*

Contents

2.1 Constraint Satisfaction/Optimization Problem

Many real problems are containing constraints, this component represents a specific need related to some parts of the problem. constraints are present in many aspects of life for example the number of hours per day is limited to 24 hours, and the next hour is the first one of the next day, etc. A model is an abstraction of a problem, which must represent all the problem details.

Definition 2.1 (CSP/COP). *A Constraint Satisfaction/Optimization Problem* (CSP)/COP *[Barták, 1999] is a 4 tuple* $<X, D, C, \bigtriangleup>$*, where:*

- X *is a set of* n *variables* $\{x_1, x_2, ..., x_n\}$.
- D *is a set of domains* $\{D(x_1), D(x_2), ..., D(x_n)\}$*, such that each variable* x_i *takes a value from its domain* D_i*.* D *is called the search space.*
- C *is a set of* m *constraints* $\{c_1, c_2, ..., c_m\}$ *between variables in subsets of* X*. Each* c_i *expresses a relation defining which variable assignment combinations are allowed for the variables in the scope of the constraint* $vars(c_i)$.
- $\bigtriangleup : D \rightarrow R$ *is an objective function assigning a numerical quality value to a solution*

A partial assignment is a set of tuple pairs, each tuple consisting of an instantiated variable and the value that is assigned to it in the current search node. A full assignment is one that containing all n *variables.*

A partial solution is a consistent partial assignment, i.e., a partial assignment in which no constraints are violated. A solution to a CSP *is a full assignment such that no constraint is violated.*

The model type is defined by the value of the objective function: when this value is infinite, the model is called a CSP *model, otherwise, the model becomes a* COP *one.*

2.2 Problem Examples

2.2.1 Classical Problems

Random Problem. A binary random problem is a set of variables, each variable X_i must take a value from its domaine D_i, where respecting all the constraints in $C\{C_1, C_2, C_3, ...C_m\}$.

Let P be a binary problem $X1 = X2$, $X2 > X3$, $X4 < X2 + 3$ with D= d1, d2, d3, d4=0..6 this problem can be seen like a graph in Fig.2.1.

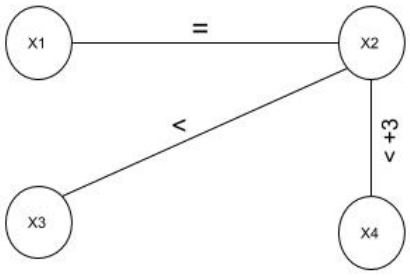

Figure 2.1 – Binary CSP example

Graph Coloring Problem. This problem can represent easily many real problems such as timetabling and channel allocation problems in mobile communication systems, in which adjoining cells can not use the same channels to avoid interference. Let consider the map in Fig.2.2(a), this problem aims to assign to each region a color where two connected areas will have different colors.

This problem can be modeled as CSP as follow :

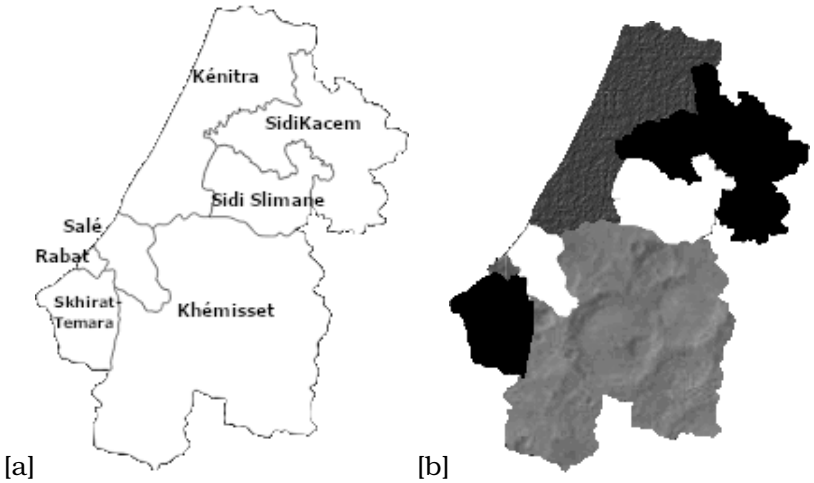

Figure 2.2 – Map of Rabat Salé Kenitra (RSK) area

— X is a set of n regions {Rabat, Salé, Kenitra, Skhirat-Temara, Sidi Slimane, Sidi Kacem, Khémisset}.

— D is a set of domains {4 available colors}

— C is a set of m constraints $\{\forall a, b \in X \text{ and } a, b \text{ are connected} \Rightarrow color(a) \neq color(b)\}$

A solution of the 4-colors map can be like represented in Fig.2.2(b).

Sudoku Puzzle Game. The Sudoku puzzle [Al-Betar et al., 2017] is a popular game across the globe that is a fixed part of most newspapers and magazines. The Sudoku puzzle is represented on a board of size 9X9. That board is subdivided into sub-grids (or blocks) of size 3×3 as shown in Fig.2.3. The cells in the board are either pre-fulfilled by fixed digits that cannot be changed or empty to be fulfilled.

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

Figure 2.3 – Example of initial board of sudoku

The solution of the Sudoku puzzle is reached when the empty cells are fulfilled by suitable numbers where the complete solution must satisfy the following constraints:

— Each row must contain digits from 1 to 9 exactly once.

— Each column must contain digits from 1 to 9 exactly once.

— Each 3 x 3 grid must contain digits 1 to 9 exactly once.

Indeed, the Sudoku puzzle can be classified in terms of difficulty into three categories: easy, medium, and hard. This is due to the number of given digits in the initial board. As the number of given digits increases, the difficulty of the Sudoku puzzle is decreased.

The sudoku puzzle can be modeled as follows.

— **Variables:** $X=\{x_{11}, x_{12}, ..., x_{99}\}$

— **Domain:** $D=\{D_{ij}=\{1..9\} | \forall i,j \in \{1..9\}\}$

— **Constraints:**

  — $\forall i \in \{1..9\} \text{AllDifferent}(x_{ij} | j \in \{1..9\})$;

  — $\forall j \in \{1..9\} \text{AllDifferent}(x_{ij} | j \in \{1..9\})$;

  — $\forall i,j \in \{1..9\} \text{AllDifferent}(x_{(3i+k)(3j+m)} | k,m \in \{1..3\})$

Meeting Scheduling Problem (MSP). The meeting scheduling problem (MSP)[Meisels and Lavee, 2004] is a decision-making process that consists of organizing meetings between participants, determining places and times, while respecting their different calendars. There are many versions of MSP in the literature that differ from one to another by taking into account some parameters like meeting duration [Sen and Durfee, 1998], participant' preferences [Rigi and Khoshalhan, 2013], etc. In this problem, a set of participants are looking for a possible meeting that satisfies the private constraints of each of them, while taking into account the time constraints to arrive between different meetings of the same participant. Most formally, general Meeting Scheduling Problems are characterized by a set of n attendees/participants, a set of m meetings each m_i takes d_i units of time, each meeting m_i will take place in a specific location l_i. Because each meeting takes place in a different location from one to another, there is a constraint called arrival-time ensuring that each participant who is involved in two separated meetings m_i and m_j can move from l_i to l_j. Arrial-time constraint between m_i and m_j can be defined as follows :

$$|time(m_i) - time(m_j)| - d_i > travel(l_i, l_j) \tag{2.1}$$

such as $time(m_i)$ is the time of the meeting m_i and $travel(l_i, l_j)$ is the duration of time needed to travel from location l_i to location l_j.

MSP can be modeled as CSP as follows :

— X = {m_1, m_2, ..., m_i} is a set of variables/meetings

— D={weekly working time-slots} it can be extracted from calender of participants $D(m_i) = \bigcap_{A_j \in \ participants(m_i)} Calender(A_j)$

— C=Arrival-times Constraints for every pair of meetings (m_i, m_j) there is a arrival-time constraint if there exists one participant involved in both meetings.

A simple MSP is illustrated in Fig.6.1, that show all the meetings that we have for all participants i.e: the participants P_1 and P_2 will attend to meeting m_1 which take place in l_1. then P_1, P_2 need 1 slot time to travel to l_2 to meet P_3 (m_2). After that P_1 must travel along 4 time-slots to l_3 where m_3 will take place, etc. The example shown in Fig.6.1 can be modeled as CSP :

— X = {m_1,m_2,m_3,m_4}

— D={D(m_1),D(m_2),D(m_3),D(m_4)}

  — D(m_1)={calender(P_1) $\cap$calender(P_2)}

  — D(m_1)={calender(P_1) $\cap$calender(P_2)$\cap$calender(P_3)}

  — D(m_1)={calender(P_1) $\cap$calender(P_4)}

  — D(m_1)={calender(P_2) $\cap$calender(P_4)$\cap$calender(P_5)}

— Arrival-time constraints C={ c_{12}, c_{22},c_{13},c_{14},c_{44} }.

Meetings	Locations	Participants
m_1	l_1	P_1,P_2
m_2	l_2	P_1,P_2,P_3
m_3	l_3	P_1,P_4
m_4	l_1	P_2,P_4,P_5

Locations	l_1	l_2	l_3
l_1	*	1	2
l_2	1	*	4
l_3	2	4	*

Figure 2.4 – Example of Meeting Scheduling Problem

2.2.2 Medical Internships Problem

The medical internships Problem is an NP resources allocation problem [Vedantham and Iyengar, 1998] where many students must be allocated to various hospital services according to some pedagogical and logistical constraints. There will be a chapter dedicated to this problem in which we describe and detailed this problem, then we propose a design based on CSP formalism to deal with it, and we experiment our model to conclude that there are some limitations on the proposed model which we adjust it by extending our model to Constraints Optimization Problem COP model.

2.3 Strategies and Algorithms

A separation between model and solver is one of the pilar power of the constraints reasoning. Once a problem is modeled, then it can be solved by a generic strategy that can be complete or not. A complete strategy ensures that it finds a solution if there exists because it runs with Depth First Search (DFS)[Awerbuch, 1985] that explores exhaustively the tree search branch by branch until it joints the solution or no solution found. An incomplete strategy is often used to find a solution but not guarantee if the solution will be found if exists due to using some inference or heuristics to avoid some parts of the search tree with no solution. As said, Since CSP is the formalism for modeling and solving NP-Complete problems. CSP' Algorithms used often First-Depth Search(DFS) method that explores the search tree as far as possible along each branch before backtracking when a conflict is detected. Here, some interesting definitions used in follow:

Definition 2.2 (Assignment). *An assignment is a couple*$\{(x_i,v_i)$ *where the variable* $x_i \in X$ *is assigned by the value* $v_i \in D\ (x_i)\}$.

Definition 2.3 (Instantiation). *An Instantiation* I *is a sequence of assignments* $\{(x_0,v_0),(x_1,v_1),...(x_i,v_i)\}$.*if I include all variables* $x_i \in X$, *I is called total Instantiation. Otherwise it's a Partial.*

Definition 2.4 (Partial Consistent Instantiation). *An Instantiation* I *is consistent if and only if all constraints in the problem are satisfied. Otherwise, it is not consistent.*

Definition 2.5 (Solution). *A solution S is a total Instantiation which is consistent.*

2.3.1 Chronological Backtracking

Algorithm 1: Backtracking Algorithm

```
1 begin
2     if I is complete then
3         return Solution(I);
4     return ;
5     x_i ← selectUnsignedVariable(X);
6     foreach v_i ∈ D(x_i) do
7         if Consistent(I ∪ (x_i,v_i)) then
8             Bachtrack(I ∪ (x_i,v_i),X,D,C);
9     return ;
```

The backtracking algorithm (BT) in Algorithm.1 [Bitner and Reingold, 1975] start with an empty instantiation I and tries to extend progressively I to a solution.

It begins by checks if I contains all variables of X (line 2), if it is, the BT Algorithm returns the current instantiation I which is the solution (line 3). Otherwise, the algorithm selects an unassigned variable x_i (line 5) and tries to extends I by searching within x_i's domain a value that will be consistent with the current instantiation I (lines 6-8). Once this value v_i is found, I will be extended with a couple (x_i,v_i) (line 7), and a recursive call is launched to attempt to extend I again (line 8). If no value of x_i is consistent with I, a backtrack is done to change the value of the latest assigned variable in I (line 9).

The Backtracking Algorithm is suffering from the repetition of many useless tries to instantiate some variables due which can not be consistent with a current I, due to some precedent instantiations that are could never belonging to the solution. These failure tries cost many time and effort before backtracking to the cause of the inconsistency. Many techniques have been proposed to avoid those failure tries and improve the efficiency of the Backtrack Algorithm. These techniques can be classified under either Look-back methods or look-ahead methods.

2.3.2 Look back Methods

Many algorithms implement the Look-Back concept which consists of doing an intelligent backtrack to the reason of conflict or inconsistency against backtracking to the precedent variable. This change makes backtracking algorithm more sufficient by avoiding some subset of the search tree with no solution. It can be also seen as learning techniques from failure situations. One of the most popular algorithms is the Conflict-directed Back jumping (CBJ) Algorithm.2 [Prosser, 1993].

Conflict-directed BackJumping

Once a backtrack conflict is detected i.e: the couple (x_i,v_i) can't extend the current instantiation. We save, in some list called $conflict\_list_i$, the variable x_j involved with x_i in the violated constraint that causes the conflict, then we forward to next value of x_i, etc. When a dead-end situation occurs (any of x_i values are inconsistent with the current instantiation), we Backjump to the last variable instantiated in $conflict\_list[i]$ because if we don't change values for at least one of theses variables responsible of conflict we can't extend the current instantiation.

Algorithm 2: Conflict-directed Backjumping

```
1  Procedure CBJ(I);
2  if I is complete then
3  |  return Solution(I);
4  else
5  |  x_i ← selectUnasignedVariable(X);
6  |  CL ←∅;
7  |  foreach v_i ∈ D(x_i) do
8  |  |  CL[i] ← CBJ_Conflict(I ∪(x_i,v_i));
9  |  |  if CL[i] is null then
10 |  |  |  CL[i] ← CBJ(I ∪(x_i,v_i));
11 |  |  if x_i ∉ CL[i] then
12 |  |  |  unassigne x_i;
13 |  |  |  return CL[i]
14 |  |  CL ← CL ∪ CL[i];
15 |  return CL \ x_i;
16 Function CBJ_Conflict(I)
17 |  if I is consistent then
18 |  |  return null;
19 |  else
20 |  |  remove v_i from D(x_i);
21 |  |  let c be a constraint violated by I;
22 |  |  return scope(c);
```

The procedure CBJ is shown in the Algorithm.2 CBJ tries to extend an initial empty Instantiation I, when a conflict has detected a function CBJ_Conflict(I) returns a CL list of variables responsible for the current conflict (lines 15-20). When a dead-end is occurring CBJ jumps to the last instantiated variable y in the union of conflict Lists, and all the variables below y will be unassigned and their domains will be reinitialized (lines 11-13).

Dynamic Backtracking (DBT)

When dead-end occurs, CBJ search looks up into assigned variables the culprit one i.e responsible for the dead-end, and it backjumps to this variable and removes all the assigned ones after the variable responsible for the dead-end. This removal wastes all the effort done to assigned variables (assigning variables and checking the constraints). To deal with this problem, DBT aims to preserve this effort done between the culprit variable and the current one when backjumping process, by using nogoods [Ginsberg and McAllester, 1994a] [Ginsberg, 1993].

Definition 2.6 (nogood). *A nogood ng is a clause of the form*

$$\neg\{(x_1 = v_1)\wedge\ldots\wedge(x_k = v_k)\wedge(x = v)\}. \tag{2.2}$$

The nogood ng is logically equivalent to

$$(x_1 = v_1)\wedge\cdots\wedge(x_k = v_k)\longrightarrow(x \neq v). \tag{2.3}$$

Meaning of ng A nogood ng $\neg(x_1 = v_1)\wedge\ldots\wedge(x_k = v_k)\wedge(x = v)$ means that the assignment $\{(x_1 = v_1),\ldots,(x_k = v_k),(x = v)\}$ is inconsistent.

There are many different ways of representing a given nogood as an implication.

Definition 2.7 (A directed nogood). *When a nogood is written as an implication, its left-hand side (lhs) and right-hand side (rhs) are defined from the position of $\longrightarrow$. and means that lhs causes rejecting of rhs.*

In follows, all used nogoods are directed ones.

Algorithm 3: Dynamic Backtracking

```
1  Procedure DBT(I);
2  if I is complete then
3  |  return Solution(I);
4  else
5  |  x_i ← selectUnsignedVariable(X);
6  |  foreach v_i ∈ D(x_i) do
7  |  |  nogood←DBT_evaluate(I ∪(x_i,v_i));
8  |  |  if nogood is null then
9  |  |  |  nogood ← DBT(I ∪(x_i,v_i));
10 |  |  if (x_i = v_i) = rhs(nogood) then
11 |  |  |  store nogood as a justification of rejection of v ;
12 |  |  else
13 |  |  |  foreach v ∈ D_i do
14 |  |  |  |  if rhs(nogood)∈ N_i^v then
15 |  |  |  |  |  N_i^v ← null;
16 |  |  |  return nogood;
17 |  return ∪_v∈ D_i N_i^v;
18 Function DBT_evaluate(I)
19 |  if I is consistent then
20 |  |  return null;
21 |  else
22 |  |  let c be a constraint violated by I;
23 |  |  return solve_nogood(I,c);
```

As shown in the Algorithm.3, DBT tries to extend a partial instantiation I (lines 5-6) by assigning x_i. For each inconsistent value DBT removes this value v_i from its domain and store a generated nogood $I \rightarrow x_i \neq v_i$ as a justification of this removal. A DBT_evaluate(I) (lines 18-20) checks the consistency of a given I, if it is, it returns a null nogood and DBT search a good value for the next variable (lines 8-9). Otherwise, we solve nogood by calling solve_nogood function (lines 22-23) to generate a new nogood from all nogoods containing the current variable x_i as rhs side.

The lhs result in nogood is a conjunction of all the rhs sides of nogoods containing x_i in rhs side except a last instantiate one within these variables, let be x_j which will be at the right side of result nogood.

Then, all the nogoods containing x_j in their lhs will be removed, so rejected values will be restored to D_j (lines 14-15) then x_j change its order to be the current variable. Next DBT loops in the search process. Hence DBT doesn't change any variables except the culprit one and has preserved all effort done before.

2.3.3 Consistencies propagation Methods

Contrary to look-back methods which preserve the tree space as it is and try to detect the backtrack node to avoid inconsistencies, look ahead methods prunes during search some branches of search tree containing no solution then reduce search space i.e. space and time complexities especially for the large and difficult problems [Sabin and Freuder, 1994]. This process is done by propagating information from assigned variables to unassigned ones yet every time a new assignment is added to a current partial solution.

Forward checking

The forward checking FC algorithm [Haralick and Elliott, 1980a] is the simplest procedure that implements the idea of propagation the information from assigned variables to unassigned variables

yet. The pseudo-code of the FC procedure is shown in the Algorithm.4, FC begins by assigning a first variable $x_i = v_i$ (line 6), then see forward to filter all the variables linked to x_i by constraints to remove from their domain all values that can not be consistent with $x_i = v_i$ (lines 7,13-15). If no variable' domains are empty, FC extends the current instantiation I (line 8). Otherwise, FC backtracks to the last variable to reassign it and restores all values removed by the Filter function (lines 10-11,16-17).

Algorithm 4: Forward Checking Algorithm

```
1  Procedure FC(I);
2  if I is complete then
3  |  return Solution(I);
4  x_i ← selectUnsignedVariable(X);
5  foreach v_i ∈ D(x_i) do
6  |  x_i←v_i;
7  |  if Filter(I,(x_i = v_i)) then
8  |  |  FC(I ∪ (x_i, v_i));
9  |  else
10 |  |  foreach x_j ∉ var(I), ∃c_ij ∉ C do
11 |  |  |  restore D(x_j);
12 Function Filter(I,(x_i = v_i))
13 |  foreach x_j ∉ var(I), ∃c_ij ∉ C do
14 |  |  foreach v_j ∈ D(x_j), (v_i, v_j ∉ c_i,j) do
15 |  |  |  remove v_j from D(x_j);
16 |  |  if (D(x_j) is empty) then
17 |  |  |  return False;
18 |  return True;
```

Hence, Forward Checking removes domain values of unassigned variables yet based on assigned ones to reduce domains and ensure consistency before extending the current instantiation. However, this principle causes removing some values useful for the consistencies of future variables. So checking the consistency not only for the variable directly linked with the current one but going deeply forward and for each variable will reduce further the search space and avoid more conflicts. This is the principle of the constraints propagation or maintaining arc consistency approach(MAC).

Arc-Consistency (AC)

Constraint propagation techniques are the most popular and the most used local consistency way in practice (CSP solvers). They aim to reduce space search by filtering before or during the search process to eliminate unuseful values from variables domains. Various levels of constraint propagation have been proposed: node [Mackworth, 1977], arc [Mackworth, 1977], path[Montanari, 1974] which implemented in a large family of algorithms called AC-x Algorithms witch we are going to develop some of them in the fellow.

Definition 2.8 (arc support)**.** *A value $v_i \in D(x_i)$ is consistent with a constraint c_{ij} in $D(x_j)$ iff there exists a value $v_j \in D(x_j) \mid (v_i, v_j)$ satisfies c_{ij}. value v_j is called a support for v_i in $D(x_j)$.*

Definition 2.9 (arc consistence)**.** *An arc $(x_i, x_j) \in c_{ij}$ is arc consistent iff $\forall\ v_i \in D(x_i)$, v_i has a support in $D(x_j)$ and $\forall\ v_j \in D(x_j)$, v_j has a support in $D(x_i)$. A constraint network is arc consistent iff all its arcs are arc consistent.*

Exemple. Let consider an CSP instance Fig.2.5 containing just two variables $x_1(0,1,2)$, $x_2(1,2,4)$ and one constraint to check c $\{x_1 > x_2\}$. The Fig.2.5(a) shows that the value 0 and 1 of x_1 have no supports in x_2 regarding the constraint $x_1 > x_2$, and respectively the values 2 and 3 of x_2 don't have supports in x_1. Thus, after removing the inconsistent values of both variables, the arc (x_1, x_2) will be arc-consistent, then the CSP instance is arc-consistent, and obviously the solution is (2,1).

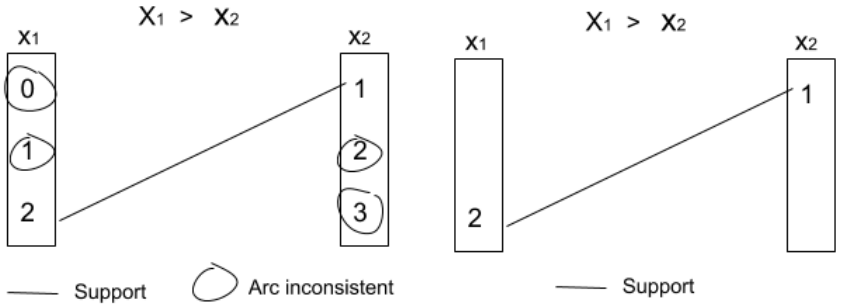

Figure 2.5 – Example of Arc-inconsistent values

AC algorithms

AC-3 algorithm. As said before, there is a succession of many algorithms which have been proposed that implement the principle of Arc-Consistency, each one improves his predecessor: AC-3[Mackworth, 1977], AC-4[Mohr and Henderson, 1986], AC-5[Van Hentenryck et al., 1992], AC-6[Régin, 1994], AC-7[Bessière et al., 1999], AC-2001[Bessière and Régin, 2001a].

Algorithm 5: The AC-3 Algorithm

```
1  Procedure AC3(X)
2  Q ← (x_i | c ∈ C, x_i ∈ X(c));
3  while Q ≠ ∅ do
4  |  select and remove(x_i,c) from Q;
5  |  if revise3(x_i,c) then
6  |  |  if D(x_i)=∅ then
7  |  |  |  return false;
8  |  |  else
9  |  |  |  Q ← Q ∪ {(x_j,c') | c' ∈ C ∧ c' ≠ c ∧ x_i,x_j ∈ X(c') ∧ j ≠ i};
10 return true;
11 Function revise3(x_i,x_j)
12 |  change ← false;
13 |  foreach v_i ∈ D(x_i) do
14 |  |  if ∄v_j ∈ D(x_j) | (v_i,v_j) satisfies c_ij then
15 |  |  |  remove v_i from D(x_i);
16 |  |  |  change ← true;
17 |  return change;
```

AC-3 proposed by Mackworth [Mackworth, 1977] is considered the simplest approach in this family, the pseudo-code in Algorithm.5 shows that the main process of AC3 is reducing the variables' domains to make a CSP arc-consistent. For that, it uses a list Q containing initially all possible linked pairs (x_i,x_j) (line 1), For each pair (x_i,x_j) (lines 2-3), AC3 calls revise3 function (lines 5, 11-16) which is looking for supports for every v_i in x_j and returns true when such a support is found, otherwise the values v_j is removed from x_j' domain. One special case here is when a value v_i from $D(x_i)$ is removed, it may be the unique support for some other variable x_k wrt c_{ik} or c' which will be arc-inconsistent. For this AC3 re-takes for filtering all constraints $c_{ik} \in C$ with $k \neq j$, and re-puts all pairs (x_k,x_i) in Q (line 9). When Q became empty the algorithm return true (line 10) which means that all the inconsistent values fr all variables have been removed, and AC3 guarantees that all the constraints are arc-consistent.

AC3 is linear space complexity refereed to the unique data structure Q, but the time complexity is important $O(ed^3)$ i.e.,: the complexity of revise function is $O(d^2)$. where e is the number of constraints

and d is the size of the largest domain.

AC-2001 algorithm. AC-2001[Bessière and Régin, 2001a] has the same framework of AC-3 ,but differs from it by revise function described in Algorithm.6 and its initialization phase which needs to initialize the pointers Last$[x_i,v_i,x_j]$. When (x_i,x_j) will be revised, AC-2001 is looking for supports for a value (x_i,v_i), it doesn't begin from the first value $v_j \in D(x_j)$ like AC-3, but it resumes from the last found support for such value using a pointer Last$[x_i,v_i,x_j]$ contains v_j (lines 4-5).

This pointer Last is updated when a support v_j is found for (x_i,v_i), AC-2001 at this time assigns v_j to Last$[x_i,v_i,x_j]$ (line 6).

AC-2001 achieves arc consistency on binary networks in $O(ed^2)$ time and $O(ed)$ space.

Algorithm 6: The revise function of AC-2001 Algorithm

```
1  Function revise2001(x_i, x_j)
2      change ← false;
3      foreach v_i ∈ D(x_i) s.t. Last[x_i, v_i, x_j] do
4          v_j ← smallest value in D(x_j) > Last[x_i, v_i, x_j] s.t. (v_i, v_j) ∈ c_ij;
5          if ∃v_j then
6              Last[x_i, v_i, x_j] ←v_j;
7          else
8              remove v_i from D(x_i);
9              change ← true;
10     return change;
```

Maintaining Arc-Consistency MAC

All filtering techniques seen from previous sections can be used in pre-treatment to prepare and make the CSP arc consistent i.e. reducing variables' domains so time and space complexity of search process.

Or categorically used these techniques during the searching process. Sabin and Freuder [Sabin and Freuder, 1997] have been proposed the Maintaining Arc-Consistency (MAC) algorithm, which establishes and maintained a full arc consistency on the future variables.

Similarly to Forward checking [Haralick and Elliott, 1980a] which combines backtrack process and filtering domain of the future/neighbors variables, the MAC algorithm Boosts this same idea much more deeply for propagating the arc consistency through all the future variables when the assignment has extended by adding a tuple (x_i,v_i).

technically, when (x_i,v_i) extends a current assignment, MAC reduce temporarily the domain of x_i to a chosen value x_i and filtering step is launched to reduce the future variables' domains to looking for supports wrt last assignment, if this will be done (arc-consistency), another filtering step is to begin to propagate the last filtering phase (2 arc-consistency) and continues to propagate much more deeply until having full arc-consistency CSP.

2.3.4 Heuristics search Methods

we have seen previously that solving a CSP problem is an NP-complete problem and every strategy tries a maximum as possible to avoid exponential complexity or just reduce it. Some early studies have shown that the order of assigning variables is very important to reduce the search complexity by reducing the search tree or detect earlier the unsolvability.[Williams et al., 2003] have proposed to begin with small backdoor variables i.e. a set of variables which one assigned to make the problem easy to solve. Many techniques have been developed to combine classical search methods with ordering heuristics to boost methods' efficiency. The order can be on variables or values and either static or dynamic.

We present in the following some heuristics examples:

Static variable order in which all variables are initially ordered based on the structural information of the problem and keep this same order all along the search process, we distinguish here between lexico where variables are lexicographical ordered, in many cases this order is default one. And deg and ddeg [Dechter and Meiri, 1994] where variables are decreasingly ordered according to their initial and current degree (the number of neighbors of each variable), and width [Freuder, 1985] where variables are ordered to minimize the width of the constraint graph.

The second kind of variable ordering heuristics is the dynamic variable ordering (DVOs) which establishes order between variables based on the current state of the search, it means that variables ordering changes continuously along with the search. dom is the most known dynamic heuristic [Haralick and Elliott, 1980b] where variables are decreasingly ordered based on their current domains sizes. this DVO suggests that the variable to be assigned next should be the one that is most likely to lead to a dead-end. The heuristic aims at proving that the search is in a sub-tree with no feasible solutions as soon as possible. Bessiere C and Smith B.M have proposed dom/deg [Bessiere and Régin, 1996] and dom/ddeg [Smith and Grant, 1997] which combine dom and deg in order to improve the search performance.

Later in 2004, dom/wdeg [Boussemart et al., 2004a] Conflict-driven variable ordering heuristics has been designed to learn from previous failures to manage the choice on future variables. It selects the first variable having the minimal ratio between domain size and weighted degree. A weight degree is associated with each constraint. When a dead-end occurs, the weight of constraints is increased. For a given variable, its weighted degree (wdeg) is the sum of weights of constraints involving this variable and not yet instantiated variables.

It has been shown that the Maintaining Arc Consistency (MAC) dom/deg and dom/wdeg can reduce or remove the need for backjumping on some problems [Boussemart et al., 2004a].

In addition to presented heuristics, researchers have been proposed [Haralick and Elliott, 1980c] some value ordering heuristics to choose the best value of domain that may participate in the solution when a variable is chosen. In general, the least constrained value is chosen first i.e most likely to succeed. min−conflict heuristics is the most popular one that attempts to minimize the number of constraint violations [Minton et al., 1993a] and maximizes the values available for future variables. Consequently, partial solutions that cannot be extended will be avoided.

We note that it's difficult to find the best absolute heuristic, but it depends on the problems' nature and structure, we will propose later profound degree heuristics which improve significantly the performance of the degree heuristic.

However, all of these heuristics and more in literature [Dechter and Pearl, 1988], [Frost et al., 1995],[Meisels et al., 1997],[Vernooy and Havens, 1999],[Kask et al., 2004] are costly in processing and time.

2.4 Conclusion

This chapter described some real-life problems and showed that they can be smoothly designed like a Constraint Problem.

Also, many algorithms based on the chronological backtracking algorithm have been developed for these problems.

The performance of these algorithms can be improved by combining intelligent backtracks, propagation of information as constraints, and finally using variables/values ordering heuristics.

CHAPTER

3 Medical Internships Planning

Hospital internships are the most essential component of any medical training to acquire and develop different clinical skills. among the big management issues related to this point is the allocation of internship students to different services undergoing several logistical and pedagogical constraints during the hospital university calendar. In this chapter, we show how we can use Constraint Programming design to model and solve this problem through the Constraint Satisfaction Problems (CSP). Then, we propose an extended model (Constraint Optimization model) to cover the limitations of the CSP model and find the optimal scheduling taking into account students' preferences. Finally, we show that using several What-If scenarios can help to suggest and recommend to decision-makers some adjustments to the problem inner rules to get the optimal solution.

Contents

3.1 Introduction

In the majority of the countries, the students of medical schools must examine, diagnose and help the doctors to deal with patients who have been referred to the hospitals. Effective management of all existing resources for students, doctors, and hospital services is necessary, while health is one of the most public needs.

Statistically, In 2016, Morocco, for example, had 36881 doctors (all specialties), 21064 nurses and 13083 (57%) of students, and 21064 (doctors and nurses) will be retired progressively in the horizon of 2025. As Morocco is a Lower Middle Income GDP country (world bank 2016)[Bank, 2018] the WHO [WHO, 2018] was established one doctor per 1000 people as necessary to deliver essential material and child health services, Morocco is at 0.62 doctors (physician) per 1000 population.

To deal with this situation, since 2008, Morocco has launched a governmental program to prepare incrementally 3300 (+30%) [Ministry, 2010] medical doctors per year in the horizon of 2025. This situation is the reason why all medical schools must optimize the occupancy of hospital services (practical and clinical courses) and adopt good management of the current resources to improve patients satisfaction.

In the present chapter, we present the component of the hospital training in the curriculum of the medical students. Next, we present our two models of this problem, the satisfaction model and the optimization one. Then we show how some what-if scenarios can help the decision-makers to adjust some constraints of the problem to get the best solution. And before concluding, we present a multi-objective optimization model for the more customized solution of this problem. Finally, we conclude and propose some perspectives.

3.2 Background

3.2.1 Related work

In past years, many works have been done to approach IA formalism to Health sector problems, especially the scheduling problems, [Latorre-Núñez et al., 2016], [Vali-Siar et al., 2018] propose a model based on constraint programming to find optimal scheduling of the operation rooms (OR) with all required staff, equipment, and beds. [Fikar and Hirsch, 2017] shows some works related to Home Health Care HHC that is expected to increase substantially. The Objective of these works is to find an optimal combination of nurses who are scheduled and routed to perform various services at patients' homes parallelly to decrease costs and guarantee service quality.

[Costa Filho et al., 2012], the authors have used CSP formalism to model and find a human resource scheduling in cooperative health service, then they proposed some heuristics based on backtracking approach which is a very classical approach to solve a CSP model.

The recent reviews presented in [Spyropoulos, 2000] [Marynissen, 2016] gather works between 1992 and 2016 on integrated hospital scheduling problems in which various techniques like genetic algorithms, multi-agent Pareto, Markov Decision Processes, particle swarm optimization, Approximate Dynamic Programming, case-based reasoning, a neural network used to find optimal resources scheduling that allows to each patient to be assigned to required path over a subset of the hospital resources, and each step of the path should be carefully scheduled. This scheduling must guarantee avoiding long waiting lists in hospitals, reducing access times for cure, taking into account emergency patients and shared resources.

All the previous works are focused on the scheduling of the patients and appointment and medical staff to find a good path for every patient especially in the high service's activity, and to plan efficiently the health resources. Therefore any of them takes the problem of scheduling internships even if planning efficiently this part of medical human resources surely improve significantly the staff availability thus, reducing patient access to the medical cure, improve physicians satisfaction and upgrade the health care services' quality.

In the CSP formalism, the solutions are assignments to a set of well-defined variables, in the problem model, of values taken in sets defined respectively for each of these variables while respecting a set of constraints linking these variables. As formulating a good model is the first step and one of

the most important phases to solve any problem, we focus in this chapter, on how to model efficiently the problem of allocating internship students to different hospital services while satisfying all the logistical and pedagogical constraints [Benamrane et al., 2014]. And then, we study and analyze its solvability, and how to use the Constraint Optimization Problems (COP) tools to optimize the solution quality when many possible solutions exist.

3.2.2 Constraint Optimization problem

A model is an abstraction of a problem, which must represent all the problem details and respects the modeler language. Surely, the quality of the solution depends on how the problem is designed.

3.2.3 Internships in Medical School

Hospital internships are an essential part of the curriculum of the medical student. Each student must pass this component to acquire some clinical skills, and to complete the theoretical knowledge taken in the courses. Hence, the medical school uses a classical software with the Object-Oriented Programming formalism (OOP) [Wegner, 1990], which is developed with the DotNet tool [Lauer, 2002], but the solution found is always incomplete (many students still not assigned and the distribution over services is not balanced and takes a lot of time and hardware resources (at least 3 powerful machines for three weeks). This situation forces the manager to complete manually this distribution, so violates some constraints. In below, some interesting definitions that will be useful later:

Definition 3.1 (Internship period p)**.** *The internship period or just period* p *is a duration of time. An academic year has a fixed number of periods.*

Definition 3.2 (Complementary service)**.** *A complementary service is a service that lasts one period.*

Cardiology, nephrology, and dermatology services are complementary services.

Definition 3.3 (Fundamental service)**.** *Fundamental service is a service that occupies two periods. In other words, it is a couple* $< FS_1, FS_2 >$*, where each component* FS_i *lasts a single period, such as* $i \in \{1,2\}$*,.*

The pediatric service, for example, is a fundamental one. It is a pair of $pediatric_1$ and $pediatric_2$, and each component takes a single period.

Definition 3.4 (Obligatory service)**.** *Obligatory service is a service that every student must do.*

Definition 3.5 (Optional service)**.** *Optional service is any service that is not obligatory.*

Definition 3.6 (Service category)**.** *All services are categorized, and each one belongs to one category.*

Definition 3.7 (Emergency service)**.** *There are some services still open and accessible 24 hours a day, seven days a week (24/7). These are called emergency services.*

We note that a service can have many options from above, and both obligatory and complementary services are not necessarily fundamental services. The problem constraints can be divided into two categories, the first one is the local constraints, which manage the relationship of each student with its periods, and the second one is the global constraints, which define the relationship between students and services in which they are assigned.

— Local constraints (regarding student):
  — **Completeness constraint:** Each student must have a fixed number of courses/internships along the year (number of periods); (1)
  — **Uniqueness constraint:** Each student should not have the same service twice; (2)
  — **Obligation constraint:** Each student must have a fixed number of obligatory services; (3)

— **Fundamental constraint:** Each student having a fundamental service s_i (part of the pair $< s_i, s_j >$) in a given period p such as $p \in [1, |Periods|-1]$, he must be allocated in the following period $p+1$ to the service s_j; (4)

— **Last period constraint:** Each student should not have the first part of a fundamental service in the latest course; (5)

— **Category constraint:** Each student should not have more than m services belong to the same category; (6)

— **24/7 constraint:** Each student should not have more than n services from emergency services list; (7)

— **Global constraints (regarding service):**

— **Capacity constraint:** Each service is characterized by its capacity that can not be exceeded; (8)

3.3 Modeling the problem as a CSP

3.3.1 Problem formulation

The goal of this section is to model the previously described problem as CSP. To do this, we translate all the problem components to a set < variables, domains, constraints>. The modeling can be done in many ways, we propose a model that we consider the clearest and the simplest one.

Variables :

A CSP variable X_i^p is a period/slot p for a given student i, such as: $X_i^p = s_j$ means that the student i is assigned to service s_j for its period p. We choose variables as above since the solution must be a complete assignment of all periods for all students.

— **Completeness constraint:**

$$\forall\, i \in students : \sum_{p=1}^{|Periods|} assign(X_i^p) = |Periods| \tag{3.1}$$

Domains :

Domains : Let $D = \{s_1, s_2, ..., s_n\}$, the set of the n services in University Hospital (UH) be the domain of all CSP variables X_i^p.

$X_i^p = s_j$ means that the student i is assigned, for his period p, to the service s_j.

Constraints:

As mentioned above, we have two types of constraints: the first type presents the constraints related to the students, that identify the relationship of each student with its periods, and the second part describes, the relationship between students and services in which they are assigned.

— **Local constraints (regarding student):**

— **Uniqueness constraint:** Each student should not have the same service twice:

$$\forall\, i \in students, \forall\, k, l \in [1, |Periods|] \text{ such as } k \neq l : X_i^k \neq X_i^l \tag{3.2}$$

— **Obligation constraint:** Each student must have a fixed number of obligatory services.
Let consider OS, the list of obligatory services, and $m \in [1, |OS|]$ the fixed number of obligatory services for all students:

$$\forall\, i \in students : \quad \sum_{p=1}^{|Periods|} isIn(X_i^p, OS) = m \tag{3.3}$$

$isIn(e, E)$ is a function returning 1 when e belongs to E.

— **Fundamental constraint:** Each student having a fundamental service s_i (part of the pair $< s_i, s_j >$) in a given period p such as $p \in [1, |Periods|-1]$, he must be allocated in the following period $p+1$ to the service s_j

$$\begin{aligned} &\forall\, FS = < s_1, s_2 > \text{ a fundamental service,} \\ &\forall p \in [1, |Periods|-1] \text{ and } \forall\, i \in students: \\ &X_i^p = s_1 \Leftrightarrow X_i^{p+1} = s_2 \end{aligned} \tag{3.4}$$

— **Last period constraint:** Each student should not have the first part of a fundamental service in the latest course

$$\begin{aligned} &\forall\, i \in students: \\ &X_i^{last\_period} \neq s_1 \end{aligned} \tag{3.5}$$

— **Category constraint:** Each student should not have more than m services belong to the same category
Let m be the maximum number of the services belonging to the same category, and $isIn(X_i^p, cat)$ is a function that return 1 if X_i^p is in the category cat and 0 otherwise.

$$\forall\, i \in students, \forall\, cat \in Categories: \quad \sum_{p=1}^{|Periods|} isIn(X_i^p, cat) \leq m \tag{3.6}$$

— **24/7 constraint:** Each student should not have more than n services from the emergency services list.
Let n be the maximum number of emergency services allowed for every student and ES a set of emergency services in the Hospital.

$$\forall\, i \in students: \quad \sum_{p=1}^{|Periods|} (isIn(X_i^p, ES)) \leq n \tag{3.7}$$

Ignoring constraints concerning non-obligatory services does not negate their existence. In fact, due to their existence in variables domains and the high number of variables, a solution can not be found without using these services.

— **Global constraints (regarding service):**

— **Capacity constraint:** Each service had a capacity that can not be exceeded

$$\forall\, s \in services, \forall\, p \in [1, |Periods|]: \quad \sum_{i=1}^{|students|} (X_i^p = s) \leq capacity(s) \tag{3.8}$$

such as $capacity(s)$ is the capacity of the service s.

3.3.2 Experimental Results

To experiment our model, we choose the real configuration of medical school in Casablanca :

— **500** students;

— **{1, 2,...,21}** services (A complete list is available in appendix section), we suppose that all the services have the same weight;

— the respective capacities are: **{16, 16, 16, 7, 36, 20, 21, 38, 12, 18, 40, 23, 41, 20, 19, 19, 19, 40, 40, 51, 17}**;

— **{18,19,20,21}** is the set of obligatory services;

— $\{<13,14>,<15,16>\}$ is the set of fundamental services;

— For each student one obligatory service is required;

— All services belonged the same category.

This problem generates more than **4000** variables, and more than **17 000** constraints. Our experiments, with 5 hours as timeout, show that the maximum number of students, which can be allocated is 380, otherwise, to allocate all the students, some constraints are violated.

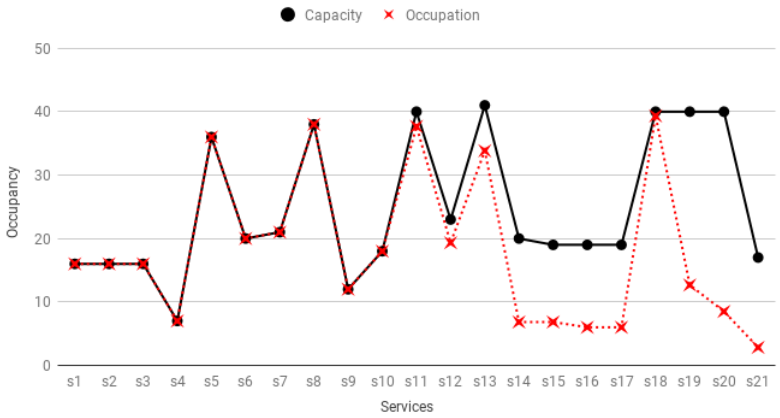

Figure 3.1 – Resolution : occupancy per services

Fig.3.1 presents the services occupancy compared to the capacities of the service. We show that the services 1 to 11 are fulfilled, and some places are still available in other services, especially for the last three ones.

3.3.3 Discussion

The services 1 to 11 are full because they are not constrained except by their capacities (8^{th} Constraint). Although some places are still available, and some students are not allocated yet, we can not assign them there, because these services are very constrained(C_3 and C_4).

The previous section shows that the model CSP is limited to solve this problem, i.e. find schedules for all the students without violating any constraint. Unfortunately, the problem is unsolvable. Thus, we need another model to deal with this situation.

3.4 Modeling the problem as a COP

As above, it is unavoidable to exceed the capacity for some services, if we want to allocate all students. This section aims to distribute optimally the capacities exceeding over services, therefore, we present a new model of our problem based on COP formalism to minimize the services occupancy.

3.4.1 Problem formulation

The variables and domains remain as in the previous CSP model, however, the constraints component have some changes as follows: All the students' constraints (from 1 to 7) remain as the CSP model, however, the global constraint referred to the service's constraint ($C_{3.8}$) must be replaced by a (some) objective function(s).

3.4.2 Mono-objective Optimization

This Optimization version is focused on the quality of the solution in terms of the services occupancy. Thus, for each service i, we denote O_i its occupancy, C_i its capacity and E_i its Exceeding such:

$$E_i = O_i - C_i \tag{3.9}$$

$E_i \leq 0$ means that the capacity of the service i is respected, otherwise, the service is overloaded. This optimization version aims to guarantee that the service capacities are as possible respected, or minimally violated. The constraint (3.8) will be replaced by the following objective function :

$$\begin{gathered} \forall\ i \in services,\ \forall\ p \in Periods: \\ Minimize(Max(E_i^p)) \end{gathered} \tag{3.10}$$

Where E_i^p is the amount of the capacity exceeding related to the service i at the period p. The experiments below take the previous configuration. The results are presented in Fig.3.2

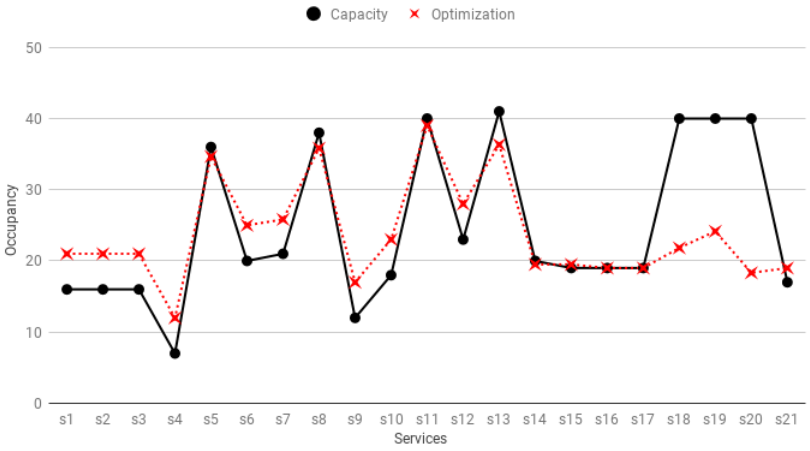

Figure 3.2 – Mono-Objective Optimization: occupancy per service

The results present the occupation of the services with the new model (all the students are allocated). We show that some services are kept under capacities, however, some services capacities are more exceeded than others.

Discussion

The objective of the new model is to minimize as much as possible the capacities violations. We note that the violations are minimal especially for the over-constrained (pedagogical constraints) services (s13 to s17) compared to others. This is due to the importance of these ones, and the activity on them when looking for a solution. These services are involved in the pedagogical (hard) constraints, which are evaluated whenever a maximum is reached, hence impacts consequently the solution quality.

We note also, that some obligatory services (s18 to s21) are still under-loaded because these services are involved in the obligatory constraint(C_3), that holds the number of the obligatory service for each student. We propose in the next section some improvements to find more equilibrium between services in terms of occupancy.

3.4.3 What-If Scenarios

This section is focusing on some suggestions to adjust some inner rules to get the best possible solution. We noted, in the previous section, that the obligatory services have a big impact on other services. Because these services are constrained by the fact that they are obligatory and by their capacities. The Obligatory service means that they are important for future doctors. And the constraint (C_3) forces that every student must have m obligatory services from the set (here $m = 1$, named Optimization 1 scenario). We propose an adjustment of this constraint to allow students to have more than just one obligatory service, we choose to change the parameter m to evaluate two scenarios $m = 1$, $m = 2$.

Experiments and Discussion

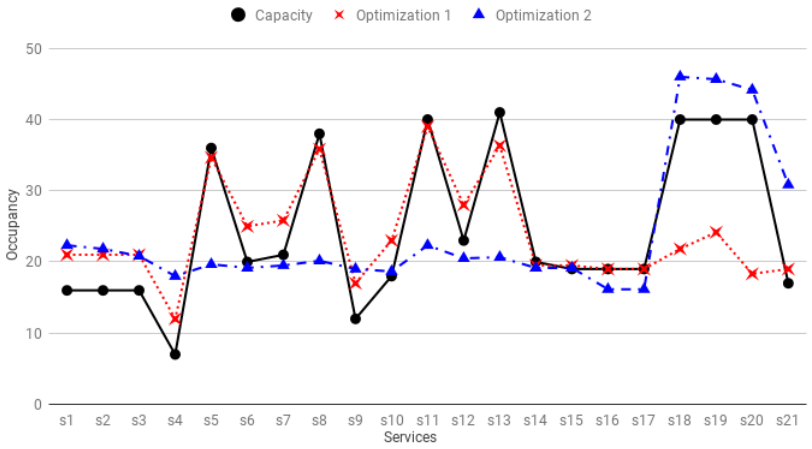

Figure 3.3 – Mono Objective Optimization: occupancy per service

The Fig.3.3 presents the comparison of the occupancy in two scenarios: when $m = 1$ (Optimization 1) and $m = 2$ (Optimization 2). We show that the obligatory services (s17 to s21) are overloaded in the optimization 2 scenario, otherwise many services became largely under-uploaded, which means good conditions for the learners as well as their supervisors. These scenarios present two visions of this solution, and choosing between them depends on the medical school objectives and some other preferences related to the supervisors, medical staff, etc.

3.4.4 Multi-objective Optimization

To improve the previous optimization model focused on the service side, we deal in this section with another optimization model, concerning the students' side by proposing an improved model which handles students' preferences. There are so many situations for which students prefer some services more than others (proximity, background, etc.), that is why we propose, in this section, a multi-objective optimization model based on Pareto front [Huang et al., 2013], which improve the previous model to deal with the students' preferences.

The multi-objective optimization model tries to find the best solution for minimizing the services occupancy while maximizing the students' preferences satisfaction.

We add another objective constraint, that represents the students preferences. Thus, the objective function (3.10) will be overwritten such:

$$\forall\, i \in Services,\ \forall\, p \in\ Periods:$$
$$\begin{cases} Min(Max(E_i^p)) \\ Max(\sum_{p=1}^{|Periods|} \sum_{i=1}^{|Student|} (SAT(X_i^p))) \end{cases} \tag{3.11}$$

such as $SAT(X_i^p)$ is a function that returns 1 if X_i^p is equal to the preferred service and 0 otherwise.

Experiments

We give the students the possibility to express their preferences on two periods from six, and we experiment with the previous configuration with the optimization 1 scenario ($m = 1$ in the previous model).

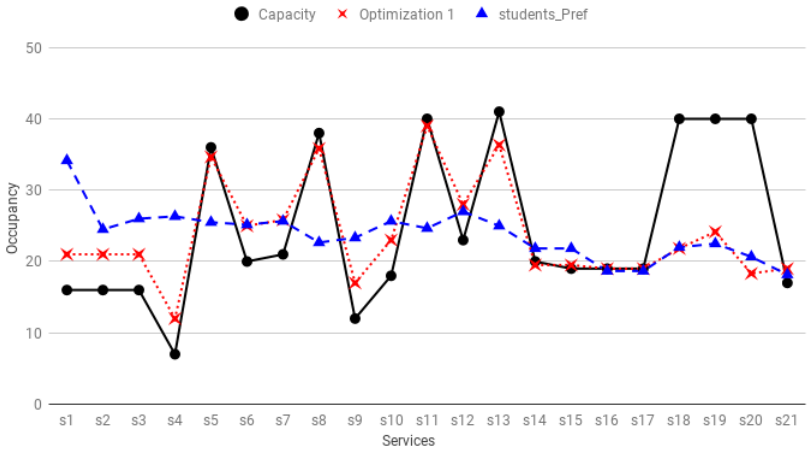

Figure 3.4 – Multi-Objective Optimization: occupancy per services

The results in Fig.3.4 present the services occupancy between the mono-objective model ($m = 1$), and the multi-objective optimization model. We note that 59% (10% of students are fully satisfied, and 49% are partially satisfied) of students preferences are satisfied depending on the preference model chosen, however, we got up some exceeding of capacities, due to the objective which is to optimize simultaneously the services occupation and the students satisfaction.

Discussion

We show that the occupation in the hard services is almost similar in the two models, however, the soft ones are very different in terms of occupation. This is because a solution is determined

Maximum exceeding capacity	0% Satisfied	50% Satisfied	Fully Satisfied
8	259	207	34
9	258	208	34
10	257	209	34
11	256	208	36
12	255	209	36
13	252	212	36
14	243	221	36
15	242	219	39
16	241	220	39
17	240	221	39
18	239	218	43
19	238	219	43
20	237	220	43
21	236	220	44
22	235	221	44
23	234	222	44
24	233	223	44
25	232	224	44
26	231	225	44
27	230	222	48
...	...	...	...

Figure 3.5 – 20mn of execution (students' satisfaction versus services' capacities exceeding)

by the hard services. This is why the solver begins to satisfy the hardest constraints and keep the soft constraints at the end to optimize them. Also, the satisfaction of students is depending on the preference model chosen, if we let for example students choose more than two services, of course, the rate of satisfaction will be decreased.

The Fig.3.5 logs execution of 20mn of our multi-objective COP model and shows how over iterations we can decrease the maximum of a capacity exceeding (by including a hard constraint holds that the upper bound will be greater than the found value) from 27 to just 8 places as a maximum of capacity exceeding, and the number of not satisfied students decreases while the fully or partially satisfied students increase.

We present also in Fig.3.6 the Pareto frontier of all possible solutions found with limit time fixed at 20mn, we show that because the two objectives functions are correlated, we can't improve the first objective without sacrificing the second objective.

3.5 Conclusion and perspectives

In this chapter, we study a real case of allocating internships of the hospital services, for which we modeled this NP problem as CSP. Then trying a complete approach for solving this problem generates the assignments of just a part of problem variables (students), hence the confirmation of the problem's unsolvability is due to the set of constraints that cannot be respected all at once.

Based on the extended COP model, We showed and demonstrated empirically that the concerned problem, although being unsatisfiable, can have very acceptable solutions in so far as it respects the most possible problem constraints. Additionally, we proposed more optimal solutions, which optimize other optimal parameters (the service's preferences), while maintaining a very acceptable solution. And we also gave the Pareto frontier, it is impossible to find the optimal global solution, however, there

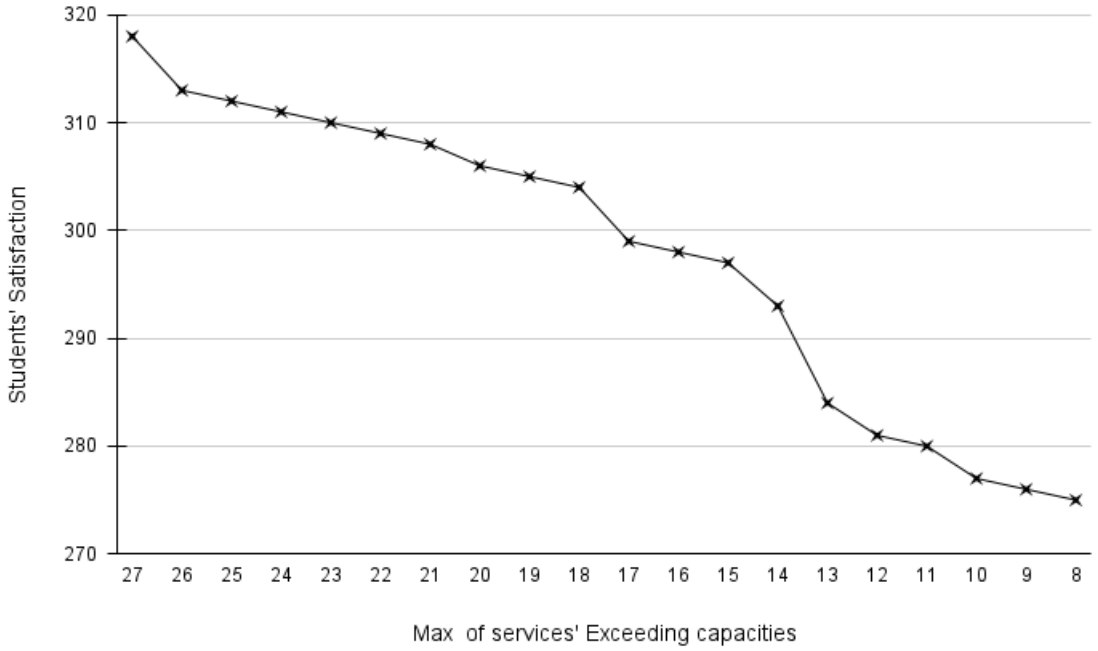

Figure 3.6 – Pareto Front for Students' satisfactions and services' capacities

is some trade-off between the two parameters, with no way to improve the first parameter without losing at the second.

As a perspective, we intend to integrate all the hospital resources (doctors, nurses, rooms, medical equipment, patients, etc.) to produce the optimized planning of human and material resources.

Appendix

MIP Application

We have realized an application [Benamrane and Erraji, 2018] that allows :
(1) to import the problem data;
(2) to choose between the solving mode and optimizing one;
(3) to show the solution in many views students, service, periods
(4) and also to manage and adjust solution following some late changes.

The complete services list

1. Biology 1 (Anatomy-pathology, Immunology)
2. Biology 2 (Biochemistry, Bacteriology)
3: Biology 3 (Hematology, Parasitology)
4: Pharmacology
5: Endocrinology
6: Nephrology
7: Neurology
8: Neurosurgery A6

9: Neurosurgery P11
10: Aile 2
11: Orthopedic Traumatology Pediatric
12: Radiotherapy
13: Surgery, Visceral Surgery
14: Pediatrics 1
15: Pediatrics 2
16: Pediatrics 3
17: Pediatrics 4; Pediatrics 5
18: Central Radiology
19: Emergency Radiology
20: Pediatric Radiology
21: 20 August Radiology

CHAPTER

4

Dynamic Centralized Constraint Programming

Everything around us is dynamic, and supporting changes and perturbations of the context is one of the most important aspects that should have every model that represents a real uses cases. The advantages of Constraint programming are its capacity and flexibility to represent many real dynamic problems. This chapter will present how Constraints programming designs the dynamic problem and present some popular approaches to deal with this kind of problem.

Contents

4.1 Introduction

As shown in previous chapters, a large number of problems can be modeled as Constraint Satisfaction Problem (CSP). thus, Constraints are naturally present in real-life problems.

The concept of CSPs indicates all these problems and consists in looking for the solutions satisfying them. To increase the efficiency of the constraint satisfaction algorithms, many techniques as filtering, improved backtracking, using efficient representations and heuristics have been developed. However, these techniques assume that the complete description of the CSP instance is completely known and fixed in advance. This is a strong limitation when dealing with real situations [Verfaillie and Jussien, 2005], where problems may evolve due to changes in the environment or their execution conditions.

The notion of dynamic CSP (DCSP) [Dechter and Dechter, 1988] has been introduced to represent such situations. A DCSP is an extension of a static CSP. It can be seen as a sequence of CSPs, where each one differs from the previous one by the addition or removal of some constraints. All other possible changes of a CSP (constraints or domains modifications, variables additions, or removals) can be expressed in terms of constraints additions or removals. The present chapter focuses on scenarios where after solving a CSP, some constraints have been added or retired, then, the old solution and reasoning are required to be effectively repaired.

This chapter will introduce a formalism of a Dynamic CSP and present some famous methods of solving it.

4.2 Dynamic Centralized CSP

Many real problems are dynamic, due to the environment or problem scenario changes. These changes are caused by the dynamism of the environment nature, spurious actions, incomplete knowledge, etc. A DCSP is a CSP that is adapted to changes. A problem change can be a restriction, which is an addition of constraints, or relaxation, that is an elimination of some existing constraints.

Definition 4.1 (DCSP)**.** *Consider* P *a dynamic constraint satisfaction problem (*DCSP*) [Dechter and Dechter, 1988].* P *is a sequence of static CSPs* $P_0,...,P_\alpha,P_{\alpha+1},...$*, where each* P_i *differs from the previous* P_{i-1} *by the addition or removal of some constraints (we assume that all CSPs possible changes can be expressed in terms of constraint additions or removals).*

$$\begin{gathered} P_\alpha = (X,D,C_\alpha) \Rightarrow P_{\alpha+1} = (X,D,C_{\alpha+1}), \\ \text{where } C_{\alpha+1} = C_\alpha \pm C, \text{ such as } C \text{ is a set of constraints.} \end{gathered} \tag{4.1}$$

The assumption in the definition is explained by the following:

— **Restriction** occurs when a new constraint is imposed on a subset of existing variables, or when a new variable is added to the system. Restriction always expands the model i.e., adds variables and adds constraints related to these variables.

— **Relaxation** occurs when constraints/variables are disabled or retired from the system because they are no longer interesting or the current CSP has no solution.

There are more forms of dynamism in the DCSP network, the majority of these can be converted to adding/removing constraints. For example: removing a value from a domain can be translated as adding a unary constraint to the concerned variable, revision an existing constraint can be converted to remove and to add a new constraint with the new parameters. That is why we are interested in the rest of this thesis only to add and remove constraints as a form of dynamic event in the DCSP network.

4.3 Dynamic CSP approches

Dynamic CSP (DCSP) methods can be clustered into 3 categories :

— Heuristics methods which consist of establishing an optimal order strategy like in CSP.

— Local repair methods, which are based on a previous consistent assignment (complete or not) and repair it, using a sequence of local modifications (modifying one variable' assignment at a time) [Bliek, 1998, Ginsberg and McAllester, 1994b]

— Methods that aim to minimize the distance between successive solutions [Barták et al., 2003, El Graoui et al., 2016, Hebrard et al., 2005, Ran et al., 2002].

The methods of the first two groups focus on improving the efficiency of the search, in terms of the effort to find the final solution (problem after perturbation), while those of the third group aim to maintain the stability as much as possible of the problem solutions.

DCSPs were defined in [Dechter and Dechter, 1988] and [Schiex and Verfaillie, 1994] as series of CSPs that differ one from the other in some of their constraints (added or removed or both). They are solved by researching a solution for each CSP in the series as fast as possible. Several methods for acquiring a new solution for a changed CSP were proposed, like nogoods, dynamic backtracking, and local search. However, none of the proposed methods is guaranteed to find the solution most similar to the previous solution (when problems are multiple solutions).

In [Roos et al., 2000], authors propose a method that, given a revised CSP and a solution to the original problem, finds a solution to the new CSP that is the most similar to the previous solution. The authors establish that this method is feasible only when the changed constraints are unary and that it causes a significant slow down in comparison with an algorithm that searches for any solution to the new problem. In a later study, [Ran et al., 2002], They offer approximation feasible methods that do not guarantee the finding of the most similar solution.

[Hebrard et al., 2005] have been studied different aspects of finding similar and diverse solutions to specific assignments for CSPs, they prove that finding a solution to a CSP that is rather close to (or enough distant from) a given set of assignments is NP-hard. They also propose an algorithm, which is based on Branch and Bound and generalized arc consistency (GAC) for finding the closest solution to a given set of assignments. They propose, in [Hebrard et al., 2007], an algebra that enables a combination of similarity constraints to a set of ideal partial assignments, as well as distance constraints from non-ideal partial assignments.

In [El Sakkout et al., 1998], the minimal perturbation problems MPP is studied for classic scheduling problems. The authors propose an integration of mathematical programming techniques with constraint programming to speed up the search on this special optimization problem. The solutions are used within the constraint program. The authors extend this approach in [El Sakkout and Wallace, 2000] and discuss its potential to detect sub-problems, which can be solved more efficiently in different CSP applications.

Authors in [Barták et al., 2003] propose a formal model of the MPP based on the CSP model and a Branch and Bound Algorithm for solving it. The originality of the proposed algorithm is that it allows incomplete solutions to the problem. In [Zivan et al., 2011], authors adopt the formalism proposed in [Barták et al., 2003], but seek to find complete solutions to the problem with minimal perturbation as in [Hebrard et al., 2005].

In bellow, We will present and describe more deeply the basic Partial-order Dynamic Backtracking (PDB) method. although PDB is earlier, it continues to interest the CP community. An extension of this approach called EPDB has been proposed in [Acodad et al., 2012] to fix the limitations of the original PDB.

4.3.1 Local repairing method: PDB as example

The dynamic scheduling strategies offer the flexibility to rebuild a new order among the variables during the search process. One of the main approaches using this strategy is Partial-order Dynamic Backtracking (PDB). However, several questions are still open and not completely explored, especially in terms of the degree of flexibility in the backtracking strategy, and the evaluation of this flexibility in challenging problems such as DCSPs. The PDB replaces the fixed order of variables, used in several approaches to solve CSP, with a partial order that is dynamic and partially modified during the search process. The PDB uses the concept of nogood introduced into Dynamic Backtracking (DBT). In

contrast to DBT, when a nogood is induced during the search and when no order is imposed between variables concerned with this conflict, there may be significant choices for the variable that will emerge as the nogood conclusion. The performance of this approach depends mainly on the quality of this choice. The order between variables is not defined at the beginning, but there is a partial order that is built during the search process, to achieve the solution if it exists, or to detect the unsolvability through the empty nogood. The partial order between variables is represented by a data structure of the order named *Safety Conditions*.

Definition 4.2 (safety conditions)**.** *A safety condition is defined as an assertion of the form:* $x < y$*, with* x *and* y *variables.*

If S is a set of safety conditions, we denote by $\leq_s$ the transitive closure of $<$, meaning that S is acyclic if $\leq_s$ is antisymmetric.

$$x <_s y \Longrightarrow x \leq_s y, y \not\leq_s x \tag{4.2}$$

In another way: $x <_s y$, if there is a sequence (possibly empty) of safety conditions:

$$x < z_1 < z_2 < \ldots < z_n < y \tag{4.3}$$

For a nogood γ, we denote by S_γ all the safety conditions $x < y$, such that x is in the antecedent of γ and y is its conclusion.

A pair $\langle \Gamma, S \rangle$, with Γ a set of nogoods and S a set of safety conditions, is acceptable if Γ is acceptable, S is acyclic and $S_\Gamma \subset S$

Definition 4.3 (Weakening)**.** *For any set* S *of safety conditions and any variable* x*, we define the weakening of* S *at* x*, denoted by* $W(S,x)$ *the set of safety conditions obtained by removing from* S *all the safety conditions of the form* $z < y$ *as* $x <_s y$ *and adding* $x < y$ *for all* y*.*

Lemma 4.1. *For any set* S *of safety conditions, any variable* x*, and any total order* $<$ *consistent with the safety condition in* $W(S,x)$*, there exists a total order consistent with* S *and appropriate to* $<$ *through* x*.*

Algorithm 7: Partial Dynamic Backtracking Algorithm

```
Procedure PDB(CSP, newConstraints)
1.  CSP.update(newConstraints);
2.  constraints ← newConstraints;
3.  solutionFound ← False;
4.  emptyNgGenerated ← False;
5.  while (¬solutionFound && ¬emptyNgGenerated) do
6.     c ← getAConstraint(constraints)
7.     if (¬isConsistant(c)) then
8.        listVar ← getListOfVar(c);
9.        ng ← GenerateNg(listVar);
10.       CSP.ngList.add(ng);
11.       RepairSolution(Rhs(ng),constraints);
13.    else
14.       constraints.remove(c);
15.       if (isEmpty(constraints)) then
16.          solutionFound ← true;
17. CSP.ReturnResult();
Function RepairSolution(var,constraints)
18. if (var.Domain.isEmpty()) then
19.    ng ← NgResolution(var);
20.    if (ng.isEmpty()) then
21.       emptyNgGenerated ← True;
22.       Return False;
23.    if (¬RepairSolution(Rhs(ng),constraints)) then
24.          Return False;
25. CSP.SC.add(var);
26. CSP.ngList.removeWithLhs(var);
27. addInConstraints(var);
28. var.ChangeValue();
29. Return True;
Procedure GenerateNg(listVar)
30. lower = CSP.getLower(listVar);
31. if(lower! = NULL) then
32.    Return getNg(listVar,lower);
33. Return getNg(listVar,getAVar(listVar));
Procedure NgResolution(var)
34. listVar = CSP.NgList.getAllVarLhs(var);
35. GenerateNg(listVar,heuristic);
```

The Algorithm.7 shows PDB procedures, to reuse solution and past reasoning techniques, we start the search with a CSP problem that contains a full assignment (e.g. past solution) keeping retroactive data structures of nogoods NgList saved during the past search process. Then we update the set of constraints that need to be satisfied and we test the consistency of every constraint (lines 1 to 7). If the inconsistency is detected, a new nogood is generated using the procedure GenerateNg(listVar) (lines 7-9). Next, we add the new nogood into the NgList and we call the recursive function RepairSolution(Rhs(ng),constraints) trying to find a new instantiation to the variable Rhs(ng) (lines 10-11). After giving a value to Rhs(ng), we select another constraint to verify (line 6). If the constraint c is consistent, it is removed from the constraints set constraints (line 14). If all constraints are satisfied, then the solution is found and the search is ended (lines 15-16). As we said, the procedure RepairSolution(var,constraints) tries to find an instantiation to the variable var, so we check if the domain of this variable is empty or not (line 18). If RepairSolution(var,constraints) cannot assign a value to the variable var (e.g. domain wipe-out), it builds a nogood ng that is the resolution nogood of all nogoods where var appears in the conclusion using the function NgResolution(var)(line 19). If the resolved nogood is empty, that means no solution can be found and the search is ended (lines 20 to 22), else the function RepairSolution(Rhs(ng),constraints) is called recursively and tries to find a new instantiation to the variable Rhs(ng). If the variable Rhs(ng) cannot change its value due to a dead-end, an empty nogood is generated, then we return False (no solution, lines 23-24). Otherwise,

RepairSolution(var,constraints) deletes all nogoods with var in the antecedents and preserves orders imposed by them in the safety conditions list SC, then it assigns a new value to var from its valid domain (line 25 to 29).

The procedure GenerateNg(listVar) is used to choose, from a list of variables, the lower variable that will appear in the conclusion of the nogood, respecting the order already imposed in the safety condition and the list of nogoods (lines 30 to 32), else the procedure chooses a variable that will be on the right-hand side of the nogood(line 33).

NgResolution(var) is called when the variable detects a dead-end. In this case, the procedure stocks in the structure listVar all the variables that are on the left-hand side of all nogoods that prohibit var's values (line 34). Then, it calls the procedure GenerateNg(listVar) to generate the relevant nogood (line 35).

4.3.2 Heuristics methods

As it turns out, the constraints network's performance appears when choosing a sub-network to update, the variable chosen is the one having more chance to take a consistent value. Thus, the variable ordering heuristics [Boussemart et al., 2004b] and [Ortiz-Bayliss et al., 2013] help to build an optimal order between variables, especially, when no conditions exist between variables. The common idea behind the use of heuristics is to construct order in such a way that the most constrained variables in the network, during the backtracking process, take the highest priority order, to retain their stability.

Given the flexibility of the PDB algorithm in the backtracking strategy (i.e. choosing the culprit variable), repair heuristics can be introduced to guide the algorithm to the most relevant search area. Hence, the variables with lower priority are those who will change their values. In PDB, the VOHs are especially useful in two situations: Creating a nogood in constraint's inconsistencies, and resolving a nogood in a domain's wipe-out. These ideas are captured by variable selection heuristics.

— *Heuristic of maximum conflicts* (conf): Given a nogood store, the number of conflicts of a variable is the number of nogoods where this variable appears, such as it's in the conclusion. This heuristic reorders variables in a nogood, with decreasing degree of priority from the most conflicted variable. Hence, the variable with a maximum conf will be on the left-hand side of the generated nogood.

— *Heuristic of minimal domain* (dom): A variable is called minimal dom if the size of its domain is less than the other variables. This heuristic chooses the variable with the smallest remaining domain to be in the higher priority order. Hence, the variable with a large domain will be on the right-hand side of the generated nogood.

— *Heuristic of degree* (deg): The degree of a variable X is the number of constraints in which X participates (since constraints are binary). This heuristic selects the variable with the largest degree to be in the higher priority order.

— *Heuristic of dynamic degree* (ddeg): The dynamic degree of a variable X is the number of constraints where X appears and X future variables (since constraints are binary). This heuristic chooses the variable with the largest dynamic degree to be in the higher priority order.

— *Heuristic of weighted degree* (wdeg): Each constraint has an initial weigh. The weighted degree (wdeg) of a variable X is the sum of the weights of the constraints where X appears and at least one X future variable:

$$\alpha_{wdeg}(X_i) = \sum_{\substack{c \in Constraints \\ such\ X_i \in vars(C) \wedge |Future\_Vars(C)| > 1}} weight[C] \tag{4.4}$$

In wdeg procedure, a constraint's weight is incremented during consistency propagation whenever this constraint causes a violation. The wdeg chooses the variable with the largest weighted degree to be in the higher priority order.

The use of one heuristic and only one H1 may be useless in some situations since involved variables can have the same heuristic values. thus, using another heuristic H2 may help to build optimal order.

Then, the heuristic is $H1 \oplus H2$. The ratio of two heuristics $H1$ and $H2$ or more than two can also be calculated as a heuristic. For example, the $dom/wdeg$ is a heuristic based on the two heuristics dom and $wdeg$.

Minimal Perturbation Problem

The 3^{th} category of the DCSP methods is the method that tries to find the closest solution between the pre-perturbation problem solution P_0 and the post-perturbation problem P_1 regarding a given distance which is called Minimal Perturbation Problem (MPP) [El Sakkout and Wallace, 2000].

Definition 4.4 (MPP). *A Minimal Perturbation Problem (MPP) is a triple* $\Pi = (\Theta, \alpha, \delta)$*, where:*

— Θ *is a CSP.*

— α *is a partial or complete assignment for* Π *that is called initial assignment.*

— δ *is a function that defines a distance between any two assignments.*

A solution to an MPP problem is a solution to Π with minimal distance from α according to δ.

Definition 4.5 (Distance set D). *Let* σ *and* γ *be partial assignments for* Θ. $D(\sigma,\gamma)$ *is the set of variables such that the value assignment for* v *in* σ *is different from the value assignment for* v *in* γ *:*

$$D(\sigma,\gamma) = \{v \in V \mid \langle v, val\rangle \in \sigma, \langle v, val'\rangle \in \gamma, val \neq val'\} \tag{4.5}$$

$D(\sigma,\gamma)$ *is called a distance set for* σ *and* γ*, and the elements of the set are called perturbations.*

Definition 4.6 (Function δ). *In an MPP, the distance function of some assignment* σ *from* α *is defined as the cardinality of the set* $D(\sigma,\alpha)$*:*

$$\delta(\sigma,\alpha) = |\, D(\sigma,\alpha)\,| \tag{4.6}$$

Otherwise, δ *is the* hamming distance *between* σ *and* α*.*

Definition 4.7 (Solution Value Assignment (SVA)). α *is the initial assignment. if* $\langle v_i, val_{i_j}\rangle \in \alpha \Longrightarrow v_i.SVA = val_{i_j}$

4.4 Extended PDB

As seen in PDB, once a constraint C is checked, if this one is inconsistent, the approach selects another constraint to check, without ensuring the satisfaction of C, which will be visited later again. This flexibility of choosing a constraint to be satisfied without any restriction can be exploited positively to make a relevant order between constraints.

The goal of EPDB [Acodad et al., 2012] is to extend the classic description of PDB, by exploiting its flexibility to repair assignments using repair-directed heuristics to control changes. EPDB is completely focused on Variable Ordering Heuristic flexibility. However, when the constraint C is violated, and when x is chosen as a culprit variable, we assign to x a new value in its domain and we test the consistency of this value according to C. Selecting a new constraint to verify will not be done before satisfying this one.

The advantages of EPDB algorithm, like intelligent backtrackers, is that it records information regarding dead-ends to avoid encountering them again, and then allows great freedom in backtracking strategy, using retroactive data structures (nogoods and safety conditions) that guide the search process in a dynamic environment. Therefore, it appears to be very flexible in terms of search space exploration. As far as the authors are aware, this point of backtracking flexibility remains open and unexploitable until today. EPBD explores and benefits the flexibility of PDB using variables ordering heuristics (VOHs) to build an optimal partial order between variables. And as a matter of fact, the problem-solving process should naturally continue as smoothly as possible after any change. And particularly, the solution to the new problem should not be too much different from the solution of

the initial problem. Using EPDB for minimal perturbation, by combining it with some general heuristic functions that maintain Solution Value Assignments *SVAs* in the research process. These heuristics were distilled from an analysis of the Minimal Perturbation concept and have the asset of being extremely simple and keeping good performance in terms of research performance. It can be implemented very efficiently within a DCSP framework and combined with various search strategies. Hence, when there is a problem change on a DCSP and a solution has to be repaired according to a constraint network, we have always to change the less constrained sub-problem. Otherwise, we keep as long as possible the most complex sub-problems untouched, to avoid falling into other dramatic perturbations. If we reason along with the concept of nogoods, this is reflected in the allocation of the variable in the less relevant sub-problem to the conclusion of nogoods, whenever there is a conflict that must be justified or a nogood that must be resolved according to a dead-end. Here is the usefulness of variable selection heuristics.

Authors in [Acodad et al., 2012] experiment EPDB for a reparation purpose and also for MPP problem on both dynamic random binary problems and dynamic meeting scheduling problems. The results show that EPDB can guide algorithms toward a good quality-oriented solution while keeping good performance in terms of run time and constraints checks.

4.5 Conclusion

In this chapter, we introduced the concept of Dynamic CSP, and we presented briefly some popular related methods class i.e. Local repair methods, Heuristics methods, Minimal Perturbation Problem. We described much more deeply the PDB approach and its extension EPDB. In the next chapter, we will present some of our contributions related to the DCSP environment: (i) PDeg a Profound Degree heuristic, (ii) An extension based-repair heuristic approaches of EPDB that we called IPDB,(iii) a novel approach for DCSP which we called Lazy Repairing Backtracking (LRB).

CHAPTER

5

Contributions on Dynamic Constraint Satisfaction Problem

In the previous chapter, we recall some fundamentals related to designing Dynamic CSP and some famously used methods to solve them. This chapter is dedicated to presenting a series of our contributions on the Dynamic CSP: from heuristic methods to improved approaches to deal with DCSP.

Contents

As outlined in the previous chapter several techniques have been developed to be used in repair algorithms. We cite, for example, the variables/values ordering heuristics and local search techniques. In this chapter, we will present a new static variable ordering heuristic called Profound Degree (pdeg), based on the classical deg heuristic to deal with (DCSP).
In addition, based on a deep analysis of EPDB and its various data structures i.e; *safety conditions* and nogoods, which are saved during the search.
we propose an Improved Partial-order Dynamic Backtracking (IPDB) to deducing directly order between two conflicted variables by supporting an additive structure of orders.
Moreover in this chapter, we propose a fast version of EPDB, the Lazy Repairing Backtracking (LRB) approach. This algorithm aims to repair solutions using EPDB advantages while eliminating unnecessary tests of orders.
This chapter is organized as follow: we present in section 1 the Pdeg heuristic, then in the next section we describe deeply data structures of EPDB and present our Improved version of EPDB, then in section 3 we outline The Lazy Repairing Backtracking (LRB), finally, we conclude and present some perspectives.

5.1 Profound Degree Heuristic (pdeg)

We distinguish between static heuristics, which calculate their values once at the beginning of the search, and dynamic heuristics that use an expensive intelligence in terms of solving time.
In this section, we propose a new static variable ordering heuristic, Profound Degree (pdeg), based on deg heuristic, which calculates the degree of influence of a given variable, on the whole constraints network, relatively to its position in the network.
We evaluate this heuristic on the Extended Partial-order Dynamic Backtracking (EPBD) approach, which is an approach to repair DCSPs solutions, and we compare it to the best-known variables ordering heuristics (VOHs) for repairing. The evaluation of performance is on random binary problems and meeting scheduling problems, with the criteria of computation time, the number of constraints checks, and Hamming distance between the former and the current solution.

5.1.1 Description of heuristic

The deg VOH [Dechter and Meiri, 1989] has a major advantage, it's that it does not compute every time dynamic information about the current state of research (such as the size of the current domain dom or the degree of the current variable ddeg [Sabin and Freuder, 1997]), but it operates static information.

However, in the context of repairing solutions, this heuristic appears limited. No information regarding the position of variables concerning the entire network is operated, whereby the change of a variable can affect not only the neighbors but all of the variables in the network with a given probability. For this, we propose to calculate an estimation of this information for each variable X at the beginning of the research, considering all the network constraints, and by associating to each constraint a weight, higher or lower, concerning its distance from X. pdeg is considered as the sum of those weights.

Now consider the simple network above in Fig.5.1. Suppose that each variable has a consistent value and due to a perturbation of the DCSP, the value of E must be changed. Assume that all constraints have the same tightness p_2. Changing E will generate the following probabilities P:

$$P(violate(C_{EB})) = P(violate(C_{EF})) = p_2$$
$$P(violate(C_{BA})) = P(violate(C_{BC})) = P(violate(C_{BD})) = P(violate(C_{FG})) = p_2 * p_2 = {p_2}^2$$

If the probabilities of violating constraints are considered as weights, knowing that $p_2 \in [0,1]$, the weights of constraints network are:

$$Weight(C_{EB})_{/E} = Weight(C_{EF})_{/E} = p_2 \geqslant$$
$$Weight(C_{BA})_{/E} = Weight(C_{BC})_{/E} = Weight(C_{BD})_{/E} = Weight(C_{FG})_{/E} = {p_2}^2$$

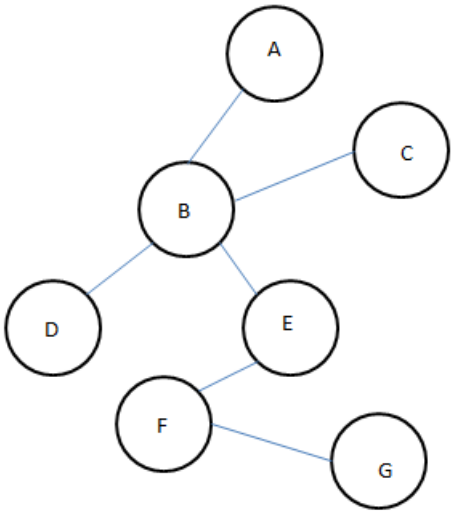

Figure 5.1 – A simple constraints network

This reflects that, whenever the constraint C is closer to the variable X, C is more influenced by the change of X, since the variables concerned by C are more influenced by changing X.

In fact, given two constraints C_1 and C_2, if the constraint C_1 is closer to X than C_2, then C_1 has a cost in calculation of the heuristic value of X which is more interesting than C_2, since changing X engenders a risk of violating C_1 more important than violating C_2.

In reality, constraints networks are not as simple as that, they are quite as figure 5.2:

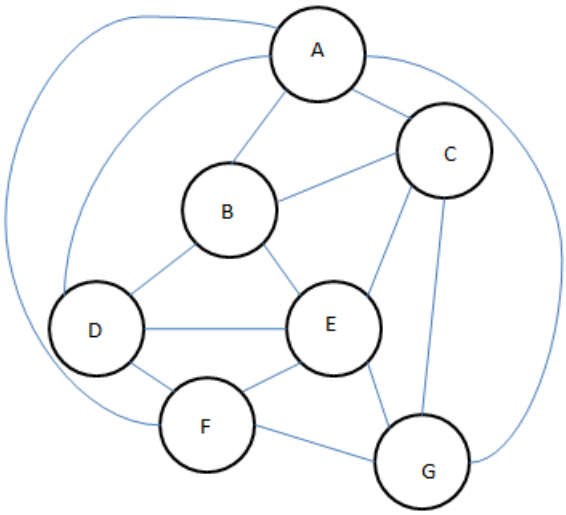

Figure 5.2 – A constraints network

On the other hand, the hardness of the constraints network (especially when the constraints used

in the modeling are intent) is not always known, the reason why choosing to replace the weight, that corresponds to the tightness of the constraints, by an average hardness of 1/2.

Inspired by this idea, pdeg heuristic accumulates, for each variable X, the weights of all constraints. These weights are higher each time the constraints are closer to X.

5.1.2 Execution of heuristic

Algorithm 8: Description of pdeg heuristic

```
Function PDEG(X,CSP)
1.  pdeg ← 0;
2.  level ← 0;
3.  size ← CSP.Vars.size();
4.  actual ← X;
5.  total ← X;
6.  while(total.size() != size)
7.     neighbors ← getAllNeighbors(actual);
8.     filter(neighbors,total);
9.     actual ← getDistinct(neighbors);
10.    a ← neighbors.size();
11.    b ← nbConstraints(actual);
12.    total.add(actual);
13.    pdeg += 1/2^incLevel() × a + 1/2^incLevel() × b;
14. Return pdeg;
```

Both the value of pdeg and the level, which is incremented each time the constraint is far from the variable in question, are initialized by 0 (lines 1 and 2). The variable size contains the number of variables in the CSP (line 3) and actual and total lists, initiated by the variable for which having to calculate the pdeg, serve respectively to store all variables in the current level and all variables already involved in the calculation (lines 4 and 5).

Will not accounting all constraints network variables, the heuristic calculates the neighborhood of the current variables, then this neighborhood is filtered to include only new variables, to not record the same constraint twice (lines 6-8).

Note that getAllNeighbors() (line 7) recognizes a neighbor many times as the number of constraints that bind it with the current variables.

The heuristic puts into actual current neighbors eliminating redundancies, in a the number of neighbors of actual variables with redundancy (which is equivalent to the number of constraints with neighbors) not yet recognized, and in b the number of constraints between these neighbors (lines 9 and 11). At this point, it adds to total the current neighbors and to pdeg the weight of neighbors, which is 1/2 to the power the level incremented, multiplied by the number of redundant neighbors, and it does the same thing for the number of constraints between neighbors, after incrementing the level (lines 12 and 13).

The heuristic leaves the while loop once all variables are accounted for, which means it counted all the constraints of the network, and the value of pdeg is returned as the profound degree of the variable X (line 14).

5.1.3 Example of simulation

We consider the constraints network above and assume that all variables have a consistent value ($A = 1$, $B = 1$ and all other variables have values consistent with the whole constraints network). We suppose that the DCSP has undergone a disturbance that is the change of the nature of the constraint C_{AB}, that was: $A = B$, and that became: $A < B$. To satisfy this constraint, the EPDB algorithm [Acodad et al., 2012] is restarted to repair the solution. EPDB chooses to change the value of A, or the one of B, depending on the used heuristic.

Using $EPDB_{deg}$, heuristic values are:

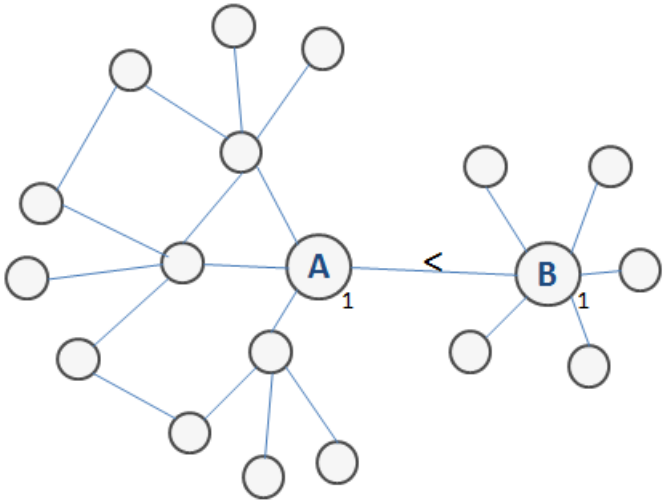

Figure 5.3 – Graph of constraints

— $deg(A) = 4$

— $deg(B) = 6$.

So the approach generates the nogood: $B = 1 \Longrightarrow A \neq 1$, since it retains the value of the most relevant variable in the network, i.e., the one whose change can disrupt more the DCSP and delay resolution. For this reason, the value of A is changed.

Now, using $EPDB_{pdeg}$, the values of used heuristic are:

— $pdeg(A) = 1/2 \times 4 + 1/2^2 \times 1 + 1/2^3 \times 14 + 1/2^4 \times 2 = 4,125$

— $pdeg(B) = 1/2 \times 6 + 1/2^2 \times 0 + 1/2^3 \times 3 + 1/2^4 \times 1 + 1/2^5 \times 9 + 1/2^6 \times 2 = 3,75$

Then the approach generates the nogood: $A = 1 \Longrightarrow B \neq 1$ and changes the value of B.

5.1.4 Experimental results

We carried out a series of experimental tests to compare the integration of pdeg heuristic, within the EPDB algorithm, to other VOHs. Heuristics used for repairing are those that improve the best the behavior of EPDB, namely the degree (deg), the domain size (dom), the number of conflicts (conf) and the ratio between the domain size and the weighted degree (dom/wdeg) [Boussemart et al., 2004b]. We evaluate the performance in terms of constraints checking (CCs), computing time (Time(sc)) and Hamming distance (HD) [Hebrard et al., 2007] and [Zivan et al., 2011]. All experiments were performed on the Java platform.

Experiments on Randoms. Experiments have been performed on sparse and dense random problems section 2.2.1, respectively $< 20,10,0.25,0.6 >$ and $< 20,10,0.75,0.27 >$. The constraints tightness is selected such as areas are as complex as possible while problems have the solution.

For logical reasons, we assume that the maximum rate of added constraints is equivalent to ~25%. For each pair, 20 instances were solved using each heuristic, and the results are presented as an average of these 20 instances.

Fig.5.4 presents the effectiveness of heuristics applied to the EPBD, for the most complex regions of the low density selected. In terms of the number of constraints tested CCs, between 10% and 24%

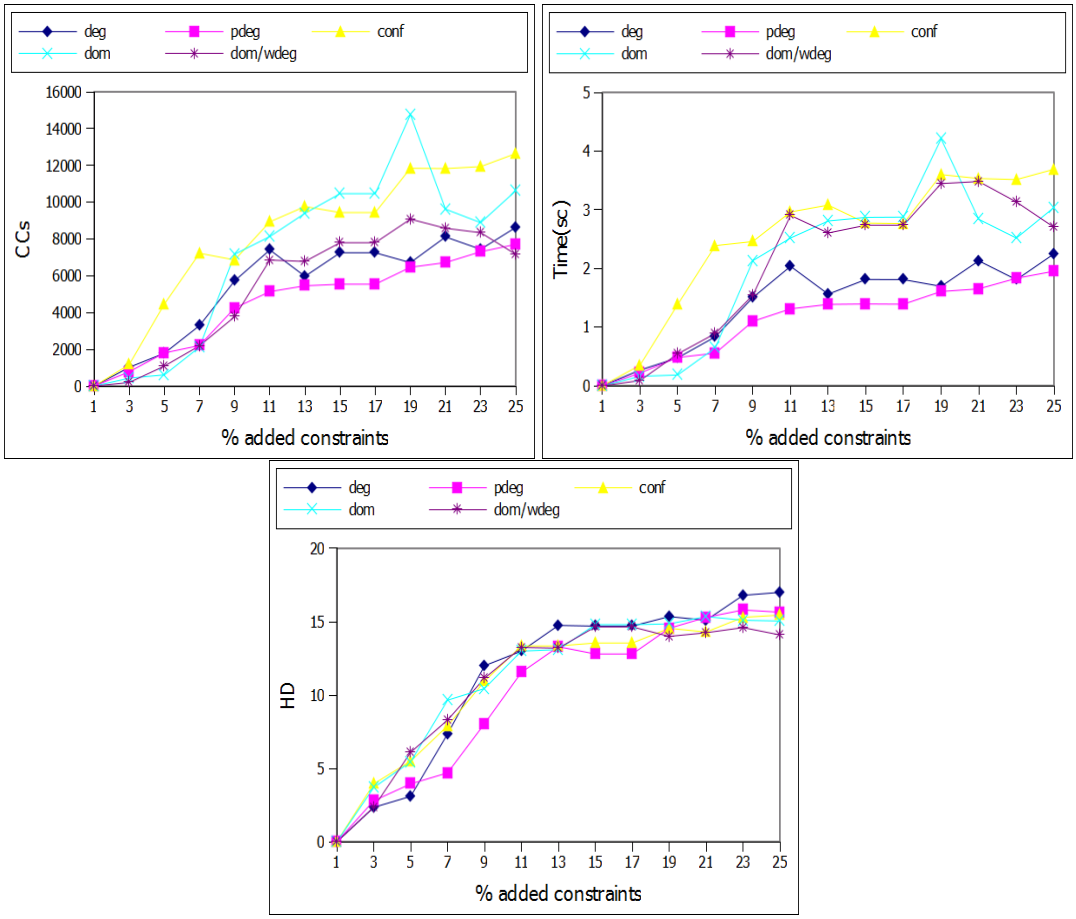

Figure 5.4 – Number of CCs, Execution time performed and HD (p_1=0.25, p_2=0.60)

of new injected constraints, pdeg is the best, elsewhere, dom/wdeg and dom are the most efficient. Otherwise, the deg is the closest to pdeg, which seems the most stable. In time graph, dom/wdeg recedes (also for dom but not in the same manner), due to the calculation of dom and wdeg heuristics, especially for wged. pdeg is almost always the best. Concerning the Hamming distance HD, there is a competition between heuristics, and pdeg remains a good competitor in most regions.

Fig.5.5 shows the effectiveness of the heuristics for the most complex areas of the high density p_1=0.75. In terms of a number of CCs and execution time, pdeg usually looks the best although not improving too much the behavior of deg. Concerning Hamming distance, no heuristic is absolutely the best.

Experiments on Dynamic Meeting scheduling problems. We present our results for the MSP section 2.2.1 class < 20,5,15,5,10,1,70 > and we vary the rate of changed constraints from ~1% to ~25%. We generated 20 different instances solved using each heuristic in EPDB.

Fig.5.6 shows the effectiveness of the heuristics in terms of number of CCs, execution time and HD. pdeg leads to a great improvement in most areas (from 10% of disrupted constraints in terms of CCs, 7% in terms of time and 11% in terms of HD).

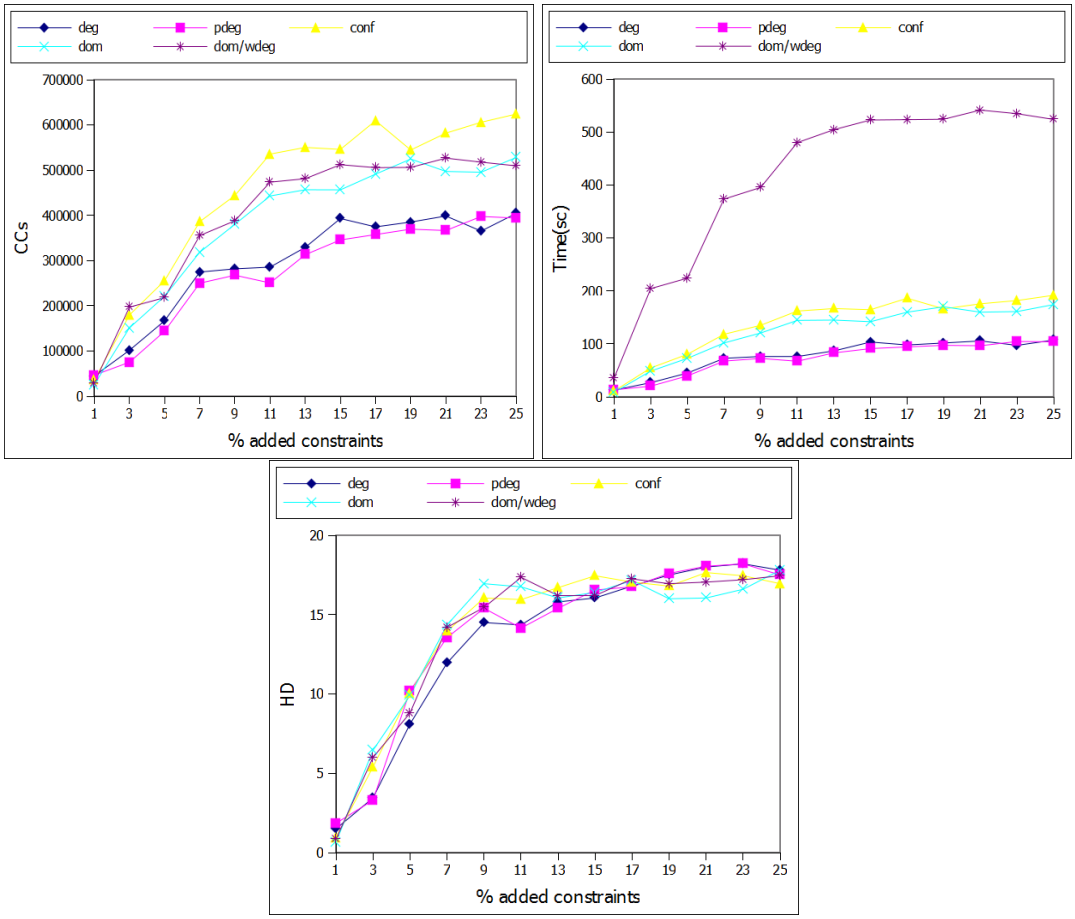

Figure 5.5 – Number of CCs, Execution time performed and HD (p_1=0.75, p_2=0.27)

5.1.5 Discussion

For low densities, pdeg behaves better than deg, more remarkably than high densities. In fact, in high densities, almost all variables are connected, i.e., for each given variable, almost all other variables in the network are its neighbors. Therefore calculating pdeg of a variable almost returns to calculate its deg. pdeg leads, especially for low densities, to a minimum Hamming distance (HD). The idea of pdeg is to change the value of the least constraining variable in the network, i.e., the one that, when changing its value, will cause the least change to other variables. And knowing that the variables before the correction have SVAs values, so a minimum change of network variables is a minimum change of SVAs, then a maximum of SVAs. Although the heuristics dom, conf, and dom/wdeg, in some regions, do fewer constraints checking than static heuristics deg and pdeg, they consume more computational time because of their dynamic aspect.

5.1.6 Conclusion

In this section, we introduce a new VOH, pdeg heuristic, for estimating the influence of variables in the whole constraint network, in the process of repairing solutions in dynamic environments. Based on the results of experiments on a wide class of problems, we found and proved that this heuristic, used in DCSPs correction, exceeds the heuristics deemed to be efficient when solving CSPs. The idea of

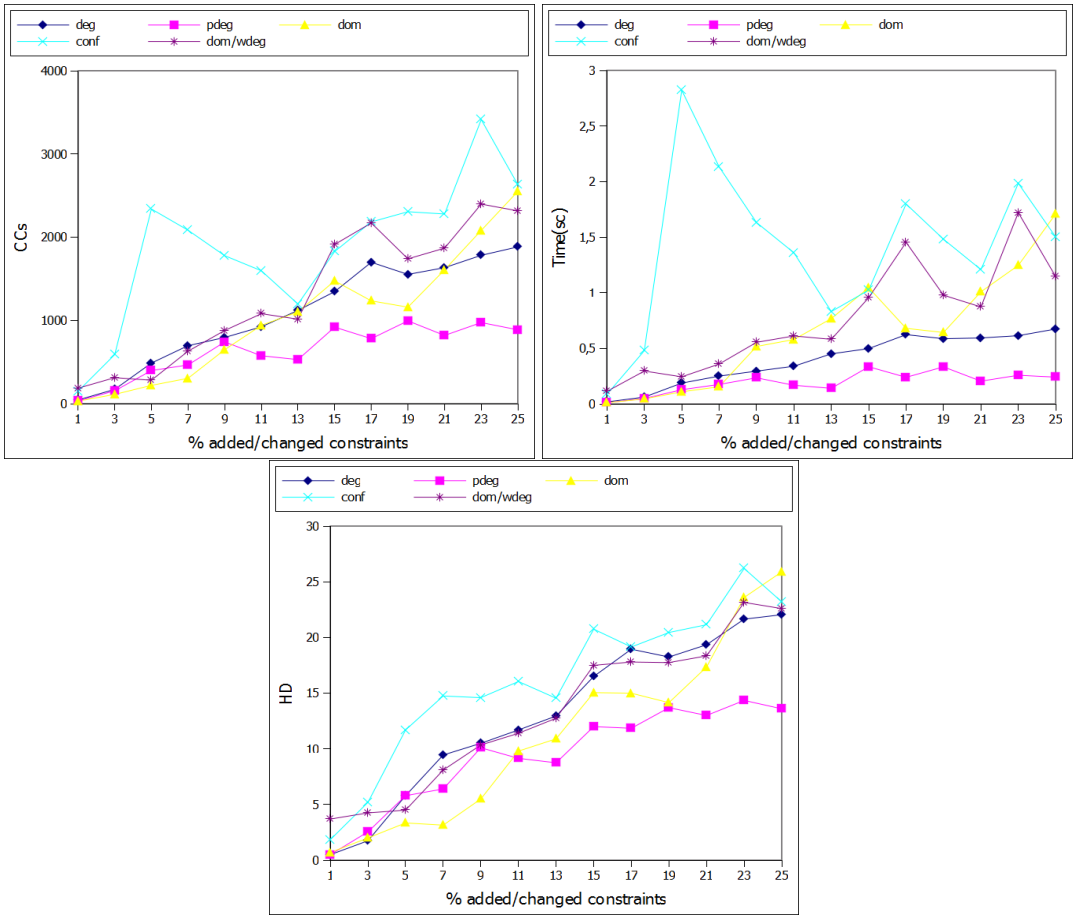

Figure 5.6 – Number of CCs, Execution time performed and HD

pdeg heuristic is to disturb only the parts of the constraints network, whose probability of influence is minimal. Indeed, for low to medium densities, as seen in the random problems with density $p_1 = 0,25$ and the MSP (and especially for medium densities), using pdeg improves remarkably the repair of a solution, not only concerning research performance (time and CCs), but also concerning the quality of the solution (Hamming distance).

5.2 Improved Partial-order Dynamic Backtracking

Dynamic Constraint Satisfaction Problems (DCSPs) have been proposed to represent problems that change over time. To effectively repair Dynamic CSPs, EPDB uses ordering heuristics and retroactive data structures, *safety conditions* and *nogoods*, which are saved during the search process.
In this section, we show that the drawback of EPDB is the exhaustive verification of orders, saved in *safety conditions* and *nogoods*, between variables. This verification affects remarkably search time, especially since orders are often indirectly deduced.
Therefore, we propose an improved version of EPDB, the Improved Partial-order Dynamic Backtracking (IPDB), which is a fast version of EPDB insofar as it deduces orders directly, by supporting an additive structure of orders, which allows having the information about all successors of each variable.
We evaluate IPDB on various kinds of problems, with the use of different variables ordering heuristics, and compare it, on the one hand, with EPDB to show its effectiveness compared to this approach, and, on the other hand, with MAC-2001 to conclude, from what perturbation rate, resolving a DCSP with an efficient approach can be more advantageous than repair.

The first improvement of Extended Partial-order Dynamic Backtracking (EPDB) [Acodad et al., 2012] is to extend the classical description of PDB, by exploiting its flexibility to repair assignments using ordering heuristics to control changes. Thus, in a conflict situation, when no order exists between variables in conflict, EPDB changes the less relevant sub-problem. Otherwise, it keeps as long as possible the most difficult sub-problems intact, to avoid falling into more complicated conflicts. Technically, this is equivalent to attribute the variable in the most complex sub-problem to the nogood conclusion, whenever a conflict has to be justified, or a nogood resolution occurs due to a dead-end. The usefulness of variable ordering heuristics (VOHs) appears in these two situations.
The second improvement of EPDB is to satisfy the current constraint C_i before selecting another one to verify. In fact, when a checked constraint C_i is violated, PDB [Ginsberg and McAllester, 1994b] changes the value of a C_i concerned variable and another constraint is checked. The satisfaction of C_i is not insured, but it will be re-checked later.

Algorithm 9: Extended Partial Dynamic Backtracking Algorithm

```
Procedure EPDB(DCSP, newCstrs, VOH)
1.  add newCstrs in DCSP.cstrs;
2.  clear DCSP.SC;
3.  solutionFound ← False;
4.  emptyNgGenerated ← False;
5.  c ← a constraint from DCSP.cstrs;
6.  while (¬solutionFound && ¬emptyNgGenerated) do
7.    if (c is not consistent) then
8.      conflictVars ← variables of c;
9.      ng ← GenerateNg(conflictVars);
10.     add ng in DCSP.ngList;
11.     Repair(Rhs(ng));
12.   else
13.     move c from DCSP.cstrs to DCSP.freeCstrs;
14.     c ← a constraint from DCSP.cstrs;
15.     if (c is null) then
16.       solutionFound ← True;
17. Return Result;
Function Repair(var)
18. if (domain of var is empty) then
19.   ng2 ← ResolutionNg(var);
20.   if (ng2 is empty) then
21.     emptyNgGenerated ← True;
22.     Return False;
23.   if (¬Repair(Rhs(ng2))) then
24.       Return False;
25. remove from DCSP.ngList nogoods as var in Lhs;
26. update DCSP.SC according to deleted nogoods;
27. move var's constraints from DCSP.freeCstrs to DCSP.cstrs;
28. change value of var;
29. Return True;
Function GenerateNg(conflictVars)
30. lower ← get lower priority in conflictVars;
31. if(lower is not null) then
32.   Return nogood between conflictVars where lower is Rhs;
33. Return nogood between conflictVars according VOH;
Function ResolutionNg(var)
34. conflictVars ← variables in Lhs of nogoods prohibiting values of var;
35. GenerateNg(conflictVars);
```

To reuse solution and past reasoning techniques, EPDB in Algorithm.9 starts the search with a DCSP that contains a full assignment (e.g. past solution) keeping retroactive data structures of nogoods ngList saved during the past search process.
EPDB updates the set of constraints cstrs which need to be verified and satisfied; it clears the Safety Conditions list SC, which contains the old orders expressed by deleted nogoods, and initiates the information of the solution obtainment and the empty nogood at false. Then, it checks the consistency of every constraint till finding a solution or generating the empty nogood (lines 1-7).
If an inconsistency is detected, a new nogood ng is generated using GenerateNg (lines 7-9). The function is called to determine the variable, among the variables in conflicts conflictVars, which will represent the right-hand side (Rhs) of the nogood. It generates the nogood respecting the existing order, which is expressed by both of the list of nogoods ngList and safety conditions SC (lines 30-32), or according to the chosen variable ordering heuristic VOH if no order exists (line 33). Next, the new nogood is added into the list of nogoods ngList, and Repair, the recursive function, is called, in order to find a new assignment to Rhs(ng), the right-hand side variable (the conclusion) of ng (lines 10 and 11).
If the constraint c is consistent, it is moved from the DCSP constraints set cstrs to freeCstrs, which saves all checked and satisfied constraints of the DCSP. Then, another constraint is chosen to be

checked (lines 12-14). If all constraints are satisfied (DCSP.cstrs is empty), the solution is found, and the search is ended (lines 15-17).
As already said, the function Repair looks for an assignment to its parameter var, therefore, it verifies its domain (line 18). If the function cannot assign a value to var due to a domain wipe out, it builds a nogood ng2, which is the resolution nogood of all nogoods where var appears in the conclusion, using the procedure ResolutionNg (lines 18 and 19). If the resolved nogood is empty, no solution can be found, thus, the search is ended (lines 20-22). Else, Repair is called recursively to look for a new assignment to Rhs(ng2), the right-hand side of the nogood ng2. If Rhs(ng2) cannot change its value due to a domain wipe out, an empty nogood is generated. The function returns False due to the solution's non-existence (lines 23 and 24). Otherwise, Repair deletes all nogoods where var exists in the antecedent and updates the safety conditions list SC (lines 25 and 26). Updating SC consists of deleting from it all orders where the right-hand side is equal to the right-hand side of any of deleted nogoods, before preserving orders imposed by them in SC. Then, it updates the set of constraints by moving to cstrs the constraints involving var and associates to it a new value from its valid domain (lines 27-28).
ResolutionNg is called when the variable var detects a dead-end. Thus, the procedure stores, in a structure conflictVars, all variables appearing on the left-hand side (Lhs) of all nogoods prohibiting var's values (line 34). Then, it calls GenerateNg to generate the relevant nogood (line 35).

5.2.1 An example run

Let's consider the CSP illustrated in Fig.5.7:

— X = {X1, X2, X3, X4}.

— D = {D_1, D_2, D_3, D_4} such as $D_i = \{1,2,3\}\ \forall i \in [1,4]$.

— C = {$X1 < X2$, $X1 \neq X4$, $X2 = X3$, $X2 \neq X4$ }.

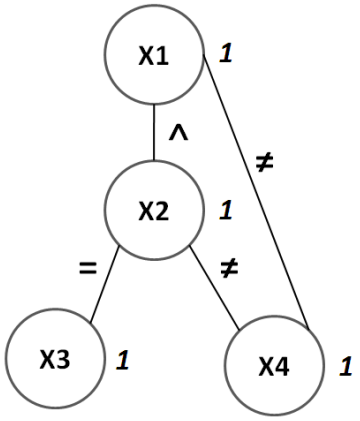

Figure 5.7 – A constraint network example

All variables are assigned by the value 1. For simplicity, we consider that this first assignment (before perturbation) was found using a simple resolution approach (FC for example), then, at the beginning of repairing, no nogood or safety condition is saved in the last search. Thus, no order is imposed between variables.
Tab.5.1 describes EPDB using the deg heuristic. The approach calculates, before starting the search, deg of each variable in the network: $deg(X1) = 2$, $deg(X2) = 3$, $deg(X3) = 1$ and $deg(X4) = 2$.
All variables have the same assigned value 1. This assignment violates more than one constraint. A constraint ordering heuristic [Hammoujan and Acodad, 2013] can be used to improve the search, but,

to treat different cases, we assume that the constraints are ordered as shown in *to check* column.
The first checked constraint *checked* is the one between X1 and X2 (line 1). This constraint is violated, and as no order exists yet between X1 and X2 (*add* column is empty), the heuristic value of X1 and X2 is verified to determine the order. $deg(X1) < deg(X2)$, thus, the generated nogood is (1). X1 changes the value and the constraint is re-checked before selecting another one (line 2). C_{12} is not consistent, then, the existence of order between X1 and X2 is tested before generating the nogood. (1) is tested then (2) is generated. X1 changes the value and the constraint, which is still not consistent, is re-checked (line 3). To generate the nogood, the existence of order between X1 and X2 is tested. Then, (1) is tested and (3) is generated. X1 has no more values in its domain, then, EPDB resolves (1), (2) and (3) as (4), removes the three no valid nogoods (line 4), saves their expressed order as a safety condition (5) and changes the value of X2, the unique cause of the domain wipeout of X1 (line 5). X1 is assigned to its first valid value, and the consistent constraint is rechecked (line 6). As C_{12} is satisfied, it's moved to the list of free constraints *free cstrs*, which contains all tested and satisfied constraints, and the next constraint to verify is checked. C_{14} is violated (line 7). The existence of order between X1 and X4 is tested, but no order in *add* contains neither X1 nor X4. Then, the heuristic value is verified. As $deg(X1) = deg(X4)$, any order is chosen, we assume that in similar cases, the chosen order is the lexicographic one. Thus, the nogood is generated as (6). X4 changes the value, then, C_{14} is consistent (line 8) and it's moved to *free cstrs*. The next constraint to check is C_{24} which is violated (line 9), (5) and (6) are tested, then, (7) is generated, the value of X4 is changed (line 10) and C_{14}, the only constraint in *free cstrs* concerning X4, is moved to *to check* list. The next checked constraint is C_{23} which is violated (line 11), the tested orders are (5) then (6), and (7). As no order exists between the two variables, the chosen heuristic is used to generate (8), then, the value of X3 is changed (line 12). C_{23} is consistent, thus, it's moved to *free cstrs*. C_{14} is then checked (line 13), it's satisfied so it's moved to *free cstrs*. *to check* is empty (line 14), thus, all constraints are satisfied and the solution is found.

Table 5.1 – Execution of $EPDB_{deg}$

	to check	**free cstrs**	$\mathbf{X_1\ X_2\ X_3\ X_4}$	**checked**	**test**	**add**	**remove**
1	$C_{12}\ C_{14}\ C_{24}\ C_{23}$		1 , 1 , 1 , 1	C_{12} ×	deg	X_2=1 ⇒ $X_1 \neq 1$ (1)	
2	$C_{12}\ C_{14}\ C_{24}\ C_{23}$		2 , 1 , 1 , 1	C_{12} ×	(1)	X_2=1 ⇒ $X_1 \neq 2$ (2)	
3	$C_{12}\ C_{14}\ C_{24}\ C_{23}$		3 , 1 , 1 , 1	C_{12} ×	(1)	X_2=1 ⇒ $X_1 \neq 3$ (3)	
4	$C_{12}\ C_{14}\ C_{24}\ C_{23}$		- , 1 , 1 , 1			$X_2 \neq 1$ (4)	(1),(2),(3)
5	$C_{12}\ C_{14}\ C_{24}\ C_{23}$		- , 2 , 1 , 1			$X_2 < X_1$ (5)	
6	$C_{12}\ C_{14}\ C_{24}\ C_{23}$		1 , 2 , 1 , 1	C_{12} √			
7	$C_{14}\ C_{24}\ C_{23}$	C_{12}	1 , 2 , 1 , 1	C_{14} ×	deg	X_1=1 ⇒ $X_4 \neq 1$ (6)	
8	$C_{14}\ C_{24}\ C_{23}$	C_{12}	1 , 2 , 1 , 2	C_{14} √			
9	$C_{24}\ C_{23}$	$C_{12}\ C_{14}$	1 , 2 , 1 , 2	C_{24} ×	(5),(6)	X_2=2 ⇒ $X_4 \neq 2$ (7)	
10	$C_{24}\ C_{23}\ C_{14}$	C_{12}	1 , 2 , 1 , 3	C_{24} √			
11	$C_{23}\ C_{14}$	$C_{12}\ C_{24}$	1 , 2 , 1 , 3	C_{23} ×	(5),(6),(7),deg	X_2=2 ⇒ $X_3 \neq 1$ (8)	
12	$C_{23}\ C_{14}$	$C_{12}\ C_{24}$	1 , 2 , 2 , 3	C_{23} √			
13	C_{14}	$C_{12}\ C_{24}\ C_{23}$	1 , 2 , 2 , 3	C_{14} √			
14		$C_{12}\ C_{24}\ C_{23}\ C_{14}$	1 , 2 , 2 , 3	NULL			

5.2.2 Motivation

EPDB has the advantage to be largely flexible compared to PDB: it exploits the flexibility of PDB to build optimal orders between variables, using VOHs, to converge to the termination with minimal checks of constraints.
The drawback of both EBDP and PDB is the verification of the current order between concerned variables before generating a nogood, in conflict or domain wipe-out situations. This verification of orders, which are saved in safety conditions and nogoods, is necessary to avoid looping, but it often consumes a significant time, since in most cases, orders are not directly deduced.
The next example explains the inconvenient of the orders verification process in both PDB and EPDB.

EPDB is used to repair a dynamic CSP. The current constraint to check is C_{12}, which is violated. A nogood concerning X_1 and X_2 must be generated, thus, the current order between X_1 and X_2 is verified.
The actual orders are shown in Tab.5.2:

Table 5.2 – Actual existing orders between variables

safety conditions	nogoods
$X_2 < X_i$ (1)	$X_j = v_j \Longrightarrow X_k \neq v_k$ (4)
$X_i < X_m$ (2)	$X_k = v_k \Longrightarrow X_l \neq v_l$ (5)
$X_j < X_k$ (3)	$X_k = v_k \Longrightarrow X_1 \neq v_1$ (6)
	$X_i = v_i \Longrightarrow X_j \neq v_j$ (7)

In fact, $X_2 < X_1$, since $X_2 < X_i$ (1), $X_i < X_j$ (7), $X_j < X_k$ (3) and $X_k < X_1$ (6). However, deducing the order between X_1 and X_2 is not directly done, as the variables involved in this order, namely X_2, X_i, X_j, X_k and X_1, are involved in the left-hand side of other nogoods and safety conditions which are stored before the concerned orders. X_i is involved in (2) which is placed before (7) and X_k in (5) which is before (6), then, (2) and (5) are also verified.
Moreover, determining order consumes an expensive time also when no order exists between variables in conflict. unfortunately, this must be proven by verifying all nogoods and safety conditions concerning them directly (orders where they appear) and indirectly (orders where their neighbours appear directly and indirectly).
As an example, suppose once again using EPDB to repair a DCSP, such as the current constraint to check C_{12} is violated. To generate a nogood, the current order between X_1 and X_2 is verified.
The actual orders are shown in Tab.5.3:

Table 5.3 – Actual existing orders between variables

safety conditions	nogoods
$X_2 < X_i$ (1)	$X_k = v_k \Longrightarrow X_l \neq v_l$ (4)
$X_i < X_m$ (2)	$X_1 = v_1 \Longrightarrow X_k \neq v_k$ (5)
$X_j < X_n$ (3)	

The approach checks (1) then (2) to test if X_2 precedes X_1, and (5) then (4) to test if X_1 precedes X_2, before concluding that no order exists between them, and making one using the specified VOH.

The goal of our proposed approach, the Improved Partial-order Dynamic Backtracking (IPDB), is to deduce orders directly while benefiting from the EPDB repairing strategy. Therefore, IPDB proceeds exactly like EPDB, but uses an additive data structure, called successors, which allows, like its name indicates, to have the information about all successors of each variable in the network.

Then, in the example of Tab.5.2, the structure successors contains the information indicated in Tab.5.4. We suppose that the order of storage of all of safety conditions and nogoods is the same as their index.

The first order is $X_2 < X_i$ (1), then, X_i and all its successors are added in successors of X_2. They are also added in successors of variables where X_2 is a successor. Therefore, this order causes the addition, in successors, of the variables following $\overset{(1)}{\rightarrow}$ till $\overset{(nbr)}{\rightarrow}$ as $nbr \neq 1$. the next added informations are respectively $X_i < X_m$ (2), $X_j < X_k$ (3) and $X_j = v_j \Longrightarrow X_k \neq v_k$ (4), which are added similarly to the first. Then, the information $X_k = v_k \Longrightarrow X_l \neq v_l$ (5) is added, X_l is added in successors of X_k. Moreover, it is added in successors of X_j which has X_k as a successor. As X_j has X_k tow times in successors, the list of successors of X_k is also added two times. This number of recurrence is important, in order to

not lose an order information about all successors, when a relation between two variables is deleted (i.e. a *safety condition*) and another existing relations (i.e. *nogoods* or *safety conditions*) imply the same successors. Next, the information $X_k = v_k \Longrightarrow X_1 \neq v_1$ (6) is added similarly to (5). Finally, since $X_i = v_i \Longrightarrow X_j \neq v_j$ (7), X_j and all its list of successors are added in successors of X_i. Moreover, they are added in successors of X_2, which has X_i as a successor.

Table 5.4 – The data structure 'successors' of IPDB

Variable	Successors
X_1	
X_2	$\overset{(1)}{\rightarrow} X_i \overset{(2)}{\rightarrow} X_m \overset{(7)}{\rightarrow} X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_i	$\overset{(2)}{\rightarrow} X_m \overset{(7)}{\rightarrow} X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_j	$\overset{(3)}{\rightarrow} X_k \overset{(4)}{\rightarrow} X_k \overset{(5)}{\rightarrow} X_l \rightarrow X_l \overset{(6)}{\rightarrow} X_1 \rightarrow X_1$
X_k	$\overset{(5)}{\rightarrow} X_l \overset{(6)}{\rightarrow} X_1$
X_l	
X_m	

Thus, to generate the nogood concerning X_1 and X_2, caused by the violation of C_{12}, the approach checks if X_2 is in $successors(X_1)$, which is not, then checks if X_1 exists in $successors(X_2)$, which is. Then, the generated nogood is the one with X_1 in conclusion.

Therefore, actual orders are as shown in Tab.5.5:

Table 5.5 – Actual existing orders between variables

safety conditions	nogoods
$X_2 < X_i$ (1)	$X_j = v_j \Longrightarrow X_k \neq v_k$ (4)
$X_i < X_m$ (2)	$X_k = v_k \Longrightarrow X_l \neq v_l$ (5)
$X_j < X_k$ (3)	$X_k = v_k \Longrightarrow X_1 \neq v_1$ (6)
	$X_i = v_i \Longrightarrow X_j \neq v_j$ (7)
	$X_2 = v_2 \Longrightarrow X_1 \neq v'_1$ (8)

and *successors* are as indicated in Tab.5.6:

Table 5.6 – The data structure 'successors' of IPDB

Variable	Successors
X_1	
X_2	$\rightarrow X_i \rightarrow X_m \rightarrow X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1 \overset{(8)}{\rightarrow} X_1$
X_i	$\rightarrow X_m \rightarrow X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_j	$\rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_k	$\rightarrow X_l \rightarrow X_1$
X_l	
X_m	

The other case where *successors* is manipulated is the removal of the order. This removal concerns the suppression of old orders (*safety conditions*). To study this case, we suppose we clean respectively all of the *safety conditions* information, therefore, updated *successors* are indicated in Tab.5.7.
The first removed order is $X_2 < X_i$ (1). This suppression causes the removal, from *successors*, of the

striped variables following $\overset{(1)}{\rightarrow}$ till $\overset{(nbr)}{\rightarrow}$ as $nbr \neq 1$. In fact, X_i and all its successors are deleted from successors of X_2. the next removed information is $X_i < X_m$ (2), which causes the suppression of X_m from successors of X_i. Finally, the last deleted information concerns $X_j < X_k$ (3). X_k and all its list of successors are removed from successors of X_j. Moreover, they are removed from successors of X_i, which contains X_j in successors.

Table 5.7 – The data structure 'successors' of IPDB

Variable	Successors
X_1	
X_2	$\overset{(1)}{\rightarrow} X_i \rightarrow X_m \rightarrow X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1 \rightarrow X_1$
X_i	$\overset{(2)}{\rightarrow} X_m \rightarrow X_j \overset{(3)}{\rightarrow} X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_j	$\overset{(3)}{\rightarrow} X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_k	$\rightarrow X_l \rightarrow X_1$
X_l	
X_m	

5.2.3 Description of the proposed data structure

As we previously said, IPDB is executed exactly like EPDB, with the difference of the way to test orders, which is done to generate nogoods. Thus, IPDB uses the additional structure of orders *successors*, and checks *successors* of the variables concerned by the nogood to test the existence of order (line 30 of Algorithm.9).
The data structure *successors* is manipulated in two phases:

— When a *nogood* is generated, in a conflict or a domain wipe-out situation, the procedure *AddRelation* is called to add the information concerning new orders established by this *nogood* (after instructions in lines 9 and 35 of Algorithm.9).

— When a *safety condition* is removed, the procedure *RemoveRelation* is called to remove information concerning orders caused by this *safety condition* (line 26 of Algorithm.9).

Algorithm 10: Manipulation of successors

```
Procedure AddRelation(X1,X2)
1.  for each Xi in DCSP.variables
2.    if successors[X2][Xi] > 0 then
3.      succeeds.add(Xi);
4.      count.add(successors[X2][Xi]);
5.  increment successors[X1][X2];
6.  for i from 0 to size of succeeds
7.    add count.get(i) in successors[X1][succeeds.get(i)];
8.  for each Xj in DCSP.variables
9.    if successors[Xj][X1] > 0 then
10.     add successors[Xj][X1] in successors[Xj][X2];
11.     for k from 0 to size of succeeds
12.       add count.get(k) × successors[Xj][X1] in successors[Xj][succeeds.get(k)];
Procedure RemoveRelation(X1,X2)
13. for each Xi in DCSP.variables
14.   if successors[X2][Xi] > 0 then
15.     succeeds.add(Xi);
16.     count.add(successors[X2][Xi]);
17. decrement successors[X1][X2];
18. for i from 0 to size of succeeds
19.   subtract count.get(i) from successors[X1][succeeds.get(i)];
20. for each Xj in DCSP.variables
21.   if successors[Xj][X1] > 0 then
22.     subtract successors[Xj][X1] from successors[Xj][X2];
23.     for k from 0 to size of succeeds
24.       subtract count.get(k) × successors[Xj][X1] from successors[Xj][succeeds.get(k)];
```

The used structure *successors* is a matrix successors[n][n], since n is the number of variables of the DCSP. successors[i][j] stores the information about how many times the variables of the index j appears in *successors* of the variables with the index i.
When a nogood is added between X_1 and X_2, the procedure AddRelation (Algorithm. 10) is called. It stores the information about successors of X_2 (line 3) and how many times they appear in its *successors* (line 4). Then, X_2 is added in *successors* of X_1 (line 5). Moreover, successors of X_2 are added, many times they appear in *successors* of X_2, in *successors* of X_1 (lines 6 and 7). X_2 and their successors are also added in *successors* of variables X_j, where X_1 is a successor. Therefore, X_2 is added in *successors* of X_j many times X_1 appears in *successors* of X_j (line 10), and its successors are added, for each appearance of X_1 in *successors* of X_j, many times they appear in *successors* of X_2 (lines 11 and 12).
When a *safety condition* between X_1 and X_2 is deleted, the procedure RemoveRelation (Algorithm.10) is called. This procedure proceeds exactly like AddRelation, by removing variables from successors, many times they are added in AddRelation.

5.2.4 Termination proof

It is proven in [Bliek, 1998] that PDB terminates. In fact, PDB respects the relative ordering conditions corresponding to the ordered nogoods. Moreover, it imposes *safety conditions*, which are relative ordering conditions, to retain some of the ordering conditions of the nogoods that have been removed. The order selected for new nogoods is required to respect both types of ordering conditions. Thus, the resulting set of conditions could not be cyclic and PDB terminates.
EPDB proceeds in the same manner as PDB. The difference is its use of VOHs to generate a nogood. VOHs are used only if no orders, in both types of ordering conditions, link the variables concerned by this nogood. Otherwise, it proceeds like PDB. Therefore, EPDB terminates.
Improved PDB behaves exactly like EPDB, the difference concerns the way to determine orders. In fact, orders are determined from the data structure *successors*, which reflects exactly all of the orders saved in nogoods and *safety conditions*. Hence, IPDB also terminates.

5.2.5 Experimental study

In this section, we evaluate IPDB compared to EPDB, to show the improvement of our proposed approach compared to EPDB. We also compare it to MAC-2001 [Bessière and Régin, 2001b, Bessière et al., 2005], to identify the maximal rate of perturbation that tolerates repairing, i.e. from which rate of disturbance the use of a good resolution algorithm is more efficient to deal DCSPs.
Surely, some integrations of MAC in intelligent backtrackers were already proposed [Jussien et al., 2000, Prosser, 1995], but we choose MAC-2001 as we judge it to be one of the most efficient resolving CSPs strategies while being the most familiar constraint propagation one.
We integrated various VOHs both of EPDB and IPDB, namely deg, dom, conf, ddeg, wdeg, dom/deg, dom/ddeg,dom/wdeg and lex. But as ddeg, wdeg, dom/ddeg,dom/wdeg and lex have a modest behaviour, we show only results concerning the other VOHs. MAC-2001 is combined to the dom/deg VOH like in [Mehta and van Dongen, 2005]. We note that the arc consistency strategy can also be improved by using dom/wdeg [Boussemart et al., 2004b], but as shown in [Mehta and Van Dongen, 2007], this heuristic is efficient essentially in combination with $AC-3$.
As evaluation criteria, we measure the solution time (time) by milliseconds and the number of constraints checks (CCs).
Experiments are performed on the Java platform. They concerns a Pentium IV PC (2.8 GHz processor and 3.24 Go RAM).

Experiments on randoms

Experiments concerning repairing. The following experiments concern uniform random binary DCSPs section 2.2.1, tests are performed on sparse and dense problems, respectively $<20,10,0.25,0.60>$ and $<20,10,0.75,0.25>$. Parameters are selected in such a way that the problem instances are the satisfiable ones located at the phase transition [Crawford and Auton, 1996]. Problems are relatively harder to solve in the phase transition region, the region between an under-constrained region where all instances are almost surely satisfiable and an over-constrained region where all instances are almost surely unsatisfiable.
For each pair of fixed density and constraints tightness $<p_1,p_2>$, 30 different instances are treated by each algorithm and the present results are the averages of these 30 runs.
The rate of injected constraints %IC = $\frac{|new\ constraints|}{|whole\ problem\ constraints|}$. It indicates the rate of disturbance that will be treated using both IPDB and EPDB. IC varies from 1% to 100%, such as for each given disturbance of d%, the added constraints are the same as those in the disturbance of $d-1$%, plus 1% more constraints. The 30 final problems (after disturbance) are the same for each rate of disturbance.
For DCSP approaches, namely IPDB and EPDB, the computed effort is the one for repairing the past solution, i.e. which existed before disturbing.

Concerning the number of CCs (1^{st} part of Tabs.5.8 and 5.9), using IPDB with a VOH is exactly the same as using EPDB with the same VOH, as the two approaches proceed in the same manner. But regarding the execution time (2^{d} part of Tabs.5.8 and 5.9), IPDB remarkably exceeds EPDB, regardless of the used VOH.
The use of IPDB to repair problems that are subject to a small disturbance is more effective than its use to repair high perturbed problems. However, from a certain disturbance, more or less interesting depending on the used VOH, the effort used by IPDB to repair problems is almost the same. This effort is not stable, but it is subject to the same sudden falls.
For low density (Tab.5.8), $IPDB_{deg}$ outperforms MAC-2001 respecting the number of CCs and execution time. $IPDB_{dom}$ is also a good concurrent of MAC-2001, especially concerning execution time, till an average disturbance. The integration of other VOHs in IPDB is beneficial only in small perturbation.
For high density (Tab.5.9), $IPDB_{deg}$ often outperforms MAC-2001, not as well as in low density, respecting the execution time. The use of other VOHs, excepting dom in some situations, is only effective in small disturbances.

Table 5.8 – Number of CCs and Execution time (p_1= 0.25, p_2 = 0.60)

%IC	MAC-2001	IPDB$_{conf}$	IPDB$_{deg}$	IPDB$_{dom}$	IPDB$_{dom/deg}$	EPDB$_{conf}$	EPDB$_{deg}$	EPDB$_{dom}$	EPDB$_{dom/deg}$	
1		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	
4		467	315	412	**172**	467	315	412	**172**	
7		3327	2862	**1094**	2945	3327	2862	**1094**	2945	
10		4685	**3430**	4012	3742	4685	**3430**	4012	3742	
13		5933	3581	**2901**	5420	5933	3581	**2901**	5420	
16		5812	**1783**	2670	4437	5812	**1783**	2670	4437	
19		4691	**1967**	3084	5485	4691	**1967**	3084	5485	
22		7824	**3257**	3836	6028	7824	**3257**	3836	6028	
25		8748	**2959**	4465	6184	8748	**2959**	4465	6184	
31		7226	**3559**	6841	14209	7226	**3559**	6841	14209	
37		11823	**3072**	6198	6452	11823	**3072**	6198	6452	
43	4992	11670	**3097**	2723	9433	11670	**3097**	2723	9433	**CCs**
49		10958	**3213**	6708	9693	10958	**3213**	6708	9693	
55		17304	**4720**	5771	13064	17304	**4720**	5771	13064	
61		16519	**4209**	8068	17922	16519	**4209**	8068	17922	
67		13945	**2930**	7914	10276	13945	**2930**	7914	10276	
73		10243	**3038**	6012	12592	10243	**3038**	6012	12592	
79		13867	**4063**	6770	10650	13867	**4063**	6770	10650	
85		5969	**3170**	9630	10926	5969	**3170**	9630	10926	
91		8619	**3333**	9722	10846	8619	**3333**	9722	10846	
97		7756	**3729**	4605	7996	7756	**3729**	4605	7996	
100		10557	**3516**	7660	7660	10557	**3516**	7660	7660	
1		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	
4		1	1	1	**0**	40	19	34	10	
7		10	6	**3**	7	265	165	51	193	
10		12	**7**	9	9	361	190	239	256	
13		16	8	**7**	13	479	200	197	377	
16		16	**4**	6	11	491	83	182	292	
19		12	**4**	8	13	375	94	235	368	
22		20	**7**	9	15	638	158	226	407	
25		22	**6**	11	14	712	142	282	387	
31		19	**7**	16	34	591	160	421	1029	
37		32	**6**	15	15	941	149	446	429	
43	18	31	7	**6**	21	926	143	190	544	**time**
49		29	**7**	15	23	889	148	381	644	
55		46	**10**	14	30	1321	213	388	904	
61		42	**9**	18	42	1218	187	467	1235	
67		37	**6**	18	25	1139	134	499	705	
73		27	**7**	14	30	789	129	400	871	
79		35	**9**	15	25	1040	174	410	781	
85		16	**7**	22	27	428	134	615	834	
91		22	**7**	22	25	630	146	632	705	
97		20	**8**	11	20	578	166	255	558	
100		28	**7**	17	17	815	156	482	483	

concerning solving. The previous experiments are focused on satisfiable problems in hard areas of low and high densities. These experiments show that IPDB using certain VOHs, specially deg, is a good concurrent of MAC-2001 although for solving (100% of IC).
In order to show where MAC-2001 is more efficient than IPDB, we execute the two approaches on random problems, not necessary satisfiable, with the same previous parameters n, d and p_1. Constraints tightness' vary from $p_2 = 0.1$ to $p_2 = 0.9$. 30 different instances are treated by each algorithm and the present results are the averages of these 30 runs.

Figs. 5.8 and 5.9 show that IPDB, regardless of the used VOH is often more efficient than MAC-2001 to treat satisfiable problems, i.e. $< p_1 = 0.25, p_2 \in [0.1, 0.60[>$ and $< p_1 = 0.75, p_2 \in [0.1, 0.25[>$, while MAC-2001 is effective to detect the non satisfiability of problems in $< p_1 = 0.25, p_2 \in]0.60, 0.9] >$ and $< p_1 = 0.75, p_2 \in]0.25, 0.9] >$. In complex areas, i.e. the transition phases $< p_1 = 0.25, p_2 = 0.60 >$ and $< p_1 = 0.75, p_2 = 0.25 >$, the majority of problems are non satisfiable, therefore, MAC-2001 is generally better than IPDB.
Generally, the use of deg in IPDB is more efficient than using another VOH.

Table 5.9 – Number of CCs and Execution time by ms (p_1 = 0.75, p_2 = 0.25)

%IC	MAC-2001	$IPDB_{conf}$	$IPDB_{deg}$	$IPDB_{dom}$	$IPDB_{dom/deg}$	$EPDB_{conf}$	$EPDB_{deg}$	$EPDB_{dom}$	$EPDB_{dom/deg}$	
1		**30525**	34660	22967	60452	**30525**	34660	22967	60452	
4		161656	**58131**	125013	330288	161656	**58131**	125013	330288	
7	102735	200448	**97458**	155499	287374	200448	**97458**	155499	287374	
10		211187	**81009**	126958	365003	211187	**81009**	126958	365003	
13		281738	**94322**	231430	475341	281738	**94322**	231430	475341	
16		367999	**93922**	184632	344709	367999	**93922**	184632	344709	
19		321162	116117	149307	386289	321162	116117	149307	386289	
22	**102735**	319757	125375	136200	544936	319757	125375	136200	544936	
25		249426	149562	135713	609473	249426	149562	135713	609473	
31		206415	130851	145220	229720	206415	130851	145220	229720	
37	102735	233581	132643	**92070**	201672	233581	132643	**92070**	201672	**CCs**
43		205376	149055	**99331**	289489	205376	149055	**99331**	289489	
49		357861	176114	191366	389047	357861	176114	191366	389047	
55		304489	179580	151199	387586	304489	179580	151199	387586	
61		263191	113543	200164	220405	263191	113543	200164	220405	
67		317448	131662	139075	198916	317448	131662	139075	198916	
73	**102735**	403323	148471	140780	191992	403323	148471	140780	191992	
79		248703	122163	211371	268997	248703	122163	211371	268997	
85		281928	121460	181720	225720	281928	121460	181720	225720	
91		243172	116682	205754	347517	243172	116682	205754	347517	
97		272052	125150	203347	170534	272052	125150	203347	170534	
100		255595	121030	181174	181174	255595	121030	181174	181174	
1		128	125	**80**	227	7293	5619	5235	13824	
4		688	**214**	445	1318	40445	10144	36637	99398	
7		838	**358**	550	1155	48605	16864	47408	81283	
10		883	**301**	454	1459	48950	13773	37818	96400	
13		1187	**349**	826	1894	68752	16589	71411	128730	
16	557	1552	**349**	666	1379	91197	16557	58646	91318	
19		1350	**429**	528	1533	77133	20310	45405	99613	
22		1341	**461**	493	2192	77261	21648	41139	142176	
25		1043	557	**483**	2403	57805	26147	38371	153990	
31		867	**483**	515	899	50371	22474	39619	56829	
37		979	493	**326**	781	54346	22957	25488	49648	**time**
43		858	554	**349**	1117	49357	26192	27014	66218	
49	**557**	1504	653	685	1520	90829	31862	56435	95264	
55		1270	669	**534**	1507	73267	32499	41078	93997	
61		1114	**416**	714	832	65328	18931	53318	50275	
67		1322	**486**	493	755	76474	22707	36982	46054	
73		1690	550	**502**	714	101622	25626	40083	42874	
79	557	1040	**448**	752	998	61322	20771	60035	57760	
85		1187	**448**	653	819	69974	20454	52992	48269	
91		1018	**426**	736	1251	58704	19488	63571	85213	
97		1136	**461**	726	605	64259	21338	58509	42483	
100		1139	**458**	653	659	100781	37693	80829	81168	

Experiments on Dynamic Meeting Scheduling Problems

A Dynamic Meeting Scheduling Problem (DMSP) is a sequence of static Meeting Scheduling Problems (MSPs): $MSP_0, MSP_1, ..., MSP_{\alpha-1}, MSP_\alpha, ...$, where the network is subject to constraints perturbations, and each MSP_i differs from the latest one MSP_{i-1} by the addition of some constraints.

We present our results for the MSP class $< 15, 25, 4, 5, 10, 1, 80 >$ and vary the rate of disturbance from ~1% to ~100%. We generated 30 different instances treated by each algorithm and the results are averages of these 30 runs. In the following experiments, the comparison is only between MAC-2001 and IPDB using deg (the best used VOH in repairing, as shown in previous experiments, and as proven [Acodad et al., 2012]).

Concerning Dynamic MSPs (Fig.5.10), $IPDB_{deg}$ is more efficient than MAC-2001 regardless of the rate of disturbance, specially respecting the number of CCs.
The use of $IPDB_{deg}$ to repair problems that are subject to a small disturbance is sufficiently efficient. However, from a given small disturbance (~8%), the effort used by $IPDB_{deg}$ to repair problems, which

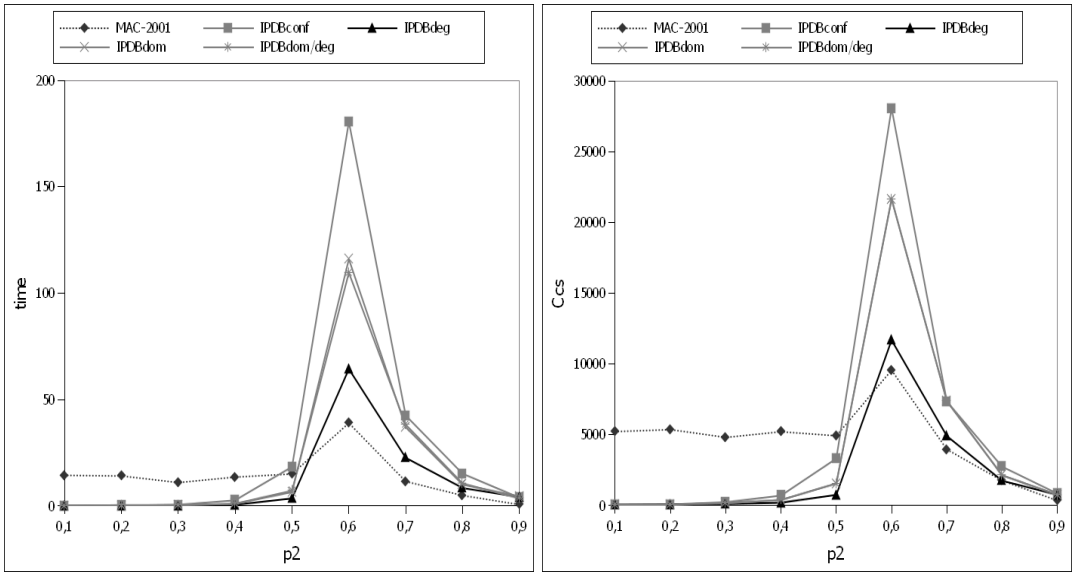

Figure 5.8 – Execution time and Number of CCs for $p_1 = 0.25$

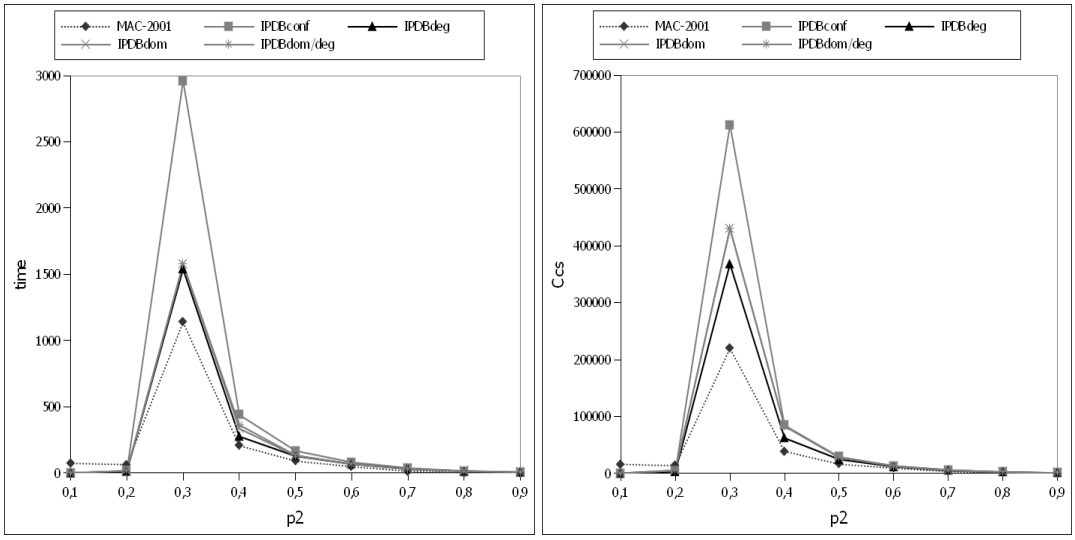

Figure 5.9 – Execution time and Number of CCs for $p_1 = 0.75$

is not stable, is almost the same.

5.2.6 Discussion

Discussion concerning repairing. As shown in experiments on satisfiable problems in transition phases (Tabs.5.8 and 5.9), IPDB is widely more powerful than EPDB in terms of runtime. This is particularly due to the way IPDB deduces the order between variables in conflict. It uses the additional data structure, *successors*, to deduce orders directly. But regarding CCs, the use of IPDB with a VOH

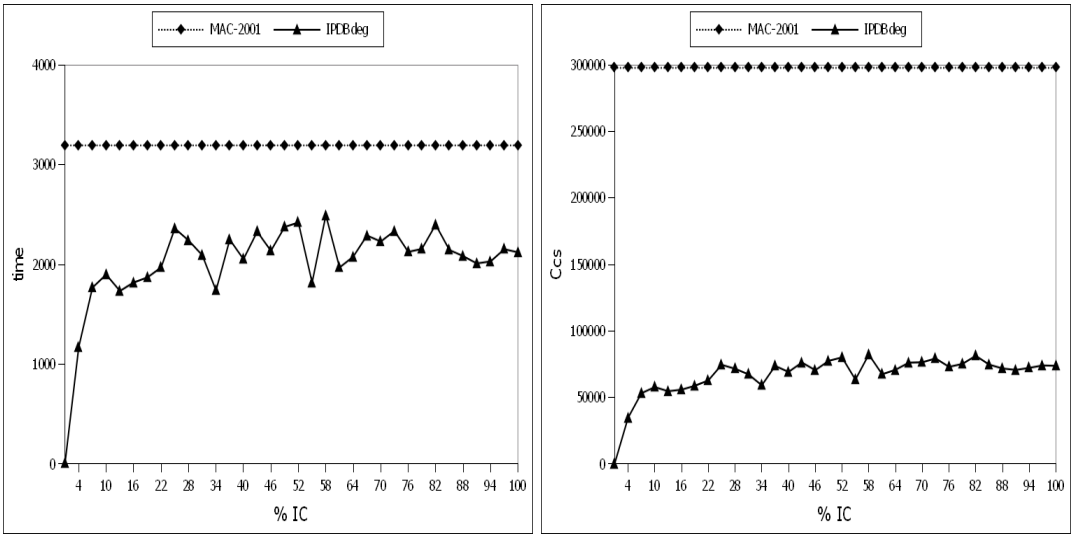

Figure 5.10 – Execution time and Number of CCs for MSPs

is the same as using EPDB with the same VOH. In fact, IPDB and EPDB proceed in the same manner. The only difference is the deduction of orders, in EPDB, from *nogoods* and *safety conditions*, while IPDB deduces orders from *successors*, which collects the information of existing orders in *nogoods* and *safety conditions*. Thus, the two approaches generate the same number of CCs.
From a certain small disturbance, more or less interesting depending on the used VOH, IPDB requires almost the same effort to repair, whatever the rate of injected constraints. This is due to the areas of chosen problems, which are the most complex areas (the transition phases). This effort is not stable, but it is subject to same sudden falls. The repairing effort depends on the assignment before disturbance (the old solution) and its consistency with the new constraints. Then, disturbing a problem with some constraints can be easier or harder than disturbing another problem with more constraints, depending on which assignment is satisfiable by more new constraints.
When problems are hard and satisfiable, IPDB_{deg} is a good concurrent of MAC-2001, even for solving (100% of IC). Nevertheless, when another VOH is used in IPDB (2_d part of Tabs.5.8 and 5.9), IPDB becomes less efficient comparing to MAC-2001. In fact, the goal of *deg* in repairing is to keep the most connected sub-problems (variables with a large neighborhood) intact as long as possible, to minimize risks of affecting neighbors, in case of domains wipe-out.

Discussion concerning solving. Concerning problems where constraints tightness is less than the peak (under the transition phase), although injecting the totality of constraints in perturbation, i.e. resolving (Figs.5.8 and 5.9), IPDB is more efficient than MAC-2001, regardless of the used VOH and problems density. In these areas, most of the problems are satisfiable, thus, the use of arc consistency is not efficient enough. The major interest of MAC-2001 is to detect earlier the future domains wipe-outs, in order to modify the partial assignments which can not be a part of the solution. However, in sparse graphs, domains dead-ends are rarely met.
Therefore, for unsatisfying problems, where constraints tightness is higher than the peak, MAC-2001 is more effective than IPDB.
On peaks, the majority of problems are unsatisfiable, then, MAC-2001 is globally more efficient than IPDB.
Like for experiments concerning repairing, when *deg* is used, IPDB has better behavior, especially for

problems with low density. In high density, the network is very connected, therefore, variables have close values of *deg*.

5.2.7 Conclusion

In this section, we introduce IPDB, which improves EPDB, to repair solutions of Dynamic CSPs rapidly, using the data structure *successors*.
Based on the results of experiments on a variety of problems classes, we show that IPDB is a great improvement of EPDB, respecting runtime, insofar as it minimizes remarkably the tests of orders.
We show that this approach is mainly efficient when using *deg* and that it is a good concurrent for MAC-2001. In fact, IPDB outperforms MAC-2001 when problems are satisfiable, although using it to solve from the scratch. Thus, IPDB has the advantage to be also destined for CSPs.
We demonstrate that IPDB is more effective to treat satisfiable problems while MAC-2001 is more efficient to detect the non-satisfiability of problems. Therefore, for complicated problems where the satisfiability is unknown, the two approaches can be separately launched and the fast resolution is then guaranteed.

Table 5.10 – Actual existing orders between variables

safety conditions:	nogoods:
$X_2 < X_i$ (1)	$X_k = v_k \Longrightarrow X_l \neq v_l$ (4)
$X_i < X_m$ (2)	$X_k = v_k \Longrightarrow X_1 \neq v_1$ (5)
$X_j < X_k$ (3)	$X_i = v_i \Longrightarrow X_j \neq v_j$ (6)

Table 5.11 – Actual existing orders between variables

safety conditions:	nogoods:
$X_2 < X_i$ (1)	$X_k = v_k \Longrightarrow X_l \neq v_l$ (4)
$X_i < X_m$ (2)	$X_1 = v_1 \Longrightarrow X_k \neq v_k$ (5)
$X_j < X_n$ (3)	

5.3 LRB

In this section, we propose a fast version of EPDB, the Lazy Repairing Backtracking (LRB) approach. This algorithm aims to repair solutions using EPDB advantages while eliminating unnecessary tests of orders.

5.3.1 Motivation

EPDB has the advantage to be largely flexible compared to PDB: it exploits the flexibility of PDB to build optimal orders between variables, using VOHs, to converge to the termination with minimal checks of constraints.
The drawback of both EBDP and PDB is the verification of the current order between concerned variables before generating a nogood, in conflict or domain wipe-out situations. This verification of orders, which are saved in safety conditions and nogoods, is necessary to avoid looping, but it often consumes a significant time, since in most cases, orders are not directly deduced.
To more clarify, suppose the next example:
EPDB is used to repair a dynamic CSP. The current constraint to check is C_{12}, which is violated. A nogood concerning X_1 and X_2 must be generated, thus, the current order between X_1 and X_2 is verified.

The actual orders are shown in the Tab.5.10:

In fact, $X_2 < X_1$, since $X_2 < X_i$ (1), $X_i < X_j$ (6), $X_j < X_k$ (3) and $X_k < X_1$ (5). But deducing the order between X_1 and X_2 is not directly done, as the variables involved in this order, namely X_2, X_i, X_j, X_k and X_1, are involved in other nogoods and safety conditions which are stored before the concerned orders. X_i is involved in (2) which is placed before (6) and X_k in (4) which is before (5), then, (2) and (4) are also verified.
Moreover, determining order takes an expensive time also when no order exists between variables in conflict. unfortunately, this must be proven by verifying all nogoods and safety conditions concerning them directly (orders where they appear) and indirectly (orders where their neighbours appear directly and indirectly).
As an example, suppose once again using EPDB to repair a DCSP, such as the currant constraint to check C_{12} is violated. To generate a nogood, the current order between X_1 and X_2 is verified.
The actual orders are shown in Tab.5.11:

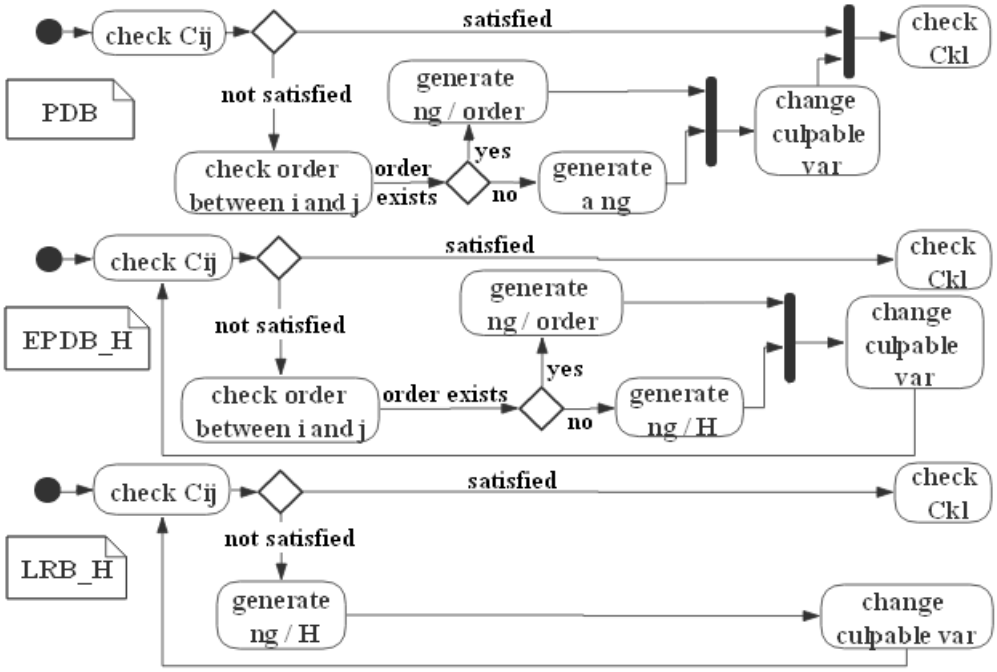

Figure 5.11 – behaviour of PDB, EPDB and LRB

The approach checks (1) then (2) to test if X_2 precedes X_1, and (5) then (4) to test if X_1 precedes X_2, before concluding that no order exists between them, and establishing one using the specified VOH. Therefore, in Lazy Repairing Backtracking (LRB), orders are deduced directly using a VOH. Thus, while safety conditions in EPDB are only used to store orders expressed by the removed nogoods, and nogoods are also used to save justifications of prohibitions (to avoid choosing a value which will cause an already met conflict and to restore values when causes are no longer valid), safety conditions are rejected and only nogoods are kept.
The only issue that can exist, when establishing the order directly using a VOH, i.e. without verifying the existence of order between concerning variables, is the termination when the VOHs values change. To remedy this problem, only static VOHs are used, to have a fixed order between variables during the search process, especially since the static VOHs, deg, and pdeg, have the best behavior in repairing [Acodad et al., 2014, 2012].

5.3.2 Behavior of LRB compared to PDB and EPDB

As shown in Fig.5.11, after checking a constraint C_{ij}, satisfied or not, PDB checks another one C_{kl}. In fact, if C_{ij} is violated, PDB tests if an order already exists between the concerned variables i and j. Next, it builds a nogood ng, respecting the order, if there exist. If no order exists between i and j, it builds any nogood ng which will impose a new order between i and j. Then, the conclusion of ng changes the value and another constraint C_{kl} is checked.
After checking a constraint C_{ij}, EPDB checks another one C_{kl} only if C_{ij} is satisfied. If C_{ij} is violated, it tests if an order already exists between the concerned variables i and j. Next, it builds a nogood ng, respecting the order if there exists. If no order exists between i and j, the nogood respects the chosen heuristic H. Then, the conclusion of ng changes the value, and C_{ij} is re-checked.
LRB, like EPDB, after checking a constraint C_{ij}, checks another one C_{kl} only if C_{ij} is satisfied. Other-

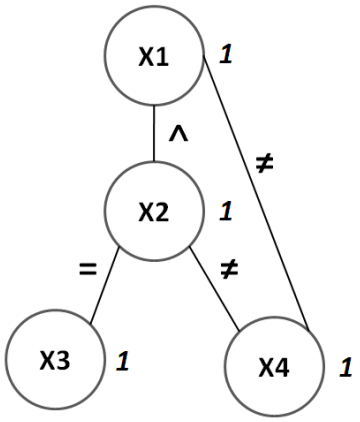

Figure 5.12 – A constraint network example

wise, LRB builds a nogood ng, respecting the chosen heuristic H. Then, the conclusion of ng changes the value, and C_{ij} is checked again.

5.3.3 An example run

Let's consider the CSP illustrated in Fig.5.12:

— X = {X1, X2, X3, X4}.

— D = {D_1, D_2, D_3, D_4} such as $D_i = \{1,2,3\}\ \forall i \in [1,4]$.

— C = {$X1 < X2$, $X1 \neq X4$, $X2 = X3$, $X2 \neq X4$ }.

All variables are assigned by the value 1. For simplicity, we consider that this first assignment (before perturbation) was found using a simple resolution approach (FC for example), then, at the beginning of repairing, no nogood or safety condition is saved in the last search. Thus, no order is imposed between variables.

Table 5.12 describes EPDB using the deg heuristic. The approach calculates, before starting the search, deg of each variable in the network: $deg(X1) = 2$, $deg(X2) = 3$, $deg(X3) = 1$ and $deg(X4) = 2$.
All variables have the same assigned value 1. This assignment violates more than one constraint. A constraint ordering heuristic [Hammoujan and Acodad, 2013] can be used to improve the search, but, to treat different cases, we assume that the constraints are ordered as shown in *to check* column.
The first checked constraint *checked* is the one between X1 and X2 (line 1). This constraint is violated, and as no order exists yet between X1 and X2 (*add* column is empty), the heuristic value of X1 and X2 is verified to determine the order. $deg(X1) < deg(X2)$, thus, the generated nogood is (1). X1 changes the value and the constraint is re-checked before selecting another one (line 2). C_{12} is not consistent, then, the existence of order between X1 and X2 is tested before generating the nogood. (1) is tested then (2) is generated. X1 changes the value and the constraint, which is still not consistent, is re-checked (line 3). To generate the nogood, the existence of order between X1 and X2 is tested. Then, (1) is tested and (3) is generated. X1 has no more values in its domain, then, EPDB resolves (1), (2) and (3) as (4), removes the three no valid nogoods (line 4), saves their expressed order as a safety condition (5) and changes the value of X2, the unique cause of the domain wipeout of X1 (line 5). X1 is assigned to its first valid value, and the consistent constraint is rechecked (line 6). As C_{12} is satisfied, it's moved to the list of free constraints *free cstrs*, which contains all tested and satisfied constraints, and the next constraint to verify is checked. C_{14} is violated (line 7). The existence of order between X1 and X4 is tested, but no order in *add* contains neither X1 nor X4. Then, the heuristic value is

Table 5.12 – Execution of EPDB_{deg}

	to check	free cstrs	X_1 X_2 X_3 X_4	checked	test	add	remove
1	C_{12} C_{14} C_{24} C_{23}		1 , 1 , 1 , 1	C_{12} ×	deg	X_2=1 ⇒ $X_1 \neq 1$ (1)	
2	C_{12} C_{14} C_{24} C_{23}		2 , 1 , 1 , 1	C_{12} ×	(1)	X_2=1 ⇒ $X_1 \neq 2$ (2)	
3	C_{12} C_{14} C_{24} C_{23}		3 , 1 , 1 , 1	C_{12} ×	(1)	X_2=1 ⇒ $X_1 \neq 3$ (3)	
4	C_{12} C_{14} C_{24} C_{23}		- , 1 , 1 , 1			$X_2 \neq 1$ (4)	(1),(2),(3)
5	C_{12} C_{14} C_{24} C_{23}		- , 2 , 1 , 1			$X_2 < X_1$ (5)	
6	C_{12} C_{14} C_{24} C_{23}		1 , 2 , 1 , 1	C_{12} √			
7	C_{14} C_{24} C_{23}	C_{12}	1 , 2 , 1 , 1	C_{14} ×	deg	X_1=1 ⇒ $X_4 \neq 1$ (6)	
8	C_{14} C_{24} C_{23}	C_{12}	1 , 2 , 1 , 2	C_{14} √			
9	C_{24} C_{23}	C_{12} C_{14}	1 , 2 , 1 , 2	C_{24} ×	(5),(6)	X_2=2 ⇒ $X_4 \neq 2$ (7)	
10	C_{24} C_{23} C_{14}	C_{12}	1 , 2 , 1 , 3	C_{24} √			
11	C_{23} C_{14}	C_{12} C_{24}	1 , 2 , 1 , 3	C_{23} ×	(5),(6),(7),deg	X_2=2 ⇒ $X_3 \neq 1$ (8)	
12	C_{23} C_{14}	C_{12} C_{24}	1 , 2 , 2 , 3	C_{23} √			
13	C_{14}	C_{12} C_{24} C_{23}	1 , 2 , 2 , 3	C_{14} √			
14		C_{12} C_{24} C_{23} C_{14}	1 , 2 , 2 , 3	NULL			

verified. As $deg(X1) = deg(X4)$, any order is chosen, we assume that in similar cases, the chosen order is the lexicographic one. Thus, the nogood is generated as (6). X4 changes the value, then, C_{14} is consistent (line 8) and it's moved to *free cstrs*. The next constraint to check is C_{24} which is violated (line 9), (5) and (6) are tested, then, (7) is generated, the value of X4 is changed (line 10) and C_{14}, the only constraint in *free cstrs* concerning X4, is moved to *to check* list. The next checked constraint is C_{23} which is violated (line 11), the tested orders are (5) then (6), and (7). As no order exists between the two variables, the chosen heuristic is used to generate (8), then, the value of X3 is changed (line 12). C_{23} is consistent, thus, it's moved to *free cstrs*. C_{14} is then checked (line 13), it's satisfied so it's moved to *free cstrs*. *to check* is empty (line 14), thus, all constraints are satisfied and the solution is found.

Similarly to EPDB, when LRB is executed using the same heuristic *deg*, the approach calculates, before starting the search, *deg* of all network variables.
It proceeds like EPDB with the following differences:

— to generate a nogood, LRB tests directly the chosen VOH values. Then, in *test* column, each constraint violation (nogood to generate) causes only one test, which is the test of *deg*, contrary to EPDB (*test* column of Tab. 5.12).

— LRB doesn't need the information about order saved in *safety conditions* since the order is determined by the used VOH. Then, when a nogood is removed, no safety condition is retained (in Tab.5.12, after removing nogoods in *remove* column of line 4, *add* column of line 5 will be empty).

5.3.4 Description of the proposed approach

To reuse solution and past reasoning techniques, LRB starts the search with a DCSP that contains the past solution, keeping retroactive data structures of nogoods *ngList* saved during the past search process.
LRB proceeds exactly like EPDB, with the following major differences:

— to generate a nogood when a conflict is detected, LRB doesn't verify the order between concerned variables as EPDB does. Then, in line 9 of Fig.9, the chosen VOH is directly used to establish the order.

— similarly, to generate the resolution nogood when a domain wipe-out is produced, LRB doesn't verify the order between the responsible variables as EPDB does. Thus, in line 35 of Fig.9, the chosen VOH is directly used to generate the nogood.

— As no order verification will be required, LRB doesn't use the *safety condition* list to store orders expressed by the removed nogoods, then, lines 2 and 26 in Fig.9 are removed.

5.3.5 Termination proof

It is proven in [Bliek, 1998] that PDB terminates. In fact, PDB respects the relative ordering conditions corresponding to the ordered nogoods. Moreover, it imposes *safety conditions*, which are relative ordering conditions, to retain some of the ordering conditions of the nogoods that have been removed. The order selected for new nogoods is required to respect both types of ordering conditions. Thus, the resulting set of conditions could not be cyclic.
LRB proceeds in the same manner of PDB, but drops *safety conditions*. To guarantee termination, it uses only static VOHs and is based on the used SVOH for the selected order.

5.3.6 Experimental study

Experiments concerning repairing

In this section, we compare LRB to EPDB, to show the improvement of the new approach. We also compare it to MAC-2001 [Bessière and Régin, 2001b, Bessière et al., 2005], to identify the maximal rate of perturbation that tolerates repairing, i.e. from which rate of disturbance the use of a good resolution algorithm is more efficient to deal DCSPs.
Surely, some integrations of MAC in intelligent backtrackers were already proposed [Jussien et al., 2000, Prosser, 1995], but we choose MAC-2001 as we judge it to be one of the most efficient resolving CSPs strategies while being the most familiar constraint propagation one.
Heuristics used in LRB and EPDB are $pdeg$ and deg. To prove that performance of LRB is due essentially to a well-chosen heuristic, we use also the lexicographic order heuristic, which is used instead of the arbitrary order, as LRB supports only the static ordering to avoid looping. MAC-2001 is combined to the dom/deg heuristic like in [Mehta and van Dongen, 2005]. We note that the arc consistency strategy can be also improved by using $dom/wdeg$ [Boussemart et al., 2004b], but as shown in [Mehta and Van Dongen, 2007], this heuristic is efficient essentially in combination with $AC-3$.
As evaluation criteria, we measure the runtime and the number of Constraints Checks (CCs).

Experiments on uniform random binary DCSP**.** Tests are performed on sparse and dense random problems section 2.2.1, respectively $< 30,10,0.25,0.45 >$ and $< 30,10,0.75,0.19 >$. The chosen problems are the satisfiable ones at the transition phase [Crawford and Auton, 1996] (the most complicated area).
For each pair of fixed density and constraints tightness $< p_1, p_2 >$, 30 different instances are treated by each algorithm and the present results are the averages of these 30 runs.
The rate of injected constraints $\%IC = \frac{|new\ constraints|}{|whole\ problem\ constraints|}$. It indicates the rate of disturbance that will be treated using both LRB and EPDB. IC varies from 1% to 100%, such as for each given disturbance of d%, the added constraints are the same as those in the disturbance of $d-1\%$, plus 1% more constraints. The 30 final problems (after disturbance) are the same for each rate of disturbance. For DCSP approaches, namely LRB and EPDB, the computed effort is the one for repairing the past solution, i.e. which existed before disturbing.
As p_2 is known, we tried to replace, in the $pdeg$ calculation, the parameter 1/2 by p_2. But results were not significantly better than those of using 1/2, then, we choose to keep the parameter 1/2 for all kinds of problems. We fix the time-out at 7 minutes. Experiments are performed on the Java platform, on a core i7 PC (3.6 GHz processor and 16 Go RAM).

Concerning the number of CCs (2^d part of Tabs.5.13 and 5.14), using LRB with a VOH is almost the same as using EPDB with the same VOH. But regarding the execution time (1^{st} part of Tabs.5.13 and 5.14), LRB exceeds EPDB very remarkably, regardless of the used VOH.
Excepting the small disturbances, MAC-2001 often produces less CCs than LRB (2^d part of Tabs.5.13 and 5.14), mainly when LRB uses the deg heuristic.
From a certain small disturbance (13% for sparse problems and 10% for those dense), LRB needs almost the same effort to repair (Tabs.5.13 and 5.14), whatever the rate of injected constraints IC. This effort is not stable, regardless of the used heuristic, but it is subject to same sudden falls.

Table 5.13 – Execution time by seconds and Number of CCs (p_1= 0.25, p_2 = 0.45)

% IC	MAC-2001	LRB_{pdeg}	LRB_{deg}	LRB_{lex}	$EPDB_{pdeg}$	$EPDB_{deg}$	
1		**0.006**	0.008	1.471	0.508	0.723	
4		**0.024**	0.026	1.835	2.242	2.662	
7		**0.048**	0.065	2.047	4.842	6.254	
10		**0.059**	0.073	2.304	5.788	7.047	
13		**0.079**	0.089	2.650	7.517	8.373	
16		**0.082**	0.093	2.860	8.081	9.338	
19		**0.082**	0.097	2.809	8.099	9.243	
22		**0.085**	0.094	2.876	8.214	9.479	
25		**0.080**	0.095	2.878	7.759	9.479	
28		**0.080**	0.099	2.860	8.020	9.467	
31		**0.078**	0.101	2.891	7.263	8.093	
34	0.094	**0.075**	0.102	2.881	7.330	8.402	**time(sc)**
40		**0.062**	0.075	2.869	8.321	8.069	
46		**0.069**	0.078	2.893	9.521	10.455	
52		**0.080**	0.093	2.894	8.919	10.799	
58		**0.065**	0.076	2.914	8.897	8.989	
64		**0.075**	0.084	2.942	8.593	8.258	
70		**0.082**	0.097	2.932	7.678	9.113	
76		**0.073**	0.106	2.871	8.189	10.227	
82		**0.085**	0.099	2.903	8.266	10.381	
88		**0.083**	0.100	2.889	8.203	10.337	
94		**0.085**	0.099	2.897	8.510	10.315	
100		**0.084**	0.099	2.868	8.230	9.888	
1		**15108**	20199	3032339	**15108**	20199	
4	169168	**58651**	65202	3806153	59779	68687	
7		**117728**	157599	4227876	123667	157298	
10		**145623**	181242	4764235	148286	180593	
13		192355	214418	5527584	189534	209805	
16		201644	225471	5992181	205562	230519	
19		201099	235586	5879701	204280	228941	
22	**169168**	207509	230595	6049455	206419	233566	
25		196193	233567	6047661	197054	235723	
28		196130	243182	6001166	203289	237828	
31		191143	247404	6060702	185955	206190	
34		183019	248225	6058750	188308	211733	**CCs**
40	169168	**151765**	185843	6015178	212178	202816	
46	**169168**	170804	192706	6066024	242173	261430	
52		197833	226562	6076485	228440	268696	
58	169168	**161864**	187689	6094090	227370	228662	
64		186754	207362	6138907	223674	210593	
70		200774	237995	6029206	195718	229670	
76		181101	255407	6029857	213139	257020	
82	**169168**	207836	246870	6084857	210158	259925	
88		201640	248743	6070954	210870	259146	
94		207763	243764	6068184	217752	257468	
100		206479	241355	6014275	207955	246269	

Table 5.14 – Execution time by seconds and Number of CCs (p_1 = 0.75, p_2 = 0.19)

% IC	MAC-2001	LRB_{pdeg}	LRB_{deg}	LRB_{lex}	$EPDB_{pdeg}$	$EPDB_{deg}$	
1		**0.698**	0.896	1.874	111.769	194.669	
4		**3.328**	4.565	13.510	370.385	-	
7		**4.698**	5.877	32.571	-	-	
10		**6.590**	7.427	40.478	-	-	
13		7.379	**6.675**	42.993	-	-	
16		7.657	**7.591**	44.719	-	-	
19		**7.847**	8.638	44.873	-	-	
22		**8.835**	9.253	44.903	-	-	
25		**5.962**	8.694	44.027	-	-	
28		**7.711**	9.874	44.113	-	-	
31		**7.848**	9.780	44.086	-	-	
34	11.054	**8.938**	10.503	44.004	-	-	**time(sc)**
40		**8.357**	9.736	43.991	-	-	
46		**8.183**	9.297	44.067	-	-	
52		10.209	**9.597**	44.148	-	-	
58		**9.427**	10.268	43.885	-	-	
64		10.794	**10.731**	43.960	-	-	
70		**10.688**	11.700	43.985	-	-	
76		**8.355**	9.925	44.026	-	-	
82		**8.946**	10.461	44.056	-	-	
88		**8.271**	9.475	44.145	-	-	
94		**7.615**	8.883	44.184	-	-	
100		**7.600**	8.898	44.242	-	-	
1	11091737	**2053499**	2676894	5300103	2063315	2687746	
4		10050253	13609155	37787281	**7700192**	-	
7		14196226	17492480	90474816	-	-	
10		19917644	22168200	112775270	-	-	
13		22323569	19950505	119366418	-	-	
16		23178909	22661403	121865776	-	-	
19		23692559	25854188	122555689	-	-	
22		26684225	27586331	122734624	-	-	
25		18061593	25937447	122755174	-	-	
28		23355364	29466473	122739959	-	-	
31		23793094	29203775	122837687	-	-	
34		27114111	31384172	122669346	-	-	**CCs**
40	**11091737**	25268174	29127000	122734902	-	-	
46		24710542	27748892	122643912	-	-	
52		30820375	28720750	122943597	-	-	
58		28428460	30704619	122704450	-	-	
64		32524541	32018965	122798424	-	-	
70		32150097	34963393	122669749	-	-	
76		25192875	29655806	122768330	-	-	
82		26922873	31354768	122696907	-	-	
88		24982344	28307565	122684444	-	-	
94		22954833	26581585	122766266	-	-	
100		22965095	26579493	122848545	-	-	

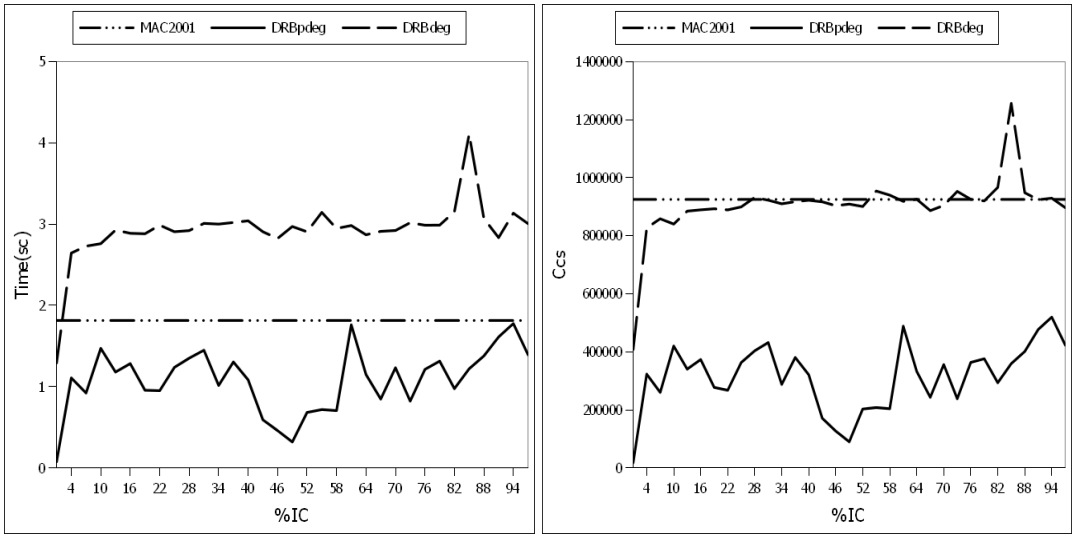

Figure 5.13 – Execution time and Number of CCs for MSPs

Regarding LRB, the use of pdeg is often more beneficial than using deg.
LRB_{pdeg} is always faster than MAC-2001 (1^{st} part of Tabs.5.13 and 5.14), even disturbing 100% of constraints, which is equivalent to resolving the whole problems. LRB_{deg} is sometimes less efficient than MAC-2001, but it is anyway a good concurrent.

Experiments on meetings scheduling problems. We present our results for the MSP section 2.2.1 class $< 15,30,4,5,10,1,85 >$ and vary the rate of disturbance from ~ 1% to ~ 100%. We generated 30 different instances treated by each algorithm and the results are an averages of these 30 runs. As LRB_{lex} and EPDB are not good concurrent, as shown in the experiments above, we include in comparison only MAC-2001, LRB_{pdeg} and LRB_{deg}.

Concerning MSPs (Fig.5.13), LRB_{pdeg} is more efficient than MAC-2001 regardless of the rate of disturbance, specially respecting the number of CCs. However, MAC-2001 is more effective than LRB_{deg} respecting runtime (except for a disturbance of 1%), even if the rate of CCs is comparable to that of LRB.

Experiments concerning solving

The previous experiments concern satisfiable problems. Experiments on random problems are more precisely focused on satisfiable problems in hard areas of low and high densities. These experiments show that LRB using a VOH, mainly pdeg, is a good concurrent for MAC-2001 although for solving (100% of IC).
In order to show where MAC-2001 is more efficient than LRB_{pdeg} and LRB_{deg}, we execute the three approaches on random problems, not necessary satisfiable, with the same previous parameters n, d and p_1. Constraints tightness' vary from $p_2 = 0.1$ to $p_2 = 0.9$. 30 different instances are treated by each algorithm and the present results are the averages of these 30 runs. Experiments are performed on the Java platform, on a Pentium IV PC (2.8 GHz processor and 3.24 Go RAM).

Figs.5.14 and 5.15 show that LRB, using either deg or pdeg, is efficient to treat satisfiable problems, i.e. $< p_1 = 0.25, p_2 \in [0.1, 0.45[>$ and $< p_1 = 0.75, p_2 \in [0.1, 0.19[>$, while MAC-2001 is effective to detect the non satisfiability of problems in $< p_1 = 0.25, p_2 \in]0.45, 0.9] >$ and $< p_1 = 0.75, p_2 \in]0.19, 0.9] >$. In complex areas, i.e. the transition phases $< p_1 = 0.25, p_2 = 0.45 >$ and $< p_1 = 0.75, p_2 = 0.19 >$, the majority of problems are non satisfiable, therefore, MAC-2001 is generally better than LRB, mainly for high densities,

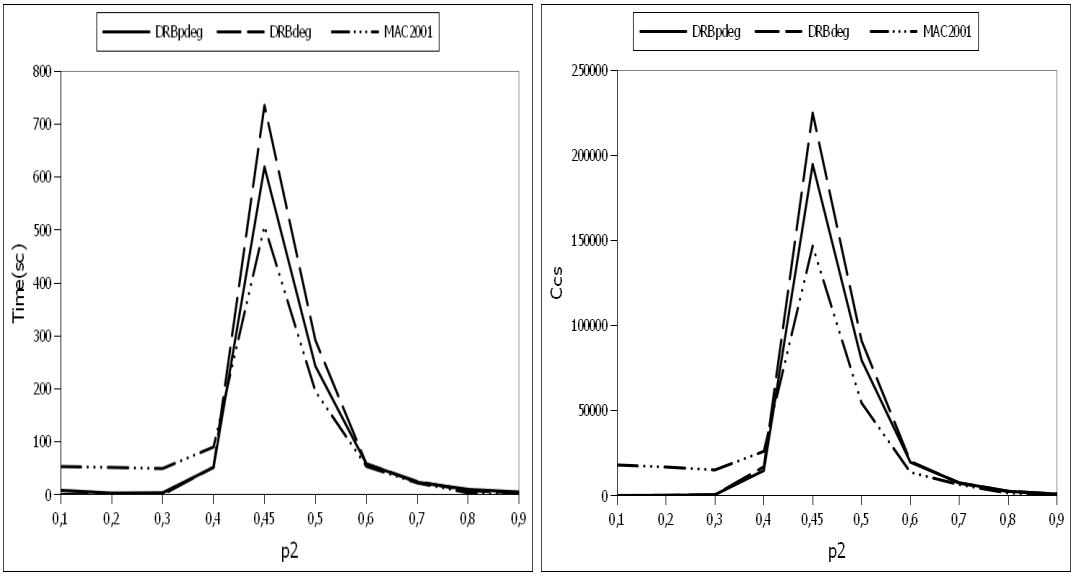

Figure 5.14 – Execution time and Number of CCs for $p_1 = 0.25$

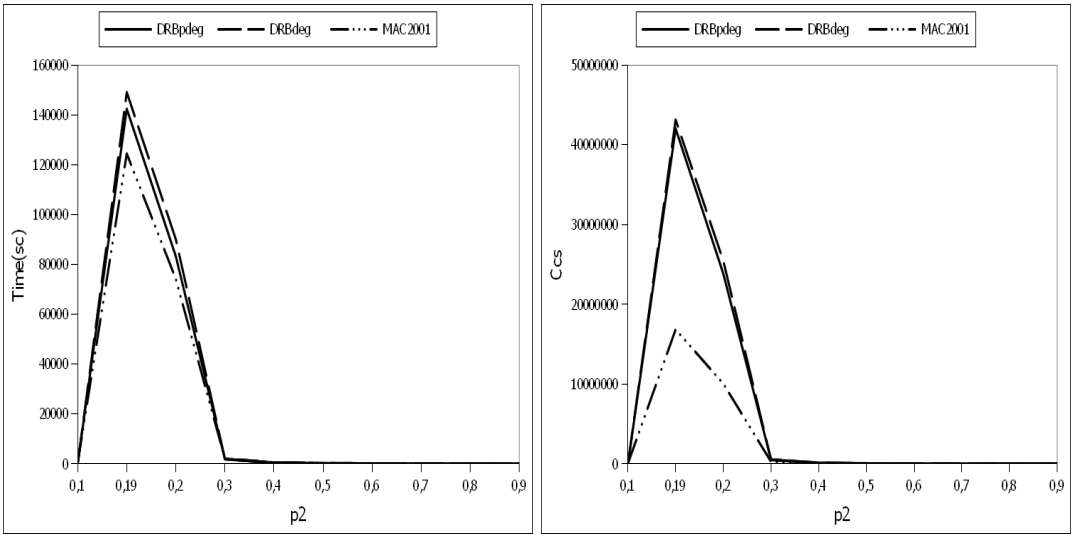

Figure 5.15 – Execution time and Number of CCs for $p_1 = 0.75$

where satisfiable problems are rare.
For high density problems (Fig.5.15), using either deg or pdeg in LRB is almost the same.

5.3.7 Discussion

Discussion concerning repairing

As shown in experiments on satisfiable problems in transition phases (Tabs.5.13 and 5.14), LRB is widely more powerful than EPDB in terms of runtime. This is particularly due to avoiding the verification of progressively saved orders, which is done in EPDB each time inconsistency is detected. But regarding CCs, the use of LRB with a VOH is almost the same as using EPDB with the same VOH. Normally, for a fixed heuristic, LRB and EPDB proceed in the same manner, with the difference of testing orders for EPDB. Thus, they must generate the same number of CCs. However, the perturbation, which is a modification in constraints, can change the used heuristic values, as both of the used VOHs deg and pdeg depend on the existing constraints. Then, in EPDB, nogoods stored before disturbance impose an order which is not necessary the same as the one expressed by the present heuristic values, whence the difference of the number of CCs between LRB and EPDB.
From a certain small disturbance, LRB requires almost the same effort to repair, whatever the rate of injected constraints, since problems are chosen from the most complex areas (the transition phases). This effort is not stable, but it is subject to the same sudden falls. In fact, the repairing effort depends on the affectation before disturbance and its consistency with the new constraints. Then, disturbing a problem with some constraints can be easier or harder than disturbing another problem with more constraints, depending on which affectation is satisfiable by more new constraints.
The use of pdeg in LRB (Tabs.5.13 and 5.14 and Fig.5.13) is often more effective than using deg, as pdeg estimates the position of each variable regarding the whole network, while deg calculates the position of each variable regarding its neighbourhood.
When problems are hard and satisfiable, LRB, mainly when using pdeg, is a good concurrent of MAC-2001, even for solving (100% of IC). Nevertheless, when no VOH is used (LRB_{lex} in 1^{st} part of Tabs.5.13 and 5.14), LRB becomes inefficient comparing to MAC-2001. In fact, the goal of VOHs in repairing is to engender minimal network changes, by keeping the hard sub-problems unchangeable as long as possible, to minimize risks of affecting other sub-problems. Therefore, the performance of LRB is also due to the use of a good VOH.

Discussion concerning solving

Concerning problems where constraints tightness is less than the peak (under the transition phase), although injecting the totality of constraints in perturbation, i.e. resolving (Figs.5.14 and 5.15), LRB using either deg or pdeg is more efficient than MAC-2001, regardless of problems density. In these areas, most of the problems are satisfiable, thus, the use of arc consistency is not efficient enough. The major interest of MAC-2001 is to detect earlier the future domains dead-ends, in order to modify the partial assignments which can not be included in a solution. But in sparse graphs, domains dead-ends are rarely met.
Therefore, for unsatisfying problems, where constraints tightness is higher than the peak, MAC-2001 is more effective than LRB.
On peaks, the majority of problems are unsatisfying, then, MAC-2001 is often more efficient than LRB. Respecting high densities, LRB_{pdeg} doesn't exceed remarkably LRB_{deg}. This is caused by the high connectivity of the network. Therefore, each variable has the majority of other variables as neighbors, then, pdeg is nearly the same as deg.

5.3.8 Conclusion and perspectives

In this section, we introduce a new approach, namely LRB, which is based on EPDB, to repair the DCSPs solutions rapidly, using static VOHs.
Based on the results of experiments on a variety of problem classes, we show that LRB is a great improvement of EPDB, respecting the runtime, insofar as it avoids the expensive tests of orders.
We prove that this approach is mainly efficient when using pdeg and that is a good concurrent for

MAC-2001. In fact, LRB outperforms MAC-2001 when problems are satisfiable, although using it to solve from the scratch. Thus, LRB has the advantage to be not only destined for DCSPs, but also to CSPs.
We demonstrate that LRB is more effective to treat satisfiable problems while MAC-2001 is more efficient to detect the problem's satisfiability. Therefore, for complicated problems where the satisfiability is unknown, the two approaches can be separately launched and the fast resolution is then guaranteed.
As perspectives, we plan to implement a CSP solver, based on LRB, which benefits from LRB advantages and resolves by repairing a complete assignment that violates a minimum number of constraints, in order to converge more rapidly to a solution. We also plan to develop a version of LRB for minimal perturbation problems [Zivan et al., 2011], to preserve DCSPs solutions as much as possible. We intend likewise to integrate constraint propagation within LRB, to benefit from the advantage of the two strategies regardless of the problem's satisfiability.

CHAPTER

6 Distributed Constraint Programming

This chapter begins a new part of this thesis which is the Distributed Constraints programming. influenced by the recent need to distribute tasks and activities. This chapter shows another advantage of using CP models to represent a problem is distributed setting.

Contents

6.1 Introduction

Distributed/Cooperative Constraints Satisfaction Problem (DisCSP) is a set of connected agents, each agent has a part of the problem. An agent can be seen as a Centralized CSP. Agents act autonomously to solve the problem i.e find a solution for each agent without violating constraints over agents.

[Faltings and Yokoo, 2005] has described many reasons and motivations to prefer modeling a problem like a Distributed and not Centralized CSP:

— **Cost of creating a central authority**: A central authority for solving the problem would require adding an element that was not present in the architecture. Examples of such systems are sensor networks or meeting scheduling.

— **Knowledge transfer costs:** Translating the problem into a common format and the communication costs may also be prohibitive for some applications due to the privacy of some problem's information.

— **Privacy/Security concerns:** Agents' constraints may be strategic information that should not be revealed to competitors, or even to a central authority. This situation often arises in e-commerce or trading applications for example (more details will be presented in a dedicated chapter).

— **Robustness against failure:** The failure of the centralized server can be fatal. In a distributed method, a failure of one agent can be less critical and other agents might be able to find a solution without the failed agent.

— **Problem decomposition:** the big problem can be divided into many small connected problems and solved parallelly.

— etc.

These reasons have motivated significant research activity in distributed constraint satisfaction. Up to the present, the field has reached a certain maturity and has developed a range of different algorithms to solve Distributed constraint satisfaction. In this chapter, we present some examples of DisCSP real cases and some algorithms to deal with them. These algorithms can be filled into three categories: complete synchronous algorithms, complete asynchronous algorithms, and incomplete asynchronous algorithms.

6.2 Distributed Constraints Satisfaction Problem

A distributed CSP (DisCSP) [Yokoo, 2012] is a constraint satisfaction problem in which variables, domains and constraints are distributed among multiple automated agents. More formally a DisCSP was described [Bessiere et al., 2003] as a five-tuple (X, D, C, A, α), where:

— X, D and C is a CSP.

— A is a set of agents.

— $\alpha : X \longrightarrow A$ is a function that maps each variable $x \in X$ to its agent $\alpha(x)$. Each variables belongs to one agent.

This distribution divides the set of constraints C into two parts: intra-agent constraints C_{inta} which is known only by the agent containing the connected variables, and inter-agent constraints C_{inter} that not be known only by the agents containing the connected variables. In general, like studies of DisCSPs [Bessiere et al., 2003, Solotorevsky and Gudes, 1997, Yokoo et al., 1998, Yokoo and Hirayama, 2000]

An agent is [Fioretto et al., 2018]

— Autonomous, as it operates without the direct intervention of humans or other entities and has full control over its own actions and internal state.

— Interactant, in the sense that it interacts with other agents in order to achieve its objectives.

— Reactive, as it responds to changes that occur in the environment and/or to the requests from other agents.

We assume in the following that: Agents do not have a global view of the problem due to privacy, security and an agent can send messages to other agents if it knows their addresses. The delay in delivering a message is finite, though random. Also, Messages are received in the same order in which they were sent. For simplify, we assume also that each agent contains only one variable, then, all constraints are inter-agent constraints i.e: $C = C_{inter}$ and $C_{inter} = \varnothing$

6.3 Distributed CSP examples

As said before, many real-life problems are naturally distributed or it's more desirable to model and deal with them distributedly. Some of these problem have received more interest from the CP community such as scheduling [Salido and Giret, 2008] sensor network [Zhang et al., 2003], resource allocation [Gür et al., 2019] , etc.

6.3.1 Distributed Meeting Scheduling Problem

The Distributed meeting scheduling problem (DisMSP)[Meisels and Lavee, 2004] is a decision-making process that consists of organizing meetings between participants, determining places and times, while respecting their different calendars. There are many variants of DisMSP in literature which differ from one to another by taking into account some parameters like meeting duration [], participant' preferences, etc. In this problem, a set of participants are looking for a possible meeting that satisfies the private constraints of each of them, while taking into account the time constraints to arrive between different meetings of the same participant. Most formally, general Meeting Scheduling Problems are characterized by A set of n attendees/participants, a set of m meetings each m_i takes d_i time, each meeting m_i will take place in a specific location l_i. Because each meeting takes place in a different location from one to another, there is a constraint called arrival-time ensuring that each participant which involved in two separated meetings m_i and m_j can move from l_i to l_j. Arrial-time constraint between m_i and m_j can be defined as fellow :

$$|time(m_i) - time(m_j)| - d_i > travel(l_i, l_j) \tag{6.1}$$

DisMSP can be modeled as CSP as follow :

— A Set of agents/participants.

— For each participant $p_i \in A$, X_i = {m_1, m_2, ..., m_i} is a set of p_i' variables/meetings.

— D={weekly working time-slots} it can be extracted from calender of participants $D(m_i) = \bigcap_{A_j \in \ participants(m_i)} Calender(A_j)$

— C=Arrival-times Constraints for every pair of meetings (m_i, m_j) there is a arrival-time constraint if there exists an agent involved in both meetings.

— For every pair of agents p_i, p_j that attends meeting m_k there is an equality inter-agent constraint between the variables x_{ik}, x_{jk} corresponding to the meeting m_k.

A simple DisMSP is illustrated in Fig.6.1, that show all the meetings that we have for all participants i.e: the participants P_1 and P_2 will attend to meeting m_1 which take place in l_1. then P_1, P_2 need 1 slot time to travel to l_2 to meet P_3 (m_2). After that P_1 must travel along four time-slots to l_3 where m_3 will take place,...etc. The example shown in Fig.6.1 can be modeled as CSP :

— A = {P_1,P_2,P_3}

— X = {m_1,m_2,m_3,m_4}

— D={D(m_1),D(m_2),D(m_3),D(m_4)}

  * D(m_1)={calender(P_1) $\cap calender(P_2)$}
  * D(m_1)={calender(P_1) $\cap calender(P_2) \cap calender(P_3)$}

Meetings	Locations	Participants
m_1	l_1	P_1,P_2
m_2	l_2	P_1,P_2,P_3
m_3	l_3	P_1,P_4
m_4	l_1	P_2,P_4,P_5

Locations	l_1	l_2	l_3
l_1	*	1	2
l_2	1	*	4
l_3	2	4	*

Figure 6.1 – Example of Distributed Meeting Scheduling Problem

* D(m_1)={calender(P_1) $\cap$calender(P_4)}
* D(m_1)={calender(P_2) $\cap$calender(P_4)$\cap$calender(P_5)}

— Arrival-time constraints C={ c_{12}, c_{22},c_{13},c_{14},c_{44} }.

6.3.2 Distributed Intelligent Marketplace

The Distributed Intelligent Marketplace problem consists of an intelligent system to support both buying and selling goods or services where customers and suppliers can meet, negotiate, make decisions and transact like in a traditional marketplace. The next chapter is dedicated to develop and detail the description and formulation of this use case.

6.4 Distributed CSP algorithms

In last decade, many Distributed algorithm have been developed to deal with Distributed CSP, some of them are refereed in follow: [Yokoo et al., 1992], [Yokoo, 1995], [Bessiere et al., 2005],[Silaghi and Faltings, 2005], [Yokoo and Hirayama, 1996]. Generally, these algorithms are falling into two categories: synchronous or asynchronous ones. In Synchronous algorithms, the first agent assigns sequentially their variables then gives the token to the next agent to processed to its own assignment. of course, there is an established order between agents, when a domain is wipe-out a simple or intelligent backtrack is launched to change the previous assignment then continue forward until finding a solution or detecting the unsolvability. Beyond the simplicity of technical programming, the limitations of these algorithms are clearly the inactivity of agents in the network except who have the token. In Asynchronous algorithm, all agents perform the assignments autonomously and parallelly, and each agent who finds its local solution shares it with some agents involved in its scope called in follow agents-view. Order between agents is still also required in order to define to who communicate the local assignment. The solution is found when no communication is holding in the network. These forms of algorithms are much more complicated and generate much more communication between agents. We will be focusing in follow on complete asynchronous algorithms, i.e: when a solution exists, the algorithm will surely find it. This kind of algorithm requires an exhaustive search of all the possibilities even using some heuristics to reduce the time complexity.

The work of Yokoo and colleagues, who proposed the asynchronous backtracking (ABT) and the asynchronous weak-commitment search (WCS) algorithms [Yokoo et al., 1992],[Yokoo, 1995],[Yokoo and Hirayama, 2000]. These algorithms require a total static or dynamic ordering among agents. When a dead-end is detected, nogoods are exchanged among agents and stored, which may require an amount of extra message passing so extra memory.

A different approach without nogood exchanging is the distributed backtracking algorithm (DIBT)[Hamadi et al., 1998], which performs graph-based backjumping. Agents are ordered according to an agent hierarchy and agent identifiers. Unfortunately, this algorithm has been shown incomplete [Yokoo and Hirayama, 2000].

Afterward, a new algorithm, called Asynchronous Aggregation Search (AAS) has been proposed in [Silaghi et al., 2000], which differs from existing methods in that it treats sets of partial solutions, exchanges information about aggregated valuations for combinations of variables, and uses customized messages to allow distributed solution detection.

Also, Distributed Dynamic Backtracking (DisDB) has been proposed in [Bessiere et al., 2001]. This algorithm makes use of some of the good properties of centralized dynamic backtracking. It ensures the completeness of the search and allows a high asynchronization level by sidestepping the unnecessary addition of links.

Six years later, a Distributed version of Forward Checking (AFC) has been introduced in [Meisels and Zivan, 2007]. Agents assign variables sequentially but perform forward checking asynchronously. AFC keeps one consistent partial assignment at all times. And forward checking is performed by sending copies of the partial assignment to all unassigned agents concurrently. The sequential assignment method of AFC leads naturally to the dynamic ordering of agents during the search.

6.4.1 Asynchronous Backtracking

Asynchronous Backtracking (ABT) Algo.11 [Yokoo, 2012] is a basic algorithm for DisCSP solving. It is executed systematically, autonomously, and asynchronously by each agent, which makes its own decisions(lines 2,13-15), informs other agents about them (lines 17-18), and no agent must wait for the others' decisions. The algorithm computes a global consistent solution (or detects unsolvability (line 3)) in finite time. Its correctness and completeness have been proven in [Yokoo et al., 1992]. ABT requires a static priority order of agents, and constraints are directed between the involved agents: the $value-sending$ agent, which is the higher priority one, and the $constraint-evaluating$ (lower priority) agent. When a $value-sending$ agent assigns its variables (lines 28,32), he sends their assignment to lower priority ($constraint-evaluating$) (lines 17,18) neighbors who try to make the assignment consistent (lines 14,15).
If a $constraint-evaluating$ agent is unable to make a consistent assignment (lines 19), it initiates backtracking by sending a nogood message to a higher priority agent (lines 25), in order to change its current value assignment. Agents keep a nogood list of backtrack messages and use it to guide the search (lines 31). A solution is found if there is quiescence in the network, while unsolvability is reached when an empty nogood is discovered (lines 21-23).
Basically ABT like presented before in its original version is not complete. Bessiere et al. have proposed various improvements called (*ABT-family* [Bessiere et al., 2005] to fix the completness issue throught integrating some additional links between initial unlinked agents. 4 versions have been proposed by authors (ABT_{during}, ABT_{all}, $ABT_{temps(k)}$, ABT_{not}) to specify how and when agents request for add links :

- ABT_{during} or just ABT in which links are added dynamically when an agent receives a nogood containing unknown higher priority agent. The new links are permanent.
- ABT_{all}: All the potentially useful links are added during a preprocessing phase. The new links are permanent.
- $ABT_{temps(k)}$: like ABT the links are dynamically, but the links are temporary i.e. after a fixed number k of messages communicating through this link, it will be removed to disconnect both agents.
- 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

Two procedures Algorithm.12 have been added by bessiere and al. When an agent receives a nogood message, it checks for each variable of nogood' left side (line 35), if it's not a part of its Agentsview an addlink message (adl) (lines 36-37) will be sent to such agent in order to establish a new logical link between these two unrelated agents (line 37), then this variable will be added to the Agents view and its value will be updated(line 38-39). When an agent receives addlink message, it adds a sender agent to its child agents' list, and a message ok will be sent to this new recognized agent(line 33-34). To be noted, these added links are logical and don't have any impact on the problem's structure or solvability. Afterward, [Zivan and Meisels, 2006] have proposed ABTDO dynamic agents ordering ABT which outperforms static order ABT by a large factor in run-time and improves the network load. Some efforts of integrating arc consistency with ABT have been concluded by developing an ABT version called ABT with Arc consistency (ABT-uac or its generalized version ABT-dac) [Brito and Meseguer, 2008].

When a nogood is detected and has an empty left-hand side, the considered value is eliminated unconditionally, once and for all. This value deletion can be propagated using standard arc consistency techniques, producing new deletions in the domains of other variables. This causes substantial reductions in the search effort required to solve a class of problems. More recently, [Mechqrane et al., 2020] have proposed *ABT-Agile* which is a general framework for reordering agents asynchronously This is done via the original notion of termination value, a label attached to the orders exchanged by agents during the search. We will present in the following, a dedicated chapter that proposes an extension of ABT called LiveABT [Benamrane et al., 2017] which is a real time Distributed Dynamic CSP algorithm.

Algorithm 11: Asynchronous Backtracking Algorithm 1/2

```
Procedure ABT()
1.  end ← false;
2.  CheckAgentView();
3.  while (¬end) do
4.     msg ← getMsg();
5.     switch (msg.type) do
6.        ok? : ProcessInfo(msg);
7.        ngd : ResolveConlict(msg);
8.        stop : end ← true;
9.     end
10. end
Procedure CheckAgentView(msg)
12. if msg is relevant then
13.    update local context ;
14. if Consistent(myValue, myContext) then Return;
15. myValue ← ChooseValue();
16. if myValue then
17.    for each child ∈ Γ+(self) do
18.       sendMsg : ok(child, myValue);
19. else Backtrack;
Procedure Backtrack()
20. newNogood ← Solve(myNogoods) ;
21. if newNogood = empty then
22.    end ← true ;
23.    sendMsg : Stop(system);
24. else
25.    sendMsg : Back(newNogood);
26.    update local context;
27.    CheckAgentView();
Function ChooseValue()
28. for each v ∈ D(self) not eliminated by myNogoods do
29.    if Consistent(v, myContext) then Return (v) ;
30.    else
31.       Add(X = valX ⇒ ¬ v;C{ci} , myNogoods);
32. Return (null);
```

Algorithm 12: Asynchronous Backtracking Algorithm 2/2

```
Procedure SetLink(msg)
33. add(msg.sender,Γ+(self));
34. sendMsg :ok?(msg.sender,myValue);
Procedure CheckAddLink(msg)
35. for each var ∈ lhs(msg.Nogood) do
36.    if (var ∉ Γ−(self))) then
37.       sendMsg :adl(var, self);
38.       add(var,Γ−(self));
39.       Update(var ← varValue);
40.    end
41. end
```

Execution Example

We show an example [Yokoo et al., 1992] of a complete version of ABT algorithm execution in Fig.6.2 on a basic problem of 3 agents connected by two inequality constraints Fig.6.3. Each agent

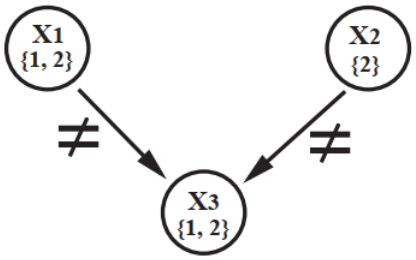

Figure 6.2 – Example ABT

try to assign it's variable then communicates its assignment to their neighbors/child. In Fig.6.3(a),

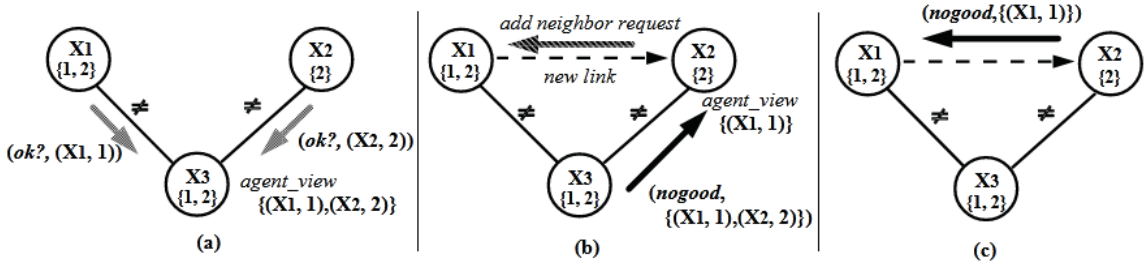

Figure 6.3 – Example ABT

after receiving ok? messages from X_1 and X_2, the agent view of X_3 will be $\{(X_1;1),(X_2;2)\}$. Since there is no possible value for X_3 consistent with this agent view, a new nogood $\{(X_1;1),(X_2;2)\}$ is generated. X_3 chooses the lowest priority agent in the nogood, i.e., X_2, to send to the nogood message. After receiving this nogood message, X_2 records it. This nogood, $\{(X_1;1),(X_2;2)\}$, contains agent X_1, which is not a neighbor of X_2. Therefore, a new link must be added between X_1 and X_2. X_2 requests X_1 to send X_1's value to X_2, adds $(X_1;1)$ to its agent view (Fig.6.3(b)), and checks whether or not its value is

consistent with the agent view. The agent view (X_1;1) and the assignment(X_2;2) violates the received nogood {(X_1;1),(X_2;2)}. However, there is no other possible value for X_2. Therefore, X_2 generates a new nogood (X_1;1), and sends a nogood message to X_1 (Fig.6.3(c)).Once X_1 receives a nogood from X_2, it changes its value, then sends ok? message { $X_1 = 1$}, X_2 and X_3 react by assigning their variables appropriately, then no communication holds in the network, the silence will be detected and the search process declares the solution.

6.4.2 asynchronous weak-commitment search

Suffering from static ordering, when a bad value of higher agent order is selected, it's imposed to the lower agents order which performs an exhaustive search when the searching solution. To skip this situation *AWC* algorithm uses the min-conflict heuristic [Minton et al., 1992] to reduce the risk to choose a bad value, in plus uses also dynamic agents ordering to revise the bad chosen values to avoid the exhaustive search. All *AWC* variables have a tentative initial values, and called as AWC({(x_1, d_1),(x_2, d_2),(x_3, d_3),...(x_n, d_n)},{}), where d_i is an initial value of x_i. AWC constructs a partial solution by adding variables one by one until a solution is found. Otherwise a partial solution that can't be extended to a solution is recorded as a constraint to avoid algorithm to construct the same assignment, differently to min-conflict backtracking algorithm who to abandons the whole nogood partial solution when a swipe out is detected. When a variable is added, its tentative initial value is revised so this new value satisfies all constraints between its variable and the current partial solution and as many as possible other variables not yet involved in the partial solution. When selecting a variable value within multiple choices, the agent prefers the value that minimizes the number of constraint violations with variables of lower priority agents. AWC changes the priority of agents dynamically based on the following rules:

— For each agent, a non-negative integer value representing the priority order of the agent is defined. We call this value the priority value.

— The order is defined such that any agent with a larger priority value has a higher priority.

— If the priority values of multiple agents are the same, the order is determined by the alphabetical order of the identifiers.

— For each agent, the initial priority value is 0

— If there exists no consistent value for x_1, the priority value of x_1 is changed to $k+1$, where k is the largest priority value of other agents.

In the asynchronous weak-commitment search, each agent concurrently assigns a value to its variable and sends the value to other agents. Then agents wait for and respond to incoming messages. The Algorithm.13 and 14 show the procedure of *AWC* search which differ from that of ABT by the following:

— Both priority value and the current value, are communicated through the 'ok' message (Algorithm.13 line 19).

— The priority order is determined by the communicated priority values. If the current value is not consistent with the *Agent_view*, i.e., some constraints with variables of higher priority agents are not satisfied, the agent changes its value to the value which is consistent with the *Agent_view*, and also minimizes the number of constraint violations with variables of lower priority agents (Algorithm.13 lines 30-32).

— When $agent_i$ can not find a consistent value with its *Agent_view*, $agent_i$ sends 'nogood' messages to other agents, and increments its priority value. If $agent_i$ has already sent an identical 'nogood' ,$agent_i$ will not change the priority value and will wait for the next message (Algorithm.14 lines 1-15).

Algorithm 13: Asynchronous Weak-Commitment 1/2

```
1  Procedure AWC(left, partial-solution);
2  while all variable in left satisfy all constraints do
3  |  terminate the algorithm;
4  |  return current assignment as a solution;
5  (x_i, d) ← a variable and value pair in left that does not satisfy some constraint;
6  values ← the list of x_i's values that are consistent with partial-solution;
7  if values is an empty list then
8  |  if partial-solution is an empty list then
9  |  |  terminate the algorithm since there exists no solution;
10 |  else
11 |  |  record partial-solution as a new constraint (nogood);
12 |  |  AWC(left, partial-solution);
13 else
14 |  value ← the value within values that minimizes the number of constraints violations with
   |   left;
15 |  remove (x_i, d) from left;
16 |  add (x_i, value) to partial-solution;
17 |  AWC(left, partial-solution);
18 Procedure Process (message);
19 if message is ('ok', (x_i, d_j, priority)) then
20 |  add(x_i, d_j, priority)) to agents_view;
21 |  Check_Agent_View();
22 if message is ('nogood', x_j, nogood) then
23 |  add nogood to Nogood_list;
24 |  Check_Agent_View();
25 Procedure Check_Agent_view();
26 while agents_view and current_value are not consistent do
27 |  if no value in D_i is consistent with agents_view then
28 |  |  backtrack;
29 |  else
30 |  |  select d ∈ D_i consistent with agents_view and d minimize the number of constraints
   |  |   violations;
31 |  |  current_value ← d;
32 |  |  send ('ok', x_i, d, current_priority) to other agents;
```

Algorithm 14: Asynchronous Weak-Commitment 2/2

```
1  Procedure Backtrack();
2  nogoods ← { V | V = inconsistent subset of agent_view} ;
3  while an empty set is an element of nogoods do
4  |  broadcast to other agents that there is no solution;
5  |  terminate;
6  while no element of nogoods is included in nogood_sent do
7  |  foreach V ∈ nogoods do
8  |  |  add V to nogood_sent;
9  |  |  foreach (x_j, d_j) ∈ V do
10 |  |  |  send ('nogood', x_i, V) to x_j
11 |  p_max ← max(x_j, d_j, p_j) ∈ Agent_view(P_j);
12 |  current_priority ← 1 + p_max;
13 |  select d ∈ D_i where d minimizes the number of constraints violations;
14 |  current_value ← d;
15 |  send('ok', x_i, d, current_priority) to other agents;
```

6.5 Conclusion

In this chapter, we introduced the concept of Distributed CSP, then Dynamic Distributed CSP, and we presented some real examples dealt with these formalisms and some of the popular approaches i.e ABT and AWC. In the next chapter, we are going to develop our contribution focused on modeling the Distributed Intelligent Marketplace problem.

CHAPTER

7

A Distributed Constraint Reasoning approach towards intelligent marketplace environment

In this chapter, we propose a new use case formulation and Distributed Constraint Reasoning protocol to deal with e-marketplace issues. More specifically, our attention is focused on the constraint-based multi-agent approach offering a flexible and confidential multilateral negotiation mechanism.

Contents

7.1 Introduction

With the growing success of e-marketplace adoption, the need for new intelligent approaches to support both buying and selling goods or services becomes a fundamental requirement. In the recent past, several techniques have been proposed, in different research areas, allowing the simulation of a market with a set of sellers proposing products and buyers who have a list of interests.

An electronic marketplace is a place where customers and suppliers can meet, negotiate, make a decision and transact like in a traditional marketplace. With the growing need for e-marketplace in our daily life, the need for new innovative techniques to support both customers and suppliers in buying and selling goods or services is increasing quickly.

In the recent past, several platforms [Eriksson et al., 1998] where people can look for a given good or service has changed the traditional ways of negotiating and doing business. These various forms of e-marketplaces are considered as revolutionary vectors of e-marketplace technology. However, most of them only serve a small part of the transaction process and need human intervention before or during the negotiation process. Nowadays, autonomous and intelligent multi-agent systems make the business processes in the e-market more efficient and revolutionary [Ren et al., 2009] and [Hemaissia et al., 2007]. These new sophisticated systems may replace both buyers and sellers in decision-making in order to reach a negotiated agreement. In particular, virtual sellers and buyers can be provided with a negotiation protocol that helps them to specify constraints and then to look for the optimal/pseudo-optimal decision regarding these specified preferences (constraints). When the combinations of multi-lateral negotiation become complicated, and the constraints both on the customers' and suppliers' sides become huge and complex, intelligent negotiation gives rise to new challenges for developers of architecture and software technologies underlying e-marketplaces. Hence, intelligent agents should be initially created with their complete set of strategies and should be able to learn and make a decision; rather than having an automation behavior, they should have the ability to acquire experience from previous negotiations they've conducted. Although completely uncontrolled e-marketplaces are still rather a vision than reality.

To achieve this challenge and remove these limitations, some projects have been initiated, such as SICS MarketSpace [Eriksson et al., 1998]. Afterwards, many negotiation mechanisms have been proposed in the literature [Fatima et al., 2007], [Lai et al., 2007], [Ren et al., 2009] and [Hemaissia et al., 2007]. The most important work, in our point of view, has been done on the negotiation process and the formulation model is still unripe and is not applicable in realistic-use environments.

One of the most promising answers to the new challenges posed by arising e-marketplaces is the "Constraint Programming" paradigm. Several papers have proposed Distributed Constraint Reasoning (DCR) as a paradigm for problem modeling and solving in the framework of Multi-agent Systems. One of the main features of DCR approaches is its distributed nature in which a set of intelligent agents can each make local decisions and communicate in order to improve global decision problems. To the best of our knowledge, few works have been interested in the formulation of e-marketplace environment using Distributed Constraint Reasoning techniques [Parunak et al., 1998] [Rahwan et al., 2002].

In this chapter, we propose an e-marketplace environment formulation and a DCR protocol to deal with e-marketplace issues. More specifically, our attention is focused on the constraint-based multi-agent approach offering a dynamic and privacy multi-lateral negotiation mechanism, namely, ABT-Trader. Also, the framework is based on two resolution phases, the registration and building network phase, in which each agent builds its distributed constraint network, and the negotiation phase, in which the sellers and buyers agents negotiate using the distributed constraint-based techniques reasoning. The satisfaction of strict preferences offers the best negotiation to buy all the wanted products, otherwise, constraint and variable relaxations are adopted to look for a feasible solution.

The remainder of this chapter is organized as follows. Section 2 describes the main problem of the e-marketplace environment, and the important issues to fully automated negotiation. Section 3 addresses the distributed constraint reasoning model using privacy and relaxation concepts. Section 5 focuses on constraint resolution protocol as a negotiation mechanism. Section 6 discusses preliminary

experiment evaluation, and section 7 gives a brief overview of related works. Finally, section 8 provides a conclusion and areas for further research.

7.2 Problem statement

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

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

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

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

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

— A multilateral negotiation space;

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

— A strict preference model that can be relaxed during negotiation.

The problem can be described as bellow :

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

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

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

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

— Each buyer has a fixed budget that cannot be exceeded.

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

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

— Each agent preference is considered as private.

7.3 Distributed constraints reasoning model

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

Agents :

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

Variables :

We assume that :

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

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

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

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

Domains :

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

Constraints :

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

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

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

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

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

$$C_{inter}^{seller} = \{C_{seller_i, buyer_j} = \{X_k^{Buyer_j} \geq \min(D(X_k^{Seller_i}))\} \tag{7.1}$$

for each seller, i offering the product k interesting the buyer j

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

$$C_{intra}^{buyer} = \{\sum_{i=1}^{k} X_i^{buyer_j} \leq Budget(Buyer_j)\} \tag{7.2}$$

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

$$C_{inter}^{buyer} = \{C_{buyer_i, seller_j} = \{X_k^{seller_j} \geq \max(D(X_k^{Buyer_i}))\} \tag{7.3}$$

for the seller i offering the product k interesting the buyer j.}

7.3.1 The Model options

Agent priority

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

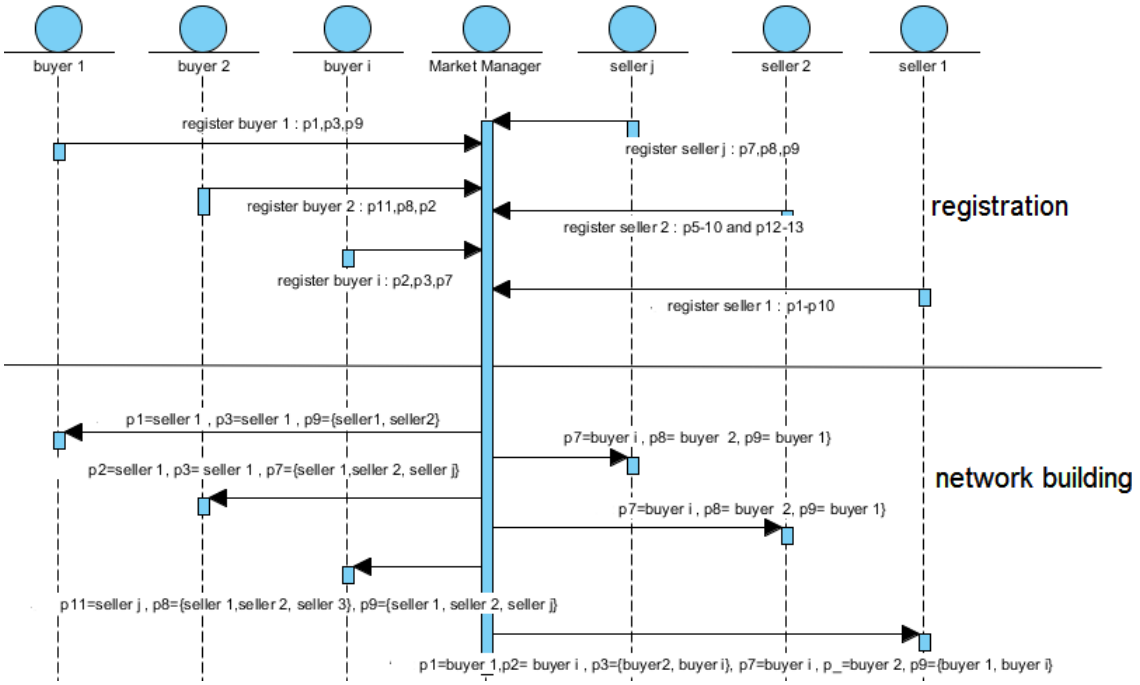

Figure 7.1 – Registration and building the DCSP network

Value Ordering heuristics

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

Variable and constraint relaxation

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

7.4 Resolution protocol

In the resolution protocol, we can observe two phases, the first phase is the building of the Distributed Constraint Satisfaction Problem network, and the second phase is the negotiation process. The registration and negotiation phases presented in Fig.7.1, and Fig.18 show two types of agents: buyer and seller.

7.4.1 Registration process

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

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

product 1, 3, and 9.

7.4.2 Negotiation process

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

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

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

7.4.3 Algorithm description

The ABT-Trader is based on ABT[Bessière et al., 2002] and used by two types of agents: buyer and seller. The pseudo-code of ABT-Trader represented in Algo.15 and 16 says that the seller stores one nogood per removed value in the NogoodStore for each negotiation till the end of the resolution. All agents start the search by calling the procedure *Setup()* in which they initialize their nogoodStores, agentViews, finalAgentViews, products, etc. Then each agent checks his type (buyer or seller). If the agent is a buyer, in this case, he asks all available sellers of each product to be his parents and to negotiate with him about the price of this product, after that, he calls the main procedure named Buyer(), if not (i.e. the agent is a seller) he runs a loop where he waits for possible buyers, and when a buyer arrives (i.e. *sell?* message received), the agent seller creates a copy of all data (i.e. domains, nogoodStores, etc) and calls a new Seller() procedure to negotiate with this buyer about the product carried in the message *sell?* .

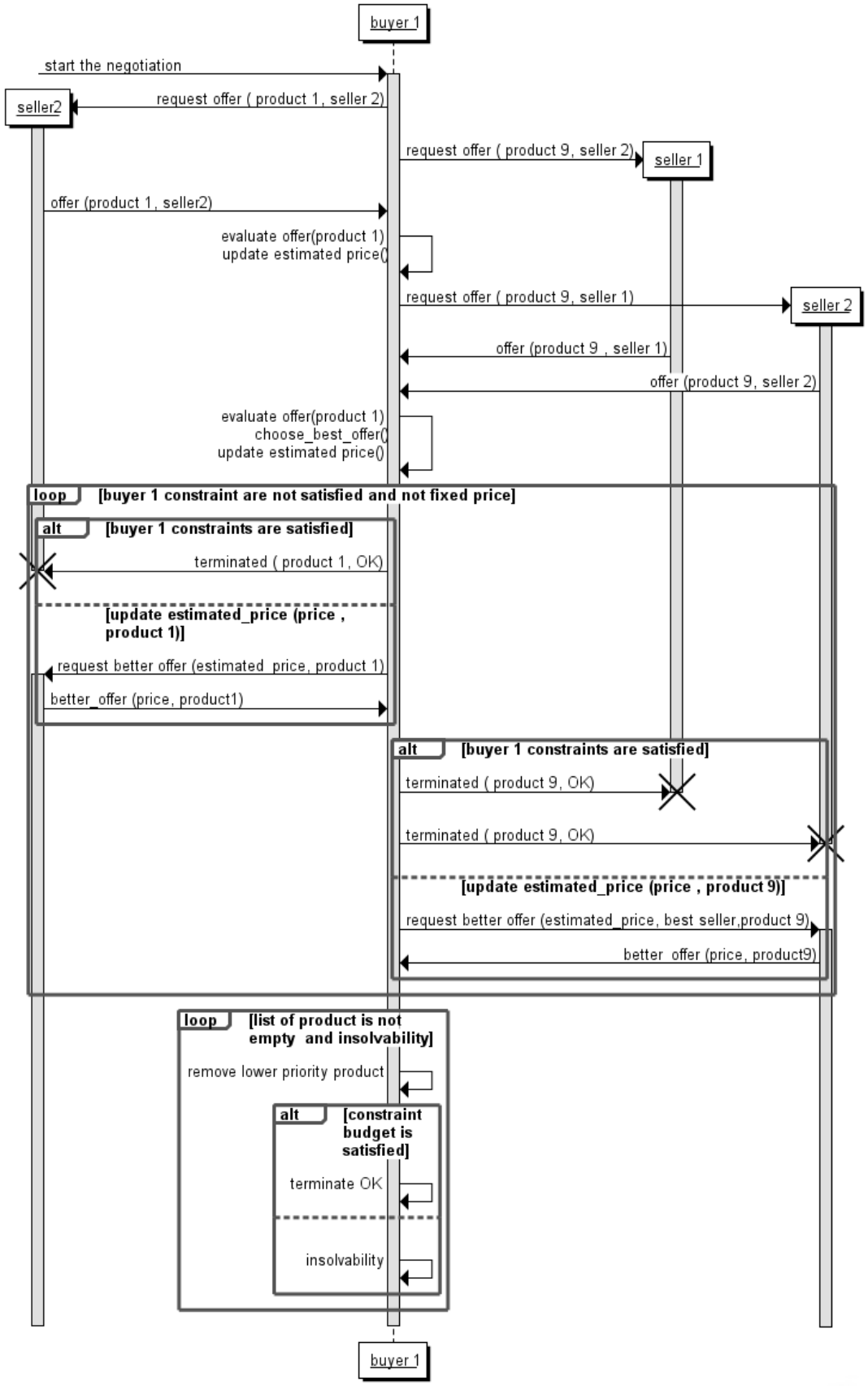

Figure 7.2 – Negotiation process

Algorithm 15: ABT$_{\text{Trader}}$ 1/2

```
1  procedure Setup();
2  initialize products, domains, myAgentViews, myFinalAgentView, nogoodStores, and type;
3  Register with all products and type;
4  if type = buyer then
5  |  foreach product ∈ products do
6  |  |  foreach seller ∈ sellers(product) do
7  |  |  |  sendMsg : sell?(seller,product);
8  |  |  |  add seller to the list of sellers;
9  |  Buyer();
10 else
11 |  while true do
12 |  |  msg ← getMsg();
13 |  |  if msg.type=sell? then
14 |  |  |  Seller(msg.Sender, msg.product);
15 |  |  else
16 |  |  |  msg.isRead← false;
17 procedure Buyer();
18 myvalues ← empty;
19 end ← false;
20 while ¬end do
21 |  msg ← getMsg();
22 |  switch (msg.type) do
23 |  |  ok? : ProcessOk(msg);
24 |  |  finalOk :ProcessFinalOk(msg);
25 procedure Seller(buyer,product);
26 ChooseValue(buyer,product);
27 end ← false;
28 copy nogoodStores and domains;
29 while ¬ end do
30 |  msg ← getMsg();
31 |  switch (msg.type) do
32 |  |  nogood : ResolveConflict(msg);
33 |  |  stop : end ← true;
34 |  delete current nogoodStores and domains;
35 procedure ProcessOk(msg);
36 Update(myAgentView,msg.Assign);
37 CheckAgentView();
38
```

In the first time, the buyer waits for all sellers to propose their prices within a time out, then he tries to choose the best-proposed values/prices and try to assign them. The buyer agent assigns all values only when all constraints are satisfied, in this case, a solution is found. Otherwise, when there is a problem in this phase (i.e. the buyer cannot assign all variables), he tries to send a nogood per product to the seller who has not sent a finalOK? message, and who has proposed the best price for this product, after that, he waits with a time out for the new propositions, in this part, he does not wait for all new propositions to arrive, he tries to find a solution when a new proposition is received. In case of all sellers have proposed their final prices for all products (i.e. the size of the agentView equal the size of FinalAgentView) and there are some not satisfied constraints, the buyer uses the priority between products to buy just products that have a high priority and those satisfy budget constraint only.

Algorithm 16: ABT$_{Trader}$2/2

```
1  procedure ProcessFinalOk(msg)
2  Update(myAgentView,msg.Assig);
3  Update(myFinalAgentView,msg.Assig);
4  CheckAgentView();
5  procedure ResolveConflict(msg)
6  if The size of values those not eliminated by the current nogoodStore > 1 then
7  |  Add(msg.Nogood,myNogoodStore);
8  ChooseValue(buyer,product);
9  procedure ChooseValue(buyer,product)
10 get all values ∈ D(product) and not eliminated by the current goodStore;
11 if values.size=1 then
12 |  sendMsg: finalOk(buyer, values.getvalue, product);
13 else
14 |  sendMsg: Ok?(buyer, values.getBiggestValue, product);
15 procedure CheckAgentView()
16 if AgentView.size ≤ sellers.size and waiting_time < time_out then
17 |  Exit this Procedure;
18 else
19 |  Choose the best prices from the agentView and try to assign them to myValues;
20 |  if myvalue are not assigned;
21 |   then
22 |  |  Choose the best price for each product and not in finalAgentView and send nogood with
   |  |   the my proposed price to its seller;
23 |  |  if No nogood sent then
24 |  |  |  Try to assign variables or products who have the high priority and send msg stop to
   |  |  |   all sellers;
25 |  |  |  end ← true;
26 |  else
27 |  |  Send msg stop to all sellers;
28 |  |  end ← true;
```

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

7.5 Experiment

7.5.1 Experimental platform

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

7.5.2 Experimental Settings

We use XML files to describe the sub-problems for each agent (i.e. buyer or seller). Fig.7.3 shows an example for an agent seller and the Fig.7.4 shows an example for an agent buyer before the network building. The inter constraints will be added during the registration process. We have used in this experimentation two buyers named C1 and C2 and three sellers named F1, F2, and F3. The agent C1 wants to buy products X, Y, and Z. C1 has 115 as a budget, and has an estimated price for each product: D(X) = [100, 200], D(Y)=[150, 300], D(Z)=[30, 70]. In case of unsolved problem, he tries to buy the products X then Y, and then Z (i.e., X is more prioritized than Y, and Y is more prioritized than Z). The agent C2 wants to buy two products X and Z. C2 has 30 as a budget, and has an estimated price for each product: D(X) = [50, 100] and D(Y)=[10, 70]. Like C1 in the case of unsolved problem, he tries to buy products X then Y (i.e X is more prioritized than Y). As a seller, F1 proposes two products X and Y, he can sell X from 25 to 50 and Y from 70 to 100. F2 proposes the three products X with a fixed price: 70, Y for 70 to 100, and Z for 40 to 60. Finally, F3 proposes X and Y as products, 80 to 120 for X, and 60 as a fixed price for Y.
We assume that there are no constraints between sellers or between buyers. And for a seller, there are no intra-constraints. Each agent was launched in a machine core i7 with 8Go of Ram, and they can communicate between them using the HTTP protocol (e.i. Each machine is connected to the internet).

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

Figure 7.3 – seller F1 definition

7.5.3 Experimental Results

The solving phase is stopped by the agent buyer when finishing all negotiations. The Fig.7.5 shows the start and the end of the buyer C1 negotiations, where he can buy all products: X from F1 for 25, Y from F2 for 50, and Z from F2 for 40, and the Fig.7.6 shows the start and the end of the buyer C2 negotiations, as a result, he can buy only the product X from the seller F1 for 25. These preliminary results show the effectiveness of our approach, since the negotiation perform a relevant negotiated price.

7.6 Related works

MarCon[Parunak et al., 1998](market-based constraints) aims to support a mix of human and artificial agents by offering a systematic method for applying markets to a wide array of problems.
A MarCon configuration is a network of alternating variables and constraints, each variable and each constraint is a separated agent. The initial network construction must have at least two constraints

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

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

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

   <agents_neighbours>
   </agents_neighbours>

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

Figure 7.4 – buyer C1 definition

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

Figure 7.5 – buyer C1 negociation log

and one variable, in other definitions, it must have a buyer, a seller which are agents constraints, and a variable which is the environment where calculation happens.
Although the platform uses the notion of agent, but it does not respect the literature of distributed constraint problems in the side as the constraints and variables are separated agents rather than having each agent with a set of variables, domains, constraints.
On the other hand, they converge toward the solution is not always valid because of the shrinking of

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

Figure 7.6 – buyer C2 negociation log

the range price. However, if the participants' price or assignment ranges do not intersect, the variable agent recommends human negotiation between the buyer and seller, so the platform became invalid.

In [Rahwan et al., 2002], authors present a DisCSP framework for one to many negotiations using conducting some concurrent coordinated one-to-one negotiation, and assume that this framework can be extended to be able to solve the Many-To-Many negotiation problem. Also, the framework is based on two negotiation levels, the agent negotiation level in which each agent uses the constraint-based techniques reasoning, and the coordination level in which the coordinator agent evaluates how well subordinates agent has done and issues the new instructions. However, negotiations are focused on one product with multi-criteria, and the DisCSP formulation is not presented.

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

7.7 Conclusion and future works

Intelligent multi-agent electronic marketplaces are promising, and they will play an essential role in e-commerce and e-business activities. With the growing success of e-marketplace adoption, the need for new intelligent approaches to support both buying and selling goods or services has become inevitable. With these approaches, the discovery procedures of products, the issues negotiation, and the solution search must use a real communication challenge.

In this chapter, advances of distributed constraint reasoning approaches were operated. We used the agent concept to describe the stakeholders to respect all their proprieties. We have proposed a new problem formulation and DCR protocol to deal with e-marketplace issues. More specifically, our attention was focused on the constraint-based multi-agent approach offering a dynamic and privacy multi-lateral negotiation mechanism, namely, ABT-Trader.

Our approach has been implemented and tested using preliminary generated problems. The opening experimental results are promising and further investigations will be conducted. We have just explored the offers destined to satisfy the continuing expectations of the market, however, we are convinced that

all those offers should be treated by our approach. Particularly, fuzzy and dynamic preferences, real case experimentation, and the learning process.

CHAPTER

8 Dynamic Distributed CSP

*This small chapter aims to present some backgrounds to design and treat distributed Dynamic CSP, Especially Dynamic Asynchronous Backtracking (*DynABT*).*

Contents

8.1 Introduction

As shown previously, Distributed Artificial Intelligence (DAI) [Ferber, 1999] is concerned with interaction and coordination among artificial automated agents to solve a given problem. In real life, a wide range of the DAI problems (e.g, Scheduling Problems [Hentenryck and Michel, 2009, Salido and Giret, 2008], Sensor Networks Problem [Zhang et al., 2003], Resources Allocation problems [Benamrane et al., 2014, Hannebauer and Müller, 2001], Smart Grid and Smart Home Problems [Davidson et al., 2009, Ramchurn et al., 2012], ServoSystems Problems [Precup et al., 2007, Türkşen and Tez, 2016], Traffic Light Synchronization Problems [De Oliveira et al., 2005, Junges and Bazzan, 2008]) are dynamic.

For example, in distributed Meeting Scheduling Problem any participant could change his private calendar while the process of searching for a solution is turning, new meetings might need to be scheduled.

These unpredictable and challenging behaviors can be related to the:

— **User**: the requirements of the user can change in the context of an interactive design;

— **Environment**: the conditions/characteristics of the environment may evolve in the framework of a dynamic design;

— **Agent**: the autonomous decisions of the agent can change in the context of a distributed system;

Certainly, it is possible to solve the new changed problem from scratch. This technique can remember nothing about the previous reasoning process, and has some drawbacks:

— Inefficiency, which is unacceptable in real-time applications context (planning, scheduling, etc.), where the time allowed for re-planning is limited;

— Instability of the successive solutions, which may be unpleasant in the framework of an interactive design or a planning activity, if some work has started based on the previous solution, or if it is necessary to keep the resource allocations as stable as possible.

— etc...

Generally, the solution found by the above approaches concerning the Dynamic Distributed CSP is based on the previous one, but these approaches can not consider the changes until solving the current problem instance, thus, they can not support real-time perturbations.

We distinguish between two scenarios of perturbation :

— Slow scenario: in which the time between two successive perturbations is enough to solve a problem, in this case, the use of the previous solution seems to be natural and simple.

— Rapid scenario: in which case, no enough time exists to solve the intermediate problems, the problem is continuously disturbed and the intermediate problems become quickly obsolete.

There are many recent applications for which the approaches must include the dynamism treatment in their native process, e.g. the many to many bargaining models especially for the high-frequency trading problem[Goldkamp and Dehghanimohammadabadi, 2019], the simulation of the biological cell division [Fages, 2016, 2020], etc.

8.2 Dynamic DisCSPs Formalism

Definition 8.1 (DDisCSP)**.** *Dynamic Distributed Constraint Satisfaction Problems (DDisCSPs) can be defined as a six-tuple ($\coprod$, α, δ) where:*

— $\coprod$ *is a DisCSPs;*

— δ *is the change function that introduces changes at different time intervals, i.e., it represents changes in the problem over time.*

DDisCSPs can be used to model problems that are distributed in nature and change over time.

All the previous CSP and DCSP characteristics and assumptions presented in the previous chapters till valid for the next.

8.3 Dynamic DisCSP Methods

Many complete and incomplete approaches [Wallace et al., 2009] have been proposed to deal with this problems' kind, such as:

The distributed hill-climbing technique called Peer-to-Peer(P2P) [Fitzpatrick and Meertens, 2003]. Periodically, each variable adjusts its own value to optimize the total values of those constraints in which it is involved, optimization being for what it currently knows of the values of its neighbors. And when a variable changes its value, it communicates the new value to its neighbors, thus enabling the preceding step to be performed without communication. The variables iterate this optimize-communicate cycle indefinitely.

A dynamic version of AWC [Yokoo and Hirayama, 2000] called DynAWC. In DynAWC [Modi et al., 2001] agents annotate their nogoods with the current value of their variable when they are stored. This has a similar effect to the annotation done in centralized problem-solving. The key difference, however, is that because the constraints are not explicitly known to the agents, the technique is limited to only handling local constraint changes such as the addition or removal of a domain element. This is why the algorithm is called Locally Dynamic AWC [Modi et al., 2001].

Authors in [Mailler, 2005], have proposed two methods for dealing with DDisCSP: (i) DynDBA like in the static version [Yokoo and Hirayama, 2000], DynDBA works by making local adaptations to the variable values and is, therefore, an incomplete search technique. This means that DynDBA continuously works to find minimal conflicting solutions even when the problem is unsatisfiable. And (ii) a modified version of the Asynchronous Partial Overlay (APO)algorithm [Mailler and Lesser, 2004]. DynAPO is quite different from DynDBA in some ways. The most profound is that, like APO, it is a complete search technique and as such can report when it has determined that a problem is unsatisfiable. Because of this, DynAPO often enters a period of quiescence when the problem is satisfied or has been determined to be unsatisfiable. The side effect of this, though, is that DynAPO lacks the pure minimization flavor of DynDBA. Based on their experiment, they concluded that The results of the tests were mixed, with neither algorithm being completely dominant. This seems to indicate that determining which algorithm is best largely depends on the dynamics of the environment, the problems overall difficulty, and a user's particular needs.

Dynamic Asynchronous Backtracking (DynABT) has been proposed in [Omomowo et al., 2008] for the same purpose which combines solution and reasoning reuse i.e. it handles problem changes by modifying the existing solution while re-using knowledge gained from solving the original(unchanged). Experimental results show that DynABT requires fewer messages and constraint checks than DynDBA. However, the latter produces more stable solutions. In the next section, we will show and describe more deeply the procedures of DynABT.

8.3.1 Dynamic Asynchronous Backtracking

Dynamic Asynchronous Backtracking (DynABT) is an asynchronous algorithm for DDisCSPs which is complete and does terminate [Omomowo et al., 2008]. It is based on ABT, while the main difference is that it is designed to treat dynamic problems by repairing existing solutions such as the MPP approaches when the problem changes. The approach treats episodic constraints changes, i.e., changes (additions and removals) occurring after each problem has been solved. DynABT uses the previous solution of the initial problem, and the knowledge that agents had e.g Agentview, Nogood_Store, etc, and the constraint stored in the nogoods to detect and remove the obsolete nogood when a constraint is removed.

When detecting inconsistency, the agent composes a nogood of the form

$$x_i = a \cap x_j = b \; \{C_1, ..C_n\} \Rightarrow x_k \neq c. \tag{8.1}$$

Thus, the justification ($\{C_1, ..C_n\}$) included in the nogoods indicates which nogoods should become obsolete when modifying some constraints.

DynABT agents maintain, like ABT, a list of higher priority agents and their values in their *Agentview* and a list of values that are inconsistent with their *Agentview* in the nogood store. Higher priority

agents send, info_messages including their value assignments to lower priority agents. When an info_messages is received, the agent updates its Agentview and checks for consistency.

Algorithm 17: Dynamic Asynchronous Backtracking Algorithm

Procedure DynABT()
1. changes ← 0;changeBox ← empty;canProceed ← true
2. ABT$^+$(ABT with nogoods containing justifications)
3. **repeat**
4. changes ← monitorChanges
5. **if**(changes) **then**
6. canProceed ← false
7. PropagateChanges(changeBox)
8. current value← value from the last solution
9. ABT$^+$()
10. **end if**
11. **until** termination condition met

Procedure PropagateChanges(changeBox)
12. **while** changeBox ≠ empty ∩ canProceed ← false **do**
13. con ← getChange;changeBox ← changeBox − con
14. Swich(con.msgType)
15. con.removeConstraint : removeConstraint(con);
16. con.addConstraint : includeConstraint(con);
17. con.adjustNogood : incoherentConstraint(con);
18. **end while**

Procedure IncludeConstraint(con)
19. newCons ← con.getConstraint()
20. add new neighbours in newCons to neighbour list
21. constraintList ← constraintList ∪ newCons

Procedure ExcludeConstraint(con)
22. incoherentConstraint(con)
23. constraint ←con.getConstraint()
24. Remove unique neighbours in constraint
from neighbour list
25. Delete unique neighbours from agentView
26. Remove constraint from constraintList

Procedure AdjustNogoods()
27. IncoherentConstraint(con)
28. constraint ←con.getConstraint()
29. **for** each nogood in nogoodStore **do**
30. **if** contains(nogood,constraint) **then**
31. return eliminated value in nogood to domain
32. remove nogood from nogoodStore
33. **end if**
34. **end for**

In DynABT Algorithm.17, each agent initializes its variables, starts the search, and solves the problem, like in ABT. Nevertheless, agents oversee the system to react when detecting changes. Problem changes are handled in two phases: the Propagation phase (lines 12-18) and the solving phase (line 9). In the propagation phase, agents are informed of constraints addition/retraction, then, they update their constraint lists, neighbour lists, Agentview and nogoods where necessary. AddConstraint, RemoveConstraint, and AdjustNogood messages are used during this phase to handle agents' behavior. After propagating all changes, the new problem is defined, the canProceed flag is set to true and the agents can move on to the Solving phase in order to solve the new problem similar to the ABT algorithm.

When an agent receives an addConstraint message (lines 19-21), it updates its constraint list and neighbour list where necessary.
When receiving a removeConstraint message (lines 22-26), the agent modifies its neighbour list by removing from its neighbour list and its Agentview neighbors that only share the excluded constraint, then, the constraint is removed.

When a constraint is removed, an adjustNogood message (lines 27-34) is broadcasted to agents that are not directly involved in this constraint. The agents receiving this message update their nogoods store: they remove the nogoods containing the rejected constraint in justification and return the values to their domains since their source of inconsistency is no longer present in the network. Thus, the new problem starts at a consistent point before beginning the Solving phase.

All the above approaches are classically used to solve Distributed CSP, but there are adapted to make them suited for dynamic problems by:

— adding procedures: especially those to respond to events of adding/ removing constraint.

— annotating the nogoods that are found with constraint justifications. Nogoods that are still valid can be reused in future problems, while nogoods that have become invalid are no longer checked [Schiex and Verfaillie, 1994].

8.4 Conclusion

In this chapter, we introduced the concept of Dynamic Distributed CSP, and we presented briefly some related works on this field of Constraint Programming, then we presented more deeply one of them i.e. DynABT. As DynABT is two phases approach, it requires a batch processing resolution. A group of changes is collected and processed, over a period, after finding a solution or detecting the unsolvability of the initial problem. In contrast, many situations require immediate actions to act within the few seconds or minutes following the changes (e.g., flight delay management in a distributed air traffic control system, Target tracking problems). In the next chapters, we will present and describe a new approach to solve Dynamic Distributed Constraint Satisfaction Problems in real-time which we called (LiveABT).

CHAPTER

9

Repairing Protocol for Incremental Dynamic DisCSPs

This chapter proposes a new approach, called LiveABT*, to solve Dynamic Distributed Constraint Satisfaction Problems. This new approach is based on the Asynchronous Backtracking (*ABT*) algorithm and real-time processing techniques that run the solver immediately against live perturbations. We evaluate our approach and compare it to the Dynamic Asynchronous Backtracking (*DynABT*), one of the latest algorithms proposed in the context of* DDisCSP*, on various problems kinds. The results show that* LiveABT *outperforms significantly* DynABT *especially for the problems much more perturbed.*

Contents

9.1 LiveABT description

LiveABT is a new behavior of ABT, which supports three more messages types, to treat dynamism of DisCSPs. The three new messages, AddConstraint, RemoveConstraint and AdjustNogood, are the same of those used in DynABT. However, they are applied in a different way, with the aim of considering the perturbations in real time.
As known, ABT is a distributed algorithm, i.e., variables are distributed among multiple self-acting agents: each agent runs by himself while managing only the context where it is involved. Therefore, to make ABT supporting changes, it is not necessary when injecting a perturbation to stop the whole multi-agent system from updating the network, as DynABT does, since the perturbation does not impact all agents directly.

Thus, as all network communications are insured by messages, such as each agent, when receiving a message, reacts appropriately (e.g., receiving an Ok message is followed by an *update Agent View* for the receiver agent, and receiving a *nogood* message is systematically treated by *change value*), handling perturbations in ABT can be simply integrated into this communication protocol.
Then, to incorporate the AddConstraint or RemoveConstraint messages, the treatment can be done in the *solving* phase exactly as for any other message.
Therefore, in LiveABT, a *propagation* phase is not required. A perturbation injection is reflected by the immediate treatment of a message sent from the system, which is generally the master. This treatment consists of two parts: supporting the perturbation locally and propagating it in the network.

9.2 LiveABT versus DynABT

As shown in Fig.9.1, DynABT is executed in two major phases, the solving phase to solve the problem and the *propagation* phase to update it. Although injecting a perturbation at a time t, DynABT continues solving the initial problem as before, but stores the perturbation for future treatment. After solving the initial problem (or detecting the unsolvability), DynABT propagates the perturbations in the *propagation* phase. Then, it restarts the *solving* phase to repair the last found solution (or last assignment).
LiveABT, differently to DynABT, is executed in one phase: the *solving* phase. Contrary to DynABT, whose *solving* phase treats a problem that is entirely initially defined, LiveABT has the advantage of supporting perturbations in its native behavior. Thus, when detecting a perturbation, this one is considered and propagated immediately, and the resolution of the whole problem (the problem after perturbation) is kept on. The propagation is done in the *solving* phase which is continuously running.

9.3 Simulation example

Let us consider the example of a binary DDisCSP containing 4 agents: A1, A2, A3 and A4. Each agent has only one variable, respectively X1, X2, X3 and X4. The problem constraints are as shown in Fig.9.2 :

9.3.1 Perturbation 1: A constraint addition

When a perturbation affects the problem, such as a new constraint must be added into the network (C_{23} between the agents A2 and A3 as shown in Fig.9.3).

The perturbation system manager sends an add constraint message $AC(A2, C_{23}, stage)$ at the initial stage (*stage*=0) to the higher priority agent concerned by the constraint (the agent A2). This message contains the address of the second part of the constraint (the agent A3) since A2 ignores A3, which is not yet in its neighbors, and the status of injecting the constraint which is incremented in each step. When A2 receives the $AC(A2, C_{23}, stage)$, it adds A3 in its neighbours list, puts C_{23} in its constraints list and sends the $AC(A3, C_{23}, stage{+}{+})$ message to A3. When A3 receives the *add constraint* message,

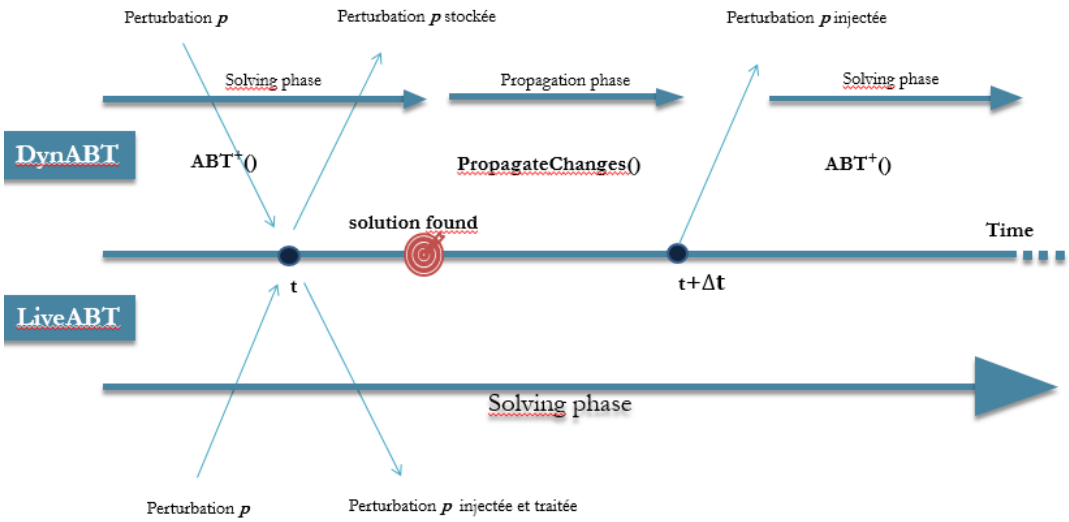

Figure 9.1 – behavior of LiveABT vs DynABT

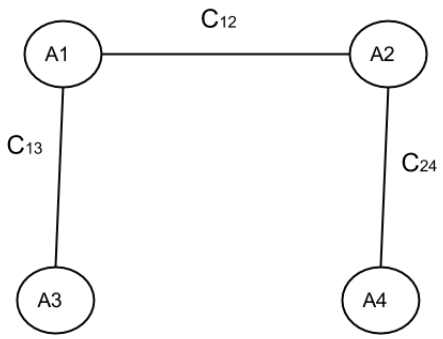

Figure 9.2 – Example of a DisCSP Network

it adds A2 to its neighbours, updates its *agentView*, puts the new constraint C_{23} in its constraints list and informs A2 that the constraint addition is finished by sending to it the $AC(A2, C_{23}, stage++)$ message. A2 reacts by sending an OK message to A3, to evaluate the new constraint and change its value if necessary. At the end of the *add constraint* process, the network becomes like Fig.9.4.

Thus, to add one constraint into the network, 4 messages are needed.

9.3.2 Perturbation 2: A constraint suppression

Removing a constraint. If the coming perturbation is a constraint suppression, LiveABT removes the constraint, then, differently, to the previous case, it broadcasts the AdjustNogood message to clean the agent's nogood stores by removing any nogood containing removed constraint as justification.

When A1 receives a remove constraint message ($RC(A1, C_{13}, 0)$) as Fig.9.5 indicates, it removes A3 from its neighbors list, removes C_{13} from its constraints list and sends the $RC(A3, C_{13}, stage++)$ message to A3. The agent A3 also removes A1 from its neighbors list, updates its *AgentView*, removes C_{13} from its constraints list and sends the $RC(A1, C_{13}, stage++)$ message to A1, informing to it that the

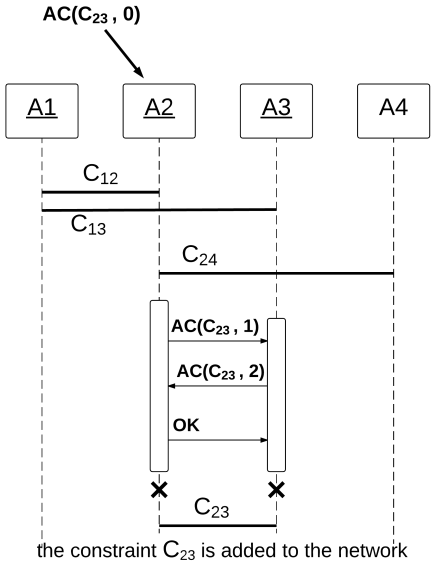

Figure 9.3 – Sequence diagram of constraint addition

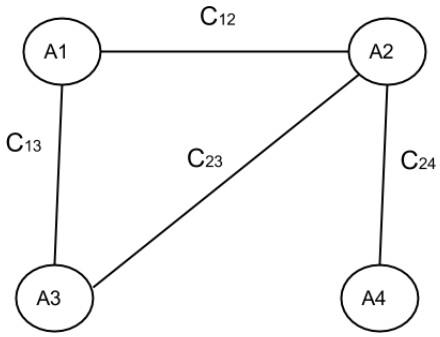

Figure 9.4 – The problem network after constraint addition

propagation of removing the constraint is finished. Then, A1 forwards this information to the system, generally the master, by the $RC(system, C_{13}, stage++)$ message. Therefore, to remove one constraint from the network, 4 messages are needed.

Updating the nogood stores. Each agent, when receiving the AdjustNogood message (AdjNg C_{13} as shown in Fig.9.6), checks his nogood store, tries to clean it, and updates his domain. Thus, in this example, A2 removes the C_{13} justification from its nogood. However, it keeps the nogood, as C_{13} is not the sole justification. A3 restores 1 in his domain and cleans his nogood store (deletes the first nogood). Moreover, A4 keeps his nogood store after the check.

To clean all the nogood stores, n messages, such as n is the number of agents in the network, are needed.

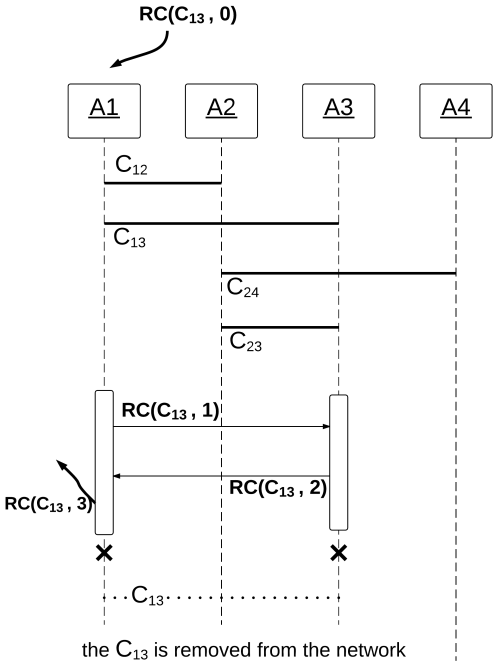

Figure 9.5 – Sequence diagram of constraint removal 1/2

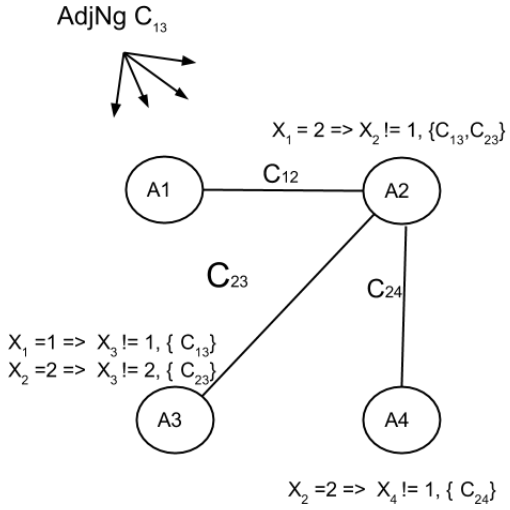

Figure 9.6 – Constraint removal 2/2

9.3.3 LiveABT Execution

The description of LiveABT is shown in Algorithm.18 and Algorithm.19. The LiveABT kernel is the main procedure (lines 1-11). It is implemented in each agent.

Algorithm 18: LiveABT Procedures 1/2

```
Procedure LiveABT_kernel()
1.  compute Γ+(self) and Γ−(self);
2.  CheckAgentView(null);
3.  end ← false;
4.  while (¬end) do
5.     msg ← getMsg();
6.     switch (msg.type)
7.        Stop : end ← true;
8.        AddConstraint : AC(con.getUpper,con,0);
9.        RemoveConstraint : RC(con.getUpper,con,0);
10       Adjustngood : IncoherentConstraint(con);
11.      else : CheckAgentView(msg);
Procedure CheckAgentView(msg)
12. if msg is relevant then
13.    update local context ;
14. if Consistent(myValue, myContext) then Return;
15. myValue ← ChooseValue();
16. if myValue then
17.    for each child ∈ Γ+(self) do
18.       sendMsg : Info(child, myValue);
19. else Backtrack;
Procedure Backtrack()
20. newNogood ← Solve(myNogoods) ;
21. if newNogood = empty then
22.    end ← true ;
23.    sendMsg : Stop(system);
24. else
25.    sendMsg : Back(newNogood);
26.    update local context;
27.    CheckAgentView(null);
Function ChooseValue()
28. for each v ∈ D(self) not eliminated by myNogoods do
29.    if Consistent(v, myContext) then Return (v) ;
30.    else
31.       Add(X=valX ⇒ ¬ v;C{ci} , myNogoods);
32. Return (null) ;
```

Each agent detects its parents (AgentView) and its children (line 1) and listens to the network. If the agent receives a CheckAgentView message, it launches the CheckAgentView function (lines 12-19). If the message comes from a higher priority agent, the receiver agent updates its agentView (lines 12-13). If its value is not consistent (lines 14-15), it calls the ChooseValue function (lines 28-32) to instantiate its variable. If the agent's variable is instantiated, the agent sends an Info message to its lower priority agents, informing them to update their agent views (lines 16-18). Else, the Backtrack is launched, to inform the value−sending agent to change its value, due to a domain wipe out (lines 20-27).
Throughout, each agent can receive an AddConstraint or RemoveConstraint perturbation message (lines 8-9).

If an agent receives the AddConstraint message (line 8), LiveABT reacts by calling the AddConstraint process (lines 33-40), which updates all information about the two agents involved in the constraint, and integrates it into the network. stage [1] represents the progress of adding (or removing) a constraint. The AC and info messages are exchanged between agents concerned by the

1. stage is a counter related to each perturbation, it is used by the agent to determine what action must be executed because one agent can be involved in many perturbations

added constraint to update their contexts and to activate the new constraint in the network.

Algorithm 19: LiveABT Procedures 2/2

```
Procedure AddConstraint(con)
33. switch (stage)
34.    0 : Add con.neighbour to my Neighbours List;
35.       Add con to Constraints List;
36.       sendMsg : AC(con.getlower,con,stage++);
37.    1 : Add con.neighbour to Neighbours List;
38.       Add con to Constraints List;
39.       sendMsg : AC(con.getupper, con, stage++);
40.    2 : sendMsg : Info(con.getlower(), myvalue);
Procedure RemoveConstraint(con)
41. IncoherentConstraint(con);
42. switch (stage)
43.    0 : Remove con.neighbour from Neighbours List;
44.       Remove unique neighbour from AgentViews;
45.       Remove con from Constraint List;
46.       sendMsg : RC(con.getlower, con, stage++);
47.    1 : Remove con.neighbour from Neighbours List;
48.       Remove unique neighbour from AgentView ;
49.       Remove con from Constraint List;
50.       sendMsg : RC(con.getupper, con, stage++);
51.    2 : sendMsg : RC(system, con, stage++);
Procedure IncoherentConstraint(con)
52. for each nogood in nogoodStore do
53.    if nogood.justification.Contains(con) then
54.       nogood.justification.Remove(con);
55.       if nogood.justification.isEmpty() then
56.          restore nogood.eliminatedValue to domain;
57.          nogoodStore.Remove(nogood);
```

If a constraint is removed (line 9), LiveABT proceeds to remove it using RemoveConstraint (lines 41-51). As for AddConstraint, RC messages are exchanged between concerned agents to accomplish the removal. Once the constraint is deleted, an AdjustNogood message is broadcasted to clean all the stored nogoods that contain the removed constraint as unique justification (lines 52-57), in order to return the removed values to their domains.

IncoherentConstraint ensures that the invalidated values by removed constraints are restored to their domains.

The Stop message (line 7) is broadcasted from the master to all the network agents, with success, when a quiescence is detected, if a solution is found, or with a failure, if an empty nogood is generated.

9.3.4 Termination and Completeness

The termination of LiveABT is assured since ABT terminates [Yokoo, 2012]. LiveABT respects the paradigm and uses its same mechanism to communicate all the pop-up perturbations in the network (Add/Remove Constraint messages become like ok or nogood messages), therefore, the approach terminates. Concerning the completeness, if the final problem (after the perturbations) has a solution, LiveABT finds it. In fact, after integrating all the perturbations, LiveABT protocol becomes the standard ABT algorithm, which is complete [Yokoo, 2012].

9.4 Experiments

We choose to compare experimentally LiveABT to DynABT since it is the latest DDisCSP approach. Furthermore, DynABT has already proved its large outperformance compared to DynDBA [Mailler, 2005] in terms of runtime and network load. Algorithms are evaluated on uniform random binary DDisCSPs, on Graph Coloring Problems, and also on Meeting Scheduling Problems.

To simulate a quantity p of perturbations like in reality, we choose to divide them over a various q randomly (standard normal distribution) moments of injections $[t_1, t_q]$, which are generated when the resolution is running, and in each moment $t_i \in [t_1, t_q]$, the problem is disturbed with $\frac{p}{q}$ perturbations. Different ways to disturb problems exist, and as problems density is fixed, we choose the same scenario of perturbation presented in [Omomowo et al., 2008] that keeps density intact. Thus, perturbations are divided into two parts of $\frac{p}{2\times q}$% of added constraints and $\frac{p}{2\times q}$% of removed constraints. Then, the initial problem and the perturbed one have the same density. For example, let's consider a problem with 60 constraints and we have p=10% is the rate of perturbation, based on the chosen scenario, 6 constraints will be changed 3 will be removed, and 3 will be added. We randomly generate q moments (injection moment). The distribution can be as $q = 3$ (1,1,1) three-time injections was generated, for each time one constraint will be added and one other will be removed, or $q = 2$=(1,2), i.e., for the first time one constraint will be added and one other will be removed and at the second time two constraints will be added/removed, we can have also q= 1(3), i.e., all the perturbations occur at one moment.
All experiments are performed on the Dischoco platform [Wahbi et al., 2011]. We evaluate the performance of the algorithms by the total number of exchanged messages among agents (#Msg) [Lynch, 1996] and non-concurrent computation. The non-concurrent computation is measured by the number of non-concurrent constraint checks
(#NCCCs) [Meisels, 2008].

9.4.1 Experiments on Random Problem

A binary random DisCSPs are characterized by $< n, d, p_1, p_2 >$, where n is the number of agents/-variables, d the number of values in each domain, p_1 the network connectivity that is defined as the ratio of existing binary constraints to possible binary constraints, and p_2 the constraints' tightness, that represents a proportion of the conflicts within the constraints.

A uniform random binary DDisCSP P is a sequence of uniform random binary DisCSPs: $P_0, P_1, ..., P_{\alpha-1}, P_\alpha$. The problem P_0 is the original DisCSP, and for each problem P_i, new constraints are added/removed to/from the previous one. In these experiments, we assume that the rates of perturbations are respectively 6% that represents an environment with lower perturbations and 30% witch represents the oscillated one.

Tests are performed on random problems from <20,10,0.2,0.1> to
<20,10,0.2,0.9>[2] with a p_2 step of 0.1.

For each fixed tightness p_2, 50 different instances are disturbed randomly and treated by each approach. The presented results are the average of the 50 runs.

Problems with constraints tightness between 0.1 and 0.6 are solvable, problems with a constraints' tightness equal to 0.7 are a mixture of solvable and unsolvable problems, and beyond 0.7, problems are unsolvable.

The results show that LiveABT outperforms DynABT, for both NCCCs and Msgs (Figs. 9.7 and 9.8), regardless of the problems solvability, especially when perturbations are important (Fig.9.8).

9.4.2 Experiments on Graph Coloring

A distributed graph-coloring problem is characterized by three parameters, i.e., the number of agents/variables n, the number of colors of each agent k, and the number of links (constraints) between agents m.

this problem aims to assign to each agent (node of the network) a color where two connected agents will have different colors.

We randomly generated problems with 50 agents/variables and m arcs by the method described in [Minton et al., 1993b]. The problems are connected and have a solution, the setting of k=3 and

2. we performed tests also on problems from <30,12,0.7,0.1> to <30,12,0.7,0.9>. The results showed the same improvement

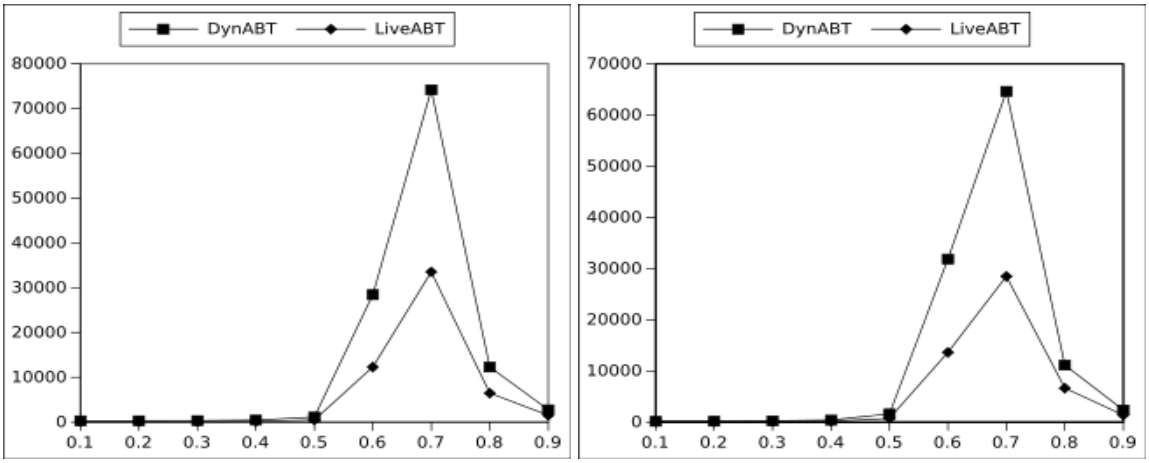

Figure 9.7 – Random Problem: Number of sent messages and NCCCs for 6% of perturbation

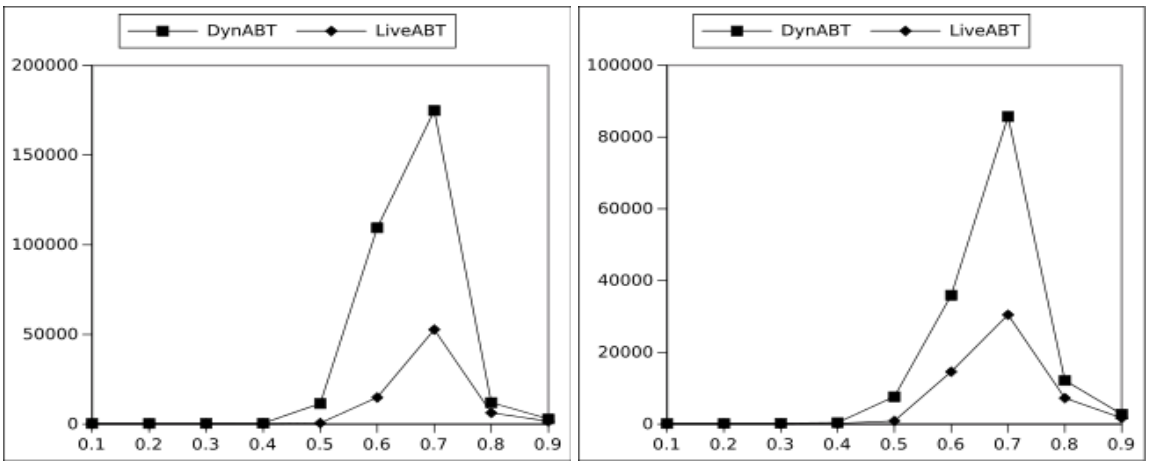

Figure 9.8 – Random Problem: Number of sent messages and NCCCs for 30% of perturbation

m=n*2.7 is the critical area [Cheeseman et al., 1991] i.e., the area where problems are difficult. We study the behavior of both LiveABT and DynABT when the perturbation rate increases.

For each rate of perturbation, 50 different instances are disturbed randomly from 2% to 40% by a step of 2% and treated with each approach. The presented results are the averages of these 50 runs.

The results in Fig.9.9 show that DynABT outperforms slightly LiveABT when the perturbations are low. However LiveABT makes less non-concurrent constraints checks (NCCCs) and sends less messages than DynABT in middle and high perturbations.

We also note that when the problem is enough disturbed, LiveABT deploys less effort to find a solution, contrary to DynABT for which the effort is linearly increased according to the perturbations rate.

9.4.3 Experiments on Meeting Scheduling Problems

The Dynamic Distributed MSP (DynDisMSP) can introduce two different types of changes:

— Additional Arrival Time constraint (traffic jams, train rescheduling, etc.)

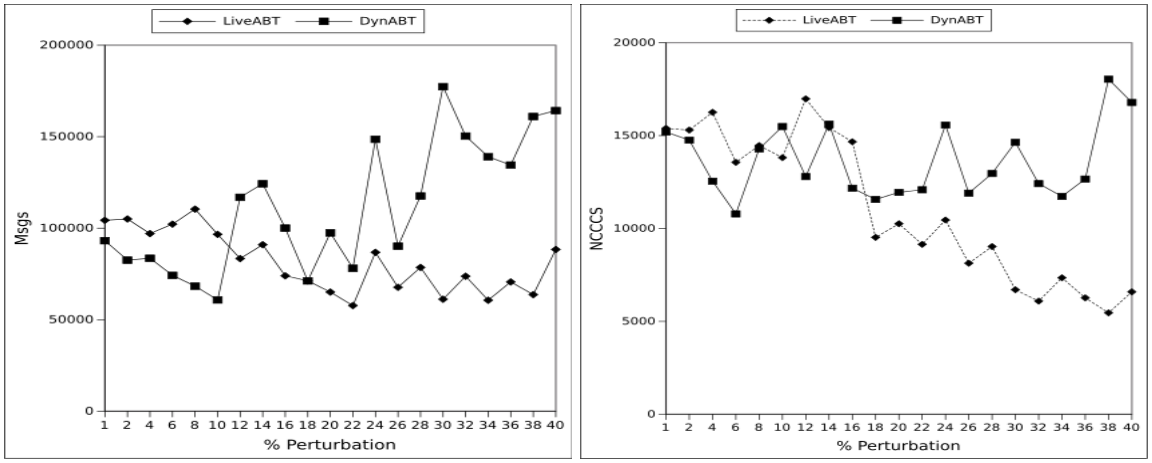

Figure 9.9 – Graph Coloring Problem: behavior of LiveABT and DynABT according to the rate of perturbation

— New links between meetings previously unconnected (a participant of meeting m_i is required to attend another meeting m_j as well).

Our experimental evaluation of DynDisMSP introduces the first type of change.

We generated a set of 50 random instances of DisMSP in order to compare the performances of LiveABT and DynABT, in terms of NCCCs and MSGs.

Distributed Meeting Scheduling Problem is characterized by $< n, m, k, d, h, a >$, where n is the number of agents, m the number of meetings, k the number of meetings per agent, d the number of days and h the number of hours per day, and a is a percentage of availability for each participant.

We present our results for the class $< 20, 4, 11, 3, 10, 1, 80 >$ and we vary the rate of perturbation constraints (i.e. Arrival Time Constraint) from 2% to 30%. We generated 50 different instances solved by each algorithm and the results are the averages of these 50 runs.

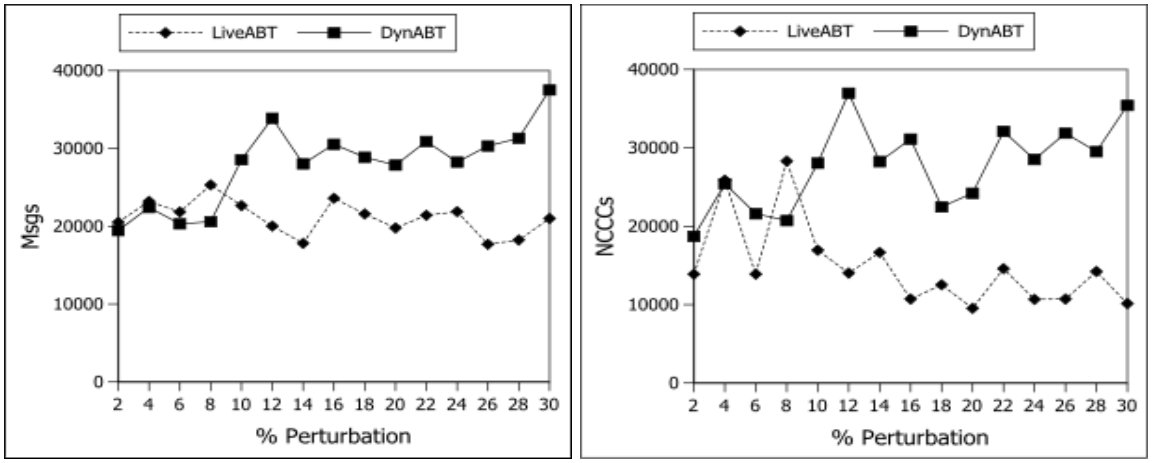

Figure 9.10 – Meetings Scheduling Problem: behavior of LiveABT and DynABT according to the rate of perturbation

The results in Fig.9.10 show that when the perturbation rate is low, DynABT and LiveABT have the same behavior in terms of checking constraints and sending messages, however, when the perturbation rate increases (from $\geqq 10$), LiveABT sends less messages and checks less constraints compared to DynABT.

9.4.4 Discussion

Firstly, for all perturbation rates on the random problems, namely 6%, and 30%, the results show that LiveABT makes less NCCCs and sends fewer messages than DynABT to treat a problem. LiveABT needs 4 messages when injecting a constraint, and n_A+4 messages for a constraint removal, where n_A is the number of agents in the network. However, in addition to this quantity (n_A+4+4) to support a perturbation (one addition and one removal), DynABT needs a larger number of broadcasted messages to stop and restart the resolution ($2\times n_A$), to propagate the perturbation in the network.

Concerning the number of NCCCs, when the DynABT solving phase is restarted to repair the solution, each agent launches the ABT process, i.e., evaluates its constraints, thus, DynABT checks at least (if the perturbation has no effect on the current solution) n_C constraints more than LiveABT, such as n_C is the number of constraints in the problem.

Secondly, the behavior of LiveABT and DynABT on the meetings scheduling and graph coloring problems shows that when the problem is more disturbing, the effort of LiveABT decreases. This is due to the impact of the constraint removal, which is much more important than adding a constraint. More precisely, when removing a constraint, all the agents are impacted, and they should clean their nogoods and recover the wasted values, however adding a constraint in real-time effects generally only the two involved agents.

Thus, if the problem is disturbed in real-time, it is relaxed during resolution.

Furthermore, the effort of DynABT increases when the perturbation rates increase because DynABT cumulates the efforts of the successive solving phases.

Finally, the major drawback of DynABT is the propagation phase, in which the algorithm propagates the perturbations, therefore, for the small perturbations (a bit of propagation phase), in Fig.9.9 and Fig.9.10, DynABT is a serious competitor for LiveABT.

9.5 Conclusion and perspectives

In this chapter, we introduced a new repair protocol for DDisCSP, based on ABT approach, to solve the problems, which are continuously disturbed in real-time. Based on the results of experiments on different rates of disturbance over different problem kinds, we showed that this protocol exceeds DynABT, regardless of problems solvability. Also, we demonstrate by studying the LiveABT behavior that in real-time, the removal of constraints has a global impact contrary to the constraints' addition.

CHAPTER

10 Conclusion

Constraint Programming is a powerful AI formalism that can describe many real-life problems and solve them. In this thesis, we have described some real-life problems and showed that they can be smoothly designed like a Constraint Problem. Also, many algorithms based on chronological backtracking algorithms have been presented. The performance of these algorithms can be improved by combining intelligent backtracks, propagation of information as constraints, and finally using variables/values ordering heuristics.
We began this thesis by studying a real case NP problem of allocating internships of the hospital services, which we have modeled as CSP. Then to fix some limitations (unsatisfiability) of the CSP model, we extended our model to COP one. Based on the COP model, We showed and demonstrated empirically that the concerned problem can have an acceptable solution although being unsatisfiable. Additionally, we proposed more optimal solutions, which optimize the service's preferences parameter, while maintaining a very acceptable solution. We also gave the Pareto frontier i.e. some trade-off between the two parameters, with no way to improve the first parameter without losing at the second.

Dynamic Constraint Satisfaction Problems (DCSPs) have been proposed to represent problems that change over time. It's very interesting to repair an established solution against resolving from scratch the whole perturbed problem. To effectively repair Dynamic CSPs, EPDB uses ordering heuristics and retroactive data structures, *safety conditions* and *nogoods*, which are saved during the search process.
We have introduced a new variable ordering heuristic (VOH), Profound degree (*pdeg*) heuristic based on classical deg heuristic, for estimating the influence of variables in the whole constraint network, in the process of repairing solutions in dynamic environments. The idea of *pdeg* heuristic is to disturb only the parts of the constraints network, whose probability of influence is minimal.
Based on the results of experiments on a wide class of problems, we found and proved that this heuristic, used in DCSPs correction, exceeds the heuristics deemed to be efficient when solving CSPs.
Afterward, we introduced IPDB, which improves EPDB, to repair solutions of Dynamic CSPs rapidly, using the data structure *successors*.
Based on the results of experiments on a variety of problems classes, we show that IPDB is a great improvement of EPDB, respecting runtime, insofar as it minimizes remarkably the tests of orders.
We showed that this approach is mainly efficient when using *deg* and that it is a good concurrent for MAC-2001.

Because many real problems are naturally distributed, we have dedicated an important part of this thesis to studying distributed Constraint problem. We began by introducing the concept of Distributed CSP, then Dynamic Distributed CSP, and we presented some real examples dealt with theses formalisms and some of the popular approaches to solve DisCSP. After that, we presented the problem

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

As perspectives of these works, concerning internships planning, we intend to integrate all the hospital resources (doctors, nurses, rooms, medical equipment, patients, etc.) to produce the optimized planning of human and material resources.

We plan to implement a CSP solver, based on IPDB, which benefits from IPDB advantages and resolves by repairing a complete assignment that violates a minimum number of constraints, in order to converge more rapidly to a solution. We also plan to develop a version of IPDB for minimal perturbation problems [Zivan et al., 2011], to preserve solutions of DCSPs as much as possible. We intend likewise to integrate constraint propagation within IPDB, to benefit from the advantage of the two strategies regardless of the problem's satisfiability.

We intend also, to develop LiveABT agent to detect autonomously the different agents' perturbations (address changing, disconnection, or addition of agents) and react appropriately without waiting for the Master instructions, and we also plan to upgrade and apply LiveABT in other realistic use especially distributed robotic control system and the smart grid problems.

Bibliography

Acodad, Y., Benamrane, A., Benelallam, I., and Bouyakhf, E. H. (2014). Profound degree: A conservative heuristic to repair dynamic csps. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 140–149. Springer. Cited page 59.

Acodad, Y., Benelallam, I., Hammoujan, S., and Bouyakhf, E. H. (2012). Extended partial-order dynamic backtracking algorithm for dynamically changed environments. In *Tools with Artificial Intelligence (ICTAI), 2012 IEEE 24th International Conference on*, volume 1, pages 580–587. IEEE. Cited pages 30, 34, 35, 39, 44, 54, and 59.

Al-Betar, M. A., Awadallah, M. A., Bolaji, A. L., and Alijla, B. O. (2017). β-hill climbing algorithm for sudoku game. In *2017 Palestinian International Conference on Information and Communication Technology (PICICT)*, pages 84–88. IEEE. Cited page 6.

Audemard, G., Boussemart, F., Lecoutre, C., and Piette, C. (2020). xcsp. Cited page 1.

Awerbuch, B. (1985). A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3):147–150. Cited page 8.

Bank, W. (2018). The world bank in morocco. Cited page 17.

Barták, R. (1999). Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS99)*, pages 555–564. MatFyzPress Prague. Cited page 5.

Barták, R., Müller, T., and Rudová, H. (2003). A new approach to modeling and solving minimal perturbation problems. In *International Workshop on Constraint Solving and Constraint Logic Programming*, pages 233–249. Springer. Cited page 30.

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons. Cited page 87.

Benamrane, A., Acodad, Y., Benelallam, I., El Houssine, B., and Mohammed, B. O. (2014). Modeling trainings in faculty of medicine and pharmacy of casablanca as constraint satisfaction problem. pages 14–20. CP 2014 : The Thirteenth International Workshop on Constraint Modelling and Reformulation. Cited pages 18 and 93.

Benamrane, A., Acodad, Y., Bouyakhf, E., and Benelallam, I. (2017). Liveabt: a real-time repairing protocol for incremental and dynamic discsps. *Int J Artif Intell*, 15:126–146. Cited page 74.

Benamrane, A. and Erraji, Z. (2018). Medical internships planning demo. Cited page 26.

Benelallam, I., Erraji, Z., Khattabi, G. E., and Bouyakhf, E. H. (2015). Dynamic jchoc: a distributed constraints reasoning platform for dynamically changing environments. In *International Conference on Agents and Artificial Intelligence*, pages 20–36. Springer. Cited page 87.

Bessiere, C., Brito, I., Maestre, A., and Meseguer, P. (2003). The asynchronous backtracking family. *LIRMM-CNRS,, Montpellier, France March*, 2003. Cited page 70.

Bessière, C., Freuder, E. C., and Régin, J.-C. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148. Cited page 13.

Bessiere, C., Maestre, A., Brito, I., and Meseguer, P. (2005). Asynchronous backtracking without adding links: a new member in the abt family. *Artificial Intelligence*, 161(1-2):7–24. Cited pages 72 and 73.

Bessiere, C., Maestre, A., and Meseguer, P. (2001). Distributed dynamic backtracking. page 772. Cited page 73.

Bessière, C., Maestre, A., and Meseguer, P. (2002). La famille abt. In *JNPC: Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets*, pages 57–67. Springer. Cited page 84.

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

Bessière, C. and Régin, J.-C. (2001a). Refining the basic constraint propagation algorithm. In *IJCAI*, volume 1, pages 309–315. Cited pages 13 and 14.

Bessière, C. and Régin, J.-C. (2001b). Refining the basic constraint propagation algorithm. In *IJCAI*, volume 1, pages 309–315. Cited pages 52 and 62.

Bessière, C., Régin, J.-C., Yap, R. H., and Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185. Cited pages 52 and 62.

Bitner, J. R. and Reingold, E. M. (1975). Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656. Cited page 9.

Bliek, C. (1998). Generalizing partial order and dynamic backtracking. *AAAI/IAAI*, 98:319–325. Cited pages 30, 51, and 62.

Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004a). Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146. Cited page 15.

Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004b). Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 146–150. IOS Press. Cited pages 33, 40, 52, and 62.

Brito, I. and Meseguer, P. (2008). Connecting abt with arc consistency. In *International Conference on Principles and Practice of Constraint Programming*, pages 387–401. Springer. Cited page 73.

Cheeseman, P., Kanefsky, B., and Taylor, W. M. (1991). Where the really hard problems are. In *IJCAI*, volume 91, pages 331–340. Cited page 105.

Costa Filho, C. F. F., Rocha, D. A. R., Costa, M. G. F., and de Albuquerque Pereira, W. C. (2012). Using constraint satisfaction problem approach to solve human resource allocation problems in cooperative health services. *Expert Systems with Applications*, 39(1):385–394. Cited page 17.

Crawford, J. M. and Auton, L. D. (1996). Experimental results on the crossover point in random 3-sat. *Artificial intelligence*, 81(1):31–57. Cited pages 52 and 62.

Davidson, E., Dolan, M., McArthur, S., and Ault, G. (2009). The use of constraint programming for the autonomous management of power flows. In *2009 15th International Conference on Intelligent System Applications to Power Systems*, pages 1–7. IEEE. Cited page 93.

De Oliveira, D., Bazzan, A. L., and Lesser, V. (2005). Using cooperative mediation to coordinate traffic lights: a case study. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 463–470. Cited page 93.

Dechter, R. and Dechter, A. (1988). *Belief maintenance in dynamic constraint networks*. University of California, Computer Science Department. Cited pages 29 and 30.

Dechter, R. and Meiri, I. (1989). *Experimental evaluation of preprocessing techniques in constraint satisfaction problems*. Citeseer. Cited pages 2 and 37.

Dechter, R. and Meiri, I. (1994). Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68(2):211–241. Cited page 15.

Dechter, R. and Pearl, J. (1988). Network-based heuristics for constraint-satisfaction problems. In *Search in artificial intelligence*, pages 370–425. Springer. Cited page 15.

El Graoui, E. M., Benelallam, I., and Bouyakhf, E. H. (2016). A commentary on" hybrid search for minimal perturbation in dynamic csps". *Constraints*, 21(2):349–354. Cited page 30.

El Sakkout, H., Richards, T., and Wallace, M. (1998). Minimal perturbation in dynamic scheduling. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98). John Wiley & Sons*. Citeseer. Cited page 30.

El Sakkout, H. and Wallace, M. (2000). Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5(4):359–388. Cited pages 30 and 34.

Eriksson, J., Finne, N., and Janson, S. (1998). To each and everyone an agent: augmenting web-based commerce with agents. In *Proceedings of the International Workshop on Intelligent Agents on the Internet and Web, Fourth World Congress on Expert Systems*. Citeseer. Cited page 80.

Fages, F. (2016). Ai in biological modelling. Cited page 93.

Fages, F. (2020). Artificial intelligence in biological modelling. In *A Guided Tour of Artificial Intelligence Research*, pages 265–302. Springer. Cited page 93.

Faltings, B. and Yokoo, M. (2005). Introduction: Special issue on distributed constraint satisfaction. *Artificial Intelligence*, 161(1-2):1–5. Cited pages 2 and 70.

Fatima, S. S., Wooldridge, M., and Jennings, N. R. (2007). Approximate and online multi-issue negotiation. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8. Cited pages 80 and 90.

Ferber, J. (1999). *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading. Cited page 93.

Fikar, C. and Hirsch, P. (2017). Home health care routing and scheduling: A review. *Computers & Operations Research*, 77:86 – 95. Cited page 17.

Fioretto, F., Pontelli, E., and Yeoh, W. (2018). Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61:623–698. Cited pages 1 and 70.

Fitzpatrick, S. and Meertens, L. (2003). Distributed coordination through anarchic optimization. In *Distributed Sensor Networks*, pages 257–295. Springer. Cited page 94.

Freuder, E. C. (1985). A sufficient condition for backtrack-bounded search. *Journal of the ACM (JACM)*, 32(4):755–761. Cited page 15.

Frost, D., Dechter, R., et al. (1995). Look-ahead value ordering for constraint satisfaction problems. In *IJCAI (1)*, pages 572–578. Citeseer. Cited page 15.

Ginsberg, M. L. (1993). Dynamic backtracking. *J. Artif. Int. Res.*, 1(1):25–46. Cited page 10.

Ginsberg, M. L. and McAllester, D. (1994a). Gsat and dynamic backtracking. In Doyle, J., Sandewall, E., and Torasso, P., editors, *International Workshop on Principles and Practice of Constraint Programming*, The Morgan Kaufmann Series in Representation and Reasoning", pages 226 – 237. Morgan Kaufmann. Cited page 10.

Ginsberg, M. L. and McAllester, D. (1994b). Gsat and dynamic backtracking. In *Principles and Practice of Constraint Programming*, pages 243–265. Springer. Cited pages 30 and 44.

Goldkamp, J. and Dehghanimohammadabadi, M. (2019). Evolutionary multi-objective optimization for multivariate pairs trading. *Expert Systems with Applications*, 135:113–128. Cited page 93.

Gür, Ş., Eren, T., and Alakaş, H. M. (2019). Surgical operation scheduling with goal programming and constraint programming: a case study. *Mathematics*, 7(3):251. Cited page 71.

Hamadi, Y., Bessiere, C., and Quinqueton, J. (1998). Backtracking in distributed constraint networks. In *Proceedings ECAI'98*, pages 219–223. Cited page 72.

Hammoujan, S. and Acodad, Y. (2013). Dynamic constraint ordering heuristics for dynamic csps. *Applied Mathematical Sciences*, 7(138):6889–6907. Cited pages 46 and 60.

Hannebauer, M. and Müller, S. (2001). Distributed constraint optimization for medical appointment scheduling. In *Proceedings of the fifth international conference on Autonomous agents*, pages 139–140. Cited page 93.

Haralick, R. M. and Elliott, G. L. (1980a). Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313. Cited pages 11 and 14.

Haralick, R. M. and Elliott, G. L. (1980b). Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313. Cited page 15.

Haralick, R. M. and Elliott, G. L. (1980c). Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313. Cited page 15.

Hebrard, E., Hnich, B., O'Sullivan, B., and Walsh, T. (2005). Finding diverse and similar solutions in constraint programming. In *AAAI*, volume 5, pages 372–377. Cited page 30.

Hebrard, E., O'Sullivan, B., and Walsh, T. (2007). Distance constraints in constraint satisfaction. In *IJCAI*, volume 2007, pages 106–111. Cited pages 30 and 40.

Hemaissia, M., El Fallah Seghrouchni, A., Labreuche, C., and Mattioli, J. (2007). A multilateral multi-issue negotiation protocol. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8. Cited page 80.

Hentenryck, P. V. and Michel, L. (2009). *Constraint-based local search*. The MIT press. Cited page 93.

Huang, K., Liu, X., Li, X., Liang, J., and He, S. (2013). An improved artificial immune system for seeking the pareto front of land-use allocation problem in large areas. *International journal of geographical information science*, 27(5):922–946. Cited page 24.

Junges, R. and Bazzan, A. L. (2008). Evaluating the performance of dcop algorithms in a real world, dynamic problem. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 599–606. Cited page 93.

Jussien, N., Debruyne, R., and Boizumault, P. (2000). Maintaining arc-consistency within dynamic backtracking. In *International Conference on Principles and Practice of Constraint Programming*, pages 249–261. Springer. Cited pages 52 and 62.

Kask, K., Dechter, R., and Gogate, V. (2004). Counting-based look-ahead schemes for constraint satisfaction. In *International Conference on Principles and Practice of Constraint Programming*, pages 317–331. Springer. Cited page 15.

Lai, G., Sycara, K., and Li, C. (2007). A pareto optimal model for automated multi-attribute negotiations. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–3. Cited pages 80 and 90.

Latorre-Núñez, G., Lüer-Villagra, A., Marianov, V., Obreque, C., Ramis, F., and Neriz, L. (2016). Scheduling operating rooms with consideration of all resources, post anesthesia beds and emergency surgeries. *Computers & Industrial Engineering*, 97:248–257. Cited page 17.

Lauer, C. (2002). Introducing microsoft dotnet. *Jul*, 2:4–5. Cited page 18.

Lynch, N. A. (1996). *Distributed algorithms*. Morgan Kaufmann. Cited page 104.

Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118. Cited pages 12 and 13.

Mailler, R. (2005). Comparing two approaches to dynamic, distributed constraint satisfaction. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1049–1056. Cited pages 94 and 103.

Mailler, R. and Lesser, V. (2004). Using cooperative mediation to solve distributed constraint satisfaction problems. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004.*, pages 446–453. IEEE. Cited page 94.

Marynissen, J. (2016). Literature review on integrated hospital scheduling problems. *SSRN Electronic Journal*. Cited page 17.

Mechqrane, Y., Wahbi, M., Bessiere, C., and Brown, K. N. (2020). Reordering all agents in asynchronous backtracking for distributed constraint satisfaction problems. *Artificial Intelligence*, 278:103169. Cited page 74.

Mehta, D. and van Dongen, M. R. (2005). Reducing checks and revisions in coarse-grained mac algorithms. In *International Joint Conference On Artificial Intelligence*, volume 19, page 236. Citeseer. Cited pages 52 and 62.

Mehta, D. and Van Dongen, M. R. (2007). Probabilistic consistency boosts mac and sac. In *IJCAI*, pages 143–148. Cited pages 52 and 62.

Meisels, A. (2008). Message delays and discsp search algorithms. In *Distributed Search by Constrained Agents*, pages 143–158. Springer. Cited page 104.

Meisels, A. and Lavee, O. (2004). Using additional information in discsp search. In *Proc. 5th workshop on distributed constraints reasoning, DCR*, volume 4. Citeseer. Cited pages 7 and 71.

Meisels, A., Shimony, S. E., and Solotorevsky, G. (1997). Bayes networks for estimating the number of solutions to a csp. In *AAAI/IAAI*, pages 179–184. Cited page 15.

Meisels, A. and Zivan, R. (2007). Asynchronous forward-checking for discsps. *Constraints*, 12(1):131–150. Cited page 73.

Ministry, M. H. (2010). La démographie médicale et paramedicale à l'horizon 2025. Cited page 17.

MiniZinc (2021). Minizinc. Cited page 1.

Minton, S., Johnston, M. D., Philips, A. B., and Laird, P. (1992). Minimizing conflicts: An heuristic repair method for constraint satisfaction and scheduling problems. *Artificial intelligence*, 58(1-3):161–205. Cited page 76.

Minton, S., Philips, A., Johnston, M. D., and Laird, P. (1993a). Minimizing conflicts: An heuristic repair method for constraint-satisfaction and scheduling problems. *Journal of Artificial Intelligence Research*, 1:1–15. Cited page 15.

Minton, S., Philips, A., Johnston, M. D., and Laird, P. (1993b). Minimizing conflicts: An heuristic repair method for constraint-satisfaction and scheduling problems. *Journal of Artificial Intelligence Research*, 1:1–15. Cited page 104.

Modi, P. J., Jung, H., Tambe, M., Shen, W.-M., and Kulkarni, S. (2001). Dynamic distributed resource allocation: A distributed constraint satisfaction approach. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 264–276. Springer. Cited page 94.

Mohr, R. and Henderson, T. C. (1986). Arc and path consistency revisited. *Artificial intelligence*, 28(2):225–233. Cited page 13.

Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information sciences*, 7:95–132. Cited page 12.

Omomowo, B., Arana, I., and Ahriz, H. (2008). Dynabt: Dynamic asynchronous backtracking for dynamic discsps. In *Artificial Intelligence: Methodology, Systems, and Applications*, pages 285–296. Springer. Cited pages 94 and 104.

Ortiz-Bayliss, J. C., Terashima-Marín, H., Özcan, E., Parkes, A. J., and Conant-Pablos, S. E. (2013). Exploring heuristic interactions in constraint satisfaction problems: A closer look at the hyper-heuristic space. In *2013 IEEE Congress on Evolutionary Computation*, pages 3307–3314. IEEE. Cited page 33.

Parunak, H. V. D., Ward, A. C., and Sauter, J. A. (1998). A systematic market approach to distributed constraint problems. In *Proceedings International Conference on Multi Agent Systems (Cat. No. 98EX160)*, pages 455–456. IEEE. Cited pages 80 and 88.

Perron, L. and Furnon, V. (2019). Or-tools. Cited page 1.

Precup, R., Preitl, S., and Korondi, P. (2007). Fuzzy controllers with maximum sensitivity for servosystems. *IEEE Transactions on Industrial Electronics*, 54(3):1298–1310. Cited page 93.

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299. Cited page 9.

Prosser, P. (1995). Mac-cbj: maintaining arc consistency with conflict-directed backjumping. Technical report, Technical Report Research Report/95/177, Dept. of Computer Science, University of Strathclyde. Cited pages 52 and 62.

Prud'homme, C., Fages, J.-G., and Lorca, X. (2017). *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. Cited page 1.

Rahwan, I., Kowalczyk, R., and Pham, H. H. (2002). Intelligent agents for automated one-to-many e-commerce negotiation. *Computer Science Preprint Archive*, 2002(7):782–788. Cited pages 80 and 90.

Ramchurn, S. D., Vytelingum, P., Rogers, A., and Jennings, N. R. (2012). Putting the'smarts' into the smart grid: a grand challenge for artificial intelligence. *Communications of the ACM*, 55(4):86–97. Cited page 93.

Ran, Y., Roos, N., and van den Herik, J. (2002). Approaches to find a near-minimal change solution for dynamic csps. In *Fourth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimisation problems*, pages 373–387. Citeseer. Cited page 30.

Régin, J.-C. (1994). A filtering algorithm for constraints of difference in csps. In *AAAI*, volume 94, pages 362–367. Cited page 13.

Ren, F., Zhang, M., and Sim, K. M. (2009). Adaptive conceding strategies for automated trading agents in dynamic, open markets. *Decision Support Systems*, 46(3):704–716. Cited page 80.

Rigi, M. A. and Khoshalhan, F. (2013). Eliciting user preferences in multi-agent meeting scheduling problem. In *Organizational Efficiency through Intelligent Information Technologies*, pages 108–124. IGI Global. Cited page 7.

Roos, N., Ran, Y., and Van Den Herik, J. (2000). Combining local search and constraint propagation to find a minimal change solution for a dynamic csp. In *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 272–282. Springer. Cited page 30.

Sabin, D. and Freuder, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. In *International Workshop on Principles and Practice of Constraint Programming*, pages 10–20. Springer. Cited page 11.

Sabin, D. and Freuder, E. C. (1997). Understanding and improving the mac algorithm. In *International Conference on Principles and Practice of Constraint Programming*, pages 167–181. Springer. Cited pages 14 and 37.

Salido, M. A. and Giret, A. (2008). Feasible distributed csp models for scheduling problems. *Engineering Applications of Artificial Intelligence*, 21(5):723–732. Cited pages 71 and 93.

Schiex, T. and Verfaillie, G. (1994). Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3(02):187–207. Cited pages 30 and 96.

Sen, S. and Durfee, E. H. (1998). A formal study of distributed meeting scheduling. *Group Decision and Negotiation*, 7(3):265–289. Cited page 7.

Silaghi, M.-C. and Faltings, B. (2005). Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161(1-2):25–53. Cited page 72.

Silaghi, M.-C., Sam-Haroud, D., and Faltings, B. (2000). Asynchronous search with aggregations. In *AAAI/IAAI*, pages 917–922. Cited page 72.

Smith, B. M. and Grant, S. A. (1997). Trying harder to fail first. *Research Report Series-University of Leeds School of Computer Studies Lu Scs Rr*. Cited page 15.

Solotorevsky, G. and Gudes, E. (1997). Solving a real-life nurses time tabling and transportation problem using distributed csp techniques. In *In Constraints and Agents: Papers from the 1997 AAAI Workshop*. Citeseer. Cited page 70.

Spyropoulos, C. D. (2000). Ai planning and scheduling in the medical hospital environment. *Artificial Intelligence in Medicine*, 20(2):101–111. Cited page 17.

Türkşen, Ö. and Tez, M. (2016). An application of nelder-mead heuristic-based hybrid algorithms: Estimation of compartment model parameters. *International Journal of Artificial Intelligence™*, 14(1):112–129. Cited page 93.

Vali-Siar, M. M., Gholami, S., and Ramezanian, R. (2018). Multi-period and multi-resource operating room scheduling under uncertainty: A case study. *Computers & Industrial Engineering*, 126:549–568. Cited page 17.

Van Hentenryck, P., Deville, Y., and Teng, C.-M. (1992). A generic arc-consistency algorithm and its specializations. *Artificial intelligence*, 57(2-3):291–321. Cited page 13.

Vedantham, S. and Iyengar, S. S. (1998). The bandwidth allocation problem in the atm network model is np-complete. *Information Processing Letters*, 65(4):179–182. Cited page 8.

Verfaillie, G. and Jussien, N. (2005). Constraint solving in uncertain and dynamic environments: A survey. *Constraints*, 10(3):253–281. Cited pages 2 and 29.

Vernooy, M. and Havens, W. S. (1999). An examination of probabilistic value-ordering heuristics. In *Australasian Joint Conference on Artificial Intelligence*, pages 340–352. Springer. Cited page 15.

Wahbi, M., Ezzahir, R., Bessiere, C., and Bouyakhf, E. H. (2011). Dischoco 2: A platform for distributed constraint reasoning. In *Proceedings of the IJCAI'11 workshop on Distributed Constraint Reasoning*, DCR'11, pages 112–121, Barcelona, Catalonia, Spain. Cited page 104.

Wallace, R. J., Grimes, D., and Freuder, E. C. (2009). Solving dynamic constraint satisfaction problems by identifying stable features. In *IJCAI*, volume 9, pages 621–627. Citeseer. Cited page 94.

Wegner, P. (1990). Concepts and paradigms of object-oriented programming. *ACM SIGPLAN OOPS Messenger*, 1(1):7–87. Cited page 18.

WHO (2018). Health statistics in morocco. Cited page 17.

Williams, R., Gomes, C. P., and Selman, B. (2003). Backdoors to typical case complexity. In *IJCAI*, volume 3, pages 1173–1178. Cited page 14.

Yokoo, M. (1995). Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 88–102. Springer. Cited page 72.

Yokoo, M. (2012). *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems*. Springer Science & Business Media. Cited pages 70, 73, and 103.

Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on knowledge and data engineering*, 10(5):673–685. Cited page 70.

Yokoo, M. and Hirayama, K. (1996). Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 401–408. MIT Press Cambridge, MA. Cited page 72.

Yokoo, M. and Hirayama, K. (2000). Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207. Cited pages 70, 72, and 94.

Yokoo, M., Ishida, T., Durfee, E. H., and Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 614–621. IEEE. Cited pages 72, 73, and 75.

Zhang, D., Mishra, S., Brynjolfsson, E., Etchemendy, J., Ganguli, D., Grosz, B., Lyons, T., Manyika, J., Niebles, J. C., Sellitto, M., et al. (2021). The ai index 2021 annual report. *arXiv preprint arXiv:2103.06312*. Cited page 1.

Zhang, W., Xing, Z., Wang, G., and Wittenburg, L. (2003). An analysis and application of distributed constraint satisfaction and optimization algorithms in sensor networks. In *AAMAS*, volume 3, pages 185–192. Cited pages 71 and 93.

Zhou, N.-F. and Kjellerstrand, H. (2016). The picat-sat compiler. In *International Symposium on Practical Aspects of Declarative Languages*, pages 48–62. Springer. Cited page 1.

Zivan, R., Grubshtein, A., and Meisels, A. (2011). Hybrid search for minimal perturbation in dynamic csps. *Constraints*, 16(3):228–249. Cited pages 30, 40, 68, and 109.

Zivan, R. and Meisels, A. (2006). Dynamic ordering for asynchronous backtracking on discsps. *Constraints*, 11(2-3):179–197. Cited page 73.

List of Figures

List of Tables

List of Algorithms

Résumé:

La programmation par contraintes (PC) est un des formalismes de l'intelligence artificielle pour traiter les problèmes combinatoires complexes qualifiés NP-complet. Le formalisme PC est un formalisme élégant qui sépare les processus de modélisation et celui de la résolution et à travers ses diverses variantes, il est capable de résoudre de nombreux problèmes réels. Dans cette thèse, nous étendons l'état de l'art de la programmation par contraintes en étudiant quelques cas pratiques et en proposant plusieurs algorithmes de résolution.

Nous commençons par modéliser un problème réel de planification i.e. la planification des stages hospitaliers du CHU de Casablanca pour résoudre et optimiser la distribution des étudiants en termes d'occupation des services ainsi que de préférences des futurs médecins. Puis, parce que de nombreuses applications réelles sont dynamiques, nous nous intéressons aux CSPs dynamiques, nous avons proposé une heuristique appelée Pdeg (Profond degree) qu'étend l'heuristique classique deg et qui exploite la profondeur du réseau de contraintes lors du choix de la variable à modifier. Puis nous avons expérimenté cette heuristique pour améliorer l'algorithme de réparation Partial-order Dynamic Backtracking PDB. Deux algorithmes IPDB et LRB ont été proposés pour réparer la solution pré-perturbée des CSPs dynamiques, rapidement en terme de temps, et efficacement en termes d'effort et de la qualité de la solution.

Toujours dans cette thèse, pour couvrir les CSPs dynamiques distribués, nous proposons d'une part une nouvelle approche basée sur les systèmes multi-Agents pour modéliser l'environnement du marché autonome intelligent (vente et achat de biens) et pouvoir négocier dans les e-marchés. Ensuite basé sur le célèbre algorithme ABT, nous avons proposé l'algorithme LiveABT pour traiter les perturbations dans les CSPs distribués en temps réel. Les expérimentations ont montré que cet algorithme surpasse la version de réparation séquentiellement réalisée par l'algorithme DynABT.

Mots-clés : *Intelligence Artificielle, Programmation par contraintes, Systèmes Multi Agents, Problèmes de satisfaction de contraintes (CSP), CSP Distribuées Dynamique, Modélisation, Planification, Optimisation.*

Abstract:

Constraint programming (PC) is one of the formalisms of artificial intelligence to deal with complex combinatorial problems qualified as NP-complete. The CP formalism is an elegant formalism that separates the modeling and the solving processes and through its various variants, it is able to solve many real problems. In this thesis, we extend the state of the art of constraint programming by studying some practical cases and proposing several resolution algorithms.

We start by modeling a real planning problem, i.e. the planning of hospital internships at the CHU of Casablanca to resolve and optimize planning in terms of service occupancy as well as the preferences of future doctors. Then, because many real applications are dynamic, we are interested in dynamic CSP, we proposed an heuristic called Pdeg (Profond degree) which extends the classic heuristic deg and which exploits the depth of the constraint network when choosing the variable to modify. Then we experimented with this heuristic to improve the Partial-order Dynamic Backtracking PDB repair algorithm. Two algorithms IPDB and LRB have been proposed to repair the pre-disturbed solution of dynamic CSPs, quickly in terms of time, and efficiently in terms of effort and quality of the solution.

Also in this thesis, to cover dynamic distributed CSPs, we propose on the one hand a new approach based on multi-Agent systems to model the intelligent autonomous market environment (sale and purchase of goods) to face the problems of e-markets. Then based on the famous ABT algorithm, we proposed the LiveABT algorithm to deal with disturbances in distributed CSPs in real time. Experiments have shown that the LiveABT algorithm outperforms the sequentially repair version performed by the DynABT algorithm.

Keywords: *Artificial Intelligence, Constraint Programming, Constraint Satisfaction Problem (CSP), Multi Agent System, Modeling, Planning, Optimizing, Dynamic Distributed CSP (DDisCSPs).*

Année Universitaire: 2021/2022