Constraint Games: Modeling and Solving Games with Constraints

Thi-Van-Anh Nguyen



UNIVERSITÉ DE CAEN BASSE NORMANDIE
U.F.R. de Sciences
ÉCOLE DOCTORALE SIMEM

T H È S E

Présentée par

Thi Van Anh NGUYEN

soutenue le

12 Décembre 2014

en vue de l'obtention du

DOCTORAT de l'UNIVERSITÉ de CAEN

Spécialité : Informatique et applications

Arrêté du 07 août 2006

Titre

Constraint Games: Modeling and Solving Games with Constraints

The work presented in this thesis was carried out at
GREYC - Université de Caen Basse-Normandie.

Jury

Francesca ROSSI	Professor	University of Padova, Italy	*Reviewer*
Lakhdar SAÏS	Professor	Université d'Artois, France	*Reviewer*
Patrice BOIZUMAULT	Professor	Université de Caen, France	*Examiner*
Lucas BORDEAUX	Research Engineer	Microsoft Research, UK	*Examiner*
Philippe JÉGOU	Professor	Université d'Aix-Marseille, France	*Examiner*
Bruno ZANUTTINI	Associate Professor	Université de Caen, France	*Examiner*
Arnaud LALLOUET	Professor	Université de Caen, France	*Supervisor*

Abstract

This thesis presents a topic at the interface of game theory and constraint programming. More precisely, we focus on modeling games in a succinct way and then computing their solutions on these compactly encoded games thanks to constraint programming.

For a long period of time, game theory has suffered a modeling difficulty due to the lack of compact representations for encoding arbitrary games. The basic game representation is still an n-dimensional matrix called *normal form* which stores utilities of all players with all joint strategy profiles. The matrix however exponentially grows with the number of players. This also causes a solving difficulty for computing solutions in games.

In the thesis, we introduce a novel framework of *Constraint Games* to model strategic interaction between players. A constraint game is composed of a set of variables shared by all the players. Among these variables, each player owns a set of *decision variables* he can control and constraints expressing his utility function. The *Constraint Games* framework is a generic tool to model general games and can be exponentially more succinct than their normal form. We also show the usefulness of the framework by modeling a few classical games as well as realistic problems.

The main solution concept defined for constraint games is *Pure Nash Equilibrium*. It is a situation in which no player has an incentive to deviate unilaterally. It has been well-known that finding pure Nash equilibrium is computationally complex. Nevertheless, we have achieved to build an efficient solver called *ConGa* which includes various ways for solving constraint games.

Keywords

Game Theory • Constraint Programming • Nash Equilibrium

Acknowledgements

First of all, I undoubtedly give a special note of thanks to my supervisor, Arnaud Lallouet. I am indebted to him for his kindly guidance and patience in conducting research and writing scientific documents. In particular, he always stood by my side during my tough time when I got failure and nothing worked. Honestly, there existed such extremely stress moments that I wanted to quit. But Arnaud did not abandon me, he always stayed with me and gave me encouragement to continue fighting. He taught me not only how to do research but also how to follow goals, not be afraid of failures, keep smiling and never give up. Without him, I would not be able to succeed in finishing this PhD thesis.

I am then grateful to my thesis committee including Francesca Rossi, Lakhdar Saïs, Patrice Boizumault, Lucas Bordeaux, Philippe Jégou and Bruno Zanuttini for their insightful comments and valuable time.

I would like to thank Microsoft Research PhD Scholarship Programme for providing the financial funding and other kinds of supports during my thesis. I also thank the members of Microsoft Research Cambridge, in particular my research mentors, Youssef Hamadi and Lucas Bordeaux, for their advices as well as the PhD scholarship administration team for their kindly help from the very first day until the end of my thesis.

I am also very grateful to my master internship supervisor, Laurent d'Orazio, who guided me the first steps to discover the world of research. Laurent is also the person who directly gave me encouragement, support and motivation to do this PhD thesis.

I would like to thank my colleagues for making my three years at Greyc is one of the most precious time in my life. Thank José Moréno (and also Gaël Dias, his supervisor for the comfortable discussions about everything), Guillaume Poezevara, Paul Martin and Romain Brixtel for nicely sharing the office with me. I am grateful to the members of the CoDaG team as well as the system administrators for their helps to accomplish the works in this thesis. I think of Winn Voravuthikunchai, Davy Gigan, Jean-philippe Métivier and the others I cannot all enumerate here. Thank you very much for all the times we had together.

On a personal note, I am grateful to Gérard Fougeray who welcomed me on the first days when I arrived in France. He did many things for helping me to overcome the tough period in my first year without family in a foreign country. And of course, I would like to thank my Vietnamese friends for the relaxing moments that make me forget the pressure of the PhD work.

At the end, I would like to thank my family for always loving me unconditionally. I am also grateful to my brother and my sister-in-law for taking care of my mother at home. Thank you so much for everything. Without you, I would not have what I have today.

To my father, with all my love.

Contents

Abstract 1

Acknowledgements 3

1 Introduction 7

I Background 11

2 Game Theory 13
2.1 Categories of Games . . . 14
2.1.1 Static and Dynamic Games . . . 14
2.1.2 Cooperative and Non-cooperative Games . . . 17
2.1.3 Complete and Incomplete Information Games . . . 19
2.1.4 Summary . . . 20
2.1.5 Specific Classes of Games . . . 20
2.2 Solution Concepts for Games . . . 22
2.2.1 Pure Nash Equilibrium . . . 22
2.2.2 Pareto Nash Equilibrium . . . 25
2.2.3 Mixed Nash Equilibrium . . . 26
2.2.4 Approximate Nash Equilibrium . . . 27
2.2.5 Strong Nash Equilibrium . . . 27
2.2.6 Other Types of Equilibrium . . . 28
2.2.7 Quantifying the Inefficiency of Equilibrium . . . 28
2.3 Representations of Games . . . 29
2.3.1 Normal Form . . . 29
2.3.2 Extensive Form . . . 30
2.3.3 Graphical Games . . . 31
2.3.4 Action-Graph Games . . . 32
2.3.5 Boolean Games . . . 33
2.3.6 Other Representations . . . 35
2.4 Existing Software in Game Theory . . . 36

3 Constraint Programming 37
3.1 Basic Notions . . . 37
3.2 Reasoning . . . 39
3.2.1 Backtracking Search . . . 39
3.2.2 Local Search . . . 43
3.3 Constraint Programming in Game Theory . . . 44

II Contributions **47**

4 Constraint Games **49**
4.1 Modeling Framework . . . 50
4.1.1 Constraint Satisfaction Games . . . 50
4.1.2 Constraint Satisfaction Games with Hard Constraints . . . 53
4.1.3 Constraint Optimization Games . . . 54
4.1.4 Constraint Optimization Games with Hard Constraints . . . 56
4.1.5 Usefulness and Compactness of Constraint Games . . . 56
4.2 Applications . . . 59
4.2.1 Games Taken from the Gamut Suite . . . 59
4.2.2 Games Taken from Real Life . . . 62
4.3 Solving Framework . . . 68
4.3.1 ConGa Solver . . . 69
4.3.2 Experimental Environment . . . 70
4.4 Conclusion . . . 70

5 Local Search for Constraint Games **71**
5.1 CG-IBR: Iterated Best Responses Algorithm in Constraint Games . . . 72
5.2 CG-TS: Tabu Search Algorithm in Constraint Games . . . 76
5.3 CG-SA: Simulated Annealing Algorithm in Constraint Games . . . 78
5.4 Experiment . . . 84
5.4.1 Description of Experimental Game Instances . . . 84
5.4.2 Experimental Results . . . 85
5.5 Related Work . . . 87
5.6 Conclusion . . . 88

6 Complete Search for Constraint Games **89**
6.1 CG-Enum-Naive: Naive Algorithm for Enumerating All Pure Nash Equilibrium . 90
6.2 Pruning Techniques . . . 91
6.2.1 Classical Techniques . . . 91
6.2.2 Advanced Techniques . . . 93
6.3 CG-Enum-PNE: Advanced Algorithm for Enumerating All Pure Nash Equilibrium 96
6.4 CG-Enum-PPNE: Algorithm for Finding All Pareto Nash Equilibrium . . . 101
6.5 Experiment . . . 103
6.6 Related Work . . . 106
6.7 Conclusion . . . 107

7 Heuristic Search for Constraint Games **109**
7.1 Deviation-based Heuristic for Variable and Value Ordering . . . 109
7.2 Heuristic Algorithm . . . 112
7.3 Experiment . . . 117
7.4 Related Work . . . 120
7.5 Conclusion . . . 121

8 Conclusion and Perspective **123**
8.1 Conclusion . . . 123
8.2 Perspective . . . 125

Bibliography **129**

Chapter 1

Introduction

The mathematical field of *game theory* [von Neumann and Morgenstern, 1944c] has been set up to address problems of strategic decision making in which agents interact in conflict situations. Game theory has an incredible success in description of economic processes [Tesfatsion, 2006], but is also used in various other domains such as biology or political sciences [Shubik, 1981, Maynard-Smith and Price, 1973] . Other issues include sequential or simultaneous interaction mode, complete or incomplete knowledge, determinism, coalitions, repetition, etc.

Nash equilibrium [Nash, 1951] is perhaps the most well-known and well-studied game-theoretic solution concept for static games in which each player is assumed to know the strategies of the other players, and no player has anything to gain by changing only their own strategy. In other words, given a set of strategy choices (called *strategy profile*) in which each player has chosen his strategy. The strategy profile is a Nash equilibrium if no player can improve his *outcome* (also called *payoff* or *utility*) by changing his strategy while the others keep theirs unchanged.

Since a few years, games have been studied in a computational perspective, raising new issues like complexity of equilibrium or succinctness of representation. The main representation for static games is still an n-dimensional matrix called *normal form* that stores utilities of all players with all possible strategy profiles [Fudenberg and Tirole, 1991]. However, normal form exponentially grows with the number of players. Actually, there is a lack of a general modeling language that can express arbitrary games while is also able to compactly encode utility functions exhibiting commonly-encountered types of structure. Probably, that is one of the biggest challenges in game theory. It seems responsible for the reason why there are too few solving tools available for generic games.

Indeed, this thesis is mainly motivated by two questions. First, is it possible to design the game representations that can compactly model a wide range of interesting games and are amenable to efficient computation? Second, how do we build efficient algorithms for computing solution concepts (e.g. Nash equilibrium) in these compactly represented games?

Along the path of exploration, we realize that most games have a natural understanding. More precisely, utilities are not randomly generated, but frequently constrained by a form of functions between players' strategies. They can thus (often) be expressed in a language. There is no doubt that it is better to understand utilities in terms of simple relationships than look up in enormous matrix. Therefore, a language for utilities could be suitable for the compact game representation that we expect to design.

On the other side, *constraint programming* [Rossi et al., 2006] is a framework for modeling and solving combinatorial and optimization problems, e.g. designing schedules, optimizing round trips, etc. Modeling a problem with constraints is fairly easy. We first need to define a set of *variables* along with their *domain*. Then, we specify a set of *constraints* which state the relations between the variables. A *solution* of the problem is an assignment of all the variables satisfying all the constraints. The solving tool for constraint problems is called *solver* which generally includes a set of constraint techniques. Once we have declared our problem, the solver runs until it finds a solution or all the solutions (according to our desire).

Now, constraint programming can solve industrial-sized problems with comparable efficiency as operational research techniques but with a clear and concise modeling language. In particular, side constraints which appear in industrial problems are easily taken into account within the framework and fully exploited during the solving phase.

In games, utility function (or preference) is usually expressed by a form of relations between strategies of players. While in constraint programming, relations between variables are stated in a form of constraints. We can see here an interesting similarity between the two fields. Indeed, our idea in this thesis is to model and solve games as constraint problems in which preferences will be encoded by constraints. Due to the efficiency of the framework, constraint programming would offer us not only compact game representations but also a set of robust techniques for solving.

In summary, the purpose of this thesis is to build a unified framework to model and solve games thanks to constraint programming. Referring to real-life problems, our framework is available to solve the problems with the following features: multiple agents partially compete for the same resources, have different (not mandatory disjoint) objectives and take decisions simultaneously.

Thesis Organization

Since this thesis is related to two different fields (game theory and constraint programming), we provide some background knowledge of these fields in Part I. The thesis contributions are then represented in Part II. The thesis organization is sketched in Figure 1.1 (on the next page).

In the first part, Chapter 2 is devoted to recall the basic elements of game theory that could allow to specify the position of the thesis in this huge field. We do the same thing, but for constraint programming, in Chapter 3.

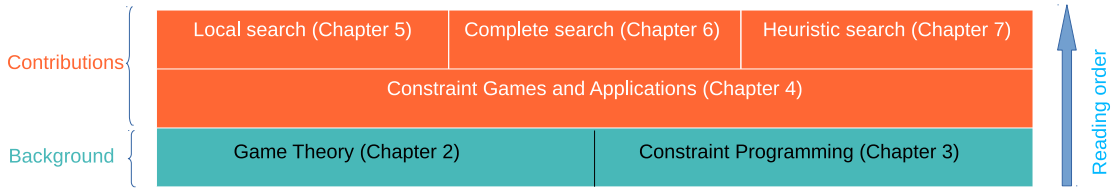


Figure 1.1: The thesis organization.

The second part expressed by the upper orange layers describes the thesis contributions. The first contribution, a compact game representation named *constraint games*, is represented in Chapter 4. We also illustrate the compactness as well as the usefulness of constraint games by modeling a set of applications taken from both classical games in the literature and real-life.

The algorithms for computing solution concepts, i.e. Nash equilibrium, are depicted in the top layer. Several local search algorithms are devised in Chapter 5. In Chapter 6, we propose a complete solver including both naive and advanced algorithms for equilibrium enumeration. Finally, a heuristic search for variable and value ordering is given in Chapter 7.

Thesis Contributions

The central contribution of the thesis is **Constraint Games** which are a new modeling framework for generic games with the solving techniques along. These techniques have been implemented in a solver for constraint games called **ConGa**. In detail, we have achieved the following outcomes.

- **A modeling framework.** By proposing constraint games, we have defined *a multiple agents language* which allows to express their interaction within the problem to model. It is a compact game representation which adapts to various modeling purposes thanks to its four natural variants. All these variants, in which optimization and/or hard constraints are allowed or not, are itemized as follows.

  - Constraint Satisfaction Games
  - Constraint Satisfaction Games with Hard Constraints
  - Constraint Optimization Games
  - Constraint Optimization Games with Hard Constraints

  We also give a *theory* of multiple players games including equilibrium notions and their computation. The main solution concepts defined for constraint games are *Pure Nash Equilibrium* (abbreviated PNE) and its subset *Pure Pareto Nash Equilibrium*.

  In this thesis, we demonstrate the utility of constraint games by modeling a few *applications* in various fields, such as economy, network or cloud computing, to name a few. In addition, these applications can be expressed in a compact and elegant way thanks to constraint games.

- **A solving framework.** Solving games is a very difficult task because determining whether a constraint game has a PNE is Σ_2^p-complete. The state-of-the-art solver is Gambit [McKelvey et al., 2014] which includes a simple enumeration algorithm with exponential complexity. We have implemented the solver *ConGa* including many search algorithms for different purposes.

  - *Local search* : Thanks to the compactness of constraint games, our local search solver may find a PNE in very large games even with the simplest algorithm called *iterated best responses*. The largest game tackled in our experiments has 200 players and 5040 strategies per player. A normal form representation of this game would have involved 200×5040^{200} entries. Additionally, two algorithms using the efficient metaheuristics sometimes strengthen better performance for the solver [Nguyen et al., 2013].
  - *Complete search* : Although finding one PNE is a very interesting problem in itself, finding all of them allows more freedom for choosing equilibrium that fulfills some additional requirements. For example, the correctness of the computation of pure Pareto Nash equilibrium relies on the completeness of PNE enumeration. We present in [Nguyen and Lallouet, 2014] a new, correct and complete solver which is based on a fast computation of equilibrium condition that we call *Nash consistency* and a pruning algorithm for *never best responses*. The experimental results demonstrate that our solver is faster than the classical game solver Gambit by one to two orders of magnitude.
  - *Heuristic search* : Local search solver can find a PNE in very large games. However, it does not guarantee to always detect a PNE as well as prove its absence. The complete solver proposed can overcome this drawback. However, it does not fit well into large games, even finding only the first PNE. Hence, we have also studied and implemented a heuristic search which allows to pick the first PNE within reasonable time.

Part I

Background

Chapter 2

Game Theory

Contents

2.1 Categories of Games 14
2.1.1 Static and Dynamic Games 14
2.1.2 Cooperative and Non-cooperative Games 17
2.1.3 Complete and Incomplete Information Games 19
2.1.4 Summary 20
2.1.5 Specific Classes of Games 20
2.2 Solution Concepts for Games 22
2.2.1 Pure Nash Equilibrium 22
2.2.2 Pareto Nash Equilibrium 25
2.2.3 Mixed Nash Equilibrium 26
2.2.4 Approximate Nash Equilibrium 27
2.2.5 Strong Nash Equilibrium 27
2.2.6 Other Types of Equilibrium 28
2.2.7 Quantifying the Inefficiency of Equilibrium 28
2.3 Representations of Games 29
2.3.1 Normal Form 29
2.3.2 Extensive Form 30
2.3.3 Graphical Games 31
2.3.4 Action-Graph Games 32
2.3.5 Boolean Games 33
2.3.6 Other Representations 35
2.4 Existing Software in Game Theory 36

Our work is based on game theory which has been well-known a huge research field of study of strategic decision making. We thus give in this chapter some background knowledge that allows to put the thesis in its place.

The central notion in game theory is *game*. Generally, a game is composed of a finite set of intelligent rational decision-makers (usually called *players*) who have the intention of achieving several purposes by choosing one or some *strategies* in their strategy set. A game can be identified by multiple factors. Among them, the type of games plays a crucial role in representing and solving games. In this chapter, the main categories of games are hence envisaged, including static or dynamic, cooperative or non-cooperative, and complete or incomplete information games.

In game theory, a *solution concept* is a formal rule for defining the outcome of a game. The solutions describe which strategies will be adopted by players, therefore, the results of games. Equilibrium concepts are the most used solution concepts. Among them, *Nash equilibrium* is the most well-known in the literature. Nash equilibrium is mainly addressed static games. A situation is called Nash equilibrium if no player has an incentive to change his choice unilaterally. The main solution concept defined in the thesis is Nash equilibrium. Nevertheless, we also give a global view related to other equilibrium.

Another important issue in game theory is *game representation*. It is a data structure capturing all information necessary to identify games. We hence enumerate and analyze both advantages and drawbacks of the existing game representations as well as their solving tools from the literature. Finally, this chapter is ended by several software available for solving games.

2.1 Categories of Games

We give here some main categories of games. Generally, a game is identified by interaction modes, by behavior between players and by game knowledge.

2.1.1 Static and Dynamic Games

A game can be specified by the interaction mode of all players, i.e. the way how they play their game. From this point of view, there are two main categories: static games and dynamic games. In a static game (also called *simultaneous game*), all players make decisions (or select a strategy) at the same time, without knowledge of the strategies that are being chosen by other players. Meanwhile, in a dynamic game (also called *sequential game*), one player chooses his strategy before the others choose theirs. Therefore, the later players must have some information of the previous choices, otherwise the difference in time would have no strategic effect. Dynamic games are hence governed by the time axis, while static games do not have a time axis as players choose their moves without being sure of the other's.

Static Games

A game is said to be *static* if all players take decision *simultaneously* and obtain the utility according to the decisions made by all players, respectively.

Definition 2.1 (Static Game)**.** *A* static game *is a 3-tuple* $(\mathscr{P}, (S_i)_{i\in\mathscr{P}}, (u_i)_{i\in\mathscr{P}})$ *where:*

- $\mathscr{P} = \{1,\dots,n\}$ *is a finite set of players (also called* agents*);*
- *for each player i,* $S_i \neq \emptyset$ *is the strategy set of the player. Let* $s_i \in S_i$ *be a strategy (also called* action*) that player i can choose. A strategy profile* $s = (s_1,\dots,s_n) \in \Pi_{i\in\mathscr{P}} S_i$ *is a tuple of strategies of n players. We denote by* s_{-i} *the set of tuples of* $(n-1)$ *players, except i, under the strategy profile s. The notation* Π *expresses the Cartesian product over the sets;*
- $u_i : \Pi_{j\in\mathscr{P}} S_j \to \mathbb{R}$ *is i's utility (also called* payoff*) function which specifies i's utility given any strategy profile.*

Rock-Paper-Scissors (Example 2.1) is a famous hand game which is one of the simplest examples of static games.

Example 2.1 (Rock-Paper-Scissors)**.** *Rock-paper-scissors is a game played by two people, where players simultaneously form one of three shapes with an outstretched hand. The "rock" beats scissors, the "scissors" beat paper and the "paper" beats rock (see Figure 2.1); if both players throw the same shape, the game is tied.*

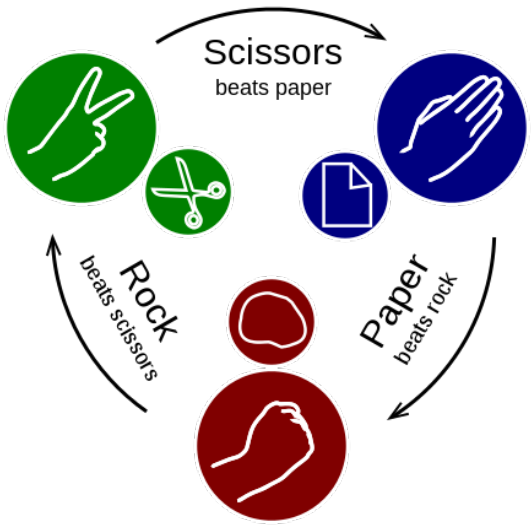


Figure 2.1: A chart showing how the three game elements interact in Rock-Paper-Scissors. Source: `http://en.wikipedia.org/wiki/Rock-paper-scissors`

This game can be formulated in static games as follows.

- $\mathscr{P} = \{1,2\}$
- Both the two players can choose a strategy from the same strategy set $S_1 = S_2 = \{rock, paper, scissors\}$
- The game has 9 strategy profiles: (rock, rock); (rock, paper); (rock, scissors); (paper, paper); (paper, rock); (paper, scissors); (scissors, paper); (scissors, rock); (scissors, scissors).
- The utility function obtained by each player is following, where "1" stands for "win"; "-1" for "lose" and "0" for "a tied game"
  - $\forall i \in \mathscr{P}, u_i(rock, rock) = u_i(paper, paper) = u_i(scissors, scissors) = 0$
  - $u_1(rock, scissors) = u_1(scissors, paper) = u_1(paper, rock) = 1$

- $u_2(rock, scissors) = u_2(scissors, paper) = u_2(paper, rock) = -1$
- $u_1(scissors, rock) = u_1(paper, scissors) = u_1(rock, paper) = -1$
- $u_2(scissors, rock) = u_2(paper, scissors) = u_2(rock, paper) = 1$

In the literature of game theory, "battle of the sexes" (Example 2.2) could be seen as a static game as well.

Example 2.2 (Battle of the sexes)**.** *" Battle of the sexes" is a two-player coordination game [Osborne and Rubinstein, 1994] in which a couple planned to go out in the evening. However, they have not yet decided where they would go. While the wife would like to go to the opera, her husband enjoys going to the football match. Both would prefer to go to the same place rather than different ones.*

We can encode this game as a static game as follows.

- $\mathscr{P} = \{wife, husband\}$
- Both the wife and the husband can choose a strategy from the same strategy set $S_{wife} = S_{husband} = \{opera, football\}$
- The game has 4 strategy profiles: (opera, opera); (opera, football); (football, opera); (football, football)
- The utility function obtained by each player is following:
  - $u_{wife}(opera, opera) = u_{husband}(football, football) = 3$
  - $u_{wife}(football, football) = u_{husband}(opera, opera) = 2$
  - $u_{wife}(opera, football) = u_{husband}(opera, football) = u_{wife}(football, opera) = u_{husband}(football, opera) = 0$

Dynamic Games

Unlike static games, in a *dynamic game* [Zaccour, 2005, Haurie and Krawczyk, 2000], the players move sequentially, i.e. a player will choose his action before the others choose theirs. Several real-life games such as chess, tic-tac-toe, and Go are typical dynamic games. Let us illustrate dynamic games by the following example of the tic-tac-toe game.

Example 2.3 (Tic-tac-toe)**.** *Tic-tac-toe is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a* 3×3 *grid. The player who succeeds in placing three respective marks in a horizontal, vertical, or diagonal row wins the game.*

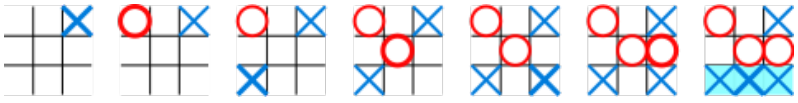


Figure 2.2: An example of the tic-tac-toe game in which player "X" wins. Source: `http://en.wikipedia.org/wiki/Tic-tac-toe`

Dynamic games are managed by the time axis. Figure 2.2 illustrates an example of tic-tac-toe along time, from the beginning with the first move of player "X" until the end when a player is capable of placing a horizontal row.

2.1.2 Cooperative and Non-cooperative Games

Since game theory focuses on studying problems in which players' goals depend on each other, a crucial characteristic for identifying a game is thus the cooperation between players. Therefore, games can be divided into two categories: cooperative and non-cooperative games. In a cooperative game, several players may collaborate to achieve their goals together. Meanwhile, in a non-cooperative game, all players are self-interested. Each player just wants to optimize his own utility without caring about what happens with the others.

Cooperative Games

The central notion of *cooperative games* is *coalition* which is a group of players making collaboration to gain the payoff together.

Definition 2.2 (Coalition)**.** *A* coalition *is a subset of players: $C \subseteq \mathscr{P}$. If C is a coalition consisting of only one player, i.e. $C = \{i\}$, C is called* singleton*. If C is formed by all players, i.e. $C = \mathscr{P}$, then C is called* grand coalition*.*

A cooperative game [Driessen, 1988] consists of two elements

(i) A set of players;
(ii) A characteristic function specifying the value created by different coalition C in the game

The definition of cooperative games is henceforth given as follows.

Definition 2.3 (Cooperative Games)**.** *A* cooperative game *is a pair $(\mathscr{P}, v)$ where:*

- $\mathscr{P} = \{1, \ldots, n\}$*, a set of finite players.*
- *A* characteristic function *which associates a vector $v(C) \in \mathbb{R}^C$ to each coalition C. Each element $v_i \in v(C)$ specifies the utility obtained by each player $i \in C$.*

Example 2.4 (Example 2.2 continued)**.** *[Luce and Raiffa, 1957] express the* battle of the sexes *game as a cooperative games as follows.*

- $\mathscr{P} = \{wife, husband\}$.
- *There are 3 coalitions in the game:* $C_1 = \{wife, husband\}, C_2 = \{wife\}, C_3 = \{husband\}$.
- *The characteristic functions of each coalition are following:*
  - $v(C_1) = (v_{wife}, v_{husband})$ *where* $v_i = 2 \vee v_i = 3, \forall i \in \{wife, husband\}$
  - $v(C_2) = (v_{wife})$ *where* $(v_{wife} = 0) \vee (v_{wife} = 2) \vee (v_{wife} = 3)$
  - $v(C_3) = (v_{husband})$ *where* $(v_{husband} = 0) \vee (v_{husband} = 2) \vee (v_{husband} = 3)$

In Example 2.4, when the wife and the husband appear in a same coalition, namely they put the common interest as the first choice. Since both prefer to go to the same place than the different one, they will choose one between two strategy profiles: (opera, opera) or (football, football). Hence, their utility in this coalition can be 2 or 3. On the other side, in the two remaining coalitions, each player is free to choose his/her strategy, then all possible utilities will be considered.

Non-cooperative Games

Non-cooperative games, which are a crucial class in game theory, capture many research interests. A *non-cooperative game* can be either *zero-sum* [von Neumann and Morgenstern, 1944a] or *non-zero-sum* [von Neumann and Morgenstern, 1944b] game. In a zero-sum game, a player's gain (or loss) is exactly balanced to the losses (or gains) of the others. If the total gains of the players are added up, and the total losses are subtracted, they will sum to zero. Thus, zero-sum games are also called *strictly competitive* games.

Example 2.5 (Berlin occupancy)**.** *Figure 2.3 illustrates a zero-sum game in which the allies countries share the occupancy of Berlin after the chute of Germany in the second world war.*

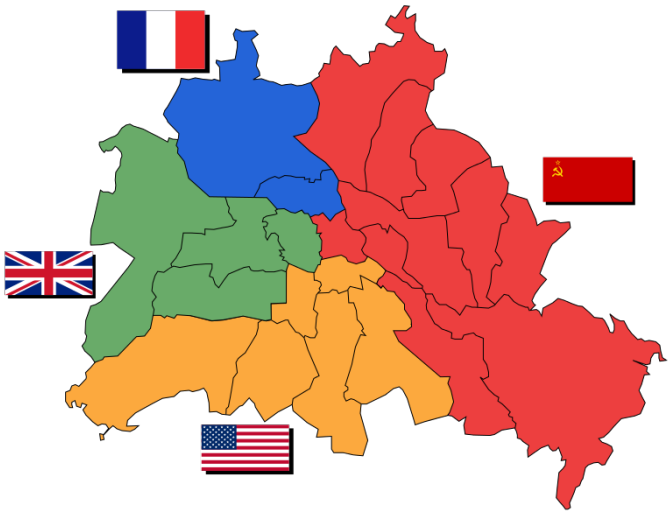


Figure 2.3: Sectors of divided Berlin. Source: `http://en.wikipedia.org/wiki/Berlin_Blockade`

As we can see in the figure, if a country widens his occupied land then the remaining area for all the others will be henceforth more restricted. Zero-sum games are an important element studied in various real-world problems in economy or politic such as the problems of dividing a set of goods or resources between several people, such that each person receives his due share.

In contrast, non-zero-sum games describe a situation in which the interacting players' aggregate gains and losses are either less than or more than zero. Non-zero-sum games are also non-strictly competitive, as opposed to the completely competitive zero-sum games. An interesting point of non-zero-sum games is that such games include both competitive and cooperative issues. Thus, they generally fit better into various real-world problems in which the interaction between players is usually complex. Prisoner's dilemma is a typical of non-zero-sum games (Example 2.6).

Example 2.6 (Prisoner's dilemma)**.** *The classical* prisoner's dilemma *[Poundstone, 1988] introduces two prisoners put in jail without being able to talk to each other. The police plans to sentence both of them one year, but proposes to each of them to testify against his partner in exchange of liberty for him and three years for his partner. But if both testify, then both are sentenced to two years in jail. In this game, each player has possibility to play 0 (defect) or 1*

(cooperate) with respect to the other player.

In the literature, [von Neumann and Morgenstern, 1944c] prove that any zero-sum game involving n players is in fact a generalized form of a zero-sum game for two players, and that any non-zero-sum game for n players can be reduced to a zero-sum game for $n+1$ players; the $(n+1)$ player representing the global profit or loss.

2.1.3 Complete and Incomplete Information Games

We have surveyed the way players move as well as the cooperation between players in a game. Another important point all players want to know is what has been happening in the game. This raises two other categories for classifying games: Complete and Incomplete Information Games. In a game of complete information, all players are perfectly informed of all other players payoff for all possible strategy profiles. If instead, a player is uncertain of the payoffs to other players then the game is one of incomplete information. In an incomplete information setting, players may not possess full information about their opponents. The games are generally used for modeling problems including uncertainty features.

Complete Information Games

A game is said to be complete information [Fudenberg and Tirole, 1991] if every player knows the payoffs and strategies available to other players. Both *Battle of the sexes* (Example 2.2) and *Prisoner's dilemma* (Example 2.6) are classified into complete information games category. Then, complete information games include *perfect* and *imperfect information games*.

In a *perfect information game*, players know the full history of the game, the moves made by the others as well as all their strategies and their payoffs. Tic-tic-toe or chess are some typical examples of such games. In contrast, there is at least one decision-node where a player does not know where he is in the game in *imperfect information games*. There may also be a chance element (as in most card games).

Incomplete Information Games

Incomplete information games (or Bayesian games) [Harsanyi, 1968] was firstly proposed by John C. Harsanyi for modeling the multi-player situation in which at least one player is unsure of the type (and so the payoff function) of another player. Because information is incomplete, in a Bayesian game, it is to specify the strategy spaces, type spaces, payoff functions and beliefs for every player. Incomplete information games are important in capturing many economic situations, where a variety of features of the environment may not be commonly known.

Example 2.7 (Incomplete information Battle of the sexes)**.** *The* battle of the sexes *game (Example 2.2) can be modified to a new version in which there is no collaboration between the two players.*

Even more, a player does not know whether the other wishes to meet or wishes to avoid her/him. This is a situation of incomplete information.

2.1.4 Summary

We summarize in Table 2.1 the game categories presented above along with the examples.

Example	Interaction mode		Behavior between players		Game knowledge	
	Static	*Dynamic*	*Cooperative*	*Non-cooperative*	*Complete information*	*Incomplete information*
Example 2.1	x			x	x	
Example 2.3		x		x	x	
Example 2.4	x		x		x	
Example 2.5	x			x	x	
Example 2.6	x			x	x	
Example 2.7	x			x		x

Table 2.1: Summary of the game categories

In this thesis, we are interested in multiple players games which are ***static, non-cooperative, complete information***. In such games, all players make decision simultaneously with no cooperation between players. Finally every player knows about the strategies and the utilities available for himself as well as for all others.

2.1.5 Specific Classes of Games

Besides the general game categories mentioned above, there also exists some special game classes having nice properties which facilitate the representation and the reasoning. Fortunately, such games very often appear in real-world problems in various fields. Hence, they are also an interesting research subject studied for decades.

In this section, we mention several specific classes of games for two reasons. First, many applications encoded by our modeling tools, constraint games, fall in these classes. Hence, we are confident that it would be more comprehensive to understand the applications with the background knowledge of the game classes. Second, in the thesis, we also concentrate on studying algorithmic game theory in order to design the solving tools for games. We thus recall here *potential games* as well as *congestion games* which are proven to always own at least one equilibrium.

Potential Games

Potential games [Rosenthal, 1973, Monderer and Shapley, 1996] were proposed by Robert W. Rosenthal in 1973. A game is said to be a potential game if the incentive of all players to change their strategy can be expressed using a single global function called the *potential function*.

Definition 2.4 (Potential Games)**.** *A* potential game *is a 4-tuple* $(\mathscr{P}, (S_i)_{i\in\mathscr{P}}, (u_i)_{i\in\mathscr{P}}, \Phi)$ *where:*

- $\mathscr{P} = \{1,\ldots,n\}$ *is a finite set of players;*

- S_i *is a strategy set for player i;*
- $u_i(s_1,\ldots,s_n)$ *is an utility function for player i over the strategy profile* $s=(s_1,\ldots,s_n)$*;*
- Φ *is a* potential function *which defines the kind of potential game.*

Definition 2.5 (Potential Functions)**.** *There are several kinds of potential function as follows.*

- *Exact potential function:* $\Phi: S \to \mathbb{R}$ *such that*
  $\forall s_{-i} \in S_{-i}, \forall s_i', s_i'' \in S_i, \Phi(s_i', s_{-i}) - \Phi(s_i'', s_{-i}) = u_i(s_i', s_{-i}) - u_i(s_i'', s_{-i})$.
- *Weighted potential function:* $\Phi: S \to \mathbb{R}$ *and a vector* $w \in \mathbb{R}^{\mathscr{P}}$ *such that*
  $\forall s_{-i} \in S_{-i}, \forall s_i', s_i'' \in S_i, \Phi(s_i', s_{-i}) - \Phi(s_i'', s_{-i}) = w_i(u_i(s_i', s_{-i}) - u_i(s_i'', s_{-i}))$.
- *Ordinal potential function:* $\Phi: S \to \mathbb{R}$ *such that*
  $\forall s_{-i} \in S_{-i}, \forall s_i', s_i'' \in S_i, u_i(s_i', s_{-i}) - u_i(s_i'', s_{-i}) > 0 \to \Phi(s_i', s_{-i}) - \Phi(s_i'', s_{-i}) > 0$.

Because the moves of all players in potential games are mapped into one function, then the potential function is usually an useful tool to analyze equilibrium properties of games.

Congestion Games

The class of congestion games [Rosenthal, 1973] is narrow. Nevertheless, it has widely been used for encoding many problems, especially in network traffic. Any game where a collection of homogeneous players has to choose from a finite set of alternatives, and where the payoff of a player depends on the number of players choosing each alternative, is a congestion game. In [Monderer and Shapley, 1996], the authors show that the class of congestion games coincides with the class of finite potential games.

Definition 2.6 (Congestion Games)**.** *A congestion game is a 4-tuple* $(\mathscr{P}, R, (\Sigma_i)_{i\in\mathscr{P}}, (d_r)_{r\in R})$ *where:*

- $\mathscr{P} = \{1,\ldots,n\}$ *is a finite set of players;*
- $R = \{1,\ldots,m\}$ *is a set of resources;*
- $\Sigma_i \subseteq 2^R$ *is the strategy space of player* $i \in \mathscr{P}$*;*
- $d_r : \{1,\ldots,n\} \to \mathbb{Z}$ *is the delay function of resource* $r \in R$*.*

For any state $s=(s_1,\ldots,s_n) \in \Sigma_1 \times \ldots \times \Sigma_n$, $\forall i \in \mathscr{P}$, let $\delta_i(s)$ be the payoff of player i, $\delta_i(s) = \Sigma_{r\in s_i} d_r$.

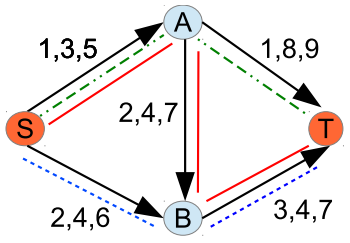


Figure 2.4: An example of the network congestion game with 3 players.

Figure 2.4 illustrates an example of congestion games in the network domain. In this game, three players take the possible paths to reach the target T from the source S. The delay function d_r

which is defined by the the number of players is noted on each path. For example, on the path SA, its delay function is 1 if one player takes SA; 3 if SA is chosen by two players; and 5 if three players pass through this path. Each player would like to allocate the minimal delay paths to reach the target T. In this figure, the path chosen by player 1 is depicted by the solid lines, then the dash lines for player 2 and the dotted lines for player 3. For the current state shown in the figure, the payoff of player 1 is 3 + 2 + 4 = 9, of player 2 is 2 + 4 = 6 and player 3 is 3 + 1 = 4.

2.2 Solution Concepts for Games

Solution concepts for games have been discussed in the game theory community for decades since there are many ways to define them [Osborne and Rubinstein, 1994]. One of the most fundamental problems in computational game theory is undoubtedly the computation of a Nash equilibrium [Nash, 1951], which models a situation where no player has an incentive to change his decision unilaterally. More precisely, Nash equilibrium is the canonical solution concept of static games. In this section, we give a brief survey of most well-known concepts, focusing on Nash equilibrium, with their existing solving tools along in the literature.

2.2.1 Pure Nash Equilibrium

A situation is said to be a *Pure-strategy Nash Equilibrium* (hereafter just *Pure Nash Equilibrium*) if in this situation, no player can improve his own utility, knowing the other players' decision.

Definition 2.7 (Best Response)**.** *A strategy s_i is said to be a* best response *of player i if and only if* $\forall s'_i \neq s_i, u_i(s_i, s_{-i}) \geq u_i(s'_i, s_{-i})$.

Definition 2.8 (Pure Nash Equilibrium)**.** *A pure strategy profile s is a* Pure Nash Equilibrium *(abbreviated PNE) if and only if* $\forall i \in \mathscr{P}, \forall s_{-i}, \forall s'_i \neq s_i, u_i(s_i, s_{-i}) \geq u_i(s'_i, s_{-i})$*, i.e. s is a best response of all players.*

Let us illustrate the couple of notions above in a static game in Figure 2.5. It is a game composed of two players. Player 1 can take a strategy from a set given by $\{1,2,3\}$ and player 2 from a set given by $\{a,b,c\}$.

Figure 2.5a depicts a matrix that stores all utilities of the two players. In each cell, the first number is the utility of player 1 and the second is for player 2. In Figure 2.5b, the strategy profile $(2,c)$) is a PNE because it is a best response of both the two players. The tuple $(2,c)$ is a best response of player 1 (Figure 2.5c) because, when player 2 keeps his strategy unchanged, i.e. strategy c, if player 1 chooses either strategy 1 or strategy 3, he will receive a worse utility. Therefore, he would not deviate to other strategies, i.e. he already gets his best response. The similar explanation is proposed for the reason why the tuple $(2,c)$ is also a best response of player 2.

	a	b	c
1	1 3	2 1	1 2
2	3 1	1 3	2 4
3	2 1	3 1	1 2

(a) Multimatrix stores all utilities of the players.

	a	b	c
1	1 3	2 1	1 2
2	3 1	1 3	2 4
3	2 1	3 1	1 2

(b) Strategy profile (2,c) is a PNE of the game.

	a	b	c
1	1	2	1
2	3	1	2
3	2	3	1

(c) Tuple $(2,c)$ is a best response of player 1.

	a	b	c
1	3	1	2
2	1	3	4
3	1	1	2

(d) Tuple (2,c) is a best response of player 2.

Figure 2.5: Illustration of Best Response *and* Pure Nash Equilibrium *in a static game*

There are many ways to define solution concepts [Osborne and Rubinstein, 1994] but PNE has the notable advantage of giving a deterministic decision for the players. Moreover, PNE for games are similar to solutions for constraint problems: not all games own a PNE, and when available, some PNE may be more desirable than others. Therefore, we define PNE as the main solution concept for our modeling tools proposed in this thesis, namely constraint games.

Algorithm 2.1. Finding all PNE in a game

```
1: function SOLVE(game G): setof tuple
2:     PNE ← ∅
3:     for all s ∈ Π_{i∈𝒫} S_i do
4:         if isPNE(s) then
5:             PNE ← PNE ∪ {s}
6:         end if
7:     end for
8:     return PNE
9: end function
```

Algorithm 2.2. Checking whether a tuple is PNE

```
1: function ISPNE(tuple s): boolean
2:     for all i ∈ 𝒫 do
3:         if isDev(s, i) then return false
4:         end if
5:     end for
6:     return true
7: end function
```

For finding PNE in games, a generic algorithm is described in Algorithm 2.1. In this algorithm, we simply enumerate all tuples of games. We then verify whether each tuple satisfies the equilibrium condition (line 3 - 7). This verification is performed in Algorithm 2.2. A tuple is said to be a PNE if no player can make a beneficial deviation from the current tuple. A player can deviate if he is able to change to another strategy to get a better utility (Algorithm 2.3).

Algorithm 2.3. Checking whether a player can deviate from the current tuple

```
1: function ISDEV(tuple t, int i): boolean
2:     for all v ∈ S_i, v ≠ s_i do
3:         if u_i(v, s_{-i}) > u_i(s) then
4:             return true
5:         end if
6:     end for
7:     return false
8: end function
```

This generic algorithm, which depends on game representations, is naive and inefficient. However, it is still the baseline algorithm for solving an arbitrary game in the literature. On the complexity of determining PNE, [Gottlob et al., 2005] prove that finding a PNE of a game, even if severe restrictions are posed is a hard problem. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows (see Theorem 2.1).

Theorem 2.1 (taken from [Gottlob et al., 2005])**.** *Determining whether a game has a PNE is NP-complete.* [1].

[Duersch et al., 2012] show that every finite symmetric quasiconcave two-player zero-sum game has a PNE. Earlier, in 1963, Shapley et al. prove that a finite two-player zero-sum game has a PNE if every 2x2 submatrix of the game has a pure equilibrium [Shapley, 1963]. In aspect of reasoning, [Cheng et al., 2004] exploit symmetry for more efficient methods, i.e. solving game by *function minimization* and *replicator dynamics*, which apply for finding one or a set of PNE, not all PNE.

The authors in [Monderer and Shapley, 1996] prove a nice property of potential games in Theorem 2.2. It thus suggests an useful game class in order to test the solving tools for detecting PNE.

Theorem 2.2. *Every finite potential game has at least one pure Nash equilibrium.*

The same thing happens with congestion games as they are a subclass of potential games. Even more, it have been proposed in Theorem 2.3 in [Rosenthal, 1973] that:

Theorem 2.3. *For every congestion game, every sequence of improvement steps is finite. Hence, every congestion game has at least one PNE.*

[1] It is the complexity in the multimatrix (also called *Normal form*) representation which is clearly described in Section 2.3.1. We also notice that all the other complexity results of Nash equilibrium in this section are for games in normal form.

Hence, we can use the potential function in order to find an equilibrium point. The problem is that the number of steps might be exponential in the size of the game. [Fabrikant et al., 2004] prove that determining whether a congestion game has a PNE is PLS-complete [2], while in the symmetric network case, PNE can be computed in polynomial time. While an unweighted congestion game always have a PNE, it is no longer true with weighted variants that rely on sharing costs proportional to players' weights. [Kollias and Roughgarden, 2011] propose a way of assigning costs to players with weights in congestion games that recovers the important properties of the unweighted model.

Local search has also been studied for finding PNE such as *iterated best responses* [Shoham and Leyton-Brown, 2009]. Additionally, the authors in [Son and Baldick, 2004] argue that any iterative PNE search algorithms based on local optimization cannot differentiate real NE and "local NE traps". Thus, they apply coevolutionary programming, a parallel and global search algorithm, to overcome this problem.

2.2.2 Pareto Nash Equilibrium

In games, it would happen that some equilibrium are more desirable than the others. Usually, the purpose of solution concepts built up on Nash equilibrium aim at removing less realistic solutions in games. *Pure Pareto Nash Equilibrium* (hereafter just *Pareto Nash Equilibrium*) is henceforth such kind of concept.

Pareto efficiency, or Pareto optimality, is a state of allocation of resources in which it is impossible to make any one individual better off without making at least one individual worse off. That is, a Pareto Optimal outcome cannot be improved upon without hurting at least one player. Pareto optimality has been widely used as the most fundamental solution concept for the multi-objective optimization problems. Moreover, this solution concept has also been applied for games, e.g. [Sen et al., 2003].

Definition 2.9 (Pareto Nash Equilibrium (taken from [Gottlob et al., 2005]))**.** *A Pure Nash Equilibrium s is a* Pareto Nash Equilibrium *(abbreviated PPNE) if there does not exist a PNE s' such that $\forall i \in \mathscr{P}, u_i(s') > u_i(s)$.*

Theorem 2.4 (taken from [Gottlob et al., 2005])**.** *Determining whether a game has at least one PPNE is NP-complete.*

Obviously, if there exists a PNE, then there also exists a PPNE. In other words, let P_1 be the set of all PNE and P_2 be the set of all PPNE, then $P_2 \subseteq P_1$. Both finding PNE and PPNE problems share the same complexity. In terms of the existing algorithms, [Gasior and Drwal, 2013] propose a distributed algorithm which provably stops at PPNE in capacity allocation game for self-managed networks.

[2] Polynomial Local Search (abbreviated PLS) is a complexity class that models the difficulty of finding a locally optimal solution to an optimization problem [Yannakakis, 2009].

2.2.3 Mixed Nash Equilibrium

Most work in game theory concerns mixed equilibrium. A mixed equilibrium is the given of a probability distribution for each player on his strategies such that no player has incentive to change his distribution. While this notion of strategy allows each game to have an equilibrium [Nash, 1951], it does not provide an implementable solution concept but more a guideline for all players to choose their action.

In static games, a player plays a pure strategy when he deterministically chooses a strategy from his strategy set. A mixed strategy is the one in which a player takes his available pure strategies with certain probabilities. We denote by $\phi(S_i)$ the set of mixed strategy for player i, than a mixed strategy $\sigma \in \phi(S_i)$ is a probability distribution over S_i.

Definition 2.10 (Expected Utility)**.** *The* expected utility *of player i under the mixed strategy profile σ, denoted by $u_i(\sigma)$, is $u_i(\sigma) = \sum_{s \in S} u_i(s) \Pi_{j \in \mathscr{P}} \sigma_j(s_j)$, where $\sigma_j(s_j)$ denotes the probability that j plays s_j.*

Definition 2.11 (Mixed Nash Equilibrium)**.** *A mixed strategy profile σ is a* Mixed Nash Equilibrium *if $\forall i \in \mathscr{P}, \sigma_i \in arg\max_{\sigma_i} u_i(\sigma_i, \sigma_{-i})$ where σ_{-i} is a tuple of mixed strategy of the other players, except i.*

Example 2.8 (Matching Pennies Game)**.** *It is a game composed of two players: player A and player B. Each player owns a penny and chooses to turn the penny to head or to tail. Both turn their penny simultaneously and immediately reveal the results together. If the pennies match, i.e. both head or both tail. Player A wins and gets a score of* 1*, while* -1 *for player B. Otherwise, if the pennies do not match (one heads and one tails) then player B wins and gets a score of* 1 *and* -1 *for player A.*

There is no PNE in this game. But since the two players turn their pennies secretly. Their plan should be "unpredictable". That is, we should randomize (or mix) between strategies. If each player turns to head or to tail with a probability of 0.5, then their expected utility is $0.5 \times 1 + 0.5 \times (-1) = 0$ *and neither A nor B can do better by deviating to another strategy. Thus, matching pennies game has one mixed Nash equilibrium being (*$0.5, 0.5$*).*

Determining whether a game has at least one mixed Nash equilibrium does not fall into a classical complexity class. More precisely, it is PPAD-hard [3] [Papadimitriou, 2007, Daskalakis et al., 2006] even if the game is composed of only two players.

Many efficient algorithms have been devised for computing mixed equilibrium such as Lemke-Howson [Lemke and Jr, 1964], Simplicial subdivision [Scarf, 1973], Porter-Nudelman-Shoham [Porter et al., 2008]. Recently, local search algorithms have been introduced in this context [Gatti et al., 2012, Ceppi et al., 2010] as well as genetic algorithms [Ismail et al., 2007].

[3] PPAD (Polynomial Parity Arguments on Directed graphs) is a complexity class introduced by C. Papadimitriou in 1994. PPAD is a subclass of TFNP (Total Function Nondeterministic Polynomial) based on functions that can be shown to be total by a parity argument.

2.2.4 Approximate Nash Equilibrium

An *approximate Nash equilibrium* [Bubelis, 1979], is a strategy profile that approximately satisfies the condition of Nash equilibrium. This allows the possibility that a player may have a small incentive to do something different.

Definition 2.12 (Approximate Nash Equilibrium)**.** *Given $\varepsilon > 0$, a mixed strategy profile σ is an* Approximate Nash Equilibrium *(or ε-Nash Equilibrium) if for $\forall i \in \mathscr{P}, \forall \sigma_i' \in \phi(S_i), u_i(\sigma) + \varepsilon \geq u_i(\sigma_i', \sigma_{-i})$.*

Clearly, an approximate Nash equilibrium falls back to a general mixed Nash equilibrium when $\varepsilon = 0$. Hence the complexity of computing an approximate Nash equilibrium of two-player games is PPAD-complete as well, even for constant values of the approximation [Daskalakis, 2011].

Computing approximate Nash equilibrium has been studied in [Hémon et al., 2008] for multi-player games. In [Daskalakis et al., 2009a], the authors propose a linear-time algorithm for games consisting of only two strategies per player. The algorithm returns a $\frac{1}{2}$-approximate Nash equilibrium in any 2-player game. Later, [Bosse et al., 2010] provide a polynomial time algorithm that achieves an approximation guarantee of 0.36392. They also extend the algorithms for 2-player games by exhibiting a simple reduction that allows to compute approximate equilibrium for multi-player games.

Following another research approach, [Cartwright, 2003] presents an imitation heuristic and an innovation heuristic to guide the players learn to play to converge to an approximate NE in large games. On the other hand, [Chien and Sinclair, 2011] specify their interest in congestion games by studying the ability of decentralized, local dynamics to rapidly reach an approximate Nash equilibrium.

2.2.5 Strong Nash Equilibrium

All the solution concepts we have previously mentioned are for static games in which there is no collaboration between players. Now, we present *Pure Strong Nash Equilibrium* [Aumann, 1959] (hereafter just *Strong Nash Equilibrium*, also called *core*) which is a Nash equilibrium concept for games with collaboration. It is a situation in which no coalition, taking the actions of its complements as given, can cooperatively deviate in a way that benefits all of its members.

From [Gottlob et al., 2005], given $K \subseteq \mathscr{P}$, a coalition, let s be a strategy profile, and $s' \in \Pi_{i \in K} S_i$ a combined strategy in K. Then, we denote by $s_{-K}[s']$ the strategy profile where for each player $i \in K$, his individual strategy $s_i \in s$ is replaced by his individual strategy profile $s_i' \in s'$.

Definition 2.13 (Strong Nash Equilibrium)**.** *A strategy profile is a* Strong Nash Equilibrium *(abbreviated SNE) if, $\forall K \subseteq \mathscr{P}, \forall s' \in \Pi_{i \in K} S_i, \exists i \in K$ such that $u_i(s) \geq u_i(s_{-K}[s'])$.*

Theorem 2.5 (taken from [Gottlob et al., 2005])**.** *Determining whether a game has at least one SNE is Σ_2^p-complete.*

Note that the existence of a PNE does not imply the existence of a SNE. Several algorithms have been proposed to compute SNE in specific classes of games, e.g., congestion games [Hayrapetyan et al., 2006, Rozenfeld and Tennenholtz, 2006]. More recently, [Gatti et al., 2013] design a spatial branch–and–bound algorithm to find a SNE in general games.

2.2.6 Other Types of Equilibrium

In order to model rationality, many other solution concepts have been proposed such as Correlated Equilibrium [Aumann, 1974], Subgame Perfect Equilibrium [Osborne, 2004], Sequential Equilibrium [Kreps and Wilson, 1982] or Stackelberg Equilibrium [Stackelberg, 1952, Sherali et al., 1983], etc. Although these notions are very interesting, they are not directly related to this thesis. Hence, we do not make a survey more detailed about them and we refer the interested readers to the references cited above.

2.2.7 Quantifying the Inefficiency of Equilibrium

The previous parts of this section provided numerous notions of equilibrium for defining solution concepts in games. But do the objective function value of an equilibrium of the game and of an optimal outcome always coincide? The answer is “no”. This leads to an important notion called “measures of the inefficiency of the equilibrium of a game” [Roughgarden and Tardos, 2007]. In fact, all of these measures are defined as the ratio between the value of an equilibrium and of an optimal outcome of games.

The *price of anarchy* (abbreviated PoA) is the most popular measure of the inefficiency of equilibrium. It resolves the issue of multiple equilibria by adopting a worst-case approach. The price of anarchy of a game is the ratio between the worst objective function value of an equilibrium of the game and that of an optimal outcome [Koutsoupias and Papadimitriou, 2009].

A game with multiple equilibrium has a large PoA even if only one of its equilibrium is highly inefficient. It has been proposed the *price of stability* (abbreviated PoS) of a game which is the ratio between the best objective function value of one of its equilibrium and that of an optimal outcome. In other words, the PoS is a measure of inefficiency designed to differentiate between games in which all equilibrium are inefficient and those in which some equilibrium are inefficient.

	Left	**Right**
Top	(2,1)	(0,0)
Bottom	(0,0)	(5,10)

Table 2.2: A simple game for calculating the PoA and the PoS

We illustrate how to calculate the PoA and the PoS in the simple game described in Table 2.2. In this game, there are two equilibrium points, (Top, Left) and (Bottom, Right), with values 3 and 15, respectively. The optimal value is 15. Thus, PoS = 15/15 = 1 while PoA = 3/15 = 1/5.

2.3 Representations of Games

In order to study any games, the first issue we must care about is how to represent it. A *game representation* is a data structure storing all information needed to specify a game. A game representation is said to be *fully expressive* if it is able to represent any arbitrary game. Finally the size of a game representation is the amount of data to express a game instance. In this section, we make a survey about the existing game representations in the literature. Note that we will generally focus on the fully expressive representations with their solving tools along.

2.3.1 Normal Form

The standard game representation of a static game is *normal form* [Fudenberg and Tirole, 1991] which is an n-dimensional matrix stating all utilities of all players for all joint strategy profiles in the game.

Definition 2.14 (Normal Form)**.** *A* normal form *representation of a game is an n-dimensional matrix which represents all players' utilities with regard to all strategy profiles of the game.*

The size of this representation is $n \times \Pi_{i \in \mathscr{P}} |S_i|$, which is $\Theta(nm^n)$ where $m = max_{i \in \mathscr{P}} |S_i|$. The matrix thus exponentially grows with the number of players. In case of two-player game, the mutilmatrix is called bimatrix and its size is $2 \times |S_1| \times |S_2|$ (see Figure 2.6).

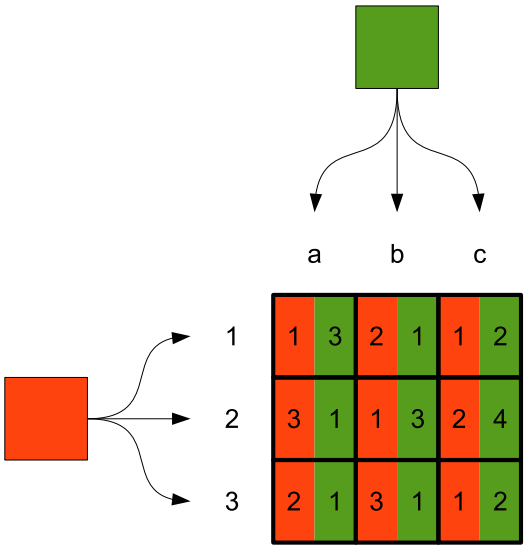


Figure 2.6: Normal form representation of a static game consisting of two players. The strategy set of player 1 *is* $\{1,2,3\}$ *while the one of player* 2 *is* $\{a,b,c\}$*. In each cell, the first number depicts the utility of player* 1 *while the second one is for player* 2*.*

Normal form is fully expressive. However, it is not a compact game representation. For example, in order to encode a game composed of 100 players and only 2 strategies per player, the normal form needs to store 100×2^{100} entries. The matrix easily becomes huge to express a game which is not really large. Accordingly, normal form is intractable for large games. This also causes a

significant difficulty to the widespread use of game-based modeling. Despite its intractability, normal form is still the basic representation for static games. All the algorithms for computing equilibrium as well as the complexity results in Section 2.2 are based on normal form.

2.3.2 Extensive Form

Another basic representation for games is *extensive form* [Fudenberg and Tirole, 1991]. This representation consists of a number of players, a game tree, and a set of payoffs. In the game tree, players are presented by nodes, strategies that all players can choose disposed in the edge and leave nodes are labeled by the utility of all players.

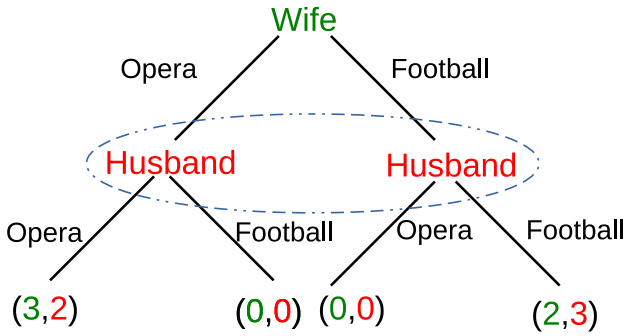


Figure 2.7: Extensive form of the game "battle of the sexes" *(Example 2.2). The first numbers in the leave nodes are the utilities for the wife while the second ones are for the husband.*

Figure 2.7 depicts the extensive form of the *battle of the sexes* game. This game has a second extensive form with the husband put at the root. However, since two players take decision at the same time, the two versions are exactly equivalent. In static games consisting of multiple players with multiple strategies taken by all players in parallel, the tree would become very complex. Indeed, we will consider only normal form as the classical representation of general static games in the rest of this thesis since it is absolutely equivalent to the extensive form associated.

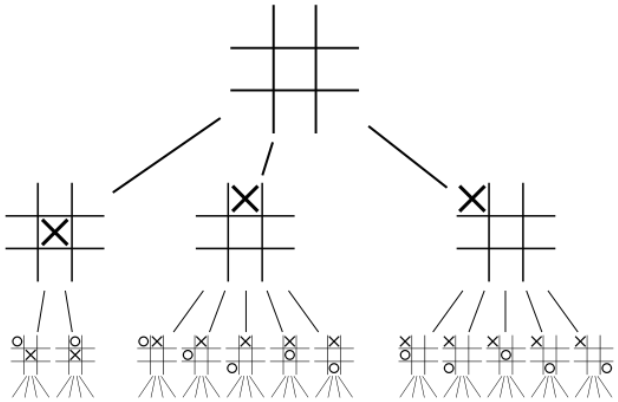


Figure 2.8: A part of an extensive form of Tic-tac-toe. *Source:* `http://en.wikipedia.org/wiki/Tic-tac-toe`.

The advantage of extensive form is that it specifies the sequence of strategies taken by all players. In other words, it expresses both the current status of a game as well as its history. Therefore, it is often used for encoding dynamic games. For example, a part of an extensive form of tic-tac-toe (Example 2.3) is given in Figure 2.8.

2.3.3 Graphical Games

As mentioned earlier, both normal form and extensive form are not succinct for encoding static games. There is thus a need to define compact representations of utility which is a challenge in computational game theory.

Several proposals have contributed to define more tractable representations, e.g. *Graphical Games* [Kearns et al., 2001, Daskalakis and Papadimitriou, 2006], for games in which a player's utility only depends on a subset of the other players. In a graphical game, each player i is represented by a vertex in a directed graph [4] G. We use $N(i) \subseteq \mathscr{P}$ to denote the *neighborhood* of player i in G - it means those vertices j such that the directed edge (i, j) appears in G. $N(i)$ includes i itself because player i's utility depends on his decision as well. Let s be a joint strategy, we use $\phi_i(s)$ to denote the projection of s onto just the players in $N(i)$.

Definition 2.15 (Graphical Games)**.** *A Graphical Game is a pair (G, $\mathscr{M}$), where:*

- *G is a directed graph over the vertices* $\{1,\ldots,n\}$*;*
- $\mathscr{M}$ *is a set of n* local game matrices*. For any joint strategy s, the local game matrix* $M_i \in \mathscr{M}$ *specifies the payoff* $M_i(\phi_i(s))$ *for player i, which depends only on the strategies taken by the players in* $N(i)$*.*

Example 2.9 (taken from [Bonzon, 2007])**.** *Adam, Dylan and Julie are invited to a party. Adam wants to accept the invitation. So does Dylan, but he will go if and only if Adam does go too. For Julie, she wants to go with Dylan but without Adam.*

Example 2.9 can be expressed by a graphical game which is composed of the directed graph in Figure 2.9 and a set of local matrices in Table 2.3.

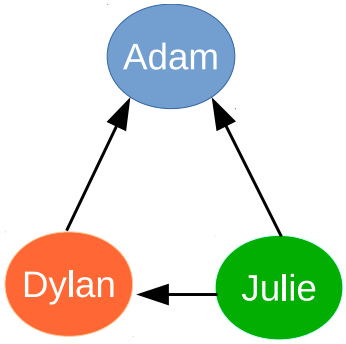


Figure 2.9: Directed graph of Example 2.9

[4] Graphical games are originally defined by [Kearns et al., 2001] on undirected graphs. While some later authors, e.g. [Daskalakis and Papadimitriou, 2006], propose the version using directed graphs that is presented here.

Adam		
Adam	0	(0)
	1	(1)

Dylan	*Dylan*	
	0	1
Adam 0	(0)	(0)
Adam 1	(0)	(1)

Julie (1 (yes))	*Dylan*	
	0	1
Adam 0	(0)	(1)
Adam 1	(0)	(0)

Julie (0 (no))	*Dylan*	
	0	1
Adam 0	(0)	(0)
Adam 1	(0)	(0)

Table 2.3: Local matrices of all the players in Example 2.9. In the tables, each player controls a set of two strategies where "1" states for "yes" and "0" for "no". For the utility of all players, "1" states for "go to the party" while "0" for "not go to the party".

In Example 2.9, Adam's utility depends on no one else. Hence, only his strategies appear in his local matrix (the table in top left). Then, because Dylan's utility is dependent of Adam, there are the strategies of Adam and Dylan in Dylan's local matrix (the table in top right). Finally, the final decision of Julie depends on both Adam and Dylan. Therefore, the strategies of all the players occur in the local matrix of Julie (the two tables in bottom). Indeed, instead of storing $3 \times 2^3 = 24$ entries in normal form, we only need to store $2+4+8 = 14$ entries in the three local matrices of the graphical game.

The total size of the representation is $\Theta(n \times m^{(\xi+1)})$ where ξ is the maximum in-degree of G [Jiang, 2011]. Graphical games allow to reduce the size of the utility tables because it only requires a number of parameters that is exponential in the size d of the largest local neighborhood. Hence, if $d \ll n$, then the graphical representation is significantly more compact than the normal form. However, it also means that this representation does not fit well to games that are not decomposable, i.e. when there exists a full interaction between all players in the games as the size would fall back to the normal form size.

For reasoning on graphical games, [Kearns et al., 2001] propose a polynomial-time algorithm to find approximate Nash equilibrium on tree graphs. Later, the same problem on games encoded by general graphs are solved by [Ortiz and Kearns, 2002, Vickrey and Koller, 2002]. [Elkind et al., 2006] show that mixed Nash equilibrium can be computed in quadratic time if the underlying graph is a path, and therefore in polynomial time if the underlying graph has maximum degree 2. On computing PNE, [Gottlob et al., 2005] try to enforce several restrictions on graphical games. They have found out that the existence of a PNE and computation problems are tractable for games that simultaneously have small neighborhood and bounded hypertree width. In this case, finding PNE is feasible in polynomial time. Recently, [Chapman et al., 2010] attempt to find the "best" pure Nash equilibrium but it is limited on tree graph.

2.3.4 Action-Graph Games

Based on graphical games, Bhat and Leyton-Brown propose *Action-Graph Games* (AGG) [Bhat and Leyton-Brown, 2004] that exploit properties of certain games like context indepen-

dence and anonymity to achieve a compact representation. Each node in the graph of AGG is represented by action (strategy), not player (Definition 2.16).

Definition 2.16 (Action Graph)**.** *An* Action Graph $G = (\mathscr{A}, E)$ *is a directed graph where* $\mathscr{A}$ *is set of nodes. We call each node* $\alpha \in \mathscr{A}$ *an action and* $\mathscr{A}$ *the set of distinct strategies. E is a set of directed edges where self edges are allowed. We say* α' *is a neighbor of* α *if there is an edge from* α' *to* α*, i.e.* $(\alpha', \alpha) \in E$.

[Jiang, 2011] extends the basic framework to three different variants of AGG: (1) the basic AGG representation (AGG-$\emptyset$); (2) AGG with function nodes (AGG-FNs); (3) AGG-FNs with additive structure (AGG-FNA). Nevertheless, for understanding the principal idea of this game representation, extracting the most basic one (AGG-$\emptyset$, Definition 2.17) is enough.

Definition 2.17 (Action-Graph Games)**.** *An* Action-Graph Game *(abbreviated AGG) is a 4-tuple* $(\mathscr{P}, S, G, u)$ *where:*

- $\mathscr{P}$ *is a finite set of players;*
- $S = \Pi_{i \in \mathscr{P}} S_i$ *is the set of strategy profiles;*
- $G = (\mathscr{A}, E)$ *is an action graph, where* $\mathscr{A} = \cup_{i \in \mathscr{P}} S_i$ *is the set of distinct actions;*
- $u = (u^\alpha)_{\alpha \in \mathscr{A}}$ *is a tuple of* $|\mathscr{A}|$ *functions where* u^α *is the utility function for action* α.

In AGG, each player's utility is calculated according to an function of the node he chooses and the numbers of tokens placed on the nodes that neighbor chooses in the graph. Then, like graphical games, matrix are still required to store utilities. The size of matrix is worst-case exponential. Thus, the size of the utility functions determines whether an AGG can be tractably represented.

On the complexity of action-graph games, in [Daskalakis et al., 2009b], the authors demonstrate that, generally, determining whether an AGG has a PNE is NP-complete, even for symmetric AGG-$\emptyset$ with bounded in-degrees, and PPAD-complete with mixed Nash equilibrium. This motivates Jiang and Leyton-Brown to identify tractable subclasses of AGG. They therefore propose a dynamic programming approach and show that if the AGG-$\emptyset$ is symmetric and the action graph has bounded tree width, their algorithm determines the existence of PNE in polynomial time [Jiang and Leyton-Brown, 2007]. They also propose in [Jiang and Leyton-Brown, 2006] a polynomial time algorithm (in the size of the AGG representation) for computing expected payoff under mixed strategy profiles.

2.3.5 Boolean Games

Graphical games and AGG are the interaction-based game representations. Another approach is to find a language-based representation, from which *Boolean Games* emerge as the most successful.

Boolean games are initially proposed by [Harrenstein et al., 2001]. Formally, a boolean game is composed of a finite set of players. Each player controls a set of propositional variables to achieve his goal (also called *preference*) which is encoded by a propositional formula.

Definition 2.18 (Boolean Games)**.** *A* Boolean Game *is a 3-tuple*$(\mathscr{P}, V, \phi)$ *where:*

- $\mathscr{P} = \{1,\ldots,n\}$ *is a finite set of players;*
- $V = (V_i)_{i\in\mathscr{P}}$ *such that* $i \neq j \leftrightarrow V_i \cap V_j = \emptyset$ *where* V_i *is a set of propositional variables controlled by player i;*
- $\phi = \{\varphi_1,\ldots,\varphi_n\}$ *is a set of goals, where each* φ_i *for player i is a satisfiable formula.*

Example 2.10 (Example 2.9 continued)**.** *The situation in Example 2.9 can be expressed as a boolean game as follows.*

- $\mathscr{P} = \{1,2,3\}$ *where* 1 *stands for Adam,* 2 *for Dylan and* 3 *for Julie;*
- $\forall i \in \mathscr{P}, V_i = \{x_i\}$ *where* x_i *is true if player i will go to the party ;*
- *The goals of all the players are following:* $\varphi_1 = x_1, \varphi_2 = x_1 \leftrightarrow x_2, \varphi_3 = \neg x_1 \wedge x_2 \wedge x_3$.

The advantage of boolean games over the interaction-based representations such as graphical games or action-graph games is that they succeed in dealing with the worst case of the previous game representations, i.e. in games where there is a full interaction between all players. However, in boolean games, each player owns a SAT problem which defines his satisfaction. There is no means to specify inside the language a non-boolean utility. This is why it is required to provide an external way to define preferences. In [Bonzon et al., 2009b], CP-nets [Boutilier et al., 2004] are used to define players' preferences. It is the drawback of boolean game with regard to graphical games or action-graph games because the two later can express non-boolean utility without using any external tool.

On the complexity, [Dunne and van der Hoek, 2004] prove that determining whether a boolean game has a PNE is Σ_2^p-complete, while checking whether a particular strategy profile is a PNE is co-NP-complete. Some work has been devised towards finding tractable classes of boolean games. In [Bonzon et al., 2009a], the authors focus on benefiting the independence between players in boolean games, i.e. decomposable boolean games, by applying the principal techniques of graphical games for boolean games. Such games can be divided into multiple subgames which are smaller and easier for solving. Finally, the global equilibrium is a joint equilibrium of all the subgames. The tractable classes proposed by [Dunne and Wooldridge, 2012] comes from two issues: development of an alternative solution concept and identification of the specific classes which are tractable. One hand, they have relaxed the notion of PNE by defining a new equilibrium concept named *k-bounded Nash equilibrium* for which no player can benefit by altering fewer than k variables. On the other hand, specific classes have been determined by performing some restrictions on games that make checking for deviation of an individual player become computationally tractable.

Many extensions of boolean games have been proposed for years. While in a basic boolean game, each player is free to choose any strategy for achieving his goal, [Dunne et al., 2008] assume that the strategies available to a player have some cost. Thus, the secondary purpose of all players is to minimize this cost. They also propose preferences over possible coalitions between players. All these ideas implied to an extensive boolean game named *cooperative boolean games* whose solution concepts are defined by core and stable sets. Sharing the same idea of cooperation, [Bonzon et al., 2012] study the properties of efficient coalitions in boolean games. Their main

contributions are to demonstrate how a coalition in boolean games is effective, in terns of the power about coalitions.

More recently, [Wooldridge et al., 2013] have also extended the basic framework of boolean games by introducing an external element what they called *taxation schemes*. The idea of taxation is inspired by the incentive mechanisms found in human societies which aims at perturbing the preferences of the players in certain ways. Like in [Dunne et al., 2008], these schemes impose a cost on every possible strategy that a player can choose. The main solution concept studied is still PNE and it is also shown that Nash outcome verification is co-NP-complete. For implementing the taxation schemes on a boolean game, Wooldridge et al. suppose that there is an external *principal* who accomplishes these tasks (e.g. like government in social life). This arises a new problem called *Implementation*. It is proved in the same paper that *Implementation* is Σ_2^p-complete.

A variable must be under control of one and only one player in the basic boolean games. [Grant et al., 2014] propose to add a set of *environment variables* in which each variable is uncontrolled by any player but imposes impact on the players' goals. The players make decisions according to their belief about the possible values assigned to the environment variables by an external player called *principal*. The game is therefore manipulated by sharing information since at each moment, the principal will announce the possible values of the environment variables to the players who therefore adapt their decision to the announcement. The solution concept of boolean games in this case is defined by *Nash stability* [Grant et al., 2010] which is derived from the PNE notion.

Iterated Boolean Games [Gutierrez et al., 2013] are also another extension of boolean games. An *iterated boolean game* can be considered as an infinite series of rounds, where at each round each player makes an assignment to his controlled variables. Goals in this game are expressed as formula of Linear Temporal Logic [Emerson, 1990]. Gutierrez et al. also prove that checking whether a strategy profile is a PNE and whether a game has at least one PNE are complete problems for PSPACE and 2EXPTIME, respectively.

For reasoning on basic boolean games, the first methods for computing all PNE or core elements have been proposed by [de Clercq et al., 2014a] in using *disjunctive answer set programming* [Brewka et al., 2011]. Their idea is to transform a boolean game into an answer set program. Henceforth, by using the saturation techniques [Baral, 2003], the obtained answer set will coincide with the set of PNE or the cores. They also note that their grounded disjunctive answer set programming has the same complexity of determining whether a boolean game has a PNE, yielding Σ_2^p-complete [Baral, 2003].

2.3.6 Other Representations

Besides the game representations described above, there are also a few attempts to design compact representations in the literature. In [de Vos and Vermeir, 1999], preferences are represented by a *choice logic program* for which stable models coincide with Nash equilibrium. In a similar

way, [Foo et al., 2004] represent games by LPOD (Logic Program with Ordered Disjunction). However, these two formalisms actually encode the reaction operator in logic and do not provide a compact representation in the worst case.

In addition, congestion games [Rosenthal, 1973], routing games [Roughgarden, 2007] are other kinds of specific games that enjoy a compact representation, as well as symmetric games [Dasgupta and Maskin, 1989], anonymous games [Brandt et al., 2009], local-effect games [Leyton-Brown and Tennenholtz, 2003]. Nevertheless, they are not general games and not fully expressive.

2.4 Existing Software in Game Theory

There are still not many software tools available for solving games as well as evaluating game-theoretic algorithms. Along with designing modeling tools, our long-term goal is to implement a robust software for reasoning on games. We recall here a generator of benchmark for games as well as a few existing solvers for several game representations in the literature.

Gamut [Nudelman et al., 2004] is a java library for easily generating a benchmark of games. Gamut includes thirty-five base game classes already studied in the literature. It also allows to choose the parameter options for games and generates random utilities as well. The main purpose of Gamut is for testing game-theoretic algorithms on the normal form representation. Gamut is available at `http://gamut.stanford.edu/`.

Gambit [McKelvey et al., 2014] is a C++ library including many existing algorithms for solving games in the literature. It provides both command line interface and graphical interface. The input file of Gambit is based on the normal form representation. Although this solver focuses on finding mixed Nash equilibrium, the authors also implemented a simple algorithm for finding pure Nash equilibrium that can be launched by the command `gambit-enumpure`. Gambit is available at `http://www.gambit-project.org/`.

Jiang and Leyton-Brown have implemented a solver for solving their action-graph games. Working on the text input file called "AGG file format", the solver provides both command line tools and graphical interface for finding Nash equilibrium. In addition, it also solves *Bayesian Action-Graph Games* (abbreviated BAGG), an extension of AGG that compactly represents Bayesian games [Jiang and Leyton-Brown, 2010]. The game instances in Gamut have also been generated in the AGG representation. In addition, both the AGG and the BAGG file format have been integrated into the Gambit solver. This solver is available at `http://agg.cs.ubc.ca/`.

A complete solver for boolean games implemented and maintained by Bauters and De Clercq is available at `http://www.cwi.ugent.be/BooleanGamesSolver.html`. This solver allows to compute PNE or core elements in boolean games. It is capable of finding Pareto optimal equilibrium as well [de Clercq et al., 2014a].

Chapter 3

Constraint Programming

Contents

3.1 Basic Notions 37

3.2 Reasoning 39

3.2.1 Backtracking Search 39

3.2.2 Local Search 43

3.3 Constraint Programming in Game Theory 44

Constraint programming (abbreviated CP) is a programming paradigm which encodes and solves problems using variables and constraints. The work in CP have been widely described in the literature. Hence, we will only recall the essential notions as well as the solving techniques that will be used or directly related to this thesis. The interested readers could find much more background knowledge of constraint programming in [Rossi et al., 2006]. At the end of this chapter, we mention several existing CP attempts in game theory.

3.1 Basic Notions

Let V be a set of variables and $D = (D_x)_{x \in V}$ be the family of their (finite) domains. For $W \subseteq V$, we denote by D^W the set of tuples on W, namely $\Pi_{x \in W} D_x$. Projection of a tuple (or a set of tuples) on a variable (or a set of variables) is denoted by $|$: for $t \in D^V$, $t|_x = t_x$, $t|_W = (t_x)_{x \in W}$. For $W, U \subseteq V$, the join of $A \subseteq D^W$ and $B \subseteq D^U$ is $A \bowtie B = \{t \in D^{W \cup U} \mid t|_W \in A \wedge t|_U \in B\}$. These notions will be used in the rest of this thesis.

A CP problem is itself a constraint network. A *constraint* $c = (W, T)$ is a pair composed of a subset $W = var(c) \subseteq V$ of variables and a relation $T \subseteq D^W$. In other words, a constraint network is formally defined as a *Constraint Satisfaction Problem* which is the essential concept in constraint programming.

Definition 3.1 (Constraint Satisfaction Problem)**.** *A* Constraint Satisfaction Problem *(abbreviated CSP) is composed of a finite set of variables V and a finite set of domain D with $D(x)$ is the domain of variable $x \in V$, together with a finite set of constraints C, each on a subset of V.*

Then the solution concept for constraint satisfaction problems is defined as follows.
Definition 3.2 (Solution)**.** *A* solution *is an assignment of all variables that satisfies all the constraints simultaneously.*
Example 3.1 (CSP)**.** *Given a CSP with the set of variables $V = \{X, Y, Z\}$, the variables' domains $D(X) = D(Y) = D(Z) = \{1, 2, 3\}$, the set of constraints $C = \{X > Y; Y > Z\}$. The unique solution is $X = 3, Y = 2, Z = 1$.*

Constraint programming is a powerful modeling framework with many facilities. One of the most useful is *global constraints* [van Hoeve and Katriel, 2006]. A constraint capturing the relations between a non-fixed number of variables is called *global constraint*. An example is the constraint `all_different`$(x_1, \ldots, x_n)$ [Lauriere, 1978], which specifies that any pair of variables must be assigned different values. Many global constraints are described in a catalog accessible at `http://sofdem.github.io/gccat/`. Generally, a global constraint can be replaced by a conjunction of a set of simpler constraints. Using global constraint not only helps to make the model more explicit and declarative - therefore, it simplifies the modeling tasks - but also facilitates the work of constraint solver by providing a better view of the structure of the problem. In addition, many specific efficient techniques for solving global constraints have been proposed, for example [Puget, 1998, Mehlhorn and Thiel, 2000], to name a few.

We have mentioned the basic notions of constraint programming through Constraint Satisfaction Problems. Many extensions of classical CSP have been developed. This survey will be continued with *Optimization* [Tsang, 1993], an extension directly impacts to our work in this thesis. In many applications, we would like to find a solution to a CSP that is optimal with respect to certain criteria. In this case, the optimization condition needs to be embedded over the CSP.
Definition 3.3 (Constraint Optimization Problem)**.** *A* Constraint Optimization Problem *(abbreviated COP) is a CSP P defined on the variables $x \in V$ together with an* objective function f : $\Pi_{x \in V} D_x \rightarrow \mathbb{R}$.

An *optimal solution* to a minimization (maximization) COP is a solution d to P that minimizes (maximizes) the value of $f(d)$. The objective function value is often represented by a variable, for example z, together with the "constraint" $\min(z)$ or $\max(z)$ for a minimization or a maximization problem, respectively.
Example 3.2 (COP)**.** *Given a COP with the set of variables $V = \{x, y\}$, the domains $D(x) = D(y) = \{1, 2\}$, the set of constraints $C = \{x \geq y; x \neq 2y\}$. The optimization condition is* $\max(x+y)$.

In Example 3.2, if the optimization condition is not taken into account, the problem is a CSP in which there are two solutions, $(x = 1, y = 1)$ and $(x = 2, y = 2)$. Among them, $(x = 2, y = 2)$ is the best solution which maximizes the optimization condition. The problem in Example 3.2 thus has only one solution, i.e. $(x = 2, y = 2)$.

3.2 Reasoning

Determining whether a given CSP has a solution is NP-complete [Cohen and Jeavons, 2006]. It is in NP because a solution of a CSP is an assignment of all variables which can be represented in polynomial space with the number of variables. We can henceforth verify in polynomial time that the solution satisfies all constraints. Then the completeness derives from the theorem of Cook [Cook, 1971] which stipulates that the satisfaction problem of the conjunction clauses, which is a restriction of CSP, is NP-complete in itself.

There are three main algorithmic techniques for solving constraint satisfaction problems: backtracking search [van Beek, 2006], local search [Hoos and Tsang, 2006], and dynamic programming [Dechter, 2006]. We recall, in this section, some techniques of the two first solving methods that inspire us to propose techniques for finding equilibrium in our constraint games.

3.2.1 Backtracking Search

An algorithm for solving Constraint Satisfaction Problems can be either complete or incomplete. Backtracking search algorithms are probably the the most canonical complete algorithms in constraint programming. From the proposals in the early day [Davis et al., 1962, Golomb and Baumert, 1965], many techniques have been suggested and evaluated for improving the backtracking search. In this section, we only survey several techniques, including constraint propagation, heuristic for variable and value ordering, backjumping and branch-and-bound techniques.

Constraint Propagation

In constraint programming, local consistency is the consistency of subsets of variables or constraints. In other words, local consistency conditions require that all consistent partial evaluations can be extended to another variable such that resulting assignment is consistent. A partial evaluation is consistent if it satisfies all constraints whose scope is a subset of the assigned variables. Several local consistencies are proposed in the literature, such as node consistency, arc consistency, and path consistency, to name a few.

A fundamental insight in improving the performance of backtracking algorithms on CSP is that local inconsistency can lead to unproductive search. Namely, inconsistency can be the reason for many deadends in the search. This motivates many algorithms to maintain the local consistency during search. The generic scheme is to perform constraint propagation [Bessière, 2006] removing local inconsistencies. Constraint propagation is the central to the process of solving a constraint problem.

Backtracking search is based on search tree. Generally, the search is divided into two main steps which repeat successively: Propagate by removing the local inconsistencies and split and solve

sub-problems recursively. The backtracking search is depicted in Algorithm 3.1 which launches the recursive algorithm in Algorithm 3.2.

Algorithm 3.1. Backtracking algorithm for solving a CSP

```
1: V: List of variables
2: D: List of domains associated to variables
3: C: Constraints

4: function BT(V,D,C) : boolean          ▷ return true if the CSP has at least a solution
5:     return rBT(∅, V, D, C)
6: end function
```

Algorithm 3.2. Solving sub-problems recursively

```
1: A: List of assignments
2: X: List of variables non assigned
3: D': List of current domains of variables in X
4: C: List of constraints

5: function RBT(A,X,D',C) : boolean   ▷ return false if there is no solution in the sub-problem
6:     if X = ∅ then
7:         return true
8:     end if
9:     D' ← propagate(A,X,D',C)
10:    if ∃x ∈ X such that D'(x) = ∅ then
11:        return false
12:    end if
13:    x ← A variable in X
14:    for all v ∈ D'(x) do
15:        return rBT(A ∪ {x ← v}, X \ {x}, D' \ D'(x), C)
16:    end for
17: end function
```

In Algorithm 3.2, a solution is found if all variables are instantiated by one of their values (line 6-8). Otherwise, we perform the function of propagation in line 9. This function removes the local consistencies in the subgame. It thus usually reduces the current domains of the variables non assigned. In Example 3.1, when X is assigned to 2, then the domain of Y becomes $D(Y) = \{1\}$. Namely the two values, 2 and 3, are removed from $D(Y)$ due to the constraint $X > Y$.

If after having propagated, there exists a variable whose domain is empty, then the assignment A can not appear in a solution because it violates at least one constraint $c \in C$ (line 10-12). In other words, there in no solution on this branch. In Example 3.1, when X is assigned to 1, then the domain of Y becomes $D(Y) = \emptyset$ due to the constraint $X > Y$. It means, there is no solution when $X = 1$.

Otherwise, we choose a variable non instantiated in X (line 13). Then, for all values in its current domain, we add branches for new nodes. We then recursively solve new sub-problems until a

solution is found, or its absence is proven (line 14-16).

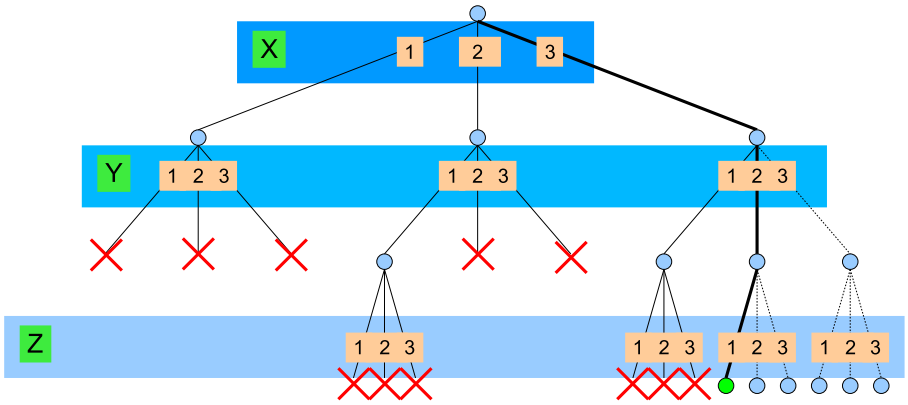


Figure 3.1: Backtracking algorithm runs on Example 3.1.

Figure 3.1 illustrates how backtracking search works on Example 3.1. Thanks to constraint propagation, many deadends stated by cross are discovered without reaching to the leaves. A solution in this example is noted by the bold lines while the dash branches depict the unexplored search space.

Heuristics for Variable and Value Ordering

It has been shown that in many problems, efficient heuristics for variable and value ordering can make a significant impact on the performance of backtracking algorithms [Bacchus and van Run, 1995, Gent et al., 1996]. There are two kinds of heuristics. The first one is called *static heuristics* in which we force a strict ordering for variable and value at the beginning of search. The second one, which captures much more research attentions, is called *dynamic heuristics* in which the variable and value ordering is adapted to the information recorded during search. In other words, it could by seen as a part of search process.

Dynamic variable ordering [Golomb and Baumert, 1965, Bacchus and van Run, 1995] aims at replying this question: given a node p, how to select a non-instantiated variable x to branch next? In the early days, [Golomb and Baumert, 1965, Haralick and Elliott, 1980] propose a heuristic named *dom* which suggests to choose the variable with the minimal domain left. Later, [Brélaz, 1979] uses a new notion called *degree* of an unassigned variable x which is the number of constraints which involves x and at least one other unassigned variable. He therefore proposes the $dom + deg$ heuristic which chooses the variable with the smallest remaining numbers of values and the variable with the highest degree. Also based on degree, the dom/deg heuristic proposed by [Bessière and Régin, 1996] divides the domain size of a variable by the degree of the variable and chooses the variable which has the minimal values numbers. In the $dom/wdeg$ heuristic [Boussemart et al., 2004] , the domain size is divided by the weighted degree. A weight, initially set to one, is associated with each constraint. Every time a constraint is responsible for a deadend, the associated weight is incremented. The weighted degree is the sum of the weights of the con-

straints which involves x and at least one unassigned variable. In other ways, several heuristics minimize specific functions, such as [Brown and Jr., 1982, Gent et al., 1996] or exploit structure-guided variable ordering heuristics for CSP represented as a graph [Freuder, 1982, Zabih, 1990].

Given a node p, the variable ordering heuristic already chooses variable x to branch next, it is required to choose value a to assign to x from its remaining domain (after having been propagated). Value ordering heuristics have been proposed in the literature which mainly based on either estimating the number of solutions or estimating the probability of a solution, for each choice of value a for x. e.g. [Ginsberg et al., 1990, Frost and Dechter, 1995, Minton et al., 1992], to name a few.

Backjumping

While backtracking always goes up one level in the search tree when all values for a variable have been tested, backjumping [Gaschnig, 1979] is a form of non-chronological backtracking algorithm as it may go up more levels (see Figure 3.2). Therefore, backjumping allows to reduce the search space, indeed, to offer more efficient performance.

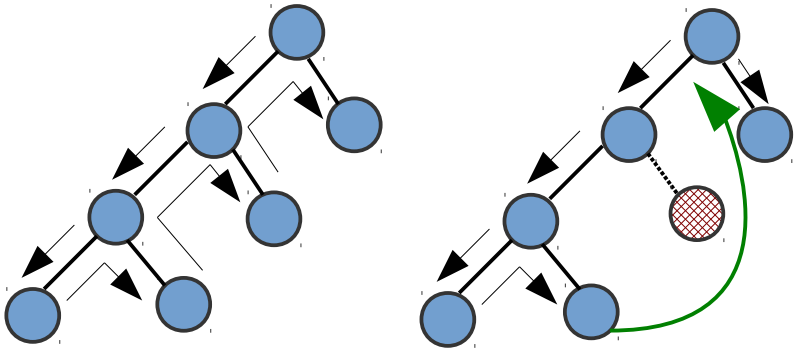


Figure 3.2: Backjumping. The figure in left side shows a search tree visited by regular backtracking. The backjumping occurs in the right side figure in which the dash node is not visited.

While constraint propagation and inconsistency removing could be seen as look-ahead techniques which try to detect deadends early by pruning values from future variables, backjumping techniques are look-back ones which try to deal with deadends in an intelligent way by recording and exploiting the reasons for failures. We have applied this idea for our complete solving tools to perform more pruning during search. This pruning is, of course, different to the standard pruning in the literature.

For the existing backjumping techniques in CP, by discovering and maintaining nogoods during search, [Gaschnig, 1979, Bruynooghe, 1981, Rosiers and Bruynooghe, 1986] design several algorithms allowing to backjump from deadends. Based on graph, [Dechter, 1990] proposes the first algorithm to jump back at internal deadends. Conflict-directed backjumping algorithm (CBJ) proposed by [Prosser, 1993] can do the same. Moreover, CBJ has also been combined with constraint propagation. [Prosser, 1993] proposes FC-CBJ, an algorithm combining forward checking

and conflict-directed backjumping. Furthermore, he also presents another algorithm called MAC-CBJ which maintains arc consistency and CBJ simultaneously. However, this algorithm only handles binary constraints. It is extended by [Chen, 2000] for general constraints.

Branch-and-bound Techniques

Backtracking search algorithms have been applied for solving Constraint Optimization Problems as well. Because the goal is to find an optimal solution which maximizes (minimizes) the objective function f, after having found all solutions of Constraint Satisfaction Problems, we need perform a complementary task to take the *best solution* between them. It is the basic technique for solving COP.

Fortunately, it is possible to use the function f to prune the search space in a technique well-known as *Branch-and-bound* (or B&B) [van Beek, 2006]. With B&B, in an optimization problem (say maximization of f), we can add the constraint $f > v$ in the remaining search after finding a solution $f = v$. This constraint helps to propagate and cut branches that lead to a solution whose f value is less than the one previously found. Therefore, the technique allows to reduce the search space. Note that this classical B&B technique finds only one best solution, not all of the best solutions of the problem.

3.2.2 Local Search

Local search [Hoos and Tsang, 2006] is classified into the category of incomplete algorithms, meaning that it does not guarantee to always find a solution as well as prove its optimality in every combinatorial problem. Nevertheless, it provides the basis for some of the most successful and versatile methods for solving the large and difficult problem instances encountered in many real-life applications. Despite impressive advances in systematic, complete search algorithms, local search methods in many cases represent the only feasible way for solving these large and complex instances.

Given a CSP C, the idea of local search for finding one solution in CSP is rather simple. For starting, an *initial point* which is a complete variables assignment is randomly chosen. The possible points around the current point which is different at least one value assigned to a variable are called *neighbors*. Deciding which neighbor will be chosen is performed by an *evaluation function heuristic*. Then, at each *step*, we move to a neighbor which will be served as the next *current point*. The process is iterated until a *termination criterion* is reached. The termination criterion could be the solution detected or user-specific such as the max step or time exceeded. The formal definition of local search is described in [Hoos and Stützle, 2004].

Probably, the first algorithms which improve over the simple *iterated improvement process* in CSP are *min-conflicts heuristics* (abbreviated MCH) [Minton et al., 1992]. In the algorithms, each at step, we choose a random variable v from a set which records all variables appeared in at least

one constraint violated. Then, a value a in the current domain of x will be selected such that the number of constraint violated is minimal. It has been well-known that MCH are essentially incomplete. They may be stagnated in local optima and is not available to escape by itself.

In order to avoid getting stuck in local optima of the given evaluation function, in the literature, [Glover, 1989] introduces *tabu search* which uses a *tabu tenure* to memorize the search space already tested. Hence, it may forbid to recheck the same state twice at one point and allows to escape the trap of local optima. Tabu search is also used for solving constraint problems. [Steinmann et al., 1997, Stützle, 1998] apply the idea of tabu search with *Min Conflict Heuristics with Tabu Search* (abbreviated TMCH) algorithms. TMCH work like MCH except that after the value of variable v is changed from a to a' then the variable/value pair (v,a) is declared tabu for next x steps, where x is the *tabu tenure* parameter. the variable/ value pair involved in each step.

By modifying evaluation function heuristic when the search processes, *Dynamic Local Search* [Hoos and Stützle, 2004] leads to a set of *Penalty-Based Local Search Algorithms* like GENET [Davenport et al., 1994], Breakout Method [Morris, 1993] or Guided Local Search [Voudouris and Tsang, 1996].

Evolutionary algorithms have widely used for CSP [Hao and Dorne, 1994, Rojas, 1996], e.g. genetic algorithms [Hao and Dorne, 1994, Lau, 1999]. Furthermore, ant colony optimization inspired by the path-finding behavior of ants [Dorigo and Stützle, 2004], is applied in CSP by [Solnon, 2001]. Besides general problems, local search has also solved several specific types of CSP instances. For example, *simulated annealing algorithms* [Kirkpatrick et al., 1983] which allow non-improving search step with certain probabilities or *iterated local search algorithms* for the graph coloring problem [Johnson et al., 1989, Paquete and Stützle, 2002].

As pointed out in Chapter 2, there are still not many tools available for solving games. In particular, it is almost intractable for very large games. It seems that a main reason is the lack of a good compact representation. Even for compact representations such as graphical games or action-graph games, matrix storing the payoffs are still required. In benefiting from the compactness of constraint games, together with local search, our wish is a robust local search solver which is able to break out this existing limitation in game theory.

3.3 Constraint Programming in Game Theory

There are a few attempts to use constraint programming in game theory. In [Apt et al., 2008], the authors try to bridge a gap between the concept notions used in the area of soft constraints and strategic games. They show that, for a class of soft constraints including weighted constraints, every optimal solution is a Nash equilibrium. Furthermore, the optimality for soft constraints and Pareto efficient joint strategy coincides in general.

In [Bordeaux and Pajot, 2004], it has been proposed to compute a mixed Nash equilibrium using continuous constraints. More recently, [Soni et al., 2007] define a new constraint satisfaction

problems for finding approximate equilibrium in graphical games. They also provide an algorithm to compute Bayes-Nash equilibrium in one-shot games for incomplete information games. Other types of equilibrium such as Stackelberg equilibrium have been investigated within the Quantified Constraint Satisfaction Problems [Bordeaux and Monfroy, 2002, Benedetti et al., 2008]. Quantified Constraint Satisfaction Problems is an extension of CSP in which some variables are universally quantified. They are often used for modeling the situations which include the uncertainty features.

In [Gottlob et al., 2005], the authors present a CSP encoding the reaction operator in graphical games. Each player has a constraint stating his best reaction for each possible strategy profile of the other players. A solution of this CSP is thus a Nash equilibrium. However, this is mainly a theoretical tool because the constraints are huge and not natural to model.

Some other formalisms try to solve a combinatorial problem by multiple agents, either by letting the agents dynamically select their variable like in SAT-Games [Zhao and Müller, 2004] and Adversarial CSP [Brown et al., 2004] or with a predefined assignment of variables to agents like in Distributed Constraint Satisfaction Problems and Distributed Constraint Optimization Problems [Faltings, 2006, Leite et al., 2014].

Definition 3.4 (Distributed Constraint Satisfaction Problems)**.** *A* Distributed Constraint Satisfaction Problems *(abbreviated DisCSP) is a 4-tuple ($\mathscr{P}$,V,D,C) where:*

- $\mathscr{P} = \{1,\ldots,n\}$ *is a finite set of n players;*
- $V = \{v_1,\ldots,v_n\}$ *is a set of n variables;*
- $D = (D_i)_{i\in\mathscr{P}}$ *is a set of n domains where D_i is the domain of v_i, respectively;*
- $C = \{c_1,\ldots,c_m\}$ *is a set of m constraints.*

Unlike in the classical CSP setting, each variable v_i of DisCSP is controlled by only player i. That means only player i is available to assign a value in D_i to v_i. He therefore knows the domain of his variable and all constraint involving v_i which can be reliably communicated with all other players. Thus, the main challenge in DisCSP is to develop distributed algorithms that solve the CSP by exchanging messages among players.

Distributed Constraint Satisfaction Problems can be extended to *Distributed Constraint Optimization Problems* (abbreviated DCOP). DCOP include a group of agents who must distributedly choose values for a set of variables such that the cost of a set of constraints over the variables is either minimized or maximized.

Both DisCSP and DCOP can be structurally represented as a graph, where the variables correspond to nodes and the constraints between pairs of variables are the edges. Naturally, there are several studies involving in the combination between the concept of graphical games and distributed constraints problems. In [Grubshtein and Meisels, 2012], graphical games are modeled as Distributed Constraint Satisfaction Problems with unique k-ary constraints in which each agent is only aware of its part in the constraints. The authors also propose in the same paper an asynchronous backtracking algorithm to find ε-Nash equilibrium of the problems. Conversely,

[Maheswaran et al., 2004] introduce the distributed algorithms for DCOP by decomposing DCOP into a graphical game and by investigating various evolutionary algorithms to compute Nash equilibrium.

Part II

Contributions

Chapter 4

Constraint Games

Contents

4.1 Modeling Framework 50
4.1.1 Constraint Satisfaction Games 50
4.1.2 Constraint Satisfaction Games with Hard Constraints 53
4.1.3 Constraint Optimization Games 54
4.1.4 Constraint Optimization Games with Hard Constraints 56
4.1.5 Usefulness and Compactness of Constraint Games 56
4.2 Applications 59
4.2.1 Games Taken from the Gamut Suite 59
4.2.2 Games Taken from Real Life 62
4.3 Solving Framework 68
4.3.1 ConGa Solver 69
4.3.2 Experimental Environment 70
4.4 Conclusion 70

In this chapter, we introduce the framework of *Constraint Games* to model strategic interaction between players. A constraint game is composed of a set of variables shared by all the players. Among these variables, each player owns a set of *decision variables* he can control while his preference depends on the decisions taken by the remaining players as well. Each player may try to improve his *utility* by choosing an assignment that optimizes his preference. Constraint games include four variants in which optimization and/or hard constraints are allowed or not. The solution concept for constraint games is defined by Nash equilibrium. They are situations in which no player may improve his preference unilaterally. We show the practical utility of the framework by modeling a few classical games in the literature as well as realistic problems. In addition, we are also interested in solving tools for constraint games. We thus propose a set of search methods for constraint games which will be detailed in the following chapters. In this chapter, we only mention several general characteristics of our solving framework.

4.1 Modeling Framework

Our main idea is to use CSP for expressing players' utilities. In this section, we present *Constraint Games*, a fully expressive representation for encoding games. Moreover, constraint games also provide a means to compactly model static games using constraints.

4.1.1 Constraint Satisfaction Games

Let $\mathscr{P}$ be a set of n players and V a finite set of variables. The set of variables is partitioned into *controlled* variables $V_c = \bigcup_{i \in \mathscr{P}} V_i$ where V_i is the subset of variables controlled by player i, and V_E the set of *uncontrolled* or *existential* variables ($V_E = V \setminus V_c$).

In the spirit of boolean games, at the beginning, we start our work by trying to encode games in which utilities are only boolean. In other words, the satisfaction of each player expressed by his preference is simply "yes" or "no". The basic representation of constraint games - *Constraint Satisfaction Games* - is henceforth designed for such games.

Definition 4.1 (Constraint Satisfaction Game)**.** *A* Constraint Satisfaction Game *(abbreviated CSG) is a 4-tuple* $(\mathscr{P}, V, D, G)$ *where:*

- $\mathscr{P}$ *is a finite set of players;*
- V *is a finite set of variables composed of a family of disjoint sets* (V_i) *for each player* $i \in \mathscr{P}$ *and a set* V_E *of* existential *variables disjoint of all the players variables;*
- $D = (D_x)_{x \in V}$ *is the family of their domains;*
- $G = (G_i)_{i \in \mathscr{P}}$ *is a family of CSP on* V.

The CSP G_i is called the *goal* of the player i. The intuition behind CSG is that, while player i can only control his own subset of variables V_i, his satisfaction will depend also on the variables controlled by all the other players. The intuition behind existential variables is that no player controls them and they are existentially quantified [1].

In terms of semantic, Constraint Satisfaction Games are almost equivalent to boolean games. The unique difference is that variables are no longer limited to propositional ones. In Chapter 2, we demonstrated how to encode Example 2.9 by graphical games (Figure 2.9 and Table 2.3) and by boolean games (Example 2.10). Now, this example will be encoded by our Constraint Satisfaction Games in Example 4.1.

Example 4.1 (Example 2.9 continued)**.** *The situation in Example 2.9 can be expressed as a Constraint Satisfaction Game as follows.*

- $\mathscr{P} = \{1,2,3\}$ *where* 1 *stands for Adam,* 2 *for Dylan and* 3 *for Julie;*
- $\forall i \in \mathscr{P}, V_i = \{x_i\}$ *where* x_i *is controlled by player i;*
- $\forall i \in \mathscr{P}, D(x_i) = \{true, false\}$, $x_i = true$ *means player i will go to the party;*

[1] We will make a deeper discussion related to existential variables in Section 4.1.3. Now, we only envisage games without this kind of variables.

- *The goals of all players are following:* $G_1 = \{x_1\}, G_2 = \{x_1 \leftrightarrow x_2\}, G_3 = \{\neg x_1 \wedge x_2 \wedge x_3\}$.

As we can see in the above example, there is no difficulty to encode an arbitrary boolean game by a Constraint Satisfaction Game. In addition, like boolean games, Constraint Satisfaction Games are also able to easily encode games in which there is a full interaction between players.

Next, we define the basic notions of static games (previously described in Definition 2.1) in the literature that link constraint games to general games.

Definition 4.2 (Strategy and Strategy Set)**.** *A* strategy s_i *for player i is an assignment of the variables V_i controlled by player i. Therefore the* strategy set S_i *of player i is D^{V_i}.*

Definition 4.3 (Strategy profile)**.** *A* strategy profile $s = (s_i)_{i \in \mathscr{P}}$ *is the given of a strategy for each player.*

A strategy of a player in CSG is a joint values assigned to his controlled variables while his strategy set will be defined by the Cartesian product over the domains of his variables. A strategy profile is a set of strategies in which each strategy is chosen by one player from his strategy set.

The boolean utility function of player i over a strategy profile s is set to 1 if s satisfies the goal of i and to 0, otherwise.

Definition 4.4 (Utility)**.** *Let $u_i(s)$ be the* utility function *of player i over s, $u_i(s) = 1 \leftrightarrow s \in sol(G_i)$ and $u_i(s) = 0 \leftrightarrow s \notin sol(G_i)$.*

We denote by s_{-i} the projection of s on $V_{-i} = V \setminus V_i$. Beneficial deviation represents the fact that a player will try to maximize his satisfaction by changing the assignment of the variables he can control if he is unsatisfied by the current assignment.

Definition 4.5 (Beneficial deviation)**.** *Given a strategy profile s, a player i has a* beneficial deviation *if $\exists s'_i \in S_i$ such that $u_i(s'_i, s_{-i}) > u_i(s_i, s_{-i})$.*

A tuple s is a *best response* for player i if this player is not able to make any beneficial deviation.

Definition 4.6 (Best response)**.** *A strategy profile $s(s_i, s_{-i})$ is a* best response *(abbreviated BR) for player i if and only if $\forall s'_i, u_i(s_i, s_{-i}) \geq u_i(s'_i, s_{-i})$.*

The main solution concept defined for CSG is pure Nash equilibrium.

Definition 4.7 (Pure Nash Equilibrium)**.** *A strategy profile s is a* Pure Nash Equilibrium (abbreviated PNE) *of the CSG $\mathscr{C}$ if and only if no player has a beneficial deviation, i.e. s is a* best response *of all players.*

Among the PNE in a CSG, it would happen that several PNE are more desirable than the others. Hence, we also propose another solution concept called *Pareto Nash Equilibrium* which is based on PNE .

Definition 4.8 (Pareto Nash Equilibrium)**.** *A PNE s is a* Pareto Nash Equilibrium *(abbreviated PPNE) if and only if there does not exist a PNE s' such that $\forall i \in \mathscr{P}, u_i(s') > u_i(s)$.*

The notions defined above will be illustrated on the following example.

Example 4.2 (CSG)**.** *We consider the following CSG: the set of players is $\mathscr{P} = \{X, Y, Z\}$. Each player owns one variable: $V_X = \{x\}, V_Y = \{y\}$ and $V_Z = \{z\}$ with $D_x = D_y = D_z = \{0, 1, 2\}$. The*

goals are $G_X = \{x \neq y, x > z\}$, $G_Y = \{x \leq y, y > z\}$ *and* $G_Z = \{x+1 = y+z\}$ [2] .

The boolean multimatrix of this example is depicted in Table 4.1.

$z=0$		y		
		0	1	2
x	0	(0,0,0)	(0,1,1)	(0,1,0)
	1	(1,0,0)	(0,1,0)	**(1,1,1)**
	2	(1,0,0)	(1,0,0)	(0,1,0)

$z=1$		y		
		0	1	2
x	0	(0,0,1)	(0,0,0)	**(0,1,0)**
	1	(0,0,0)	(0,0,1)	(0,1,0)
	2	(1,0,0)	(1,0,0)	**(0,1,1)**

$z=2$		y		
		0	1	2
x	0	(0,0,0)	(0,0,0)	*(0,0,0)*
	1	**(0,0,1)**	(0,0,0)	(0,0,0)
	2	*(0,0,0)*	**(0,0,1)**	(0,0,0)

Table 4.1: Example 4.2 is encoded in normal form. The pure Nash equilibrium are depicted in bold and in italic where classical PNE for which no player is satisfied are depicted in italic.

Each solution of G_i is a best response of player $i \in \mathscr{P}$. For example 120 (which stands for $x = 1, y = 2, z = 0$) is a best response of player X because it is a solution of G_X. It is a best response of player Y and player Z as well. So 120 is a pure Nash equilibrium. 100 is also a best response of player X. However, it is not a PNE of the CSG because player Y may deviate from 0 to 1 to get a better strategy 110, a solution of G_Y. Player Z is able to do the same with 121. The strategy profile 221 is a PNE because it is a solution for Y and Z, and player X is unable to deviate because neither 021, 121 or 221 are solution of G_X. Note that an equilibrium can leave one or more player unsatisfied if no assignment is able to satisfy their goal. It may happen like for 022 that no player is satisfied. Among the PNE of this CSG, only the PNE depicted in bold are Pareto Nash equilibrium while the PNE in italic, i.e. $022, 202$, are not because they are dominated by the PNE 120.

We have defined the basic notions of constraint games through Constraint Satisfaction Games. Now, let us discuss about its theoretical aspect, namely the complexity of constraint games. We prove in the following theorem that determining whether a CSG has a PNE is a hard problem.

Theorem 4.1. *Deciding whether a given CSG has at least one PNE is* Σ_2^p*-complete.*

Proof. The proof is adapted from the Σ_2^p-completeness of boolean games [Bonzon et al., 2006].

Membership comes from the simple algorithm in which one guesses a strategy profile and checks that no player has a beneficial deviation. Each verification consists in proving that a player i has no solution if the strategy profile is not winning for i. This verification is in coNP because for a strategy profile s, proving that there exists a solution for player i amounts to solve the CSP G_i, which is in NP. Since the number of players is finite, there is a polynomial number of calls to a coNP oracle (actually one for each player) and thus the problem is in Σ_2^p.

[2] In this chapter, we give many examples illustrating the use of constraint games. In order to make this example and the others easy to understand, the constraints in the players' goals are often equalities or inequalities. However, the constraints in constraint games are not limited to only these kinds of constraints. In contrast, our framework can deal with all constraints defined in constraint programming, including arbitrary constraints, table constraints and soft constraints, as well as arbitrary ways of aggregating the various constraints.

For hardness, we introduce a special case of CSG: the 2-player 0-sum game. In this kind of game, when one player wins, the other player looses. Thus it is enough to represent only the goal of the first player, the other one being deduced by negation. Such a CSG can be represented by $(\mathscr{P} = \{1,2\}, V, D, C)$ where C is the goal of player 1 (the goal of player 2 is straightforwardly deduced by negation).

We perform a reduction from a $\exists\forall$-QCSP[3] to a 2-player 0-sum CSG. $\exists\forall$-QCSP are known to be Σ_2^p-complete. This reduction proves that even 2-player 0-sum CSG are at least as hard than solving a $\exists\forall$-QCSP. Together with membership of the Σ_2^p class, it gives the exact complexity for n-player CSG.

The reduction is from the QCSP $Q = \exists X \forall Y C$ where X and Y are disjoint sets of variables to the 2-player 0-sum CSG $G = (\{1,2\}, X \cup Y \cup \{x,y\}, (D, D_x, D_y), C \vee (x = y))$ where x is a new variable controlled by player 1, y a new variable controlled by player 2 and $D_x = D_y$ are domains composed at least of 2 elements. It is obvious that the conversion can be performed in polynomial time. If Q is valid, then let s_1 be the assignment of variables of X and let s_2 be an assignment of variables of Y. Because Q is valid, $\forall s_2' \in D^Y$, $(s_1, s_2') \in sol(C)$. Thus (s_1, s_2) is a PNE because player 1 is winning and player 2 has no beneficial deviation. Conversely, if Q is not valid then for any assignment $s_1 \in D^X$ of player 1, player 2 can play $s_2' \in D^Y$ such that $(s_1, s_2') \notin sol(C)$. Then if player 1 plays $x = v$ and if $(s_1, s_2) \in sol(C)$, then player 2 can play s_2' and $y = w$ with $w \neq v$. Thus player 2 has a beneficial deviation and (s_1, s_2, v, w) is not an equilibrium. If $(s_1, s_2) \notin sol(C)$ and player 2 plays $y = w$, then player 1 can play $x = w$ and player 1 has a beneficial deviation. Thus (s_1, s_2, w, w) is not an equilibrium. In conclusion, G has a PNE if and only if Q is valid, proving the Σ_2^p-hardness. □

4.1.2 Constraint Satisfaction Games with Hard Constraints

The goals of players in constraint games could be considered as soft constraints or preferences. It may happen however some games have rules that forbid several strategy profiles as they model impossible situations. It is natural to reject such profiles by setting *hard constraints* shared by all players. Hard constraints have been introduced in game theory under the name of *shared constraints* [Rosen, 1965] and are not related to constraint programming but to general constrained optimization. Hard constraints can be easily expressed in the framework of constraint games by adding an additional CSP on the whole set of variables in order to constrain the set of possible strategy profiles.

Definition 4.9 (Constraint Satisfaction Game with Hard Constraints)**.** *A* Constraint Satisfaction Game with Hard Constraints *(abbreviated CSG-HC) is a 5-tuple* $(\mathscr{P}, V, D, C, G)$ *where:*

- $(\mathscr{P}, V, D, G)$ *is a CSG;*
- C *is a CSP on* V*.*

[3] QCSP stands for "Quantified Constraint Satisfaction Problems".

Example 4.3. *We consider the following CSG-HC: the set of players is* $\mathscr{P} = \{X,Y,Z\}$. *Each player owns one variable:* $V_X = \{x\}, V_Y = \{y\}$ *and* $V_Z = \{z\}$ *with* $D_x = D_y = D_z = \{1,2,3\}$. *The goals are* $G_X = \{x = y+z\}$, $G_Y = \{x \geq z > y\}$ *and* $G_Z = \{x = y = z\}$. *The set of hard constraints is composed of the following constraints* $C = \{x \leq y \leq z\}$

It is useful to distinguish a strategy profile which does not satisfy any player's goal from a strategy profile which does not satisfy the hard constraints. The former can be a PNE if no player has a beneficial deviation while the latter cannot. In Example 4.3, the strategy profile (1,2,3) is a PNE although all the players' goals are not satisfied (but no player can deviate to a better strategy) while the strategy profile (3,2,1) is not a PNE because it does not satisfy the hard constraints. Therefore hard constraints provide an increase of modeling expressibility (without however changing the general complexity of CSG).

It is also necessary to specify the notion *utility* in CSG-HC (see Definition 4.4 for CSG). In CSG, it is allowed that all joint strategy of all players are considered as acceptable situations. In CSG-HC, the condition is more restricted. Only strategy profiles, which satisfy hard constraints, are permitted. The others are simply forbidden and therefore have no utility.

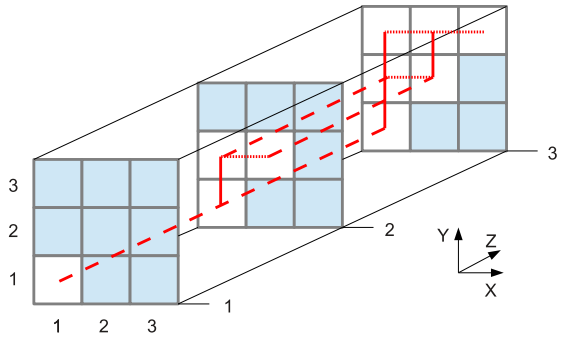


Figure 4.1: Illustration of the satisfiable search space restricted by hard constraints $x \leq y \leq z$ *in Example 4.3.*

From the definition of solution, any strategy profile which is rejected by hard constraints cannot be a solution. Thus the hard constraints exclude the largest common unsatisfiable part of the search space. The intended meaning of the hard constraints is that beneficial deviation is only allowed in the satisfiable subspace defined by the additional CSP. An example is given in Figure 4.1 where the plain and dotted red lines represent the possible deviations for each player. In addition, using hard constraints could greatly improves readability of the model and solving efficiency.

4.1.3 Constraint Optimization Games

Boolean games are a powerful modeling tool for games because it can be exponentially more compact than normal form, even for games with full interaction between players. Nevertheless, there is no way to express the optimization condition inside boolean games without using external tools. In reality, utilities in both classical games and real-world applications are non-boolean and

that requires to incorporate optimization inside frameworks.

It is not difficult to integrate optimization condition into constraint games. A *Constraint Optimization Game* is an extension of CSG in which each player tries to optimize his goal. This is achieved by adding for each player i an optimization condition $\min(x)$ or $\max(x)$ where $x \in V$ is a variable to be optimized by player i.

Definition 4.10 (Constraint Optimization Game)**.** *A* Constraint Optimization Game *(abbreviated COG) is a 5-tuple* $(\mathscr{P}, V, D, G, opt)$ *where :*

- $(\mathscr{P}, V, D, G)$ *is a CSG;*
- $opt = (opt_i)_{i \in \mathscr{P}}$ *is a family of optimization conditions for each player of the form* $\min(x_i)$ *or* $\max(x_i)$ *where* $x_i \in V$.

Let us show how to encode the prisoner's dilemma game as a COG in the following example.

Example 4.4 (Example 2.6 continued)**.** *The prisoner's dilemma can be represented by the values given in Table 4.2. It is also encoded by the following COG:*

- $\mathscr{P} = \{A, B\}$
- $V_A = \{x\}$, $V_B = \{y\}$, $V_E = \{z_A, z_B\}$
- $D(x) = D(y) = \{0, 1\}$
- $G_A = \{z_A = -x + 2y + 1\}, G_B = \{z_B = 2x - y + 1\}$
- $opt_A = min(z_A), opt_B = min(z_B)$

		y	
		0	1
x	0	(1,1)	(3,0)
	1	(0,3)	(2,2)

Table 4.2: The bimatrix of the prisoners' dilemma game in normal form.

The set of existential variable V_E appears in the above example. In constraint games, each existential variable is not under control of any player. In several cases, existential variables can be stated as non decision ones because their values can be computed by side-effect.

For the notions in COG, from CSG, the utility function needs to be slightly adapted. We give below the notion in case of maximization [4].

Definition 4.11 (Utility (COG))**.** *Let* $u_i(s)$ *be the* utility function *of player i over s, then* $u_i(s) = -\infty \leftrightarrow s \notin sol(G_i)$ *and* $u_i(s) = s|_{x_i} \leftrightarrow s \in sol(G_i)$ *where* $opt_i = \max(x_i)$.

Given this, the other notions are the same as for CSG.

[4] For readability, in the remaining sections, for COG, we assume that each player would like to maximize his utility, i.e. the optimization condition is $\max(x)$. It is not difficult to cast into the inverse case, we just need change the sign of variable x.

4.1.4 Constraint Optimization Games with Hard Constraints

Constraint Optimization Games can also be extended with hard constraints the same way Constraint Satisfaction Games are, yielding COG-HC.

Definition 4.12 (Constraint Optimization Game with Hard Constraints)**.** *A* Constraint Optimization Game with Hard Constraints *(abbreviated COG-HC) is a 6-tuple* $(\mathscr{P},V,D,C,G,opt)$ *where C is a CSP on V and* $(\mathscr{P},V,D,G,opt)$ *is a COG.*

We give in the following example a simple Constraint Optimization Game with Hard Constraints.

Example 4.5. *We consider the following COG-HC: the set of players is* $\mathscr{P} = \{X,Y,Z\}$*. Each player owns one variable:* $V_X = \{x\}, V_Y = \{y\}, V_Z = \{z\}$ *and* $V_E = \{a,b,c\}$ *with* $D_x = D_z = \{1,2,3\}, D_y = \{1,\ldots,5\}$*. The goals are* $G_X = \{x = y+z, a = x \times z\}$*,* $G_Y = \{x \geq z > y, b = x+z-y\}$ *and* $G_Z = \{x = y = z, c = x+y-z\}$*. The optimization condition of each player is:* $opt_1 = max(a), opt_2 = max(b), opt_3 = max(c)$*. The hard constraints set is composed of the following constraints* $C = \{x \leq y \leq z\}$*.*

Note that for games with hard constraints, a strategy profile is an equilibrium if it satisfies hard constraints. Like in CSG-HC, a strategy profile is ruled out in COG-HC if it violates at least one hard constraint. Hence, it does not has an utility function. Given this, the other notions are the same as for COG.

4.1.5 Usefulness and Compactness of Constraint Games

We have previously defined constraint games. For ending this section, let us mention here the usefulness and the compactness of our game representation.

Usefulness of Constraint Games

Constraint games are a generic tool which is able to encode any arbitrary static games. This argument is proven through the normal form presentation in Proposition 4.1.

Proposition 4.1. *Given G an arbitrary game which is represented by normal form, then G can be encoded by a constraint game as well.*

Proof. A game in normal form is a structure $\mathscr{N} = (\mathscr{P}$, S, U) where:

- $\mathscr{P}$ is a set of n players;
- $S = \{S_1,\ldots,S_n\}$ is a family of all players' strategy set;
- $U = \{u_1,\ldots,u_n\}$ is a set of all players' utility function where $u_i : \Pi_{j\in\mathscr{P}} S_j \rightarrow \mathbb{R}$

A normal form is equivalent to a Constraint Optimization Game. In order to model a game as a COG, we need to determine a 5-tuple $\mathscr{G} = (\mathscr{P},V,D,G,opt)$. Each element in $\mathscr{G}$ can be specified from $\mathscr{N}$ as follows.

- $\mathscr{P}$ is identical to the set of players in $\mathscr{N}$
- $V = (V_i)_{i \in \mathscr{P}}$ where $V_i = \{x_i, o_i\}$ is controlled by player i
- $\forall i \in \mathscr{P}, D(x_i) = S_i, D(o_i) = \mathbb{R}$
- $\forall i \in \mathscr{P}, G_i = \{c_i\}$ is the goal of player i where $c_i = (W, T)$
  - $W = \{x_i | i \in \mathscr{P}\} \cup \{o_i\}$
  - $T = \{(a_1, \ldots, a_n, u_i(a_1, \ldots, a_n)) | (a_1, \ldots, a_n) \in \Pi_{j \in \mathscr{P}} S_j\}$
- $\forall i \in \mathscr{P}, opt_i = max(o_i)$

Since all the elements in $\mathscr{G}$ can be defined from $\mathscr{N}$, the proposition is proven. □

It has been well-known that normal form is a fully expressive game representation. According to Proposition 4.1, if a game can be modeled by normal form then it is also possible to be encoded by constraint games. Henceforth, our representation is fully expressive as well. We illustrate the transformation process from a normal form representation into a constraint game in the following example.

Example 4.6. *Given a normal form of an arbitrary game that are represented in Table 4.3*

		B			
		b_1	b_2	b_3	b_4
A	a_1	(2,1)	(4,3)	(3,5)	(7,7)
	a_2	(8,2)	(1,4)	(6,8)	(5,9)

Table 4.3: An arbitrary normal form representation

Here is the constraint game model associated to this normal form.

- $\mathscr{P} = \{A, B\}$
- *Each player controls a variable* $V_A = \{x_1, u_1\}, V_B = \{x_2, u_2\}$ *where* V_i *is controlled by player* $i \in \mathscr{P}$.
- $D(x_1) = \{a_1, a_2\}, D(x_2) = \{b_1, b_2, b_3, b_4\}, D(u_1) = D(u_2) = \mathbb{R}$
- *Because the utilities do not have a specific structure, the goal of each player is represented in the following table constraints where the table in left is the goal of player A, and the table in right is the goal for player B.*

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{u_1}$
a_1	b_1	2
a_1	b_2	4
a_1	b_3	3
a_1	b_4	7
a_2	b_1	8
a_2	b_2	1
a_2	b_3	6
a_2	b_4	5

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{u_2}$
a_1	b_1	1
a_1	b_2	3
a_1	b_3	5
a_1	b_4	7
a_2	b_1	2
a_2	b_2	4
a_2	b_3	8
a_2	b_4	9

- *The optimization of the two players are* $opt_A = max(u_1)$ *and* $opt_B = max(u_2)$.

We have shown that every normal form can be represented as a constraint game. It must be similar for graphical games or action-graph games because these representations are based on

matrix storing utilities of all players, like in normal form but using local matrix which are often smaller. Similarly, it is also not difficult to encode boolean games in constraint games. In other words, the boolean games representation is a particular case of constraint games.

Now, given a constraint game, is it possible to encode it by other representations, such as normal form, graphical games, action-graph games or boolean games? The answer is fairly negative. It can be explained by a comparison between constraint games and each other game representation as follows.

- **Constraint Games vs. Normal Form.** Normal form allows any strategy profile to be given an utility. Namely, there does not exist any hard rule that all players must respect. By using hard constraint, constraint games exclude some strategies profiles from being a solution, and this is unrepresentable with normal form, even by giving dummy values for forbidden joint strategies. For example, in case a strategy profile does not satisfy the hard constraints, we give it an utility of $-\infty$ (actually, a negative value large enough). But there are situations where all players obtain this score and normal form may report a spurious equilibrium. We illustrate one of these situations in the following example.

  Example 4.7. *We consider the following CSG-HC: the set of players is* $\mathscr{P} = \{X,Y,Z\}$. *Each player owns one variable:* $V_X = \{x\}, V_Y = \{y\}$ *and* $V_Z = \{z\}$ *with* $D_x = D_z = \{0,1,2\}, D_y = \{0,1\}$. *The goals are* $G_X = \{x \neq y, x > z\}$, $G_Y = \{x \leq y, y > z\}$ *and* $G_Z = \{x+y = z\}$. *The set of hard constraints is C =* $\{$*all_different(x,y,z)*$\}$.

  The normal form associated to this game may be as follows.

$z=0$		y	
		0	1
x	0	$(-\infty,-\infty,-\infty)$	$(-\infty,-\infty,-\infty)$
	1	$(-\infty,-\infty,-\infty)$	$(-\infty,-\infty,-\infty)$
	2	$(-\infty,-\infty,-\infty)$	(1,0,0)

$z=1$		y	
		0	1
x	0	$(-\infty,-\infty,-\infty)$	$(-\infty,-\infty,-\infty)$
	1	$(-\infty,-\infty,-\infty)$	$(-\infty,-\infty,-\infty)$
	2	(1,0,0)	$(-\infty,-\infty,-\infty)$

$z=2$		y	
		0	1
x	0	$(-\infty,-\infty,-\infty)$	(0,0,0)
	1	(0,0,0)	$(-\infty,-\infty,-\infty)$
	2	$(-\infty,-\infty,-\infty)$	$(-\infty,-\infty,-\infty)$

Table 4.4: A spurious normal form corresponding to a game with hard constraints.

  In this game, there are 14 forbidden strategy profiles in which each player's utility is described by the notation $-\infty$. However, if we encode this game by the normal form above, two forbidden strategy profiles, $(0,0,0)$ and $(1,1,1)$, will be reported as PNE because no player can deviate to another strategy to gain a greater utility.

- **Constraint Games vs. Graphical Games and Action-Graph Games.** Modeling games with hard constraints are also out of capacity of graphical games and action-graph games since they are also based on matrix storing utilities like normal form.

- **Constraint Games vs. Boolean Games.** As mentioned earlier, boolean games can encode

only games with boolean utilities. External tools need to be used if we would like to model constraint optimization games. [5]

Compactness of Constraint Games

First, it may be noticed that constraint games are never larger than normal form. But just like boolean games, constraint games can be exponentially more succinct than the utility matrix. An example is when a player has constraints encoded by a CSP $x_i = y_i$ for $2n$ variables (x_i) and (y_i) on a domain with d elements. The representation takes n constraints while there are d^n solutions, which means that a payoff matrix, even in a sparse representation that only keeps positive entries, would be exponential.

In the worst case, however, constraint games, just like CSP can blow-up to a size similar to their set of models, as shown by a simple counting argument. In any formal language, using N bits we can encode 2^N models while there are 2^{2^N} boolean functions of N inputs. As for CSP, our feeling is that this worst case does not happen often in practice. Moreover, we believe that the modeling facilities offered by constraint languages, especially with global constraints, allows to encode many useful problems in an elegant way.

4.2 Applications

Strategic games are an important object of research as many existing games have been studied in the literature for decades. Moreover, the developed models can also be applied to real-life problems. In this thesis, we succeed in testing our modeling framework on classical games taken from Gamut suite as well as on games inspired by problems in real-life. Additionally, the benchmarks in our experiments are constructed by the examples described here.

4.2.1 Games Taken from the Gamut Suite

Gamut [Nudelman et al., 2004] is a suite of game generators designated for testing game-theoretic algorithms. With gamut, instances of games from thirty-five base game classes can be easily generated. We encode here a few classical games taken from the Gamut suite by constraint games.

Guess Two Thirds Average. This game was first proposed by Ledoux in his french magazine [Ledoux, 1981]. Let see how to play the game in Example 4.8.

Example 4.8 (Guess Two Thirds Average (abbreviated GTTA))**.** *In this game, all players guess a number in a range from* 0 *to* 100*. The winners are the players whose number is the closest to*

[5] Elise Bonzon proposed, in her PhD thesis, an extension of boolean games using CP-nets can encode games with non-boolean utilities. The interested readers could find many examples as well as her explanation in [Bonzon, 2007].

two thirds of the average of the numbers guessed by all players. Let m be the number of winners, then all these players with get a score of $100/m$*. All others get a score of zero.*

We need the following existential variables to model this game by constraint games. These variables are in the set V_E.

- $TwoThirdsAvg$ stands for the two thirds average of the numbers guessed by all players.
- $\forall i \in \mathscr{P}, y_i$ stands for the absolute value of the distance between player i's number and the two thirds average.
- z stands for the minimum distance between the numbers and the two thirds average.
- $\forall i \in \mathscr{P}, choice_i$ is a boolean variable for player i. Each variable $choice_i$ is set to 1 if player i's number is the closest to the two thirds average. Otherwise, it is set to zero.
- $\forall i \in \mathscr{P}, p_i$ stands for the payoff of player i.

GTTA can be modeled by a COG as follows:

- $\mathscr{P} = \{1, \ldots, n\}$
- $\forall i \in \mathscr{P}, V_i = \{x_i\}$
- $\forall i \in \mathscr{P}, D(x_i) = \{0, \ldots, 100\}$
- $\forall i \in \mathscr{P}$, G_i contains the following constraints:
  - $TwoThirdsAvg = (2/3) \times ((\sum_{i=1}^{n} x_i)/n)$
  - $\forall i \in \mathscr{P}, y_i = |x_i - TwoThirdsAvg|$
  - $z = \min(y_1, \ldots, y_n)$
  - $\forall i \in \mathscr{P}, choice_i = 1 \Leftrightarrow y_i = z$
  - $choice_i = 1 \rightarrow p_i = 100/(\sum_{i=1}^{n} choice_i)$
  - $choice_i = 0 \rightarrow p_i = 0$
- $\forall i \in \mathscr{P}$, the optimization condition $opt_i = \max(p_i)$

Location Game (Gamut Version). This game is inspired by [Hotelling, 1929]. We call it Location Game (Gamut Version) (Example 4.9) in order to distinguish to another game including hard constraints which will be described in Example 4.12.

Example 4.9 (Location Game (Gamut Version))**.** *Two vendors sell an identical product in their shops settled at a street of length m. The distances from the shop of the vendors' shop from the beginning of the street are* l_1 *and* l_2 *(with* $l_1 < l_2$*), respectively. Each vendor i needs to determine the price* p_i *for their product from his finite set* ρ_i *in order to maximize his benefit* b_i*. There is exactly one client's house at each point from* 1 *to m at the street. Each client will buy the product of the vendor for whom he has to pay the minimal cost of the product price plus transport. We assume that the transport cost unit of each customer* j *is* u_j*. It may exist that the amounts a client needs to pay to both vendors are the same. In this case, the client will privilege vendor labeled by a smaller number (For example, if a client has to choose between vendor* 1 *and vendor* 2*, then he will choose vendor* 1*) .*

We need the following existential variables to model this game by constraint games. These vari-

ables are in the set V_E.

- $\forall i \in \{1,2\}, j \in \{1,\ldots,m\}$ each variable $cost_{ij}$ stands for the cost client j has to pay to vendor i.
- $\forall i \in \{1,2\}, j \in \{1,\ldots,m\}$ each $choice_{ij}$ is a boolean variable which is set to 1 if client j chooses vendor i and to 0 if otherwise.
- $\forall i \in \{1,2\}$ each variable b_i stands for the profit of vendor i.

The game can be encoded by a COG as follows.

- $\mathscr{P} = \{1,2\}$
- $\forall i \in \mathscr{P}, V_i = \{p_i\}$
- $\forall i \in \mathscr{P}, D(p_i) = \rho_i$
- $\forall i \in \mathscr{P}$, G_i contains the following constraints:
  - $\forall i \in \mathscr{P}, \forall j \in \{1,\ldots,m\}, cost_{ij} = |j - l_i| \times u_j + p_i$
  - $\forall j \in \{1,\ldots,m\}, cost_{1j} \geq cost_{2j} \rightarrow choice_{1j} = 1$
  - $\forall j \in \{1,\ldots,m\}, cost_{1j} < cost_{2j} \rightarrow choice_{2j} = 1$
  - $\forall i \in \mathscr{P}, b_i = p_i \times \sum_{j=1}^{m} choice_{ij}$
- $\forall i \in \mathscr{P}$, the optimization condition $opt_i = \max(b_i)$

Minimum Effort Game It is a coordination game which demonstrates the coordination with multiple equilibrium. The equilibrium will be reached if all players choose the same strategy.
Example 4.10 (Minimum Effort Game (abbreviated MEG))**.** *In this game, given an identical strategy set A for each player, his payoff is determined by the formula $a + b \times M - c \times E$ where E is the player's effort and M is the minimum effort of all the players. a, b, c are the parameters of the game.*

We need the following existential variables to model this game by constraint games. These variables are in the set V_E.

- M stands for the minimum effort of all players.
- $\forall i \in \mathscr{P}, p_i$ stands for the payoff of player i.

MEG can be modeled by a COG as follows:

- $\mathscr{P} = \{1,\ldots,n\}$
- $\forall i \in \mathscr{P}, V_i = \{e_i\}$
- $\forall i \in \mathscr{P}, D(e_i) = A$
- $\forall i \in \mathscr{P}$, G_i contains the following constraints:
  - $M = \min(e_1,\ldots,e_n)$
  - $p_i = a + b \times M - c \times e_i$
- $\forall i \in \mathscr{P}$, the optimization condition $opt_i = \max(p_i)$

Travelers' Dilemma. This game can be viewed as an extension of prisoner's dilemma but for more than two players.

Example 4.11 (Travelers' Dilemma (abbreviated TD))**.** *An airline loses n suitcases belonging to n different travelers. All the suitcases are identical and contain the same items. All the travelers are told to claim the value of their suitcase between 2 and 100 (They can not discuss each other). If any travelers write down the lowest value, he will get an extra value* $n*, and the remaining travelers will get an minus value* $n*. All travelers would like to maximize the value they would be reimbursed by the airline.*

Like GTTA, TD has only one PNE when all players take the minimal number as their strategy.

We need the following existential variables to model this game by constraint games. These variables are in the set V_E.

- y stands for the minimal value chosen by all the travelers.
- $\forall i \in \mathscr{P}, choice_i$ is a boolean variable which is set to 1 if the value of player i is minimal, otherwise, it is set to zero.
- $\forall i \in \mathscr{P}, p_i$ stands for the payoff of player i.

TD can be expressed by a COG as follows.

- $\mathscr{P} = \{1, \ldots, n\}$
- $\forall i \in \mathscr{P}, V_i = \{x_i\}$
- $\forall i \in \mathscr{P}, D(x_i) = \{2, \ldots, 100\}$
- $\forall i \in \mathscr{P}$, G_i contains the following constraints:
  - $y = \min(x_1, \ldots, x_n)$
  - $\forall i \in \mathscr{P}, choice_i = 1 \Leftrightarrow x_i = y$
  - $choice_i = 1 \rightarrow p_i = x_i + n$
  - $choice_i = 0 \rightarrow p_i = x_i - n$
- $\forall i \in \mathscr{P}$, the optimization condition $opt_i = \max(p_i)$

Several other classical games from the Gamut suite are also available to be encoded by constraint games, such as: collaboration game, greedy game or NPlayer chicken game, to name a few.

4.2.2 Games Taken from Real Life

Besides the classical games, constraint games are also capable of expressing new, interesting problems taken from various fields such as economics (and Example 4.12), scheduling (Example 4.13), network (Example 4.14 and Example 4.15) and cloud computing (Example 4.16). We show that complex games can be easily and naturally modeled by constraint games.

Economics. Since the earliest days, game theory has mainly used in economics. It is not surprising that games have widely been applied for modeling many problems in this field. We illustrate

here a decision making problem in economics which can be encoded by constraint games. This problem is an extended version of Location Game described in Example 4.9 with n players.

Example 4.12 (Location Game with Hard Constraints)**.** *A group of n ice cream vendors would like to choose a location numbered from* 1 *to m for their stand in a street. Each vendor i wants to find a location l_i. He already has fixed the price of his ice cream to p_i and we assume there is a customer's house at each location. The customers choose their vendor by minimizing the sum of the distance between their house and the vendor plus the price of the ice cream. Because no two vendors can stand at the same places, one need add a global constraints in the hard constraints set of the game.*

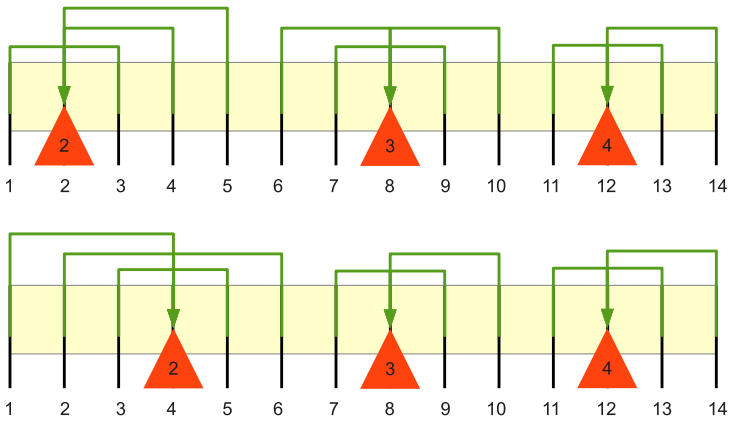


Figure 4.2: Location game with hard constraints.

A possible situation for 3 vendors and 14 customers is depicted in Figure 4.2. In this figure, the arrows depict the behavior of customers. The triangles represent the ice cream vendors with their selling price. In the first situation on top, player 1 can improve benefits by shifting two places to the right, giving situation 2 at bottom. The strategy profile depicted at the top of the figure is not an equilibrium since the left player can deviate and "steal" a customer to the middle player by shifting two positions on the right.

The Location Game with Hard Constraints can be easily modeled by a COG-HC as follows:

- $\mathscr{P} = \{1,\ldots,n\}$
- $\forall i \in \mathscr{P}, V_i = \{l_i\}$
- $\forall i \in \mathscr{P}, D(l_i) = \{1,\ldots,m\}$
- the hard constraints C are the following:
  - no two vendors are located at the same place: $all\_different(l_1, l_2, \ldots, l_n)$
  - $\forall i \in \mathscr{P}, \forall c \in [1..m]$, $\text{cost}_{ic} = |c - l_i| + p_i$
  - $\forall c \in [1..m]$, $\min_c = \min(\text{cost}_{1c}, \ldots, \text{cost}_{nc})$
  - $\forall c \in [1..m], (\min_c = \text{cost}_{ic}) \leftarrow (\text{choice}_{ic} = 1)$
  - $\forall c \in [1..m], \sum_{i \in \mathscr{P}} \text{choice}_{ic} = 1$
- $\forall i \in \mathscr{P}$, G_i contains the following constraint: $\text{benefit}_i = p_i . \sum_{c=1}^{m} \text{choice}_{ic}$
- $\forall i \in \mathscr{P}$, the optimization condition $opt_i = \max(\text{benefit}_i)$

An interesting feature of Example 4.12 is that it uses global constraints like *all_different* the same way as in constraint programming. It also shows the interest of modeling hard constraints in games since it is perfectly natural to think that no two vendors can settle at the same place. It is possible to transform this problem into a CSG by fixing a minimal profit mp_i for each player i and stating that player i is satisfied if his benefits is over mp_i. It can be done by adding the constraint $\text{benefit}_i \geq mp_i$ to G_i instead of the optimization condition. In the Gamut version of the game (Example 4.9), vendors do not choose location but prices, because there is no way to express that vendors should choose different locations in a normal form game like we do here with the *all_different* constraint.

Scheduling. Scheduling is a process of creating a schedule. It decides how to order tasks and how to commit resources between the variety of possible tasks. Scheduling is a crucial application of constraint programming, and also an important issue in computing, such as making real-time schedule on multiple processors, etc. Example 4.13 shows an application of strategic scheduling, which often occurs in real-life when multiple players share the same resource.

Example 4.13 (Strategic Scheduling)**.** *Consider a set $\mathscr{P} = \{1,\ldots,n\}$ of $n > 1$ players. Each player i owns a task of cost d_i units to perform on any machine taken among m machines. All machines are running in parallel. Each machine j has a maximal capacity of c_j units. In case a machine is overloaded, it simply stops and does not perform the task of the players who chose it. Thus the goal of each player is to choose carefully a non-overloaded machine, taking into account the decisions of the other players.*

The strategic scheduling problem can be expressed as a Constraint Satisfaction Game as follows:

- $\mathscr{P} = \{1,\ldots,n\}$
- $V_i = \{m_i\}$, m_i is the machine chosen by player i.
- $\forall i \in \mathscr{P}, D(m_i) = \{1,\ldots,m\}$
- C is composed of channeling constraints for boolean variables $choice_{ij}$ stating that player i chooses machine j: $(m_i = j) \leftrightarrow (choice_{ij} = 1)$
- $\forall i \in \mathscr{P}, G_i$ is composed of the capacity constraint:

$$\sum_{j=1}^{m} (choice_{ij} \times \sum_{k \in \mathscr{P}} (choice_{kj} \times d_k)) \leq c_j$$

  This constraint is true if the machine chosen by player i is not overloaded and false otherwise.

Interestingly, strategic scheduling is not equivalent to classical scheduling. In a classical CSP, it would be required that there is an assignment which satisfies all players. This is the case in the instance Sch_1 with 3 players having a task of respective cost $2,3,1$ and two machines of capacity 3. This instance is depicted in the left part of Figure 4.3 and the two Nash equilibrium correspond to the solution of the CSP composed of the union of the player's goals. With a slight change, Sch_2 is the same problem but the cost of any task is 2. Then there is no global assignment

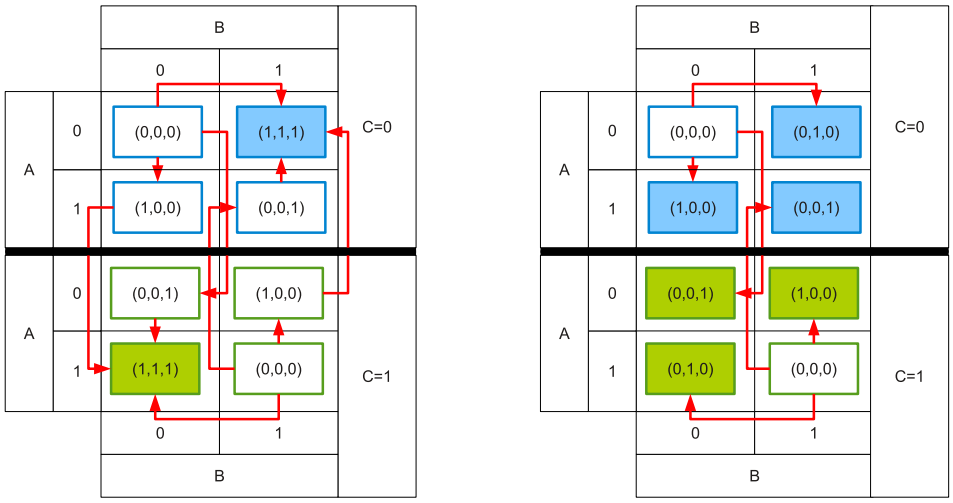


Figure 4.3: Two examples of strategic scheduling (Sch_1 on the left and Sch_2 on the right). Nash equilibrium have a colored background and the arrows depict beneficial deviations.

which allows to place 2 tasks on the same machine and there is no situation where all players are simultaneously satisfied. This situation is depicted in Figure 4.3, right-hand side. However, the strategic formulation allows to find solutions in which one player can run her task while the others are canceled, yielding 6 Nash equilibrium. More complex versions of strategic scheduling can be found in [Vöcking, 2007].

Network. Computer network or internet actually plays an indispensable role in modern life. In reality, network aims at serving many people in parallel. That may thus cause a conflict between users' objective. It also means that using game theory to model problems in network is a very interesting idea. We give here two problems which can be naturally encoded by constraint games.

The first problem (Example 4.14) is inspired by [Bouhtou et al., 2007] and taken from telecommunication industry.

Example 4.14 (Network Game)**.** *A network provider owns m links to transfer data. Each link j is specified by 3 parameters: capacity c_j, speed per data unit s_j and price per data unit p_j. A group of n clients would like to transfer data across theses links (client i from a source x_i to a target y_i, each source and target are fully connected to each link of the vendor and each path has to cross a tolled arc). In order to reach a link j of the network, each client i has to suffer a fixed fee α_{ij} and a fixed delay β_{ij}.*

Hence, with any link j customer i chooses, he has to pay an addition cost α_{ij} per data unit and it also takes an additional time β_{ij} per data unit to transit on the tolled arc. Each customer could always choose another provider with the time ψ_i, so if the provider's data speed offered is competitive, he therefore wishes to minimize the cost for transferring his data.

This problem can be modeled by a COG-HC as follows:

- $\mathscr{P} = \{1, \ldots, n\}$

- $\forall i \in \mathscr{P}, V_i = \{r_i\}$
- $\forall i \in \mathscr{P}, D(r_i) = \{1, \ldots, m\}$
- C is composed of the following constraints:
  - channeling constraints for boolean variables stating that link j is requested by data d_i of player i: $(r_i = j) \leftrightarrow (choice_{ij} = 1)$
  - capacity constraints: $\forall j \in \{1, \ldots, m\}, \sum_{i=1}^{n} choice_{ij} \times d_i \leq c_j$
- $\forall i \in \mathscr{P}, G_i$ is composed of the following constraints:
  - $cost_i = \sum_{j=1}^{m} choice_{ij} \times d_i \times (p_j + \alpha_{ij})$
  - $time_i = \sum_{j=1}^{m} choice_{ij} \times d_i \times (s_j + \beta_{ij})$
  - $time_i \leq \psi_i$
- $\forall i \in \mathscr{P}, opt_i = min(cost_i)$

While in the first problem, all players would like to minimize the cost they have to pay. In the second problem described in Example 4.15, they want to cross their data through network in the fastest way.

Example 4.15 (Hypercube Network Congestion Games)**.** *Congestion Games [Rosenthal, 1973] are a well-known class of games in game theory. In a congestion game, there are a set of players and a set of resources. Each player can choose a subset of resources and his payoff depends on the resources he chooses and the number of players choosing the same ones (see Section 2.1.5).*

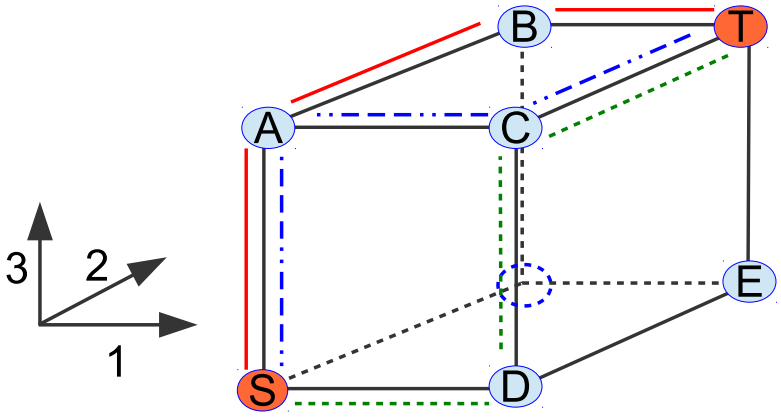


Figure 4.4: An example of m-dimensional hypercube network congestion game with m = 3 and 3 players

Figure 4.4 depicts a network congestion game which will be modeled by constraint games. In this game, network is created as a m-dimensional hypercube. To reach T from S, each player must choose m paths with different directions. Henceforth, each player owns m variables with the domain from 1 *to m and all values instantiated of his variables must be different. In the figure, the bold lines depict the paths chosen by player* 1*, the dash lines for player* 2 *and the dotted lines for player* 3*. The strategies of player* 1*, player* 2 *and player* 3 *are (3,2,1), (3,1,2) and (1,3,2), respectively. The delay function on each path is determined by the number of players selecting the path. Let s and s' be the paths chosen by two different players. Then s_j and s'_j share a same path if and only if $s_j = s'_j$ and $\forall k \in [1, j-1], s_k \in s'$ and $s'_k \in s$. In Figure 4.4, the paths SA and*

CT are shared by the players while the others are not. Finally, the players' payoff are 4 (2 + 1 +1 for the paths SA - AB - BT), 5 (2 + 1 + 2 for the paths SA - AC - CT) and 4 (1 + 1 + 2 for the paths SD - DC - CT).

This problem can be modeled by the following COG-HC:

- $\mathscr{P} = \{1,\ldots,n\}$
- $\forall i \in \mathscr{P}, V_i = \{v_{i1},...,v_{im}\}$
- $\forall i \in \mathscr{P}, \forall j \in [1,m], D(v_{ij}) = \{1,\ldots,m\}$
- C is composed of the following constraints:
  - each player chooses m paths with different directions:
    $\forall i \in \mathscr{P}$, *all_different*$(v_{i1},...,v_{im})$
  - channeling constraints for boolean variables d_{ijk} for delay caused by each variable:
    $\forall i,i' \in \mathscr{P}, i \neq i', \forall j \in [1,m], \forall k \in [1,m-1]; (d_{ijk} = 1) \leftrightarrow$
    cardinality_atleast$(1,(v_{i1},\ldots,v_{i(j-1)}), (v_{i'1},\ldots,v_{i'(j-1)})) \wedge (v_{ij} = v_{i'j})$
- $\forall i \in \mathscr{P}, G_i$ is composed of the following constraint:
  $\text{delay}_i = \sum_{j=1}^{m}(\sum_{k=1}^{m-1} d_{ijk} + 1)$
- $\forall i \in \mathscr{P}$, the optimization condition $opt_i = \min(\text{delay}_i)$

Cloud computing. Cloud computing is an emergent field which refers to the on-demand delivery of IT resources and applications via the Internet with pay-as-you-go pricing. Resource allocation is a central issue in cloud computing where clients use and pay computing resources on demand. In order to manage conflicting interests between clients, [Jalaparti et al., 2010] has proposed the framework of *CRAG* (Cloud Resource Allocation Game) in which resource assignments are defined by game equilibrium (Example 4.16)

Example 4.16 (Cloud Resource Allocation Game)**.** *A cloud computing provider owns a set $\mathscr{M} = \{M_1,\ldots,M_m\}$ of m machines, each machine M_j having a capacity c_j representing the amount of resource available (for example CPU-hour, memory) (see Figure 4.5 on the next page). The cost of using machine j is given by $l_j(x) = x \times u_j$ where x is the number of resources requested and u_j some unit cost. A set of n clients $\mathscr{P} = \{1,2,\ldots,n\}$ wants to use simultaneously the cloud in order to perform tasks. Client $i \in \mathscr{P}$ has m_i tasks $\{T_{i1},\ldots,T_{im_i}\}$ to perform, with respective requested capacity of $\{d_{i1},\ldots,d_{im_i}\}$. Each client $i \in \mathscr{P}$ chooses selfishly an allocation r_{ik} for the task T_{ik} ($k \in [1..m_i]$) and wishes to minimize his cost $cost_i = \sum_{k=1}^{m_i} l_{r_{ik}}(d_{ik})$. We assume that the provider's resources amount is sufficient to accommodate the resources requested by all of the clients: $\sum_{i=1}^{n}\sum_{k=1}^{m_i} d_{ik} \leq \sum_{j=1}^{m} c_j$.*

This problem can be modeled by the following COG-HC:

- $\mathscr{P} = \{1,\ldots,n\}$
- $\forall i \in \mathscr{P}, V_i = \{r_{i1},\ldots,r_{im_i}\}$
- $\forall i \in \mathscr{P}, \forall k \in [1..m_i], D(r_{ik}) = \{1,\ldots,m\}$
- C is composed of the following constraints:

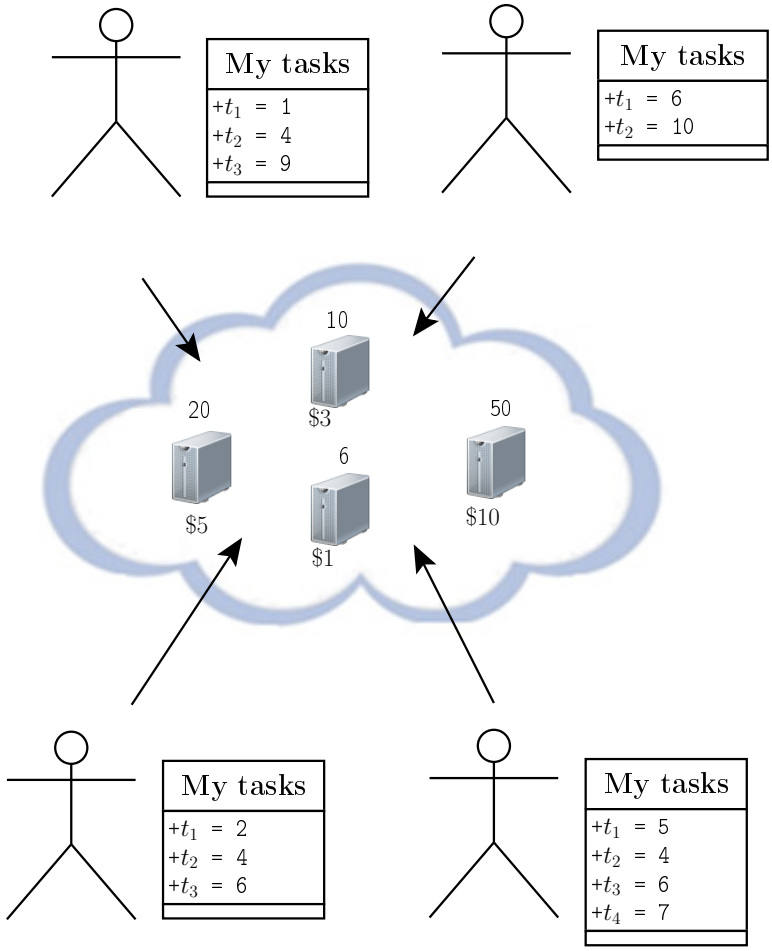


Figure 4.5: An example of a Cloud Resource Allocation Game with 4 clients and 4 machines.

  - channeling constraints for boolean variables stating that machine j is requested by task t_{ik}: $(r_{ik} = j) \leftrightarrow (choice_{ijk} = 1)$
  - capacity constraints: $\forall j \in [1..m], \sum_{i=1}^{n} \sum_{k=1}^{m_i} choice_{ijk} \times d_{ik} \leq c_j$
- $\forall i \in \mathscr{P}, G_i$ is composed of the following constraint: $cost_i = \sum_{j=1}^{m} \sum_{k=1}^{m_i} choice_{ijk} \times l_j(d_{ik})$
- $\forall i \in \mathscr{P}, opt_i = \min(cost_i)$

Along with the problems described here, we are confident that constraint games can express many others as well. That suggests a real utility of this new modeling framework for a potential wide range of problems in practice.

4.3 Solving Framework

As mentioned in Section 2.4, there are still not many solving tools for generic games. This motivates us to concentrate our interest on designing a new solving framework for games. In this section, we give a bird's-eye view about ConGa, our solver for solving constraint games. Additionally, we also specify the environment for all the experiments we have performed.

4.3.1 ConGa Solver

We have built a solver called *ConGa* on top of the constraint library Choco [Choco Team, 2013]. This solver allows to express constraint games and solves them using multiple tools. These tools are composed of local search algorithms (Chapter 5), complete algorithms (Chapter 6) and heuristic algorithms (Chapter 7). We will give full information about these algorithms in the following chapters. Another important point is that our solver accepts all the constraints provided by Choco, and reuses the existing constraint propagators in the library.

The *ConGa* solver has been built up on Choco-2.1.5 released in 2013. Choco [Choco Team, 2010] is a java library for constraint programming. It is a complete solver for solving constraint satisfaction problems. Generally, given a constraint satisfaction problem, for solving it in Choco, we just need to define a set of variables along with their domains. We then define a set of constraints for the problem. They are the essential elements. Choco also provides multiple heuristics for variables and values ordering as well as the extension to constraint optimization problems. After we have declared the model, Choco takes care of the remaining parts and returns the solutions according to our desire, for example, only the first solution or all the solutions.

Choco basically solves a CSP by backtracking search (Section 3.2.1). It defines two types of *environments* in backtracking system: `EnvironmentTrailing` and `EnvironmentCopying`. Then it defines the notion of *world*. A world contains values of storable objects or operations that permit to backtrack to its state. The environment therefore pushes and pops worlds when storing or restoring previous states of the solver.

Reusing solver is a very crucial task in ConGa because we need to often restore previous states of the solver in the algorithms. Hence we have applied `EnvironmentTrailing` which is more efficient than `EnvironmentCopying` in our solver. Every operation applied to a data type is pushed in a trailer (`worldPush`). When a world is pushed, the index of the last operation is stored. When a world is popped (`worldPop`), these operations are popped and unapplied until reaching the last operation of the previous world. This technique will be frequently used in the algorithms of the ConGa solver in the next chapters.

On the other side, checking players' deviation is an essential task for finding pure Nash equilibrium since it is repeatedly executed during search. Unfortunately, reusing solver in Choco for this task is very costly even though the current solver state requires only little modification. That is why we have implemented a custom solver for only verifying whether a player can deviate from a current strategy profile. In the custom solver, for player i, it is composed of hard constraints (if existed) and the player's goal. The difference here is that all variables controlled by the remaining players, except i, have been assigned before the propagation are performed. The custom solver henceforth becomes smaller and simpler for solving in comparison to the general Choco solver. It hence offers a better performance than Choco.

4.3.2 Experimental Environment

All the experiments presented in this thesis have been executed on a server of 48-core AMD Opteron 6174 of 4-processor at 2.2 GHz with 256 GB of RAM. The operating system installed is ubuntu 64bit 12.04 LTS. All the benchmarks in our experiments are constructed by the examples described in Section 4.2.

4.4 Conclusion

In this chapter, we have presented *Constraint Games*, the first framework allows to model and solve static games by using constraint programming. Constraint games come in two issues: Constraint Satisfaction Games and Constraint Optimization Games, with or without hard constraints. The most prominent advantage of constraint games is that they provide a new compact yet natural encoding to games. The main solution concept defined for constraint games is the one of Nash equilibrium. They are the situations in which no player has an incentive to deviate from the current ones. We have also demonstrated the usefulness of constraint games by using this framework for modeling a few classical games as well as games inspired by applications in real-world problems. Finally, we have mentioned several basic elements of our solver ConGa, including the solving tools which will be precised in the following chapters.

Chapter 5

Local Search for Constraint Games

Contents

5.1 CG-IBR: Iterated Best Responses Algorithm in Constraint Games 72

5.2 CG-TS: Tabu Search Algorithm in Constraint Games 76

5.3 CG-SA: Simulated Annealing Algorithm in Constraint Games 78

5.4 Experiment 84

5.4.1 Description of Experimental Game Instances 84

5.4.2 Experimental Results 85

5.5 Related Work 87

5.6 Conclusion 88

Local search has been recognized as an efficient method for solving large and complex combinatorial problems. Because of the huge size of several problems, it even emerges as the unique feasible way for reasoning. Local search has been widely used to find a solution in constraint programming (see Section 3.2.2). In contrast, few local search researches have been devised to compute a pure Nash equilibrium or a mixed Nash equilibrium in game theory (see Section 2.2). The main difficulty, in our opinion, comes from the intractability of the classical game representation, namely normal form.

We present here three local search algorithms, including two metaheuristics to find a pure Nash equilibrium in constraint games. The simplest algorithm is *CG-IBR*, iterated best responses on the constraint games representation. Then we propose *CG-TS*, a tabu search based on the CG-IBR algorithm. The last algorithm called *CG-SA* is a metaheuristic search inspired by the simulated annealing algorithm. All these algorithms have been implemented in the solver ConGa. The experimental results demonstrate that our local search solver is able to solve game instances whose size is beyond the size accessible to algorithms based on normal form [1].

[1] A part of work in this chapter has been published in [Nguyen et al., 2013].

5.1 CG-IBR: Iterated Best Responses Algorithm in Constraint Games

Iterated best responses (abbreviated IBR) [Shoham and Leyton-Brown, 2009] is the simplest local search algorithm to find a PNE in any game representation. In constraint games, we have also implemented an algorithm of IBR, what we call *CG-IBR*. In spite of its naivety, it could be considered as the baseline algorithm to evaluate the next ones. CG-IBR (Algorithm 5.1) is an iterative process starting at an arbitrary pure strategy profile (line 2). If there exists some hard constraints in games, the initial strategy profile must satisfy them.

At each step, if there exists a player for whom the current strategy profile is not a best response, then this player deviates to his best response which will be considered as the candidate in the next step (line 7). The process stops when all players are no longer able to change their strategy. The current candidate is thus a best response of all players, i.e. a PNE is found (line 9). Otherwise, the algorithm fails to find one PNE within the max step (line 13).

Algorithm 5.1. CG-IBR, Iterated Best Response Algorithm for Constraint Games

```
1:  function CG-IBR(Constraint Game CG): tuple
2:      s ← get a random solution of the solver (with hard constraints if they exist)
3:      step ← 1
4:      while step ≤ max_step do
5:          s' ← neighborIBR(s,𝒫)              ▷ find a neighbor of the current solution to move
6:          if s' ≠ null then
7:              s ← s'
8:          else                                ▷ s is best response of all players
9:              return s
10:         end if
11:         step++
12:     end while
13:     return null
14: end function
```

Algorithm 5.2. Find a neighbor in the CG-IBR algorithm

```
1:  function NEIGHBORIBR(tuple s, set of players 𝒫): tuple
2:      while 𝒫 ≠ ∅ do
3:          pick player i from 𝒫 randomly
4:          s' ← findBR_C*G(s,i)               ▷ find a random best response of player i over s
5:          if s' ≠ null then
6:              return s'
7:          end if
8:          remove i from 𝒫
9:      end while
10:     return null
11: end function
```

Algorithm 5.2 shows how a neighbor is chosen in CG-IBR. Given a set of players $\mathscr{P}$, a player is randomly picked to verify whether he can deviate (line 3). If the player can make a beneficial deviation from s to a best response s' then s' will be considered as the next candidate (line 4-7). In

line 4, the notation "*" is replaced by "S" for Constraint Satisfaction Games (Algorithm 5.3) and by "O" by Constraint Optimization Games (Algorithm 5.4). Otherwise, no neighbor is returned (line 10).

Example 5.1. *We consider the following CSG-HC: the set of players is* $\mathscr{P} = \{X,Y,Z\}$. *Each player owns one variable:* $V_X = \{x\}, V_Y = \{y\}$ *and* $V_Z = \{z\}$ *with* $D_x = D_y = D_z = \{1,2,3\}$. *The goals are* $G_X = \{y \leq z, x \geq z\}$, $G_Y = \{x \leq y, y \geq z\}$ *and* $G_Z = \{x+y=z\}$. *The hard constraint is* $\{x \neq y+z\}$.

The following schema illustrates how CG-IBR runs on Example 5.1.

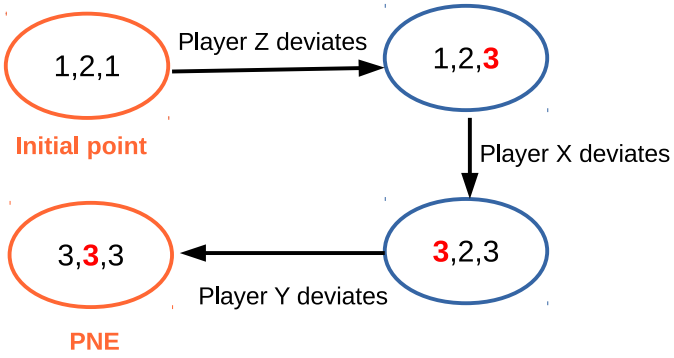


Figure 5.1: CG-IBR runs on an simple example.

At the beginning, an initial point is randomly chosen. However it must satisfy the hard constraint. An example of forbidden strategy profile could be $(3,2,1)$ since this tuple violates the hard constraint. During search, we let the players deviate until a pure Nash equilibrium is detected or other termination conditions are met. In Figure 5.1, we start at the tuple $(1,2,1)$. Only player Z can deviate from $(1,2,1)$, we thus let Z deviate to his best response $(1,2,3)$ (which is a solution of his goal). Now, both the two players, X and Y, can deviate. We randomly pick player X at first, so we let him deviate to $(3,2,3)$. Finally, player Y deviates from $(3,2,3)$ to his best response $(3,3,3)$ which is a best response of the two remaining players as well. We thus find a PNE of the example and the algorithm terminates. In some other constraint games, this process may stop within the max step without finding a PNE.

We distinguish the process of finding a best response for a tuple s of player i for CSG and COG in two separated algorithms (Algorithm 5.3 and Algorithm 5.4). In Algorithm 5.3, a solver including the goal G_i and the hard constraints (if they exist) for games is created (line 2). There may be more than one best response. In a CSG, it means that several assignments are solution of the player's solver. If player i could deviate to a best response, it should be a random one in his set of best responses. We thus set a random heuristic for variable and value ordering in line 3. In Example 5.1, a random variable ordering could be $z \rightarrow x \rightarrow y$ while a random value ordering for player Y could be $2 \rightarrow 1 \rightarrow 3$.

Line 4 - 10 are devoted to instantiate the values taken from s to the variables controlled by the remaining players. This current state of the solver needs to be saved (line 11) in order to be restored later (line 18). Then, we assign the values for player i's variables (line 12 - 14). If the

Algorithm 5.3. Find a random *best response* for CSG in the CG-IBR algorithm

```
1: function FINDBR_CSG(tuple s, int i): tuple
2:     initialize solver S_i with G_i (and hard constraints if they exists)
3:     set a random heuristic for variable and value ordering in S_i
4:     for all j ∈ 𝒫 do          ▷ instantiate the values from s to the variables controlled by other players
5:         if i ≠ j then
6:             for all (k = 1; k ≤ |V_j|; k++) do
7:                 add constraints V_jk = s_jk
8:             end for
9:         end if
10:    end for
11:    save the current state of the solver
12:    for all (k = 1; k ≤ |V_i|; k++) do          ▷ instantiate the values to the variables of player i
13:        add constraints V_ik = s_ik
14:    end for
15:    if S_i.isFeasible() then                    ▷ s is solution of player i
16:        return null
17:    end if
18:    restore the previous state of the solver
19:    sol ← S_i.getSolution()                     ▷ find the first solution of the solver
20:    if sol = null then                          ▷ S_i has no solution,
21:        return null
22:    else
23:        return sol                              ▷ player i can deviate from s to sol
24:    end if
25: end function
```

tuple s is a solution of the solver than it is also his best response (line 15 -17). Otherwise, we try to find another solution for which the goal is satisfied (line 19). If no solution is detected (line 20), the goal G_i is not satisfied by any tuple $s = (s_i, s_{-i}), \forall s_i \in S_i$. Thus, the player cannot deviate to another strategy (line 21). Otherwise, the first solution found by the solver will be returned (line 23).

Three following cases can appear when we search a best response from a tuple s for a player.

(i) The tuple s is already the player's best response.

(ii) There is no solution for the player's goal with every tuple including s_i, so all strategies $s = (s_i, s_{-i})$ are the player's best responses.

(iii) The tuple s is not a best response, and the player can deviate to one of his best responses.

In Example 5.1, the tuple $(1,2,3)$ is a best response of player Z because $(1,2,3)$ is a solution of player Z's goal (line 16 in Algorithm 5.3). While $(3,3,3)$ is a best response of player Z even the player's goal is not satisfied because player Z does not either change to another strategy in order to be satisfied (line 21). Finally, the tuple $(1,2,1)$ is not a best response of player Z because it is not a solution of the player's goal. Moreover, Z is available to deviate to the tuple $(1,2,3)$ for improving his satisfaction (line 23).

Algorithm 5.4. Find a random *best response* for COG in the CG-IBR algorithm

```
1:  function FINDBR_COG(tuple s, int i): tuple
2:      initialize solver S_i with G_i and opt_i (and hard constraints (if exists))
3:      set a random heuristic for variable and value ordering in S_i
4:      for all j ∈ 𝒫 do          ▷ instantiate the values from s to all variables controlled by other players
5:          if i ≠ j then
6:              for all (k = 1; k ≤ |V_j|; k++) do
7:                  add constraints V_jk = s_jk
8:              end for
9:          end if
10:     end for
11:     save the current state of the solver
12:     for all (k = 1; k ≤ |V_i|; k++) do          ▷ instantiate the values from s to the variables of player i
13:         add constraints V_ik = s_ik
14:     end for
15:     if S_i.isFeasible() then                                          ▷ s is a solution of S_i
16:         get payoff x from s
17:     else
18:         payoff x from s is set to −∞                    ▷ −∞ is a large enough negative number
19:     end if
20:     restore the previous state of the solver
21:     sol ← S_i.getOptSolution()                          ▷ find the first random optimal solution
22:     if sol = null then                                                ▷ S_i has no solution,
23:         return null          ▷ goal G_i is not satisfied by any tuple s = (s_i, s_−i) with s_i ∈ S_i
24:     else
25:         get the optimal payoff xOpt
26:         if xOpt > x then                                              ▷ s is not best response
27:             return sol                                  ▷ Player i can deviate from s to sol
28:         else                                                          ▷ s is an optimal solution of S_i
29:             return null
30:         end if
31:     end if
32: end function
```

The same process for COG depicted in Algorithm 5.4 is fairly similar but more complex. The difference is that we need to find optimal solutions of Constraint Optimization Problem. Unlike in CSG, being a solution of G_i does not guarantee to be player i's best response. That is why we need to compute the payoff of the current tuple (line 15 - 19). We recall that in all Constraint Optimization Games, we assume that each player would like to maximize his objective variable. If s is not a solution, we set his payoff to be a very large negative number (line 18).

Line 21 is devoted to find a solution which optimizes the objective variable x in the optimization condition opt_i of the player. Like in CSG, no solution found implies that goal G_i is never satisfied from s (line 22 - 24). Otherwise, we get the optimal payoff in line 25. We note that the solver can reach to the optimal value from different points but this value is unique. So if the current payoff is less than the optimal one, s is not a best response than the optimal solution is returned (line 26 -27). Otherwise, no tuple is returned (line 29).

We have specified how the algorithm CG-IBR works for finding one PNE in constraint games. We also prove its correctness in the following proposition.

Proposition 5.1. *The CG-IBR algorithm in constraint games is correct.*

Proof. A PNE is reported by CG-IBR if and only if it satisfies the hard constraints and no player is able to deviate from it. Hence, the CG-IBR algorithm is correct. □

Besides the *iterated best responses* algorithm for constraint games, another algorithm called *iterated better responses* could be naturally derived. Given a strategy profile $s = (s_i, s_{-i})$, a strategy profile $s' = (s'_i, s_{-i})$ is said to be a *better response* from s for player i if $u_i(s) < u_i(s')$. Note that s' is not mandatory to be a *best response* of player i because it may exist another strategy profile $s'' = (s''_i, s_{-i})$ such that $u_i(s') < u_i(s'')$. For Constraint Satisfaction Games, the *iterated best responses* algorithm and the *iterated better responses* algorithm are exactly identical. In Constraint Optimization Games, there is only a little difference in Algorithm 5.4. It is no longer necessary to compute the optimal payoff in line 25. We only need to detect a greater payoff than the current one instead.

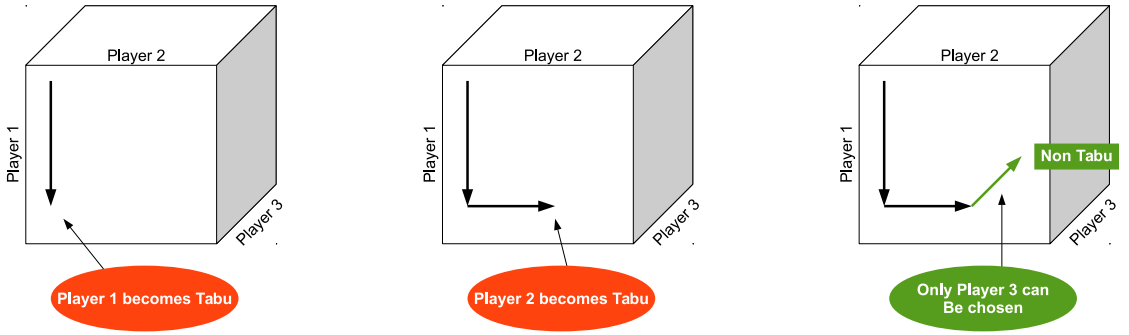
In these naive algorithms, a random best response of a player who can deviate is chosen without any condition. Actually, we would like to improve the performance of our local search solver by studying several metaheuristics which select a next candidate according to some rules. We therefore describe a couple of new algorithms in the following sections.

5.2 CG-TS: Tabu Search Algorithm in Constraint Games

Tabu search (abbreviated TS) is a metaheuristic proposed by Fred W. Glover [Glover, 1989]. By using memory structures that describe the visited solutions, it allows to avoid the trap of local optima. Since if a potential solution has been previously visited within a certain short-term period, it is marked as "forbidden" so that the algorithm does not consider that possibility repeatedly.

In our local search solver, we have implemented an algorithm called *CG-TS* which is inspired by the tabu search for avoiding getting stuck in infinite looping. In constraint games, a strategy profile is not a PNE if a player can deviate to his best response. A neighbor in CG-IBR and CG-TS is thus defined by a best response of this player. Henceforth, there does not exist "real" local optima in these two algorithms, but infinite loops. It means we can always move to a neighbor if the current candidate is not a PNE, but we move in infinite looping instead.

In CG-TS, the visited strategy profiles are not saved in the tabu list but the players who have recently moved. The tabu list is used to forbid a player to be chosen too early after his recent movement (See Figure 5.2). In contrast with classical search space where a local search algorithm wants to escape local optima, it may happen that the trajectory gets stuck in cycles. Thus a tabu list of size L allows to avoid direct cycles of length L and in practice allows also larger cycles to be escaped.



(a) After having deviated, player 1 is recorded in the tabu list.

(b) After having deviated, player 2 is recorded in the tabu list.

(c) Player 1 and 2 are recorded in the tabu list. Only player 3 can deviate.

Figure 5.2: Illustration of using the tabu list in a game of 3 players

CG-TS is depicted in Algorithm 5.5. In this algorithm, we keep two global variables that describe the *tabu list* and the *tabu length*. The main difference with CG-IBR is that the process of choosing a neighbor is divided into two phases. First, we verify whether a non-tabu player could make a deviation (line 8). If no one can, tabu players are checked against deviation (line 10).

Algorithm 5.5. CG-TS, Tabu Search Algorithm for Constraint Games

```
1:  global variables
2:  Tabu list : a queue 𝓜
3:  Tabu length: L
4:  function CG-TS(Constraint Game CG): tuple
5:      s ← get an arbitrary solution of solver (with hard constraints if they exist)
6:      step ← 1
7:      while step ≤ max_step do
8:          s' ← neighborTabu(s, 𝒫 \ 𝓜)          ▷ check deviation of non-tabu players
9:          if s' = null then
10:             s' ← neighborTabu(s, 𝓜)          ▷ check deviation of tabu players
11:             if s' = null then return s       ▷ no player could deviate
12:             end if
13:         end if
14:         s ← s'
15:         step++
16:     end while
17:     return null
18: end function
```

Finding a neighbor in CG-TS is specified in Algorithm 5.6. This algorithm is similar to Algorithm 5.2. The difference is that we need to perform some additional operations for dealing with the tabu list.

The tabu list $\mathscr{M}$ is implemented by a queue. When a player deviates, he is pushed in $\mathscr{M}$ (line 6). As soon as the size of the tabu list is greater than the tabu length L, the first element in the queue will be popped (line 7-9).

Algorithm 5.6. Find a neighbor in the CG-TS algorithm

```
1: function NEIGHBORTABU(tuple s, list of players 𝒫′): tuple
2:     while 𝒫′ ≠ ∅ do
3:         pick a player i from 𝒫′ randomly
4:         s′ ← findBR_C*G(s,i)
5:         if s′ ≠ null then
6:             𝒜.push(i)                          ▷ add player i into the tabu list
7:             if |𝒜| > L then
8:                 𝒜.pop()                        ▷ remove the first element from the queue
9:             end if
10:            return s′
11:        end if
12:        remove i from 𝒫′
13:    end while
14:    return null
15: end function
```

Like the CG-IBR algorithm, the CG-TS algorithm is correct as well (see Proposition 5.2).

Proposition 5.2. *The CG-TS algorithm in constraint games is correct.*

Proof. A reported equilibrium is correct because it has been successively checked against deviation for all tabu and non-tabu players and it satisfies the hard constraints. □

5.3 CG-SA: Simulated Annealing Algorithm in Constraint Games

Simulated annealing (abbreviated SA) [Kirkpatrick et al., 1983] is a generic probability metaheuristic which was originally inspired from the process of annealing in metal work. Annealing involves heating and cooling a material to alter its physical properties due to the changes in its internal structure. As the metal cools, its new structure becomes more fixed. Due to the annealing technique with slow cooling, we will obtain a crystal solid state with the global minimum of energy.

The intuition of the simulated annealing algorithm is composed of two key ideas. First, given a situation s and a set of its neighbors, we use a function for evaluating the neighbors compared with s. We move from the current candidate s to a neighbor if it is proven to be better than s. Nevertheless, the algorithm also moves to a worse neighbor with an acceptance probability. This gives the algorithm the possibility to jump out if it gets stuck in a local optimum.

Figure 5.3 illustrates how to apply SA in a simple problem. We would like to climb to the highest point in this schema, therefore a point s' is said to be better a point s if s' is higher than s. We assume that the algorithm starts at the point A. Because the point B is better than A, thus we climb to B at the next step. Unfortunately, B is a local optimum of the problem. In order to avoid getting stuck at this point, the algorithm also allows to descend to the point C even though C is worse than B. Finally, we have opportunity to reach to D, the optimal point.

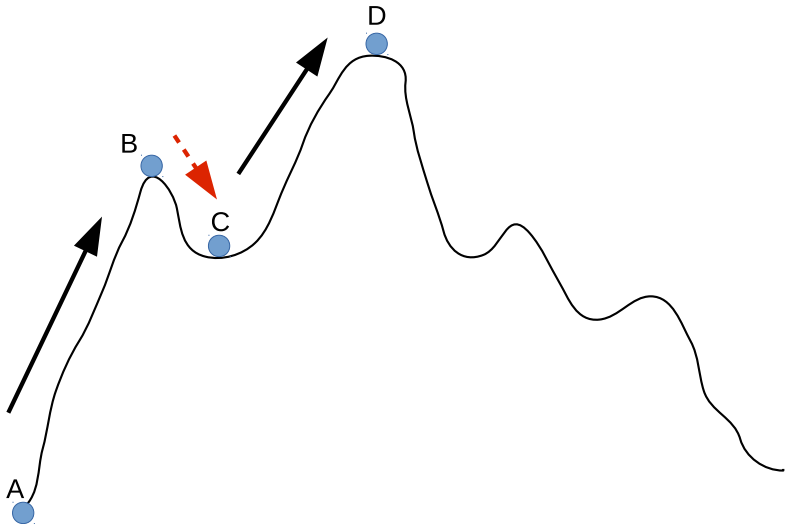


Figure 5.3: The simulated annealing algorithm runs on a simple example.

For solving constraint games, based on the simulated algorithm, we integrate into ConGa an algorithm named *CG-SA*. More precisely, in CG-SA, inspired from the process of annealing in metal work, we initially set the temperature TI high and then allow it to slowly "cool" as the algorithm runs. Let s be a strategy profile, then BR_s be the set of players for whom s is a best response. In Example 5.1, given a strategy profiles $s_1 = (1,2,3)$, then $BR_{s_1} = \{Z\}$ because s_1 is a best response of player Z. While the strategy profile $s_2 = (3,3,3)$ is a PNE because $BR_{s_2} = \{X,Y,Z\}$, namely s_2 is a best response of all the players. We assume that s is the current candidate of the algorithm, and s' is a neighbor of s. Then s' is said to be better than s if $|BR_{s'}| > |BR_s|$, hence the algorithm moves from s to s' unconditionally. However, even if s' is worse than s than the movement can be still accepted if $e^{\frac{|BR_{s'}|-|BR_s|}{TI}} \geq random([0,1])$ where $e^{\frac{|BR_{s'}|-|BR_s|}{TI}}$ is the acceptance probability.

At high temperature, the search is more random because it is suited to explore quickly large areas of the search space. But at low temperature, the search is more like hill-climbing. As the temperature is reduced that the algorithm gradually focuses on an area of the search space in which hopefully, a close to optimum solution can be found. This gradual "cooling" process makes the simulated annealing algorithm remarkably effective at finding a close to optimum solution when dealing with large problems which contain numerous local optima.

In Algorithm 5.7, we seek to apply CG-SA to increase the size of BR_s to $|\mathscr{P}|$, namely, to detect a PNE. In this algorithm, we set the initial temperature TI (line 2) and the cooling rate cr (line 3). TI plays an important role in CG-SA as it makes a direct impact on the algorithm performance. TI must be high enough such that the final candidate is independent of the starting one, but not too high because this may make the algorithm run very slowly. TI is calculated from the formula $e^{-\frac{|\mathscr{P}|}{TI}} = \tau$ where τ will be fixed later in experiments.

Algorithm 5.7. CG-SA, Simulated Annealing Algorithm for Constraint Games

```
1: function CG-SA(Constraint Game CG): tuple
2:     TI ← set the initial temperature
3:     cr ← set the cooling rate
4:     s ← get an arbitrary solution of the solver (with hard constraints if they exist)
5:     BR_s ← ∅                          ▷ BR_s is the number of players for whom s is best response
6:     Ψ ← s                             ▷ Ψ is the best strategy profile found so far
7:     BR_Ψ ← BR_s
8:     incTL ← 1
9:     while TI > ε do
10:        TL ← incTL × |𝒫|
11:        for (iter = 1 to TL) do
12:            if |BR_s| > |BR_Ψ| then                ▷ keep track of the best solution
13:                (Ψ, BR_Ψ) ← (s, BR_s)
14:            end if
15:            (s', BR_s') ← neighborSA(s,BR_s,TI)    ▷ get a neighbor of the current solution
16:            if (s', BR_s') = null then             ▷ no good enough neighbor is found
17:                if |BR_s| = |𝒫| then               ▷ solution s is the best response of all players
18:                    return s                       ▷ return the PNE
19:                else
20:                    (s, BR_s) ← (Ψ, BR_Ψ)          ▷ restart from the best solution
21:                end if
22:            else
23:                (s, BR_s) ← (s', BR_s')            ▷ move from the current solution to the neighbor
24:            end if
25:        end for
26:        incTL ← incTL + 1
27:        TI ← TI × (1 - cr)                         ▷ cool system
28:    end while
29:    return null                                    ▷ the algorithm fails to find a PNE
30: end function
```

For each strategy profile s, we keep along with a list of players BR_s (line 5). During search, we keep track of the best strategy profile which is a best response of the maximal number of players (line 6 - 7 and line 12 - 13). The algorithm may restart at this best strategy profile in line 20.

In line 10, TL is the number of iterations at a given temperature. In order to avoid cooling the system too fast, we perform TL times of iterations before decreasing the temperature (line 11-25). For example, let assume that at the current moment, the temperature $TI = 10$ and $TL = 100$. Then we iterate the movement from the current strategy profile to its neighbor for 100 times at the temperature 10 before TI is decreased, e.g. down to $TI = 9$. TL may vary from temperature to temperature. It is necessary to spend sufficiently long time at lower temperatures. Thus, TL is increased when we go down with TI (line 10 and line 26).

Line 15 is devoted to detect a neighbor (Algorithm 5.8). If no neighbor is found than one of two following possibilities occurs.

(i) No player can change to a better strategy since s is a best response of all players. Therefore, a PNE is returned (line 17 -18).

(ii) All neighbors are worse than the current candidate. Moreover, their acceptance possibilities

are not high enough to be chosen as the next candidate. The algorithm gets stuck in local optima. It should thus restarts with the best solution found so far (line 20).

Finally, the decreasing rule of the temperature TI is described in line 27. The algorithm may also stop when $TI \leq \varepsilon$ where ε is fixed later in experiments.

Algorithm 5.8. Find a random neighbor in the CG-SA algorithm

```
1: function NEIGHBORSA(tuple s, list of players BR_s, double TI): (s',BR_s')
2:     P' ← P \ BR_s                    ▷ P' is the list of players for whom s may be not a best response
3:     while P' ≠ ∅ do
4:         pick player i from P' randomly
5:         s' ← findBR-C*G(s,i)          ▷ find a random best response of player i from s
6:         if s' = null then
7:             BR_s ← BR_s ∪ {i}         ▷ s is a best response of player i
8:         else
9:             BR_s' ← eval(s', i, BR_s)
10:            r ← a random number in rang [0,1]
11:            if (|BR_s'| > |BR_s|) or (e^((|BR_s'|−|BR_s|)/TI) ≥ r) then
12:                return (s', BR_s')
13:            end if
14:        end if
15:        remove i from P'
16:    end while
17:    return null
18: end function
```

Finding a neighbor of a tuple s is specified in Algorithm 5.8. A neighbor s' is chosen in this algorithm according to the comparison between the size of the list BR_s and $BR_{s'}$. From the list of players for whom s may not be a best response (line 2), we try to find a best response which could be a good neighbor (line 3 -16). In line 5, the function of finding a random best response is previously specified in Algorithm 5.3 for CSG and in Algorithm 5.4 for COG. It is computationally complex to detect for which player a strategy profile is a best response because, for CSG, this concerns solving a Constraint Satisfaction Problem, while for COG, it is a Constraint Optimization Problem. Hence, it is necessary to decrease the times of finding best responses as much as possible. This is the reason why we always keep a list BR_s along with s during search. In line 3 - 16, the list is gradually filled when s is a best response of the player (line 6 - 7).

We find a best response s' from s and evaluate it against s (line 9 - 13). Line 9 is devoted to determine the list $BR_{s'}$ (Algorithm 5.9). The new strategy profile s' is accepted if it is either a better strategy profile or a worse one with a high enough acceptance probability (line 11-13).

Given a neighbor s', Algorithm 5.9 demonstrates how to determine the list $BR_{s'}$. Because s' is already a best response of player i, the list $BR_{s'}$ initially includes i (line 2). The main idea of this algorithm is as follows. Given the current list BR_s, for evaluating neighbor s', we seek to find a set of players $BR_{s'}$. This list is composed of player $j \in BR_s$ who does not make a beneficial deviation from s' (line 3 - 9). Hence, the upper bound of $|BR_{s'}|$ is $|BR_s|+1$. Therefore, s' is better than s only if no player in BR_s can deviate from s'. In all of the remaining cases, s' is worse than s. However, we remind that in CG-SA, a worse strategy profile can be still chosen if its probability is greater

Algorithm 5.9. Determine the list $BR_{s'}$ of a neighbor s' in the CG-SA algorithm

```
1:  function EVAL(tuple s', player i, list of players BR_s ): list of players BR_s'
2:      BR_s' ← {i}
3:      while BR_s ≠ ∅ do
4:          pick player j from BR_s randomly
5:          if dev-C*G(s', j) then return BR_s'
6:          end if
7:          BR_s' ← BR_s' ∪ {j}
8:          remove j from BR_s
9:      end while
10:     return BR_s'
11: end function
```

than or equal to a random number in [0,1]. It is possible to check the deviation for all players in BR_s. Nevertheless, the deviation verification for players is computationally complex. In addition, we may have to evaluate many neighbors of s in Algorithm 5.8. Henceforth, in Algorithm 5.9, if there exists a player in BR_s making a deviation then s' may be not better than s. In consequence, the list $BR_{s'}$ will be returned immediately (line 5). Although this evaluation between s and s' is incomplete, it allows us not to spend too much time for the costly deviation verification (for CSG in Algorithm 5.10 and for COG in Algorithm 5.11).

Figure 5.4 illustrates the evaluation (in Algorithm 5.9) between two strategies s and s' in a game composed of 5 players.

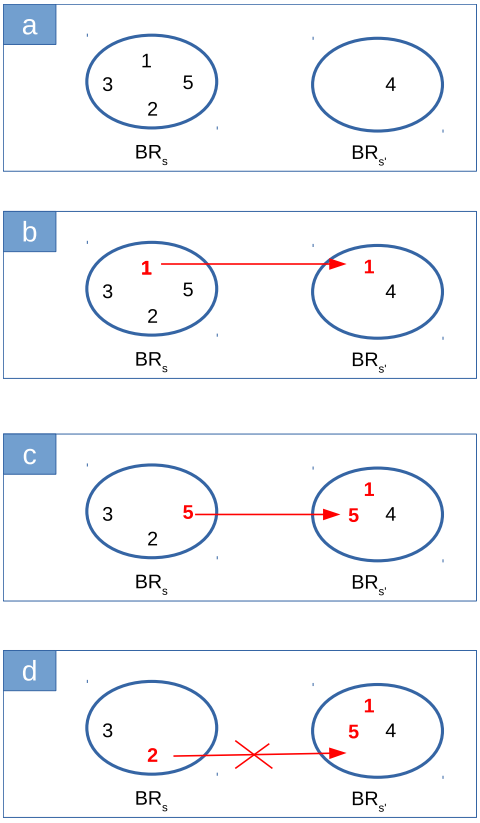


Figure 5.4: Algorithm 5.9 works on an example.

The strategy s is a best response of players $1,2,3,5$. The strategy s' is a best response of player 4 from s (Figure 5.4.**a**). In Figure 5.4.**b**, we randomly pick player 1 and verify if this player can deviate from s'. Because 1 can not deviate, it is inserted into $BR_{s'}$. The same thing happen with player 5 (Figure 5.4.**c**). In Figure 5.4.**d**, we find out that player 2 can make a beneficial deviation from s'. This function immediately stops and the list $BR_{s'} = \{1,4,5\}$ is returned.

Algorithm 5.10. Verify whether a player can make a beneficial deviation in CSG

```
1: function DEV-CSG(tuple s, int i): boolean
2:     initialize solver S_i with G_i (and hard constraints if they exists)
3:     for all j ∈ 𝒫 do                ▷ instantiate the values from s to the variables controlled by the others
4:         if i ≠ j then
5:             for all (k = 1; k ≤ |V_j|; k++) do
6:                 add constraints v_jk = s_jk where v_jk ∈ V_j
7:             end for
8:         end if
9:     end for
10:    set search tree to begin with ∀v_ik ∈ V_i, v_ik = s_ik
11:    fsol ← S_i.getSolution()                ▷ find the first solution of the solver
12:    if fsol ≠ null then
13:        if fsol = s then                    ▷ s is a solution of the solver
14:            return false
15:        else
16:            return true                     ▷ player i can deviate from s to fsol
17:        end if
18:    else        ▷ S_i has no solution, i.e. goal G_i is not satisfied by any tuple s = (s_i, s_−i) with s_i ∈ S_i
19:        return false                        ▷ s is also best response
20:    end if
21: end function
```

Algorithm 5.11. Verify whether a player can make a beneficial deviation in COG

```
1: function DEV-COG(tuple s, int i): boolean
2:     initialize solver S_i with G_i and opt_i (and hard constraints if they exists)
3:     for all j ∈ 𝒫 do                ▷ instantiate the values from s to the variables controlled by the others
4:         if i ≠ j then
5:             for all (k = 1; k ≤ |V_j|; k++) do
6:                 add constraints v_jk = s_jk where v_jk ∈ V_j
7:             end for
8:         end if
9:     end for
10:    set search tree to begin with ∀v_ik ∈ V_i, v_ik = s_ik
11:    fsol ← S_i.getSolution()                ▷ find the first solution of the solver
12:    if (fsol ≠ null) and (fsol = s) then
13:        get the current payoff x of player i from fsol
14:        add constraint v_opt_i > x          ▷ branch and bound
15:    end if
16:    if S_i.getSolution() ≠ null then
17:        return true                         ▷ player i can deviate to a solution of S_i
18:    end if
19:    return false
20: end function
```

Algorithm 5.10 is devoted to verify whether player i could make a beneficial deviation from s for

CSG. In line 3 - 9, we instantiate the values to the variables controlled by the remaining players to check whether s is a solution of player i. Because the search tree is set to begin with the variables and values of player i in s (line 10), if the solver finds a solution and s is a solution then s must be identical to the first solution $fsol$ (line 12 - 14). Otherwise, player i could make a beneficial deviation from s to $fsol$ (line 16). The solver has no solution suggests that the goal G_i is never satisfied hence player i cannot deviate to a better solution (line 19).

A similar procedure for COG is depicted in Algorithm 5.11. But it is more complex because s is a solution of the solver does not guarantee that player i cannot make a deviation. Hence, we need calculate the current payoff of i (line 13) and post a complementary branch and bound technique in line 14. Then, the solver has a solution means that player i can change to another strategy to improve his payoff (line 16 -18).

The two above algorithms are similar to the ones of finding a random best response (Algorithm 5.3 and Algorithm 5.4). But they are faster because we only need to check whether the player is able to deviate to a *better utility*. While in the previous algorithms, it is required to detect a random strategy profile such that the player gets the *best utility*. We also note that these two algorithms will be reused in the complete algorithms in Chapter 6.

Finally, we prove the correctness of the algorithm in the following proposition.
Proposition 5.3. *The CG-SA algorithm is correct.*

Proof. A PNE s is reported only if $|BR_s| = |\mathscr{P}|$, namely s is a best response of all players. In addition, s also satisfies the hard constraints. Indeed the CG-SA algorithm for constraint games is correct. □

5.4 Experiment

We have implemented three algorithms described above in the ConGa solver. We will demonstrate in this section the usefulness and the compactness of our constraint games representation as well as our solver ConGa. In terms of modeling tools, constraint games can overcome the drawbacks of the existing game representations in Section 2.3, because they can encode the game instances much larger or unrepresentable by the others. In terms of solving tools, thanks to the compactness of the game representation, ConGa could find a PNE in very large games even with the most simple algorithm CG-IBR. Additionally, the two metaheuristic algorithms (CG-TS and CG-SA) sometimes offer a better performance to the solver.

5.4.1 Description of Experimental Game Instances

We provide here the detailed description of the game instances in our experiments. Each game instance is specified by two main parameters: clients number and vendors number for *Location*

Games with Hard Constraint (abbreviated LGHC) (Example 4.12), dimensions number and players number for *Hypercube Network Congestion Games* (abbreviated HNCG) (Example 4.15). In the tables, v stands for the number of variables controlled by each player, d for the domain size of each variable, s for the number of strategies per player.

#client	**40**				**50**				**60**			
#vendor	***v***	***d***	***s***	***NF***	***v***	***d***	***s***	***NF***	***v***	***d***	***s***	***NF***
10	1	40	40	1.05E+17	1	50	50	9.77E+17	1	60	60	6.05E+18
11	1	40	40	4.61E+18	1	50	50	5.37E+19	1	60	60	3.99E+20
12	1	40	40	2.01E+20	1	50	50	2.93E+21	1	60	60	2.61E+22
13	1	40	40	8.72E+21	1	50	50	1.59E+23	1	60	60	1.70E+24
14	1	40	40	7.36E+23	1	50	50	8.54E+24	1	60	60	1.10E+26

Table 5.1: Game instances description of Location Games with Hard Constraints in the local search solver experiments. The number aE+b is equal to $a \times 10^b$.

#dimension	**5**				**6**				**7**			
#player	***v***	***d***	***s***	***NF***	***v***	***d***	***s***	***NF***	***v***	***d***	***s***	***NF***
100	5	5	120	100×120^{100}	6	6	720	100×720^{100}	7	7	5040	100×5040^{100}
120	5	5	120	120×120^{120}	6	6	720	120×720^{120}	7	7	5040	120×5040^{120}
140	5	5	120	140×120^{140}	6	6	720	140×720^{140}	7	7	5040	140×5040^{140}
160	5	5	120	160×120^{160}	6	6	720	160×720^{160}	7	7	5040	160×5040^{160}
180	5	5	120	180×120^{180}	6	6	720	180×720^{180}	7	7	5040	180×5040^{180}
200	5	5	120	200×120^{200}	6	6	720	200×720^{200}	7	7	5040	200×5040^{200}

Table 5.2: Game instances description of Hypercube Network Congestion Games in the local search solver experiments.

All the games experimented are Constraint Optimization Games with Hard Constraints, thus they can not be encoded by normal form and by boolean games (see Section 4.1.5). However, in order to demonstrate the compactness of constraint games, in the above tables, we give the number of entries which would be involved in the induced normal form representation in the column labeled by *NF*. As we can see in the tables, these game instances are out of storage capacity of the normal form representations. Especially, the largest game in our experiments consists of 200 players and 5040 strategies per player. A normal form representation of this game would have stored 200×5040^{200} entries (Table 5.2).

5.4.2 Experimental Results

In the experiments, each algorithm has been launched 100 times per instance. The max step is set to 50×10^6. If an algorithm fails to find a PNE within the max step then the time taken will be noted by the notation "–".

The average time results of the three algorithms on LGHC are depicted in Table 5.3 (on the next page). As we can see in the table, the solver is already able to find a PNE by CG-IBR within reasonable time for the large instances (the size of game instances is detailed in Table 5.1). The time needed for CG-IBR has been seen as the baseline time for evaluating the other algorithms, i.e. CG-TS and CG-SA. In several instances, CG-TS offers a better performance than CG-IBR

#client	40			50			60		
#vendor	*CG-IBR*	*CG-TS*	*CG-SA*	*CG-IBR*	*CG-TS*	*CG-SA*	*CG-IBR*	*CG-TS*	*CG-SA*
10	4.78	5.20	**4.18**	12.89	11.59	**7.78**	47.21	41.41	**23.21**
11	6.37	5.78	**3.81**	57.07	68.69	**25.22**	274.69	276.13	**82.06**
12	24.63	23.06	**11.13**	78.87	83.56	**35.01**	89.84	84.50	**31.04**
13	–	–	–	57.00	72.67	**22.40**	406.69	400.34	**108.80**
14	231.22	153.79	**86.60**	78.51	63.68	**23.02**	1143.49	633.41	**398.73**

Table 5.3: Average time (in seconds) taken of CG-IBR, CG-TS and CG-SA on Location Game with Hard Constraints. The tabu length is set to round of quarter of the number of players in CG-TS. For CG-SA, the cooling rate is set to 0.005. T1 is calculated from the formula $e^{\frac{-|\mathscr{P}|}{T1}} = 0.2$.

but not for all the instances. There even exists some instances on which CG-TS is worse than CG-IBR. In addition, the time needed for CG-IBR and CG-TS is almost the same. This suggests that the *tabu list* is not sufficient to significantly improve the solver. Actually, both the two algorithms randomly select a neighbor without being driven by an efficient metaheuristic (see Algorithm 5.2 and Algorithm 5.6). CG-SA can climb over this obstacle. By including an evaluation between the current candidate and its neighbors (see Algorithm 5.8), CG-SA beats both CG-IBR and CG-TS, as mentioned in the table. Moreover, in most of the cases, CG-SA is approximately 2 times faster than CG-IBR.

All the three algorithms have not found a PNE within the max step on the instance LGHC.13.40. A hypothesis is this instance does not contain a PNE. Like other local search algorithms, not surprisingly, these algorithms cannot guarantee to find a PNE as well as prove the absence of PNE in constraint games.

We also notice that the standard deviations in all these game instances are very large. It may be caused by the sensitiveness of initial point which are randomly generated at each launch for each instance in LGHC. Unfortunately, our interest in this thesis does not include heuristics for choosing good initial points. In fact, we leave this work for perspectives. We then analyze the solver performance with standard deviation in the instances of Hypercube Network Congestion Games which are less dependent of initial points.

The below tables (on the next page) represent the experimental results of the three algorithms on HNCG. The abbreviations *avg, sdt* stand for average time taken (in second), and standard deviation. The rate of *sdt* on *avg* in percent is depicted in the column labeled by *rate*.

The tables show an impressive result because all the algorithms run very fast with small standard deviation on the very large game instances (The size of game instances is detailed in Table 5.2). Namely, the algorithms converge to a PNE in short time. It is because Hypercube Network Congestion Games belong to the class of congestion games. The most prominent property of this game class is that from an arbitrary initial point, local search algorithm always converges to a PNE. It is also proven that the number of improvements is upper-bounded by $2 \times \sum_{r \in R} \sum_{i=1}^{n} |d_r(i)|$ where n is the number of players, R is the set of resources and d_r is the delay function of resource r (see Definition 2.6) [Rosenthal, 1973]. In other words, the algorithms have never get stuck in infinite looping or local optima.

#dimension	5			6			7		
#player	*avg*	*std*	*rate(%)*	*avg*	*std*	*rate(%)*	*avg*	*std*	*rate(%)*
100	8.69	0.58	6.71	13.35	1.04	7.81	305.83	62.31	20.38
120	9.37	0.47	5.03	17.97	1.61	8.95	445.90	64.03	14.36
140	10.44	0.39	3.71	18.45	1.03	5.58	1066.49	158.65	14.88
160	22.91	1.79	7.81	22.91	1.79	7.81	209.86	17.92	8.54
180	13.93	0.53	3.80	30.74	3.77	12.25	499.28	64.40	12.90
200	15.83	0.60	3.77	30.74	2.30	7.48	694.34	85.29	12.28

Table 5.4: Time taken of CG-IBR on Hypercube Network Congestion Games

#dimension	5			6			7		
#player	*avg*	*std*	*rate(%)*	*avg*	*std*	*rate(%)*	*avg*	*std*	*rate(%)*
100	8.56	0.55	6.42	12.58	0.76	6.06	286.50	62.75	21.90
120	9.40	0.42	4.43	18.19	2.22	12.22	422.44	76.66	18.15
140	10.43	0.46	4.38	17.92	1.00	5.58	991.97	166.25	16.76
160	11.38	0.48	4.25	21.44	1.84	8.57	191.71	15.61	8.14
180	13.96	0.53	3.77	29.55	3.40	11.51	462.01	68.26	14.77
200	15.88	0.53	3.36	29.39	2.11	7.18	626.77	88.34	14.10

Table 5.5: Time taken of CG-TS on Hypercube Network Congestion Games. The tabu length is set to round of quarter of the number of players.

#dimension	5			6			7		
#player	*avg*	*std*	*rate(%)*	*avg*	*std*	*rate(%)*	*avg*	*std*	*rate(%)*
100	8.17	0.35	4.26	20.72	2.15	10.40	329.44	38.80	11.78
120	9.46	0.44	4.66	30.55	5.74	18.79	861.45	203.39	23.61
140	10.86	0.56	5.17	30.95	3.73	12.06	1162.59	229.93	19.78
160	12.59	0.63	5.04	40.65	4.40	10.81	1521.89	176.50	11.60
180	15.35	0.28	1.80	58.80	9.56	16.27	2515.50	328.13	13.04
200	17.38	0.81	4.67	52.39	6.53	12.47	7464.92	1399.45	18.75

Table 5.6: Time taken of CG-SA on Hypercube Network Congestion Games. The cooling rate is set to 0.1. T1 is calculated from the formula $e^{\frac{-|\mathscr{D}|}{T1}} = 0.2$.

Generally, the results of CG-SA are worse than the ones of CG-IBR and of CG-TS. In CG-SA, the cooling process is very slow with acceptance probability in order to jump out local optima. Moreover, it also requires the time for the evaluation of neighbors. This procedure often takes large computation time. It is reasonable that CG-SA does not work well in these game instances without local optimum (or infinite loops). It rather fits well on the games like LGHC that contain numerous local optima.

5.5 Related Work

Local search has not been largely used for computing Nash equilibrium in games. Several related works often concentrate on finding mixed Nash equilibrium and very few works address PNE for general games.

The most trivial local search algorithms for finding PNE are *Iterated Best/Better Responses* which can be used on whatever game representations [Shoham and Leyton-Brown, 2009]. In

[Son and Baldick, 2004], the authors apply hybrid coevolutionary programming for differentiating local optima and PNE, not for computing PNE. There are also several works using local search for computing mixed equilibrium on normal form, such as [Gatti et al., 2012, Ceppi et al., 2010] as well as genetic algorithms [Ismail et al., 2007].

In [Ortiz and Kearns, 2002], it has been proposed an iterative and local message-passing algorithm for computing mixed Nash equilibrium on arbitrary undirected graphs. Similarly, [Vickrey and Koller, 2002] use a greedy local search algorithm in their proposal of finding approximate mixed Nash equilibrium. With the proposal which is not for finding Nash equilibrium but for learning graphical games, [Duong et al., 2009] explore two versions of simulated annealing algorithm for loss-minimizing neighborhoods.

Finally, to the best of our knowledge, no local search method has been proposed for computing Nash equilibrium in both action-graph games and classical boolean games so far.

5.6 Conclusion

In this chapter, we have presented a set of local search algorithms which allow to quickly detect one pure Nash equilibrium in constraint games. In detail, we have implemented three algorithms in the ConGa solver. First, we have proposed a baseline local search algorithm, what we called CG-IBR, the iterated best responses algorithm for solving constraint games. Second, we have modified CG-IBR by using a tabu list in order to avoid getting stuck in infinite looping. This algorithm is named CG-TS. Third, we have described CG-SA, an adapted version of the simulated annealing algorithm in constraint games.

Thanks to the compactness of constraint games, the local search solver is capable of solving game instances which are out of storage capacity of normal form. More precisely, among the three algorithms, CG-IBR and CG-TS are reasonable for solving some specific game classes such as congestion games in which the games do not consist of any local optima/infinite loop. While CG-SA seems well suitable for solving games including many local optima.

In our experiments, we have only tested a set of parameters randomly chosen for CG-SA, e.g. the cooling rate. It has been well-known that selecting good parameters in the simulated annealing algorithm could offer a much better performance. We are thus confident that the CG-SA algorithm can be more improved by a deeper study on good rules for choosing these parameters. There are also other lines of work for improvement on the existing algorithms, such as, heuristic for selecting good initial points, or restarts techniques. Moreover, the current solver could be extended by implementing other kinds of local search, for example, genetic algorithms.

Chapter 6

Complete Search for Constraint Games

Contents

6.1 CG-Enum-Naive: Naive Algorithm for Enumerating All Pure Nash Equilibrium . . . 90

6.2 Pruning Techniques . . . 91

6.2.1 Classical Techniques . . . 91

6.2.2 Advanced Techniques . . . 93

6.3 CG-Enum-PNE: Advanced Algorithm for Enumerating All Pure Nash Equilibrium . . . 96

6.4 CG-Enum-PPNE: Algorithm for Finding All Pareto Nash Equilibrium . . . 101

6.5 Experiment . . . 103

6.6 Related Work . . . 106

6.7 Conclusion . . . 107

Although finding one PNE is a very interesting problem in itself, finding all of them allows more freedom for choosing equilibrium that fulfills some additional requirements. We thus present in this chapter a complete solver for constraint games. In this solver, we first implement the naive algorithm on the constraint games representation. Then, we improve this algorithm by including several advanced techniques in an algorithm named *CG-Enum-PNE*. Among the PNE of a constraint game, some PNE are probably more desirable than the others. In other words, these PNE may be also Pareto Nash equilibrium while the others are not. It has been well-known that the correctness of Pareto Nash equilibrium is based on the completeness of PNE enumeration. Hence, it is also interesting to detect all PPNE after we have obtained the set of all PNE. That is what we have done in an algorithm called *CG-Enum-PPNE*. Finally, the experimental results demonstrate that our complete solver is faster than the normal form solver Gambit by one or two orders of magnitude. [1]

[1] This chapter could be seen as an extended version of our published paper [Nguyen and Lallouet, 2014].

6.1 CG-Enum-Naive: Naive Algorithm for Enumerating All Pure Nash Equilibrium

A natural algorithm is to use "generate and test" to enumerate equilibrium (Algorithm 2.1 in Section 2.2.1). This naive algorithm is however the only known algorithm for finding PNE [Turocy, 2013] and from the complexity result, it is unlikely that any fast (polynomial) algorithm could exist. This algorithm is actually the basis of the implementation of the Gambit solver for the PNE enumeration.

Algorithm 6.1 is similar to the one implemented in Gambit. The difference is that it works on constraint games, not on normal form.

Algorithm 6.1. The naive algorithm for enumerating all PNE

```
1:  function CG-ENUM-NAIVE(Constraint Game CG): setof tuples
2:      Nash ← ∅
3:      initialize solver S (with hard constraints if they exist)
4:      s ← S.getSolution()
5:      while s ≠ nul do
6:          if isPNE(s) then
7:              Nash = Nash ∪ {s}
8:          end if
9:          s ← S.getSolution()
10:     end while
11:     return Nash
12: end function
```

This naive algorithm consists in enumerating all strategy profiles which satisfy hard constraints (line 3 -10), testing each of them for each player for deviation (line 6) and skipping to the next profile when the Nash verification is done (line 9). The procedure of checking whether a tuple is a PNE is depicted in Algorithm 6.2. We recall that the function dev-C*G(s,i) in line 3 is previously specified in Algorithm 5.3 for CSG and Algorithm 5.4 for COG in Chapter 5.

Algorithm 6.2. Verify whether a tuple is a PNE

```
1: function ISPNE(tuple s): boolean
2:     for i ∈ 𝒫 do
3:         if dev-C*G(s, i) then
4:             return false
5:         end if
6:     end for
7:     return true
8: end function
```

Despite its simplicity, the naive algorithm benefits from the compactness of the framework, in particular in the games with hard constraints, because it only needs to consider strategy profile in a satisfiable search space already restricted by hard constraints.

Proposition 6.1. *CG-Enum-Naive is correct and complete.*

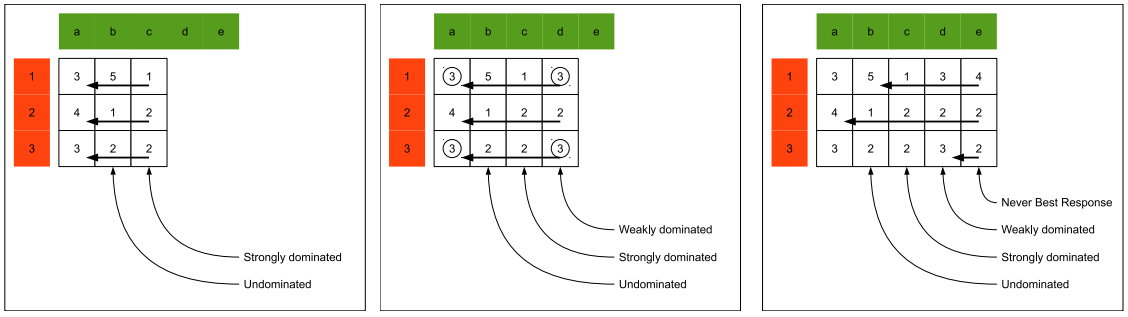
Proof. A strategy profile is said to be a PNE only if no player is able to deviate from it. Clearly, the satisfiable search space of the games consists of all tuples which do not violate hard constraints. The deviation checking is made on all strategy profiles in this satisfiable search space. The algorithm is thus correct and complete. □

6.2 Pruning Techniques

We give, in this section, a set of pruning techniques available for solving constraint games. At the beginning, we survey several existing techniques in the literature. We then propose our advanced techniques for efficiently solving constraint games.

6.2.1 Classical Techniques

Elimination of dominated strategies can be seen as a form of propagation for games [Apt, 2004, Apt, 2011]. Several types of domination have been devised, among them the best-known are *strict dominance* (also called *strong dominance*), *weak dominance* and *never best response*.



(a) Strict dominance *(b) Weak dominance* *(c) Never best response*

Figure 6.1: Dominance notions. In Figure 6.1a, strategy c is strictly dominated by strategy a. In Figure 6.1b, strategy d is weakly dominated by strategy a. In Figure 6.1c, strategy e is a never best response.

A player's strategy (e.g. strategy c in Figure 6.1a) is said to be strictly dominated by another strategy (e.g. strategy a in Figure 6.1a) if the player is always able to deviate from it to the other strategy in order to obtain a better utility, no matter what other players choose.

Definition 6.1 (Strict Dominance)**.** *Strategy s_i is said to be* strictly dominated *by strategy s'_i (or equivalently s'_i strictly dominates s_i) if* $\forall s_{-i}, u_i(s'_i, s_{-i}) > u_i(s_i, s_{-i})$.

A player's strategy s_i (e.g. strategy d in Figure 6.1a) is said to be weakly dominated by another strategy s'_i (e.g. strategy a in Figure 6.1a) if there is at least one set of other players' strategies for which s_i gives a worse utility than s'_i, while all other sets of other players' strategies give s_i the same utility as s'_i.

Definition 6.2 (Weak Dominance)**.** *Strategy s_i is said to be* weakly dominated *by strategy s'_i (or equivalently s'_i weakly dominates s_i) if $\forall s_{-i}, u_i(s'_i, s_{-i}) \geq u_i(s_i, s_{-i})$ and $\exists s_{-i}, u_i(s'_i, s_{-i}) > u_i(s_i, s_{-i})$.*

A strategy profile is said to be *never best response* (e.g. strategy d in Figure 6.1c) of a player if whatever remaining players choose, the player can always deviate to another strategy to get a better utility.

Definition 6.3 (Never Best Response)**.** *Strategy s_i is said to be* never best response (abbreviated NBR) *if $\forall s_{-i}, \exists s'_i$ such that $u_i(s'_i, s_{-i}) > u_i(s_i, s_{-i})$.*

These dominance notions derive a set of elimination techniques as follows.

Elimination of strictly dominated strategies. Clearly, a player would not like to choose his strictly dominated strategy because he can always deviate to a better strategy. Then, this strategy can be eliminated. The resulting game becomes smaller, thus simpler for solving. In general, an elimination of strictly dominated strategies is not an one step process. It may be an iterative procedure. Let us illustrate this process in the following example.

Example 6.1. *Consider the following 2-player game encoded by normal form*

	A	*B*	**C**
X	**2**,**3**	**4**,*5*	**1**,**1**
Y	*5*,**4**	*8*,*1*	*0*,**2**
Z	**3**,**3**	**7**,*2*	**5**,**2**

We see that C is strictly dominated by A and X is strictly dominated by Z. Thus we eliminate these two strategies. The game becomes as follows.

	A	*B*
Y	**5**, *4*	**8**, *1*
Z	**3**, *3*	**7**, *2*

Now Z is strictly dominated by Y, so we get

	A	**B**
Y	*5*, **4**	*8*, **1**

Finally, the pure Nash equilibrium of the game is

	A
Y	*5*, *4*

Given a game G, if G is finite and solved by elimination of strictly dominated strategies, then the resulting joint strategy is the unique pure Nash equilibrium of the game. Additionally, it is established by [Gilboa et al., 1990] that all iterated eliminations of strictly dominated strategies yield the same outcome. In other words, strict dominance is order independent.

Elimination of weakly dominated strategies. In the literature, several classical games can be solved by the elimination of weakly dominated strategies, such as *Guess Two Thirds Average* (Example 4.8) or *Travelers' dilemma* (Example 4.11). However, a weakly dominated strategy may appear in a pure Nash equilibrium as well. In consequence, during this eliminating process, we may "lose" several pure Nash equilibrium in games.

Elimination of never best responses. While with a strictly dominated strategy, the player can make beneficial deviation to one strategy, the player may deviate to a set of different strategies with a NBR. Hence, NBR notion can be seen as the extension of strictly dominated dominance. In other words, a strictly dominated strategy is a never best response as well. But the vice versa is not always true. As the class of never best responses is larger, there is more opportunity to detect never best responses than dominated strategies. The interesting issue is that never best responses of a player do not either appear in PNE since the player can always deviate. So they can be removed from the player's strategy set. For example, in Figure 6.1c, we can eliminate strategy e from the game.

In conclusion. The techniques described above could be seen as pruning techniques for solving games. However, the detection of domination for a strategy is computationally complex, because we need to consider the joint strategies of all remaining players which is well-known to be a Cartesian product. These techniques are henceforth not used in practice.

6.2.2 Advanced Techniques

In our complete solver, we prune never best responses, but in subgames. The second key technique is the fast deviation verification.

The first technique consists in elimination of *never best responses* in subgames. A subgame in constraint games is defined as follows.

Definition 6.4 (Subgame)**.** *Let* $CG = (\mathscr{P}, V, D, G, opt)$ *be a constraint game. A game* $CG' = (\mathscr{P}, V, D', G, opt)$ *is a subgame of* CG *if* $\exists i \in \mathscr{P}, D'_i \subseteq D_i$ *and* $\forall j \neq i \rightarrow D'_j = D_j$.

We denote by $<_{opt_i}$ the (partial) order on strategy profiles such that $s <_{opt_i} s'$ if $s_{-i} = s'_{-i}$ and $s|_x < s'|_x$ when $opt_i = \min(x)$ (resp. $s|_x > s'|_x$ when $opt_i = \max(x)$). In other words, $s <_{opt_i} s'$ means that player i gets a better utility on s than on s' with regard to the identical s_{-i} taken by other players.

Definition 6.5. *A strategy* s_i *for player* i *is a* never best response *if* $\forall s_{-i}, \exists s'_i$ *such that* $(s'_i, s_{-i}) <_{opt_i} (s_i, s_{-i})$.

Iterative elimination of NBR is a sound pruning for games [Apt, 2004] which additionally is stronger than elimination of strictly dominated strategies. But unfortunately their detection is very costly in the n-player case since it needs to know that this action will never been chosen by

player i for all strategy profiles of the other players. However, being a NBR in a subgame is a sufficient condition for not being an equilibrium.

Proposition 6.2. *Let CG be a constraint game and CG′ a subgame of CG such that $D_i \subseteq D'_i$. Let $s_j \in D'_j$ be a NBR in CG′ with $j \neq i$. Then for all s_{-j}, if $s = (s_j, s_{-j})$ is a PNE, then $s_i \notin D'_i$.*

Proof. The proof is by contradiction. Suppose there exists a PNE $s = (s_j, s_{-j})$ with $(s_{-j})_i \in D'_i$. Because s is a PNE, we have $\forall k \in \mathscr{P}, s_k \in br(s)|_k$. Then because s_j is a NBR for j in CG', there exists s'_j such that $(s'_j, s_{-j}) <_{opt_j} (s_j, s_{-j})$. Thus $s_j \notin br(s)|_j$. □

Let us illustrate the proposition above in the following example.

Example 6.2. *Given a constraint game CG including: the set of players is $\mathscr{P} = \{1,2,3\}$. Each player owns one variable: $V_1 = \{s_1\}, V_2 = \{s_2\}$ and $V_3 = \{s_3\}$ with $D_1 = D(s_1) = \{1\}, D_2 = D(s_2) = \{1,2,3\}, D_3 = D(s_3) = \{1,2\}$. The goals are $G_1 = \{s_1 \neq s_2, s_1 > s_3\}$, $G_2 = \{s_1 \leq s_2, s_2 > s_3\}$ and $G_3 = \{s_1 + s_2 = s_3\}$.*

Let CG' be a subgame of CG such that $D'_1 = \{1\}, \mathbf{D'_2} = \{\mathbf{1}\}, D'_3 = \{1,2\}$ (according to Proposition 6.2, $i = 2$). We see that $s_3 = 1$ is a NBR of player 3 in CG' because while $s_1 = 1$ and $s_2 = 1$, player 3 would like to choose the strategy $s_3 = 2$ which is a solution of his goal G_3 (according to Proposition 6.2, $j = 3$). Then, for all s_{-3}, if $s = (s_3, s_{-3})$ is a PNE, then $(s_2 = 1) \notin D'_2$. In other words, if (1,1,1) (which stands for $s_1 = 1, s_2 = 1, s_3 = 1$) is a PNE then $1 \notin D'_2$.

By applying inductively Proposition 6.2 on a sequence of assignments of strategies for player 1 to k with $k < n$, we see that if we detect that a strategy s_k is NBR for player k in the subgame defined by the sequence of assignments, then this strategy will not participate to a PNE and we can prune it. In Example 6.2, the strategy $s_3 = 1$ is a NBR for player 3 in the subgame defined by the sequence of assignments (1,1) (which stands for $s_1 = 1$ and $s_2 = 1$), then we can remove 1 from D'_3 when we deal with this subgame.

We then present our technique for fast deviation checking. The issue which motivates us to propose this technique is the complexity of the verification process for each player's deviation. We first demonstrate that the naive technique (in Algorithm 6.1) is subject to a form of trashing. The following example shows that some deviations are performed several times.

Example 6.3. *Let G be the 2-player game defined by the following table:*

G		*y*		
		1	*2*	*3*
x	*a*	*(0,1)$_\alpha$*	*(1,0)*	*(1,0)*
	b	*(0,1)$_\beta$*	*(0,0)*	*(1,0)*
	c	*(1,0)*	*(1,1)*	*(0,0)*

We assume that the enumeration starts by player x. The first tuple to be enumerated is $(a,1)$ denoted by α. Deviation is checked for player y and no deviation is found. Then deviation is checked for player x and a deviation towards $(c,1)$ is found. Thus this tuple is not a PNE. The

next candidate is $(b,1)$ *denoted by* β*. This tuple is checked for player y and again no deviation is found. But when checked for player x, the same deviation towards* $(c,1)$ *is found as for tuple* α*.*

This form of trashing is a strong motivation to investigate search and pruning techniques for constraint games. To introduce our technique, we first recall [Gottlob et al., 2005] where the authors introduce (originally for graphical games) a CSP composed of *Nash constraints* to represent best responses for each player.

Definition 6.6 (GGS-CSP)**.** *Let* $CG = (\mathscr{P}, V, D, G, opt)$ *be a COG. The* Nash constraint *of player* $i \in \mathscr{P}$ *is* $g_i = (V_c, T)$ *where* $T = \{t \in D^V \mid \nexists t' \in D^V \text{ s.t. } t' <_{opt_i} t\}$*. The* GGS-CSP $\mathscr{G}(CG)$ *of CG is the set of Nash constraints for all players.*

This CSP has the important property that it defines the PNE of the game:

Theorem 6.1. *([Gottlob et al., 2005]) t is a PNE of* $CG \leftrightarrow t \in sol(\mathscr{G}(CG))$

Then it follows that a PNE of a constraint game, *CG*, has a support in all of its Nash constraints.

Our technique consists to perform a traversal of the search space by assigning the variables of each player in turn according to a predefined ordering on $\mathscr{P}$. For each candidate tuple, we seek for supports by performing an incremental computation of the Nash constraints. Each computed deviation is recorded in a table for each player. By retrieving tuples in Nash constraints, we can avoid computing costly deviations.

However, since we are studying general games, each Nash constraint has the same arity as the whole problem, which is challenging in terms of space requirements. First, note that any tuple deleted from a table does not hinder the correctness of the Nash test. It may only force a deviation to be computed twice. Hence we are free to limit the size of the tables and trade space with time. In practice, two features limit the size of the tables.

First, deviation checks are performed in reverse player ordering. It means that a tuple checked for the first player must have succeed the deviation test for all other players. In practice for most problems, this limits the number of tuples reaching the upper levels. Second, we can delete a tuple t recorded in a table when we can ensure that no candidate t' will deviate anymore towards t. This property is given by the following independence of subgames theorem. We denote by $br_i(t)$, the set of best responses from t for player i such that $\forall t' \in br_i(t), t'_{-i} = t_{-i}$ and t' is a best response of player i.

Proposition 6.3. *Let CG be a constraint game and* CG' *a subgame of CG such that* $D'_i \subseteq D_i$*. Let* $D'' = D \setminus D'$*,* $t' \in D'^V$ *and* $t'' \in D''^V$*. Then* $\forall j \in \mathscr{P}, j \neq i \rightarrow br_j(t') \cap br_j(t'') = \emptyset$*.*

Proof. The proof is by contradiction. Suppose there exists $t' \in D'^V$ and $t'' \in D''^V$ such that $br_j(t') \cap br_j(t'') \neq \emptyset$. Let $s \in br_j(t') \cap br_j(t'')$. Then since $s \in br_j(t')$, $s_i \in D'^{V_i}$. Since $s \in br_j(t'')$, $s_i \in D''^{V_i}$. Hence the contradiction. □

Example 6.4. *We consider the following constraint game CG: the set of players is* $\mathscr{P} = \{X, Y, Z\}$*. The set of variables V is composed of :* $V_X = \{x\}, V_Y = \{y\}$ *and* $V_Z = \{z\}$ *with* $D_x = \{2,3\}, D_y = \{0,1,2\}, D_z = \{1,2\}$*. The goals are* $G_X = \{x = y+z\}$*,* $G_Y = \{y = x-z\}$ *and* $G_Z = \{x+1 = y+z\}$*.*

Let CG' be a subgame of CG such that $D'_x = \{2,3\}, D'_y = \{0,1,2\}, \mathbf{D'_z} = \{\mathbf{1}\}$, then $D''_z = \{2\}$. Given $t' = (2,2,1) \in D'^V, t'' = (2,2,2) \in D''^V$. For player X, $br_X(t') = \{(3,2,\mathbf{1})\}, br_X(t'') = \{(2,2,\mathbf{2}),(3,2,\mathbf{2})\}$, then $br_X(t') \cap br_X(t'') = \emptyset$. For player Y, $br_Y(t') = \{(2,1,\mathbf{1})\}, br_Y(t'') = \{(2,0,\mathbf{2})\}$, then $br_Y(t') \cap br_Y(t'') = \emptyset$.

This proposition is also illustrated in Figure 6.2. If we split the search space following player Z, best responses for players X and Y are forced to remain in different subspaces.

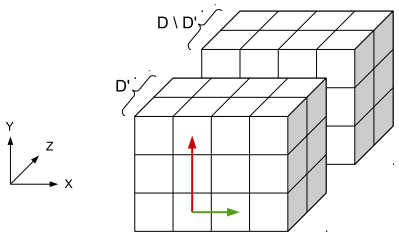


Figure 6.2: Independence of subgames

By applying inductively Proposition 6.3 on a sequence of assignments of strategies for player 1 to k with $k < n$, we see that two branches of the search tree will not share best responses for the remaining unassigned players. Hence we can freely remove all tuples from the table of subsequent players once the branch is explored.

6.3 CG-Enum-PNE: Advanced Algorithm for Enumerating All Pure Nash Equilibrium

In order to simplify the exposition, we assume that each player i only controls one variable x_i with domain D_i. The extension to more than one variable per player is not difficult (indeed our complete solver does not have this limitation since many examples require a player to control several variables) . We propose a tree-search algorithm for finding all PNE. The general method is based on three ideas:

- All candidates (except those which are detected as NBR) are generated in lexicographic order;
- Best responses for each player are recorded in a table;
- Whenever a domination check is performed, it first checks this player's recorded best responses.

We assume given an ordering on players from 1 to n. The main algorithm (Algorithm 6.3) launches the recursive traversal (Algorithm 6.4) starting by player 1. We distinguish the original domains of the variables (called D) used to compute deviations from their actual domain explored by the search tree (called A) and subject to pruning by arc-consistency on hard constraints.

Propagation of hard constraints allows to ensure that no forbidden tuple will be explored. If the propagation returns *false*, then at least one domain has been wiped out and there is no solution in

Algorithm 6.3. CG-Enum-PNE, the complete algorithm for Nash equilibrium enumeration

```
1: global:
2:     BR: array[1..n] of tuples                ▷ best responses for all players
3:     cnt: array[1..n] of integer              ▷ counters for NBR detection
4:     Nash: set of tuples                      ▷ Nash equilibrium
5:     S: global solver

6: function CG-ENUM(Game CG): setof tuples
7:     Nash ← ∅
8:     initialize solver S with hard constraints
9:     A ← D
10:    enum(A, 1)
11:    return Nash
12: end function
```

Algorithm 6.4. Enumerate the strategy profiles in the CG-Enum-PNE algorithm

```
1: procedure ENUM(domains A, int i)
2:     if S.propagate(A) then
3:         if i > n then
4:             checkNash(tuple(A),n)
5:         else
6:             BR[i] ← ∅
7:             cnt[i] ← Π_{j>i}|D_j|
8:             while A_i ≠ ∅ do
9:                 choose v ∈ A_i
10:                B ← A
11:                enum((B_{-i}, (B_i = {v})), i+1)
12:                A_i ← A_i − {v}
13:                if cnt[i] ≤ 0 then
14:                    checkEndOfTable(A, i)
15:                    break
16:                end if
17:            end while
18:        end if
19:    end if
20: end procedure
```

this subspace. Otherwise domains A are reduced according to arc-consistency. Values for each player are then recursively enumerated. When a tuple is reached, it is checked for Nash condition (line 4) by Algorithm 6.5. Otherwise, at least one domain remains to be explored. Each player i owns a table $BR[i]$ of best responses, initialized empty (line 6) and a counter $cnt[i]$ initialized with the size of the subspace needed to detect potential never best responses (line 7). For the sake of efficiency, the table $BR[i]$ is actually implemented by a search tree. Hence insertion and search are done in $O(n)$. After the recursive call of *enum* (line 11), we test whether all the subspace after player i has been checked for deviation. Then all subsequent values are *never best responses*. In this case an exit from the loop causes backjumping to the ancestor node. This backjumping is done after the exploration by *checkEndOfTable* (Algorithm 6.8) of the potential values which are stored in the table and belong to the unexplored space (lines 13-16).

Algorithm 6.5. Verify whether a tuple is PNE in the CG-Enum-PNE algorithm

```
1: procedure CHECKNASH(tuple t, int i)
2:     if i = 0 then
3:         Nash ← Nash ∪ {t}
4:     else
5:         d ← search_table(t, BR, i)
6:         if d = ∅ then
7:             d ← findBestResponses-C*G(t, i)
8:             if d = ∅ then d ← {(t_{-i}, y) | y ∈ D_i} end if
9:             insert_table(i, BR, d)
10:            cnt[i] - -
11:        end if
12:        if t ∈ d then checkNash(t, i − 1) end if
13:    end if
14: end procedure
```

Algorithm 6.6. Find the set of best responses for CSG in the CG-Enum-PNE algorithm

```
1: function FINDBESTRESPONSES-CSG(tuple t, int i): set of tuples
2:     d ← ∅
3:     initialize solver S_i with G_i (and hard constraints if they exist)
4:     add constraints x_j = t_j for all j ≠ i
5:     sol ← S_i.getSolution()
6:     while sol ≠ nil do
7:         d ← d ∪ {sol}
8:         sol ← S_i.getSolution()
9:     end while
10:    return d
11: end function
```

Algorithm 6.7. Find the set of best responses for COG in the CG-Enum-PNE algorithm

```
1: function FINDBESTRESPONSES-COG(tuple t, int i): set of tuples
2:     d ← ∅
3:     initialize solver S_i with G_i and opt_i (and hard constraints if they exist)
4:     add constraints x_j = t_j for all j ≠ i
5:     save the current state of S_i
6:     sol ← S_i.getOptimalSolution()
7:     if sol = nil then
8:         return d
9:     else
10:        get the optimal value opt of player i from sol
11:        restore the previous state of S_i
12:        add constraints opt_i = opt
13:        sol ← S_i.getSolution()
14:        while sol ≠ nil do
15:            d ← d ∪ {sol}
16:            sol ← S_i.getSolution()
17:        end while
18:    end if
19:    return d
20: end function
```

The *checkNash* procedure in Algorithm 6.5 verifies whether a player can make a beneficial deviation from a tuple. Since the exploration of the search tree is done level by level, the verification starts from the deepest level. First the tuple is searched in the table of stored best response for this player (line 5). If not found, a solver for G_i is called in function *findBestResponses-C*G* (line 7). This function returns the set of deviation for player i from a tuple t. There can be more than one deviation. In a CSG (Algorithm 6.6), it means that several assignments satisfy the constraints of G_i. In a COG (Algorithm 6.7), it means that the optimal value is reached for more than one point.

If d is empty, it means that there is no possible action for player i which can satisfy the constraints of her goal. Indeed, a tuple can be an equilibrium even if a (or all) player is unsatisfied. In this case we return all strategy profiles in which each profile is (t_{-i}, y) where $y \in D_i$ is a strategy controlled by player i since all these strategies can participate to a PNE (line 8).

Algorithm 6.6 is devoted to detect the set of best responses of player i from the tuple t. A strategy profile $s = (t_{-i}, y)$ is said to be a best response of player i in CSG if s is a solution of G_i. In line 4, we add the constraints into the solver S_i for assigning values from t_{-i} for all variables controlled by other players. Then, we launch S_i for finding all the player's best responses from t (line 5-9). The similar function for COG is depicted in Algorithm 6.7. However, it is more complex because we have to find all optimal solutions of the solver S_i. Hence, it is required to solve a Constraint Optimization Problem instead.

The procedure *checkEndOfTable* depicted in Algorithm 6.8 is used when the subset has been explored and we are able to perform backjumping. In this case, all tuples of the table which belong to the unexplored zone are checked for PNE.

Algorithm 6.8. Check end of table in the CG-Enum-PNE algorithm

```
1: procedure CHECKENDOFTABLE(domain A, int i)
2:     for all t ∈ BR[i] such that t ∈ Π_{i=1..n} A_i do
3:         checkNash(t, n)
4:     end for
5: end procedure
```

Let us illustrate how CG-Enum-PNE algorithm works on Example 6.5 in Figure 6.3 (on the next page).

Example 6.5. *Given a CSG: the set of players is* $\mathscr{P} = \{X, Y, Z\}$. *Each player owns one variable:* $V_X = \{x\}, V_Y = \{y\}$ *and* $V_Z = \{z\}$ *with* $D_x = D_y = \{1, \ldots, 9\}$ *and* $D_z = \{1, 2, 3\}$. *The goals are* $G_X = \{x = yz\}$, $G_Y = \{y = xz\}$ *and* $G_Z = \{xy \leq z \leq x + y, (x+1)(y+1) \neq 3z\}$.

The resolution starts in Figure 6.3.**a** (abbreviated 6.3.**a**) with tuple 111. During the descent to the leaf, X counter is initialized to $|D_Y| \times |D_Z| = 9 \times 3 = 27$, Y's counter to $|D_Z| = 3$, and Z's counter to 1. Tuple 111 is first checked for deviation for Z. It is found in 6.3.**b** that best responses for Z are 111 itself and 112, thus we store these two values in Z's table. Then we check players Y and X for deviation and find that none of them deviate. Thus 111 is a PNE, we record it and store it in Y and X's tables. Y's counter is decremented to 2.

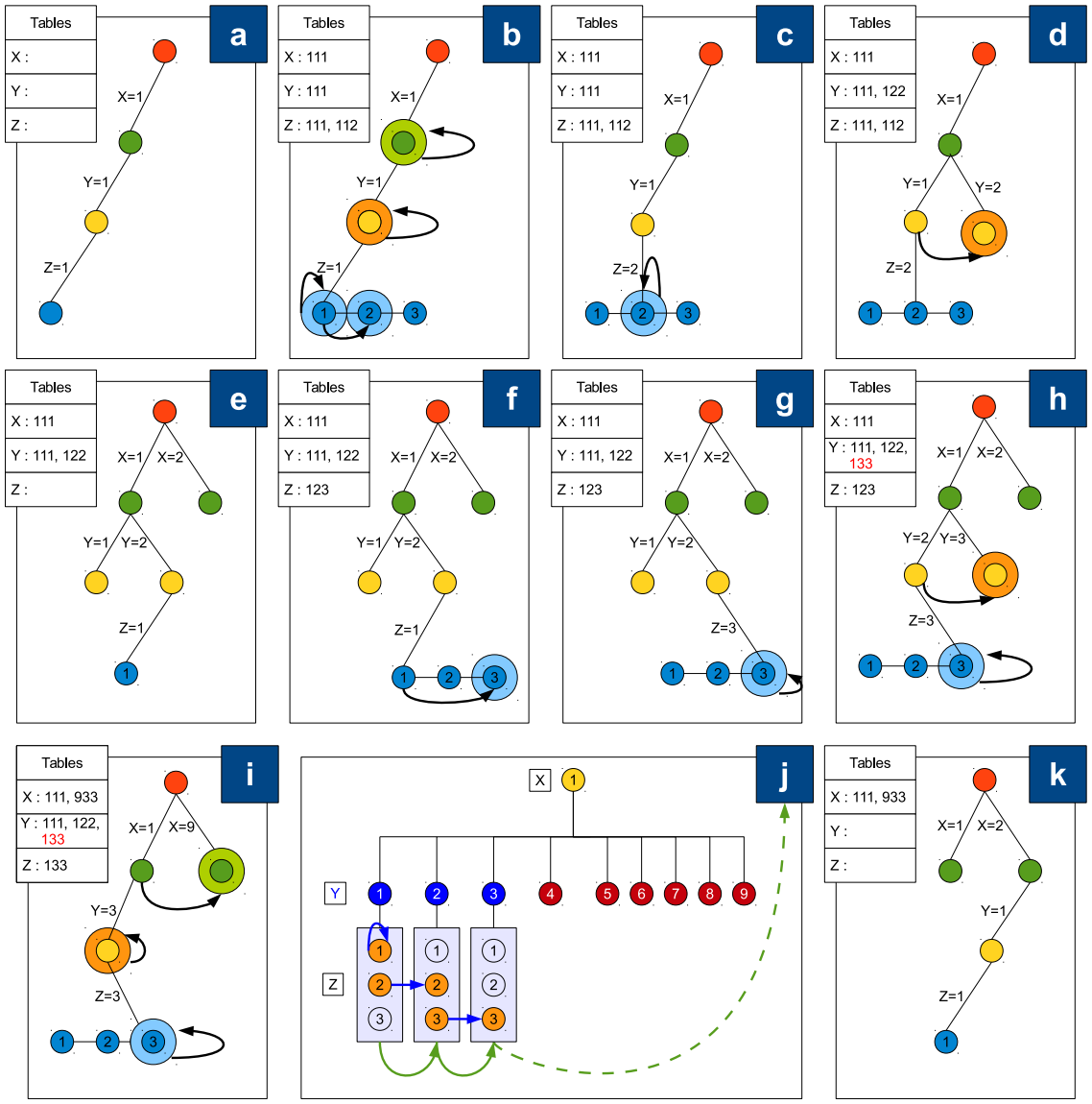


Figure 6.3: The CG-Enum-PNE algorithm runs on an example.

Since the possible deviations for Z are explored, we continue by *checkEndOfTable* for Z to 112 as in 6.3.**c**. This tuple is obviously a best response for Z (as found in Z's table), and is checked for deviation for the other players in reverse order. It is thus checked for Y in 6.3.**d** and a deviation is found to 122, inserting 122 in Y's table. Y's counter is decremented to 1.

Then backtracking occurs in 6.3.**e** at Y's level, which resets the table for Z. The next candidate is 121. In 6.3.**f**, a deviation is found for Z to 123. In 6.3.**g**, the tuple 123 is considered by *checkEndOfTable* for Z, checked for deviation for Z and found stable by looking up in Z's table. A further check for Y in 6.3.**h** finds a deviation to 133 and stores this tuple in Y's table. Y's counter is decremented to 0.

In this example, the domain of player Z is of size 3. Hence $cnt[Y]$ was initialized to 3. All tested tuples are of the form $1yz$ where 1 is the value on X. It happens in this example that only by exploring values 1, 2, 3 for Y yield a complete traversal of the subspace defined by Z (all values

from Z's domain have been considered). Thus we know that, from that point, only 133 recorded in Y's table can be a PNE with $X = 1$. The other remaining values of player Y are NBR. It is sufficient to check tuple 133 in *checkEndOfTable* as depicted in 6.3.**i**. Deviation is found for player X to 933, so 133 is not a PNE. Then, backjumping can occur. Figure 6.3.**j** makes a small summary of player Y's deviation possibilities. We see that tuples 111, 112, 123 are stable for Z, thus these tuples were lifted to Y level to be checked for Y's deviation. Solid arrows depict the deviations recorded for player Y on the different values for Z. After having checked 133, we can backjump to the next value of player X (the dotted arrows) (6.3.**k**).

In general *checkEndOfTable* tests all tuples of $BR[Y]$ which belong to the unexplored part of the search space. Here, we perform a lazy NBR detection that is incomplete but comes almost for free because it only takes a counter. Note that by Proposition 6.3, when exploring $X = 2$, the table for Y can be reset because no other tuple will deviate to a tuple where $X = 1$.

Note that the tables are actually implemented by a tree whose nodes represent all players but i, with i's best responses attached on the leaves. Thus the search for deviation in the table does not depend on the number of recorded tuples but only on the number of players and is performed in $O(|\mathscr{P}|)$.

Proposition 6.4. *CG-Enum-PNE is correct and complete.*

Proof. Correctness comes from the one of PNE check (Theorem 6.1). A reported PNE has been checked for deviation for every player. Either the tuple has been recorded in the table as deviation from another one, or had been directly checked by the solver against the player's goal. Completeness is due to the traversal of the whole search space and soundness of never best response pruning. □

6.4 CG-Enum-PPNE: Algorithm for Finding All Pareto Nash Equilibrium

Based on CG-Enum-PNE, we propose an algorithm called CG-Enum-PPNE which allows to find all Pareto Nash equilibrium in a constraint game. The unique difference between the two algorithms is that, in CG-Enum-PPNE, we perform an additional checking of Pareto optimality among the pure Nash equilibrium of the game. Hence, we just need to modify Algorithm 6.5 to a new algorithm in CG-Enum-PPNE (Algorithm 6.10). Given this, the other algorithms are the same like the ones implemented in CG-Enum-PNE.

Given t, t' two pure Nash equilibrium in a constraint game, Algorithm 6.9 verifies whether there exists a Pareto dominance between the two tuples. We assume that all players would like to maximize their utilities. In Algorithm 6.9, the tuple t is said to dominate the tuple t' if $\forall i \in \mathscr{P}, t|_{x_i} > t'|_{x_i}$ where the optimization condition of player i is $opt_i = \max(x_i)$ (line 3 - 9). Otherwise, there is no Pareto dominance if $\exists i, j \in \mathscr{P}$ such that $t|_{x_i} \geq t'|_{x_i}$ and $t|_{x_j} \leq t'|_{x_j}$ (line 11 - 15) (See

Algorithm 6.9. Verify whether a PNE is dominated in Pareto Optimality

```
1: function DOMPARETO(tuple t, tuple t'): integer
2:     dom ← true
3:     for i ∈ 𝒫 do
4:         if (t|_{x_i} ≤ t'|_{x_i}) then
5:             dom ← false
6:             break
7:         end if
8:     end for
9:     if dom then return 1
10:    end if
11:    for i ∈ 𝒫 do
12:        if (t|_{x_i} ≥ t'|_{x_i}) then
13:            return 0
14:        end if
15:    end for
16:    return -1
17: end function
```

Definition 4.8). In the remaining case, the tuple t is dominated by the tuple t' in Pareto optimality (line 16). In detail, the function in Algorithm 6.9 returns 1 if t dominates t', -1 if t is dominated by t' and 0 if there is no dominance.

Algorithm 6.10. Verify whether a tuple is PPNE

```
1: procedure CHECKPARETONASH(tuple t, int i)
2:     if i = 0 then
3:         isPPNE ← true
4:         for all t' ∈ Nash do
5:             x ← domPareto(t, t')
6:             if x = 1 then
7:                 Nash ← Nash \ {t'}
8:             end if
9:             if x = −1 then
10:                isPPNE ← false
11:                break
12:            end if
13:        end for
14:        if isPPNE then
15:            Nash ← Nash ∪ {t}
16:        end if
17:    else
18:        d ← search_table(t, BR, i)
19:        if d = ∅ then
20:            d ← findBestResponses-C*G(t, i)
21:            if d = ∅ then d ← {(t_{−i}, y) | y ∈ D_i} end if
22:            insert_table(i, BR, d)
23:            cnt[i] - -
24:        end if
25:        if t ∈ d then checkParetoNash(t, i − 1) end if
26:    end if
27: end procedure
```

Algorithm 6.10 shows how to verify whether a tuple is a PPNE. This algorithm is derived from

Algorithm 6.5 with an additional procedure for checking the Pareto dominance in line 3 - 16. When a new PNE t is detected, all the PNE dominated by t are removed from the set of Nash equilibrium (line 6 - 8). Similarly, the tuple t is inserted into the Nash set only if it is not dominated by an existing PNE (line 14-16).

Proposition 6.5. *CG-Enum-PPNE is correct and complete.*

Proof. From the correctness of CG-Enum-PNE (Proposition 6.4), a PPNE reported is a correct PNE. Then, because the dominance check for Pareto optimality successively progresses during search, a tuple is said to be PPNE when it is not dominated by any other PNE. Indeed, the algorithm is correct. Moreover, the PPNE set is reduced from the PNE set by eliminating the PNE dominated in Pareto optimality. Hence, the algorithm is complete as well. □

6.5 Experiment

We have implemented the algorithms described above in a complete solver which has been integrated into the ConGa solver. In this section, we first compare the performance among the PNE enumeration algorithms. Then, we evaluate the algorithm for finding all PPNE in constraint games.

We have performed experiments on classical games of the Gamut suite [Nudelman et al., 2004] and some games with hard constraints. Results are summarized in the below tables in which the name of the game is followed by the number of players and the size of the domain. Gamut games are *Guess Two Thirds Average* (abbreviated GTTA, Example 4.8), *Location Game, Gamut version* (abbreviated LGGV, Example 4.9), *Minimum Effort Game* (abbreviated MEG, Example 4.10) and *Traveler's Dilemma* (abbreviated TD, Example 4.11). The other games are *Location Game with Hard Constraints* (abbreviated LGHC, Example 4.12) and *Cloud Resource Allocation Game* (abbreviated CRAG, Example 4.16).

For each instance, we have compared the CG-Enum-PNE algorithm (Algorithm 6.3) to the game solver Gambit and to the naive enumeration algorithm named CG-Enum-Naive (Algorithm 6.1). This algorithm works like Gambit by examining each tuple but the only difference is that it uses the compact constraint game representation. In all the tables, time taken for each solver is given in seconds and time out is set to 20000 seconds. The number $aE{+}b$ is equal to $a \times 10^b$. TO stands for *Time Out*, MO for *Memory Out* and "–" means there is no information (for example, if the generation times out, it is not possible to launch the resolution).

The experiment on Gambit is divided into two steps: we first generate the normal form matrix (column NF gen. in Table 6.2) and then we launch the command line `gambit-enumpure` on the normal form to find all PNE. Table 6.1 (on the next page) describes the experimental game instances over normal form on games without hard constraints. We do not mention the information involving in games with hard constraints anymore because they are unrepresentable by normal form. Therefore, they are not applicable (abbreviated N/A) for being solved by Gambit.

Name	#S	#Tuple	NF	Size
GTTA.3.101	101	1.0E+6	3× (1.0E+6)	14 MB
GTTA.4.101	101	1.0E+8	4× (1.0E+8)	1.8 GB
GTTA.5.101	101	1.0E+10	5× (1.0E+10)	216 GB
LGGV.2.1000	1000	1.0E+6	2× (1.0E+6)	11 MB
LGGV.2.2000	2000	4.0E+6	2× (4.0E+6)	40 MB
LGGV.2.3500	3500	1.2E+7	2× (1.2E+7)	116 MB
LGGV.2.5000	5000	2.5E+7	2× (2.5E+7)	231 MB
LGGV.2.20000	20000	4.0E+9	2× (4.0E+9)	3.7 GB
MEG.3.100	100	1.0E+6	3× (1.0E+6)	14 MB
MEG.4.100	100	1.0E+8	4× (1.0E+8)	1.9 GB
MEG.5.100	100	1.0E+10	5× (1.0E+10)	**241 GB**
MEG.30.2	2	1.1E+9	30× (1.1E+9)	91 GB
MEG.35.2	2	3.4E+10	35× (3.4E+10)	–
TD.3.99	99	9.7E+5	3× (9.7E+5)	7.8 MB
TD.4.99	99	9.6E+7	4× (9.6E+7)	997 MB
TD.5.99	99	9.1E+9	5× (9.1E+9)	119 GB

Table 6.1: Description of the game instances solved by the complete algorithms

In Table 6.1, the notation *#S* stands for the number of strategies controlled by each player; *#Tuple* for the total of strategy profiles of the game instances; *NF* for the number of entries storing in every game instance's normal form; finally *Size* for the size of files saved in external memory of all the instances. As predicted, the size of the normal form soon becomes intractable. Especially, it grows up to 241GB for MEG.5.100.

Name	NF Gen.	Gambit	CG-Enum-Naive			CG-Enum			#PNE
	Time	*Time*	*Time*	*#Cand*	*#Dev*	*Time*	*#Cand*	*#Dev*	
GTTA.3.101	2	18	4	1.0E+6	1.4E+6	**1**	1.0E+4	1.0E+4	1
GTTA.4.101	125	1844	337	1.0E+8	1.3E+8	**12**	1.0E+6	1.0E+6	1
GTTA.5.101	14705	*MO*	*TO*	–	–	**816**	1.0E+8	1.1E+8	1
LGGV.2.1000	2	131	340	1.0E+6	1.0E+6	**8**	2.0E+3	1.5E+3	0
LGGV.2.2000	6	534	1448	4.0E+6	4.0E+6	**33**	4.0E+3	3.5E+3	0
LGGV.2.3500	18	2614	6893	1.2E+7	1.2E+7	**96**	7.0E+3	6.0E+3	0
LGGV.2.5000	35	7696	19990	2.5E+7	2.5E+7	**205**	1.0E+4	9.0E+3	0
LGGV.2.20000	551	*MO*	*TO*	–	–	**3462**	4.0E+4	3.9E+5	0
MEG.3.100	1	14	1	1.0E+6	1.0E+6	**1**	1.9E+4	1.5E+4	100
MEG.4.100	96	1555	46	1.0E+8	1.0E+8	**10**	1.9E+6	1.3E+6	100
MEG.5.100	11414	*MO*	3844	1.0E+10	1.0E+10	**447**	1.9E+8	1.2E+8	100
MEG.30.2	8998	*MO*	**437**	1.1E+9	2.1E+9	1085	5.4E+8	1.1E+9	2
MEG.35.2	*TO*	–	**15408**	3.4E+10	6.9E+10	*TO*	–	–	2
TD.3.99	2	15	1	9.7E+5	9.8E+5	**1**	1.9E+4	1.5E+4	1
TD.4.99	82	1572	28	9.6E+7	9.7E+7	**10**	1.9E+6	1.3E+6	1
TD.5.99	9301	*MO*	3338	9.1E+9	9.6E+9	**488**	1.8E+8	1.2E+8	1
CRAG.7.9	N/A	N/A	302	4.7E+6	5.3E+6	**58**	1.0E+6	5.9E+5	1
CRAG.8.9	N/A	N/A	3137	4.2E+7	4.8E+7	**546**	9.5E+6	5.3E+6	1
CRAG.9.9	N/A	N/A	*TO*	–	–	**4980**	4.3E+7	4.8E+7	1
LGHC.4.30	N/A	N/A	27	6.5E+5	8.0E+5	**9**	1.4E+5	4.4E+4	24
LGHC.5.30	N/A	N/A	701	1.7E+7	2.1E+7	**231**	4.1E+6	1.2E+5	240
LGHC.6.30	N/A	N/A	19040	4.3E+8	5.5E+8	**17263**	1.1E+8	3.2E+7	2160

Table 6.2: Results for Gamut and other games solved by ConGa and by Gambit

In Table 6.2, we have measured the time needed to generate the normal form, then the time required for the game to be solved by Gambit. For CG-Enum-Naive and CG-Enum-PNE, we

have measured the time needed to solve an instance, the number of candidate profiles and the number of deviation checks performed when checking whether a candidate is a PNE. Finally, the number of PNE in each game instance is depicted in the last column.

As we can see in the table, the size of normal form rapidly exceeds the capacity of Gambit. On the other side, in spite of its simplicity, the naive algorithm on constraint games is already faster than Gambit in most of the instances. From a simple reasoning, the number of candidates for CG-Enum-Naive is simply $|D^{V_c}|$, and the number of checks is comprised between the number of candidates and an upper bound of $|D^{V_c}| \times |\mathscr{P}|$. For games without hard constraints, the naive algorithm simply enumerates all strategy profiles in the whole search space, then checks each tuple against the equilibrium condition. That is why the entries in the column *#Tuple* in Table 6.1 and in the column *#Cand* of CG-Enum-Naive (in Table 6.2) are identical. We recall that the games with hard constraints are not implementable in Gambit, indicated by N/A for *not applicable*. The naive algorithm runs better for these games since it does not require to check all strategy profiles in the whole search space. In other words, this algorithm only tests tuples in the feasible search space restricted by hard constraints.

Not surprisingly, CG-Enum-PNE prunes most of the time a large part of the search space, mainly thanks to NBR detection. But interestingly, most of the time, it also saves deviation checks, meaning that the solution is found in the tables before a check is performed. A notable counterexample is the Minimum Effort Game with a domain of size 2 (MEG.30.2 and MEG.35.2) for which neither the tables nor the NBR detection are working because the domains are too small. In all other benchmarks, CG-Enum-PNE outperforms both Gambit and CG-Enum-Naive by one order of magnitude or even more.

A potential problem of CG-Enum-PNE could be that the size of the tables grows too much. It is easy to build an example for which the first player will get a table of exponential size: the game with no constraint for each player. In this game, each profile is a PNE and is thus stored in the table of the first player. However, this behavior has not been observed in practice in any of our examples. Tables rather stay of reasonable size, either because they belong to lower level players and they are often reset or because many profiles do not reach high level players.

Next, we evaluate the performance of CG-Enum-PPNE over CG-Enum-PNE. Clearly, the existence of PPNE depends on the PNE existence. For the games in which there does not contain a PNE or they have only one PNE, it is worthless performing algorithms for enumerating all PPNE. That is why, among the examples, we compare the two algorithms - CG-Enum-PNE and CG-Enum-PPNE - on only Location Games with Hard Constraints (Example 4.12) due to the number of PNE in each game instance (see Table 6.2). The time taken of these algorithms is depicted in Figure 6.4 (on the next page).

The unique difference between the two algorithms is an additional checking process for the Pareto dominance after having partially detected the PNE set. Thus, the additional time taken of CG-Enum-PPNE depends on the size of the existing PNE set. In Figure 6.4, the two curves almost

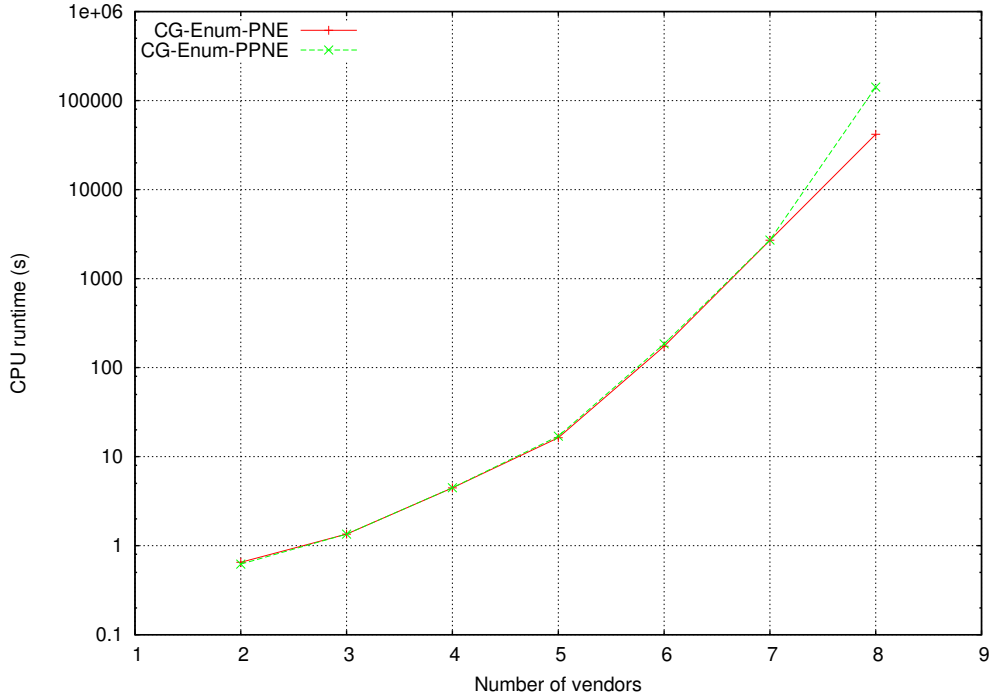


Figure 6.4: Time taken of CG-Enum-PNE and CG-Enum-PPNE on Location Games with Hard Constraints. The street length of all instances is set to 20.

overlap in most of the instances. This suggests that not much time is required for checking Pareto dominance, also means that the existing PNE set is fairly small during search. The additional time is more clearly noticeable on the last instance consisting of 8 vendors because the PNE set significantly grows up in this case.

6.6 Related Work

The generic algorithm for computing all PNE on normal form is the "generate and test" algorithm (Algorithm 2.1 in Section 2.2.1). Although this algorithm is quite trivial and inefficient, it is still the baseline algorithm. In addition, it is the unique algorithm implemented in Gambit [McKelvey et al., 2014] for enumerating all PNE in general normal form games.

In [Gottlob et al., 2005], the authors focus on theoretical results on determining and computing PNE in graphical games rather than PNE enumeration algorithms. They also use constraints network for modeling the best responses set of each player. Then the solution obtained is a PNE. However, the number of constraints is huge and it is not natural for encoding games. There have not existed efficient algorithms for enumerating all PNE in general graphical games yet. However, Daskalakis and Papadimitriou succeed in determining several tractable classes in which PNE can be computed in polynomial time. In [Daskalakis and Papadimitriou, 2006], they map from any graphical game to a Markov random field (abbreviated MRF) in order to inherit the problem of finding a maximum a posteriori configuration of the MRF [Lauritzen, 1996]) for de-

termining the existence of PNE in graphical games. Then, they apply the junction tree algorithm [Jensen et al., 1990] to derive polynomial time algorithms for checking the existence of PNE and for computing all PNE for several classes of graphical games. In this paper, they demonstrate that in the following classes, deciding whether a game has a PNE is in P when computing all PNE can be done in polynomial time.

- Graphical games where the hypergraph of the game is acyclic
- Graphical games with bounded treewidth
- Graphical games with bounded hypertreewidth
- Graphical games with $\Theta(\log n)$ treewidth, bounded number of strategies and bounded neighborhood size
- Graphical games with primal graphs of treewidth $\Theta(\log n)$ and bounded cardinality strategy sets

Recently, [Chapman et al., 2010] propose an algorithm called Valued Nash Propagation which integrates the optimization problem of maximizing a criterion with the constraint satisfaction problem of finding PNE in graphical games. However, this algorithm is limited on only tree graphs, not general graphs.

Like in graphical games, determining whether an action-graph game (abbreviated AGG) has a PNE in NP-complete in general. Similarly, there have not existed efficient algorithms for computing all PNE in general AGG. However, in some classes of AGG, there exist polynomial time algorithms. In [Jiang and Leyton-Brown, 2007], the authors use dynamic programming approach for computing all PNE in such tractable classes. They henceforth demonstrate that if the game is symmetric and the action graph has bounded treewidth, their algorithm determines the existence of pure Nash equilibrium and computes these equilibrium in polynomial time.

In [de Clercq et al., 2014a], the authors present a method for computing PNE as well as Pareto optimal equilibrium and core elements for boolean games that is based on disjunctive answer set programming [Brewka et al., 2011]. They use answer sets as strategy profiles and then create a programming with three parts: *(i) a part representing strategy profiles and verifying the players' satisfaction by their goals, (ii) a part describing possible strategies for all players, and (iii) a part checking the deviation of every player and selecting PNE by saturation techniques [Baral, 2003].*

6.7 Conclusion

In this chapter, we have presented a complete solver integrated into ConGa for constraint games. The main contributions are a couple of advanced techniques based on a fast computation of Nash consistency and pruning of *never best responses* along with the algorithm CG-Enum-PNE. The experimental results demonstrate a significant improvement of our complete solver over the Gambit solver.

Finding PNE in constraint games is computationally complex because determining whether a constraint game has at least a PNE is Σ_2^p-complete (Theorem 4.1 in Section 4.1). It is hence very interesting to determine tractable classes of constraint games in which all PNE can be computed in polynomial time.

We have also presented a naive algorithm based on CG-Enum-PNE for enumerating all pure Pareto Nash equilibrium. Besides the pruning techniques already implemented in the CG-Enum-PNE algorithm, we could also use the Pareto constraint as global constraint attached to hard constraints in constraint games for pruning larger search space. This kind of constraint has been explored in constraint programming for solving bi-objective combinatorial optimization problems exactly [Hartert and Schaus, 2014]. Similarly, by using Pareto constraint in constraint games, the dominated tuples can be eliminated from the search space. Nevertheless, the difference is that, in constraint games, a tuple is Pareto dominated by a PNE, not by any tuple in general. Hence, the Pareto global constraints are only added into the solver after having already detected pure Nash equilibrium. That raises another problem in constraint games that is the use of heuristics for finding PNE in short time. In addition, the heuristic search must be complete as well. We therefore concentrate our interest on heuristic search for constraint games that will be presented in the following chapter (Chapter 7).

As mentioned earlier, finding all pure Nash equilibrium in games is a difficult task. In consequence, there are still not many techniques as well as solving tools addressing this problem. By proposing several techniques and the complete solver in this chapter, we hope that they will open directions for further investigations on efficient algorithmic techniques to compute pure Nash equilibrium in game theory.

Chapter 7

Heuristic Search for Constraint Games

Contents

7.1 Deviation-based Heuristic for Variable and Value Ordering 109

7.2 Heuristic Algorithm 112

7.3 Experiment 117

7.4 Related Work 120

7.5 Conclusion 121

In many cases, an algorithm to compute one PNE might be more desirable than an enumeration algorithm. The local search solver described in Chapter 5 addresses these cases. Not surprisingly, large games can be solved but with no guarantee of finding an equilibrium or prove its absence. On the other side, the complete solver with the static heuristic designed in Chapter 6 can be modified to detect only the first PNE and of course, it allows to prove the absence of PNE. However, the enumeration algorithms in the complete solver work as the exhaustive search and become sometimes impractical, especially for large games. There is thus a need of a dynamic heuristic method that can be used to speed up the process of finding the first PNE. Inspired by the analogue with the efficiency of dynamic heuristics in Constraint Programming, in this chapter, we focus our interest on heuristic search for constraint games that allows to pick the first PNE within reasonable time as well as prove that there does not exist a PNE.

7.1 Deviation-based Heuristic for Variable and Value Ordering

Dynamic heuristics for backtracking search have been largely studied in constraint programming. Actually, there are two main kinds of heuristic: *(i) for variables and (ii) for values*. In constraint programming, a variable/value ordering heuristic is a function that guides the solver toward a solution. When the solver has to make a choice for the next branching, a heuristic provides an ordering of the variables/values from the best to the worst where the best candidate owns the

highest probability for solving the problem, i.e lead to a solution. On the other side, to the best of our knowledge, heuristics have been rarely used for computing Nash equilibrium in games.

Our aim, with the proposal of a dynamic heuristic for variable and value ordering in constraint games, is to explore the search tree in order to find a PNE as fast as possible, i.e. visiting as few nodes as possible.

Deviation-based Heuristic. An assignment is said to be a solution in CSP if it satisfies all constraints simultaneously. Many dynamic heuristics have been proposed to find a solution or all solutions in constraint programming. These heuristics are often based on the capacity of satisfying/violating the constraints of the current partial assignment. In constraint games, the situation has changed. Because of the interaction between players, being a solution of a player or many players' goals does not assure to be a PNE. Henceforth, the classical heuristics proposed in constraint programming cannot work in constraint games anymore. It is thus required a novel view for dealing with heuristics in constraint games instead.

The first question has to be solved is: "What is a *good* strategy in constraint games?". Like in constraint programming, a good strategy allows the solver to lead to a solution, i.e. a PNE in constraint games. That is to say, this strategy should guide the solver to a PNE. As defined earlier, given a strategy profile s, $s = (s_i, s_{-i})$ is a PNE if and only if *s is a best response of all players*. Thus, s_i may be a good strategy of player i if $s = (s_i, s_{-i})$ is his best response.

PNE can be also defined as follows. A strategy profile is a PNE if no player has an incentive to deviate while *the others keep their strategies unchanged*. Thus, the strategy s_i could be better when it is not only in a best response of player i but also a part of other players' best response. In other words, (s_i, s_{-i}) is also a best response of player $j \neq i$.

Based on the analysis above, we propose a deviation-based heuristic that could drive the solver to choose good strategies during search. Before presenting this heuristic, we first define a notion called *evaluation function of a strategy* as follows.

Definition 7.1 (Evaluation function of a strategy)**.** *Given s_i a strategy of player i. Let A_{s_i} be a set composed of all strategy profiles $s = (s_i, s_{-i})$ such that each strategy profile s is a best response of player i. $\forall s \in A_{s_i}$, let $\phi(s)$ be the number of players for whom s is a best response. We denote by $eval(s_i)$ the evaluation function of s_i, then $eval(s_i) = \sum_{s \in A_{s_i}} \phi(s)$.*

Let us show how to determine the evaluation function of a strategy in the following example.

Example 7.1. *We consider the following CSG: the set of players is $\mathscr{P} = \{X, Y, Z\}$. Each player owns one variable: $V_X = \{x\}, V_Y = \{y\}$ and $V_Z = \{z\}$ with $D_x = D_y = \{1,2\}, D_z = \{1,2,3,4\}$. The goals are $G_X = \{x \geq y, x < z\}$, $G_Y = \{x \leq y, y \neq z\}$ and $G_Z = \{x + y = z\}$.*

We will determine the evaluation function of a strategy of player Z in Example 7.1, let say, $z = 3$. There are 4 strategy profiles involving in this strategy, i.e. $(1,1,3), (1,2,3), (2,1,3), (2,2,3)$ (the tuple $(1,1,3)$ stands for $x = 1, y = 1, z = 3$, it is the same for the other tuples). Among them, only $(1,2,3)$ and $(2,1,3)$ are the best responses of player Z, thus $A_{(z=3)} = \{(1,2,3), (2,1,3)\}$.

$z=3$		Players		
		Z	Y	X
Tuples	(1,2,3)	+1	+1	0
	(2,1,3)	+1	0	+1

Table 7.1: Calculating the evaluation function of strategy $z=3$ of player Z in Example 7.1.

Table 7.1 shows how to determine the evaluation function of the strategy. We see that $eval(z=3)=\phi((1,2,3))+\phi((2,1,3))=2+2=4$. We have $\phi((1,2,3))=2$ because $(1,2,3)$ is a best response of player Z and Y and $\phi((2,1,3))=2$ because $(2,1,3)$ is a best response of player Z and X.

Let θ be a strategy in constraint games. At the beginning, $eval(\theta)$ is set to zero. During search, $eval(\theta)$ will be increased progressively. The deviation-based heuristic dynamically sorts the strategies from the best to the worst according to their function evaluation. Because each strategy of a player is composed of the values assigned to his decision variables, we compute and update scores for all values during search. Given a strategy s_i of player i, when $eval(s_i)$ is increased by an amount ε, the score of each value $v \in s_i$ will gain an amount ε. All values with their scores are stored in a table called *Heuristic Table* (abbreviated HTB) (see Figure 7.1).

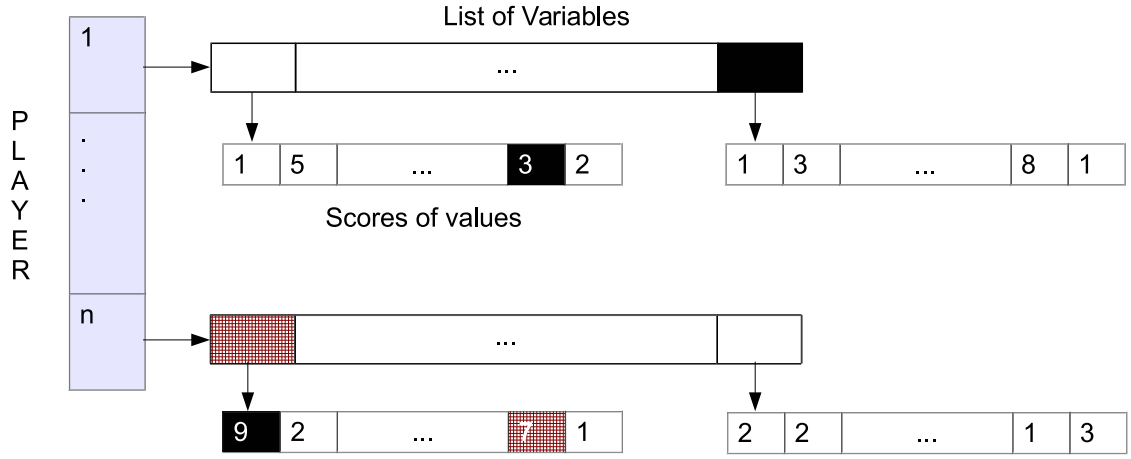


Figure 7.1: A situation of the heuristic table

Figure 7.1 also illustrates how the heuristic chooses variables and values during search. The black cells indicate the forbidden choices. Namely, the variables have been already instantiated or the values are not in the actual domain of their variables. Among the allowed variables and values in Figure 7.1, the maximal score is 7. Our heuristic chooses the pair (variable, value) whose score is maximal (in the red cross cells).

Restarts. In constraint programming, ordering heuristics for backtracking algorithms may make mistakes in some instances. These are cases in which small changes to variable or value heuristic can lead to great difference in running time. A technique called *randomization and restarts* has been proposed to correct these mistakes. The usual method for randomizing a backtracking

algorithm is to randomize the variable or value ordering heuristics according to a restart strategy $S = (t_1, t_2, t_3, \ldots)$. The idea is that a randomized backtracking heuristic is restarted at random and runs for t_1 steps. If no solution is found, the algorithm runs for t_2 step and so on. Because S is an infinite sequence and is generated in increasing, the randomized algorithm always gives the correct answer when it terminates.

In constraint games, we maintain the heuristic table recording scores of all values instead of using a function in the heuristic like in constraint programming. At the beginning, all scores are initially set to zero. Variables and values are thus selected fairly randomly. A situation may occur that we get stuck in a branch that does not lead to any PNE. Hence, restarts techniques need to be performed in order to escape this trap. In other words, we first launch the algorithm in order to collect enough information in the heuristic table. We henceforth restart the algorithm taking into account the scores recorded in the heuristic table for detecting a PNE. Another reason is that, like other heuristics, our heuristic may also make mistakes during search. Thus, the restart is used for fixing them. The restart strategy S is also used in constraint games. Many restart suites have been proposed but in this preliminary work, we only focus on the Luby suite with a geometrical factor of 2, given by $(1,1,2,1,1,2,4,\ldots)$ [Luby et al., 1993].

7.2 Heuristic Algorithm

We propose in this section a tree-search complete algorithm including the dynamic heuristic for variable and value ordering which has been presented previously. Overall, in this algorithm, we make the branching for variables and values according to the dynamic heuristic. This heuristic is done by recording the information involved in best responses in previous search steps. The algorithm stops as soon as a PNE is detected. Otherwise, we regularly perform the restarts according to the Luby suite in order to avoid getting stuck in the branches that do not contain any PNE.

Before starting the detailed description of the pseudo code, we first describe the following global variables that will have been using during search.

- **Global solver** S**.** The global solver is taken from the constraint game. It includes all variables along with all hard constraints (if they exist). Note that it does not contain any player's goal. This solver will be reinitialized when we restart the algorithm.

- **Luby suite** S_u**.** The Luby suite $S_u = \{1,1,2,1,1,2,4,...\}$ aims at determining the number of steps to restart where each element u in S_u is a coefficient.

- **Best responses tables.** We store the best responses of each player in a table (like in Algorithm CG-Enum-PNE in Section 6.3 of Chapter 6)

- **Heuristic table.** The heuristic table is used for storing scores of each value per variable during search. It is actually implemented by a 3-dimensional array whose initial state is specified in Figure 7.2.

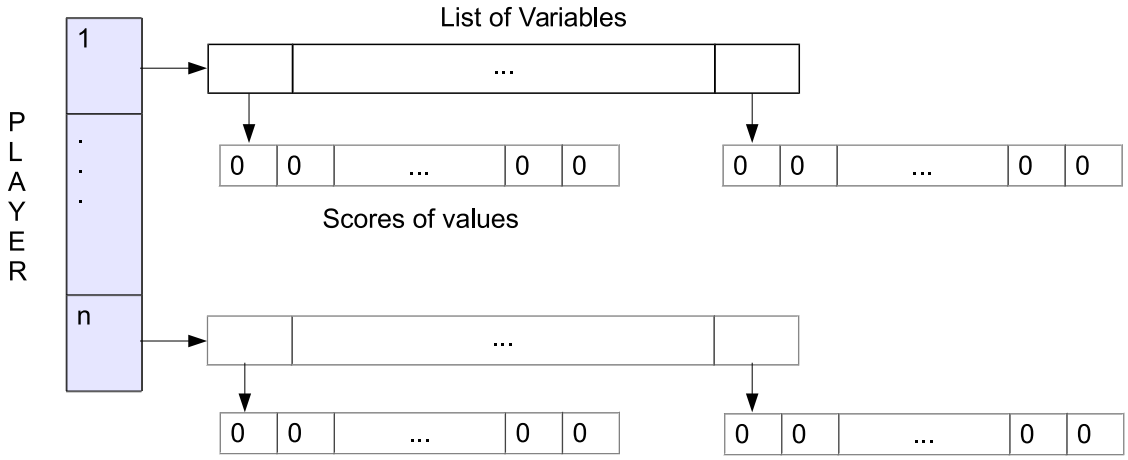


Figure 7.2: Initial state of the heuristic table

The space needed to stock the heuristic table is $|\mathscr{P}| + |V| + \sum_{v \in V} |D_v|$ while the time complexity to access and update a score is $\Theta(1)$.

Algorithm 7.1 specifies the main algorithm which runs until finding a PNE or proving its absence.

Algorithm 7.1. CG-Heuristic, the heuristic search algorithm to find the first PNE

```
1:  global variables:
2:      S: global solver
3:      S_u: Luby suite
4:      BR: best responses tables
5:      HTB: heuristic table

6:  function CG-HEURISTIC(Game CG): tuple
7:      stop ← false        ▷ stop is set to be true if the algorithm detects the first PNE or proves its absence
8:      nbRestart ← 1                                   ▷ nbRestart counts the number of restarts
9:      nbStep ← 0                                      ▷ nbStep counts the number of steps at each restart
10:     S ← CG.getSolver()                              ▷ get the initial global solver
11:     σ ← S_u[nbRestart] × η                          ▷ σ determines the maximal number of steps to restart
12:     L ← ∅                                           ▷ L states the list of variables already assigned
13:     while !stop do
14:         FirstPNE ← rSolve(L, σ, nbStep)
15:         if FirstPNE ≠ null then                     ▷ the first PNE is found
16:             stop ← true
17:         else
18:             if nbStep < σ then                      ▷ the game has no PNE
19:                 stop ← true
20:             else                                    ▷ restart
21:                 nbRestart++
22:                 nbStep ← 0
23:                 σ ← S_u[nbRestart] × η
24:                 S ← CG.getSolver()
25:             end if
26:         end if
27:     end while
28:     return FirstNE
29: end function
```

In Algorithm 7.1, at the beginning, we initialize the restart parameters in line 7 -12. We then perform the recursive algorithm (Algorithm 7.2) (line 14) for finding the first PNE. If it has not been found yet, two following possibilities will happen.

First, the maximal number of steps for each restart is determined by $\sigma = S_u[\text{nbRestart}] \times \eta$ where η is fixed according to each game instance. If the number of strategy profiles visited, i.e. number of steps, is less than σ, then the whole search space has been already wiped out. The game thus has no PNE (line 18 - 19).

nbRestart	$\mathbf{S_u}$**[nbRestart]**	η	σ
1	1	1000	1000
2	1	1000	1000
3	2	1000	2000

Table 7.2: Calculating the maximal number of steps (σ) for restarting in a constraint game

For example, given a constraint game $\mathscr{G}$ consisting of 1500 strategy profiles. We apply the CG-Heuristic algorithm in order to detect the first PNE of $\mathscr{G}$. We fix the number η to 1000. Table 7.2 specifies the number σ for each restart. We first let the algorithm run for 1000 steps (the algorithm verifies 1000 strategy profiles), no PNE is found. At the second restart, the algorithm is run for 1000 steps. No PNE is detected either. At the third restart, $\sigma = 2000$. After having visited 1500 strategy profiles, i.e. the whole search space of the game, we do not find a PNE ($nbStep < \sigma$ in this case). It means that there is no PNE in $\mathscr{G}$.

Second, we restart the algorithm by reinitializing the restart parameters (line 21 - 24). More precisely, the number of steps is reset to zero (line 22). In line 23, the new maximal number of steps σ is recalculated. And finally, the solver is initialized from the constraint game (line 24).

Algorithm 7.2 depicts the recursive algorithm of CG-Heuristic. The propagation function on hard constraints rules out all forbidden joint strategies in the game (line 2). If after having been propagated, the domain of one variable becomes empty, then all strategy profiles in the subspace do not satisfy the hard constraints. In other words, there is no PNE on this branch (line 3 - 5).

Otherwise, a variable and a value are selected by the heuristic for exploring a new search space (line 7). Because we need to restore the actual state of the global solver in the future, the current state is therefore saved in line 8 before the value is assigned to the variable (line 9). Since we perform here a dynamic heuristic for variable and value ordering, the list L is necessary to keep track of the variables already assigned. When a variable is assigned, it is pushed into L (line 10). When this variable is no longer assigned, it is popped from L (line 12).

We continue to recursively deepen the search tree (line 11). If a PNE is not found and the restart requirement is not met (line 13), we backtrack and explore the search space (line 14 - 18).

The domains of all variables contain only one value confirms that an entire strategy profile is reached (line 22 - 26). We first increase the steps counter (line 22), than we get this tuple to verify whether it is a PNE (line 23 - 26).

Algorithm 7.2. The recursive algorithm in the CG-Heuristic algorithm

```
1: function RSOLVE(list of variables L, int σ, int nbStep) returns tuple
2:     S.propagate()
3:     if ∃v ∈ V such that D(v) = ∅ then                ▷ there is no solution in this subspace
4:         return null
5:     end if
6:     if ∃v ∈ V such that |D(v)| ≥ 2 then
7:         (var,val) ← heuristic(L)                     ▷ heuristic is encoded in Algorithm 7.4
8:         S.worldPush()                ▷ save the current state of the global solver to restore later
9:         S.instantiate(var, val)        ▷ remove all values except val from the domain of var
10:        L.push(var)                                  ▷ push var into L
11:        tuple ← rSolve(L, σ, nbStep)
12:        L.pop(var)                                   ▷ pop var from L
13:        if (tuple = null) and (nbStep < σ ) then
14:            S.worldPop()                ▷ restore the previous state of the solver
15:            if |D(var)| ≥ 2 then
16:                S.remove(var, val)          ▷ remove val from the domain of var
17:                rSolve(L, σ, nbStep)
18:            end if
19:        end if
20:        return tuple
21:    else                                             ▷ reach to the entire strategy profile
22:        nbStep++
23:        if checkNash(D) then return D                ▷ return the first PNE found
24:        else
25:            return null
26:        end if
27:    end if
28: end function
```

Checking whether a tuple is a PNE is depicted in Algorithm 7.3 (This algorithm is similar to Algorithm 6.5 in Chapter 6.). Because our heuristic is based on deviation, we update the scores of values in the heuristic table in this algorithm.

The verification starts from the deepest level, i.e. from the last player (line 3). We first look up into the player's best responses table from the current tuple (line 4). If the player's best responses toward tuple t have not been computed yet (line 5), we than compute them by solving a Constraint Satisfaction Problem or Constraint Optimization Problem (line 6). The notation "*" is replaced either by S for CSG (Algorithm 6.6) or by O for COG (Algorithm 6.7).

In the previous chapter (Chapter 6), for making the advanced algorithm in Section 6.3 simpler to understand, we assume that each player controls only one variable. In this chapter, because we no longer keep this limitation while presenting the heuristic algorithm, the two algorithms, i.e. Algorithm 6.6 and Algorithm 6.7, need to be slightly adapted. The unique modification is line 4 in both the algorithms. In the current version, we assume that player j controls the variable x_j, thus we add the constraint $x_j = t_j$ for all $j \neq i$. In the adapted version for the heuristic algorithm, player j controls a set of variable V_j. Line 4 must be hence changed to "$\forall j \neq i, \forall x_{jk} \in V_j$, add the constraints $x_{jk} = t_{jk}$". Given that, all others are the same.

We note that in line 7, S_i is the initial strategy set of player i. In line 8, the best responses detected

Algorithm 7.3. Check whether a tuple is PNE in the CG-Heuristic algorithm

```
1:  function CHECKNASH(tuple t) returns bool
2:      ε ← 0                                          ▷ compute the score the values may gain
3:      for (i ← n; i ≥ 1; i ← i -1) do
4:          dev ← search_table(t, BR, i)
5:          if dev = ∅ then
6:              dev ← findBestResponses-C*G(t, i)
7:              if dev = ∅ then dev ← S_i end if
8:              insert_table(i, BR, dev)
9:          end if
10:         if t_i ∉ dev then
11:             for (j ← n; j > (n - ε) ; j ← j -1) do
12:                 for all v ∈ V_j do                 ▷ update the heuristic table
13:                     HTB[j][v][t_j|_v] += ε
14:                 end for
15:             end for
16:             return false
17:         else
18:             ε++
19:         end if
20:     end for
21:     return true
22: end function
```

by the previous algorithms (Algorithm 6.6 and Algorithm 6.7) are added into the player's table to reuse in the future. We use the same data structure for the tables like in Chapter 6, namely they are actually implemented by a set of trees. Henceforth, the search for deviation in the tables and the insert into the trees in $\Theta(|\mathscr{P}|)$ for the time complexity. Player i obviously deviates if the current tuple is not his best response (line 10). Otherwise, the variable ε is incremented by one point (line 18).

The variable ε expresses the number of players for whom the tuple t is a best response. We check the deviation from player n to player 1 and this verification stops as soon as a player makes a beneficial deviation (line 16). If t is not a PNE then the player at position n - ε can deviate. In other words, the players in range $[n-\varepsilon+1,n]$ do not make any beneficial deviation. Then $\forall j \in [n-\varepsilon+1,n], \forall v \in t_j$, the score of value v gains an amount ε (line 11 -15).

The evaluation process presented in Algorithm 7.3 is based on Definition 7.1 but we do not compute the number of all players for whom t can be a best response. In this algorithm, we immediately break from the verification loop when a player deviates (line 16). Although this evaluation is incomplete, it help us not to spend too much time on the costly deviation verification.

Finally, the heuristic function is described in Algorithm 7.4. We remind that this function has been previously illustrated in Figure 7.1.

Among the variables non assigned along with their current domain, Algorithm 7.4 returns the pair (var,val) whose score is maximal. During search, we also store an array called VP. This array specifies to which player a variable x belongs. We use this array to determine player p of the variable x in line 7. We recall that the time complexity to access to a score in the heuristic table is

only $\Theta(1)$. Let x be the variable such that $|D(x)|$ is maximal. Thus the complexity of the heuristic function in the worst case is $\Theta(|D(x)| \times |V \setminus L|)$.

Algorithm 7.4. Select the best pair (var,val) for next branching in the CG-Heuristic algorithm

```
1: function HEURISTIC(list of variables L) returns (var,val)
2:     max ← 0
3:     var ← a variable x_0 in V \ L
4:     val ← a value in D(x_0)
5:     for all x ∈ (V \ L) do
6:         for all v ∈ D(x) do
7:             p ← VP[x]
8:             if HTB[p][x][v] ≥ max then
9:                 max ← HTB[p][x][v]
10:                (var, val) ← (x, v)
11:            end if
12:        end for
13:    end for
14:    return (var,val)
15: end function
```

We presented in this section the algorithm including the heuristic proposed in Section 7.1. Furthermore, this algorithm is available to integrate other heuristics as well. In other words, it could be seen as a sample for using heuristics in constraint games. In order to adapt the algorithm to another heuristic, we need to modify the algorithm in spirit of two following questions.

(i) How to calculate the scores for variables and values in constraint games in order to record in the heuristic table? (that is what we have done in Algorithm 7.3.)

(ii) How to select the best pair (var,val) from the heuristic table for next branching? (that is what we have done in Algorithm 7.4.)

In conclusion, we believe that there is a wide range of opportunities for designing many efficient heuristics in constraint games.

7.3 Experiment

In this section, we show some experimental results on using heuristics in constraint games. We have built up a heuristic solver on the top of ConGa that includes the heuristic mentioned above. We compare the performance of this solver with the two other solvers in ConGa, i.e. the local search solver (Chapter 5) and the complete solver (Chapter 6). We modify the current version of the advanced algorithm CG-Enum-PNE (Algorithm 6.3) in the complete solver to stop as soon as the first PNE is detected. All the three solvers hence find only one PNE of games. We no longer make a comparison with Gambit because it is significantly slower than ConGa, as demonstrated in Section 6.5 in the previous chapter.

Hypercube Network Congestion Games

First of all, we compare the performance of the heuristic solver on Hypercube Network Congestion Games (abbreviated HNCG in Example 4.15) with the complete solver with a static heuristic for variable and value ordering.

The experimental result is given in Table 7.3. For the complete solver, we give the number of candidates (strategy profiles) needed to detect the first PNE. For the heuristic solver, we provide the number of restarts as well as the number of candidates visited to detect one PNE. The two solvers may report the PNE different but the existence of PNE is identical. This is demonstrated in the last column labeled by *PNE*. Time needed to compute an equilibrium is measured in seconds. The time out is set to 3600 seconds (1 hour). Finally, the number $aE+b$ is equal to $a \times 10^b$.

#Players	Complete Search		Heuristic Search			PNE
	#Cand	*Time*	*#Restart*	*#Cand*	*Time*	
4	1.4E+3	1.93	2	**5.1E+2**	**1.89**	Yes
5	7.9E+3	3.88	2	**5.5E+2**	**2.08**	Yes
6	1.6E+5	12.52	3	**1.1E+3**	**3.01**	Yes
7	3.3E+6	173.56	62	**9.3E+4**	**19.50**	Yes
8	4.3E+7	2481.86	2046	**5.5E+6**	**1231.11**	Yes

Table 7.3: Results for solving the 4-dimensional hypercube network congestion game CG-Enum-PNE (Algorithm 6.3, implemented in the complete solver in Chapter 6) and CG-Heuristic (Algorithm 7.1, implemented in the heuristic solver in Chapter 7) for computing only one PNE. The number η in the heuristic algorithm is fixed to 500.

According to the results depicted in Table 7.3, the heuristic solver works very well. It is more powerful than the complete solver, in terms of both the number of candidates and the running time. The deviation-based heuristic is thus useful to guide the solver toward a PNE.

Location Games with Hard Constraints

Hypercube Network Congestion Games belong to the class of congestion games. It has been proven in [Rosenthal, 1973] that every congestion game always has at least one PNE. We henceforth evaluate the performance of our heuristic solver on Location Games with Hard Constraints (abbreviated LGHC, in Example 4.12) in which some instances may have no PNE.

In the below tables, the name of game is followed by the number of players (or vendors) and the size of domain (or the number of customers). For example, LGHC.6.30 stands for the game with 6 vendors and 30 customers (or the street length is 30). We let our experiments run up to one day (or 24 hours or 86400 seconds). The notation "–" means we have no information (for example, in case of expiring the time-out). Finally, the notation *TO* stands for "Time out".

First, we evaluate the performance of the advanced algorithm with a static heuristic for variable and value ordering in the complete solver and the dynamic heuristic algorithm described in this chapter. The experimental results are depicted in Table 7.4.

Name	CG-Enum-PNE		CG-Heuristic				PNE
	#Cand	*Time*	η	*#Restart*	*#Cand*	*Time*	
LGHC.3.30	**4.5E+3**	**1.87**	1000	62	1.8E+5	5.47	No
LGHC.4.30	4.5E+4	5.02	300	6	**3.0E+3**	**2.54**	Yes
LGHC.5.30	7.3E+5	20.37	6000	6	**6.4E+4**	**9.48**	Yes
LGHC.6.30	1.4E+7	295.18	10000	6	**1.1E+5**	**15.60**	Yes
LGHC.7.30	**1.4E+9**	**66199.02**	–	–	–	TO	No

Table 7.4: Results for solving Location Games with Hard Constraints (Example 4.12) by CG-Enum-PNE (Algorithm 6.3, implemented in the complete solver in Chapter 6) and CG-Heuristic (Algorithm 7.1, implemented in the heuristic solver in Chapter 7) for picking only one PNE.

As we can observe in the table, for two instances without PNE (LGHC.3.30 and LGHC 7.30), the heuristic solver is slower than the complete solver. A reason is that while the complete solver includes the pruning of *never best responses*, the heuristic solver does not contain any pruning technique to restrict the search space [1]. In addition, by using restarts techniques, in games without PNE, it may visit some strategy profiles twice or even more. In contrast, on the remaining instances having at least one PNE, like HNCG, the heuristic solver works very well.

Name	Heuristic Solver	Local Search Solver						PNE
	CG-Heuristic	CG-IBR		CG-TS		CG-SA		
	Time	*avg*	*sdt*	*avg*	*sdt*	*avg*	*std*	
LGHC.3.30	5.47	–	–	–	–	–	–	No
LGHC.4.30	2.54	0.80	0.06	0.80	0.06	0.84	0.09	Yes
LGHC.5.30	9.48	1.04	0.16	1.10	0.27	0.95	0.04	Yes
LGHC.6.30	15.60	1.13	0.19	1.16	0.26	1.04	0.07	Yes
LGHC.7.30	TO	–	–	–	–	–	–	–

Table 7.5: Results for solving Location Games with Hard Constraints (Example 4.12) by CG-Heuristic (Algorithm 7.1, implemented in the heuristic solver in Chapter 7) and three algorithms (CG-IBR (Algorithm 5.1), CG-TS (Algorithm 5.5) and CG-SA (Algorithm 5.7)) in the local search solver (Chapter 5) for picking only one PNE. For CG-TS, the tabu length is set to be round of quarter of the number of players. For CG-SA, the cooling rate is set to 0.1. TI is calculated from the formula $e^{\frac{-|\mathscr{P}|}{TI}} = 0.2$.

It does not totally accurate to compare local search and complete search since they are absolutely different methods. However, in next experiments, we compare the performance of the heuristic solver with the three algorithms implemented in the local search solver, including:

(i) CG-IBR, the *iterated best responses* algorithm in constraint games

(ii) CG-TS, the *tabu search* algorithm in constraint games

(iii) CG-SA, the *simulated annealing* algorithm in constraint games

[1] Designing a pruning technique for computing PNE in games is computationally complex. Removing *strictly dominated strategy* or *never best responses* (NBR) whose time complexity in boolean games raises to Σ_2^p-complete [Bonzon et al., 2006] has been known as the unique pruning techniques in the literature. For constraint games, the unique technique that has been designed is pruning of NBR in subgames [Nguyen and Lallouet, 2014]. Even in subgames, the detecting of NBR is still difficult. In [Nguyen and Lallouet, 2014], a static heuristic is used for variable and value ordering. We can thus detect NBR in a lazy way by the counter for each player in subgames. Even though we may not detect all NBR, this detection is almost free. Unfortunately, in this chapter, we can no longer apply this technique of counter because the variable and value ordering is not static anymore. Investigating a pruning technique with dynamic heuristics is a challenge, we thus let it for future work.

Our aim is to provide a global view of all possible methods for solving constraint games.

In these experiments, each local search algorithm is launched 100 times per instance. The max step is set to 50×10^6. If an algorithm fails to find a PNE within the max step than the time taken will be noted by the notation "–". The abbreviations *avg, std* stand for average time (in seconds) and standard deviation. The experimental results are given in Table 7.5.

In this table, all the instances are too small to be solved by the local search solver. The results of the three algorithms are thus almost similar. It is not surprising that the local search solver is much faster than the heuristic solver. It has been well-known that local search is particularly designed for detecting a solution in very large problems. However, the dead point of local search is problems without solution because there is no way to prove this absence. On the other side, the heuristic algorithm can overcome this drawback, for example, on the instance LGHC.3.30 in the table. It is because the heuristic search is complete as well.

7.4 Related Work

Heuristics have been largely used in constraint programming and in game theory. However, there are a few works related to apply heuristics for computing Nash equilibrium in games. In addition, these works mainly focus on finding mixed Nash equilibrium. In this section, we concentrate on heuristic methods on normal form as well as other compact game representations (Section 2.3).

In [Porter et al., 2008], the authors apply the support-enumeration methods for finding a sample mixed Nash equilibrium in games. Instead of searching through all mixed strategy profiles, the support-enumeration methods search through support profiles and check whether there is an equilibrium with this particular support. The algorithm proposed by Porter et al. uses heuristics to order its exploration of the space of supports and this order can speed up equilibrium computing. Earlier, [Cartwright, 2003] applies an imitation dynamic for computing approximate Nash equilibrium. Nevertheless, they are rather dynamic models of learning than computing equilibrium in games.

There are few attempts of applying heuristics on graphical games. In [Vickrey and Koller, 2002], the authors propose a number of heuristics for finding approximate Nash equilibrium in graphical games based on a hill-climbing approach and on constraint satisfaction that essentially provide a junction tree approach for arbitrary graphical games. Following the same goal, a distributed, message-passing algorithm for computing approximate Nash equilibrium in graphical games is described in [Ortiz and Kearns, 2002]. In this algorithm, the authors apply a promising and effective heuristic in loopy graph.

On the other side, [Duong et al., 2009] apply heuristics for learning graphical games, not for computing equilibrium. Their goal is to seek algorithms that derive a normal form to a graphical game. In this paper, they present four algorithms for learning the graphical structure and payoff function of a game from data. For evaluating the algorithms' performance, they define three

metrics based on minimizing empirical loss. One of the algorithms is a greedy heuristic, along with a branch and bound algorithm and two local search algorithms. They demonstrate that when many profiles are observed, greedy is nearly optimal and considerably better than the other algorithms.

On action-graph games (AGG), [Jiang and Leyton-Brown, 2006] concentrate on computing expected payoffs under a mixed strategy profile. They integrate a simple heuristic to minimize the number of intermediate configurations in their algorithm which runs in polynomial time in the size of the AGG representation.

Solving boolean games has not been largely explored yet. The first heuristic approach using a stochastic and iterative algorithm on boolean games has been recently proposed in [de Clercq et al., 2014b]. Moreover, the authors also argue that for some classes of boolean games, this algorithm is guaranteed to converge to a PNE without checking the deviation condition. This heuristic could be seen as one of the first heuristic methods for computing PNE. However, as mentioned earlier, boolean games are not available to encode games with non-boolean utilities without using external tools. Our heuristic presented in this chapter thus may be considered as the first heuristic for speeding the computation of the first pure Nash equilibrium in general games [2].

7.5 Conclusion

The main contributions in this chapter are the deviation-based heuristic and the algorithm allowing to include heuristics for solving constraint games. Using the observation that local search solvers based on Iterated Best Responses are very efficient, we propose the heuristic based on deviation to evaluate the most promising candidate for branching. Together with the use of restarts, we show that this heuristic yields to interesting speed-ups to find the first equilibrium. Furthermore, we give here several ideas for improving our work.

In terms of storage, a potential problem of the heuristic algorithm is that each player's table for recording his best responses could grow too fast. It is because we only add new elements into the tables without removing them. It is thus necessary to study an effective method for frequently cleaning these tables. It is also required to use a more compressed data structure instead. A proposal is *Multi-valued Decision Diagram* (abbreviated MDD) [Miller and Drechsler, 1998, Miller and Drechsler, 2002] which is a compressed representation of sets or relations. In MDD, operations are directly performed on the compressed representation, i.e. without decompression. Furthermore, we need to apply efficient algorithms in MDD for adding new tuples or retrieving them in short time because these operations are frequently invoked in the heuristic algorithm.

[2] We remind that the term "general games" in this thesis demonstrates a set of games in which each element is an arbitrary static game, includes no collaboration between players and is complete information game. In addition, the players' utilities are not limited on boolean.

In terms of running time, on the one hand, the algorithm visits nodes in the search tree according to the dynamic heuristic, but on the other hand, it includes no pruning technique like in the complete search (Chapter 6). Henceforth, integrating several techniques such as pruning *never best response* would give it a better performance.

We have concentrated on examining the deviation-based heuristics. Another interesting direction could be the propagation-based heuristics. Moreover, we would even combine these two kinds of heuristics. Other lines of improvement could be using other restarts suite or deciding when we perform restarts technique during search.

In summary, it has been known that heuristics are not largely studied for finding Nash equilibrium in games. By proposing the heuristic along with the algorithm for constraint games in this chapter, we hope that they could motivate further research to discover efficient heuristics for computing Nash equilibrium in game theory.

Chapter 8

Conclusion and Perspective

Contents

8.1 Conclusion 123

8.2 Perspective 125

In this chapter, we summarize the contents and the contributions of the thesis by discussing the results along with some possible directions for future work.

8.1 Conclusion

The purpose of this thesis is to build a unified framework to model and to solve optimization problems in which multiple players partially compete for the same resources and have different goals (not mandatory disjoint). In particular, we focus our interest on static games which allow simultaneous moves, have complete information and include no collaboration. A static game is composed of a finite set of self-interested players. Each player takes a strategy from his strategy set to optimize his own utility. Because all the players choose their strategy simultaneously, they have not been informing the choices of the others. However, they know the utility functions of all the players, this game is thus classified into complete information games. Many examples can be cited as static games, for example, Rock-Paper-Scissors or Prisoners' dilemma.

In this thesis, our research interest includes the following directions.

- **Modeling.** The basic representation of games is an n-dimensional matrix called *normal form* whose size is exponential in the number of players. The intractability of this representation is a severe limitation to the widespread use of game-based modeling. There is thus a need to define a multiple players language for encoding games. This key issue has been addressed by several types of compact representations. An approach is the interaction-based representations from which two prominent representations are *graphical games* [Kearns et al., 2001, Kearns, 2007] and *action-graph games* [Bhat and Leyton-Brown, 2004, Jiang et al., 2011].

Another approach is the language-based modeling where the most well-known is *boolean games* [Harrenstein et al., 2001]. In a boolean game, each player controls a set of propositional variables while his preference is encoded by a set of propositional formula. The generic framework presented in this thesis, *constraint games*, can be considered as an extension of boolean games but provides a richer modeling language due to optimization and hard constraints. In a constraint game, each player's preference is expressed by a CSP. As proven by the compactness of constraint programming, for encoding games, this framework can be exponentially more compact than normal form. In addition, hard constraints can be provided to limit the joint strategies all players may take. This allows unrealistic equilibrium to be ruled out. Note that these constraints are global to all players, and they provide crucial expressivity in modeling.

- **Applications.** In this thesis, we succeed in testing our constraint games on both classical games and a large range of applications inspired by real-life problems. It has been demonstrated that constraint games are declarative and generic. Moreover, this game representation can express new, interesting problems taken from emergent fields, for example, applications in cloud computing.

- **Solving.** Despite the high interest of games for modeling strategic interaction, it is still difficult to find an efficient solver for computing Nash equilibrium in games. The normal form game solver Gambit [McKelvey et al., 2014] is currently considered as state-of-the-art. In this thesis, we have achieved to implement a solver named *ConGa* which allows to use multiple solving tools inside the same modeling environment, including local search (Chapter 5), complete search (Chapter 6) and heuristic search (Chapter 7).

Several parts of the work in this thesis have been validated by accepted papers in proceedings of international peer-reviewed conferences.

- Thi-Van-Anh Nguyen and Arnaud Lallouet : **A Complete Solver for Constraint Games**, *runner-up best student paper*, in CP'14, International Conference on Principles and Practice of Constraint Programming, Lyon, France, pages 58-74, September 8-12, 2014.
- Thi-Van-Anh Nguyen, Arnaud Lallouet and Lucas Bordeaux : **Constraint Games: Framework and Local Search Solver**, in ICTAI'13, International Conference on Tools with Artificial Intelligence, Washington DC, USA, pages 963-970, November 4-6, 2013.

The content of this thesis extensively elaborates upon previous published work and mistakes (if any) are corrected. Large sections of this work are previously unpublished, although they may appear in some papers in the future.

In conclusion, we believe that using constraint programming in game theory is an interesting proposal that would be worth studying in depth in the future. Additionally, we are confident that there is a wide range of opportunities for improvement, since this discipline has been until now largely unexplored.

8.2 Perspective

In this thesis, we focus on modeling and solving static games. In reality, along with simultaneous moves, many optimization problems also demand sequential moves between agents which can be modeled by dynamic games. For example, in the applications taken from cloud computing and network field (Example 4.16 and Example 4.14), it is natural to suggest that the provider would like to maximize their profit gained by all his clients. Therefore, there are two levels of making decisions in these applications. First, the provider fixes the price for his resource. Second, the clients minimize the cost that they have to pay to the provider according to this price. This scenario of bi-level decision-making, or even multiple levels, often appears in real-world economic problems, transportation or energy planning, etc.

Indeed, an interesting idea of future work is to mix static and dynamic games into a unified modeling and solving framework. This is what we call *Multilevel Constraint Games* (abbreviated MCG). Like constraint games, we will investigate this proposal in three directions: *(i) Modeling (ii) Applications and (iii) Solving*.

Modeling Framework

Quantified Constraint Satisfaction Problems (abbreviated QCSP) are an interesting extension of classical CSP where some variables may be universally quantified. QCSP model scenarios where uncertainty does not allow us to decide the values for some variables. Many research results related to QCSP have been developed. Among them, we have particular interest in *Quantified Constraint Optimization Problems* [Benedetti et al., 2008] which bring optimization into multi-level decision problems. This suggests an interesting idea of using constraint programming for encoding multilevel optimization games with multiple players that would be potential future work after this thesis.

Probably, the most well-known multilevel model in game theory is Stackelberg game [Stackelberg, 1952]. In this game, there are two actors: a *leader* and a *follower* who perform decisions sequentially but having no control on each other. The leader moves first to achieve his goal, henceforth, the follower takes decision to optimize his objective function according to the decision already made by the leader. The key idea here is that the leader and the follower have different objectives depending on each other, maybe in conflict. The original concept of Stackelberg was extended to an oligopolistic market with one leader and a group of followers by [Sherali et al., 1983].

Indeed, we plan to extend constraint games to multilevel constraint games. This new framework should be able to capture both two types of structures, i.e. *(i) "a leader and a follower" and (ii) "a leader and a group of followers"* where subgames among followers are encoded by constraint games.

Generally, the Stackelberg model can be solved to find the subgame Nash equilibrium, i.e. the strategy profile that serves best each player, given the strategies of the leader and that entails every follower playing in a Nash equilibrium in the subgame. Finally the leader chooses the strategies that offer him the optimum for his goal. This raises a solution concept called *Stackelberg Nash Equilibrium* which should be studied as the main solution concept of MCG.

The intuition meaning of the solution concept defined for MCG is as follows. Given a strategy chosen by the leader, the followers will optimize their outcome along with this strategy. In his turn, the leader reaches to his best in respect of equilibrium between all the followers. Therefore, Stackelberg Nash equilibrium assure a good balance between preferences of all the players in games.

Finally, the framework should be developed more deeply, to prove its properties, such as space complexity, existence of equilibrium and complexity of computing such equilibrium, etc.

Applications

Actually, many applications require a new language which allows multiple agents to express partially competitive objectives. In addition, there is always a need to define a modeling language for multilevel problems. Therefore, it is not difficult to demonstrate the usefulness of multilevel constraint games by real-world problems. Besides the extension of the examples already represented in this thesis (Example 4.14, Example 4.15 or Example 4.16), we also provide another example in order to illustrate a potential range of interesting applications that multilevel constraint games would capture to.

The following example is taken from aerial industry. The problem is to plan aircraft arriving at an airport [Soomer and Franx, 2008].

Example 8.1 (Arrival Aircraft Planning)**.** *Consider a set $\mathscr{F} = \{1, \ldots, n\}$ of airlines whose flights would land on p runways at the airport. Each airline $i \in \mathscr{F}$ has to schedule m_i flights. We assume that an aircraft is used for one and only one flight. Each flight/aircraft f_{ij} of airline i arrives in the neighborhood of the airport at a time t_{ij} and is scheduled by the airport authority to land on a runway at a time s_{ij}. If $s_{ij} > t_{ij}$, then the aircraft has to wait. Depending on its gas reserve, the aircraft f_{ij} is able to wait for a time r_{ij} but this is increasing its costs. Note that r_{ij} is constant and already determined for each flight. A runway can be used by aircraft f_{ij} for a landing time of ld_{ij}. Each airline company has control on the time t_{ij} to present an aircraft and is willing to suffer a minimum cost c_i for all its aircraft.*

This problem is illustrated in Figure 8.1. There are two runways labeled by R_1 and R_2 at the airport. At the moment t_1, an aircraft a_1 arrives to the airport. Because both the two runways are available, a_1 would land on whatever runway. The airport authority allows it to use R_1. This runway will be occupied by a_1 until the moment t_5. At the moment t_2, an aircraft a_2 arrives. Because R_1 is already occupied, so a_2 is commanded to land on R_2, for a period from t_2 to t_4. At the moment t_3, an aircraft a_3 would like to land on at the airport. Unfortunately, both the two

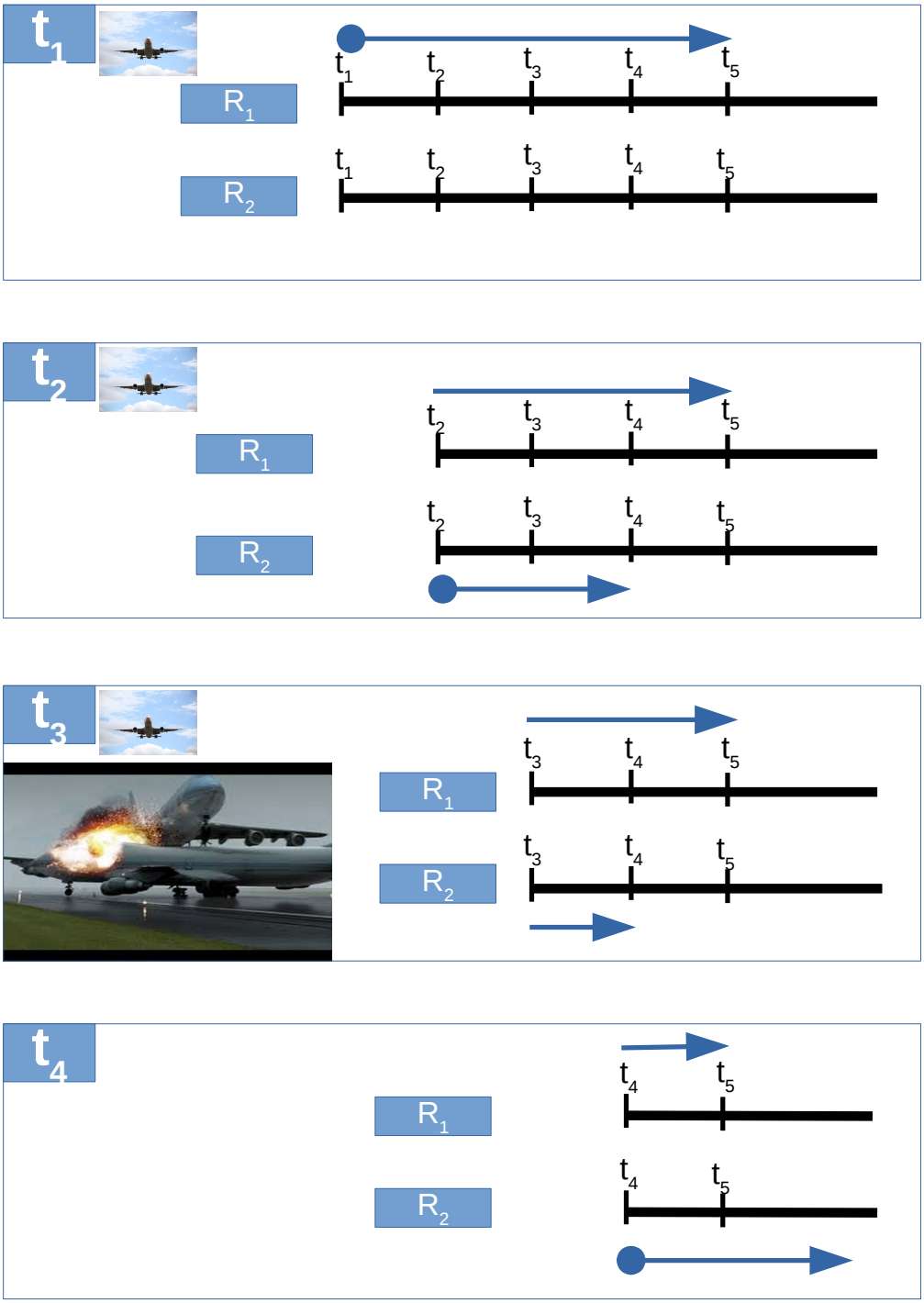


Figure 8.1: Arrival Aircraft Planning Problem

runways are not available to be used. If a_3 lands on whatever runway, two aircraft will collide. So a_3 has to wait until the moment t_4 when R_2 is reset to be free. Therefore, a_3 has to bear a delay time $t_4 - t_3$. In the classical problem, all aircraft belong to an airline. But here, the aircraft are controlled by different companies. So, the problem is how to find an equilibrium for the delay time of all companies.

The problem described in Example 8.1 is obviously a bi-level optimization problem in which the airport authority plays the role of leader and the airline companies as followers. Each company must wait for the authority's permit for landing while the authority need to assure a balance between all companies and to achieve the minimum delay for his best serve.

Solving Framework

Solving constraint games is computationally complex. In consequence, it is unlikely that solving multilevel constraint games is an easy task. But we think that it would be very interesting to investigate specialized techniques for computing Stackelberg Nash equilibrium in MCG. Another line of work is to try to define proper notions of approximation, both on the semantics side and on the solving side which could allow to drastically restrict the search space. In this perspective, local search will be studied. In summary, we are confident that an efficient solver for multilevel constraint games would be very appreciated in practice.

Bibliography

[Apt, 2004] Apt, K. R. (2004). Uniform proofs of order independence for various strategy elimination procedures. *CoRR*, cs.GT/0403024.

[Apt, 2011] Apt, K. R. (2011). A primer on strategic games. *CoRR*, abs/1102.0203.

[Apt et al., 2008] Apt, K. R., Rossi, F., and Venable, K. B. (2008). Comparing the notions of optimality in cp-nets, strategic games and soft constraints. *Ann. Math. Artif. Intell.*, 52(1):25–54.

[Aumann, 1974] Aumann, R. (1974). Subjectivity and correlation in randomized strategies. *Journal of Mathematical Economics*, (1):67–96.

[Aumann, 1959] Aumann, R. J. (1959). Acceptable points in general cooperative N-person games. In Luce, R. D. and Tucker, A. W., editors, *Contribution to the theory of game IV, Annals of Mathematical Study 40*, pages 287–324. University Press.

[Bacchus and van Run, 1995] Bacchus, F. and van Run, P. (1995). Dynamic variable ordering in csps. In *CP*, pages 258–275.

[Baral, 2003] Baral, C. (2003). *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA.

[Benedetti et al., 2008] Benedetti, M., Lallouet, A., and Vautard, J. (2008). Quantified constraint optimization. In Stuckey, P. J., editor, *CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 463–477. Springer.

[Bessière and Régin, 1996] Bessière, C. and Régin, J.-C. (1996). Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. In *CP*, pages 61–75.

[Bessière, 2006] Bessière, C. (2006). Constraint propagation. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 3. Elsevier.

[Bhat and Leyton-Brown, 2004] Bhat, N. A. R. and Leyton-Brown, K. (2004). Computing nash equilibria of action-graph games. In *UAI*, pages 35–42.

[Bonzon, 2007] Bonzon, E. (2007). *Modélisation des interactions entre agents rationnels : les jeux booléens*. PhD thesis, Université Paul Sabatier - Toulouse III, France.

[Bonzon et al., 2009a] Bonzon, E., Lagasquie-Schiex, M.-C., and Lang, J. (2009a). Dependencies between players in boolean games. *Int. J. Approx. Reasoning*, 50(6):899–914.

[Bonzon et al., 2012] Bonzon, E., Lagasquie-Schiex, M.-C., and Lang, J. (2012). Effectivity functions and efficient coalitions in boolean games. *Synthese*, 187(1):73–103.

[Bonzon et al., 2006] Bonzon, E., Lagasquie-Schiex, M.-C., Lang, J., and Zanuttini, B. (2006). Boolean games revisited. In Brewka, G., Coradeschi, S., Perini, A., and Traverso, P., editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 265–269. IOS Press.

[Bonzon et al., 2009b] Bonzon, E., Lagasquie-Schiex, M.-C., Lang, J., and Zanuttini, B. (2009b). Compact preference representation and boolean games. *Autonomous Agents and Multi-Agent Systems*, 18(1):1–35.

[Bordeaux and Monfroy, 2002] Bordeaux, L. and Monfroy, E. (2002). Beyond np: Arc-consistency for quantified constraints. In Hentenryck, P. V., editor, *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 371–386. Springer.

[Bordeaux and Pajot, 2004] Bordeaux, L. and Pajot, B. (2004). Computing equilibria using interval constraints. In Faltings, B., Petcu, A., Fages, F., and Rossi, F., editors, *CSCLP*, volume 3419 of *Lecture Notes in Computer Science*, pages 157–171. Springer.

[Bosse et al., 2010] Bosse, H., Byrka, J., and Markakis, E. (2010). New algorithms for approximate nash equilibria in bimatrix games. *Theor. Comput. Sci.*, 411(1):164–173.

[Bouhtou et al., 2007] Bouhtou, M., Erbs, G., and Minoux, M. (2007). Joint optimization of pricing and resource allocation in competitive telecommunications networks. *Networks*, 50(1):37–49.

[Boussemart et al., 2004] Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004). Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150.

[Boutilier et al., 2004] Boutilier, C., Brafman, R. I., Domshlak, C., Hoos, H. H., and Poole, D. (2004). Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Intell. Res. (JAIR)*, 21:135–191.

[Brandt et al., 2009] Brandt, F., Fischer, F. A., and Holzer, M. (2009). Symmetries and the complexity of pure nash equilibrium. *J. Comput. Syst. Sci.*, 75(3):163–177.

[Brélaz, 1979] Brélaz, D. (1979). New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256.

[Brewka et al., 2011] Brewka, G., Eiter, T., and Truszczynski, M. (2011). Answer set programming at a glance. *Commun. ACM*, 54(12):92–103.

[Brown and Jr., 1982] Brown, C. A. and Jr., P. W. P. (1982). An empirical comparison of backtracking algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(3):309–316.

[Brown et al., 2004] Brown, K. N., Little, J., Creed, P. J., and Freuder, E. C. (2004). Adversarial constraint satisfaction by game-tree search. In de Mántaras, R. L. and Saitta, L., editors, *ECAI*, pages 151–155. IOS Press.

[Bruynooghe, 1981] Bruynooghe, M. (1981). Solving combinatorial search problems by intelligent backtracking. *Inf. Process. Lett.*, 12(1):36–39.

[Bubelis, 1979] Bubelis, V. (1979). On equilibria in finite games. *International Journal of Game Theory*, 8(2):65–79.

[Cartwright, 2003] Cartwright, E. J. (2003). Learning to play approximate nash equilibria in games with many players. The warwick economics research paper series (twerps), University of Warwick, Department of Economics.

[Ceppi et al., 2010] Ceppi, S., Gatti, N., Patrini, G., and Rocco, M. (2010). Local search methods for finding a nash equilibrium in two-player games. In Huang, J. X., Ghorbani, A. A., Hacid, M.-S., and Yamaguchi, T., editors, *IAT*, pages 335–342. IEEE Computer Society Press.

[Chapman et al., 2010] Chapman, A. C., Farinelli, A., de Cote, E. M., Rogers, A., and Jennings, N. R. (2010). A distributed algorithm for optimising over pure strategy nash equilibria. In *AAAI*.

[Chen, 2000] Chen, X. (2000). *A Theoretical Comparison of Selected CSP Solving and Modeling Techniques*. PhD thesis, University of Alberta.

[Cheng et al., 2004] Cheng, S.-G., Reeves, D. M., Vorobeychik, Y., and Wellman, M. P. (2004). Notes on the Equilibria in Symmetric Games. In *International Joint Conference on Autonomous Agents & Multi Agent Systems, 6th Workshop On Game Theoretic And Decision Theoretic Agents*.

[Chien and Sinclair, 2011] Chien, S. and Sinclair, A. (2011). Convergence to approximate nash equilibria in congestion games. *Games and Economic Behavior*, 71(2):315–327.

[Choco Team, 2013] Choco Team (2008-2013). *Choco : An Open Source Java Constraint Programming Library*. Ecole des Mines de Nantes.

[Choco Team, 2010] Choco Team (2010). choco: an Open Source Java Constraint Programming Library. Research report 10-02-INFO, École des Mines de Nantes.

[Cohen and Jeavons, 2006] Cohen, D. and Jeavons, P. (2006). The complexity of constraint languages. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 8. Elsevier.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *STOC*, pages 151–158.

[Dasgupta and Maskin, 1989] Dasgupta, P. and Maskin, E. (1989). The existence of equilibrium in discontinuous economic games. I: Theory. Economic organizations as games, 49-81.

[Daskalakis, 2011] Daskalakis, C. (2011). On the complexity of approximating a nash equilibrium. In *SODA*, pages 1498–1517.

[Daskalakis et al., 2006] Daskalakis, C., Goldberg, P. W., and Papadimitriou, C. H. (2006). The complexity of computing a Nash equilibrium. In *Proceedings of the 38th annual ACM Symposium on Theory of Computing*, STOC '06, pages 71–78. ACM.

[Daskalakis et al., 2009a] Daskalakis, C., Mehta, A., and Papadimitriou, C. H. (2009a). A note on approximate nash equilibria. *Theor. Comput. Sci.*, 410(17):1581–1588.

[Daskalakis and Papadimitriou, 2006] Daskalakis, C. and Papadimitriou, C. H. (2006). Computing pure nash equilibria in graphical games via markov random fields. In *Proceedings 7th ACM Conference on Electronic Commerce (EC-2006), Ann Arbor, Michigan, USA, June 11-15, 2006*, pages 91–99.

[Daskalakis et al., 2009b] Daskalakis, C., Schoenebeck, G., Valiant, G., and Valiant, P. (2009b). On the complexity of nash equilibria of action-graph games. In *SODA*, pages 710–719.

[Davenport et al., 1994] Davenport, A. J., Tsang, E. P. K., Wang, C. J., and Zhu, K. (1994). Genet: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *AAAI*, pages 325–330.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397.

[de Clercq et al., 2014a] de Clercq, S., Bauters, K., Schockaert, S., Cock, M. D., and Nowé, A. (2014a). Using answer set programming for solving boolean games. In *KR*.

[de Clercq et al., 2014b] de Clercq, S., Bauters, K., Schockaert, S., Mihaylov, M., Cock, M. D., and Nowé, A. (2014b). Decentralized computation of pareto optimal pure nash equilibria of boolean games with privacy concerns. In *ICAART 2014 - Proceedings of the 6th International Conference on Agents and Artificial Intelligence, Volume 2, ESEO, Angers, Loire Valley, France, 6-8 March, 2014*, pages 50–59.

[de Vos and Vermeir, 1999] de Vos, M. and Vermeir, D. (1999). Choice logic programs and nash equilibria in strategic games. In Flum, J. and Rodríguez-Artalejo, M., editors, *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 266–276. Springer.

[Dechter, 1990] Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312.

[Dechter, 2006] Dechter, R. (2006). Tractable structures for constraint satisfaction problems. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 7. Elsevier.

[Dorigo and Stützle, 2004] Dorigo, M. and Stützle, T. (2004). *Ant colony optimization*. MIT Press.

[Driessen, 1988] Driessen, T. (1988). *Cooperative Games, Solutions and Applications*. Kluwer Academic Publishers.

[Duersch et al., 2012] Duersch, P., Oechssler, J., and Schipper, B. C. (2012). Pure strategy equilibria in symmetric two-player zero-sum games. *Int. J. Game Theory*, 41(3):553–564.

[Dunne and van der Hoek, 2004] Dunne, P. E. and van der Hoek, W. (2004). Representation and complexity in boolean games. In Alferes, J. J. and Leite, J. A., editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 347–359. Springer.

[Dunne et al., 2008] Dunne, P. E., van der Hoek, W., Kraus, S., and Wooldridge, M. (2008). Cooperative boolean games. In Padgham, L., Parkes, D. C., Müller, J. P., and Parsons, S., editors, *AAMAS (2)*, pages 1015–1022. IFAAMAS.

[Dunne and Wooldridge, 2012] Dunne, P. E. and Wooldridge, M. (2012). Towards tractable boolean games. In van der Hoek, W., Padgham, L., Conitzer, V., and Winikoff, M., editors, *AAMAS*, pages 939–946. IFAAMAS.

[Duong et al., 2009] Duong, Q., Vorobeychik, Y., Singh, S. P., and Wellman, M. P. (2009). Learning graphical game models. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 116–121.

[Elkind et al., 2006] Elkind, E., Goldberg, L. A., and Goldberg, P. W. (2006). Nash equilibria in graphical games on trees revisited. In *EC*, pages 100–109.

[Emerson, 1990] Emerson, E. A. (1990). Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072.

[Fabrikant et al., 2004] Fabrikant, A., Papadimitriou, C. H., and Talwar, K. (2004). The complexity of pure nash equilibria. In *STOC*, pages 604–612.

[Faltings, 2006] Faltings, B. (2006). *Distributed Constraint Programming*, chapter 20, pages 699–729. Handbook of Constraint Programming. Elsevier.

[Foo et al., 2004] Foo, N. Y., Meyer, T., and Brewka, G. (2004). Lpod answer sets and nash equilibria. In Maher, M. J., editor, *ASIAN*, volume 3321 of *Lecture Notes in Computer Science*, pages 343–351. Springer.

[Freuder, 1982] Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *J. ACM*, 29(1):24–32.

[Frost and Dechter, 1995] Frost, D. and Dechter, R. (1995). Look-ahead value ordering for constraint satisfaction problems. In *IJCAI (1)*, pages 572–578.

[Fudenberg and Tirole, 1991] Fudenberg, D. and Tirole, J. (1991). *Game Theory*. The MIT Press.

[Gaschnig, 1979] Gaschnig, J. (1979). *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University.

[Gasior and Drwal, 2013] Gasior, D. and Drwal, M. (2013). Pareto-optimal nash equilibrium in capacity allocation game for self-managed networks. *Computer Networks*, 57(14):2817–2832.

[Gatti et al., 2012] Gatti, N., Patrini, G., Rocco, M., and Sandholm, T. (2012). Combining local search techniques and path following for bimatrix games. In de Freitas, N. and Murphy, K. P., editors, *UAI*, pages 286–295. AUAI Press.

[Gatti et al., 2013] Gatti, N., Rocco, M., and Sandholm, T. (2013). On the verification and computation of strong nash equilibrium. In *AAMAS*, pages 723–730.

[Gent et al., 1996] Gent, I. P., MacIntyre, E., Prosser, P., Smith, B. M., and Walsh, T. (1996). An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *CP*, pages 179–193.

[Gilboa et al., 1990] Gilboa, I., Kalai, E., and Zemel, E. (1990). On the order of eliminating dominated strategies. *Operations Research Letters*, 9(2):85 – 89.

[Ginsberg et al., 1990] Ginsberg, M. L., Frank, M., Halpin, M. P., and Torrance, M. C. (1990). Search lessons learned from crossword puzzles. In *AAAI*, pages 210–215.

[Glover, 1989] Glover, F. (1989). Tabu search - part i. *INFORMS Journal on Computing*, 1(3):190–206.

[Golomb and Baumert, 1965] Golomb, S. W. and Baumert, L. D. (1965). Backtrack programming. *J. ACM*, 12(4):516–524.

[Gottlob et al., 2005] Gottlob, G., Greco, G., and Scarcello, F. (2005). Pure nash equilibria: Hard and easy games. *J. Artif. Intell. Res. (JAIR)*, 24:357–406.

[Grant et al., 2010] Grant, J., Kraus, S., and Wooldridge, M. (2010). Intentions in equilibrium. In *AAAI*.

[Grant et al., 2014] Grant, J., Kraus, S., Wooldridge, M., and Zuckerman, I. (2014). Manipulating games by sharing information. *Studia Logica*, 102(2):267–295.

[Grubshtein and Meisels, 2012] Grubshtein, A. and Meisels, A. (2012). Finding a nash equilibrium by asynchronous backtracking. In *CP*, pages 925–940.

[Gutierrez et al., 2013] Gutierrez, J., Harrenstein, P., and Wooldridge, M. (2013). Iterated boolean games. In *IJCAI*.

[Hao and Dorne, 1994] Hao, J.-K. and Dorne, R. (1994). A new population-based method for satisfiability problems. In *ECAI*, pages 135–139.

[Haralick and Elliott, 1980] Haralick, R. M. and Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313.

[Harrenstein et al., 2001] Harrenstein, P., van der Hoek, W., Meyer, J.-J. C., and Witteveen, C. (2001). Boolean Games. In van Benthem, J., editor, *TARK*. Morgan Kaufmann.

[Harsanyi, 1968] Harsanyi, J. C. (1968). Games with incomplete information played by 'bayesian' players, part iii. the basic probability distribution of the game. *Management Science*, 14(7):486–502.

[Hartert and Schaus, 2014] Hartert, R. and Schaus, P. (2014). A support-based algorithm for the bi-objective pareto constraint. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2674–2679.

[Haurie and Krawczyk, 2000] Haurie, A. and Krawczyk, J. (2000). *An introduction to dynamic games*.

[Hayrapetyan et al., 2006] Hayrapetyan, A., Tardos, É., and Wexler, T. (2006). The effect of collusion in congestion games. In *STOC*, pages 89–98.

[Hémon et al., 2008] Hémon, S., de Rougemont, M., and Santha, M. (2008). Approximate nash equilibria for multi-player games. In *SAGT*, pages 267–278.

[Hoos and Stützle, 2004] Hoos, H. H. and Stützle, T. (2004). *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann.

[Hoos and Tsang, 2006] Hoos, H. H. and Tsang, E. (2006). Local search methods. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 5. Elsevier.

[Hotelling, 1929] Hotelling, H. (1929). Stability in competition. *Economic Journal*, pages 41–57.

[Ismail et al., 2007] Ismail, I. A., Ramly, N. A. E., Kafrawy, M. M. E., and Nasef, M. M. (2007). Game theory using genetic algorithms. In Ao, S. I., Gelman, L., Hukins, D. W. L., Hunter, A., and Korsunsky, A. M., editors, *World Congress on Engineering*, Lecture Notes in Engineering and Computer Science, pages 61–64. Newswood Limited.

[Jalaparti et al., 2010] Jalaparti, V., Nguyen, G., Gupta, I., and Caesar, M. (2010). Cloud resource allocation games. Technical report, University of Illinois at Urbana-Champaign.

[Jensen et al., 1990] Jensen, F., Lauritzen, S., and Olesen, K. (1990). Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly*, 4:269–282.

[Jiang and Leyton-Brown, 2006] Jiang, A. X. and Leyton-Brown, K. (2006). A polynomial-time algorithm for action graph games. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 679–684.

[Jiang and Leyton-Brown, 2007] Jiang, A. X. and Leyton-Brown, K. (2007). Computing pure nash equilibria in symmetric action graph games. In *AAAI*, pages 79–85.

[Jiang and Leyton-Brown, 2010] Jiang, A. X. and Leyton-Brown, K. (2010). Bayesian action-graph games. In *NIPS*, pages 991–999.

[Jiang et al., 2011] Jiang, A. X., Leyton-Brown, K., and Bhat, N. A. R. (2011). Action-graph games. *Games and Economic Behavior*, 71(1):141–173.

[Jiang, 2011] Jiang, X. (2011). *Representing and Reasoning with Large Games*. PhD thesis, University of British Columbia, Canada.

[Johnson et al., 1989] Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C. (1989). Optimization by simulated annealing: An experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892.

[Kearns, 2007] Kearns, M. (2007). *Graphical Games*, chapter 7, pages 159–209. Algorithmic game theory. Cambridge University Press.

[Kearns et al., 2001] Kearns, M. J., Littman, M. L., and Singh, S. P. (2001). Graphical models for game theory. In Breese, J. S. and Koller, D., editors, *UAI*, pages 253–260. Morgan Kaufmann.

[Kirkpatrick et al., 1983] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220:671–680.

[Kollias and Roughgarden, 2011] Kollias, K. and Roughgarden, T. (2011). Restoring pure equilibria to weighted congestion games. In *ICALP (2)*, pages 539–551.

[Koutsoupias and Papadimitriou, 2009] Koutsoupias, E. and Papadimitriou, C. H. (2009). Worst-case equilibria. *Computer Science Review*, 3(2):65–69.

[Kreps and Wilson, 1982] Kreps, D. M. and Wilson, R. (1982). Sequential Equilibria. *Econometrica*, 50(4):863–894.

[Lau, 1999] Lau, T. (1999). *Guided Genetic Algorithm*. PhD thesis, University of Essex, Colchester, UK.

[Lauriere, 1978] Lauriere, J.-L. (1978). A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29 – 127.

[Lauritzen, 1996] Lauritzen, S. L. (1996). Temporal and modal logic. In *Graphical Models*. Clarendon Press, Oxford.

[Ledoux, 1981] Ledoux, A. (1981). Concours résultats complets. les victimes se sont plu à jouer. *Jeux et Stratégie*, pages 10–11.

[Leite et al., 2014] Leite, A. R., Enembreck, F., and Barthès, J.-P. A. (2014). Distributed Constraint Optimization Problems: Review and perspectives. *Expert Systems with Applications*, 41(11):5139–5157.

[Lemke and Jr, 1964] Lemke, C. and Jr, J. H. (1964). Equilibrium points of bimatrix games. *SIAM*, 12:413–423.

[Leyton-Brown and Tennenholtz, 2003] Leyton-Brown, K. and Tennenholtz, M. (2003). Local-effect games. In *IJCAI*, pages 772–780.

[Luby et al., 1993] Luby, M., Sinclair, A., and Zuckerman, D. (1993). Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180.

[Luce and Raiffa, 1957] Luce, R. and Raiffa, H. (1957). *Games and Decisions: Introduction and Critical Survey*. Dover books on advanced mathematics. Dover Publications.

[Maheswaran et al., 2004] Maheswaran, R. T., Pearce, J. P., and Tambe, M. (2004). Distributed algorithms for dcop: A graphical-game-based approach. In *ISCA PDCS*, pages 432–439.

[Maynard-Smith and Price, 1973] Maynard-Smith, J. and Price, G. R. (1973). The logic of animal conflict. *Nature*, 246:15–18.

[McKelvey et al., 2014] McKelvey, R. D., McLennan, A. M., and Turocy, T. L. (2014). Gambit: Software tools for game theory.

[Mehlhorn and Thiel, 2000] Mehlhorn, K. and Thiel, S. (2000). Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *CP*, pages 306–319.

[Miller and Drechsler, 1998] Miller, D. M. and Drechsler, R. (1998). Implementing a multiple-valued decision diagram package. In *ISMVL*, pages 52–57.

[Miller and Drechsler, 2002] Miller, D. M. and Drechsler, R. (2002). On the construction of multiple-valued decision diagrams. In *ISMVL*, pages 245–253.

[Minton et al., 1992] Minton, S., Johnston, M. D., Philips, A. B., and Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artif. Intell.*, 58(1-3):161–205.

[Monderer and Shapley, 1996] Monderer, D. and Shapley, L. S. (1996). Potential games. *Games and Economic Behavior*, 14(1):124 – 143.

[Morris, 1993] Morris, P. (1993). The breakout method for escaping from local minima. In *AAAI*, pages 40–45.

[Nash, 1951] Nash, J. (1951). Non-cooperative games. *Annals of Mathematics*, 54(2):286–295.

[Nguyen and Lallouet, 2014] Nguyen, T.-V.-A. and Lallouet, A. (2014). A complete solver for constraint games. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 58–74.

[Nguyen et al., 2013] Nguyen, T.-V.-A., Lallouet, A., and Bordeaux, L. (2013). Constraint games: Framework and local search solver. In *ICTAI*, pages 963–970.

[Nudelman et al., 2004] Nudelman, E., Wortman, J., Shoham, Y., and Leyton-Brown, K. (2004). Run the gamut: A comprehensive approach to evaluating game-theoretic algorithms. In *AAMAS*, pages 880–887. IEEE Computer Society. `http://gamut.stanford.edu/`.

[Ortiz and Kearns, 2002] Ortiz, L. E. and Kearns, M. J. (2002). Nash propagation for loopy graphical games. In *NIPS*, pages 793–800.

[Osborne, 2004] Osborne, M. (2004). *An introduction to game theory*. Oxford Univ. Press, New York.

[Osborne and Rubinstein, 1994] Osborne, M. and Rubinstein, A. (1994). *A Course in Game Theory*. The MIT Press.

[Papadimitriou, 2007] Papadimitriou, C. H. (2007). *The complexity of Finding Nash Equilibria*, chapter 2, pages 29–51. Algorithmic game theory. Cambridge University Press.

[Paquete and Stützle, 2002] Paquete, L. and Stützle, T. (2002). An experimental investigation of iterated local search for coloring graphs. In *EvoWorkshops*, pages 122–131.

[Porter et al., 2008] Porter, R., Nudelman, E., and Shoham, Y. (2008). Simple search methods for finding a nash equilibrium. *Games and Economic Behavior*, 63(2):642–662.

[Poundstone, 1988] Poundstone, W. (1988). *Labyrinths of Reason: Paradox, Puzzles, and the Fragility of Knowledge*. Anchor Doubleday Publishing Company.

[Prosser, 1993] Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299.

[Puget, 1998] Puget, J.-F. (1998). A fast algorithm for the bound consistency of alldiff constraints. In *AAAI/IAAI*, pages 359–366.

[Rojas, 1996] Rojas, M. C. R. (1996). From quasi-solutions to solution: An evolutionary algorithm to solve csp. In *CP*, pages 367–381.

[Rosen, 1965] Rosen, J. B. (1965). Existence and uniqueness of equilibrium points for concave n-person games. *Econometrica*, 33(3):520–534.

[Rosenthal, 1973] Rosenthal, R. W. (1973). A class of games possessing pure-strategy nash equilibria. *International Journal of Game Theory*, 2(1):65–67.

[Rosiers and Bruynooghe, 1986] Rosiers, W. and Bruynooghe, M. (1986). Empirical study of some constraints satisfaction algorithms. In *AIMSA*, pages 173–180.

[Rossi et al., 2006] Rossi, F., van Beek, P., and Walsh, T., editors (2006). *Handbook of Constraint Programming*. Elsevier.

[Roughgarden, 2007] Roughgarden, T. (2007). *Routing Games*, chapter 18, pages 461–486. Algorithmic game theory. Cambridge University Press.

[Roughgarden and Tardos, 2007] Roughgarden, T. and Tardos, E. (2007). *Introduction to the Inefficiency of Equilibria*, chapter 17, pages 443–459. Algorithmic game theory. Cambridge University Press.

[Rozenfeld and Tennenholtz, 2006] Rozenfeld, O. and Tennenholtz, M. (2006). Strong and correlated strong equilibria in monotone congestion games. In *WINE*, pages 74–86.

[Scarf, 1973] Scarf, H. E. (1973). *The computation of economic equilibria*. Yale University Press. with collaboration of T. Hansen.

[Sen et al., 2003] Sen, S., Airiau, S., and Mukherjee, R. (2003). Towards a pareto-optimal solution in general-sum games. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, pages 153–160, New York, NY, USA. ACM.

[Shapley, 1963] Shapley, L. (1963). *Some Topics in Two-person Games*. Memorandum (Rand Corporation). Rand Corporation.

[Sherali et al., 1983] Sherali, H. D., Soyster, A. L., and Murphy, F. H. (1983). Stackelberg-Nash-Cournot equilibria: characterizations and computations. *Oper. Res.*, 31(2):253–276.

[Shoham and Leyton-Brown, 2009] Shoham, Y. and Leyton-Brown, K. (2009). *Multiagent Systems - Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.

[Shubik, 1981] Shubik, M. (1981). Chapter 7 game theory models and methods in political economy. volume 1 of *Handbook of Mathematical Economics*, pages 285 – 330. Elsevier.

[Solnon, 2001] Solnon, C. (2001). Ants can solve constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*.

[Son and Baldick, 2004] Son, Y. S. and Baldick, R. (2004). Hybrid coevolutionary programming for nash equilibrium search in games with local optima. *IEEE Trans. Evolutionary Computation*, 8(4):305–315.

[Soni et al., 2007] Soni, V., Singh, S. P., and Wellman, M. P. (2007). Constraint satisfaction algorithms for graphical games. In *AAMAS*, page 67.

[Soomer and Franx, 2008] Soomer, M. and Franx, G. (2008). Scheduling aircraft landings using airlines' preferences. *European Journal of Operational Research*, 190(1):277 – 291.

[Stackelberg, 1952] Stackelberg, H. V. (1952). *The Theory of the Market Economy*. Oxford University Press, New York. Translated from the German.

[Steinmann et al., 1997] Steinmann, O., Strohmaier, A., and Stützle, T. (1997). Tabu search vs. random walk. In *KI*, pages 337–348.

[Stützle, 1998] Stützle, T. (1998). *Local Search Algorithms for Combinatorial Problems - Analysis, Improvements and New Applications*. PhD thesis, TU Darmstadt, FB Informatik, Darmstadt, Germany.

[Tesfatsion, 2006] Tesfatsion, L. (2006). Chapter 16 agent-based computational economics: A constructive approach to economic theory. volume 2 of *Handbook of Computational Economics*, pages 831 – 880. Elsevier.

[Tsang, 1993] Tsang, E. (1993). Foundations of constraint satisfaction.

[Turocy, 2013] Turocy, T. L. (2013). Personal communication.

[van Beek, 2006] van Beek, P. (2006). Backtracking search algorithms. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 4. Elsevier.

[van Hoeve and Katriel, 2006] van Hoeve, W.-J. and Katriel, I. (2006). Global constraints. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 6. Elsevier.

[Vickrey and Koller, 2002] Vickrey, D. and Koller, D. (2002). Multi-agent algorithms for solving graphical games. In *AAAI/IAAI*, pages 345–351.

[Vöcking, 2007] Vöcking, B. (2007). *Selfish load balancing*, chapter 20, pages 517–542. Algorithmic game theory. Cambridge University Press.

[von Neumann and Morgenstern, 1944a] von Neumann, J. and Morgenstern, O. (1944a). *Formulation of the general theory: zero-sum n-person games*, chapter 6. Princeton University Press.

[von Neumann and Morgenstern, 1944b] von Neumann, J. and Morgenstern, O. (1944b). *General non-zero-sum games*, chapter 11. Princeton University Press.

[von Neumann and Morgenstern, 1944c] von Neumann, J. and Morgenstern, O. (1944c). *Theory of Games and Economic Behavior*. Princeton University Press.

[Voudouris and Tsang, 1996] Voudouris, C. and Tsang, E. P. K. (1996). Partial constraint satisfaction problems and guided local search. In *Proc., Practical Application of Constraint Technology (PACT'96)*, pages 337–356.

[Wooldridge et al., 2013] Wooldridge, M., Endriss, U., Kraus, S., and Lang, J. (2013). Incentive engineering for boolean games. *Artif. Intell.*, 195:418–439.

[Yannakakis, 2009] Yannakakis, M. (2009). Equilibria, fixed points, and complexity classes. *Computer Science Review*, 3(2):71 – 85.

[Zabih, 1990] Zabih, R. (1990). Some applications of graph bandwidth to constraint satisfaction problems. In *AAAI*, pages 46–51.

[Zaccour, 2005] Zaccour, G., editor (2005). *Dynamic Games and Applications*. Springer.

[Zhao and Müller, 2004] Zhao, L. and Müller, M. (2004). Game-SAT: A preliminary report. In Hoos, H. and Mitchell, D., editors, *SAT*, pages 357–362.

Constraint Games: Modeling and Solving Games with Constraints

This thesis presents a topic at the interface of game theory and constraint programming. More precisely, we focus on modeling games in a succinct way and then computing their solutions on these compactly encoded games thanks to constraint programming.

For a long period of time, game theory has suffered a modeling difficulty due to the lack of compact representations for encoding arbitrary games. The basic game representation is still an n-dimensional matrix called *normal form* which stores utilities of all players with all joint strategy profiles. The matrix however exponentially grows with the number of players. This also causes a solving difficulty for computing solutions in games.

In the thesis, we introduce a novel framework of *Constraint Games* to model strategic interaction between players. A constraint game is composed of a set of variables shared by all the players. Among these variables, each player owns a set of *decision variables* he can control and constraints expressing his utility function. The *Constraint Games* framework is a generic tool to model general games and can be exponentially more succinct than their normal form. We also show the usefulness of the framework by modeling a few classical games as well as realistic problems.

The main solution concept defined for constraint games is *Pure Nash Equilibrium*. It is a situation in which no player has an incentive to deviate unilaterally. It has been well-known that finding pure Nash equilibrium is computationally complex. Nevertheless, we have achieved to build an efficient solver called *ConGa* which includes various ways for solving constraint games.

Keywords: Game Theory, Constraint Programming , Nash Equilibrium

Constraint Games: Modélisation et Résolution des Jeux avec Contraintes

Cette thèse se situe dans le cadre à la fois de la théorie des jeux et de la programmation par contrainte. Plus précisément, nous nous concentrons sur modéliser les jeux de manière succincte et puis calculer leurs solutions sur ces représentations compactes grâce à la programmation par contraintes.

Depuis longtemps, la théorie des jeux subit une difficulté de modélisation en raison de l'absence de représentations compactes pour modéliser les jeux. La représentation basique est encore une matrice appelée *forme normale* qui stocke les utilités des joueurs avec tous les profils de stratégie. Pourtant, cette matrice s'agrandit exponentiellement avec le nombre de joueurs. Cela cause une difficulté de la résolution des jeux.

Dans cette thèse, nous introduisons un nouveau framework de *Constraint Games* afin de modéliser l'interaction stratégique entre les joueurs. Un *constraint game* est composé d'un ensemble de variables partagées par tous les joueurs. Parmi ces variables, chaque joueur possède un ensemble de variables décisionnelles qu'il peut contrôler et les contraintes représentant sa fonction d'utilité. Le framework de *Constraint Games* est un outil générique pour modéliser les jeux généraux et peut être exponentiellement plus succinct que leur forme normale. Nous montrons également l'utilité du framework pour modéliser quelques jeux classiques et problèmes réels.

Le concept principal de solution défini pour *Constraint Games* est l'*Équilibre de Nash en Stratégies Pures*. C'est une situation dans laquelle aucun joueur n'a un intérêt à dévier unilatéralement. Il est bien connu qu'il est complexe de calculer les équilibres de Nash en stratégies pures dans les jeux. Pourtant, nous réussissons à construire un solver appelé *ConGa* qui inclut plusieurs méthodes pour résoudre *Constraint Games* efficacement.

Mots clés: Théorie des Jeux, Programation par Contraintes , Équilibre de Nash

Discipline: Informatique et applications
Laboratoire: GREYC CNRS UMR 6072, Sciences 3, Campus 2, Bd Marechal Juin, Université de Caen, 14032 Caen