Une nouvelle approche au General Game Playing dirigée par les contraintes

Eric Piette

UNIVERSITÉ D'ARTOIS

Une nouvelle approche au `General Game Playing` dirigée par les contraintes

THÈSE

présentée et soutenue publiquement le 09 décembre 2016

en vue de l'obtention du

Doctorat de l'Université d'Artois

(Spécialité Informatique)

par

Éric Piette

Composition du jury

Rapporteurs :	Christian Bessière	Université de Montpellier
	Tristan Cazenave	Université Paris-Dauphine
	Arnaud Lallouet	Huawei Technologies
Examinateurs :	Frédéric Koriche	Université d'Artois, co-directeur de thèse
	Sylvain Lagrue	Université d'Artois, directeur de thèse
	Anastasia Paparrizou	Université d'Artois
	Florian Richoux	Université de Nantes
	Sébastien Tabary	Université d'Artois, co-directeur de thèse

CENTRE DE RECHERCHE EN INFORMATIQUE DE LENS – CNRS UMR 8188
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : Tél.: +33 (0)3 21 79 17 23 – Fax : +33 (0)3 21 79 17 70
http://www.cril.univ-artois.fr

Remerciements

N'allez pas là où le chemin peut mener.
Allez là où il n'y a pas de chemin et laissez une trace.
[Ralph Waldo Emerson]

À l'instar de ce philosophe, j'ai choisi la recherche semée de défis grâce à laquelle j'espère laisser une trace de mon passage. Mais bien heureusement, au cours de ce périple, je n'étais pas seul et il m'est important de rendre hommage à toute personne m'ayant accompagné au cours de celui-ci.

Naturellement, mes premiers remerciements se dirigent vers mes encadrants, Fred, Seb et Sylvain. Cette aventure n'aurait pu avoir lieu sans la cohésion de groupe issue de l'addition des compétences de chacun d'entre eux. À chaque défi, questionnement ou retournement de situation, ils étaient là au bon moment avec la soif de savoir qui les caractérisent afin d'élucider chacun des mystères rencontrés. Pour cela, je leur adresse un très grand MERCI.

Je tiens également à remercier Christian Bessière, Tristan Cazenave et Arnaud Lallouet, qui me font l'honneur d'être rapporteurs de ce manuscrit, ainsi que Anastasia Paparrizou et Florian Richoux qui en sont les examinateurs. Défendre mes travaux face à un jury d'une telle qualité est pour moi une grande fierté.

Jamais, il ne m'aurait été possible de survivre aux épreuves d'une thèse sans le soutien de chacun des membres du CRIL que je remercie chaleureusement. À mes yeux, ce laboratoire est une mine de savoir et de joie de vivre dont la réputation est – je peux maintenant parler par expérience – absolument méritée. Je remercie plus particulièrement, ceux que j'aime à surnommer, l'équipe AOE, qui ont à de nombreuses reprises su me surprendre par diverses stratégies plus folles les unes que les autres. Quand vous voulez les gars !

Assez originalement, au sein de ce laboratoire se trouve un autre membre de ma fratrie, aujourd'hui président de l'association française portant ses propres initiales CP. Mon vocabulaire n'est pas assez développé pour te dire à quel point, je suis heureux d'avoir pu travailler à tes côtés et il l'est encore moins pour te remercier pour les indénombrables conseils de vie que tu as su me prodiguer depuis mon plus jeune âge. Toi qui me connais, *Bro'*, sache que jamais je n'aurais pu espérer un meilleur partenaire d'enfance m'ouvrant les portes de l'informatique dès mes débuts. Cécile, j'en profite pour te remercier tout d'abord pour la relecture bien nécessaire, mais aussi pour le bien que tu lui apportes.

Je remercie de tout mon cœur, mes premiers encadrants, à savoir mes parents qui m'ont poussé tout au long de ma vie et permis d'atteindre ce résultat aujourd'hui. Indéniablement un énorme merci à vous pour tout. Votre soutien ces nombreuses années m'ont très certainement permis de persévérer afin de résoudre une à une chacune des contraintes rencontrées. De même, il m'est important de jeter mon regard vers la génération supérieure, Mémé et Pépé. Qui, derrière ce placard magique rempli de jeux à n'en plus finir mais aussi pour les nombreux tas de cartes usées au fil des dimanches, m'ont fourni mes premiers amours à ce domaine ludique.

Mon plus grand merci se dirige vers Marine, cette fameuse petite fée qui a su dompter la bête qui était en moi et la remettre sur le droit chemin à de nombreuses reprises. Son soutien indéniable à chaque moment, notamment lors des compétitions, est aussi ce qui a permis de concrétiser ce travail. Mes remerciements pour cette « grande » pâtissière ne se limiteront en rien à ces quelques lignes mais en chacun de mes actes avenirs comme elle le sait si bien. Ma bien aimée, merci pour tout et pour tout ce qui viendra.

Je tiens également à remercier ma belle-famille, Agnes, Franck, Margaux, Monique, ... Merci ! Jamais un jour, je n'aurais espéré avoir une telle seconde famille accueillante et pleine de joies. Vos encouragements, notamment pendant l'écriture de ce manuscrit, raisonnent encore dans ma tête. Bien sûr, je ne t'oublie pas, François, toi qui n'es plus là, tu as su trouver le temps de me souhaiter la bienvenue parmi les vôtres et cela je ne l'oublierai jamais.

Bien que la place me manque, je me tourne désormais vers vous, mes amis. Merci à Luigi d'avoir survécu à mes effroyables insomnies en tant que binôme au cours de mes études pré-doctorales. De même, merci également à deux autres bidochons, Clément et Nico qui se reconnaîtront dans leur surnom. Merci aussi à Aurélien, Coco, David, F.M. pour ses longues nuits déjantées de « réflexions intenses sur le sens de la vie » autour de maintes liqueurs et délicieuses chopes (santé les gars !). Merci à Nadia et Jordan pour ce petit coin de zénitude qu'ils nous offrent à chacun de nos passages. Un merci tout particulier à Ludo, pour sa vision de la vie, sa bonne humeur sans fin et ses blagues toutes aussi colorées que les miennes. Enfin merci à tout ceux qui m'ont entouré au cours de ces années et dont je n'ai plus la place pour citer le nom et qui se reconnaîtront dans ces lignes.

Et je terminerai par te remercier toi. Oui toi, Lecteur ! Merci de l'intérêt que tu portes à mes travaux en ouvrant ce mémoire. Bonne lecture !

L'Intelligence Artificielle se définit
comme le contraire de la bêtise naturelle.

[Woody Allen]

Table des matières

Introduction **1**

Partie I État de l'art

Chapitre 1 *General Game Playing* **7**

1.1 Bases en théorie des jeux 7

1.1.1 Jeux en forme normale 8

1.1.2 Jeux en forme extensive 12

1.1.3 Jeux stochastiques 16

1.2 *General Game Playing* 18

1.2.1 *Game Description Langage* (GDL) 19

1.2.2 Compétition internationale : IGGPC 24

1.2.3 Game Description Langage with Incomplete Information (GDL-II) 26

1.3 Différentes approches appliquées au GGP 32

1.3.1 Algorithmes de type Minimax 32

1.3.2 Heuristiques de recherche 35

1.3.3 Méthodes de Monte Carlo 37

1.3.4 Variantes et améliorations de UCT 40

1.3.5 Techniques de parallélisation de MCTS 43

1.3.6 Réseau de propositions ($propnet$) 48

1.3.7 Approches pour les jeux GDL-II 52

1.4 Conclusion 54

Chapitre 2 Programmation par contraintes **55**
2.1 Réseaux de contraintes 55
2.1.1 Définitions et notations 56
2.1.2 Exemple de représentation : Le problème des n-reines 60
2.2 Problème de Satisfaction de contraintes (CSP) 61
2.2.1 Inférence 63
2.2.2 Arc-cohérence 65
2.2.3 Arc-cohérence généralisée 72
2.2.4 Stratégies de recherche 79
2.3 Réseaux de contraintes stochastiques (SCSP) 90
2.3.1 Définitions et Notations 90
2.3.2 Algorithmes de résolutions 94
2.3.3 Quelques algorithmes incomplets 98
2.4 Conclusion 101

Partie II Contributions

Chapitre 3 Modélisation par contraintes de jeux GDL **105**
3.1 Équivalence entre modèles 106
3.1.1 *Ground* instance GDL propositionnel 106
3.1.2 P-GDL $\Longleftrightarrow$ SCSP 107
3.2 De GDL à SCSP : Exemple 108
3.2.1 Extraction des variables 109
3.2.2 Extraction des domaines et des distributions de probabilités 109
3.2.3 Extraction des contraintes 109
3.3 Extension du modèle aux jeux à information partagée 112
3.3.1 Fragment PSPACE de GDL-II 112
3.3.2 De GDL-II à SCSP 114
3.4 Expérimentations 115
3.4.1 Jeux GDL 115
3.4.2 Jeux GDL-II 117
3.5 Modélisation des stratégies 122
3.6 Conclusion 123

Chapitre 4 Une approche GGP dirigée par les contraintes **125**
4.1 Un fragment SCSP pour GDL . . . 126
4.2 MAC-UCB . . . 127
4.2.1 Phase de pré-traitement . . . 127
4.2.2 Phase de résolution : MAC . . . 128
4.2.3 Phase de simulation : UCB . . . 131
4.3 MAC-UCB-II . . . 132
4.4 Évaluations expérimentales . . . 134
4.4.1 Dilemme exploration/exploitation . . . 134
4.4.2 MAC-UCB Vs. FC-UCB . . . 135
4.4.3 MAC-UCB face aux meilleures approches GDL . . . 136
4.4.4 MAC-UCB-II face aux meilleures approches GDL-II . . . 140
4.5 WoodStock : un programme-joueur dirigé par les contraintes . . . 144
4.5.1 Protocole de communication . . . 145
4.5.2 Phase de traduction . . . 145
4.5.3 Phase de résolution . . . 146
4.5.4 Phase de simulation . . . 147
4.6 WoodStock en compétition . . . 147
4.6.1 *Tiltyard* Open 2015 . . . 147
4.6.2 Tournoi continu GGP . . . 149
4.7 Conclusion . . . 150

Chapitre 5 Détection de symétries par les contraintes pour le General Game Playing **153**
5.1 Détection de symétries dans le *General Game Playing* . . . 155
5.1.1 Jeux et symétries . . . 155
5.1.2 Méthode des graphes de règles (méthode RG) . . . 157
5.2 Détection de symétries dans la programmation par contraintes . . . 158
5.2.1 Deux types de symétries . . . 158
5.2.2 Exemple d'application . . . 160
5.3 Symétries de jeux GDL et symétries dans un SCSP . . . 163
5.4 MAC-UCB-SYM . . . 168
5.4.1 Détection en pratique des symétries . . . 168
5.4.2 Profondeur des stratégies minimax . . . 168
5.5 Résultats expérimentaux . . . 169
5.5.1 Détection des symétries entre la méthode RG et les contraintes . . . 169
5.5.2 Étude de la détection de symétries par la méthode RG . . . 170
5.5.3 MAC-UCB-SYM dans le cadre GDL . . . 172

5.5.4 `MAC-UCB-II-SYM` dans le cadre `GDL-II` 174
5.5.5 WoodStock et `MAC-UCB-SYM`, Champion `GGP` 2016 178
5.6 Conclusion 180

Conclusion **181**

Table des figures **183**

Liste des tableaux **185**

Bibliographie **187**

Introduction

C'est en 1956 que le concept d'Intelligence Artificielle (IA) est énoncé pour la première fois publiquement lors de la conférence de *Dartmouth* et déjà les jeux sont mentionnés comme source de défis. La même année, Arthur Samuel, un pionnier du domaine, développe le premier programme jouant aux dames qui, dès 1962, parvient à battre un joueur de bon niveau devenant ainsi la première machine à vaincre l'homme de l'histoire. Suite à cette victoire, deux grandes tendances ont émergé en IA pour résoudre les jeux : les approches dédiées, se concentrant sur la résolution d'un jeu particulier, et les approches génériques, s'appliquant à une grande variété de jeux. Son objectif reste le même, réaliser un programme-joueur capable de jouer stratégiquement.

Par programme-joueur, on entend un programme informatique ayant la capacité de jouer à un ou plusieurs jeux. On parle de programme-joueur dédié si il a été conçu pour un jeu en particulier et de programme-joueur générique si il a été conçu pour un ensemble de jeux.

Les approches dédiées

De nombreux programmes dédiés à différents jeux font leur apparition dont notamment DeepBlue qui, en 1997, remporte la victoire sur *Garry Kasparov*, le champion international en titre d'échecs (un jeu composé d'autant d'états que d'atomes dans l'univers : 10^{80}) relevant le défi de l'époque. L'approche dédiée a fait l'objet de nombreuses études dont voici quelques exemples représentatifs de programmes-joueurs dédiés les plus connus :

Les jeux « **solitaires** » aussi nommés casse-têtes ou puzzles. Il s'agit de jeux à information complète à un joueur souvent résolus comme des problèmes de planification ou de contraintes. Les Sudoku [PEREZ & MARWALA 2008], le *Rubik's Cube* [KORF 1997], le jeu du solitaire (ou *peg solitaire*) [JEFFERSON *et al.* 2006], le Rush Hour [COLLETTE *et al.* 2006] ou encore le Sokoban [JUNGHANNS & SCHAEFFER 2001] illustrent cette catégorie de jeux.

Les jeux multijoueurs à information complète opposent un ensemble de joueurs ayant pleine connaissance de l'état du jeu à chaque tour. Généralement, deux joueurs s'affrontent en face à face : Awalé [ROMEIN & BAL 2003], les Dames [SCHAEFFER 2008], les Échecs [FERNANDEZ & SALMERON 2008], le Go [SILVER *et al.* 2016], Othello [VAN ECK & VAN WEZEL 2008], le Morpion (aussi nommé *Tic-tac-toe*) [GRIM *et al.* 2005], le Puissance 4 (ou *Connect 4*) [SCHNEIDER & GARCIA ROSA 2002], Hex [ARNESON *et al.* 2010]. Il existe également des études sur des jeux opposant plus de deux joueurs comme par exemple les Dames chinoises [STURTEVANT 2002].

Les jeux à information imparfaite. Ce sont des jeux où chaque état est sous l'influence du hasard qui peut être par exemple modélisé par l'utilisation de dés ou de pièces. Voici quelques exemples représentatifs de cette catégorie : Le Monopoly [FRAYN 2005], le Backgammon [AZARIA & SIPPER 2005] ou le Yahtzee [PAWLEWICZ 2011].

Les jeux à information incomplète. Il s'agit de jeux où chaque joueur ne possède pas l'ensemble des données lui permettant de connaître complètement l'état du jeu et/ou les actions des autres joueurs.

Le Bridge [GINSBERG 2011], le Poker [BOWLING *et al.* 2015], le Mah-jong [DE BONDT 2012] ne sont que quelques exemples de cette catégorie utilisant un ensemble de cartes pour cacher l'information aux joueurs adverses. D'autres jeux comme, le Go aveugle [CAZENAVE 2006] ou le Kriegspiel [CIANCARINI & FAVINI 2010] font intervenir un arbitre désignant si les actions réalisées par chaque joueur sont légales. De nombreuses recherches en `IA` existent notamment sur les techniques de bluff [HURWITZ & MARWALA 2007] souvent utilisées dans de tels jeux.

Certaines catégories de jeux sont encore peu étudiées comme les jeux à règles altérables (ex : Democrazy ou Tempête sur l'échiquier) ou encore les *wargames* (Warhammer 4000 ou Blood bawl). Ces jeux sont souvent basés sur des règles du jeu évoluant tout au long de la partie. Ainsi, la réalisation d'un programme-joueur capable d'une abstraction telle que celle nécessaire à ces jeux et l'évolution incessante des règles du jeu posant des problèmes de transcription en un modèle facilement simulable rendent l'étude de ces catégories de jeux très ardue en Intelligence Artificielle.

Parmi toutes ces études, quelques programmes-joueurs dédiés ont obtenu une grande renommée en ayant vaincu les champions du monde dans ces jeux jusqu'au bien connu `AlphaGo`, qui vient de vaincre *Lee Sedol*, le champion incontesté du jeu de Go, impliquant un jeu d'une complexité bien plus élevée que les échecs (10^{152} états). Le tableau 1 indique quelques exemples de programme-joueur parmi les plus connus ordonnés par leur année de publication.

Année	**Jeu**	**Programme-joueur**	**Auteur(s)**
1997	Échecs	Deep Blue	[CAMPBELL *et al.* 2002]
1997	Othello	Logistello	[BURO 2002]
2002	Scrabble	Maven	[SHEPPARD 2002]
2008	Dames	Chinook	[SCHAEFFER 2008]
2015	Poker Texas Hold'em limit [1]	Cepheus	[BOWLING *et al.* 2015]
2016	Go	AlphaGo	[SILVER *et al.* 2016]

TABLE 1 – Programmes-joueurs à renommée mondiale suite à leurs victoires sur les champions humains.

Vers une approche générique : le *General Game Playing*

Parallèlement aux approches spécialisées rencontrant de nombreux succès spectaculaires, une seconde voie est envisagée par la recherche en Intelligence Artificielle : la programmation de programmes-joueurs génériques, c'est à dire ne se contentant pas d'un seul jeu mais capable de jouer à tout un panel de jeux. Dès 1959, le *General Problem Solver* est initié par *Simon*, *Shaw* et *Newell*, comme première tentative d'un système artificiel pour résoudre n'importe quel problème. Ce système, qui a eu une grande influence sur l'évolution de l'Intelligence Artificielle et des Sciences Cognitives, résolvait très bien des problèmes simples, mais était limité dès lors que la combinatoire du problème augmentait.

Historiquement, la première modélisation générique fut présentée par [PITRAT 1968] proposant un langage spécifique permettant de décrire les règles de jeux à information complète. Ce langage se limite aux jeux à deux joueurs et se réalisant sur des plateaux rectangulaires tels que le *Tic-tac-toe* ou les échecs. Dans le même registre [PELL 1994] propose un système nommé *Metagamer* permettant de décrire les règles de jeux similaires aux échecs.

1. Notons que ce programme-joueur n'a la capacité de vaincre l'humain que lors d'un Poker très particulier se jouant uniquement à deux joueurs. Le Poker à n joueurs reste un challenge de l'intelligence artificielle.

Au cours de la même décennie, [GHERRITY 1993] propose un programme nommé SAL permettant d'apprendre un jeu déterministe à somme nulle mais ne fournit pas de langage propre pour décrire génériquement un jeu. [EPSTEIN 1994] initie HOYLE qui s'avéra proche de SAL mais moins efficace, tandis que [LEVINSON 1994] propose MORPH II un système d'apprentissage automatique des règles d'un jeu. Malheureusement, ce dernier se restreint aux petits jeux ne possédant que très peu d'états.

MULTIGAME proposé par [ROMEIN 2001] au début des années 2000 est un système réputé pour son efficacité de par l'utilisation d'une mémoire distribuée. Mais il se limite aux jeux déterministes à un ou deux joueurs avec information complète et les jeux fournis par ce système sont soumis à la qualité de l'auteur des jeux.

Plus tard, [LEFLER & MALLETT 2002] propose le système commercial dénommé *Zillions of games* permettant de représenter plus de 1700 jeux à information complète « solitaire » et/ou de pions. Toutefois, le langage logique associé à la description des règles des jeux n'est pas facilement utilisable ce qui empêche le développement de programmes-joueurs génériques efficaces.

En 2005, [KAISER 2005] propose le langage RGL (pour *Regular Game Language*) décrivant des chemins dans un graphe. RGL est capable de représenter les jeux déterministes ou non-déterministes mais également les jeux à information complète ou incomplète. Malheureusement RGL ne s'imposera pas dans le domaine du *General Game Playing* suite à l'apparition d'un autre langage : GDL.

Depuis 2005, les approches génériques visant à jouer intelligemment à une grande variété de jeux, sans expertise ni technologie dédiée à un jeu particulier, réémergent sous le nom de *General Game Playing* (GGP) au travers d'un projet initié par le *Stanford Logic Group* [2]. Ce domaine consiste à développer des algorithmes qui, juste après avoir reçu les règles d'un nouveau jeu, sont capables de jouer stratégiquement à ce jeu sans aucune intervention humaine. Les règles du jeu sont décrites dans un langage logique, appelé *Game Description Language* (GDL) qui de par sa simplicité permet de représenter l'ensemble des jeux tour par tour à information complète [LOVE *et al.* 2006]. Une extension dénommée *Game Description Language with Incomplete Information* (GDL-II) permettant de modéliser les jeux à information incomplète a également été proposée [THIELSCHER 2010].

Le cadre du *General Game Playing* ne se limite pas uniquement à l'univers ludique mais permet également la résolution de problèmes de prise de décisions séquentielles ou multi-agents dont notamment des applications directes au domaine de la robotique. Il motive la recherche scientifique pour l'élaboration d'approches génériques. De nombreuses techniques de résolutions ont ainsi vu le jour, mettant en application des méthodes telles que la programmation logique et l'*Answer Set Programming*, la construction automatique de fonctions d'évaluations et d'heuristiques ou encore ce qui fait à ce jour référence : les méthodes de type Monte Carlo. Chacune des ces méthodes a donné naissance à un ou plusieurs programmes-joueurs génériques se confrontant les uns aux autres au travers de compétitions annuelles ou continues afin de déterminer les meilleures techniques et approches de résolutions [GENESERETH & BJÖRNSSON 2013].

La réalisation d'un tel joueur est l'objectif principal du *General Game Playing* (GGP) sur lequel se concentre ce manuscrit.

Objectifs et plan de la thèse

Les travaux présentés dans cette thèse ont pour objectif d'aborder le problème du *General Game Playing* d'un point de vue original basé sur la résolution de réseaux de contraintes stochastiques. En effet, par son approche déclarative de la résolution de problèmes combinatoires, la programmation par contraintes (CSP) offre un cadre prometteur pour relever le défi. Plus précisément, les réseaux de

2. Stanford Logic Group : http://games.stanford.edu/

contraintes stochastiques représentent un outil puissant permettant non seulement de modéliser et de résoudre des jeux de stratégie mais également de modéliser l'intervention du hasard.

Ce manuscrit est composé de cinq chapitres, les deux premiers présentent l'état de l'art tandis que les trois suivants mettent en avant les principales contributions. Tout d'abord, certaines notions essentielles issues de la théorie des jeux, nécessaires à la compréhension de ce document, sont détaillées. Puis, le sujet central de cette thèse, le *General Game Playing* est présenté en détail. Dans un premier temps, nous présentons le formalisme GDL permettant de décrire chacun des jeux sur lesquels se confrontent les différents programmes-joueurs lors des compétitions GGP. Suite à cela, les approches GGP remportant le plus de succès depuis 2005 sont décrites jusqu'à celles utilisées actuellement par les champions GGP les plus récents. Enfin, un tour d'horizon des approches GGP pour les jeux à information incomplète est réalisé. Le second chapitre s'intéresse à mettre en évidence le cadre des réseaux de contraintes. Après avoir présenté ce cadre, l'inférence, l'arc-cohérence et les stratégies de recherche dans un CSP sont détaillées avant de mettre en avant les réseaux de contraintes stochastiques qui sont au centre de cette thèse.

La seconde partie du manuscrit met l'accent sur nos contributions. La première contribution, détaillée dans le troisième chapitre, se consacre à élaborer une modélisation par contraintes stochastiques pour le formalisme GDL et à son extension GDL-II. Le SCSP obtenu est montré équivalent au jeu GDL original. La section suivante s'attarde à identifier un fragment de GDL-II montré PSPACE-complet représentant l'ensemble des jeux à information partagée et nous établissons un processus de traduction similaire de ce fragment vers SCSP. Enfin, avant de conclure ce chapitre, nous montrons expérimentalement que la génération d'un SCSP équivalent est compatible avec les conditions imposées par les compétitions GGP et que la modélisation de différents types de stratégies de jeux est envisageable par l'ajout de contraintes au réseau de contraintes stochastiques obtenu.

Le chapitre 4 identifie le fragment SCSP généré à l'aide du processus de traduction décrit auparavant et propose de l'exploiter afin d'établir une approche GGP dirigée par la résolution de réseaux de contraintes stochastiques combinée à une méthode d'échantillonnage de type Monte Carlo. Cet algorithme porte le nom de MAC-UCB suite à l'association de l'algorithme de *Maintening Arc-Consistency* (MAC) et de la méthode d'*Upper Confidence Bounds* (UCB). Puis, nous montrons qu'une version pour GDL-II, dénommée MAC-UCB-II est envisageable de par la conservation du fragment SCSP obtenu par le processus de traduction d'un jeu GDL-II en réseau de contraintes. Suite à ce constat, nous montrons expérimentalement que MAC-UCB se montre compétitif face aux meilleures approches GGP à information complète et incomplètes. Puis, après avoir présenté WoodStock, notre programme-joueur générique implémentant MAC-UCB, nous montrons qu'actuellement, il se présente comme le champion GGP de la compétition continue.

Enfin, le dernier chapitre s'attarde à la notion de symétries dans les jeux. Nous commençons par détailler la meilleure méthode GGP de détection de symétries présentant quelques défauts, empêchant notamment son application dans le cadre d'une compétition. Ensuite, une description de la recherche de symétries dans le cadre des réseaux de contraintes est effectuée. Les symétries de structure de jeu et les symétries de stratégies minimax sont mises en relation avec la notion de symétries dans les SCSP. Nous montrons que la détection de symétries appliquée via notre modèle par contraintes pour GGP permet d'englober théoriquement l'ensemble des symétries de jeux. Le cadre pratique est décrit via une extension de notre approche dénommée MAC-UCB-SYM permettant de détecter les symétries sous les contraintes de compétitions GGP. Par la suite, nous montrons expérimentalement que MAC-UCB-SYM surpasse l'ensemble des meilleures approches GGP et que la détection de symétries est primordiale afin de minimiser l'espace de recherche ce qui a permis à WoodStock de devenir champion international de *General Game Playing* en remportant la compétition *IGGPC* 2016.

Première partie

État de l’art

Chapitre 1

General Game Playing

Sommaire

1.1 Bases en théorie des jeux 7
1.1.1 Jeux en forme normale 8
1.1.2 Jeux en forme extensive 12
1.1.3 Jeux stochastiques 16
1.2 *General Game Playing* 18
1.2.1 *Game Description Langage* (GDL) 19
1.2.2 Compétition internationale : IGGPC 24
1.2.3 Game Description Langage with Incomplete Information (GDL-II) . . . 26
1.3 Différentes approches appliquées au GGP 32
1.3.1 Algorithmes de type Minimax 32
1.3.2 Heuristiques de recherche 35
1.3.3 Méthodes de Monte Carlo 37
1.3.4 Variantes et améliorations de UCT 40
1.3.5 Techniques de parallélisation de MCTS 43
1.3.6 Réseau de propositions ($propnet$) 48
1.3.7 Approches pour les jeux GDL-II 52
1.4 Conclusion 54

1.1 Bases en théorie des jeux

La théorie des jeux vise à étudier les comportements d'agents (ou joueurs) rationnels interagissant dans un environnement commun appelé « jeux ». Ses domaines d'application sont multiples, Économie, Sciences Politique, Biologie, Psychologie, Études des Langues et enfin l'Informatique.

Par agent rationnel, on parle ici d'agents cherchant à optimiser leurs préférences ; les agents peuvent être **coopératifs** (préférences compatibles), **compétitifs** (préférences incompatibles ou opposées), ou de manière générale coopératifs pour certains et compétitifs pour d'autres. Usuellement, la notion de préférences est modélisée par une fonction d'utilité (aussi nommée fonction de gain) visant à quantifier le degré de préférence d'un agent sur un ensemble de solutions possibles.

Définition 1.1 (Fonction d'utilité) *Soit l'ensemble fini S ordonné par le symbole de relation de préférence $\preceq$. Une utilité est un élément $e \in S$, représentant les motivations des joueurs. La fonction $u : S \to \mathbb{R}$ est dite fonction d'utilité si et seulement si $\forall i, j \in S : u(i) \leq u(j) \Leftrightarrow i \preceq j$.*

En théorie des jeux, la définition générale d'un jeu suppose que le jeu se termine en un temps fini. On parle alors de jeux finis. Autrement dit, tout jeu fini possède un état représentant le début d'un jeu et un nombre fini d'états désignant la fin d'un jeu au cours desquels les utilités de chaque joueur sont assignées respectivement. Dans ces jeux, le nombre de joueurs est fini ainsi que le nombre d'actions possibles pour chaque joueur impliquant un nombre fini d'états possibles.

Les combinaisons d'actions de chacun des agents peuvent représenter deux types de jeux : les **jeux simultanés** et les **jeux séquentiels** (aussi nommés « tour par tour »). Bien qu'en théorie, il est supposé que tous les joueurs agissent simultanément, dans un jeu simultané, chaque action prise individuellement provoque une modification de l'état du jeu. À l'opposé, un jeu séquentiel est un jeu au cours duquel, une seule action de la combinaison d'actions provoque une modification de l'état du jeu. On dit alors que les autres joueurs « ne font rien » souvent noté par l'action *noop*. La différence entre ces deux classes de jeux et que dans un jeu séquentiel, un joueur peut déterminer sa stratégie uniquement en fonction des actions précédentes des autres joueurs alors que dans un jeu simultané, il n'a pas conscience des actions des autres joueurs réalisées au même moment.

Il est possible de distinguer plusieurs autres classes de jeux en fonction des informations que chaque joueur perçoit de chaque état du jeu. On parle de **jeux à information complète** (ou jeux déterministes) si chaque joueur connaît l'ensemble des actions réalisées par tous les joueurs composant le jeu, lui permettant ainsi de déduire l'état dans lequel il se trouve. À l'opposé, si l'accès à l'ensemble des actions réalisées par chaque joueur n'est pas garanti, on parle alors de **jeux à information incomplète**. Les joueurs ne sont informés que partiellement des actions des autres joueurs et sont obligés d'estimer l'état dans lequel il se trouve afin de définir la stratégie la plus adaptée. Notons que les jeux simultanés peuvent être vus comme des jeux à information incomplète, sachant qu'un joueur n'a pas conscience des actions des autres joueurs au même tour.

Finalement, on parle des **jeux à information imparfaite** pour l'ensemble des jeux impliquant une notion de chance ou de hasard, souvent représentée par un joueur dit « environnement ». Par exemple, l'utilisation de dés, de pièces, etc.

1.1.1 Jeux en forme normale

La forme normale, également connue sous le nom de forme stratégique, est la représentation la plus couramment utilisée. Ici, un jeu équivaut à une représentation de l'utilité de chaque joueur dans chaque état composant le jeu.

Définition 1.2 (Jeu en forme normale) *Un jeu en forme normale est un tuple $(J, A, \boldsymbol{u})$ où :*

- *$J = \{1, 2, \ldots, k\}$ représente un ensemble de k joueurs, indicé par j ;*
- *$A = A_1 \times A_2 \times \ldots \times A_k$ où A_j est l'ensemble des actions pour chaque joueur j. Chaque vecteur $\boldsymbol{a} = (a_1, \ldots, a_k)$ est appelée une combinaison d'actions ;*
- *$\boldsymbol{u} = (u_1, \ldots, u_k)$ où $u_j : A \to \mathbb{R}$ est le résultat de la fonction d'utilité du joueur j ;*

Une façon naturelle permettant de représenter des jeux à 2 joueurs sous leur forme normale est d'utiliser une matrice de dimension k. En général, chaque ligne correspond à une action possible pour le joueur 1, chaque colonne correspond à une action possible pour le joueur 2 et chaque cellule correspond à la conséquence de la combinaisons des actions l'indiçant. L'utilité obtenue par chaque joueur est indiquée dans la cellule avec (u_1,u_2).

Exemple 1.1 *Le Matching pennies est un jeu simultané à deux joueurs. Chaque joueur possède une pièce. Au même moment, les deux joueurs retournent leur pièce sur $Pile$ ou $Face$. Une fois que les deux pièces sont retournées, le joueur 1 remporte le jeu si les deux pièces présentent le même coté, sinon le joueur 2 remporte la partie. Le tableau 1.1 représente la matrice de gain associée aux deux joueurs.*

1 \ 2	$Face$	$Pile$
$Face$	(1,-1)	(-1,1)
$Pile$	(-1,1)	(1,-1)

TABLE 1.1 – Matrice de gain du *Matching Pennies*

Notons que cette notion est primordiale car quelque soit la représentation utilisée d'un jeu à information complète, ce dernier peut être réduit en forme normale, bien que la matrice de gain peut être de taille exponentielle.

Il existe quelques classes de jeux en forme normale méritant une attention particulière. La première d'entre elles correspond aux jeux de coopération pure (originalement nommé *common-payoff game*). Dans ces jeux, quelque soit la séquence d'actions réalisées, tous les joueurs obtiennent le même gain.

Définition 1.3 (Jeu de coopération pure) *Un jeu de coopération pure est un jeu où pour toutes combinaisons d'actions $\boldsymbol{a} \in A_1 \times \ldots \times A_k$ et deux joueurs i, j alors $u_i(\boldsymbol{a}) = u_j(\boldsymbol{a})$.*

Remarque 1.1 *Dans la suite de ce manuscrit, cette classe de jeu sera plus simplement désignée par « jeu coopératif »*

À l'inverse des jeux coopératifs, viennent les jeux de compétitions pures, aussi nommés jeux à somme nulle. Dans cette classe de jeu, la somme des utilités de l'ensemble des joueurs est toujours égale à 0, on parle de jeux de compétitions pures car ici plus le gain d'un joueur est fort plus celui des autres joueurs est faible.

Définition 1.4 (Jeu à somme nulle) *Un jeu $(J, A, \boldsymbol{u})$ est dit à somme nulle si et seulement si $\boldsymbol{a} \in A_1 \times \ldots \times A_k \sum_{j=1}^{k} u_j(\boldsymbol{a}) = 0$.*

Remarque 1.2 *Notons qu'il est possible de généraliser les jeux à somme nulle par les jeux à somme constante. Dans ce contexte, la somme des utilités de chaque joueur est toujours égale à une même constante.*

Quelque soit le nombre de joueurs impliqués, un jeu évolue suite aux différentes séquences d'actions réalisées. Par conséquent, les gains obtenus par chaque joueur en fin de jeu ne dépendent pas uniquement du comportement choisi mais également des actions des autres joueurs. Ainsi, un agent rationnel cherche à optimiser son gain final via l'utilisation d'un plan d'actions spécifiant la réalisation du prochain mouvement en fonction de chacune des décisions entreprises par les autres agents. On parle alors de **stratégie**.

Une **stratégie** est dite **pure** si il n'existe qu'une seule action à réaliser en fonction de chaque combinaison d'actions réalisable par les autres joueurs. À l'opposé, si il existe une distribution de probabilité définie sur plusieurs actions pour décider du comportement à adopter selon la combinaison d'actions réalisées par les autres joueurs, la stratégie est dite mixe.

Définition 1.5 (Stratégie mixe) *Soit un jeu en forme normale (J, A, $\boldsymbol{u}$). On note $P(X)$ l'ensemble de toutes les distributions de probabilités sur l'ensemble X. Alors l'ensemble des stratégies mixes du joueur j est $St_j = P(A_j)$. L'ensemble des séquences de stratégies mixes est le produit cartésien de toutes les stratégies mixes $St_1 \times \ldots \times St_k$.*

Par $st_j(a_j)$ nous désignons la probabilité que l'action a_j soit jouée par la stratégie mixe st_j. Le sous-ensemble d'actions associée à une probabilité positive par la stratégie mixe st_j est appelé support de st_j.

Définition 1.6 (Support) *Le support d'une stratégie mixe st_j d'un joueur j est l'ensemble des stratégies pures $\{a_j \mid st_j(a_j) > 0\}$.*

Remarque 1.3 *Notons qu'une stratégie pure est un cas particulier d'une stratégie mixe où le support est un singleton.*

De manière évidente, il existe une infinité de stratégies pures pour chaque joueur, ce qui implique une infinité de stratégies mixes. En réalisant une stratégie mixe, un joueur peut obtenir une gamme variée d'utilités. Pour évaluer de telles stratégies, il est nécessaire d'introduire l'utilité attendue. Intuitivement, cette notion correspond à la moyenne pondérée des gains possibles par la réalisation de chaque action composant la stratégie mixe.

Définition 1.7 (Utilité attendue) *Soit un jeu en forme normale (J, A, $\boldsymbol{u}$), l'utilité attendue u_j du joueur j sur le vecteur de stratégies mixes $\boldsymbol{st} = (st_1, \ldots, st_k)$ est défini par :*

$$u_j(\boldsymbol{st}) = \sum_{\boldsymbol{a} \in A} u_j(\boldsymbol{a}) \prod_{j=1}^{k} st_j(a_j)$$

Remarque 1.4 *Dans la suite de ce manuscrit, si aucune précision n'est apparente, toute notion d'utilité fait référence à l'utilité attendue.*

À l'origine, la théorie des jeux a été établie afin de répondre à la question suivante : « Quelle est la réaction optimale d'un agent ? Comment un agent doit réagir pour avoir le plus de chance de gagner ? ».

Face à ce problème, la notion de « concept de solution » est utilisée en identifiant les sous-ensembles de gains atteignables par chaque joueur. L'un des concepts de solutions, le plus couramment utilisé est l'équilibre de Nash, en préférant le point de vue d'un agent à celui d'un spectateur extérieur au jeu.

Supposons qu'un agent connaisse l'ensemble des stratégies décidées par chacun des autres agents du jeu. Dans ce cadre, le problème de décision visant à optimiser l'utilité de l'agent en question devient simple. En effet, il suffit de réaliser la stratégie maximisant l'utilité attendue, on parle alors de « meilleure réaction » (*best response*). Afin de donner une définition formelle de cette notion, il est nécessaire d'introduire $\boldsymbol{st}_{-j} = (st_1, \ldots, st_{j-1}, st_{j+1}, \ldots, st_k)$ désignant le vecteur de stratégies $\boldsymbol{st}$ sans st_j. Autrement dit, $\boldsymbol{st} = (st_j, \boldsymbol{st}_{-j})$.

Définition 1.8 (Meilleure réaction) *La meilleure réaction d'un joueur j au vecteur de stratégie $\boldsymbol{st}_{-j}$ est la stratégie mixe $st_j^* \in St_j$ tel que $u_j(st_j^*, \boldsymbol{st}_{-j}) \geq u_j(st_j, \boldsymbol{st}_{-j})$ pour toute stratégie $st_j \in St_j$.*

Malheureusement, dans la plupart des cas, la connaissance des stratégies adverses est difficilement possible. Cependant, il est possible d'utiliser la notion de meilleure réaction pour définir un point central en théorie des jeux, un équilibre de Nash.

Définition 1.9 (Équilibre de Nash) *Un vecteur de stratégies $\boldsymbol{st} = (st_1 \ldots, st_k)$ est un équilibre de Nash si et seulement si pour tout agent j, st_j est la meilleure réaction pour $\boldsymbol{st}_{-j}$.*

Intuitivement, un équilibre de Nash n'avantage aucun changement de stratégie pour aucun joueur, la situation est dite stable.

Exemple 1.2 *Soit un jeu à deux joueurs où l'ensemble des stratégies du joueur* 1 *est* $St_1 = \{st_{1,1}, st_{1,2}\}$ *et l'ensemble des stratégies du joueur* 2 *est* $St_2 = \{st_{2,1}, st_{2,2}, st_{2,3}\}$. *Le tableau 1.2 représente l'utilité associée à chaque séquence de stratégies* St, *où* $u(St)$ = $(st_{1,x}, st_{2,y})$ *associées aux joueurs si le joueur* 1 *réalise la stratégie* x *et le joueur* 2 *la stratégie* y.

1 \ 2	$st_{2,1}$	$st_{2,2}$	$st_{2,3}$
$st_{1,1}$	(5,2)	(4,4)	(5,4)
$st_{1,2}$	(3,1)	(2,0)	**(6,2)**

TABLE 1.2 – Équilibre de Nash

Ici, le vecteur de stratégies $\boldsymbol{st} = (st_{1,2}, st_{2,3})$ *associée à* $u(\boldsymbol{st})$ = $(6, 2)$ *représente un équilibre de Nash car :*

— $(st_{1,1}, st_{2,3}) \leq (st_{1,2}, st_{2,3})$ *car* $(5 \leq 6)$ *;*
— $(st_{1,2}, st_{2,1}) \leq (st_{1,2}, st_{2,3})$ *car* $(1 \leq 2)$ *;*
— $(st_{1,2}, st_{2,2}) \leq (st_{1,2}, st_{2,3})$ *car* $(0 \leq 2)$.

Propriété 1.1 *Dans tout jeu fini multi-joueurs possédant un nombre fini de stratégies pures, il y a au moins un équilibre de Nash, bien qu'il puisse être constitué de stratégies mixes [*NASH *1951].*

Exemple 1.3 *Reprenons, le jeu du Matching Pennies illustré dans l'exemple 1.1. Il est évident qu'il n'existe pas d'équilibre de Nash composé uniquement de stratégies pures car les gains obtenus par chaque joueur sont opposés. Toutefois, il existe un équilibre de Nash composé uniquement de stratégies mixes si chaque joueur décide de réaliser la stratégie mixe consistant à choisir le coté d'une pièce via une probabilité de* $\frac{1}{2}$.

Évidemment, chaque agent ne peut être sûr de la stratégie de chacun des autres joueurs. Il est alors possible d'interpréter les stratégies mixes sur la notion d'incertitude rencontrée par chaque agent : la stratégie mixe du joueur j est déterminée par l'évaluation de la probabilité que chacun des autres joueurs décide de jouer chacune des stratégies possibles. À l'équilibre, toutes les actions du support la stratégie mixe du joueur j sont les meilleures réactions en fonction de son évaluation des stratégies adverses.

Bien que le concept de solutions le plus couramment utilisé est l'équilibre de Nash, d'autres concepts existent. Par la suite nous présentons $maxmin$ et $minmax$ deux concepts plus restrictifs.

La stratégie $maxmin$ du joueur j d'un jeu à k joueurs à somme constante est une stratégie qui maximise le gain de j dans le pire cas, c'est à dire une situation où tous les autres joueurs décident de réaliser une stratégie minimisant le gain de j. La valeur $maxmin$ pour le joueur j est le gain minimum garanti via la stratégie $maxmin$.

Définition 1.10 (Maxmin) *La stratégie* $maxmin$ *du joueur* j *est* $arg\ max_{st_j}\ min_{\boldsymbol{st}_{-j}} u_j(st_j, \boldsymbol{st}_{-j})$ *et la valeur* $maxmin$ *pour le joueur* j *est* $max_{st_j}\ min_{\boldsymbol{st}_{-j}} u_j(st_j, \boldsymbol{st}_{-j})$.

La stratégie $maxmin$ est un concept qui prend tout son sens dans les jeux simultanés mais également dans les jeux séquentiels. En effet, il s'agit du meilleur choix de stratégie à adopter pour optimiser le gain du joueur j dans l'hypothèse où chacun des joueurs adverses $-j$ (rationnels) en décide autant provoquant

ainsi la minimisation du gain de j. De plus, si jamais un autre joueur décide de ne pas minimiser le gain de j, alors le gain obtenu sera au minimum celui qu'apporte la stratégie $maxmin$.

La stratégie $minmax$ et la valeur $minmax$ associée représentent le concept de solution inverse de $maxmin$. En effet, dans les jeux à deux joueurs, la stratégie $minmax$ pour le joueur j contre tous les autres joueurs $-j$ est une stratégie qui maintient le gain maximum de $-j$ à son minimum et la valeur $minmax$ de $-j$ à son minimum. Un exemple d'utilisation est lorsqu'un agent souhaite minimiser le gain d'un autre joueur de par la maximisation du gain d'un troisième joueur sans modifier son propre gain.

Définition 1.11 (Minmax) *Dans un jeu à k joueurs, la stratégie $minmax$ d'un joueur i contre chaque joueur $j \neq i$ est le membre i du vecteur de stratégies mixes $\boldsymbol{st}_{-j}$ dans $arg\, min_{\boldsymbol{st}_{-j}}\, max_{st_j}\, u_j(st_j, \boldsymbol{st}_{-j})$ où $-j$ désigne l'ensemble des joueurs différents de j. La valeur $minmax$ de j est $min_{\boldsymbol{st}_{-j}}\, max_{st_j}\, u_j(st_j, \boldsymbol{st}_{-j})$.*

Dans les jeux à deux joueurs, la valeur $minmax$ d'un joueur est toujours égale à sa valeur $maxmin$. Cette relation est à l'origine de la propriété de von Neumann [VON NEUMANN 1928].

Propriété 1.2 *Dans tout jeu fini à deux joueurs et à somme nulle, les concepts de solutions d'équilibre de Nash, de stratégie $maxmin$ et de stratégie $minmax$ sont équivalents : Pour chaque joueur, l'utilité attendue pour un équilibre de Nash est celle de la valeur $maxmin$ qui est aussi celle de la valeur $minmax$.*

Exemple 1.4 *Pour l'exemple du Matching Pennies illustré dans l'exemple 1.1, les valeurs $minmax$ et $maxmin$ sont égales à 0. Comme vu précédent, l'unique équilibre de Nash consiste à choisir aléatoirement entre $Pile$ et $Face$ avec une probabilité identique. Cette stratégie est identique pour tous les joueurs si l'on choisit la stratégie $minmax$ ou $maxmin$.*

Toutefois, pour les jeux avec plus de deux joueurs, il peut exister plusieurs équilibres de Nash où la valeur $maxmin$ d'un joueur est différente de sa valeur $minmax$.

Le dernier concept de solution illustré dans cette partie reflète l'idée que chacun des joueurs pourrait ne pas changer de stratégie afin de réaliser la meilleure réaction lorsque le gain potentiel à gagner est très faible. Ce concept s'intitule un équilibre epsilon-Nash (ϵ-Nash).

Définition 1.12 (Un équilibre de ϵ-Nash) *Soit $\epsilon > 0$. Un vecteur de stratégies $\boldsymbol{st} = (s_1, \ldots, s_k)$ est un équilibre de ϵ-Nash si pour tout agent j et pour toute stratégie $st'_j \neq st_j$, $u_j(st_j, \boldsymbol{st}_{-j}) \geq u_j(st'_j, \boldsymbol{st}_{-j}) - \epsilon$.*

Ce concept possède plusieurs propriétés intéressantes. Un ϵ-équilibre de Nash existe toujours ; en effet, tout équilibre de Nash est encadré par une succession d'ϵ-équilibre de Nash pour tout $\epsilon > 0$. De plus, l'argument selon lequel un joueur peut être indifférent à l'obtention d'un gain quasi nul est intéressant, notamment suite à la représentation approximée de nombres réels en informatique. Enfin, cette notion est importante pour les algorithmes d'apprentissage souvent utilisés en algorithmique des jeux.

1.1.2 Jeux en forme extensive

En théorie des jeux, il existe de nombreuses représentations d'un jeu. Précédemment, nous avons illustré, via la définition 1.2, la forme normale. Toutefois, bien qu'elle convienne aux notions mathématiques, elle n'est pas très appropriée pour les exemples concrets car la matrice de gain associée peut être de taille exponentielle. Une autre représentation souvent utilisée est la forme extensive permettant de rendre le cadre temporel explicite à la modélisation du jeu et ainsi de modéliser tout jeu séquentiel

aisément. Nous commençons par mettre en avant la forme extensive des jeux à information complète puis celle des jeux à information incomplète. Notons F l'ensemble des fluents d'un jeu où un **fluent** désigne une entité ou un objet composant le jeu.

Définition 1.13 (Jeu à information complète en forme extensive) *Un jeu à information complète en forme extensive est un tuple* $(J, A, S, S_{ter}, L, d, succ, \boldsymbol{u})$*, où :*

- $J = \{1, 2, \ldots, k\}$ *représente l'ensemble des* k *joueurs du jeu ;*
- A *est l'ensemble des actions possibles ;*
- $S = 2^F$ *est l'ensemble des états, on distingue par* $S_{ter} \subseteq S$ *l'ensemble des états terminaux (on note* S_{-ter} *l'ensemble des états non terminaux) ;*
- $L : S_{-ter} \to 2^A$ *désigne l'ensemble des actions légales à partir de chaque état non terminal ;*
- $d : S_{-ter} \to J$ *est une fonction qui à partir de chaque état non terminal désigne les joueurs décidant de leurs actions ;*
- $succ : S_{-ter} \times A \to S$ *est une fonction de succession qui fait correspondre un état non terminal et une action à un nouvel état (terminal ou non) tel que pour* $s_1, s_2 \in S_{-ter}$ *et pour tout* $a_1, a_2 \in A$*, si on a* $succ(s_1, a_1) = succ(s_2, a_2)$ *alors* $s_1 = s_2$ *et* $a_1 = a_2$ *;*
- $\boldsymbol{u} = (u_1, \ldots, u_k)$ *est un vecteur de fonctions d'utilité pour tout joueur* j *sur l'ensemble des états terminaux* S_{ter} *telle que* $u : S_{ter} \to \mathbb{R}$*.*

Comme précédemment, nos définitions sont restreintes aux jeux finis représentés par des « arbres de jeux ». Un arbre de jeu (*game tree*) aussi nommé arbre de recherche (*search tree*) est composé de nœuds représentants les états du jeu. Chaque nœud est relié à ses successeurs par une branche correspondant à l'union des actions possibles de chaque joueur dans l'état correspondant. L'état initial du jeu est la racine de l'arbre et les états terminant le jeu correspondent aux feuilles. On associe à chaque feuille de l'arbre une fonction d'utilité retournant l'utilité de chaque joueur dans cet état de jeu.

Exemple 1.5 *La figure 1.1 illustre l'arbre de recherche d'un jeu à un joueur composé de sept états nommés* $s_1, \ldots, s_7$*. L'état initial est* s_1 *(ici pointé par une flèche) et les états* s_4*,* s_5*,* s_6 *et* s_7 *sont des états terminaux (colorés pour l'exemple) étiquetés par l'utilité obtenue par le joueur. Chaque branche représente l'action réalisée par le joueur entre deux états. Par exemple, si l'état courant est* s_1 *et que le joueur réalise l'action* a*, le nouvel état courant sera* s_2 *mais si il réalise l'action* b*, l'état courant deviendra* s_3*. Ainsi, on comprend que la suite d'actions permettant d'atteindre l'utilité maximale (un score de 100) à partir de l'état initial est* $\{(b), (a)\}$*.*

La figure 1.2 représente l'arbre de recherche d'un jeu à deux joueurs composé également de sept états portant les mêmes noms. L'état initial et les états terminaux sont les mêmes. Chaque branche représente un couple d'actions possibles, le premier membre illustre l'action du joueur 1 *et le second l'action du joueur* 2*. Les utilités étiquetées sous les états terminaux suivent la même logique en indiquant d'abord le score du joueur* 1 *puis du joueur* 2*.*

Remarque 1.5 *Bien que différents nœuds correspondent souvent à des états différents, il est possible que différents nœuds correspondent à un même état (ce qui arrive quand plusieurs combinaisons d'actions provoquent l'apparition d'un même état du jeu). De ce fait, un graphe d'états permet de représenter de manière plus dense la structure d'un jeu, toutefois, dans le cadre de nos travaux, il est graphiquement plus représentatif d'illustrer cette structure par un arbre de recherche.*

Une stratégie pure pour un joueur dans un jeu à information complète est une spécification complète de toute action à réaliser quelque soit l'état dans lequel se trouve le joueur.

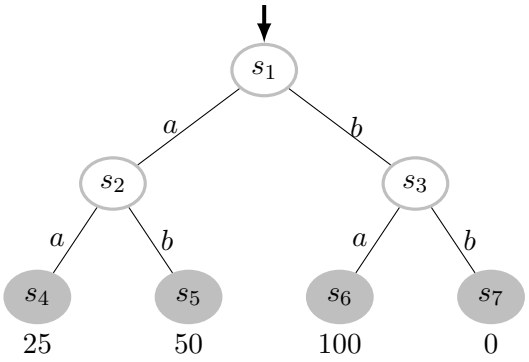


FIGURE 1.1 – Un arbre de recherche d'un jeu à un joueur.

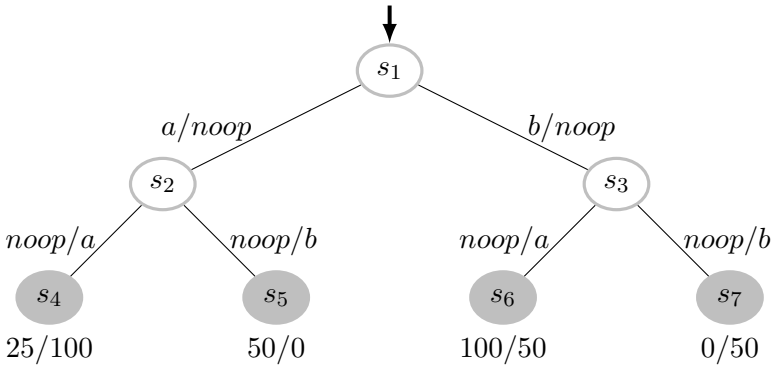


FIGURE 1.2 – Un arbre de recherche d'un jeu à deux joueurs.

Définition 1.14 (Stratégie pure) *Soit* $(J, A, S, S_{ter}, L, d, succ, \boldsymbol{u})$ *un jeu à information complète en forme extensive, la stratégie pure d'un joueur* j *est représentée par le produit cartésien suivant :*

$$\prod_{s \in S_{-ter}, d(s)=j} L(s)$$

Propriété 1.3 *Tout jeu en forme extensive possède un équilibre de Nash défini uniquement par des stratégies pures.*

Notons également que la définition de la meilleure réaction et d'un équilibre de Nash est la même dans un jeu en forme normale ou en forme extensive.

Les arbres de recherche ont originalement été dédiés pour les jeux séquentiels à information complète mais peuvent facilement être étendus aux jeux à information incomplète. Un jeu à information incomplète en forme extensive est représenté par un arbre de jeu où chaque nœud de l'arbre est partitionné en ensemble d'informations ; intuitivement, si deux nœuds sont dans le même ensemble d'informations alors le joueur ne peut les distinguer.

Définition 1.15 (Jeu à information incomplète en forme extensive) *Un jeu à information incomplète en forme extensive est un tuple* $(J, A, S, S_{ter}, L, d, succ, \boldsymbol{u}, I)$*, où :*

- $(J, A, S, S_{ter}, L, d, succ, \boldsymbol{u})$ *représente un jeu à information complète ;*
- $I = \{I_1, \ldots, I_k\}$ *où* $I_j = (I_{j,1}, \ldots, I_{j,k_j})$ *est un ensemble de classes équivalentes sur (i.e. une partition de)* $\{s \in S_{-ter} : d(s) = j\}$ *où la propriété* $L(s) = L(s')$ *et* $d(s) = d(s')$ *est vérifiée dès qu'il existe un* i *pour chaque* $s \in I_{j,i}$ *et* $s' \in I_{j,i}$*.*

Autrement dit, pour un agent un ensemble d'informations correspond à un ensemble d'états potentiels dans lequel ce dernier peut se trouver en fonction des informations partielles qu'il possède de l'état courant et où les mêmes actions sont possibles dans chaque état.

Exemple 1.6 *La figure 1.3 représente l'arbre de recherche d'un jeu à un joueur impliquant un joueur « environnement » représentant le lancer d'une pièce. L'état initial est symbolisé par un losange correspondant à l'action de l'environnement. Les états* s_1 *et* s_2 *sont atteignables selon le choix de l'environnement. De ce fait, ils font partie du même ensemble d'informations, symbolisé ici par un rectangle. Si l'environnement choisit* $pile$ *alors l'action* $'a'$ *du joueur lui permet d'atteindre l'état terminal le plus favorable (associée à une utilité de* 100*). Toutefois, si l'environnement choisit* $face$ *alors c'est l'action* $'b'$ *qui permet d'atteindre l'état le plus favorable.*

La notion de stratégie pure peut s'étendre aux jeux à information incomplète.

Définition 1.16 (Stratégie pure (à information incomplète)) *Soit* $(J, A, S, S_{ter}, L, d, succ, \boldsymbol{u}, I)$ *un jeu à information incomplète en forme extensive, la stratégie pure d'un joueur* j *est représentée par le produit cartésien suivant :*

$$\prod_{I_{j,i} \in I_j} L(I_{j,i})$$

Notons que la définition d'une stratégie pure pour un jeu à information complète peut alors être vue comme un cas particulier d'une stratégie pure à information incomplète où chaque ensemble d'informations est un singleton. Ainsi, la formalisation d'une **stratégie mixe** découle naturellement de la définition d'une stratégie pure en ajoutant une distribution de probabilités sur un vecteur de stratégies pures. De plus, dans le cadre de jeu à information incomplète, une stratégie mixe peut être étendue en **stratégie**

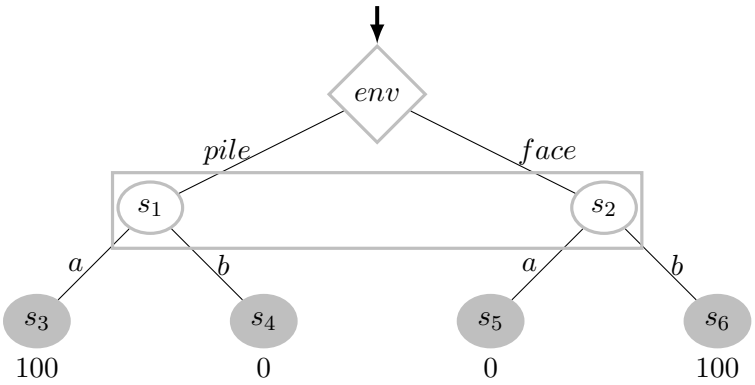


FIGURE 1.3 – Un arbre de recherche d'un jeu à information incomplète et imparfaite.

comportementale en ajoutant une probabilité sur chaque ensemble d'informations indépendante de la probabilité de réaliser chaque stratégie pure.

En général, une stratégie mixe et une stratégie comportementale sont incomparables, du fait que dans certains jeux, le score maximal ne peut être atteint que par l'une ou l'autre des deux stratégies. Cependant, il existe une classe importante de jeux à information incomplète avec mémoire parfaite (*perfect recall*) où ces deux stratégies coïncident. Comme le nom de cette classe de jeux le laisse à penser, ici aucun joueur ne peut oublier la moindre information apprise suite à la réalisation d'une action et chaque agent peut répertorier l'ensemble des mouvements qu'il a pratiqué par le passé.

Définition 1.17 (Mémoire parfaite) *Un joueur j possède une mémoire parfaite dans un jeu à information incomplète si pour tout couple d'états (s,s′) dans le même ensemble d'informations du joueur j et pour tout chemin $s_0, \boldsymbol{a}_0, \ldots, s_m, \boldsymbol{a}_m, \ldots, s$ à partir de l'état initial du jeu jusqu'à l'état s (où s_i représente les nœuds et $\boldsymbol{a}_i$ les actions) et pour tout chemin $s'_0, \boldsymbol{a}'_0, \ldots, s'_m, \boldsymbol{a}'_m, \ldots, s'$ de l'état initial à l'état s' on a :*

- *$s = s'$;*
- *Pour tout $0 \leq i \leq m$ si $d(s_i) = j$ (où s_i représente un nœud de décision du joueur j) alors s_i et s'_i appartiennent à la même classe d'équivalence pour j ;*
- *Pour tout $0 \leq i \leq m$ si $d(s_i) = j$ alors $\boldsymbol{a}_i = \boldsymbol{a}'_i$.*

Il s'agit d'un jeu à mémoire parfaite si chaque joueur qui le compose possède une mémoire parfaite.

Clairement, tout jeu à information complète est un jeu à mémoire parfaite.

Propriété 1.4 *Dans un jeu à mémoire parfaite, toute stratégie mixe d'un joueur donné peut être remplacée par une stratégie comportementale équivalente et vice-versa. Ici, deux stratégies sont équivalentes si les mêmes probabilités sont appliquées sur chaque score atteignable pour chaque joueur par les deux stratégies.*

Concluons cette partie, en indiquant que l'ensemble des équilibres de Nash reste inchangé pour toute stratégie comportementale dans les jeux à mémoire parfaite.

1.1.3 Jeux stochastiques

Au cours de cette section, nous mettons en avant une troisième et dernière représentation utilisée en théorie des jeux, « les jeux stochastiques » aussi connues sous le nom de « jeux de Markov ». Ce cadre

est intéressant car il permet de modéliser les jeux où les agents jouent à plusieurs reprises à plusieurs jeux en forme normale et où le jeu joué au temps t est soumis à une distribution de probabilités sur les actions prises par l'ensemble des agents au cours du jeu pratiqué au temps $t-1$.

Définition 1.18 (Jeu stochastique (ou jeu de Markov)) *Un jeu stochastique G est un tuple $(s_0, S_{ter}, L, P, \boldsymbol{u})$ définit sur un ensemble J de k joueurs, un ensemble de fluents F et un ensemble d'actions possibles A où :*

- *$s_0 \in 2^F$ est l'état initial ;*
- *$S_{ter} \subseteq 2^F$ est l'ensemble des états terminaux ;*
- *$L : J \times 2^F \to 2^A$ assigne chaque joueur j et chaque état s à l'ensemble $L_j(s)$ représentant les actions légales de j dans s ;*
- *$P : 2^F \times A \times 2^F \to [0,1]$ assigne chaque état s et chaque séquence d'actions $\boldsymbol{a} \in \boldsymbol{L}(s)$ à une distribution de probabilités $P(\cdot \mid s, \boldsymbol{a})$ sur S, où $\boldsymbol{L}(s) = L_1(s) \times \ldots L_k(s)$;*
- *$\boldsymbol{u} : J \times S_{ter} \to \mathbb{R}$ assigne chaque joueur j et chaque état terminal s à une utilité $u_j(s)$.*

Une séquence de jeu dans G est une séquence infinie $(s_0, \boldsymbol{a}_1, s_1, \boldsymbol{a_2}, \ldots)$, où $\boldsymbol{a}_t \in \boldsymbol{L}(s_{t-1})$ et $P(s_t \mid s_{t-1}, \boldsymbol{a}_t) > 0$, pour chaque tour $t \in \{1, 2, \ldots\}$. L'ensemble de tous les états dans G atteignables depuis s_0 en utilisant différentes séquences de jeu dans G sont désignés par S_G. On dit qu'un jeu G est jouable si $S_{ter} \subseteq S_G$.

Un historique de longueur T dans G est la suite finie $\boldsymbol{h} = (s_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_{T-1}, s_T)$ de séquences de jeu dans G. $\boldsymbol{h}$ est dit complet si s_T est le seul état terminal de la séquence. Dans ce cas, nous utilisons $u_j(\boldsymbol{h})$ pour désigner $u_j(s_T)$ et $P(\boldsymbol{h})$ pour représenter $\prod_{t=1}^{T} P(s_t \mid s_{t-1}, \boldsymbol{a}_t)$.

Ainsi, G est un jeu à horizon T si il est jouable et que chaque séquence de jeu dans G inclue un historique complet de taille au plus T. Chaque état terminal $s \in S_{ter}$ est inclus dans plusieurs historiques complets dans G de taille au plus T. Notons que tout jeu stochastique G à horizon T peut aussi être représenté en forme extensive désignant un arbre sur l'ensemble des historiques complets de G chacun associé à sa propre distribution de probabilités et à son utilité.

Définissons maintenant les stratégies usuelles d'un joueur dans un jeu stochastique. Soit $\boldsymbol{h}_t$ l'historique des t niveaux d'un jeu stochastique et soit H_t l'ensemble de tous les historiques possibles de cette taille. L'ensemble des stratégies déterministes est le produit cartésien $\prod_{t,H_t} A_i$ nécessitant une décision dans chaque historique possible à chaque temps t.

Dans l'ensemble des représentations vues précédemment, la stratégie d'un agent est souvent définie via plusieurs stratégies déterministes. Toutefois, dans ce cadre, les stratégies comportementales sont définies indépendamment sur chaque historique.

Définition 1.19 (Stratégie comportementale) *Une stratégie comportementale $st_j(h_t, a_j)$ retourne la probabilité de réaliser l'action a_j dans l'historique h_t*

Une stratégie de Markov apporte une restriction supplémentaire aux stratégies comportementales où à chaque temps t, la distribution de probabilités sur les actions ne dépend que de l'état courant.

Définition 1.20 (Stratégie de Markov) *Une stratégie de Markov st_j est une stratégie comportementale dans laquelle $st_j(h_t, a_j) = st_j(h'_t, a_j)$ si $s_t = s'_t$, où s_t et s'_t sont des états terminaux de h_t et h'_t respectivement.*

Une dernière restriction possible est de supprimer la dépendance temporelle, comme le réalisent les stratégies stationnaires.

Définition 1.21 (Stratégie stationnaire) *Une stratégie stationnaire st_t est une stratégie de Markov dans laquelle $st_t(h_{t,1}, a_j) = st_j(h'_{t,2}, a_j)$ si $s_{t,1} = s'_{t,2}$ où $s_{t,1}$ et $s'_{t,2}$ sont les états terminaux de $h_{t,1}$ et $h'_{t,2}$ respectivement.*

Pour finir, un jeu stochastique à information partielle (POSG pour *Partially observable stochastic game*) comprenant des coups légaux permet de décrire un jeu stochastique multijoueurs avec information incomplète.

Définition 1.22 (Jeu stochastique à information partielle (POSG)) *Pour un ensemble fini S, soit Δ_S signifiant la probabilité sur S, c'est à dire, l'ensemble de toutes les distributions de probabilités sur S. Un jeu stochastique à information partielle est un tuple $G = (J, S, s_0, S_{ter}, A, L, P, B, \boldsymbol{u})$ où :*

- *J est l'ensemble fini $\{1, 2, \ldots, k\}$ de tous les joueurs ;*
- *$S = 2^F$ est l'ensemble fini des états incluant l'état initial s_0, et un sous ensemble S_{ter} d'états terminaux ;*
- *A est l'ensemble fini des actions ;*
- *$L : J \times S \to 2^A$ définit l'ensemble des actions légales $L_j(s)$ du joueur j dans l'état s ; Nous supposons $L_j(s) = \emptyset$ pour tout $s \in S_{ter}$;*
- *$P : S \times A^k \hookrightarrow \Delta_S$ est la fonction de transition partielle de probabilité, qui attribue à chaque état $s \in S$ et à chaque combinaison d'actions $\boldsymbol{a} \in \boldsymbol{L}(s)$, une distribution de probabilités sur S ;*
- *$B : J \times S \to \Delta_S$ est la fonction de croyance qui attribue chaque joueur j et chaque état $s \in S$ à une distribution de probabilités $B_p(s)$ sur S, représentant l'état de croyance de p à s ;*
- *$\boldsymbol{u} : J \times S_{ter} \to [0, 1]$ est la fonction d'utilité qui attribue à chaque joueur j et chaque état terminal $s \in S_{ter}$ une valeur $u_j(s) \in [0, 1]$, représentant l'utilité de j dans s.*

Suite à la présentation des différentes notions issues de la théorie des jeux au cours de cette section, il est maintenant possible de s'intéresser au contexte principal de ce manuscrit, le *General Game Playing*.

1.2 *General Game Playing*

Le *General Game Playing* (ou souvent nommé GGP) a pris toute son ampleur en 2005 au travers d'un projet initié par le *Stanford Logic Group* [3] [GENESERETH & LOVE 2005]. Il représente un des défis majeurs de l'intelligence artificielle ayant pour but la réalisation de programmes-joueurs génériques capables de jouer efficacement sans intervention humaine. Ainsi l'expertise d'un tel programme ne dépend pas de l'expertise de son concepteur mais de l'association des meilleures techniques des différents thèmes de recherche qui composent l'intelligence artificielle (représentation des connaissances, apprentissage automatique, prise de décision rationnelle, etc).

De plus, GGP intègre la modélisation d'un grand nombre de différents types de jeux [SCHREIBER 2014]. Ainsi, un programme-joueur doit être capable de jouer à : des jeux à information complète ou incomplète, des jeux déterministes ou avec intervention du hasard, des jeux à 1 ou k joueurs, des jeux simultanés ou tour par tour, des jeux avec ou sans communication entre les joueurs, etc. Notons que la description de GGP par [GENESERETH & LOVE 2005] n'englobe pas tous les jeux, par exemple, les jeux-vidéo. Pour ce dernier, un autre thème de recherche existe décrit par [SCHAUL 2013] que nous détaillerons pas ici, car il sort du cadre de nos travaux.

La grande variété de problèmes considérés par [GENESERETH & LOVE 2005] partage une structure abstraite commune. Chaque jeu implique un nombre fini de joueurs et un nombre fini d'états, incluant un état distinct initial et un ou plusieurs états terminaux. À chaque tour de jeu, chaque joueur possède un nombre fini d'actions (aussi nommé actions légales). L'état courant du jeu est mis à jour par les conséquences simultanées de l'action de chaque joueur (qui peut être *noop* désignant l'action de ne rien faire). Le jeu commence avec un état initial et après un nombre fini de tours se termine sur un état terminal au cours duquel un score compris entre 0 et 100 est attribué à chaque joueur.

3. Stanford Logic Group : `http://games.stanford.edu/`

Comme vu dans la section précédente, il est utile de représenter les jeux par des arbres de recherche, malheureusement ces arbres peuvent être très larges suite à une évolution exponentielle du nombre d'états possibles à chaque tour. Par exemple, un jeu d'échecs possède plusieurs milliers de coups possibles et plus de 10^{80} états.

1.2.1 *Game Description Langage* (GDL)

Pour répondre à ce problème, [LOVE *et al.* 2006] propose le langage *Game Description Language* (très souvent abrégé GDL) permettant d'encoder les jeux à information complète dans une forme plus compacte que leur représentation directe par un arbre de recherche. GDL propose de modéliser chaque état du jeu en utilisant la logique du premier ordre pour définir les différents composants du jeu à modéliser (actions légales, état initial et terminal, évolution du jeu, etc). [THIELSCHER 2011a] montre que GDL est un langage suffisamment générique pour décrire des jeux de nombreux types.

Avant toute chose, il est utile de définir le vocabulaire de la logique du premier ordre et les propriétés nécessaires à un programme GDL.

Programmation logique du premier ordre et propriétés associées

Définition 1.23 (Langage du premier ordre) *Un langage du premier ordre* $(\{c_1, c_2, \ldots, c_k\}, \{p_1, p_2 \ldots, p_n\})$ *est formé d'un ensemble de constantes :* $\{c_1, c_2, \ldots, c_k\}$ *et d'un ensemble de symboles de relations, aussi nommés prédicats :* $\{p_1, p_2, \ldots, p_n\}$.

À chaque prédicat p_i *est associé un entier, nommé arité, qui fixe son nombre d'arguments. Si l'arité d'un prédicat* p *est égale à :* 1*,* p *est dit unaire ;* 2*,* p *est dit binaire ;* $n \geq 3$*,* p *est dit* n*-aire.*

Un programme logique est un ensemble fini de faits et de théorèmes sur ces faits. Les faits sont également parfois appelés fluents. Un langage logique du premier ordre définit les faits par des atomes, des termes et des littéraux quant aux théorèmes, ils sont définis par des clauses de Horn (ou plus simplement règles).

Un terme permet de désigner une entité (ou objet). Il est défini sur un ensemble de symboles de fonctions et de constantes :

— Si X est une variable alors X est un terme ;
— Si c est une constante alors c est un terme ;
— Si f est un symbole de fonction et $t_1, \ldots, t_n$ sont des termes alors $f(t_1, \ldots, t_n)$ est un terme.

Remarque 1.6 *Au cours de ce manuscrit les constantes débutent par des minuscules (ou des entiers) contrairement aux variables où la première lettre est une majuscule. Les fonctions de termes sont désignées par leurs noms suivies des termes qui la composent entre parenthèses. Par exemple,* $cell(X, Y, b)$ *désigne la fonction de termes « cell » composée des termes* X*,* Y*,* b*. Les variables de cette fonction sont* X *et* Y *et* b *représente une constante. Il est à noter qu'une composition de fonctions de termes est une fonction de termes, par exemple,* $board(cell(X, Y, b), cell(W, Z, b))$ *est une fonction de termes.*

Un atome est défini sur un ensemble de symboles de prédicats. Si p est un prédicat et $t_1, \ldots, t_n$ sont des termes alors $p(t_1, \ldots, t_n)$ est un atome.

Remarque 1.7 *Dans ce manuscrit, l'écriture d'un atome est la même que pour une fonction de termes. Par exemple si* p *est un prédicat, si* f *est une fonction de termes,* a *une constante et* Y *une variable alors* $r(a, Y)$ *et* $r(a, f(a, Y))$ *sont des atomes. Bien que les fonctions de termes peuvent contenir d'autres fonctions de termes, un atome, lui, ne peut contenir aucun autre atome et ne peut être contenu dans une fonction de termes.*

Un littéral est un atome ou une négation d'atome. Un atome est appelé un littéral positif et la négation d'un atome un littéral négatif. Une clause de Horn (ou règle) est une expression distinguant sa « tête » par un atome et son « corps » par une conjonction de 0 ou plusieurs littéraux.

Exemple 1.7

$$q(X, Y) \leftarrow p(X, T), \neg r(Y)$$

est une règle dont l'atome $q(X, Y)$ est la tête et les littéraux $p(X, T)$ et $\neg r(Y)$ composent le corps de la règle. Ici $\neg r(Y)$ est un littéral négatif.

Une notion importante en programmation logique est la **dérivation logique**. Supposons un ensemble d'atomes $a_1, \ldots, a_n$ que nous appelons prémisses et un atome a_c que nous appelons conclusion. Supposons que l'on désire montrer que a_c peut être dérivé de $a_1, \ldots, a_n$; on notera cette intention $a_1, \ldots, a_n \vdash a_c$, cette dernière expression est appelé un séquent. La notion de dérivation syntaxique est formalisée à l'aide de règles de déduction. Elle est intimement liée à la notion sémantique de conséquence logique ($\models$), en effet, on aura : $a_1, \ldots, a_n \vdash a_c$ si et seulement si $a_1, \ldots, a_n \models a_c$.

Définition 1.24 (L'univers de *Herbrand*) *L'univers de Herbrand d'un programme logique est l'ensemble de tous les termes pouvant être formés par les constantes du programme. Dit autrement, c'est l'ensemble de toutes les constantes et de toutes les fonctions de termes de la forme $f(t_1, \ldots, t_n)$ où f est une fonction n-aire et $t_1, \ldots, t_n$ sont des éléments de l'univers de Herbrand.*

Définition 1.25 (Base de *Herbrand*) *La base de Herbrand d'un programme logique est l'ensemble de tous les atomes pouvant être formés par les constantes du programme. Dit autrement, c'est l'ensemble de toutes les séquences de la forme $p(t_1, \ldots, t_n)$ où p est un prédicat n-aire et $t_1, \ldots, t_n$ sont des éléments de l'univers de Herbrand.*

Exemple 1.8 *Soit un jeu G composé des termes a,b et des prédicats p,q. La base de Herbrand est $\{p(a), p(b), q(a, a), q(a, b), q(b, a), q(b, b)\}$.*

Ces deux dernières définitions sont fondamentales et permettent d'introduire plusieurs notions couramment utilisées en programmation logique. On parle d'un **ensemble de données** d'un programme logique P comme un sous-ensemble de la base de *Herbrand* de P et de l'**interprétation d'une règle** de P comme une règle où chaque occurrence d'une même variable est remplacée par un même terme de l'univers de *Herbrand* du programme P.

Soit un ensemble de données D. Une interprétation I satisfait une règle r si et seulement si I satisfait chaque littéral positif et négatif du corps de r. I satisfait un littéral positif t si $t \in D$ et I satisfait un littéral négatif $\neg t$ si $t \notin D$. On dit alors qu'un ensemble de données D satisfait un programme logique P si et seulement si il existe au moins une interprétation satisfaisante pour chaque règle. D est alors désigné comme un **modèle du programme logique** P. Généralement, un programme logique peut avoir plusieurs modèles, et donc naturellement plusieurs ensembles de données satisfaisants un programme P. D est **minimal** si et seulement il n'existe aucun sous-ensemble de D qui est un modèle de P.

Une règle est dite **permise** si et seulement si chaque variable apparaissant dans la tête de la règle ou dans un littéral négatif composant le corps de la règle apparaît au moins une fois dans un littéral positif appartenant au corps de la règle. Toute règle permise est décidable. Dit autrement, il est possible pour toute interprétation d'une règle de savoir si cette interprétation est satisfaisante ou non. Un programme logique est dit permis si toutes les règles qui le composent sont permises.

On associe à un programme logique un **graphe de dépendance**. Le graphe de dépendance associé à un programme logique correspond à un graphe orienté et étiqueté dans lequel les nœuds représentent

les prédicats du programme et où un arc relie 2 sommets T et C si et seulement si T est la tête d'une règle où C appartient au corps de cette même règle. L'étiquette « - » est associée à l'arc (T, C) si la règle contenant ces deux littéraux est de la forme :

$$T \leftarrow \ldots, not(C), \ldots.$$

Sinon l'étiquette associée est « + ». Un programme logique est dit **récursif** si il existe un circuit dans le graphe de dépendance associé au programme.

Un programme logique P est dit **stratifié** si et seulement si le graphe de dépendance de P ne contient pas de cycle impliquant un arc étiqueté par « - ». Tout programme logique statifié garantit sa **terminaison**. Dit autrement, un programme logique stratifié a un nombre fini de strates (de niveaux). Un programme logique $\mathcal{P}$ ne possède qu'un et seulement un modèle minimal si $\mathcal{P}$ est permis et stratifié.

Définition 1.26 (`GDL` valide) *Un programme `GDL` est dit valide si il est permis et stratifié.*

Remarque 1.8 *Au sein du General Game Playing, seuls les programmes `GDL` valides sont considérés. De ce fait, dans la suite de ce manuscrit, nous nous intéresserons uniquement aux programmes `GDL` valides.*

`GDL` est un langage logique purement déclaratif du premier ordre similaire à *Prolog* sans la notion d'arithmétique. Dans un programme `GDL`, les entités (les objets) qui composent le jeu sont décrites par des termes et les propriétés sur ces entités sont décrites par des prédicats. Les atomes qui le composent sont souvent catégorisés en deux groupes distincts : la base de *Herbrand* représentant les composants booléens du jeu (les joueurs, les indices d'une ligne ou d'une colonne d'un plateau, la valeur d'une case, etc) et les atomes d'actions représentant les actions réalisées par les joueurs.

Exemple 1.9 *Reprenons l'exemple du Matching Pennies (illustré dans l'exemple 1.1) composé de plusieurs entités et pouvant être modélisé ainsi :*

— *Les joueurs $j1$ et $j2$ (représentant les rôles du jeu) ;*

— *Les cotés d'une pièce $pile$, $face$ ou $inconnue$ (avant que la pièce ne soit dévoilée).*

Le prédicat binaire $piece$, associé à un joueur et une pièce, identifie la pièce actuelle de chaque joueur. Par exemple $piece(j1, face)$ indique que la pièce retournée par $j1$ est sur le coté $face$.

La seule action possible pour les deux joueurs est de retourner une pièce, ici nous représentons cette action par le prédicat unaire $retourne(C)$ où C désigne le coté d'une pièce.

L'**état d'un jeu** est un sous-ensemble de la base de *Herbrand* du jeu. Tous les atomes inclus dans un état sont vrais et ceux exclus sont faux.

Exemple 1.10 *$piece(j1, inconnue)$ et $piece(j2, inconnue)$ représentent les atomes inclus dans l'état initial du matching pennies. Dans cet état, ces deux atomes sont vrais et tous les autres sont faux.*

À chaque tour de jeu, chaque rôle peut réaliser une ou plusieurs actions légales. Une action dans un état du jeu correspond à une séquence d'actions, une par rôle. Pour chaque action réalisée, quelques atomes de la base deviennent vrais et d'autres faux, permettant d'atteindre un nouvel état du jeu et par conséquent à de nouvelles actions légales.

Exemple 1.11 *À l'état initial du Matching Pennies, $j1$ et $j2$ peuvent réaliser les actions $retourne(face)$ ou $retourne(pile)$.*

Dans un programme `GDL` valide, pour chaque état et chaque séquence d'actions, il n'existe qu'un seul état suivant possible.

Exemple 1.12 *Si à l'état initial du Matching Pennies,* $j1$ *réalise le mouvement* $retourne(face)$ *et* $j2$ *le mouvement* $retourne(pile)$ *alors le seul est unique état suivant est composé des atomes :* $piece(j1, face)$ *et* $piece(j2, pile)$.

Dans un jeu GDL, il n'y a qu'un seul état initial et un ou plusieurs états terminaux. À chaque état terminal est associé un score pour chaque joueur.

Exemple 1.13 *Reprenons l'état obtenu suite à l'action des joueurs* $j1$ *et* $j2$ *dans l'exemple précédent :* $piece(j1, face)$ *et* $piece(j2, pile)$. *Ici, les pièces obtenues ne sont pas retournées sur le même coté, alors* $j1$ *obtient un score de* 0 *indiquant qu'il a perdu et* $j2$ *un score de* 100 *indiquant qu'il remporte la partie.*

Remarque 1.9 *Notons que si le score d'un joueur n'est pas défini dans un programme GDL, son score est assimilé à* 0.

Un jeu GDL débute à l'état initial. Les différents joueurs réalisent leurs actions légales dans cet état afin d'atteindre un nouvel état. Ce processus est répété jusqu'à ce que le programme GDL atteigne un état terminal au cours duquel le jeu s'arrête et les joueurs obtiennent les scores adéquats.

Syntaxe de GDL

GDL impose quelques mots-clés du langage pour définir une syntaxe identique quelque soit le jeu décrit. Le tableau 1.3 présente les dix mots-clés du langage [4].

Mot-clé	Description
`role(j)`	`j` est un joueur
`input(j,a)`	`a` est une action possible pour `j`
`base(p)`	`p` est un atome du jeu
`init(p)`	L'atome `p` est vrai à l'état initial
`true(p)`	L'atome `p` est vrai à l'état courant
`does(j,a)`	Le joueur `j` réalise l'action `a` à l'état courant
`next(p)`	L'atome `p` est vrai dans l'état suivant
`legal(j,a)`	L'action `a` est légale pour le joueur `j` à l'état courant
`terminal`	L'état courant est terminal
`goal(j,n)`	`j` reçoit un score de `n` dans l'état courant

TABLE 1.3 – Mots-clés de GDL

Les différents prédicats utilisés comme mots-clés de GDL doivent répondre à certaines règles d'écriture :

- Les mots-clés $role$, $base$, $input$ et $init$ utilisés dans un programme GDL sont obligatoires et doivent être décrits complètement afin de permettre la validité d'un programme GDL ;
- Les mots-clés $legal$, $goal$ et $terminal$ utilisés en tant que tête d'une règle, ne peuvent accepter que le mot-clé $true$ dans le corps de cette même règle ;
- Le mot-clé $next$ utilisé en tant que tête d'une règle ne peut accepter que les mots-clés $true$ et $does$ dans le corps de cette même règle ;

4. Notons que les mots-clés *input* et *base* ne sont pas présents dans l'article original présentant GDL. Ils ont été ajoutés en 2008, dans le but de rendre plus aisé une potentielle compilation de GDL vers d'autres modèles.

— Les mots-clés $does$ et $true$ ne peuvent pas être utilisés en tant que tête d'une règle ;
— Le score possible d'un joueur défini par le mot-clé `goal` varie entre 0 et 100.

À cet ensemble de mots-clés, il est possible d'ajouter le mot-clé *not* permettant d'indiquer qu'un littéral est négatif et le mot-clé *distinct(d1,d2)* permettant d'indiquer que $d1 \neq d2$.

Exemple 1.14 *La figure 1.4 correspond au programme* `GDL` *du Matching Pennies.*

```
% roles
role(j1).
role(j2).
% base de Herbrand
base(piece(J,C)) ← role(J), cote(C).
base(piece(J,inconnue)) ← role(J).
% actions possibles
input(J,retourne(C)) ← role(J), cote(C).
% coté d'une pièce
cote(pile).
cote(face).
% état initial
init(piece(j1,inconnue)).
init(piece(j2,inconnue)).
% actions légales
legal(J,retourne(C)) ← role(J), cote(C), true(piece(J,inconnue)).
% mis à jour de l'état du jeu
next(piece(P,C)) ← does(P,retourne(C)).
% états terminaux
terminal ← not(true(piece(P,inconnue))).
% scores
goal(J1,100) ← true(piece(J1,S)), true(piece(J2,S)).
goal(J1,0) ← true(piece(J1,S1)), true(piece(J2,S2)), distinct(S1,S2).
goal(J2,0) ← true(piece(J1,S)), true(piece(J2,S)).
goal(J2,100) ← true(piece(J1,S1)), true(piece(J2,S2)), distinct(S1,S2).
```

FIGURE 1.4 – Le programme `GDL` correspondant au jeu « Matching Pennies »

Il commence par la description des joueurs $j1$ et $j2$ via l'utilisation du mot-clé role. Puis, la base de Herbrand est définie via l'utilisation du mot-clé base, ici nous définissons les différents atomes du seul prédicat ne représentant pas une action. Par la suite, les atomes d'actions sont caractérisés par l'utilisation du mot-clé input pour chaque joueur, ici seul l'atome $retourne(C)$ pour chaque joueur J est concerné.

Suite à cela, l'état initial est déclaré à l'aide du mot-clé init, ici il s'agit de définir l'état initial par les atomes $piece(j1, inconnue)$ et $piece(j2, inconnue)$.

Puis, par l'utilisation du mot-clé legal, nous définissons les règles logiques permettant de représenter les actions légales de chaque joueur en fonction de l'état courant défini par le mot-clé true.

Ensuite, la même chose est réalisée pour définir l'évolution du jeu par des règles logiques dont la tête est définie par le mot-clé next. Ici, on indique l'évolution des termes du prédicat $piece$ en fonction des actions réalisées à l'état courant par chaque joueur identifié par l'utilisation du mot-clé does.

Finalement, on définit un état comme terminal par l'utilisation du mot-clé terminal en fonction de l'état courant et les scores associés à chaque joueur par l'utilisation du mot-clé goal. Ici on associe un score de 100 au joueur qui remporte la partie et un score de 0 au joueur qui la perd.

Remarque 1.10 *Au cours de ce manuscrit, nous utilisons une syntaxe infixée pour présenter les programmes* `GDL` *pour des raisons de lisibilité. Toutefois, il est à noter qu'en pratique les programmes* `GDL` *utilisent une syntaxe préfixée.*

Le *General Game Playing* propose au travers de `GDL` un langage permettant de modéliser un grand nombre de jeux d'une grande variété. Afin de retrouver l'ensemble des jeux `GDL` valides existants, le lecteur pourra se référer à [SCHREIBER 2014]. Afin de motiver la recherche sur ce thème, une compétition annuelle est organisée au moment des conférences internationales $AAAI$ ou $IJCAI$, dénommée `IGGPC` (pour *International General Game Playing Competition*) [GENESERETH & BJÖRNSSON 2013].

1.2.2 Compétition internationale : `IGGPC`

Chaque compétition de *General Game Playing* se pratique selon le même processus via des matchs entre un à plusieurs joueurs selon le jeu `GDL` concerné. Chaque joueur reçoit le programme `GDL` et possède un nombre prédéfini de secondes ($startclock$), on parle de phase de pré-traitement, au cours duquel chaque joueur se prépare à jouer (analyse du jeu, développement d'une stratégie, etc). Une fois ce temps passé, le match commence. Au cours du match, chaque joueur possède un temps prédéfini ($playclock$), lui permettant de décider de sa prochaine action afin de l'envoyer. Le match se déroule tour par tour jusqu'à atteindre un état terminal du jeu `GDL` où les scores correspondants à cet état sont assignés à chaque joueur.

Bien que la validité d'un programme `GDL` permet que toutes les actions légales de chaque joueur dans un état donné soient calculables et que dans un état donné une action ne permet d'atteindre qu'un seul et unique nouvel état, il est nécessaire d'ajouter de nouvelles contraintes à `GDL` afin d'éviter des jeux potentiellement problématiques en dehors du cadre d'une compétition. C'est pourquoi, les jeux `GDL` utilisés lors d'une compétition doivent être soumis à quelques restrictions.

Définition 1.27 (Terminaison) *Un programme* `GDL` *se termine si toute stratégie réalisée à partir de l'état initial atteint un état terminal après un nombre fini d'étapes.*

Définition 1.28 (Jouabilité) *Un programme* `GDL` *est jouable si et seulement si chaque joueur possède au moins une action légale dans tout état non terminal atteignable depuis l'état initial.*

Définition 1.29 (Gagnabilité forte) *Un programme* `GDL` *est dit gagnable fortement si et seulement si, pour au moins un joueur, il existe une stratégie pure stationnaire propre au joueur qui mène à un état terminal du jeu où le score du joueur en question est maximal quelques soient les actions des autres joueurs.*

Définition 1.30 (Gagnabilité faible) *Un programme* `GDL` *est dit gagnable faiblement si et seulement si, pour chaque joueur, il existe une stratégie pure stationnaire qui mène à un état terminal où le score du joueur en question est maximal.*

Définition 1.31 (Bien formé) *Un programme* `GDL` *est dit bien formé si il se termine, qu'il est jouable et qu'il est gagnable faiblement.*

Propriété 1.5 *Tout jeu à un joueur bien formé est gagnable fortement.*

Au sein d'une compétition de `GGP`, seuls les programmes `GDL` bien formés sont considérés.

Remarque 1.11 *Dans la suite de ce manuscrit, nous ne considérerons que les jeux* `GDL` *bien formés.*

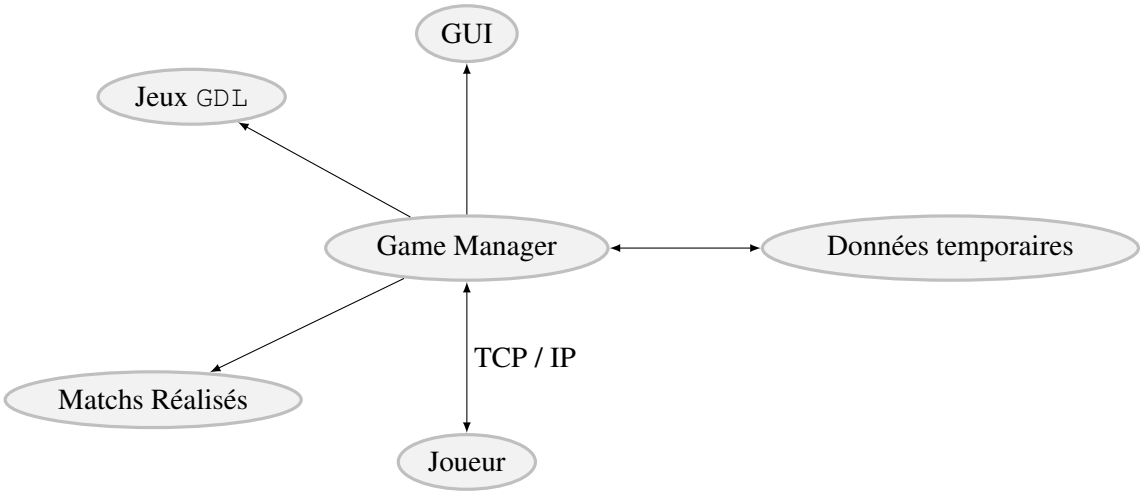


FIGURE 1.5 – Diagramme de l'environnement d'une compétition de GGP

Au sein de la compétition, un protocole de communication est utilisé pour permettre la communication entre les différents protagonistes d'un match. Ce protocole porte le nom de GCL pour *Game Communication Language*. Les compétitions de *General Game Playing* ont pour habitude de s'organiser en ligne autour d'un environnement particulier. La figure 1.5 présente cet environnement.

La partie principale, le *Game Manager* possède une base de données comprenant les différents jeux de la compétition décrits en GDL, les résultats obtenus pour les différents matchs de la compétition et un ensemble de données temporaires représentant les matchs en cours. Chaque match représente un programme GDL et la liste des différents joueurs ayant participé au match associés à leurs scores. Le *Game Manager* fournit également une interface graphique (GUI) pour les différents spectateurs souhaitant suivre les matchs de la compétition.

Enfin, la communication entre le *Game Manager* et chaque joueur de la compétition se réalise via le langage GCL pour *Game Communication Language* au travers de connexions HTTP. Chaque joueur est à l'écoute des différents messages HTTP du *Game Manager* sur un port désigné pour la compétition. Nous détaillons brièvement les cinq types de messages, toutefois, pour plus d'informations, le lecteur pourra se référer à [LOVE *et al.* 2006] :

- *info()* permettant de vérifier que le joueur est toujours en ligne. Si c'est le cas, le joueur doit répondre *available* ;
- *start(id, role, description, startclock, playclock)* permet d'indiquer à un joueur GGP qu'un nouveau match commence. id représente l'identifiant du match auquel le joueur participe, $role$ le rôle que le joueur représente dans le jeu, $description$ représente le programme GDL du match, $startclock$ le nombre de secondes alloué au joueur avant que la première action ne soit demandée, $playclock$ le nombre de secondes alloué au joueur entre chaque tour de jeu pour donner sa prochaine action ;
- *play(id,move)* permet au joueur d'indiquer l'action $move$ qu'il souhaite réaliser dans le match d'identifiant id ;
- *stop(id,move)* permet d'indiquer au joueur que le match id a atteint un état terminal du jeu GDL. $move$ correspond au dernier mouvement réalisé qui a permis d'atteindre l'état terminal ;
- *abort(id)* permet d'indiquer au joueur que le match id s'est terminé anormalement sans atteindre un état terminal.

Depuis la fondation de GDL en 2005, de nombreux programmes-joueurs ont vu le jour et certains ont

été élus champions en remportant la compétition IGGPC. Le tableau 1.4 référence les champions annuels de GGP et leurs auteurs.

Pour finir, une compétition GGP en ligne se déroule en continue sur le serveur Tiltyard [SCHREIBER 2014] où plus de $1,000$ compétiteurs s'affrontent sur plus de 150 jeux. Afin d'établir un classement entre les différents participants, le système *Agon* [5] est utilisé. Il s'agit d'une variante du classement Elo (couramment utilisé pour les échecs) au contexte GGP permettant de calculer la qualité (la compétence généraliste) de chaque joueur associée à la difficulté associée au rôle que représente le programme-joueur au cours d'un match dans chaque jeu sur la base de l'historique de l'ensemble des matchs réalisés sur le serveur. Ce système supporte les jeux à un joueur, multi-joueurs, les jeux asymétriques, les jeux à somme nulle ou non, etc.

1.2.3 Game Description Langage with Incomplete Information (GDL-II)

Lors des compétitions de GGP, le langage GDL est utilisé pour représenter arbitrairement un jeu. Il suppose que chaque joueur est informé immédiatement des mouvements des autres joueurs et de l'état courant du jeu. C'est pourquoi il se limite aux jeux à information complète tout en ne permettant pas d'intégrer les jeux à information imparfaite où une notion de chance est présente, utilisant par exemple une distribution de cartes ou l'utilisation d'un dé. De plus, comme vu précédemment, un grand nombre de jeux (comme le *Go* aveugle, le *Poker* ou encore *Diplomacy*) fait partie des jeux à information incomplète, i.e. les jeux dont les joueurs ne connaissent pas l'ensemble des valeurs des termes de l'état courant (pouvant par exemple être modélisé par un brouillard de guerre, ou plus simplement, une carte face cachée) ou les actions des autres joueurs.

Année	Programme-joueur	Auteur(s)
2005	Cluneplayer [6]	Jim Clune
2006	Fluxplayer [7]	Stephan Schiffel, Michael Thielscher
2007	Cadiaplayer [8]	Yngvi Björnsson, Hilmar Finnsson
2008	Cadiaplayer	Yngvi Björnsson, Hilmar Finnsson, Gylfi Þór Guðmundsson
2009	Ary [9]	Jean Méhat
2010	Ary	Jean Méhat
2011	TurboTurtle	Sam Schreiber
2012	Cadiaplayer [10]	Yngvi Björnsson, Hilmar Finnsson
2013	TurboTurtle	Sam Schreiber
2014	Sancho [11]	Steve Draper, Andrew Rose
2015	Galvanise [12]	Richard Emslie

TABLE 1.4 – Les champions de *General Game Playing*

5. Agon : http://www.ggp.org/researchers/analysis_agonRating.html
4. Cluneplayer : [CLUNE 2007]
5. Fluxplayer : [SCHIFFEL & THIELSCHER 2007b]
6. Cadiaplayer : [FINNSSON & BJÖRNSSON 2008]
7. Ary : http://www.ai.univ-paris8.fr/~jm/ggp/ et [MEHAT & CAZENAVE 2008]
8. Cardiaplayer (upgraded) : [FINNSSON & BJÖRNSSON 2011]
9. Sancho : http://sanchoggp.blogspot.fr/2014/07/sancho-is-ggp-champion-2014.html
10. Galvanise : https://bitbucket.org/rxe/galvanise_v2

Pour répondre à la modélisation de cette classe de jeux, une extension du langage GDL est proposée par [THIELSCHER 2010] dénommée GDL−II (pour *GDL with Incomplete Information*) [THIELSCHER 2011b]. Elle propose d'ajouter deux mots-clés à GDL afin de décrire un environnement représentant la notion de chance dans un jeu et les informations auxquelles accèdent chaque joueur à chaque tour afin de représenter les jeux à information incomplète. La figure 1.5 indique les mots-clés spécifiques à GDL−II.

Mot-clé	Description
sees(j, p)	Le joueur j perçoit l'information p dans l'état courant
random	Représentant le hasard par un joueur dénommé *random*

TABLE 1.5 – Mots-clés spécifiques à GDL−II

Remarque 1.12 *Il est également possible d'ajouter le mot-clé* percept(j, p) *permettant d'indiquer que p est une information potentiellement perceptible pour le joueur j. Son objectif est le même que les mots-clés input et base en permettant de simplifier la détection par un programme-joueur de l'ensemble des informations pouvant être perçu par chaque joueur. Toutefois en pratique percept n'est pas utilisé et ne sera jamais utilisé lors des jeux GDL-II expérimentés au cours de ces travaux.*

Les nouveaux prédicats utilisés comme mots-clés de GDL−II doivent répondre à certaines règles d'écriture :

— Le mot-clé *random* doit uniquement être utilisé en tant que terme constant pour le prédicat $role$;
— Le mot-clé *sees* utilisé en tant que tête d'une règle ne peut accepter que les mots-clés *true* et *does* dans le corps de cette même règle.

Notons que les informations perçues et indiquées par les règles dont la tête utilise le prédicat *sees* peuvent représenter n'importe quelle information allant du mouvement d'un autre joueur à un nouveau terme présent dans aucune autre tête de règle. Même les communications entre joueurs, qu'elles soient publiques ou privées, peuvent être décrites par ces règles de perception.

Par définition, les informations perçues par un joueur sont privées et indiquées lors de l'utilisation du prédicat *sees*. Toutefois, en GDL−II, il est possible de représenter n'importe quel jeu à information complète en ajoutant la règle suivante :

$$sees(J, action(Q, A)) \leftarrow role(J), does(Q, A).$$

L'utilisation de cette règle permet d'indiquer que tous les joueurs seront informés de l'ensemble des actions des autres joueurs à chaque tour. À l'aide de ces informations, un programme-joueur de GDL−II peut déterminer l'ensemble des valeurs de tous les termes du jeu à chaque tour et donc jouer à un jeu à information complète.

En GDL−II, l'environnement est représenté par *role(random)* indiquant que les actions réalisables par ce joueur à chaque tour sont pondérées par une distribution de probabilité uniforme. De ce fait, il est également possible en GDL−II de représenter des actions indéterministes pour chaque autre joueur en représentant les effets de chaque action par une conséquence simultanée avec l'une des actions de l'environnement. Notons que l'utilité de l'environnement n'étant pas une information pertinente, il n'est pas nécessaire de l'indiquer dans un programme GDL−II et sera toujours assimilé à 0.

Le jeu « *Hidden Matching Pennies* », une variante du *Matching Pennies* présenté lors de l'exemple 1.9, est utilisé pour illustrer GDL−II

Exemple 1.15 *Le « Hidden Matching Pennies » est un jeu tour par tour composé d'un joueur nommé pour l'exemple $j1$ et de l'environnement random. Le joueur $j1$ possède une pièce tandis que l'environnement est associé à deux pièces, l'une est perçue par $j1$, la seconde lui est cachée. Au premier tour, l'environnement choisit le côté de chacune de ces pièces, puis, au second tour, $j1$ retourne sa propre pièce. $j1$ remporte 100 points si toutes les pièces sont retournées sur le même côté, 50 points si au moins l'une des pièces de l'environnement est sur le même coté que $j1$ et 0 point sinon.*

Afin de représenter ce jeu, nous avons besoin de désigner ces composants par les termes et prédicats suivants :

- *piece(C) représente la pièce associée à $j1$;*
- *pieces(C1,C2) représentent les pièces associées à l'environnement ;*
- *control(J) permet d'indiquer le tour de jeu du joueur J ;*
- *retourne(C) représente l'action de $j1$ de retourner sa pièce sur le côté C ;*
- *choisit(C1,C2) représente l'action de l'environnement de choisir les côtés $C1$ et $C2$ pour ces deux pièces ;*
- *noop représente l'action de ne rien faire si ce n'est pas le tour de jeu du joueur.*

La figure 1.6 correspond au jeu GDL-II du « Hidden Matching Pennies ».

Il commence par la description des différents rôles $j1$ et $random$. Puis, la base de Herbrand est définie par l'utilisation des mots-clés base et input comme pour un programme GDL classique.

L'état initial correspondant aux pièces retournées faces cachées en début de partie est désigné par les atomes suivants : piece(inconnue) et pieces(inconnue,inconnue).

On définit l'action $noop$ comme légale pour tout joueur dont ce n'est pas le tour et la légalité des actions $choisit(C1, C2)$ et $retourne(C)$ respectivement pour l'environnement et pour $j1$ si c'est leur tour.

L'évolution du jeu établit par les règles dont la tête est composée du prédicat next définit le passage de contrôle de $random$ à $j1$ et les valeurs des prédicats $piece$ et $pieces$ en fonction des actions des joueurs.

Afin de définir la perception de $j1$ sur les fluents du jeu, on utilise les règles dont la tête est composée du prédicat $sees$. Ici, $j1$ ne perçoit que $C1$ dans l'action $choisit(C1, C2)$.

Finalement, on définit les états terminaux comme des états ne contenant aucune piece inconnue et les scores de $j1$ par l'utilisation du prédicat $goal$ selon les valeurs des fluents $piece$ et $pieces$ dans ces états.

Au cours de ce manuscrit, nous nous concentrons essentiellement sur deux classes de jeux :

- Les jeux GDL avec intervention d'un joueur environnement (*random*) ;
- Les jeux GDL-II.

Sémantique de GDL + *random*

Au cours de cette partie, nous présentons la sémantique de GDL + *random* de par la construction d'un jeu de Markov à partir d'un programme valide.

Rappelons qu'un état d'un jeu GDL G est un sous-ensemble de l'univers de *Herbrand* Σ. Puisque les restrictions syntaxiques de GDL garantissent une dérivabilité finie des termes instanciés, tout état est une partie finie de Σ_F correspondant aux termes représentatifs des fluents.

De plus, les règles composant un programme GDL peuvent naturellement être décomposées :

- en règles *init* décrivant l'état initial ;
- en règles *legal* représentant les actions légales de chaque joueur à l'instant courant ;
- en règles *next* capturant le passage de l'état courant vers l'état suivant caractérisé par les effets des différentes actions réalisées ;

```
% rôles
role(j1).
role(random).
% base de Herbrand
base(piece(C)) ← true(cote(C)).
base(piece(inconnue)).
base(pieces(C1,C2)) ← true(cote(C1)), true(cote(C2)).
base(pieces(inconnue,inconnue)).
base(control(J)) ← role(J).
% actions possibles
input(j1,retourne(C)) ← true(cote(C)).
input(random,choisit(C1,C2)) ← true(cote(C1)), true(cote(C2)).
input(J,noop) ← role(J).
% coté d'une pièce
cote(pile).
cote(face).
% état initial
init(piece(inconnue)).
init(pieces(inconnue,inconnue)).
init(control(random)).
% actions légales
legal(random,choisit(C1,C2)) ← true(control(random)), cote(C1),cote(C2).
legal(j1,retourne(C)) ← true(control(j1)), cote(C).
legal(J,noop) ← not(true(control(J)).
% mis à jour de l'état du jeu
next(pieces(C1,C2)) ← does(P,choisit(C1,C2)).
next(piece(C)) ← does(P,retourne(C)).
next(pieces(C1,C2)) ← not(true(control(random)), true(pieces(C1,C2)).
next(piece(C)) ← not(true(control(j1)), true(piece(C)).
next(control(j1)) ← true(control(random)).
% les informations perçues
sees(j1,pieces(C1,_)) ← does(random,choisit(C1,C2)).
% états terminaux
terminal ← not(true(piece(inconnue))), not(true(pieces(inconnue,inconnue))).
goal(j1,100) ← true(piece(C)), true(pieces(C,C)).
goal(j1,50) ← or(true(piece(C1)),true(piece(C2))), true(pieces(C1,C2)),
distinct(C1,C2).
goal(j1,0) ← true(piece(C1)), true(pieces(C2,C2)), distinct(C1,C2).
```

FIGURE 1.6 – Le programme GDL-II correspondant au jeu « Hidden Matching Pennies »

— en règles *terminal* permettant de prendre connaissance de la terminaison ou non du jeu dans l'état courant ;
— en règles *goal* indiquant les scores des différents joueurs lors d'un état terminal.

Ainsi, les instances des prédicats $role(R)$ avec $R \neq random$ définissent les k joueurs du modèle de Markov. $role(random)$ est représenté par un joueur environnement noté $k+1$.

L'état initial s_0 est construit via les termes impliqués dans les règles *init* du programme `GDL`.

Afin de capturer les coups légaux d'un joueur dans un état donné $s = \{f_1, \ldots, f_m\}$, notons s_{true} l'ensemble des faits $\{true(f_1), \ldots, true(f_m)\}$. Les termes instanciés de $legal(J, A)$ dérivables de $G \cup S_{true}$, définissent toutes les actions légales des joueurs J dans l'état s. L'ensemble des actions de l'environnement (indépendantes de l'état courant) sont notées $L(k+1)$ et sont construites à partir des termes de $legal(random, A)$. Les états terminaux S_{ter} et le vecteur de fonctions $\boldsymbol{u}$ sont construits de manière analogue, en utilisant les atomes $terminal$ et $goal(j, c)$.

Remarque 1.13 *Notons que les états résultants ne sont pas forcément équiprobables. Par exemple, un dé « pipé » avec une probabilité de $\frac{1}{2}$ de tomber sur 6 peut être modélisé par dix actions légales de l'environnement dont cinq donnent le même effet (tomber sur un 6).*

La définition suivante reprend formellement la sémantique d'un jeu `GDL` avec *random* par un jeu de Markov.

Définition 1.32 (Jeu de Markov d'un programme `GDL`) *Soit un jeu `GDL` valide G. Un jeu de Markov $(J, S, A, P, \boldsymbol{u})$ décrit G tel que :*

— $J = \{j : G \models \texttt{role(j)}\}$ *;*
— $S = 2^F$ *;*
— $A_j = \{a : G \cup s_{\texttt{true}} \models \texttt{legal(j,a)}, s \in S\}$ *;*
— $P(s' \mid s, \boldsymbol{a}) = \frac{1}{|L_{k+1}(s)|}$ *si* $s' \in succ(s, \boldsymbol{a})$ *et* 0 *sinon ;*
— $u_j(s) = \begin{cases} c & \text{si } G \cup s_{\texttt{true}} \models \texttt{goal(j,c)}, \\ 0 & \text{sinon.} \end{cases}$

Naturellement, l'état initial et les états terminaux sont définis par :

$$s_0 = \{f \in \Sigma_F : G \models \texttt{init}(f)\};$$
$$S_{ter} = \{s \in S : G \cup s_{\texttt{true}} \models \texttt{terminal}\}.$$

Ici, $s_{\texttt{true}} = \{true(f) \mid f \in s\}$ et $succ(s, \boldsymbol{a})$ est l'ensemble des états s' pour lesquels il existe une action $a_{k+1} \in L_{k+1}(s)$ tel que pour tout $f' \in s'$, $G \cup s_{\texttt{true}} \cup \{does(j, a_j)\}_{j=1}^{k+1} \models next(f')$. De plus, $P(\cdot \mid s, \boldsymbol{a})$ est une distribution uniforme sur l'ensemble des états résultants de s via les actions $(\boldsymbol{a}, a_{k+1})$ combinant le vecteur d'actions $\boldsymbol{a}$ des k joueurs et l'ensemble des actions possibles de $a_{k+1} \in L_{k+1}(s)$.

Remarque 1.14 *Notons que de manière évidente cette modélisation englobe l'ensemble des jeux `GDL` sans intervention d'un environnement. En effet, si $role(random)$ n'est pas présent dans le programme G la distribution de probabilités devient :*

$$P(s' \mid s, \boldsymbol{a}) = 1 \text{ si } s' \in succ(s, \boldsymbol{a}) \text{ et } 0 \text{ sinon.}$$

Sémantique de `GDL-II`

De nombreuses descriptions de jeux à information incomplète sont proposées dans la littérature (voir e.g. [SCHIFFEL & THIELSCHER 2011]). Nous nous concentrons ici sur une variante de [GEISSER *et al.* 2014].

La sémantique d'un programme GDL-II G reprend le formalisme d'un POSG (voir définition 1.22). Soit $\mathcal{B}$ représentant la base de *Herbrand* de G ; A (resp. F) l'ensemble des termes d'actions (resp. fluents) apparaissant dans $\mathcal{B}$. Nous utilisons $G \models A$ afin d'indiquer que l'atome A est vrai dans le seul ensemble résultant de G. k représente le nombre de joueurs impliqués dans le jeu et $k+1$ le joueur environnement.

Chaque état s est un sous-ensemble de F où l'état initial s_0 et les états S_{ter} sont définis comme précédemment. L'ensemble $L_j(s)$ des actions légales du joueur j dans l'état s est donné par $G \cup s^{true} \models legal(j, a)$. Spécialement, $L_{k+1}(s)$ représente l'ensemble des actions légales du joueur représentant l'environnement. Toute séquence d'actions (étendue à l'environnement) $\boldsymbol{a} = (a_1, \ldots, a_k, a_{k+1}) \in L_1(s) \times \ldots L_k(s) \times L_{k+1}(s)$ génère un successeur s' de s donné par $\{f : G \cup s^{true} \cup \{does(j, a_j)\}_{j=1}^{k+1} \models next(f)\}$. La distribution de probabilités P sur tous ces successeurs est définie similairement.

L'état de croyance $B_j(s)$ du joueur j de tout successeur s' de s est donné par la distribution jointe de l'ensemble des fluents de F perçus ou non par le joueur j. Si le fluent est perçu par j alors $P(f)$ est égale à 1, sinon à $\frac{1}{2}$. Finalement, le score $u_j(s)$ du joueur j à l'état terminal s est la valeur c tel que $G \cup s^{true} \models goal(j, c)$.

La définition suivante reprend formellement la sémantique d'un jeu GDL-II par un jeu de Markov à information partielle (POSG).

Définition 1.33 (POSG d'un programme GDL-II) *Soit un jeu GDL-II valide G. Un jeu POSG $(J, S, s_0, s_{ter}, A, L, P, B, \boldsymbol{u})$ décrit G tel que :*

- $J = \{j : G \models \texttt{role(j)}\}$;
- $S = 2^F$;
- $A_j = \{a : G \models \texttt{legal(j,a)}\}$;
- $L_j(s) = \{a : G \cup s_{\texttt{true}} \models \texttt{legal(j,a)}, s \in S\}$;
- $P(s' \mid s, \boldsymbol{a}) = \frac{1}{|L_{k+1}(s)|}$ *si* $s' \in succ(s, \boldsymbol{a})$ *et* 0 *sinon ;*
- $B_j(s) = \prod_{f \in F} P(f)$, *où* $\begin{cases} P(f) = 1 & \textit{si } G \cup s_{\texttt{true}} \cup \{does(j, a_j)\}_{j=1}^{k+1} \models \texttt{sees(j,f)}; \\ P(f) = \frac{1}{2} & \textit{sinon}; \end{cases}$
- $u_j(s) = \begin{cases} c & \textit{si } G \cup s_{\texttt{true}} \models \texttt{goal(j,c)}, \\ 0 & \textit{sinon}. \end{cases}$

Naturellement, l'état initial et les états terminaux sont définis par :

$$s_0 = \{f \in \Sigma_F : G \models \texttt{init}(f)\};$$
$$S_{ter} = \{s \in S : G \cup s_{\texttt{true}} \models \texttt{terminal}\}.$$

Ici, $s_{\texttt{true}}$ et $succ(s, \boldsymbol{a})$ sont définis similairement à la définition 1.32.

Compétitions GDL-II et protocole de communication

Le protocole de communication GCL utilisé par un programme GDL classique nécessite quelques modifications afin de permettre la communication entre le *game manager* et un programme-joueur de GDL-II sans que chaque joueur soit automatiquement informé des actions des autres joueurs.

Deux types de messages subissent ces modifications :

- $play(id, turn, move, percept)$ permet au *game manager* d'indiquer au joueur qu'au tour $turn$ du match id, il a reçu le mouvement $move$ et de lui indiquer l'ensemble $percept$ de tout ce qu'il perçoit du jeu dans ce nouvel état ;
- $stop(id, turn, move, percept)$ possède les mêmes arguments que play mais indique que l'état courant atteint est un état terminal.

Bien que l'extension de GDL pour les jeux à information incomplète soit récente (datant de 2011), deux compétitions ont déjà eu lieu utilisant ces deux nouveaux messages au sein de GCL et révélant les premières approches à information incomplète dans le contexte GGP :

— La première compétition s'intitulant la GO-GGP (pour *German Open in General Game playing*) [13] a notamment révélé trois programmes-joueurs : Fluxii (une variante de FluxPlayer [SCHIFFEL & THIELSCHER 2007b] pour GDL-II), StarPlayer [SCHOFIELD *et al.* 2012] et TIIGR [MOTAL 2011] ;

— La seconde compétition s'intitulant la AI GGP (pour *Artificial Intelligence General Game Playing Competition*) [14] a notamment révélé : Une version pour GDL-II de CadiaPlayer [FINNSSON & BJÖRNSSON 2008] et Nexusbaum [EDELKAMP *et al.* 2012].

Les différents programmes-joueurs génériques réalisés depuis 2005, ont permis de mettre en avant différentes approches pour le *General Game Playing* dont nous présentons les plus reconnues dans la section suivante.

1.3 Différentes approches appliquées au GGP

On peut distinguer différentes approches. Les plus anciennes mettent l'accent sur l'extraction de connaissances et la création de fonctions d'évaluation fondées sur des heuristiques tout en utilisant des algorithmes de parcours d'arbre de type Minimax ([KUHLMANN *et al.* 2006], [SCHIFFEL & THIELSCHER 2007a], [CLUNE 2007], [KAISER 2007] et [KIRCI *et al.* 2011]). Plus récemment les algorithmes de type Monte Carlo et ses variantes ([FINNSSON & BJÖRNSSON 2011] et [MEHAT & CAZENAVE 2008]) ont considérablement amélioré l'exploration des arbres des jeux. Dernièrement, associée aux algorithmes de Monte Carlo, l'utilisation de réseaux de propositions, dénommés $propnet$, permet d'accélérer les techniques de simulation appliquées à l'exploration d'arbre.

Comme vu dans la section 1.1.2, les jeux sont souvent représentés sous forme d'arbre de recherche où chaque nœud correspond à un état du jeu et chaque branche reliant deux nœuds correspond à une action légale des différents joueurs provoquant l'apparition d'un nouvel état (et donc d'un nouveau nœud). La racine représente l'état initial du jeu et chaque feuille représente un état terminal où le score final est associé à chaque joueur.

Avec cette représentation, on peut considérer que chaque joueur doit choisir l'action qui lui sera la plus favorable afin d'essayer de suivre un chemin dans l'arbre lui permettant d'atteindre un état terminal où son score est maximum face aux autres joueurs. Dans ce but, de nombreuses techniques ont été mises au point.

Les premières d'entre elles sont les algorithmes de type Minimax qui ont permis notamment à Cluneplayer et à Fluxplayer d'atteindre le rang de champion de GGP en 2005 et 2006 respectivement.

1.3.1 Algorithmes de type Minimax

Appliquer Minimax à un programme-joueur consiste à appliquer la stratégie suivante « à chaque tour de jeu maximiser le score du joueur que nous représentons tout en minimisant celui des adversaires ». L'hypothèse principale de Minimax consiste à considérer que les joueurs adversaires suivent la même stratégie. Cette hypothèse représente le pire des cas. En effet, si l'un des adversaires ne suit pas cette stratégie, il permet au joueur que nous représentons d'atteindre plus facilement un état terminal qui nous

13. GO-GGP : http://fai.cs.uni-saarland.de/kissmann/ggp/go-ggp/
14. AI GGP : http://ai2012.web.cse.unsw.edu.au/ggp.html

avantage. Toutefois, cette hypothèse suggère que le jeu est à somme à nulle, sous-entendant que maximiser le score d'un joueur minimise celui des autres joueurs, ce qui n'est pas toujours le cas, notamment dans les jeux coopératifs.

Exemple 1.16 *La figure 1.7 représente l'exploration d'un arbre de recherche correspondant à un jeu à deux joueurs avec Minimax.*

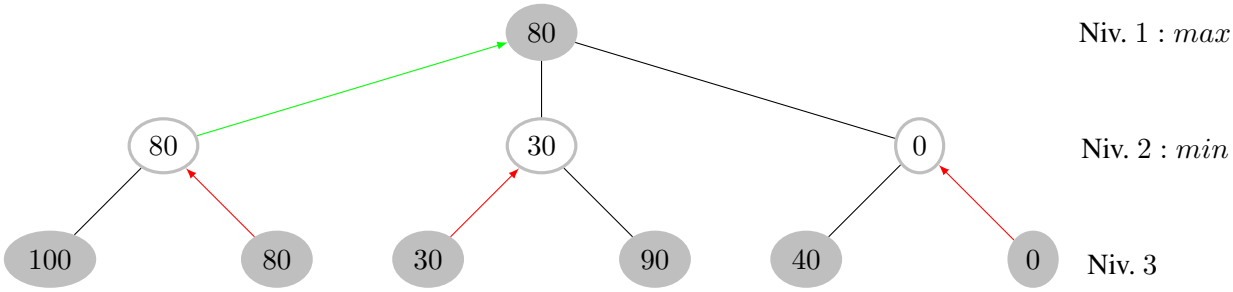


FIGURE 1.7 – Exploration d'un arbre de recherche avec Minimax.

Tout d'abord, il est nécessaire de parcourir l'arbre en profondeur jusqu'à un état terminal afin d'obtenir le score du joueur que nous représentons. Dans la figure, il s'agit de la valeur associée à chaque feuille de l'arbre. De ce fait, comme il est supposé que le joueur adversaire minimise notre score à chaque fois qu'il joue, on construit le niveau précédent en indiquant le score minimal obtenu dans chaque feuille. Dans la figure, il s'agit de remonter la valeur minimale de chaque feuille obtenue par la flèche rouge partant de chaque nœud du niveau 3. Enfin, pour obtenir le niveau précédent correspondant à l'action du joueur que nous représentons, il est nécessaire de choisir la valeur maximum des nœuds du niveau inférieur. Dans la figure, la valeur du nœud racine est obtenue en choisissant la valeur du nœud du niveau inférieur indiqué par la flèche verte. Une fois l'arbre construit, le joueur choisit l'action qui correspond à la valeur maximum obtenue. Dans l'exemple, le joueur choisit l'action correspondant à la branche la plus à gauche du nœud racine portant la valeur 80 *du niveau* 2*. Minimax est répété à chaque fois que l'on doit choisir une action.*

Minimax suppose qu'il est nécessaire de construire une fonction de maximisation et une fonction de minimisation. De ce fait, une variante dénommée Négamax propose une simplification. Le principe est le même, mais il suppose que la valeur du joueur que nous représentons est la valeur du handicap des joueurs adversaires. Dit autrement, on suppose que $max(score) = -min(-score)$.

Exemple 1.17 *La figure 1.8 représente l'exploration du même arbre de recherche que l'exemple précédent avec Négamax.*

Le principe est ici le même que pour Minimax. L'unique différence est la suppression de la fonction de maximisation pour le joueur que nous représentons en remplaçant les valeurs des nœuds construits par la négation des valeurs choisies par la fonction de maximisation. Dans la figure, le niveau 2 est ainsi composé uniquement de valeurs négatives.

Minimax suppose que l'arbre de recherche correspondant au jeu est exploré complètement. Malheureusement, un grand nombre de jeux est associé à un arbre de recherche très imposant. Prenons comme exemple, les échecs dont l'arbre de recherche possède plus de 10^{80} nœuds ou encore le Go avec un arbre de plus de 10^{152} nœuds. En pratique, Minimax ou Négamax ne sont donc pas applicables.

Pour répondre à ce problème, l'algorithme Alpha-Beta (symbolé par α-β) est utilisé et très largement utilisé dans les jeux multi-joueurs [KORF 1991]. Il permet de couper certaines branches dont il semble évident qu'elles ne mèneront pas à un meilleur score. On parle de l'élagage d'arbre.

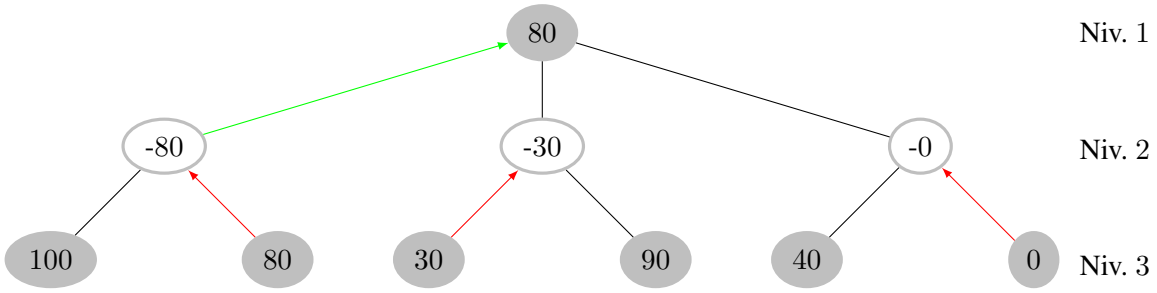


FIGURE 1.8 – Exploration d'un arbre de recherche avec Négamax.

À chaque état dans le jeu, le joueur que représente le programme-joueur tente d'évaluer les bornes maximales (α) et minimales (β) entre lesquelles se situe son score. Ces bornes sont par défaut initialisées à $alpha$ = -∞ et β = +∞. Le joueur commence par explorer une branche jusqu'à atteindre un état terminal. Puis, en suivant les différentes phases de maximisation et de minimisation vues dans Minimax, il remonte une valeur jusqu'à la racine de l'arbre en partant du score obtenu dans l'état terminal atteint. Les bornes de α-β sont alors ajustées.

L'algorithme permet ainsi d'utiliser ces bornes pour ne pas explorer les branches de l'arbre de recherche qui selon les fonctions de maximisation et de minimisation n'apporteraient que des valeurs en dehors de ces bornes. Ce qui est le cas pour tout sous-arbre d'un nœud construit par la fonction de minimisation dont la valeur associée est inférieure à la borne α et pour tout sous-arbre d'un nœud construit par la fonction de maximisation dont la valeur associée est supérieure à la borne β.

Exemple 1.18 *La figure 1.9 représente l'exploration du même arbre de recherche que l'exemple précédent avec Alpha-Beta.*

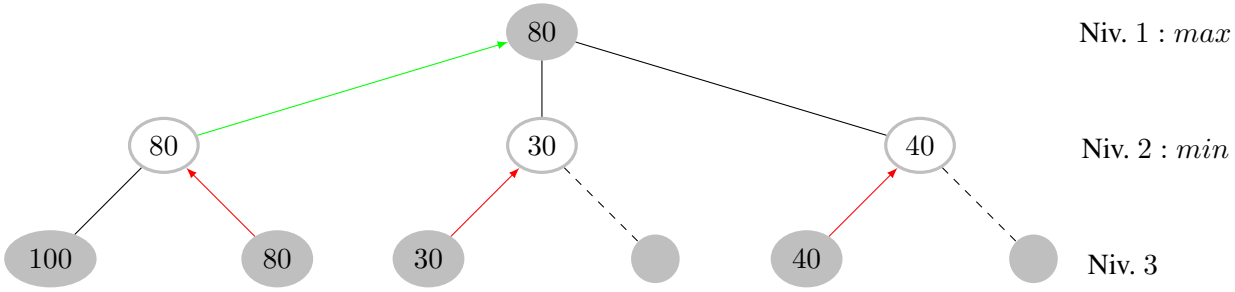


FIGURE 1.9 – Exploration d'un arbre de recherche avec Alpha-Beta.

Initialement, α= -∞ et β= +∞. Le joueur commence par explorer le sous-arbre de gauche afin de remonter la valeur 80 *via le processus Minimax usuel. A ce moment, la borne $\alpha = 80$.*

Ainsi, lors de l'exploration de la seconde branche, la première valeur remontée du sous-arbre de gauche est 30*. De ce fait, il n'est pas nécessaire d'explorer la seconde valeur, car nous sommes sur un niveau où la fonction de minimisation est appliquée, on sait alors d'avance que quelque soit la valeur s'y trouvant, la fonction de minimisation retournera une valeur inférieure ou égale à* 30*. Cette valeur de retour étant inférieure à α, la fonction de maximisation du niveau* 1 *retournera quoi qu'il en soit α, on dit alors qu'on coupe la branche de l'arbre non exploré, schématisé par des pointillés sur la figure.*

Il en va de même pour la branche la plus à droite de la racine, où la valeur 40 *est obtenue par l'exploration de la branche de gauche. Cette valeur est également strictement inférieure à α.*

Dans cet exemple, α-β permet de ne pas s'intéresser à deux nœuds, mais les répercussions sur des arbres bien plus grands sont importantes en évitant d'explorer des sous-arbres entiers.

Remarque 1.15 *De manière équivalente, l'élagage d'arbre réalisé par α-β peut s'appliquer à Négamax.*

Bien qu'Alpha-Beta permette de ne pas explorer l'arbre de recherche dans son intégralité pour simplifier la recherche de la meilleure action, il n'est pas suffisant pour des jeux dont l'arbre de recherche possède une grande profondeur. Ainsi, les recherches dans les arbres sont souvent accompagnées d'heuristiques de recherche. Dans la section suivante, nous présentons les heuristiques de recherche les plus souvent utilisées et qui ont fait ces preuves par leurs utilisations par de nombreux champions de GGP.

1.3.2 Heuristiques de recherche

Lors des premiers travaux en *General Game Playing*, les nœuds non terminaux étaient évalués selon différents critères comme le nombre de pions en possession des joueurs, le contrôle du centre, etc. Malheureusement ces heuristiques demandaient une connaissance du type de jeu joué. Ainsi lors des différentes compétitions de *General Game Playing*, plusieurs autres approches ont été explorées pour l'extraction des caractéristiques du jeu et la création d'heuristiques.

[KUHLMANN *et al.* 2006] propose de déduire des caractéristiques directement de la syntaxe des règles GDL et d'utiliser des heuristiques pour guider l'exploration de l'arbre du jeu et explorer préférentiellement les coups les plus prometteurs. En examinant les règles du jeu, leur programme déduit des éléments de structure comme la relation de successeurs en comparant les termes constants possibles d'un même prédicat et permet également de détecter la présence de compteurs qui utilisent ces relations de successeurs pour changer d'état. Ces travaux ont permis de détecter la présence d'un plateau de jeu à deux dimensions notamment. Ce type de détection permet de créer des heuristiques en fonction des caractéristiques. Plus tard [KUHLMANN & STONE 2007] propose de stocker les règles d'un jeu afin de réutiliser les connaissances acquises. Malheureusement le programme-joueur générique associé nommé UTexax LARG ne gagnera jamais de compétition GGP.

[KAISER 2007] propose une approche similaire à [KUHLMANN *et al.* 2006]. Toutefois, ce dernier utilise une approche statistique pour détecter les caractéristiques d'un jeu. Les différents prédicats sont comparés pour en extraire les différents termes constants et en calculer la variance dans le but de déterminer si un terme est associé à une pièce du jeu ou non. Cette démarche ne se révèlera pas efficace pour le programme-joueur associé nommé Ogre.

[SCHIFFEL & THIELSCHER 2007a] opère une détection similaire mais cette fois-ci basée sur la sémantique des prédicats. Ces travaux se concentrent sur la propriété suivante : une relation binaire qui est antisymétrique, fonctionnelle, injective et dont le graphe est acyclique peut être considérée comme une relation de successeur. Cette façon de procéder permet de détecter des relations de plus haut niveau comme la relation d'ordre. Les éléments structurels détectés permettent d'évaluer des caractéristiques comme la position de pièces sur un plateau de jeu ou la valeur des compteurs. C'est l'une des idées à l'origine du programme-joueur générique Fluxplayer qui a été nommé champion de GGP en 2006.

Puis de nouvelles heuristiques basées uniquement sur la structure de l'arbre de recherche furent développées.

La première en date est l'exploration en profondeur limitée. Elle consiste à explorer l'arbre de recherche jusqu'à un certain niveau de profondeur p. En appliquant cette heuristique à un algorithme de type Minimax, une seule différence existe. Si la recherche atteint une profondeur p et que le nœud obtenu n'est pas terminal, son score est associé à 0.

Cette heuristique comporte plusieurs problèmes, tout d'abord il est possible que la profondeur p considérée ne soit pas suffisante pour atteindre le moindre état terminal. Ensuite, l'association du score

nul à tous les états non terminaux atteints n'est pas très informative et peut provoquer la coupure de branches dans l'arbre de recherche qui aurait pu amener le joueur à une victoire. Et enfin, la difficulté de cette heuristique est de déterminer la profondeur p optimale avant de commencer à jouer, si p est trop petite, les chances sont fortes de ne pas atteindre un état terminal, si p est trop grande, la recherche peut ne pas se finir dans le temps accordé par un match de compétition `GGP`.

Une première solution est d'explorer non pas l'arbre en profondeur mais d'abord en largeur. Malheureusement, très souvent l'espace de recherche nécessaire pour une telle recherche est bien trop important suite au fait que la largeur de l'arbre grandit exponentiellement.

Une solution bien plus utilisée en pratique est d'utiliser une technique d'approfondissement avec fonctions d'évaluation qui explore l'arbre en augmentant itérativement la limite de profondeur p selon un temps donné par le joueur (inférieur au temps autorisé par le match `GGP` pour déterminer sa prochaine action). Le temps restant étant utilisé pour estimer le gain potentiel du joueur sur chaque état non-terminal. Enfin, les actions sont classées en fonction de ce gain ce qui permet à des techniques comme α-β d'être plus efficaces en le poussant potentiellement à élaguer plus l'arbre lors du choix des prochaines actions.

La question ici étant de choisir, les bonnes fonctions d'évaluation. Voici deux fonctions d'évaluation couramment utilisées en `GGP` :

1. La mobilité d'un joueur. Elle peut se caractériser par le nombre d'actions légales ou encore le nombre d'états atteignables dans un état donné. De cette manière, on favorise les états sur lesquels nous avons le moins de connaissances et qui, par hypothèse, ont le plus de chance de nous amener à un état terminal favorable. Il est également possible de choisir une heuristique similaire, où l'on favorise un état où les joueurs adverses ont le moins possible d'états futurs atteignables.

2. La proximité de score (*Goal proximity*). Elle consiste à déterminer la proximité entre un état non-terminal et un état terminal favorable. La méthode consiste à compter le nombre de littéraux positifs dans l'état courant qui sont similaires à ceux d'un état terminal favorable et à évaluer les états non-terminaux par ce nombre.

Il est notamment possible de combiner plusieurs fonctions d'évaluation entre elles pour en faire une nouvelle. Bien sûr, aucune fonction d'évaluation ne garantit le choix optimal d'une action, car les heuristiques sont fondées sur des propriétés locales de chaque état.

Finalement, certains programmes-joueurs comme Maligne [KIRCI *et al.* 2011] utilisent `GIFL` (pour *Game Independent Feature Learning algorithm*) consistant à extraire les prédicats vrais d'un état terminal qui ont une influence sur le résultat obtenu en conjonction avec la dernière action effectuée. Maligne détermine les faits qui ont une influence en supprimant tour à tour chaque fait et en vérifiant si cette ablation a eu un impact sur le score obtenu. Si c'est l'adversaire qui obtient une victoire, l'association des prédicats ayant causée cette défaite constitue une caractéristique défensive. L'association de ces prédicats est alors considérée comme une caractéristique à rechercher ou à éviter pour l'évaluation des états non-terminaux. Cependant certaines combinaisons plus complexes de prédicats ne sont pas détectées par cette technique ce qui limite l'efficacité pour certains jeux et provoque même dans certains cas une contre-performance. Même si l'heuristique utilisée par Maligne est efficace, elle ne lui a jamais permis de devenir champion.

Dans la section suivante, nous mettons en avant les techniques de type Monte Carlo basées sur une analyse probabiliste de l'arbre de recherche qui peuvent également être vues comme une heuristique afin de simuler le gain de chaque état non-terminal et qui ont permis dès 2007 à `Cadiaplayer` de devenir champion de `GGP`.

1.3.3 Méthodes de Monte Carlo

Définition 1.34 (Méthode Monte Carlo) *L'appellation de « méthode de Monte Carlo » désigne tout algorithme qui utilise des nombres aléatoires ou pseudo-aléatoires pour calculer un résultat de façon approché.*

Dans le GGP elles sont très utiles comme alternative quand l'exploration complète de l'arbre de recherche associée n'est pas possible au vu de sa taille afin de simuler l'utilité attendue (aussi nommée gain) pour les états non-terminaux.

Définition 1.35 (Gain d'un état non-terminal) *Le gain d'un état non-terminal correspond au résultat d'une ou de plusieurs simulations réalisées à partir de cet état permettant d'évaluer la probabilité qu'un joueur obtienne la victoire dans cet état.*

Le principe de base des méthodes de Monte Carlo se décompose en deux phases : une phase d'exploration et une phase d'exploitation. La première phase est la phase d'exploration, où sur une profondeur limitée les nœuds de l'arbre sont explorés. Puis vient la phase d'exploitation, où à chaque état du jeu non-terminal, un nombre de simulations (aussi nommée *playouts*) est réalisé à partir de chaque action légale du joueur que représente le programme-joueur dans la limite du temps accordé pour déterminer la prochaine action à réaliser.

Définition 1.36 (Simulation) *Une simulation consiste à tirer au hasard une action parmi l'ensemble des actions légales du jeu état après état jusqu'à atteindre un état terminal.*

Le score obtenu à la suite d'une simulation est attribué à l'action choisie au début du *playout* afin d'obtenir après l'ensemble des simulations, un score moyen pour chaque action légale possible. L'action légale associée au score moyen le plus grand est réalisée.

Cependant, une utilisation directe du principe de Monte Carlo peut s'avérer être un mauvais choix. Notamment suite au fait que cette méthode suppose que les joueurs adverses jouent aléatoirement mais aussi dans certains cas particuliers.

Exemple 1.19 *Supposons que nous lançons directement Monte Carlo sur un état du jeu où seules deux actions sont possibles. La première action du jeu mène à un ensemble de scores moyens (50/100). Tandis que la seconde action dirige les simulations vers un seul et unique score optimal (100/100) et une grande quantité de défaites (0/100). Le score moyen obtenu de la première action sera bien plus élevé que la seconde. L'action jouée sera donc la première alors que la seule possibilité de gagner est de jouer la seconde.*

Comme le montre l'exemple précédent, les méthodes de type Monte Carlo appliquées directement dans un arbre de recherche peuvent poser problème. Pour remédier à ce problème, une variante des techniques de Monte Carlo dénommée MCTS (pour *Monte Carlo Tree Search*) propose une manière plus efficace.

Les deux méthodes proposent de construire un arbre incrémentalement et de simuler un gain pour l'ensemble des états non terminaux. Toutefois, MCTS propose une méthode cyclique en quatre étapes permettant de ne pas séparer strictement la phase d'expansion et d'exploration de l'arbre que propose les techniques de type Monte-Carlo standards [BROWNE *et al.* 2012] :

— *Selection* : MCTS choisit un nœud déjà exploré correspondant à un état non-terminal. Ce nœud est choisi selon une stratégie donnée, dénommée *Tree Policy*, correspondant souvent au nombre de visites déjà réalisées et aux gains déjà obtenus sur les autres nœuds ;

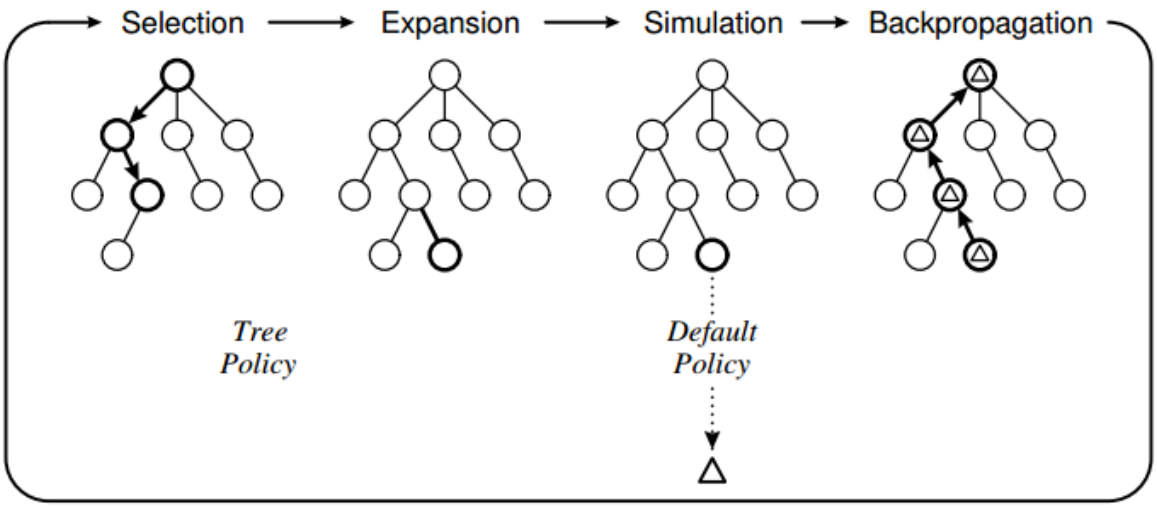


FIGURE 1.10 – Les quatres étapes des méthodes de type MCTS réalisées en cycle

— *Expansion* : Les successeurs du nœud choisis durant la première étape sont ajoutés à l'arbre de recherche ;
— *Simulation* : MCTS effectue une simulation sur le nœud choisi au cours de la première étape. Pour rappel, il choisit successivement des actions au hasard jusqu'à atteindre un état terminal. Toutefois, cette simulation peut également être guidée par une stratégie dénommée *Default Policy* ;
— *Backpropagation* : La valeur obtenue sur l'état terminal atteint est remontée jusqu'à la racine en actualisant le nombre de visites et le gain.

La figure 1.10 représente les quatre étapes réalisées en cycle par les méthodes de type MCTS. L'algorithme 1.1 décrit une itération de MCTS. L'arbre MCTS en entrée est une structure de données usuelle stockant en mémoire les nœuds de l'arbre de recherche. Chaque nœud s dans l'arbre contient également les résultats des précédentes itérations réalisées sur s. Dans les états terminaux, l'algorithme retourne l'utilité du joueur j obtenue (ligne 2). Si l'état courant est déjà stocké dans l'arbre MCTS, les statistiques obtenues dans le nœud correspondant sont utilisées pour sélectionner une combinaison d'actions $\boldsymbol{a}$ (ligne 5). Ces actions sont exécutées (ligne 6) en respectant la probabilité de réaliser cette combinaison d'actions pour atteindre l'état suivant s'. L'algorithme est appelé récursivement sur le nouvel état s' (ligne 7). Le résultat de cet appel est utilisé pour mettre à jour les statistiques du nœud correspondant à s (ligne 8). Si l'état courant s n'est pas présent dans l'arbre MCTS, il est ajouté (ligne 12) et son utilité attendue est estimée par la politique de simulation choisie (ligne 13).

L'efficacité des techniques de type MCTS dépend de la qualité de la stratégie de sélection (*Tree Policy*) et de la stratégie de simulation (*Default Policy*).

Les stratégies de sélection sont fondées sur un compromis entre exploration et exploitation. Pour avoir une idée fiable de l'ensemble du jeu, il est nécessaire d'explorer la plus grand partie possible de l'arbre, mais cette exploration est coûteuse en temps. D'un autre coté, exploiter immédiatement les actions les plus prometteuses peut sembler une meilleure idée, mais elle risque de conduire à écarter les actions qui auraient pu se révéler gagnantes à long terme (ex : il peut parfois être nécessaire de jouer une action défavorable à court terme (sacrifier une pièce) pour obtenir une victoire plus tard). On parle du dilemme exploration/exploitation.

L'algorithme UCT (pour *Upper Confidence bounds applied to Trees*) [KOCSIS & SZEPESVÁRI 2006] est le plus populaire parmi les méthodes MCTS au point où généralement appliquer MCTS sous entend l'utilisation de UCT en l'absence de précision. Le principe d'UCT est de choisir comme départ de chaque

Algorithme 1.1 : MCTS

Données : Arbre MCTS abr, l'état courant s
Résultat : Utilité attendu

si $s \in S_{ter}$ **alors**
  retourner $u_j(s)$
fin
si $s \in abr$ **alors**
  $\boldsymbol{a} \leftarrow$ Selection(s)
  Échantillonner s' en respectant $P(\cdot \mid s, \boldsymbol{a})$
  $u_{s'} \leftarrow \texttt{MCTS}(abr, s')$
  Mis à jour (s,$\boldsymbol{a}$,$u_{s'}$)
  retourner $u_{s'}$
fin
sinon
  Ajouter s en tant que nouvelle feuille de abr
  $u_s \leftarrow$ Simulation(s)
  retourner u_s
fin

simulation un nœud enfant j qui maximise la valeur U_j [BROWNE *et al.* 2012] via une formule nommée UCB (pour *Upper Confidence bounds*) :

$$U_j = \overrightarrow{X_j} + 2C_p\sqrt{\frac{2ln(n)}{n_j}}$$

$\overrightarrow{X_j}$ représente le gain moyen obtenu pour ce nœud, C est une constante permettant de pondérer le dilemme exploration/exploitation, n est le nombre de fois où le nœud parent a été visité et n_j le nombre de visites du nœud j.

Si $n_j = 0$ alors UCT est indéfini. Pour chaque nœud parent, les nœuds enfants non explorés sont visités au moins une fois avant que le choix d'exploiter le meilleur ne survienne. L'autre élément notable, est la nécessité de choisir une valeur pour C, choix qui va déterminer l'exploration et l'exploitation de l'algorithme. C est souvent désigné comme constante d'UCT. Afin d'avoir plus d'informations sur cette constante, le lecteur pourra se référer à cette analyse de UCT dans les jeux multi-joueurs : [STURTEVANT 2008]. Il est à noter que bien qu'UCT soit reconnue comme la technique de MCTS la plus efficace, [COQUELIN & MUNOS 2007] montre que ce n'est pas toujours le cas.

MCTS est l'une des techniques les plus employées dans le domaine du GGP. Elle a fait ses preuves [LONG *et al.* 2010] et a permis au programme-joueur Cadiaplayer ([BJORNSSON & FINNSSON 2009], [FINNSSON & BJÖRNSSON 2011]) de devenir champion. Cadiaplayer utilise en tant que stratégie de sélection UCT [KOCSIS & SZEPESVÁRI 2006] et en tant que stratégie de simulation une distribution de Gibbs (aussi nommée distribution de Boltzmann).

Pour information, une distribution de Gibbs applique cette formule :

$$P(a) = \frac{e^{Qh(a)/\tau}}{\sum_{b=1}^{n} e^{Qh(b)/\tau}}$$

avec $P(a)$ la probabilité qu'une action a de désirabilité $Qh(a)$ soit choisie. Le paramètre τ permet de doser l'influence de $Qh(a)$: quand τ tend vers 0, la probabilité favorise un nombre plus restreint d'actions, et si τ augmente, $P(a)$ tend vers une distribution uniforme. La désirabilité d'une action est évaluée

en fonction d'heuristiques. Pour plus d'informations sur les stratégies de simulation via la distribution de Gibbs, le lecteur pourra se référer à [VIGODA 1999].

Par la suite, nous présentons les différentes améliorations apportées aux techniques de MCTS et plus particulièrement à UCT, appliquées lors des compétitions de *General Game Playing*.

1.3.4 Variantes et améliorations de UCT

Les concepteurs du programme-joueur Cadiaplayer ont proposé plusieurs améliorations à UCT au cours des différentes années lui valant le titre de champion plusieurs années.

MAST

La première d'entre elles est l'utilisation de MAST (pour *Move-Average Sampling Technique*) [FINNSSON & BJÖRNSSON 2008]. L'objectif est qu'après chaque simulation lors de la remontée du gain obtenu au travers des nœuds de l'arbre, la valeur moyenne des gains pour chaque action est stockée dans une table dénommée *lookup table*. Elle permet d'évaluer la valeur d'une action à n'importe quel tour d'un match GGP. L'estimation de cette valeur associée à une action est utilisée pour détourner le choix des actions lors de la stratégie de simulation. Cette stratégie présente un avantage significatif pour de nombreux jeux (exemple : occupation de la case centrale au Morpion, ou des angles à l'Othello). De plus, lors de sa dernière victoire à la compétition IGGPC, Cadiaplayer a utilisé TO-MAST, une variante de MAST où les valeurs des différents nœuds sont mises à jour uniquement si ils ont déjà été explorés ce qui permet de baser la décision de la prochaine action à réaliser sur des données plus robustes. Finalement, les concepteurs de Cadiaplayer proposent d'utiliser conjointement à MAST, FAST (pour *Features-to-Actions Sampling Technique*) utilisant $TD(\lambda)$ (pour *Temporal Difference learning*) [SUTTON 1988] permettant d'évaluer différentes caractéristiques du jeu (position ou déplacement d'une pièce vers une case, etc). Elle est utilisée lors de la phase de pré-traitement et les différentes évaluations sont utilisées de manière pondérée avec MAST pour estimer la valeur des actions et orienter la recherche.

PAST

La seconde variante utilisée par Cadiaplayer aux techniques de MCTS est PAST (pour *Predicate-Average Sampling Technique*). C'est le même principe que MAST sauf que c'est la valeur moyenne de chaque couple action-prédicat qui est calculée. Au cours de l'étape de *Backpropagation*, pour chaque nœud où l'action a est choisie, la valeur de chaque prédicat « vrai » est mise à jour. Ainsi, lors de l'utilisation de la distribution de Gibbs par Cadiaplayer au cours de la phase de simulation, la moyenne la plus élevée au sein des prédicats est utilisée pour le calcul de la probabilité. Cette valeur maximale est utilisée plutôt que la moyenne de l'ensemble des prédicats pour limiter le temps de calcul (généralement court au sein d'une compétition GGP). PAST permet de choisir, selon le contexte, les actions les plus favorables.

Remarque 1.16 *Notons que MAST, TO-MAST et PAST sont des variantes inconciliables de la même technique. Toutefois, il n'existe pas encore de travaux pertinents poussant au choix optimal entre ces trois variantes.*

RAVE

La dernière amélioration apportée à UCT par *cadiaplayer* est l'utilisation de RAVE (pour *Rapid Action Value Estimation*) [GELLY & SILVER 2011]. Il s'agit d'utiliser l'heuristique AMAF (pour *All-Moves-As-First* [BOUZY & HELMSTETTER 2003], utilisée couramment dans différents programmes-joueurs dédiés au jeu de Go, dans le contexte du GGP. AMAF correspond au nombre de simulations sur l'ensemble

des actions réalisées sur l'ensemble de l'arbre de recherche incluant les actions des joueurs adversaires. RAVE permet de mettre à jour l'ensemble des actions rencontrées à la suite d'une simulation sur le nœud sélectionné par UCT à l'aide du gain obtenu en prenant en compte l'ensemble des actions jouées au cours de la simulation et pas uniquement la première action. Contrairement à AMAF qui utilise une fonction d'évaluation par action, ici seuls les gains des actions de la branche courante sont actualisés permettant de distinguer les actions propices dans une situation et néfastes dans d'autres. Le gain de chaque action est estimé séparément par UCT et RAVE. Les deux valeurs sont pondérées dans le but de favoriser RAVE pour sélectionner l'action de départ à l'origine de la simulation tant que le nombre de simulations est trop faible pour se fier à UCT. À mesure que la valeur de UCT s'affine, le poids de RAVE diminue. Une constante est utilisée pour indiquer le nombre d'évaluations à partir duquel la valeur UCT est considérée suffisamment fiable pour remplacer totalement RAVE.

Notons qu'une version améliorée de RAVE est proposée par [CAZENAVE 2015] sous le nom de GRAVE (pour *Generalized RAVE*) où la valeur AMAF n'est pas utilisée uniquement pour l'état choisi par UCT mais également pour les états plus hauts dans l'arbre. En effet, un état supérieur dans l'arbre peut proposer une évaluation plus poussée bien qu'il soit nécessaire de réaliser plus de simulations afin de l'obtenir. GRAVE utilise RAVE uniquement si le nombre de simulations est suffisant pour de tels états. Une constante dénommée ref est utilisée afin de détecter le premier état plus haut dans l'arbre qui possède un nombre de simulations supérieur à ref pour appliquer RAVE. GRAVE se montre plus performant que RAVE dans le cadre du GGP.

L'algorithme 1.2 représente le déroulement récursif de GRAVE. $tref$ représente une table de transposition permettant de détecter le dernier nœud simulé possédant plus de simulations que ref. Elle stocke également pour chaque nœud, le nombre de simulations favorables (victoires), le nombre de simulations totales, et les valeurs équivalentes vis à vis de l'heuristique AMAF. Ainsi, si le nœud choisi possède plus de simulations que ref alors RAVE est utilisé, sinon on recherche via $tref$, le premier nœud sur le chemin entre le nœud choisi et la racine qui possède assez de simulations pour utiliser RAVE sur celui-ci. La valeur β_a est une valeur permettant de décider du choix de l'action à simuler, son calcul dépend du nombre de victoires, du nombre de simulations et des valeurs équivalentes vis à vis de l'heuristique AMAF. Ce calcul correspond à l'algorithme RAVE. Notons que l'algorithme de RAVE est proche de GRAVE, il suffit d'initialiser ref à 0.

Par le passé, en 2009, le programme-joueur Ary [MEHAT & CAZENAVE 2008] a détroné Cadiaplayer au titre de champion de GGP par l'utilisation de ces mêmes tables de transpositions. Pendant une partie, différentes successions d'actions peuvent amener à un même état. Afin de ne pas lancer de nouvelles simulations sur un état déjà connu, les états et leurs évaluations sont stockés dans une table de transposition permettant de diminuer significativement le temps de calcul en réutilisant des informations déjà acquises. Le temps ainsi gagné permet de réaliser un nombre beaucoup plus important de simulations sur les états non-terminaux et par la même occasion d'obtenir une vision de l'arbre de recherche plus détaillée.

NMC

En 2010, Ary reste champion et utilise une seconde technique dénommée NMC (pour *Nested Monte-Carlo Search Algorithm*) [CAZENAVE 2009]. Cette technique est utilisée uniquement pour les jeux à un joueur. Elle consiste à orienter les simulations successives en fonction des précédentes en ayant mémoriser le chemin menant au meilleur score précédemment, puis à répéter la première action à l'origine de ce chemin. À chaque action jouée, une nouvelle série de simulations est effectuée et potentiellement le chemin obtenant le meilleur score est remplacé. Cette approche répond efficacement au dilemme exploration/exploitation et présente l'avantage de ne nécessiter aucun ajustement de paramètre contrairement à UCT. Toutefois, les performances de UCT et de NMC se surpassent l'une l'autre selon le jeu, ainsi [MÉ-

Algorithme 1.2 : GRAVE

Données : nœud n, Table de transposition $tref$, Valeur ref
Résultat : Résultat Res

$actions \leftarrow actions_{legales}(n)$
si $terminal(n)$ **alors**
  retourner $score(n)$
fin
$t \leftarrow tref[n]$
si $t \neq \emptyset$ **alors**
  si $t.nbSimulations > ref$ **alors**
    $tref[n] \leftarrow t$
  fin
  $max \leftarrow$ -∞
  pour chaque $a \in actions$ **faire**
    $v \leftarrow t.victoires[a]$
    $nb \leftarrow t.nbSimulations[a]$
    $va \leftarrow tref.victoiresAMAF[a]$
    $nba \leftarrow tref.nbSimulationsAMAF[a]$
    $\beta_a \leftarrow \frac{nba}{nba+nb+bias \times nba \times nb}$
    $AMAF \leftarrow \frac{va}{nba}$
    $valeur \leftarrow (1 - \beta_a) \times \frac{v}{nb} + \beta_a \times AMAF$
    si $valeur > max$ **alors**
      $max \leftarrow valeur$
      $choixAction \leftarrow a$
    fin
  fin
  $realise(n, choixAction)$
  $res \leftarrow GRAVE(n, tref, ref)$
  $t.res \leftarrow res$
fin
sinon
  new $tref[n]$
  $res \leftarrow simulation(n)$
  $t.res \leftarrow res$
fin
retourner res

HAT & CAZENAVE 2010] présente une fonction d'évaluation combinant les deux approches. À chaque état, l'action réalisée est celle recevant la meilleure évaluation combinée des deux algorithmes.

Finalement, dans des travaux très récents, [CAZENAVE *et al.* 2016] propose une version de NMC aux jeux à deux joueurs. Ces travaux montrent qu'en moyenne NMC est plus performant que UCT. Mais au moment de l'écriture de ce manuscrit, aucun programme-joueur n'a encore utilisé cette technique dans le cadre d'une compétition.

Dans la sous-section suivante, nous présentons les différentes techniques de parallélisation pouvant être apportées aux techniques de MCTS.

1.3.5 Techniques de parallélisation de MCTS

Une amélioration de l'évaluation du gain d'un état calculée par une recherche MCTS peut être obtenue en augmentant la taille limite de l'arbre construit, permettant d'explorer une plus grande partie de la branche correspondant à l'action la plus prometteuse de l'espace de recherche. Cependant, il est également possible d'augmenter le nombre de simulations afin d'affiner les évaluations de chaque état non terminal. [KOCSIS & SZEPESVÁRI 2006] montre qu'en théorie la probabilité de décision de la meilleure action via UCT tend vers 1 quand le nombre de simulations tend vers l'infini.

Avec pour objectif d'augmenter le nombre de simulations lors de l'exploration de l'arbre du jeu, quatre techniques de parallélisation existent.

Parallélisation au niveau de l'arbre

Historiquement, la première technique de parallélisation d'une technique de MCTS est proposée par [GELLY *et al.* 2006]. Elle est appliquée au jeu de Go via le programme-joueur *Mogo* en construisant un arbre unique au sein d'une mémoire partagée par plusieurs threads réalisant les simulations où un mutex protège l'accès à l'arbre. La figure 1.11 montre schématiquement des simulations parallèles réalisées au niveau de l'arbre où chaque couleur correspond à un thread.

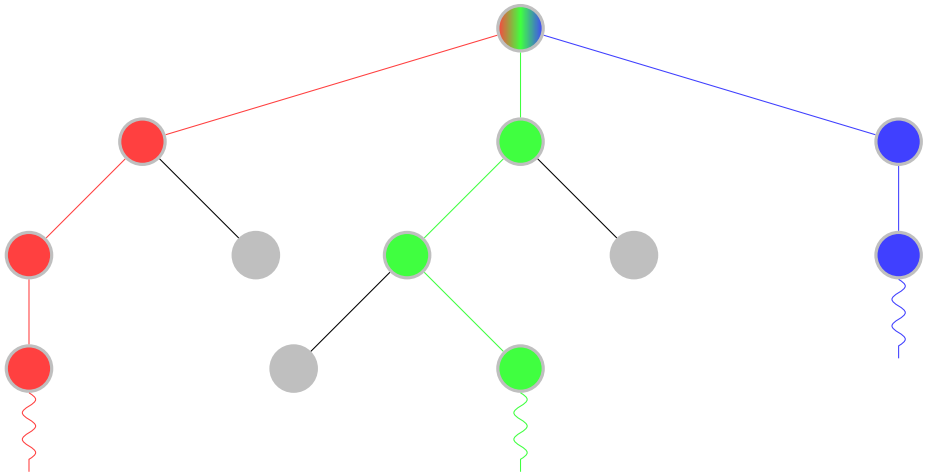


FIGURE 1.11 – Parallélisation au niveau de l'arbre des techniques de MCTS

L'efficacité de la technique de parallélisation, nommée *tree parallelization* ([CHASLOT *et al.* 2008] et [CHASLOT 2010]) est limitée par le poids de la synchronisation par mutexes.

Pour passer outre la limite [CHASLOT *et al.* 2008] propose d'utiliser des mutexes locaux où chaque nœud de l'arbre possède son propre mutex. Cette technique permet de verrouiller une zone limitée de

l'arbre et permet aux threads de travailler parallèlement. Pour éviter le surcoût dû aux fréquents verrouillages et déverrouillages des mutexes, ils utilisent un système de verrouillage fondé sur une attente active (*spin-lock*).

Malheureusement, cette amélioration est au prix d'une augmentation de la taille de la structure de données représentant chaque nœud [ENZENBERGER & MÜLLER 2009]. De plus, puisque les résultats des n-1 simulations précédentes ne sont pas encore remontés au moment où le thread n démarre une simulation à partir d'un nœud, les n threads risquent d'explorer la même zone prometteuse de l'arbre [GELLY *et al.* 2008].

[CHASLOT *et al.* 2008] et [CHASLOT 2010] utilisent une perte virtuelle (*virtual loss*) pour éviter qu'un nœud ne soit exploré inutilement et que tous les threads attendent le déblocage du même mutex local. Elle consiste à diminuer l'évaluation d'un nœud quand il est choisi comme point de départ d'une simulation. Ce nœud ne sera choisi par le thread suivant que si son évaluation est toujours supérieure à celle des autres nœuds sinon c'est le nœud qui possède la seconde meilleure évaluation qui sera choisi comme origine d'une simulation par le thread suivant et ainsi de suite. Cette perte virtuelle est supprimée lorsque l'évaluation du nœud est terminée.

Finalement, [ENZENBERGER & MÜLLER 2009] montre que l'utilisation de mutexes provoque rapidement des temps d'attente qui empêchent le passage à l'échelle. Ils proposent un algorithme de *tree parallelization* sans mutex qui limite les erreurs provoquées par des accès concurrents à l'arbre de recherche. Les expériences réalisées sur le jeu de Go montrent que la recherche effectuée avec n threads est équivalente en terme de parties gagnées à la version séquentielle avec n fois plus de temps.

Parallélisation à la racine de l'arbre

Un autre type de méthodes de parallélisation nommé originalement *root parallelization* est proposé par [CAZENAVE & JOUANDEAU 2007]. La figure 1.12 montre schématiquement des simulations parallèles réalisées au niveau de la racine de l'arbre où chaque couleur correspond à un thread.

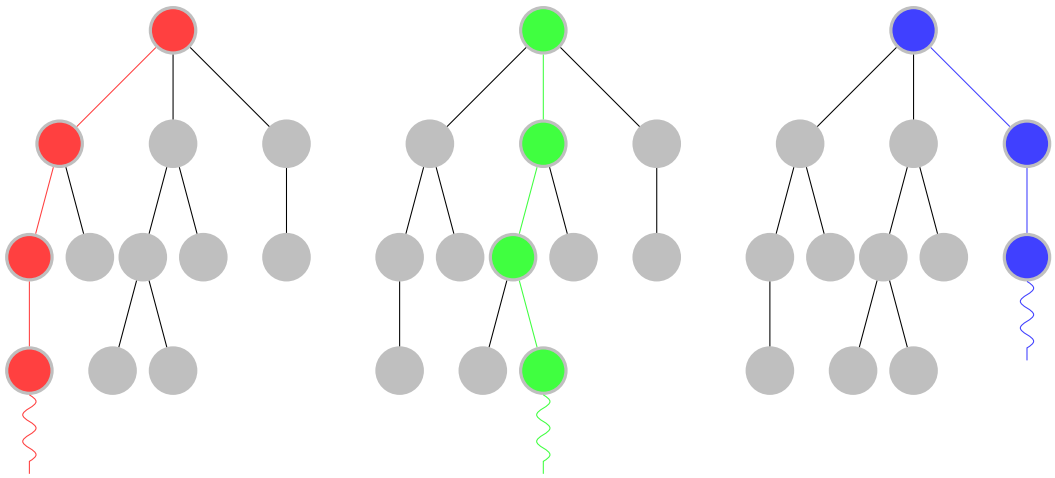


FIGURE 1.12 – Parallélisation au niveau de la racine des techniques de MCTS

Elle consiste à développer n arbres UCT distincts sur chacun des esclaves à partir de la racine et peut se présenter sous la forme de deux approches : *single-run* et *multiple-run*.

À la fin du temps alloué, *single-run* additionne, par le processus maître, les valeurs calculées par les différents esclaves pour les nœuds enfants de la racine et le meilleur nœud est choisi.

Quant à *multiple-run*, où le temps est découpé en plusieurs phases, la mise à jour des valeurs des nœuds est réalisée à la fin de chacune d'elles. Dans cette approche, Le processus maître regroupe les valeurs calculées par les différents esclaves et, en calculant une moyenne, relance la recherche de chaque esclave avec les valeurs mises à jour. Cette dernière méthode implique une augmentation des temps de communication mais donne de meilleurs résultats car elle apporte un partage des connaissances qui permet de mieux orienter la recherche. [GELLY *et al.* 2008] étend cette technique en synchronisant à intervalles de temps réguliers les évaluations des nœuds enfants de la racine qui ont été suffisamment visités sur k niveaux de profondeur.

[MÉHAT & CAZENAVE 2011a] propose le premier programme-joueur appliquant l'approche *single-run* tout en expérimentant différentes stratégies pour fusionner les résultats des différentes évaluations de type `UCT` effectuées par les esclaves :

- La première d'entre elle est *Best* consistant à choisir le coup ayant la meilleure évaluation toutes recherches confondues ;
- La seconde est `Sum` fusionnant les nb évaluations e_n d'un nœud n en les pondérant par le nombre de simulations s_n de chaque nœud par rapport au nombre total de simulations :

$$\frac{\sum_{i=1}^{nb} e_{n_i} \times s_{n_i}}{\sum_{i=1}^{nb} s_{n_i}}$$

- `Sum` favorise malheureusement les « fausses bonnes actions ». Ainsi, une stratégie similaire dénommée `Sum10` est utilisée pour ne prendre en compte que les 10 meilleures actions de chaque évaluation `UCT`. Dans cette approche, les mauvaises évaluations correspondant aux « fausses bonnes actions » ne sont pas prises en compte et ne peuvent pas fausser les évaluations de `UCT` qui n'auraient pas détectées ces actions comme mauvaises ;
- La dernière stratégie est `Rave`. Elle consiste à cumuler les évaluations sans tenir compte du nombre de simulations, c'est un moyen simple d'éviter le biais en faveur des « fausses bonnes actions » :

$$\frac{\sum_{i=1}^{nb} e_{n_i}}{\sum_{i=1}^{nb} s_{n_i}}$$

Les résultats expérimentaux montrent que les meilleures performances sont obtenues avec `Sum` et `Sum10` dont les résultats sont équivalents. Comparé à une recherche séquentielle effectuée sur un même laps de temps, le gain apporté par la parallélisation sur n esclaves en terme de jeu est positif bien que parfois inférieur ou identique à une recherche séquentielle bénéficiant de n fois plus de temps. Toutefois, la performance de ce type de parallélisation pour le domaine du *General Game Playing* dépend principalement de la capacité du joueur à garder les processus esclaves occupés.

La parallélisation au niveau de la racine ne fonctionne pas pour tous les jeux [MÉHAT & CAZENAVE 2011b] contrairement à la parallélisation au niveau de l'arbre. [SOEJIMA *et al.* 2010] montre que la parallélisation au niveau de l'arbre est plus efficace que la parallélisation au niveau de la racine à cause du partage de connaissances que permet la première approche. De plus, l'arbre généré par la première approche est potentiellement plus grand. Toutefois, la première approche n'a jamais été appliquée au *General Game Playing* car elle n'est pas applicable aux systèmes distribués.

Parallélisation au niveau des feuilles

La troisième technique de parallélisation, nommée *leaf parallelization*, est utilisable dans une architecture avec mémoire distribuée [CAZENAVE & JOUANDEAU 2007]. La figure 1.13 montre schématiquement des simulations parallèles réalisées au niveau des feuilles où chaque couleur correspond à un thread.

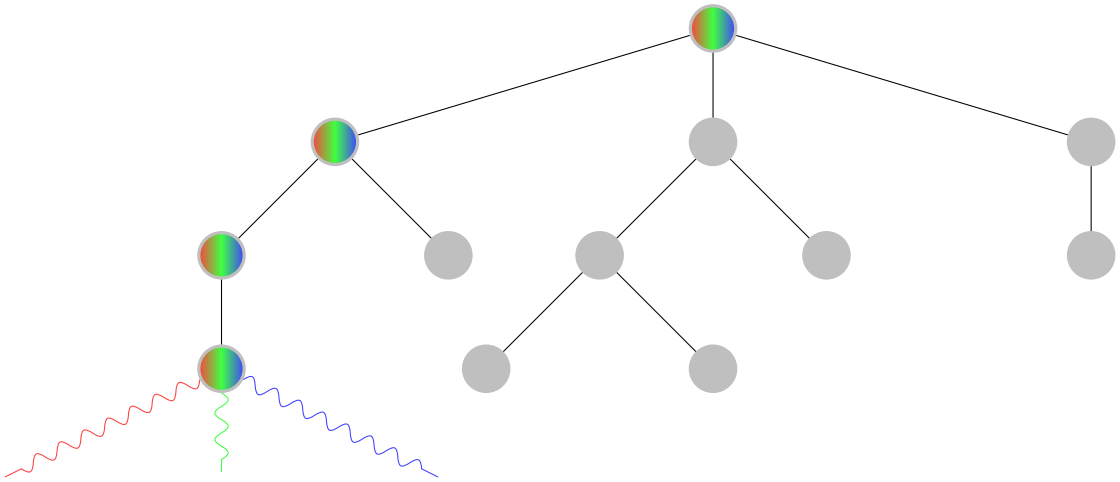


FIGURE 1.13 – Parallélisation au niveau des feuilles des techniques de MCTS

L'arbre est construit par le processus maître et seules les simulations sont exécutées par les processus esclaves. À partir d'une feuille, chaque esclave fait un certain nombre de simulations et retourne l'évaluation moyenne obtenue. Le processus maître effectue alors une moyenne de ces moyennes.

Dans cette approche, le processus maître doit attendre le résultat d'un groupe de simulations avant d'en lancer un autre et à chaque nouveau nœud une nouvelle communication avec les processus esclaves est nécessaire : ceci provoque un surcoût important en temps de synchronisation et de communication. Le second inconvénient de cette méthode est que toutes les simulations sont effectuées à partir du même état sans ajustement de l'évaluation comme dans la version séquentielle de UCT. Suite à ces deux défauts, cette approche est reconnue comme inférieure aux deux premières.

Toutefois, le programme-joueur Cadiaplayer [FINNSSON & BJÖRNSSON 2008] reprend son titre de champion de GGP en 2012 en utilisant une parallélisation aux feuilles où chaque processus esclave lance plusieurs simulations à partir d'un même état pour améliorer l'équilibre entre le poids des communications et les temps de calcul.

Pour améliorer la performance de la parallélisation au niveau des feuilles, [CHASLOT *et al.* 2008] tente de stopper les threads restants si les résultats des premiers indiquent qu'une action est peu intéressante, ce qui permet de relancer une évaluation sur un autre nœud. Cependant, cette modification ne permet pas un bon passage à l'échelle car un grand nombre de simulations est rapidement annulé.

[CAZENAVE & JOUANDEAU 2007] tente d'utiliser des communications étagées : chaque processus enfant reçoit un état à partir duquel il réalise une simulation et retourne le résultat dès qu'il est prêt. Le processus maître lui envoie alors un autre état à partir duquel travailler. Ainsi le processus maître n'attend pas de collecter les résultats renvoyés par tous les processus esclaves et chaque esclave peut travailler sur un état distinct. Sur le jeu de *Go*, cette technique a permis un bon passage à l'échelle jusqu'à 16 esclaves.

Malgré ces améliorations, la technique de parallélisation aux feuilles reste moins performante que la parallélisation au niveau de l'arbre ou de la racine car elle souffre de limitations provoquées par le processus maître et par les communications [SOEJIMA *et al.* 2010].

Parallélisation avec arbre distribué

Sur un système distribué, seules les parallélisations au feuilles ou à la racine sont envisagées puisqu'aucun accès à une mémoire commune n'est possible pour stocker l'arbre. Cependant la mémoire

disponible sur les différentes machines n'est pas utilisée de manière optimale : dans la parallélisation aux feuilles, l'arbre occupe la mémoire du serveur et celle des clients est sous-exploitée, tandis qu'avec la parallélisation à la racine, un même arbre est dupliqué sur chaque client amenant à stocker des informations redondantes. De plus, avec la parallélisation à la racine sur un système distribué, la perte virtuelle ne peut pas être appliquée ce qui peut expliquer un passage à l'échelle limité [YOSHIZOE *et al.* 2011].

[GRAF *et al.* 2011] propose une solution distribuant le stockage de l'arbre sur différents clusters. Comme vu dans la section précédente, l'utilisation d'une table de transposition dans laquelle chaque nœud associé à un clé de hashage est stocké renforce fortement les techniques de MCTS. L'idée est d'utiliser cette clé de hashage sur différents clusters afin de choisir la mémoire du cluster où le nœud devra être stocké.

Au cours de la descente dans l'arbre jusqu'à la feuille la plus prometteuse, un cluster peut recevoir une requête de déplacement vers un nœud. Si ce nœud n'existe pas dans la table de transposition, le cluster crée un nouveau nœud et lance une simulation. Dans le cas où le nœud existe dans la table, le cluster sélectionne un de ces nœuds enfants. Si ce nœud enfant est stocké localement, le cluster entame une nouvelle simulation, sinon il envoie une requête de déplacement vers le cluster chargé du stockage du nœud. Lorsqu'une simulation retourne une évaluation pour un nœud, un message de mise à jour est envoyé à tous les clusters stockant les nœuds parents.

Chaque cluster possède un processeur d'entrée/sortie dédié aux communications avec les autres clusters. Les messages reçus par un cluster sont accumulés dans une queue et traités par l'un des processeurs de calcul dès qu'il se libère de la tâche précédente. Les échanges d'informations entre les clusters sont ainsi effectués en parallèle avec les simulations.

Pour limiter le nombre de messages, certains nœuds très souvent visités (la racine notamment) sont dupliqués et stockés dans le cache local de chaque cluster. Un paramètre contrôle la quantité de nœuds dupliqués sur les différents clusters : au pire, avec le partage de la totalité des nœuds, on peut obtenir l'équivalent d'une parallélisation à la racine avec synchronisation régulière. Un autre paramètre contrôle le nombre de simulations effectuées pour un nœud avant synchronisation. Ce paramètre permet de limiter les communications qui dans le pire des cas en réduisant au minimum les communications, il est possible d'obtenir l'équivalent de la parallélisation à la racine correspondant à l'approche *single-run* avec une unique synchronisation à la fin.

Lors de la recherche, le nombre de requêtes de déplacement peut vite faire exploser les temps de communication et empêche un bon passage à l'échelle. Pour éviter de redescendre dans l'arbre depuis la racine à chaque ajout d'un nouveau nœud, [YOSHIZOE *et al.* 2011] améliore la technique de parallélisation de [GRAF *et al.* 2011] en proposant d'effectuer une recherche en profondeur d'abord. Chaque fois qu'une simulation est effectuée et une évaluation remontée, l'évaluation d'un nœud parent est comparée à celle de ses frères. Si le nœud possède toujours la meilleure évaluation de la « fraterie », l'évaluation n'est pas remontée davantage et la recherche suivante démarre à partir de ce nœud. Cette façon de procéder donne un résultat globalement équivalent à UCT mais permet d'économiser beaucoup de déplacements dans l'arbre de communications pour mettre à jour l'évaluation des nœuds. De plus, [YOSHIZOE *et al.* 2011] propose également une modification de l'évaluation UCB pour ajouter le principe de perte virtuelle permettant d'éviter qu'un nœud ne soit trop visité suite aux mises à jour moins fréquentes de l'évaluation des nœuds :

$$\frac{w_i}{s_i + d_i} + C \times \sqrt{\frac{ln(t + d_\tau)}{s_i + d_i}}$$

avec d_i le nombre de recherches en cours à partir du nœud i, s_i le nombre de simulations déjà effectuées, w_i le nombre de victoires obtenues, d_τ le nombre de recherches en cours pour tous les nœuds apparentés,

t le nombre total de visites déjà effectuées pour tous les nœuds enfants et C la constante répondant au dilemme exploitation/exploration.

Parallélisation de NMC

Finalement, l'algorithme NMC utilisé par le programme-joueur Ary et proposé par [CAZENAVE 2009] pour les jeux à un joueur a également été proposé dans une version parallèle : [CAZENAVE & JOUANDEAU 2009]. Comme dans la technique de parallélisation à partir des feuilles, un processus maître construit l'arbre de jeu, tandis que les simulations sont réalisées par des processus esclaves. Le processus maître envoie des nœuds à explorer à des processus « médians ». Un tel processus effectue une descente dans l'arbre et demande, à chaque niveau, aux processus esclaves de faire une simulation pour chaque nœud enfant. Ensuite le processus « médian » choisit le nœud qui a la meilleure évaluation pour passer au niveau suivant.

Avant l'envoi d'une tâche à un esclave, chaque processus « médian » communique la nature de cette tâche à un processus répartiteur qui doit lui indiquer en retour l'identifiant de l'esclave auquel envoyer le travail. Un algorithme *Round-Robin* est utilisé pour classer les travaux en fonction de leur taille estimée et les distribuer aux différents esclaves. Dans une autre version, *lastMinute*, un esclave informe le répartiteur des travaux quand il n'a plus de travail à faire, le travail le plus long de la file d'attente lui est alors donné, sinon, si aucune tâche n'est en attente, l'identifiant de l'esclave est enregistré dans la liste des esclaves disponibles.

Bilan sur les différentes techniques de parallélisation

Dans toutes les techniques de parallélisation, le passage à l'échelle peut être entravé par trois types de surcoût [SOEJIMA *et al.* 2010] :

- Un surcoût en calculs quand une partie non prometteuse de l'arbre est développée ou qu'un excédent de simulations est effectué par différents processus en parallèle ;
- Un surcoût en synchronisations quand le résultat des calculs de plusieurs threads est attendu par un processus maître ou quand le blocage d'un mutex génère une attente ;
- Un surcoût en communications quand de nombreux échanges d'informations sont nécessaires, dans une architecture distribuée notamment.

Cependant [BOURKI *et al.* 2010] indique que toutes ces difficultés ne sont pas les seules à empêcher le passage à l'échelle. Il signale qu'au jeu de *Go*, le passage à l'échelle de MCTS n'est pas constant. En faisant s'affronter deux joueurs identiques effectuant n et $n \times 2$ simulations, la différence de niveau de jeu n'est pas aussi marquée entre eux quand la valeur de n augmente. Ceci est à prendre en compte lors de l'augmentation des simulations au cours de la parallélisation.

Par conséquent, les techniques de parallélisation pour MCTS appliquées au *General Game Playing* sont encore en plein essor. Afin de rendre plus efficace les techniques de simulation, les réseaux de propositions sont utilisés pour examiner les règles d'un jeu GDL.

1.3.6 Réseau de propositions ($propnet$)

Au cours de l'exploration d'arbre de recherche correspondant aux différents jeux proposés par le *General Game Playing*, il est nécessaire d'utiliser un programme capable d'examiner les règles du jeu GDL pour établir la liste des actions légales de chaque état, de déterminer le prochain état du jeu en fonction de la réalisation d'une action et de déterminer si cet état est terminal ainsi que le score associé à chaque joueur pour construire l'arbre de recherche.

GDL étant un langage logique similaire à Prolog, les premiers programmes-joueurs se sont basés sur des programmes d'interprétation de Prolog pour construire les arbres de recherche ou sur les ASP (pour

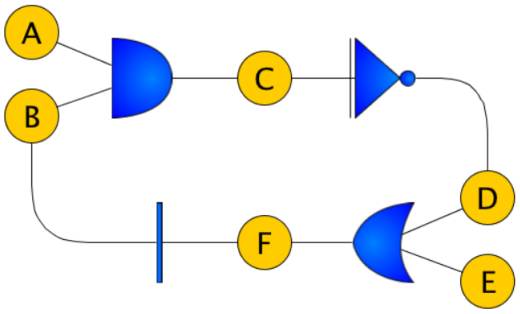


FIGURE 1.14 – Un réseau de propositions $propnet$

Answer Set Programming) [CEREXHE *et al.* 2013]. Bien qu'ils soient efficaces, les principes d'unification, de récursivité et de retour sur trace (*backtracking*) restent lents et limitent l'exploration des arbres de recherche notamment pour les jeux dont les règles comportent un grand nombre de variables.

Ainsi, pour répondre à ce problème le concept de réseau de propositions a été initié. Ce concept est issu de AP, un Automate Propositionnel, inventé par le *Stanford Logic Group* [GENESERETH & THIELSCHER 2014] pour les systèmes multi-agents discrets et dynamiques tel que les jeux [COX *et al.* 2009]. Il apporte une amélioration conséquente à la vitesse d'interprétation des règles et ainsi un nombre de simulations de parties réalisables beaucoup plus important.

Définition 1.37 (Réseau de propositions) *Un réseau de propositions, aussi nommé $propnet$ (pour Propositional Network), est un hypergraphe biparti orienté composé de propositions connectées par les connecteurs logiques de conjonction (ET $\wedge$), de disjonction (OU $\vee$), de négation (NON $\neg$) et de transitions simulant la dynamique du jeu. Ce dernier recopie son entrée sur sa sortie à chaque itération et ainsi permet de simuler le passage d'un état à un autre du jeu.*

Les propositions sont partitionnées en trois classes :
- Les propositions d'actions (*input propositions*) représentant les actions possibles des joueurs (autrement dit, ceux ne possédant pas de paramètres d'entrée) ;
- Les propositions de faits (*base propositions*) correspondant aux faits décrivant l'état courant (ceux dont les arcs entrants proviennent de connecteurs de transition) ;
- Les propositions de vues (*view propositions*) regroupant les faits des autres situations du jeu (ceux dont les arcs entrants ne proviennent pas de connecteurs de transition).

L'avantage principale de cette représentation est sa densité.

Propriété 1.6 *Si un jeu est décrit par un ensemble de n faits, il peut être décrit par 2^n états différents correspondant à l'ensemble des combinaisons de toutes les valeurs booléennes des faits.*

Exemple 1.20 *La figure 1.14 représente un réseau de propositions $propnet$. Ce réseau est composé de six propositions : A, B, C, D, E, F. A et E représentent des propositions d'actions, B est une proposition de faits et C, D et F représentent des propositions de vues.*

Le programme-joueur `TurboTurtle` est le premier à avoir utilisé un $propnet$ pour accélérer les simulations réalisées lors de l'utilisation des techniques de `MCTS` lors d'une compétition `GGP` ce qui lui a valu le titre de champion en 2013.

Lors des premières compétitions de `GGP`, les mots-clés *input* et *base* n'étaient pas obligatoires pour modéliser un jeu `GDL` valide. Ainsi, comme il est nécessaire pour modéliser un $propnet$ de connaître l'ensemble des termes constants, des techniques de *grounding* et de *tabling* ont été proposées.

Définition 1.38 (*Grounding*) *Le grounding [*KISSMANN & EDELKAMP *2010] correspond à l'instanciation de toutes les variables des règles* `GDL` *du jeu afin d'obtenir un programme* `GDL` *ne contenant qu'un ensemble de propositions logiques où tous les termes sont constants. La méthode utilisée pour réaliser cette étape est dénommée Grounder.*

Exemple 1.21 *Soit la règle* `GDL` *legal sous forme infixée décrivant l'action légale du joueur désigné par le terme variable P de modifier une case ($cell$) dont les indices sont désignés par les termes variables X et Y si dans l'état courant, la même case est vide (désignée par le terme constant b) et qu'il s'agit du tour du joueur P (indiqué par le prédicat $control(P)$) :*

$$legal(P, mark(X, Y)) \leftarrow true(control(P)), true(cell(X, Y, b).$$

En appliquant l'étape de grounding sur cette règle logique nous obtenons l'ensemble de règles logiques suivants :

$$\begin{aligned}
legal(xplayer, mark(1,1)) &\leftarrow true(control(xplayer)), true(cell(1,1,b).\\
legal(xplayer, mark(1,2)) &\leftarrow true(control(xplayer)), true(cell(1,2,b).\\
legal(xplayer, mark(1,3)) &\leftarrow true(control(xplayer)), true(cell(1,3,b).\\
&\dots\\
legal(oplayer, mark(1,1)) &\leftarrow true(control(oplayer)), true(cell(1,1,b).\\
legal(oplayer, mark(1,2)) &\leftarrow true(control(oplayer)), true(cell(1,2,b).\\
legal(oplayer, mark(1,3)) &\leftarrow true(control(oplayer)), true(cell(1,3,b).
\end{aligned}$$

[VITTAUT & MÉHAT 2014] décrit une technique utilisant Prolog pour instancier efficacement les règles `GDL` avec pour objectif d'atteindre un $propnet$. Les règles `GDL` originales subissent un ensemble de pré-traitements visant à les simplifier et à préparer leur instanciation. Ensuite, l'interprète Prolog est utilisé pour unifier les variables avec toutes les valeurs possibles. Certains interprètes comme $BProlog$, XSB ou YAP proposent un système de mémorisation nommé *tabling*.

Définition 1.39 (*Tabling*) *Le Tabling consiste à regrouper toutes les instanciations possibles des variables composant une règle logique* `GDL`*. Dit autrement, il s'agit d'une structure permettant de stocker les règles unifiées pour ne pas avoir à rechercher à chaque fois les instanciations possibles en examinant chaque règle.*

L'origine des mots-clés *input* et *base* vient du fait qu'il existe certains jeux `GDL` valides possédant un nombre très important de termes constants possibles. De ce fait, le temps et l'espace nécessaires aux phases de *grounding* et de *tabling* sont très importants et peuvent parfois dépasser le temps de pré-traitement accordé lors des matchs de `GGP` et demander une mémoire conséquente. Toutefois, un programme-joueur dénommé LeJoueur ayant remporté une compétition de `GGP` annexe qui s'intitule *2015 Tiltyard Open* utilise un *Grounder* [VITTAUT & MÉHAT 2014] qui est annoncé comme capable de se terminer sur une majorité des jeux disponibles [SCHREIBER 2014].

De `GDL` au réseau de propositions

Dans cette partie, nous considérons que si les mots-clés *input* et *base* ne sont pas utilisés, un *grounder* a permis de les modéliser.

La modélisation d'un jeu GDL en un réseau de propositions commence par modéliser l'ensemble des propositions de faits. Tout d'abord par les différents prédicats déduits du mot-clé *base* et ensuite par les propositions d'actions déduits du mot-clé *input*.

Enfin, les propositions de vues sont construites sur les règles logiques dont la tête utilise le mot-clé *next* en construisant les liens entre les propositions de faits et les propositions d'actions en utilisant les connecteurs logiques correspondants au corps de chaque règle logique. À savoir l'utilisation de $\neg$ pour les négations, de $\wedge$ pour chaque prédicat constant du corps de la règle et $\vee$ pour les règles possédants la même tête et le même corps.

Le fait *terminal* correspond à un nœud spécial indiquant la terminaison du réseau de propositions, ainsi, si nécessaire des propositions de vues peuvent être ajoutées. De même, les faits correspondants aux scores et identifiés par le mot-clé *goal* sont ajoutés aux réseaux de contraintes en les connectant au nœud *terminal*.

Les règles GDL correspondantes aux actions légales peuvent être représentées de plusieurs manières. La méthode la plus utilisée consiste à ajouter un nœud à chaque proposition d'action qui ne passe à 1 (pour *vrai*) que si l'action est légale.

Au début de chaque match, les propositions de bases sont initialisées à l'aide des termes constants désignés par le mot-clé *init*. En propageant ces faits dans le réseau de propositions comme la propagation d'un signal dans un circuit électrique, il est possible de connaître directement si le fait *terminal* prend la valeur 1 (pour *vrai*) ou 0 (pour *fausse*). Si *terminal* est *vrai*, les scores sont directement obtenus et sinon les propositions de vues permettent de connaître directement les prochains coups légaux indiqués par la valeur 1 dans le graphe sur des propositions d'actions. Une action peut alors être choisie et la proposition d'action correspondant à l'action passe à 1 (pour *vrai*). Les connecteurs de transition sont alors activés pour recopier leurs valeurs d'entrées sur leur sortie et permettre au réseau de propositions d'atteindre l'état suivant.

Exemple 1.22 *Reprenons le réseau de propositions illustré en figure 1.14.*

Considérons qu'initialement la proposition de fait B est vraie suite aux prédicats init du jeu GDL, comme le montre la figure 1.15. Considérons ensuite que les joueurs réalisent l'action correspondant à la proposition d'action A et ne réalisent pas l'action correspondant à la proposition d'action E comme le montre la figure 1.16

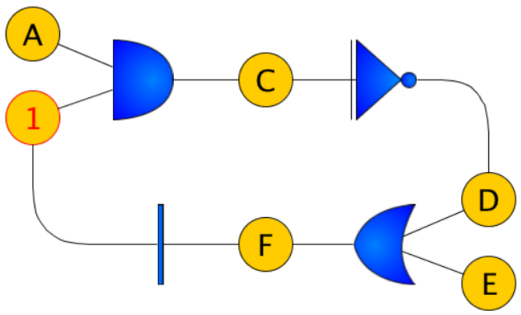


FIGURE 1.15 – Un *propnet* à l'état initial

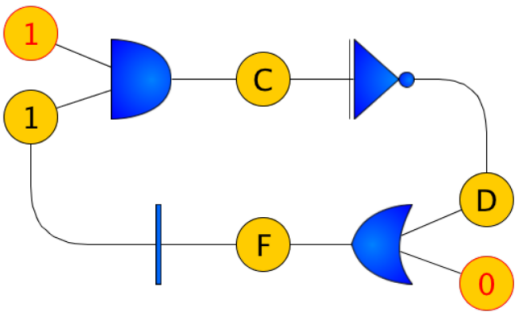


FIGURE 1.16 – Un *propnet* après les actions

Le signal « électrique » est ainsi représenté par les valeurs de A, B et E. Sa propagation permet de déduire les valeurs des propositions de vues C, D et F comme le montre la figure 1.17. Enfin l'application des connecteurs de transitions modifient les valeurs des propositions de faits afin d'atteindre l'état suivant. Dans notre exemple B devient faux comme le montre la figure 1.18.

L'implémentation d'un propnet peut se réaliser par différentes structures que nous ne détaillerons pas

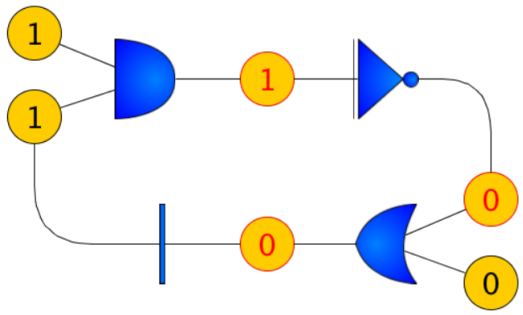


FIGURE 1.17 – Propagation de signal

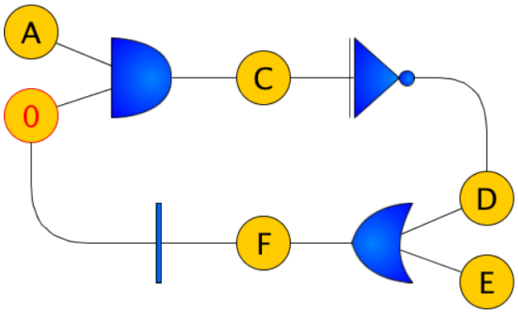


FIGURE 1.18 – État suivant après propagation

dans ce manuscrit, si le lecteur souhaite plus d'informations à ce sujet, il pourra se référer à [GENESERETH & THIELSCHER 2014] chapitre 10.

Au cours de la section suivante, nous nous intéressons aux approches réalisées pour les jeux à information incomplète via le formalisme `GDL-II`.

1.3.7 Approches pour les jeux `GDL-II`

La plupart des approches utilisées pour un jeu `GDL` classique sont adaptables au contexte `GDL-II` car la structure des jeux des deux contextes est semblable. Toutefois, les jeux à information incomplète imposent à un programme-joueur de prendre en considération les informations partielles qu'il acquiert à chaque état afin de déterminer une stratégie efficace pour ce type de jeux. Comme vu dans la section 1.1.2, une méthode usuelle en théorie des jeux à information incomplète est de calculer les « ensembles d'informations ».

Information Set Search (ISS)

Un ensemble d'informations débute par un état initial, complètement connu du joueur, à partir duquel il est possible de calculer tous les états résultants de toutes les actions légales de l'ensemble des joueurs. Ainsi, au cours du jeu et à partir de chaque état potentiel dans lequel un joueur se trouve, il est possible de générer l'ensemble des états pouvant potentiellement résulter de chaque combinaison de mouvements légaux. Les perceptions du joueur sont utilisées comme filtrage pour supprimer les états en contradiction avec ces derniers.

À l'aide de l'ensemble des états possibles pouvant résulter des actions légales, un programme-joueur peut déterminer l'ensemble des actions garanties comme légales. De ce fait, en théorie, l'évaluation d'une action est possible à l'aide d'un ensemble d'informations complet à chaque état du jeu. Par conséquent, réutiliser le principe de l'algorithme Minimax est envisageable pour déterminer l'action à réaliser avec le plus haut potentiel de réussite.

Cette variante de Minimax pour les jeux à information incomplète est dénommée `ISS` (pour *Information Set Search*) [PARKER *et al.* 2010]. Cette méthode est utilisée par `Fluxii`, un programme-joueur pour `GDL-II` adapté de `FluxPlayer` [SCHIFFEL & THIELSCHER 2007b] dédié à un programme `GDL` classique. Ainsi en réutilisant la détection automatique d'heuristiques de `FluxPlayer` et l'algorithme `ISS`, `Fluxii` a remporté la `GO-GGP`, la première compétition de `GGP` pour `GDL-II`.

Toutefois, dans la plupart des jeux `GDL-II`, il existe un grand nombre d'entités non visibles par chaque joueur et donc un nombre conséquent d'ensemble d'informations. De ce fait, maintenir l'ensemble complet des ensembles d'informations tout au long du jeu est impossible.

MCTS pour GDL-II

Le programme-joueur StarPlayer utilisant HyperPlay [SCHOFIELD *et al.* 2012] propose une solution permettant d'appliquer une méthode similaire à MCTS. Plutôt que générer l'ensemble des ensembles d'informations, il propose de restreindre la génération des états potentiels suivants uniquement à quelques uns d'entre eux, nommés états hypothétiques, en sélectionnant aléatoirement quelques combinaisons d'actions à l'état courant. Puis au tour suivant, si les perceptions du joueur sont en contradiction avec les états hypothétiques choisis, ces états sont supprimés et remplacés par d'autres états choisis eux aussi au hasard.

Individuellement, chaque état hypothétique peut être traité par un programme-joueur comme si il était l'état courant d'un jeu à information complète afin d'appliquer les meilleures techniques de MCTS. Les résultats obtenus sur chaque action possible d'un état hypothétique peuvent être combinées en une valeur d'utilité pour l'état et permettre de déterminer l'état hypothétique, le plus favorable et donc l'action présentant le plus grand avantage. Notons que cette approche est facilement parallélisable en traitant chaque état hypothétique par un fil.

Cependant, cette méthode possède de nombreux désavantages. Évaluer un nombre fini d'états hypothétiques comme des états d'un jeu à information complète présente ses limites. En effet, en pratique, il n'est possible de simuler qu'un petit nombre d'états face au nombre exponentiel d'états possibles provoquant souvent le calcul de stratégies faussées. De plus, ne considérer qu'un sous-ensemble des états possibles ne garantit pas la légalité des actions à chaque tour du joueur. Finalement, le plus grand désagrément est lié au fait de considérer l'état d'un jeu GDL-II comme à information complète. Une conséquence non négligeable est qu'une action n'est pas forcément utile lors la simulation de l'utilité d'un état si les informations perçues lors de cet état ne sont pas en corrélation avec cette action.

Plus récemment, [SCHOFIELD & THIELSCHER 2015] propose HyperPlayII une méthode basée sur HyperPlay permettant d'englober des ensembles d'informations plus important et permettant d'éviter la simulation d'ensemble d'informations inutile. Toutefois, cette technique n'a pas encore été employée lors d'une compétition mais se révèle plus efficace qu'HyperPlay expérimentalement.

Finalement, UCT peut également être adapté pour les jeux à information incomplète [SCHÄFER 2008]. Les programmes-joueurs CadiaPlayer, Nexusbaum, et TIIGR pour GDL-II utilisent cet algorithme. CadiaPlayer a remporté AI GGP, la seconde compétition GGP pour GDL-II suite à l'expérience de ces concepteurs dans les différentes compétitions IGGPC en GDL classique.

Minimisation de regret

Une autre approche pour les jeux à information incomplète est la minimisation de regret. Cette technique permet de minimiser la différence entre l'utilité obtenue suite à la stratégie actuelle et les utilités des autres stratégies si des décisions différentes auraient été prises précédemment.

Définition 1.40 (Regret) *Soit un jeu $G = (J, A, \boldsymbol{u})$ et un vecteur d'actions $\boldsymbol{a} = (a_1, a_2, \ldots, a_k)$. Le regret du joueur j jouant l'action a_j est :*

$$R_j(a_j) = max_{a' \in A_j}\, u_j(a'_j, \boldsymbol{a}_{-j}) - u_j(a_j, \boldsymbol{a}_{-j})$$

Le regret est une notion couramment utilisée en apprentissage. Pour définir ce concept dans un cadre temporel il est nécessaire de définir le regret moyen séquentiel.

Définition 1.41 (Regret moyen séquentiel) *Soit st_j^t la stratégie utilisée par le joueur j au temps t. Le regret moyen séquentiel du joueur j au temps T est*

$$R_j^T(a_j) = \frac{1}{T} max_{st_j^* \in St_j} \sum_{t=0}^{T} (u_j(st_j^*, \boldsymbol{st}_{-j}^t) - u_j(st^t))$$

Une stratégie de minimisation de regret assure à un joueur que sa stratégie est la meilleure possible face aux autres stratégies, quelque soit le comportement des autres joueurs. CFR (pour *CounterFactual Regret*) propose un algorithme itératif basé sur une forme spécifique d'apprentissage en ligne dénommée la minimisation de regret contrefactuel. La minimisation de regret contrefactuel réalisée par CFR permet de rechercher un équilibre de ϵ-Nash. Le lecteur pourra se référer à [ZINKEVICH *et al.* 2008] pour plus d'informations.

CFR est l'une des méthodes les plus performantes pour les jeux à information incomplète. Par exemple [BOWLING *et al.* 2015] propose un programme-joueur dédié au poker dénommé Cepheus utilisant $CFR+$, une variante de CFR résolvant le *Heads-up Limit Hold'em Poker*.

Son principale avantage lui vient qu'elle se base sur l'ensemble des états d'informations et non sur chaque état du jeu, il lui est donc possible de générer des équilibres de ϵ-Nash pour des jeux associés à des arbres de recherche imposants. Cependant, il n'y a aucune garantie théorique du succès de CFR sur des jeux à plus de deux joueurs ou à somme non nulle. [SHAFIEI *et al.* 2009] propose le premier programme-joueur utilisant CFR en limitant la profondeur de l'arbre à explorer, mais il n'a jamais été mis en pratique lors d'une compétition.

1.4 Conclusion

La capacité de jouer efficacement à n'importe quel jeu de stratégie qu'implique le *General Game Playing* constitue l'un des défis majeurs de l'intelligence artificielle. Cette capacité à amener les chercheurs à confronter diverses approches efficaces mises en avant dans ce chapitre. Bien entendu, le problème GGP ne se cantonne par aux « univers ludiques » : il permet de modéliser des problèmes de prise de décision séquentielle mono ou multi-agents, possèdent des applications en robotiques ([RAJARATNAM & THIELSCHER 2015] et [THIELSCHER 2015]) et combinant tout un panel de compétences en IA, il permet l'éducation de ce domaine [THIELSCHER 2011c].

Les meilleurs programmes-joueurs actuels utilisent les techniques de MCTS accompagnées par de nombreuses améliorations. La qualité des actions réalisées augmentent avec la quantité de simulations effectuées. C'est pourquoi en plus des améliorations étudiées et utilisées, l'étude de la parallélisation des techniques de MCTS et l'apparition des réseaux de propositions ont permis de diminuer le temps utilisé pour interpréter les jeux GDL et ainsi ont augmenté le nombre de simulations possibles par seconde.

Ceci a permis à Sancho et Galvanise de devenir champion de GGP en 2014 et 2015 en optimisant différentes heuristiques pouvant être utilisées via à un $pronet$ et à $LeJoueur$ de gagner la compétition *2015 Open Tiltyard* par l'utilisation des méthodes de parallélisation. De nombreuses autres voies sont actuellement empruntées par la recherche pour optimiser les programmes-joueurs : Recherche automatique de nouvelles heuristiques génériques via $propnet$, décomposition de jeux GDL en sous-jeux [ZHAO *et al.* 2009], etc. Au cours de ce travail, nous proposons une nouvelle approche au *General Game Playing* basée sur la programmation par contraintes.

Chapitre 2

Programmation par contraintes

Sommaire

2.1 Réseaux de contraintes 55
2.1.1 Définitions et notations 56
2.1.2 Exemple de représentation : Le problème des n-reines 60
2.2 Problème de Satisfaction de contraintes (CSP) 61
2.2.1 Inférence 63
2.2.2 Arc-cohérence 65
2.2.3 Arc-cohérence généralisée 72
2.2.4 Stratégies de recherche 79
2.3 Réseaux de contraintes stochastiques (SCSP) 90
2.3.1 Définitions et Notations 90
2.3.2 Algorithmes de résolutions 94
2.3.3 Quelques algorithmes incomplets 98
2.4 Conclusion 101

Le problème de satisfaction de contraintes (ou CSP pour *Constraint Satisfaction Problem*) est l'un des thèmes de recherche les plus actifs en Intelligence Artificielle. Un grand nombre de problèmes en Intelligence Artificielle et d'autres domaines en informatique peuvent être considérés comme des cas particuliers de problèmes CSP ([NADEL 1990] et [BUSCEMI & MONTANARI 2008]) Dans ce chapitre, une introduction au problème CSP est effectuée, de sa définition formelle aux algorithmes actuellement utilisés en pratique pour sa résolution.

2.1 Réseaux de contraintes

La programmation par contraintes a été développée dès les années 70 afin de formaliser et résoudre informatiquement de nombreux problèmes. Cette formalisation s'appuie sur la construction d'un réseau de contraintes permettant de modéliser le problème à résoudre. Toutefois, avant d'introduire le concept de réseau de contraintes et le problème de satisfaction associé, il convient dans un premier temps de définir les composants au cœur de ce cadre.

2.1.1 Définitions et notations

Une **variable** est une entité à laquelle il est possible d'affecter une valeur. À chaque variable est associée un **domaine** : il s'agit de l'ensemble des valeurs distinctes pouvant être affectées à cette variable. Une variable est discrète (resp. continue) si elle est associée à un domaine discret [15] (resp. continu). Une variable est soit assignée (une valeur lui a été affectée), soit libre (aucune valeur ne lui a été affectée). Une variable dont le domaine ne contient qu'une seule valeur est dite singleton. Nous notons *dom(x)* le domaine fini de valeurs associé à une variable x.

Remarque 2.1 *Pour les différents exemples présentés dans ce chapitre, des lettres de l'alphabet seront utilisées (sauf indication contraire) pour désigner les valeurs contenues dans les domaines des variables : par exemple, la variable x telle que dom(x) = {a,b} est une variable dont le domaine contient les valeurs a et b.*

La valeur a appartenant à *dom(x)* est notée aussi (x, a), on parle de couple (variable,valeur). Affecter une valeur $a \in$ *dom(x)* à x consiste à restreindre *dom(x)* à $\{a\}$: la variable x est alors assignée (par la valeur a) et on note $x = a$. Une valeur est valide pour une variable si elle appartient au domaine de cette variable. La taille de *dom(x)*, représentée généralement par |*dom(x)*| (cardinalité de l'ensemble *dom(x)*), correspond au nombre de valeurs appartenant à *dom(x)*.

Définition 2.1 (Instanciation) *Une instanciation I d'un ensemble $X = \{x_1, \ldots, x_q\}$ de q variables est une application qui associe à chaque variable x_i, une valeur $I[x_i] \in dom(x_i)$.*

Nous notons *vars(I)* l'ensemble des variables dans une instanciation I. Soit $x \in vars(I)$: la valeur associée à la variable x dans I est notée $I[x]$. Une instanciation I est valide si et seulement si $\forall x \in vars(I), I[x] \in dom(x)$. Soit X l'ensemble de variables : une instanciation I est dite complète sur X si $\forall x \in X, x \in vars(I)$, sinon elle est dite partielle.

Exemple 2.1 *Soit un ensemble de variables $X = \{x, y, z\}$ avec dom(x) = dom(y) = dom(z) = $\{i, j, k\}$.*
— *$I = \{(x, i)\}$ est une instanciation partielle sur X avec $vars(I) = \{x\}$ et $I[x] = i$;*
— *$I = \{(x, i), (y, k), (z, j)\}$ est une instanciation complète sur X avec $vars(I) = \{x, y, z\}$.*

Un réseau de contraintes formé d'un ensemble de variables, est également composé d'un ensemble de contraintes appliquées sur ces dernières et leurs domaines. Une relation définie sur une séquence $D_1, D_2, \ldots, D_q$ d'ensembles finis d'entiers est un sous-ensemble du produit cartésien $\prod_{i=1}^{q} D_i$.

Dans le cadre d'un réseau de contraintes, nous parlerons de produit cartésien des domaines de variables. Soient $dom(x_1), dom(x_2), \ldots, dom(x_q)$ les domaines respectifs de q variables $x_1, x_2, \ldots, x_q$: le produit cartésien $dom(x_1) \times dom(x_2) \times \ldots \times (dom(x_q)$, noté aussi $\prod_{i=1}^{q} dom(x_i)$, correspond à l'ensemble $\{(v_1, \ldots, v_q) \mid \forall 1 \leq i \leq q, v_i \in dom(x_i)\}$. Une **relation** définie sur $\prod_{i=1}^{q} dom(x_i)$ est un sous-ensemble de $\prod_{i=1}^{q} dom(x_i)$.

Exemple 2.2 *Soient trois variables x, y, z avec $dom(x) = \{i, j\}$, $dom(y) = \{i, k\}$ et $dom(z) = \{j, k\}$. Le produit cartésien des domaines de ces trois variables correspond à :*
$dom(x) \times dom(y) \times dom(z) = \{(i, i, j), (i, i, k), (i, k, j), (i, k, k), (j, i, j), (j, i, k), (j, k, j), (j, k, k)\}$.
Un exemple de relation sur le produit cartésien des domaines de ces trois variables est donné par : $\{(i, k, j), (i, k, k), (j, k, j), (j, k, k)\}$.

Définition 2.2 (Contrainte) *Une contrainte c porte sur un ensemble de variables appelé portée (ou scope) noté $scp(c)$. Toute variable de $scp(c)$ est dite impliquée dans c. Elle est définie par une relation*

15. Dans ce manuscrit, nous considérons uniquement les domaines finis

notée $rel(c)$ qui autorise (et indirectement interdit) les valeurs que peuvent prendre simultanément (dans leurs domaines) les variables appartenant à sa portée et qui vérifie $rel(c) \subseteq \prod_{x \in scp(c)} dom(x)$. L'arité d'une contrainte c correspond au nombre de variables impliquées dans sa portée, c'est à dire $|scp(c)|$. Dans le cas où cette arité vaut : 1, la contrainte est dite unaire ; 2, la contrainte est dite binaire ; $n > 2$, la contrainte est dite n-aire. Afin de simplifier l'écriture, nous notons c_S, la contrainte dont $scp(c) = S$.

Deux variables appartenant à la portée d'au moins une même contrainte sont dites voisines ou connectées. Le nombre de contraintes dans lesquelles une variable x est impliquée se dénomme degré, nous le notons $deg(x)$.

Définition 2.3 (Tuple) *Au niveau d'une contrainte c d'arité r, un tuple $\tau = (v_1, \ldots, v_r)$ correspond à une instanciation de l'ensemble des variables de $scp(c)$ telle que $\forall 1 \leq i \leq r, v_i \in dom(x_i), x_i \in scp(c)$. Un tuple contenant r valeurs est appelé un r-tuple. Un tuple $\tau = \{(x_1, v_1), \ldots, (x_r, v_r)\}$ est valide si et seulement si $\forall 1 \leq i \leq r\ v_i \in dom(x_i)$, sinon il est invalide. Un tuple τ est autorisé (resp. interdit) dans une contrainte c si et seulement si $\tau \in rel(c)$ (resp. $\tau \notin rel(c)$). Un tuple τ est support (resp. conflit) dans une contrainte c si et seulement si τ est un tuple valide et autorisé (resp. interdit) dans c.*

La valeur affectée à une variable x dans un tuple τ est notée $\tau[x]$. Chaque élément du produit cartésien $\prod_{i=1}^{r} dom(x_i)$ des domaines des variables de la portée de c est représenté par un r-tuple.

Exemple 2.3 *Soit c une contrainte d'arité 3 et de portée $scp(x, y, z)$ avec $dom(x) = dom(y) = dom(z) = \{i, j, k\}$ et soit une instanciation $I = \{(x, i), (y, k), (z, j)\}$ dans c. Le tuple τ correspondant à I sur c est noté $\tau = (i, k, j)$ avec $\tau[x] = i, \tau[y] = k$ et $\tau[z] = j$. Le tuple (i, j, k) est valide car $i \in dom(x), j \in dom(y)$ et $k \in dom(z)$, par contre le tuple (i, o, k) est invalide car $o \notin dom(y)$.*

Une contrainte peut être représentée en intention, en extension ou de manière implicite. Nous parlons alors de contraintes en intention, de contraintes en extension et de contraintes globales.

Définition 2.4 (Contrainte en intention) *Une contrainte en intention est une contrainte dans laquelle les instanciations autorisées sont définies par un prédicat, c'est à dire une fonction qui associe vrai ou faux à toute instanciation. Ce prédicat est défini à partir d'opérateurs logiques, relationnels et arithmétiques.*

Définition 2.5 (Contrainte en extension) *Une contrainte en extension est une contrainte dont les relations sont représentées sous forme de listes de tuples. On parle de relation de type support (resp. conflit) quand il s'agit de la liste des tuples autorisés (resp. interdits) par la contrainte.*

Une contrainte en extension est aussi appelée contrainte table. Nous noterons $rel(c)$ la liste des tuples autorisés dans c et $\overline{rel}(c)$ la liste des tuples interdits dans c. Le nombre de tuples autorisés (resp. interdit) d'une contrainte c est noté $|rel(c)|$ (resp. $|\overline{rel}(c)|$). Le i-ème tuple de c est donné par $rel_i(c)$. Le nombre de tuples pouvant être conséquent, il est important de choisir la sémantique de la représentation (support ou conflit) qui va permettre de minimiser le nombre de tuples de la relation.

Exemple 2.4 *Soit la contrainte c avec $scp(c) = \{x, y\}$ où $dom(x) = dom(y) = \{i, j, k\}$. Sa représentation en intention et en extension :*

- *$rel(c)$: $x \neq y$ est la représentation de c en intention ;*
- *$rel(c) = \{(i, j), (i, k), (j, i), (j, k), (k, i), (k, j)\}$ (ou $\overline{rel}(c) = \{(i, i), (j, j), (k, k)\}$) est la représentation de c en extension. On préférera la sémantique en conflit pour c qui est représentée par moins de tuples.*

Le tuple valide (k, j) est autorisé car il appartient à $rel(c)$, il s'agit donc d'un tuple support dans c. Par contre, le tuple valide (j, j) est interdit car il n'appartient pas à $rel(c)$: il s'agit donc d'un tuple conflit.

Définition 2.6 (Projection/restriction d'une instanciation) *Soit un ensemble de variables X et une instanciation I. La projection/restriction de I sur X, notée $I[X]$ est définie par $I[X] = \{(x, i) \mid (x, i) \in I \wedge x \in X\}$.*

Définition 2.7 (Instanciation satisfaisante/insatisfaisante) *Une instanciation I couvre une contrainte c si et seulement si $scp(c) \subseteq vars(I)$. Une instanciation I satisfait une contrainte c si et seulement si elle couvre c et que le tuple correspondant à la restriction de I sur $scp(c)$ est un support dans c ; on dit également que la contrainte c est satisfaisante par I. À contrario, l'instanciation I ne satisfait pas la contrainte c si elle couvre cette contrainte et que le tuple correspondant à la restriction de I sur $scp(c)$ est un conflit dans c ; la contrainte c est alors insatisfaite (ou violée) par I.*

Lorsqu'une contrainte est satisfaite quelque soit l'instanciation considérée, cette contrainte est dite **universelle**. Formellement, une contrainte $c \in C$ est dite universelle si et seulement si c est satisfaite par toutes les instanciations de $rel(C)$.

Définition 2.8 (Contrainte globale) *Une contrainte globale est une contrainte avec une sémantique implicite et dont la portée peut contenir un nombre quelconque de variables.*

Ce type de contraintes permet généralement de capturer des problèmes récurrents (ré-utilisabilité) et d'intégrer des techniques spécifiquement dédiées (c'est à dire spécifiquement adaptées à la sémantique de la contrainte).

Exemple 2.5 *Une des contraintes globales les plus connues est sans doute la contrainte AllDifferent dont la sémantique impose que les valeurs affectées aux variables de sa portée soient toutes différentes. Elle peut être définie pour deux variables, AllDifferent(x,y) (similaire à la contrainte en intention c avec $rel(c) : x \neq y$), tout comme elle peut être définie pour n variables, $AllDifferent(x_1, \ldots, x_n)$. La sémantique est la même dans ces deux utilisations : que ce soient pour deux ou n variables, les valeurs choisies doivent être toutes différentes. Même si la contrainte globale AllDifferent est équivalente à un ensemble de contraintes binaires, son intérêt réside dans sa technique de filtrage dédiée : pour plus d'informations, voir [*RÉGIN *1994] et [*PUGET *1998].*

Définition 2.9 (Réseau de contraintes) *Un réseau de contraintes $\mathcal{P}$ (ou CN pour Constraint Network) est défini par un couple (X, D, C) où :*

- *$X = \{x_1, x_2, \ldots, x_n\}$ est l'ensemble fini des n variables du problème ;*
- *$D = \{dom(x_1), dom(x_2), \ldots, dom(x_n)\}$ est l'ensemble fini des n variables du problème ;*
- *$C = \{c_1, c_2, \ldots, c_m\}$ est l'ensemble fini des m contraintes du problème portant sur les variables composant X.*

L'ensemble X d'un réseau de contraintes $\mathcal{P}$ est noté $vars(\mathcal{P})$ et l'ensemble C de $\mathcal{P}$ est noté $cons(\mathcal{P})$. Pour toute contrainte $c \in cons(\mathcal{P})$, nous avons $scp(c) \subseteq vars(\mathcal{P})$. Nous notons n le nombre de variables de $\mathcal{P}$ ($|vars(\mathcal{P})|$), d la taille du plus grand domaine des variables de $\mathcal{P}$, m le nombre de contraintes ($|cons(\mathcal{P})|$) et r l'arité maximale des contraintes dans $\mathcal{P}$. L'arité maximale r de $\mathcal{P}$ permet de qualifier et de catégoriser le réseau de contraintes. La nomination des réseaux de contraintes selon r reprend celle des contraintes de la définition 2.2 selon l'arité associée. À savoir, un réseau de contraintes tel que $r = 1$ et $r = 2$ est appelé respectivement réseau unaire et binaire. Pour $n > 2$, nous parlerons de réseau n-aire.

Exemple 2.6 *Soit le réseau de contraintes $\mathcal{P} = (X, D, C)$ avec :*

- *$X = \{x_1, x_2, x_3, x_4, x_5\}$;*
- *$dom(x_1) = dom(x_2) = dom(x_3) = dom(x_4) = dom(x_5) = \{0, 1\}$;*
- *$C = \{c_1, c_2, c_3\}$ tel que $c_1 = \{x_1 + x_2 < x_4\}$, $c_2 = \{x_4 \neq x_5\}$, $c_3 = \{x_3 = x_5\}$.*

L'arité maximale r de $\mathcal{P}$ étant l'arité de c_1 à savoir 3, le réseau de contraintes $\mathcal{P}$ est alors qualifié de 3-aire.

Un réseau de contraintes est **normalisé** si et seulement si quelles que soient les contraintes c_1 et $c_2 \in cons(\mathcal{P})$, $c_1 \neq c_2 \Rightarrow scp(c_1) \neq scp(c_2)$. Pour normaliser un réseau, toutes les contraintes de même portée doivent être fusionnées.

Définition 2.10 (Instanciation localement cohérente) *Une instanciation I est localement cohérente sur un réseau de contraintes $\mathcal{P}$ si et seulement si I est valide et que toutes les contraintes couvertes par I sont satisfaites. Si $vars(I) = vars(\mathcal{P})$, I correspond alors une instanciation localement cohérente complète, sinon elle est dite partielle.*

Une **solution** d'un réseau de contraintes $\mathcal{P}$ est une instanciation localement cohérente complète sur $\mathcal{P}$. L'ensemble des solutions d'un réseau de contraintes de $\mathcal{P}$ est noté *sols($\mathcal{P}$).*

Définition 2.11 (Instanciation globalement incohérente) *Une instanciation est globalement incohérente si elle ne peut pas être étendue à une solution, sinon elle est globalement cohérente. Une instanciation globalement incohérente est appelée aussi nogood.*

Propriété 2.1 *Une instanciation localement incohérente est nécessairement globalement incohérente, cependant la réciproque n'est pas vraie.*

Un réseau de contraintes est **satisfaisable** si et seulement si il admet au moins une solution, sinon il est insatisfaisable. Deux réseaux de contraintes $\mathcal{P}$ et $\mathcal{P}'$ sont équivalents si et seulement si $sols(\mathcal{P}) = sols(\mathcal{P}')$.

Propriété 2.2 *Tout réseau de contraintes peut être transformé en temps polynomial en un réseau binaire de contraintes équivalent.*

À ce jour, plusieurs procédés de transformation ont été proposés : l'une des premières appelée méthode duale [DECHTER & PEARL 1989], nous vient du domaine des bases de données, et consiste à générer un réseau binaire de contraintes en transformant chaque contrainte du problème original en variables de la nouvelle représentation. Quelques modifications syntaxiques permettent ensuite d'obtenir un réseau binaire de contraintes équivalent. Une autre technique connue est la transformation « avec variables cachées » [DECHTER 1990b], qui permet cette fois de conserver les variables originales du problème, même si de nouvelles sont introduites pour simuler les contraintes n-aires à travers des contraintes binaires.

Plusieurs procédés ayant pour objectif de visualiser un réseau de contraintes ont été mis au moint. Un réseau de contraites binaire (resp. n-aire) est souvent représenté comme un graphe de contraintes (resp. hypergraphe de contraintes).

Définition 2.12 (Graphe et hypergraphe de contraintes) *À chaque réseau de contraintes binaires $\mathcal{P} = (X, D, C)$ peut être associé un graphe des contraintes obtenu en représentant chaque variable du réseau par un sommet et chaque contrainte binaire $c \in C$ qui porte sur les variables x_i et x_j par une arête entre les sommets x_i et x_j. Dans le cas des réseaux de contraintes n-aires, une représentation par des hypergraphes est utilisée, en remplaçant les arêtes par des hyper-arêtes.*

Exemple 2.7 *La figure 2.1 donne une représentation du réseau de contraintes de l'exemple 2.6.*

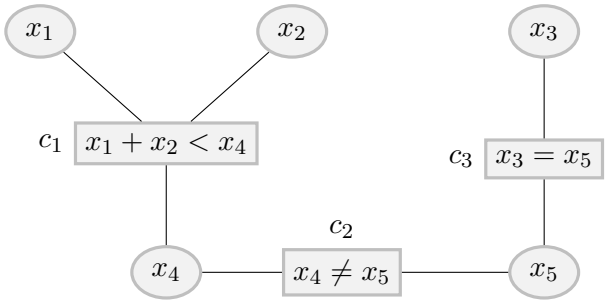


FIGURE 2.1 – Graphe de contraintes d'un CSP

Le graphe de contraintes est un outil intéressant de modélisation d'un réseau de contraintes, mais il ne permet pas de visualiser les tuples autorisés ou interdits entre variables. Pour cela, on définit le graphe de micro-structure d'un réseau de contraintes.

Définition 2.13 (Graphe de micro-structure) *Tout réseau de contraintes composé de n variables peut être représenté par un graphe n-parti appelé graphe de micro-structure. On regroupe au sein de chaque partie de ce graphe des sommets représentant les couples (variable,valeur), et une arête un n-uplet autorisé (ou interdit) par l'une des contraintes du problème.*

On parle de micro-structure de comptabilité si les hyper-arêtes représentent les n-uplets autorisés et de micro-structure d'incompatibilité (ou de micro-structure complémentaire) si les hyper-arêtes représentent les n-uplets interdits.

Remarque 2.2 *Dans la suite de ce manuscrit, lorsqu'aucune information n'est fournie, l'appellation micro-structure désigne une micro-structure de comptabilité.*

De telles représentations sont notamment utilisées afin de détecter (et éliminer) des symétries au niveau du problème ([GENT & SMITH 1999], [FAHLE *et al.* 2001] et [COHEN *et al.* 2006]) comme nous le verrons en Section 4 ou encore dans le développement de méthodes de décomposition de problèmes ([JÉGOU 1993] et [CHMEISS *et al.* 2003]).

Exemple 2.8 *La figure 2.2 représente la micro-structure du réseau de contraintes de l'exemple 2.6.*

Nous donnons dans ce qui suit un exemple de modélisation d'un problème concret sous la forme d'un réseau de contraintes.

2.1.2 Exemple de représentation : Le problème des n-reines

Le problème des n-reines est un problème « jouet », proposé initialement en 1848 par un joueur d'échecs du nom de Max Bassel. Ce problème consiste à disposer n reines sur un échiquier $n \times n$ de manière à ce qu'aucune d'entre elles ne soit en prise [16] avec les autres.

Ce problème se formalise naturellement à l'aide d'un réseau de contraintes. En effet, en exploitant le fait qu'une seule reine est placée par colonne, le problème se réduit au choix de la ligne. En conséquence de quoi, une modélisation du problème considère chaque colonne i comme une variable x_i de domaine $dom(x_i) = \{1, 2, \ldots, n\}$ où chaque valeur de x_i désigne le numéro de ligne où se place la reine.

16. Deux reines sont en prise si elles se trouvent sur une même colonne, une même ligne ou une même diagonale de l'échiquier.

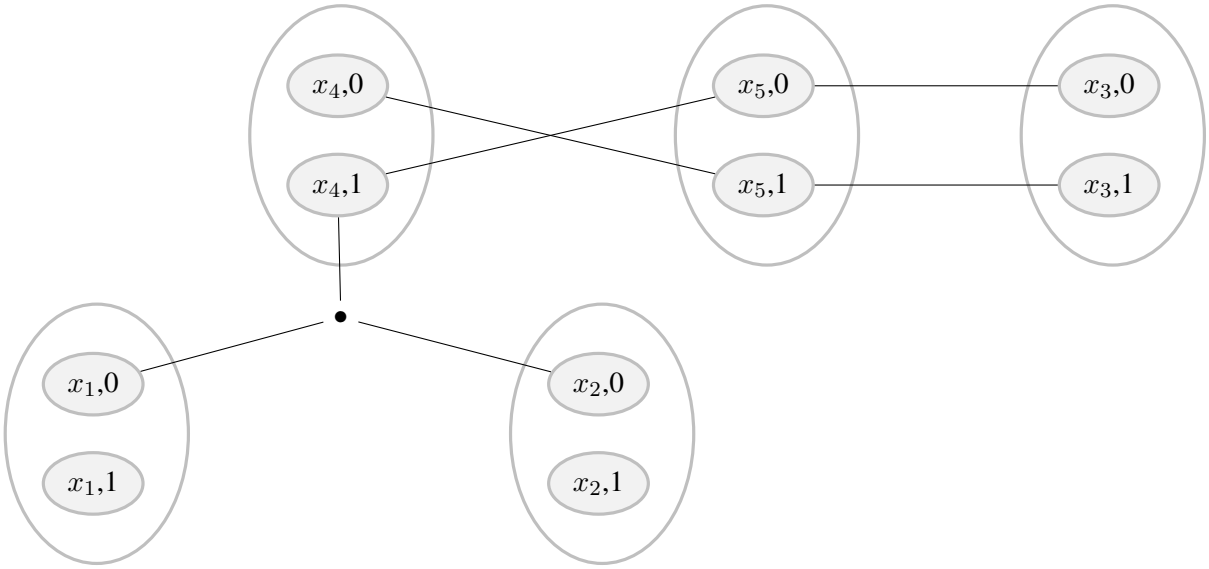


FIGURE 2.2 – Microstructure de compatibilité d'un réseau de contraintes

Les contraintes quant à elles doivent retranscrire l'énoncé du problème, c'est-à-dire que deux reines ne doivent jamais se retrouver sur la même ligne ($x_i \neq x_j$) ou la même diagonale ($|x_i + i - j| \neq x_j$ et $|x_i - i + j| \neq x_j$). Si nous nous plaçons dans le cadre général avec n-reines, nous obtenons le réseau de contraintes $\mathcal{P} = (X, D, C)$ suivant :

— $X = \{x_1, x_2, \ldots, x_n\}$;
— $dom(x_1) = dom(x_2) = \ldots = dom(x_n) = \{1, 2, \ldots, n\}$;
— C est défini par $\forall i, j : (x_i \neq x_j) \wedge (|x_i - x_j| \neq |i - j|)$.

Le réseau de contraintes correspondant au problème des 4-reines illustré par son graphe de contraintes en figure 2.3 ne comporte que 2 solutions indiquées en figure 2.4 :

— L'instanciation $I_1 = \{(x_1 = 3), (x_2 = 1), (x_3 = 4), (x_4 = 2)\}$
— L'instanciation $I_2 = \{(x_1 = 2), (x_2 = 4), (x_3 = 1), (x_4 = 3)\}$

La résolution d'un réseau de contraintes consiste (généralement) à trouver une solution ou prouver l'absence de solution(s). Il s'agit du problème de satisfaction de contraintes (ou CSP pour *Constraint Satisfaction Problem*). Toutefois, afin de caractèriser ce problème, il est utile de le situer dans la hiérarchie polynomiale. Pour plus de précisions, le lecteur pourra se référer aux ouvrages : [TURING *et al.* 1991], [PAPADIMITRIOU 1994] et [CREIGNOU *et al.* 2001].

2.2 Problème de Satisfaction de contraintes (CSP)

La résolution d'un réseau de contraintes consiste (généralement) à trouver une solution ou prouver l'absence de solution(s). Il s'agit du problème de satisfaction de contraintes (aussi nommé CSP pour *Constraint Satisfaction Problem*). La démarche consiste donc à modéliser un problème sous la forme d'un réseau de contraintes (abordé dans la section 2.1), puis de le résoudre dans le cadre de la programmation par contraintes [ROSSI *et al.* 2006] à l'aide d'un solveur.

Définition 2.14 (Problème de satisfaction de contraintes) *Le problème de satisfaction de contraintes (ou CSP pour Constraint Satisfaction Problem) est le problème de décision qui consiste à savoir si un*

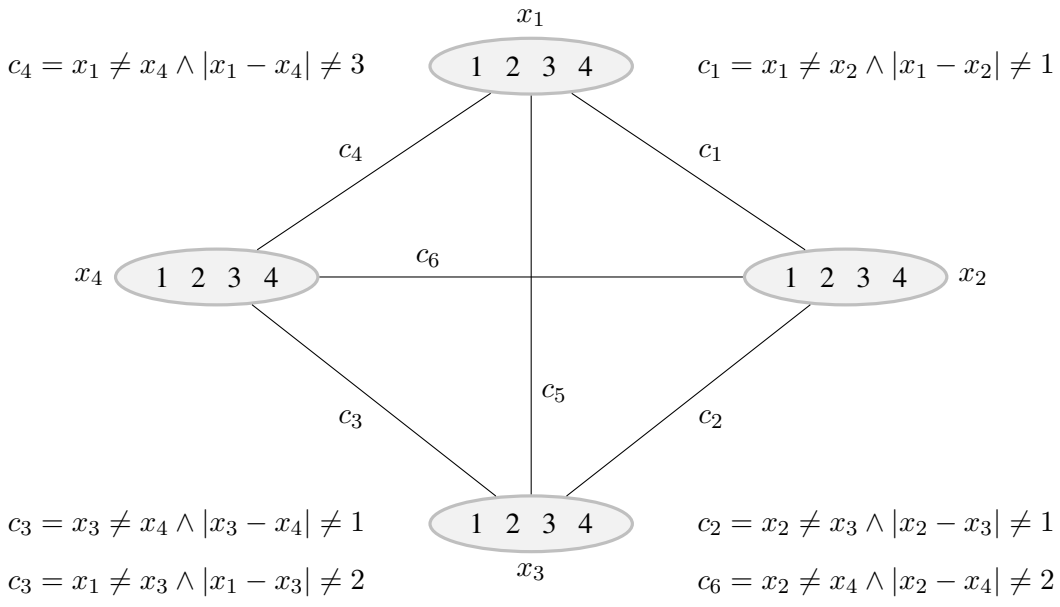
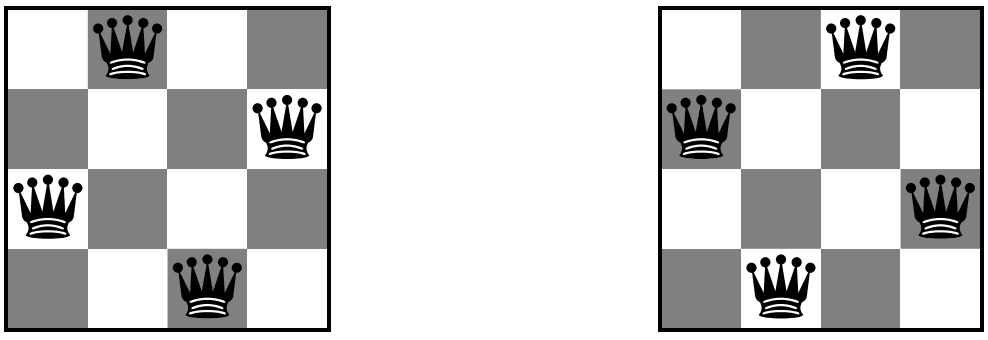


FIGURE 2.3 – Graphe de contraintes du problème des 4-reines



$I_1 = \{(x_1 = 3), (x_2 = 1), (x_3 = 4), (x_4 = 2)\}$ $I_2 = \{(x_1 = 2), (x_2 = 4), (x_3 = 1), (x_4 = 3)\}$

FIGURE 2.4 – Les deux solutions du problème des 4-reines.

réseau de contraintes est satisfaisable ou non. Par abus de langage, un réseau de contraintes peut aussi être appelé CSP.

Propriété 2.3 *Décider si un réseau de contraintes est satisfaisable est un problème NP-complet.*

Dans le cadre de la programmation par contraintes, la modélisation et la résolution d'un CSP sont deux étapes distinctes. Nous utilisons un programme appelé solveur de contraintes pour résoudre un réseau de contraintes. Les solveurs sont des programmes qui traitent des réseaux de contraintes et qui prouvent la satisfaisabilité d'un réseau en retournant au moins une solution, soit son insatisfaisabilité. Il existe des solveurs dédiés, développés pour la résolution efficace de problèmes spécifiques et des solveurs génériques qui vont être capables de traiter tout type de problème dans une efficacité relativement correct par rapport aux solveurs spécifiquement dédiés. À titre d'exemple, différents solveurs existent : des solveurs propriétaires comme *Comet* [17] et *ILOG* [18], et des solveurs libres comme *AbsCon* [19], *Choco* [20], *Gecode* [21] et *Minion* [22].

Généralement les solveurs de contraintes résolvent des CSP en alternant des phases d'inférence et de recherche. Nous présentons les spécificités de ces phases dans les deux sections suivantes.

2.2.1 Inférence

Les principales méthodes de résolutions de CSP consistent en une énumération des solutions potentielles à travers le parcours d'un arbre de recherche, associée à une opération de filtrage éliminant les affectations trivialement non solutions au problème. Dans ce paragraphe, nous présentons l'opération à l'origine du filtrage dans les arbres de recherche, l'inférence consistant à faire des déductions qui vont permettre de simplifier le réseau de contraintes à résoudre, l'objectif étant d'éviter de parcourir des zones inutiles de l'espace de recherche. Ces déductions vont en effet permettre l'identification et la suppression de valeurs ne pouvant participer à aucune solution pour le réseau. Certaines instanciations globalement incohérentes vont ainsi pouvoir être identifiées et évitées. L'inférence consiste alors à supprimer (ou filtrer) pour chaque variable certaines valeurs de son domaine ne permettant pas d'étendre cette instanciation à une instanciation localement cohérente.

Exemple 2.9 *Reprenons le problème des n-reines présenté dans la sous-section 2.1.2 et plus particulièrement le sous-problème où $n = 4$.*

Considérons l'instanciation partielle $\{(x_1, 1)\}$ positionnant une reine dans la case $(1, 1)$ du plateau.

On peut déduire de cette instanciation que $x_2 \notin \{1, 2\}$, $x_3 \notin \{1, 3\}$ et $x_4 \notin \{1, 4\}$.

Les domaines restants sont réduits à $dom(x_2) = \{3, 4\}$, $dom(x_3) = \{2, 4\}$ et $dom(x_4) = \{2, 3\}$.

Toutefois, si $x_2 = 3$ alors $x_3 \notin \{1, 3\}$. x_3 ne pouvant plus être instanciée à aucune valeur de son domaine sans violer une contrainte x_2 est fixée à 4 entrainant la déduction que $x_3 \notin \{4\}$ ($dom(x_3) = \{2\}$) et $x_4 \notin \{4\}$ ($dom(x_4) = \{3\}$).

Ainsi, x_3 est fixée à (la dernière valeur de son domaine) 2, on en déduit que $x_4 \notin \{3\}$. x_4 ne peut donc plus être instanciée sans violer une contrainte.

Suite à ces déductions, le mécanisme d'inférence utilisé ici nous signale que l'instanciation $\{(x_1, 1)\}$ ne peut être étendue à une instanciation localement cohérente.

Nous pouvons donc simplifier le réseau de contraintes dans lequel $dom(x_1) = \{2, 3, 4\}$.

17. *Comet tutorial* : http://www.cs.upc.edu/~larrosa/comet.pdf
18. *ILOG Solver website :* www.ilog.com
19. *AbsCon website* : http://www.cril.univ-artois.fr/~lecoutre/software.html
20. *Choco constraint programming system website* : http://choco.sourceforge.net
21. *Generic constraint development environment website* : http://www.gecode.org
22. *Minion Open Source software website* : http://minion.sourceforge.net/

Même si dans le pire des cas le problème CSP n'est pas solvable en temps polynomial, la majorité des algorithmes d'inférence s'exécute en temps polynomial. L'inférence est donc centrale dans le cadre de la programmation par contraintes et permet d'aider à la résolution d'un grand nombre d'instances du problème CSP. En pratique, on cherche à établir une propriété particulière sur le réseau de contraintes appelée *cohérence* locale. Cette propriété peut être établie sur le réseau initial ou/et maintenue lors du processus de résolution.

Remarque 2.3 *Dans la littérature, la notion de cohérence est parfois désignée par la notion de consistance.*

La forme de cohérence locale la plus simple est la 1-cohérence, appelée aussi cohérence de nœud. Elle concerne les contraintes unaires et permet de vérifier que toutes les valeurs des variables sous la portée d'une contraine unaire sont cohérentes.

Définition 2.15 (cohérence de nœud) *Soient un CSP $\mathcal{P} = (X, D, C)$ et $c \in C$ telle que $scp(c) = \{x\}$. Nous avons :*

- *Une valeur $i \in dom(x)$ est dite 1-cohérente si et seulement si $i \in rel(c)$;*
- *Une variable est dite 1-cohérente si et seulement si $\forall i \in dom(x)$, i est 1-cohérente ;*
- *Le CSP $\mathcal{P}$ est dit 1-cohérent si et seulement si $\forall x \in X$, x est 1-cohérente.*

Exemple 2.10 *Considérons le réseau de contrainte $\mathcal{P} = (X, D, C)$ tel que $X = \{x, y\}$ avec $dom(x) = dom(y) = \{1, 2, 3, 4\}$ et $C = \{c_1 : x > 2, c_2 : x \neq y\}$. La cohérence de nœud peut être appliquée à c_1 qui est unaire ($|scp(c_1)| = 1$) comme le montre la figure 2.5. Elle permet de supprimer les valeurs 1 et 2 de $dom(x)$. Le réseau de contraintes obtenu après ces suppressions est 1-cohérent.*

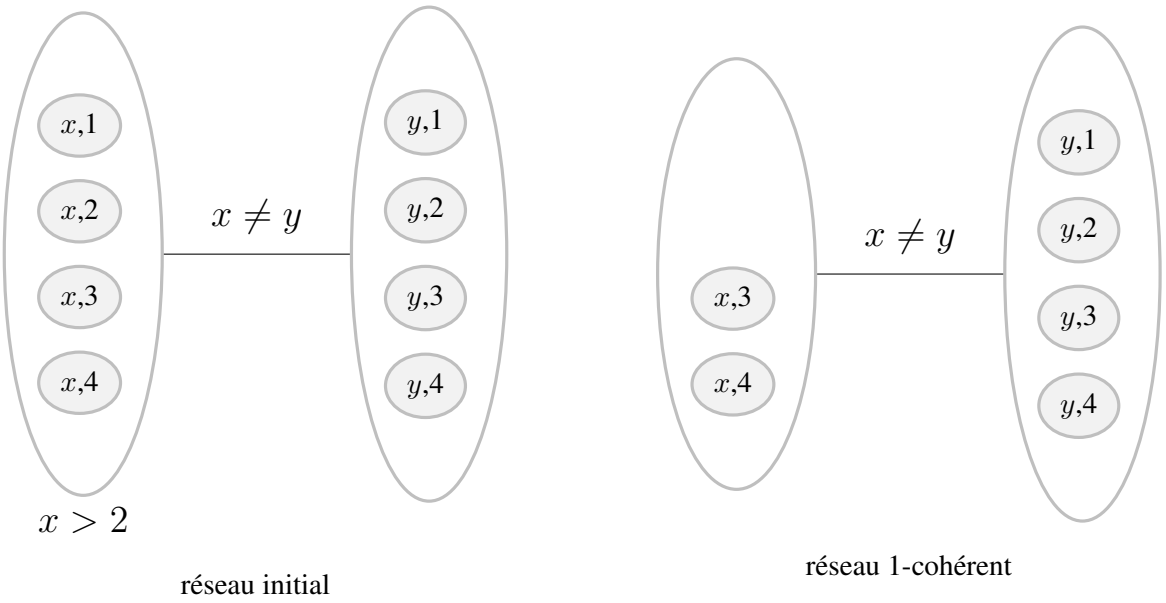


FIGURE 2.5 – Suppression de valeurs par cohérence de nœud.

Afin d'établir cette forme de cohérence il suffit de réduire le domaine de chaque variable aux valeurs qui satisfont les contraintes unaires de cette variable. Après avoir rendu le réseau de contraintes 1-cohérent, les contraintes unaires peuvent être supprimées du réseau.

La propriété de 1-cohérence étant limitée uniquement aux contraintes unaires, intéressons nous à la k-cohérence permettant de redéfinir les notions de cohérences au cadre général.

Définition 2.16 (k-cohérence) *Soit un CSP $\mathcal{P}$ de n variables et k un entier tel que $1 \leq k < n$. Le réseau $\mathcal{P}$ est k-cohérent [*FREUDER *1978] si et seulement si pour tout ensemble X de k-1 variables, une instanciation I sur X qui est localement cohérente sur $\mathcal{P}$ peut être étendue, considérant une variable $y \in vars(\mathcal{P}) \setminus X$ et une valeur $a \in dom(y)$, en une instanciation localement cohérente I' telle que $I' = I \cup \{(y, a\})$. Sinon, il est k-incohérent.*

Afin de définir la cohérence globale, introduisons à présent la définition de k-cohérence forte.

Définition 2.17 (k-cohérence forte) *Un réseau de contraintes $\mathcal{P}$ avec n variables est k-cohérent fort, avec $1 \leq k < n$ si et seulement si $\mathcal{P}$ est i-cohérent pour tout $1 \leq i \leq k$.*

Définition 2.18 (cohérence globale) *Un réseau de contraintes de n variables est globalement cohérent si et seulement si il est n-cohérent fort.*

Propriété 2.4 *Un réseau de contraintes globalement cohérent est satisfaisable.*

Pour établir la k-cohérence, un algorithme d'inférence va modifier le réseau initial. On parle alors de filtrage par k-cohérence [FREUDER 1978]. Le réseau obtenu correspond généralement à un réseau équivalent plus explicite (ou simplifié). Nous noterons ϕ la cohérence cible et $\phi(P)$ le réseau obtenu en établissant ϕ sur $\mathcal{P}$. Si la cohérence ϕ est vérifiée sur $\mathcal{P}$, on dit que $\mathcal{P}$ vérifie ϕ.

Afin d'illustrer ces notions de cohérence et leur maintien, nous allons dans un premier temps nous intéresser plus particulièrement à la 2-cohérence dans le cadre de réseaux binaires (normalisés). Appelée aussi arc-cohérence, il s'agit d'une cohérence fondamentale en programmation par contraintes car, dans l'absolu, tout réseau de contraintes peut se ramener à un réseau binaire de contraintes équivalents, moyennant une opération de transformation polynomiale. Nous verrons également que de nombreuses tentatives d'optimisation pour établir l'arc cohérence ont été proposées depuis un bon nombre d'années à travers des algorithmes plus ou moins efficaces. Dans un deuxième temps, nous nous intéresserons à l'arc cohérence généralisée dans le cadre de réseaux n-aires (normalisés).

2.2.2 Arc-cohérence

D'après la notion générale de k-cohérence, la 2-cohérence ou arc-cohérence correspond à la propriété que toute instanciation localement cohérente d'une variable peut être étendue en une instanciation localement cohérente de deux variables dans un réseau binaire.

Définition 2.19 (Valeur arc-cohérente) *Soit c une contrainte binaire telle que $scp(c) = \{x, y\}$. Une valeur $a \in dom(x)$ est arc-cohérente dans c si et seulement si il existe au moins une valeur $b \in dom(y)$ telle que $(a, b) \in rel(c)$. La valeur b est appelée support de a pour la contrainte c. De plus, par la propriété de bi-directionnalité [*BESSIÈRE & FREUDER *1999], a est réciproquement support de b pour la contrainte c.*

Définition 2.20 (Contrainte arc-cohérente) *Une contrainte c est arc-cohérente si toutes les valeurs présentes dans le domaine des variables composant $scp(c)$ sont arc-cohérentes dans c.*

Définition 2.21 (Réseau arc-cohérente) *Un CSP $\mathcal{P} = (X, D, C)$ est arc-cohérent si et seulement si chaque $c \in C$ est arc-cohérent.*

En appliquant l'inférence sur chaque contrainte d'un réseau, il est possible de le rendre arc-cohérent. Voici un exemple.

Exemple 2.11 *Considérons un réseau de contraintes $\mathcal{P} = (X, D, C)$ où :*
— $X = \{x, y, z\}$ *avec* $dom(x) = dom(y) = dom(z) = \{i, j, k\}$;
— $C = \{c_1, c_2, c_3\}$ *de portée respective* $scp(c_1) = \{x, z\}$, $scp(c_2) = \{x, y\}$, $scp(c_3) = \{y, z\}$;
— $rel(c_1) = \{(i, j), (i, k), (j, k)\}$, $rel(c_2) = \{(j, i), (k, i), (k, j)\}$ *et* $scp(c_3) = \{(i, k), (j, j), (k, i)\}$.
Sa micro-structure est représentée en figure 2.6

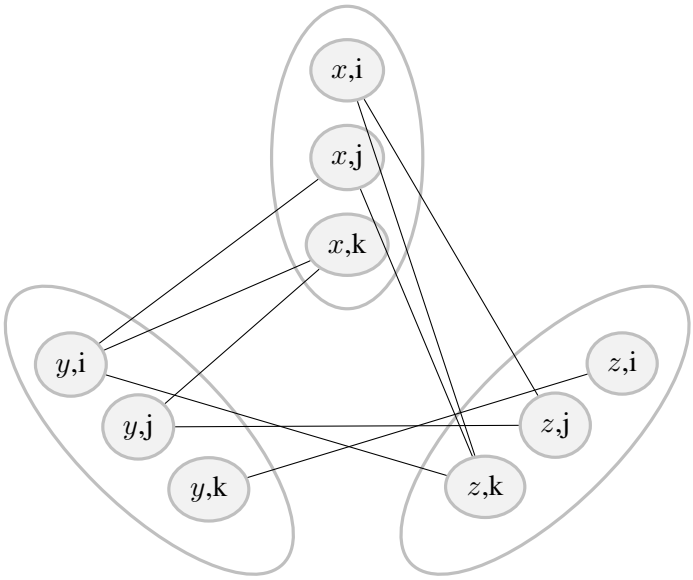


FIGURE 2.6 – Un réseau non arc-cohérent

$\mathcal{P}$ n'est pas arc-cohérent car certaines valeurs appartenant aux domaines des différentes variables ne sont pas arc-cohérentes. Par exemple, la valeur k dans $dom(x)$ n'a pas de support parmi les valeurs dans $dom(z)$ pour la contrainte c_1. Même constat pour la valeur i dans $dom(x)$ pour la contrainte c_2. Cela implique que les contraintes c_1 et c_2 ne sont pas arc-cohérentes (au contraire de c_3). Afin d'établir l'arc-cohérence de $\mathcal{P}$, nous allons effectuer un filtrage par arc-cohérence. Il faut d'abord établir l'arc-cohérence pour les contraintes c_1 et c_2.

Pour cela, les valeurs (x, k) et (z, i), qui n'ont pas de support pour c_1, sont supprimées. La suppression de ces valeurs entraîne également la perte d'un support dans c_2 pour les valeurs (y, i) et (y, j) et dans c_3 pour (y, k), ce qui est représenté par des traits en pointillés dans la figure 2.7. Établir l'arc-cohérence d'une contrainte peut avoir des répercussions sur les autres contraintes du réseau. Dans notre exemple, la contrainte c_3 qui était à l'origine arc-cohérente ne l'est plus car la valeur (y, k) n'a plus aucun support pour cette contrainte. Ce phénomène correspond au processus de propagation de contraintes. Le réseau $\mathcal{P}$ obtenu en figure 2.8 n'est toujours pas arc-cohérent.

Dans un second temps, l'arc-cohérence des contraintes c_2 et c_3 doit être établie. Les valeurs (x, i), (y, j) et (y, k) sont supprimées, ce qui a pour but de faire perdre des supports aux valeurs (z, j) et (z, k) comme illustré par des pointillés en figure 2.9. La valeur (z, j) n'ayant plus de support pour c_3, l'arc cohérence est établie en supprimant (z, j). Le réseau résultant $\phi(\mathcal{P})$ illustré par la figure 2.10, avec l'arc-cohérence, est arc-cohérent : en effet, les contraintes c_1, c_2 et c_3 sont toutes arc-cohérentes.

Dans l'exemple 2.11, nous avons vu que des contraintes à l'origine arc-cohérentes pouvaient devenir

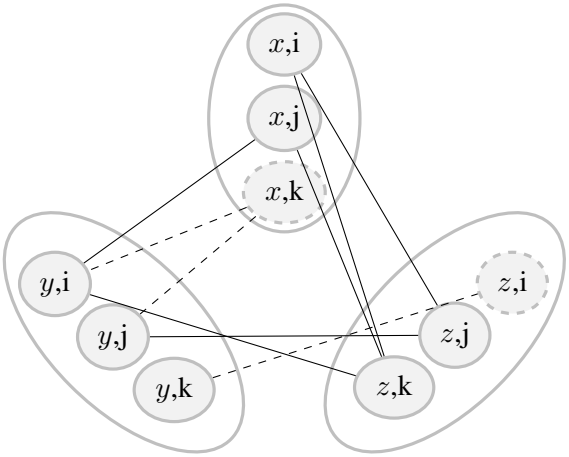


FIGURE 2.7 – On applique l'inférence sur la contrainte c_1

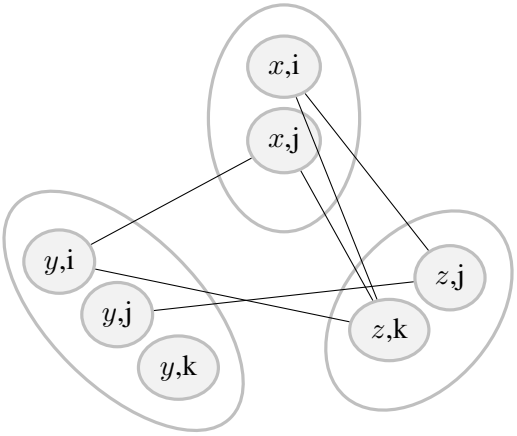


FIGURE 2.8 – Un réseau $\mathcal{P}$ toujours non arc-cohérent

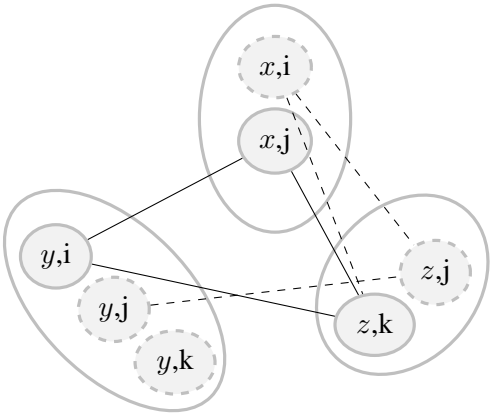


FIGURE 2.9 – On applique l'inférence sur la contrainte c_2 et c_3

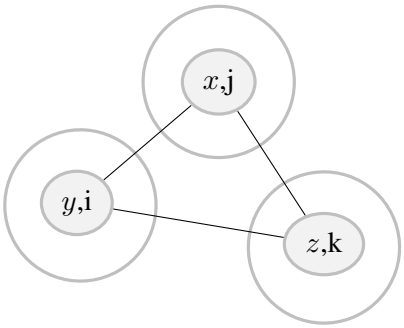


FIGURE 2.10 – $\phi(\mathcal{P})$ est un réseau arc-cohérent.

non arc-cohérentes lors du filtrage par arc-cohérence. Les contraintes peuvent ainsi passer d'un état à l'autre durant tout le processus de filtrage des différentes valeurs non arc-cohérentes. Le processus de filtrage s'exécute tant qu'il n'a pas atteint un point fixe, c'est à dire que l'arc-cohérence est établie pour toutes les contraintes du réseau, incluant le cas où un domaine vide apparaît. Il est important de noter que ce processus est confluent par la propriété de stabilité par union [BESSIERE 2006]. Le réseau obtenu est arc-cohérent et correspond au plus grand sous-réseau équivalent au réseau d'origine : il est unique et représente la fermeture par arc-cohérence.

Un grand nombre d'algorithmes a été proposé dans le but d'établir de façon efficace l'arc-cohérence dans un réseau de contraintes. La plupart d'entre eux se compose d'un algorithme de progagation incorporant une procédure de révision. Ils sont regroupés en deux familles : les algorithmes dits à gros grain et les algorithmes dits à grain fin. Le tableau 2.1 répertorie les algorithmes d'arc-cohérence les plus reconnus et indique la complexité temporelle et spatiale respective à chaque algorithme.

Algorithme	**Comp. temporelle**	**Comp. spatiale**	**Grain**	**Auteur**
$AC3$	$\mathcal{O}(m \times d^3)$	$\mathcal{O}(m)$	Gros	[MACKWORTH 1977]
$AC4$	$\mathcal{O}(m \times d^2)$	$\mathcal{O}(m \times d^2)$	Fin	[MOHR & HENDERSON 1986]
$AC6^{16}$	$\mathcal{O}(m \times d^2)$	$\mathcal{O}(m \times d)$	Fin	[BESSIÈRE 1994]
$AC7$	$\mathcal{O}(m \times d^2)$	$\mathcal{O}(m \times d^2)$	Fin	[BESSIÈRE & FREUDER 1999]
$AC3_d$	$\mathcal{O}(m \times d^3)$	$\mathcal{O}(m + n \times d)$	Gros	[VAN DONGEN 2002]
$AC3.2/3.3$	$\mathcal{O}(m \times d^2)$	$\mathcal{O}(m \times d)$	Gros	[LECOUTRE *et al.* 2003]
$AC2001/3.1$	$\mathcal{O}(m \times d^2)$	$\mathcal{O}(m \times d)$	Gros	[BESSIERE & DEBRUYNE 2005]
$AC3^{rm}$	$\mathcal{O}(m \times d^3)$	$\mathcal{O}(m \times d)$	Gros	[LECOUTRE & HEMERY 2007]

TABLE 2.1 – Algorithmes établissant la cohérence d'arc et leur complexité.

La distinction entre approche à grain fin ou gros grain se fait à travers la nature et le niveau des données stockées et traitées lors de la propagation : triplets *contraintes-variable-valeur* pour le grain fin, couples *contrainte-variable* pour le gros grain. Les procédures de révisions des algorithmes à grain fin sont plus fines et apportent certaines optimisations. Cependant, les structures qu'elles nécessitent sont souvent coûteuses en espace et requièrent une implémentation relativement complexe. Nous présentons plus en détail les algorithmes les plus utilisés à savoir AC3, AC2001/3.1 et $AC3^{rm}$.

Ces trois algorithmes partagent le même algorithme de propagation dénommé $doAC$ présenté dans l'Algorithme 2.1 mais leur particularité est issue de la technique de révision utilisée (appelé à la ligne 4). Avec en entrée un réseau de contraintes $\mathcal{P} = (X, D, C)$, l'algorithme $doAC$ calcule la fermeture de $\mathcal{P}$ par arc-cohérence et si un domaine vide est présent, il retourne faux afin d'indiquer une incohérence globale, *vrai* sinon. Pour cela, il parcourt un ensemble Q contenant tous les couples contraintes-variables (c, x) appelés aussi arc.

Tous les arcs (c, x) présents dans Q sont soumis à l'algorithme *réviser* : les valeurs appartenant à $dom(x)$ n'ayant pas de support pour c sont supprimées de $dom(x)$. Si la révision est effective (au moins une valeur supprimée), alors l'algorithme *réviser* retourne *vrai*. Si c'est le cas et que $dom(x)$ est vide alors une incohérence globale est détectée. Dans le cas contraire, Q est mis à jour. Toutes les contraintes impliquant la variable x dans leur portée doivent être révisées : les arcs correspondants sont ainsi ajoutés dans Q.

La technique de révision utilisée par AC3 [MACKWORTH 1977] (voir Algorithme 2.2) propose la technique de révision *réviser-3* la plus évidente. Considérant un arc (c, x), on vérifie que chaque valeur appartenant à $dom(x)$ possède un support dans la contrainte c. Dans le cas contraire, la valeur est sup-

Algorithme 2.1 : $doAC$

Données : Réseau de contraintes $\mathcal{P} = (X, D, C)$
Résultat : Booléen

```
1  Q = {(c, x) | c ∈ C ∧ x ∈ scp(c)}
2  tant que Q ≠ ∅ faire
3      Choisir et éliminer (c, x) de Q
4      si réviser(c,x) alors
5          si dom(x) = ∅ alors
6              retourner Faux
7          fin
8          Q ← Q ∪ {(c', y) | c' ∈ C ∧ x ∈ scp(c') ∧ y ∈ scp(c') ∧ y ≠ x ∧ c' ≠ c}
9      fin
10 fin
11 retourner Vrai
```

Algorithme 2.2 : réviser-3

Données : Variable x, Contrainte c avec $scp(c) = \{x, y\}$
Résultat : Booléen

```
1  suppression ← faux
2  pour chaque a ∈ dom(x) faire
3      si ∄b ∈ dom(y) | (a, b) ∈ rel(c) alors
4          dom(x) ← dom(x) \ {a}
5          suppression ← vrai
6      fin
7  fin
8  retourner suppression
```

primée de $dom(x)$. Si au moins une valeur appartenant à $dom(x)$ est supprimée, l'algorithme retourne *vrai*, indiquant que la révision est effective, *faux* sinon.

Dans le cadre d'un réseau binaire, la complexité de l'algorithme de cohérence d'arc AC3 est de $\mathcal{O}(m \times d^3)$, où e correspond à $|cons(\mathcal{P})|$ et d à la taille du plus grand domaine.

AC3 n'est pas optimal en temps. Une explication de sa non-optimalité est la recherche systématique de supports pour toutes les valeurs appartenant à $dom(x)$ dans $scp(c)$ pour un arc (c, x), et cela en recommençant au début du domaine à chaque reprise. Comme vu précédemment, une contrainte peut être révisée plusieurs fois pour établir l'arc-cohérence. Lors de chaque nouvelle révision, il suffit de repartir du dernier support trouvé. C'est à partir de ce constat qu'a été proposé l'algorithme AC2001/3.1 [BESSIERE & DEBRUYNE 2005] dont la technique de révision *réviser-2001* est détaillée dans l'Algorithme 2.3.

Algorithme 2.3 : réviser-2001

Données : Variable x, Contrainte c avec $scp(c) = \{x, y\}$
Résultat : Booléen

```
1  suppression ← faux
2  pour chaque a ∈ dom(x) faire
3      si last[c, x, a] ∉ dom(y) alors
4          si ∄b ∈ dom(y) | b > last[c, x, a] ∧ (a, b) ∈ rel(c) alors
5              dom(x) ← dom(x) \ {a}
6              suppression ← vrai
7          fin
8          sinon
9              last[c, x, a] ← b
10         fin
11     fin
12 fin
13 retourner suppression
```

Soit l'arc (c, x) avec $scp(c) = \{x, y\}$. L'idée est de chercher pour chaque valeur (x, a) un support dans $dom(y)$ pour c lors de la première révision de (c, x), puis de le stocker dans une structure $last[c, x, a]$: le support de (x, a) dans le domaine de y pour c. Lors de la révision suivante, avant de chercher un support pour la valeur (x, a), on teste si le dernier support trouvé est encore valide, c'est à dire si $last[c, x, a] \in dom(y)$. Si ce support est toujours valide, on passe à la valeur suivante de x, sinon on cherche à partir de $last[c, x, a]$ (aussi appelé point de reprise) dans $dom(y)$ un nouveau support pour (x, a) (de par la propriété d'uni-directionnalité [BESSIÈRE & FREUDER 1999] selon laquelle la recherche de supports est effectuée dans une seule direction). Tout nouveau support est stocké. S'il existe au moins une valeur appartenant à $dom(x)$ pour laquelle aucun support n'a été trouvé, l'algorithme retourne *vrai* indiquant que la révision est effective.

Dans le cadre d'un réseau binaire, l'algorithme de cohérence d'arc AC2001/3.1 a une complexité temporelle de $\mathcal{O}(m \times d^2)$ et une complexité spatiale de $\mathcal{O}(m \times d)$. Il est optimal en temps contrairement à AC3 mais nécessite une structure supplémentaire *last*.

Pour améliorer AC2001/3.1 en pratique, l'algorithme $AC3^{rm}$ [LECOUTRE & HEMERY 2007] (rm pour résidus multi-directionnels) exploite la bi-directionnalité des contraintes (multi-directionnalité dans le cas non binaire) déjà introduite dans $AC3.2$ [LECOUTRE *et al.* 2003]. La procédure de révision *réviser-*3^{rm} est présentée dans l'Algorithme 2.4.

Ici, si une valeur $b \in dom(y)$ est un support de la valeur $a \in dom(x)$ pour une contrainte c de

Algorithme 2.4 : réviser-2001

Données : Variable x, Contrainte c avec $scp(c) = \{x, y\}$
Résultat : Booléen

```
1  suppression ← faux
2  pour chaque a ∈ dom(x) faire
3      si supp[c, x, a] ∉ dom(y) alors
4          si ∄b ∈ dom(y) | (a, b) ∈ rel(c) alors
5              dom(x) ← dom(x) \ {a}
6              suppression ← vrai
7          fin
8          sinon
9              supp[c, x, a] ← b
10             supp[c, y, b] ← a
11         fin
12     fin
13 fin
14 retourner suppression
```

portée $scp(c) = \{x, y\}$, alors le couple variable-valeur (x, a) est réciproquement un support du couple (y, b) pour c. Les supports stockés sont appelés supports résiduels ou plus simplement résidus. L'idée est que lorsqu'un support $\tau = (a, b)$ est trouvé pour un couple variable-valeur (x, a) dans la contrainte c, il va être également enregistré pour les autres variables-valeurs appartenant à τ : (y, b) va ainsi être enregistré comme résidu de (x, a) et réciproquement (x, a) résidu de (y, b). Comme dans AC2001/3.1, une structure additionnelle est nécessaire : *supp*. Elle permet de stocker pour chaque valeur le dernier résidu bi-directionnel trouvé. Lors de chaque révision d'arc, comme dans AC2001/3.1, la validité du résidu est testée, et dans le cas où il n'est plus valide, un nouveau tuple support doit être trouvé. Dans le cadre de la bi-directionnalité des contraintes, la propriété d'uni-directionnalité ne peut pas être exploitée. C'est à dire que lorsqu'un support est devenu invalide, nous n'avons pas de point de reprise et on doit rechercher un nouveau support dans l'ensemble du domaine. En effet, avec la bi-directionnalité nous n'avons aucune certitude que le dernier support trouvé corresponde au dernier plus petit support trouvé.

Dans le cadre d'un réseau binaire, l'algorithme de cohérence d'arc $AC3^{rm}$ a une complexité en temps $\mathcal{O}(m \times d^3)$ et une complexité spatiale de $\mathcal{O}(m \times d)$.

Nous avons donc présenté les trois algorithmes à gros grain les plus utilisés : AC3 non optimal, puis AC2001/3.1 optimal en temps mais demandant une complexité spatiale plus importante et enfin $AC3^{rm}$. L'algorithme $AC3^{rm}$ n'est pas optimal en théorie. Cependant il est mieux adapté et donc plus efficace en pratique, notamment pour des réseaux comportants des contraintes dont la dureté est faible ou forte (c'est à dire acceptant beaucoup ou peu de supports) [LECOUTRE & HEMERY 2007]. Il est tout à fait compétitif vis à vis de AC2001/3.1.

Les algorithmes établissant l'arc-cohérence (et la cohérence en général) peuvent être utilisés en prétraitement et/ou en cours de la recherche. Dans ce cadre, maintenir la cohérence d'arc à l'aide de l'algorithme $AC3^{rm}$ à chaque étape d'une recherche s'avère notamment être plus efficace que de maintenir la cohérence d'arc AC2001/3.1.

2.2.3 Arc-cohérence généralisée

La notion d'arc-cohérence s'applique également aux réseaux non binaires : on parle d'arc-cohérence généralisée (notée GAC pour *Generalized Arc Consistency*) ou d'hyper arc-cohérence [KRZYSZTOF 2003].

Définition 2.22 (Valeur arc-cohérent généralisée) *Soit c une contrainte d'arité $r > 2$ et une variable $x \in scp(c)$. Une valeur $a \in dom(x)$ est arc-cohérente généralisée dans c si et seulement si il existe un tuple $\tau \in rel(c)$ tel que τ soit valide et $\tau[x] = a$.*

Définition 2.23 (Contrainte arc-cohérent généralisée) *Une contrainte c est arc-cohérente généralisée si et seulement si toute valeur $a \in dom(x)$ est arc-cohérente généralisée, avec $x \in scp(c)$.*

Définition 2.24 (Réseau arc-cohérent généralisé) *Un réseau est arc-cohérent généralisé si et seulement si toutes ses contraintes sont arc-cohérentes généralisées.*

Dans la suite de ce manuscrit, nous représentons les contraintes n-aires de grande arité en extension par une table positive (resp. negative) contenant la liste des tuples autorisés (resp. interdit) pour la contrainte.

Pour établir GAC via ces algorithmes sur des contraintes tables, il existe plusieurs techniques de recherche de supports pour une valeur. Deux approches sont distinguées : les techniques classiques traitant la liste originale des tuples autorisés valides et invalides pour une contrainte, puis des techniques optimisées traitant uniquement la liste des tuples autorisés valides via l'utilisation de structures additionnelles. À noter que ces différentes techniques peuvent être utilisées en pré-traitement ou/et au cours de la recherche.

Algorithmes classiques

Une première approche appelée généralement $GAC - valid$ consiste à itérer les tuples valides pour une contrainte jusqu'à ce qu'un tuple soit autorisé. De manière réciproque, une deuxième approche appelée généralement $GAC - allowed$ consiste à parcourir la liste des tuples autorisés jusqu'à ce qu'un tuple soit valide. Malheureusement, parcourir la liste des tuples valides ou autorisés peut s'avérer coûteux et très pénalisant, le nombre de tuples pouvant augmenter de façon exponentielle avec l'arité de la contrainte. Pour pallier cet inconvénient, une troisième approche proposée consiste à alterner parcours des tuples valides et parcours des tuples autorisés (méthode appelée naturellement $GAC - valid + allowed$). Le principe est de chercher le premier tuple valide contenant cette valeur et de chercher parmi la liste des tuples autorisés supérieurs ou égaux à ce tuple un tuple autorisé. Si un tel tuple existe, alors un support a été trouvé. L'intérêt de cette approche est clairement de réaliser des sauts dans les listes de tuples et ainsi éviter de considérer bon nombre de tuples [LECOUTRE & SZYMANEK 2006].

Les algorithmes $AC3$, $AC2001/3.1$ et $AC3^{rm}$ abordés dans la section précédente ont été présentés dans le cadre de contraintes binaires. Ils sont généralisables pour des contraintes n-aires et sont appelés $GAC3$, $GAC2001/3.1$ et $GAC3^{rm}$. Concernant les techniques $GAC - valid$, $GAC - allowed$ et $GAC - valid + allowed$, elles peuvent être exploitées au niveau des algorithmes de révision 2.2, 2.3 et 2.4.

Algorithmes optimisés

L'un des algorithmes de filtrage les plus efficaces pour établir l'arc-cohérence généralisée sur des contraintes positives (tuples autorisés) non binaires est appelé technique de réduction tabulaire simple (`STR` pour *Simple Tabular Reduction* [ULLMANN 2007]. À la différence des approches classiques ,

le principe de STR est de maintenir dynamiquement la liste des tuples autorisés et valides pour une contrainte et permettre ainsi de trouver plus rapidement un support pour une valeur.

L'une des particularités de STR est qu'il permet, au delà de maintenir les tuples autorisés valides au cours de la recherche, de récupérer efficacement la liste des tuples autorisés valides lors de l'utilisation de la technique du *backtracking* (couramment utilisée pour la résolution des CSP et abordée dans la section suivante). Pour cette raison, nous présentons l'algorithme STR maintenant GAC au cours d'une stratégie de recherche.

Soit une contrainte table (en extension) c définie par la table *table[c]*, STR utilise les quatres structures additionnelles suivantes :

- Un tableau *position[c]* de taille t qui fournit un accès indirect aux tuples appartenant à *table[c]*. Les valeurs dans *position[c]* représentent une permutation de $\{0, 1, \ldots, t-1\}$. Le i^{eme} tuple valide (et autorisé) dans c est accessible par *table[c][position[c[i]]]* ;
- Une valeur *currentLimit[c]* qui correspond à la position du dernier tuple valide dans *table[c]*. La table courante de c est composée d'exactement *currentLimit[c]* + 1 tuples. Les valeurs dans *position[c]* aux indices de 0 à *currentLimit[c]* sont les positions des tuples courants de c ;
- Un tableau *LevelLimits[c]* de taille n + 1 (n correspondant aux nombres de variables dans le réseau) où *LevelLimits[c][l]* correspond à la position du premier tuple invalide dans *table[c]* supprimé quand la recherche était au level l, correspondant au nombre de variables assignées dans l'assignation courante partielle de la recherche. Autrement dit, *levelLimits[c][l]* est une sauvegarde au level l, sinon *levelLimits[c][l]* = -1. Exploité pour la technique de *backtracking*, *levelLimits[c]* est indicé de 0 à n, le level 0 correspondant aux tuples supprimés lors d'une éventuelle phase de pré-traitement. Les tuples supprimés au level l sont via les valeurs dans *position[c]* aux indices allant de *currentLimit[c]* + 1 à *levelLimits[c][l]* ;
- Un tableau *valeursGAC* de taille égale à l'arité de c et contenant pour chaque variable appartenant à $scp(c)$ les valeurs prouvées comme étant GAC-cohérentes.

Algorithme 2.5 : $doGAC^{str}$

Données : Réseau de contraintes $\mathcal{P} = (X, D, C)$, Variable x
Résultat : Booléen

```
1  Q ← {x}
2  tant que Q ≠ ∅ faire
3      Choisir et éliminer x de Q
4      pour chaque c ∈ C | x ∈ scp(c) faire
5          X_reduits ← STR(P, c)
6          pour chaque x ∈ X_reduits faire
7              si dom(x) = ∅ alors
8                  retourner Faux
9              fin
10             Q ← Q ∪ {x}
11         fin
12     fin
13 fin
14 retourner Vrai
```

Pour résumer, le principe de STR est de découper une table en deux sous-ensembles de sorte qu'après chaque étape de la recherche, chaque tuple appartient à un et un seul de ces sous-ensembles. L'un de ces sous-ensembles correspond à la table courante et regroupe tous les tuples courants, c'est à dire les

tuples autorisés et valides pour une contrainte (autrement dit les supports). Commençons par présenter l'algorithme $doGAC^{str}$ (Algorithme 2.5).

Contrairement aux approches présentées précédemment, l'algorithme $doGAC^{str}$ maintient globalement GAC sur chacune des contraintes du réseau et n'effectue pas de révisions successives d'arcs contrainte-variable. Il est appliqué au réseau dès qu'une décision (positive ou négative voir Définition 2.25) a été prise sur une variable x passée en paramètre.

Une queue de propagation Q contient initialement la variable x et contiendra les variables dont le domaine a été modifié pendant l'établissement de GAC. Tant que la queue de propagation n'est pas vide, une variable en est extraite et STR est appliqué (algorithme 2.6) à chaque contrainte contenant cette variable dans sa portée. L'ensemble $X_{reduits}$ de variables dont le domaine a été réduit durant l'application de STR sur une contrainte est retourné : si au moins l'une de ces variables a un domaine vide alors le réseau est prouvé non GAC-cohérent (faux est retourné), sinon chacune de ces variables est ajoutée dans la queue de propagation. Si aucun domaine n'est vide après l'établissement de GAC, alors le réseau est GAC-cohérent (vrai est retourné).

Après avoir initialisé la structure *valeursGAC* à l'ensemble vide ($\emptyset$) pour toutes les variables non assignées de la portée, l'algorithme STR parcourt les tuples aux indices allant de 0 à $currentLimit[c]$ dans $position[c]$:

- Si le tuple est valide (vérifié par l'Algorithme 2.7), alors pour chaque variable x on ajoute dans *valeursGAC[x]* les valeurs respectives dans le tuple pour lesquelles un support vient d'être trouvé et qui sont donc GAC-cohérentes ;
- Si par contre le tuple est invalide, alors il est supprimé de la table courante via un appel à l'algorithme 2.8 avec comme paramètre la position du tuple et le level courant, c'est à dire le nombre de variables déjà assignées. Si c'est le premier tuple supprimé au cours de ce niveau, alors *levelLimits[c]* est mis à jour avec *currentLimit[c]*. Un tuple est supprimé en inversant sa position dans *position[c]* avec le dernier tuple courant pointé par *currentLimit[c]* et en décrémentant ensuite la valeur de *currentLimit[c]*.

Une fois les tuples courants parcourus, on vérifie pour chaque variable si toutes les valeurs de son domaine sont GAC-cohérentes (autant de valeurs dans le domaine que dans *valeursGAC*). Si au moins une valeur n'est pas GAC-cohérente, le domaine de la variable est mis à jour pour contenir uniquement les valeurs GAC-cohérentes. De plus, si le domaine est vide, le réseau n'est pas GAC-cohérent et c'est un échec (FAILURE retourné), sinon la variable est ajoutée dans l'ensemble $X_{reduits}$ contenant les variables pour lesquelles le domaine a été réduit.

L'algorithme 2.9 est appelé lors d'un retour en arrière pour restaurer les tuples supprimés dans un niveau précédent. Cette restauration de tuples se fait en manipulant la structure *levelLimits[c]* à ce niveau. Si des tuples ont été supprimés lors de ce niveau, alors on met à jour la limite des tuples courants *currentLimit[c]* afin qu'elle pointe vers le premier tuple supprimé lors de ce niveau qui est référencé par *levelLimits[c]*, puis on réinitialise *levelLimits[c]* à -1.

Notons que les décisions négatives ne sont pas considérées pour le retour en arrière. En effet, nous considérons la stratégie de recherche qui consiste pour chaque variable à d'abord affecter une valeur, puis ensuite la réfuter (voir les schémas de branchement dans la section suivante). Une fois ces deux décisions prises, on a tout exploré pour cette variable. Si un échec apparaît, le retour en arrière doit être alors effectué sur une assignation de variable antérieure (décision positive). C'est notamment la raison pour laquelle la taille de *levelLimits[c]* est égale au nombre de variables (pour une variable, une affectation de valeur à la fois).

La complexité en temps de STR pour une contrainte c est de $\mathcal{O}(r'd + rt')$ avec r' le nombre de variables non assignées dans $scp(c)$ et t' la taille de la table courante de c (nombre de tuples). On y retrouve les complexités respectives $\mathcal{O}(r')$, $\mathcal{O}(rt')$ et $\mathcal{O}(r'd)$ des trois boucles qui constituent STR. La complexité spatiale pour cette même contrainte c est de $\mathcal{O}(n + rt)$ avec n pour la taille de la structure

Algorithme 2.6 : STR

Données : Réseau de contraintes $\mathcal{P} = (X, D, C)$, Contrainte c
Résultat : Ensemble de variables

pour chaque $x \in scp(c) \mid x \notin assigned(\mathcal{P})$ **faire**
  $valeursGAC[x] \leftarrow \emptyset$
fin
$i \leftarrow 0$
tant que $i \leq currentLimit[c]$ **faire**
  $index \leftarrow position[c][i]$
  $\tau \leftarrow table[c][index]$
  si $estUnTupleValide(c, \tau)$ **alors**
    pour chaque $x \in scp(c) \mid x \notin assigned(\mathcal{P})$ **faire**
      si $\tau[x] \notin valeursGAC[x]$ **alors**
        $valeursGAC[x] \leftarrow valeursGAC[x] \cup \{\tau[x]\}$
      fin
      $i \leftarrow i + 1$
    fin
  fin
  sinon
    $supprimerTuple(c, i, |assigned(\mathcal{P})|)$
  fin
fin
$X_{reduits} \leftarrow \emptyset$
pour chaque $x \in scp(c) \mid x \notin assigned(\mathcal{P})$ **faire**
  si $|valeursGAC[x]| < |dom(x)|$ **alors**
    $dom(x) \leftarrow valeursGAC[x]$
    si $dom(x) = \emptyset$ **alors**
      retourner FAILURE
    fin
    $X_{reduits} \leftarrow X_{reduits} \cup \{x\}$
  fin
fin
retourner $X_{reduits}$

Algorithme 2.7 : $estUnTupleValide$

Données : Contrainte c, Tuple τ
Résultat : Booléen

pour chaque $x \in scp(c)$ **faire**
  si $\tau[x] \notin dom(x)$ **alors**
    retourner *Faux*
  fin
fin
retourner *Vrai*

Algorithme 2.8 : $supprimerTuple$

Données : Contrainte c, Entier i, Entier p

si $levelLimits[c][p] = -1$ **alors**
  $levelLimits[c][p] \leftarrow currentLimit[c]$
fin
$echanger(position[c][i], position[c][currentLimit[c]])$
$currentLimit[c] \leftarrow currentLimit[c] - 1$

Algorithme 2.9 : $restaurerTuples$

Données : Contrainte c, Entier p

si $levelLimits[c][p] \neq -1$ **alors**
  $currentLimit[c] \leftarrow levelLimits[c][p]$
  $levelLimits[c][p] \leftarrow -1$
fin

levelLimits[c], rt pour *table[c]*, r pour *valeursGAC* et t pour *position[c]*.

Afin de bien comprendre comment fonctionne l'algorithme `STR`, nous allons considérer une contrainte d'arité 3 et l'évolution de ces différentes structures lorsque `STR` est appliqué à cette contrainte pendant la recherche.

Exemple 2.12 *Soit c une contrainte table positive d'arité 3 avec $scp(c) = \{x, y, z\}$ et $dom(x) = dom(y) = dom(z) = \{i, j, k\}$. Elle est définie par $rel(c) = \{(i, j, i), (i, k, i), (i, k, j), (i, k, k), (j, k, i), (j, k, k), (k, i, j)\}$. La figure 2.11 illustre la contrainte table c ainsi que les structures STR associées à l'état initial.*

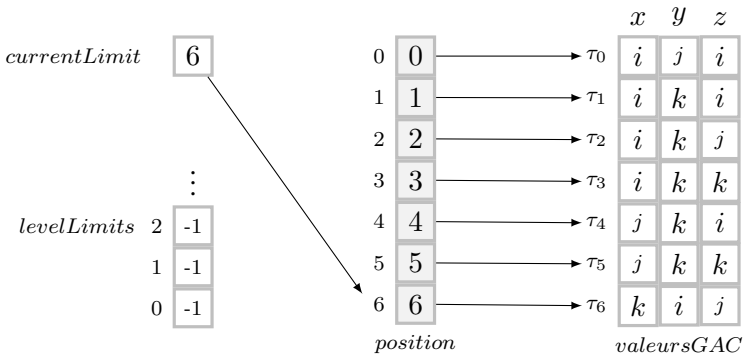


FIGURE 2.11 – Les structures STR initialisées pour la contrainte c

Avant le début de la recherche, tous les tuples autorisés dans c sont valides : la table courante, nommée position, regroupe donc tous les tuples appartenant à table[c]. De ce fait, la position du dernier tuple courant stockée dans currentLimit[c] correspond à la position du dernier tuple de table[c]. Le tuple référencé par position[c][0] correspond au premier tuple valide dans c, c'est à dire $\tau_0 : (i, j, i)$. La recherche n'ayant pas encore commencé, aucun tuple n'a été supprimé et donc le tableau levelLimits est initialisé à -1. Considérons maintenant que la phase de recherche débute par la décision $y = k$: nous sommes au niveau 1, c'est à dire lors de l'assignation de la première variable.

`STR` *doit être appliqué à c. Le tableau position[c] est parcouru et les tuples aux différentes positions*

sont analysés. Le premier tuple considéré τ_0 à la position 0 n'est plus valide : il est alors échangé avec le dernier tuple courant, en l'occurence le tuple τ_6 à la position currentLimit[c] = 6 et levelLimits[c][1] est mis à jour (Figure 2.12). De plus, le pointeur currentLimit[c] est décrémenté (Figure 2.13). Ensuite, c'est le tuple τ_6 qui est considéré car il est maintenant en tête de position[c]. Le tuple τ_6 n'est pas valide. Il est échangé lui aussi avec le dernier tuple valide. Le résultat de STR *appliqué à la contrainte c après la décision $y = k$ est illustré dans la Figure 2.14. Notons que les tuples supprimés au cours du niveau 1 correspondent aux tuples positions allant de currentLimit[c] +1 à levelLimits[c][1], à savoir les tuples des positions* 5 *et* 6 *(en pointillés sur la figure).*

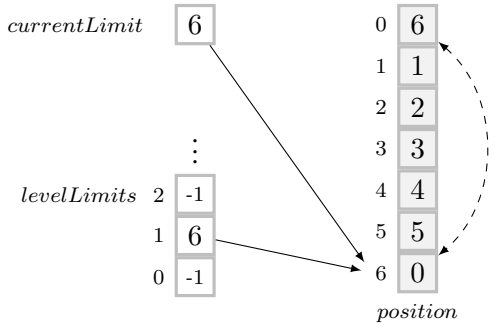


FIGURE 2.12 – Inversion dans *position* des tuples aux positions 0 et 6, puis mis à jour de *levelLimits* à 6 pour le niveau 1

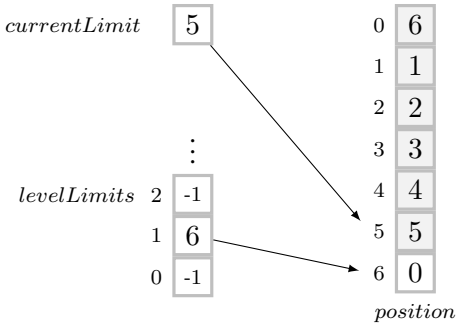


FIGURE 2.13 – *currentLimit* est décrémenté. La table courante ne contient plus que 6 tuples.

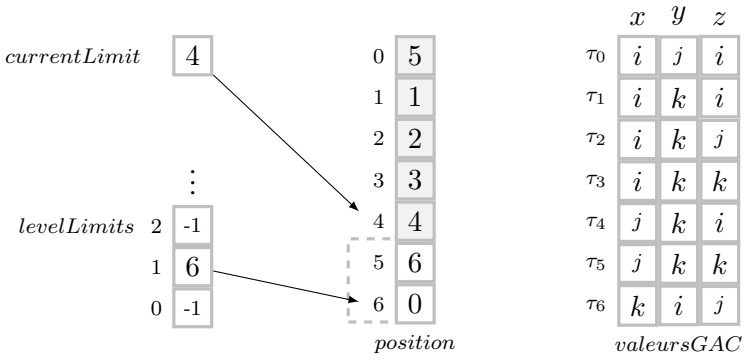


FIGURE 2.14 – STR appliqué à la contrainte c après la décision $y = c$

Après une seconde assignation de variable (niveau 2), imaginons que l'information $x \neq i$ soit propagée à c depuis une autre depuis une autre contrainte du réseau. Selon le même principe qu'au niveau 1, les tuples non valides sont échangés avec les tuples valides et la valeur currentLimit[c] est décrémentée. De plus, la valeur (z, j) n'a plus de support : en effet son seul support $\tau_2 = (i, k, j)$ a été supprimé au cours de STR. Le résultat de cette deuxième exécution est illustré dans la Figure 2.15. Imaginons maintenant qu'un retour en arrière d'un niveau soit réalisé à partir du niveau 2. La structure levelLimits[c][2] permet de restaurer en temps constant les tuples qui avaient été supprimés lors de ce niveau : en effet, si on revient en arrière sur la décision ayant amenée la propagation de l'information $x \neq i$, ces tuples redeviennent valides : on affecte alors à currentLimit[c] la position stockée dans levelLimits[c][2] pour englober à nouveau ces tuples dans la table courante et la valeur levelLimits[c][2] est remise à

-1. L'état des structures de STR après un retour en arrière depuis le niveau 2 est illustré en Figure 2.16. On peut remarquer à la fin que les tuples valides ne sont pas ordonnés de la même façon qu'ils ne l'étaient lors de l'initialisation. Ce n'est pas important dans `STR`*, ce qui est primordial, c'est réellement le partitionnement entre tuples valides et tuples non valides.*

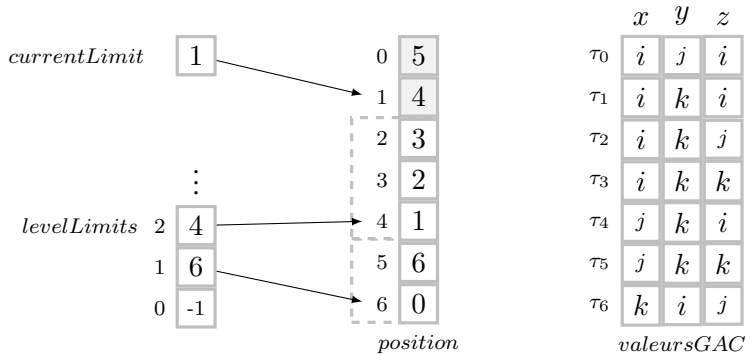


FIGURE 2.15 – `STR` appliqué à la contrainte c après la propagation de $x \neq i$ depuis une autre contrainte. La valeur (z, j) n'ayant plus de support, elle est supprimée de $dom(z)$

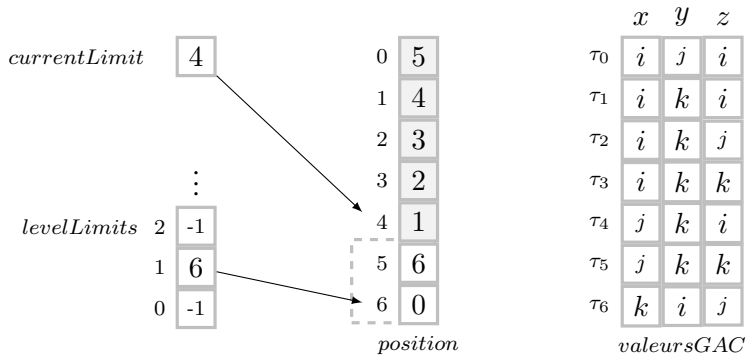


FIGURE 2.16 – État des structures de `STR` pour la contrainte c après la restauration conséquente au retour en arrière au niveau 1

Une version optimisée de `STR` appelée `STR2` ([LECOUTRE 2009] et [LECOUTRE 2011]) a été proposée. Deux remarques pouvant être faites sur `STR` sont à l'origine de cette nouvelle version :

— Il est inutile de chercher des supports pour les valeurs du domaine d'une variable si toutes les valeurs du domaine ont déjà étés prouvées `GAC`-cohérentes ;

— Après l'exécution de `STR` sur une contrainte, tous les tuples restants sont des tuples valides. Cela veut dire que tant que le domaine d'une variable n'est pas réduit, les tuples restent valides par rapport à cette variable et donc les tests de validité des valeurs pour cette variable sont inutiles.

Pour éviter ces recherches de supports et ces tests de validité inutiles, l'algorithme exploite trois structures additionnelles :

— Un ensemble S^{sup} qui contient les variables non assignées dont le domaine contient au moins une valeur pour laquelle aucun support n'a encore été trouvé ;

— Un ensemble S^{val} qui contient les variables non assignées dont le domaine a été réduit depuis le dernier appel de $STR2$;

— Un tableau *lastSize[c][x]* qui contient pour chaque variable x la taille de son domaine après chaque appel à $STR2$.

L'exploitation de ces structures pour optimiser STR (Algorithme 2.6) peut être observée dans STR2 (Algorithme 2.10). La structure S^{val} est initialisée avec la dernière variable assignée (*lastAssigned(P)* si elle appartient à la portée de la contrainte. En effet, cette variable assignée doit être contenue dans S^{val} car son domaine a été réduit. La structure S^{sup} est quant à elle initialisée avec toutes les variables non assignées de la portée de la contrainte. Le parcours des tuples courant est identique à STR, cependant le test de validité d'un tuple diffère (Algorithme 2.11) : Il considère uniquement les variables présentes dans S^{val}. Dès qu'il est vérifié que toutes les valeurs du domaine d'une variable sont GAC-cohérentes, alors cette variable est supprimée de S^{sup}. Pour finir, on ne s'intéresse qu'à la recherche d'un support pour les variables appartenant à S^{sup}.

La complexité en temps de STR2 pour une contrainte c correspond à $\mathcal{O}(r' \times d + r'' \times t')$ avec r'' le nombre de variables non assignées dans $scp(c)$ pour lesquelles les tests de validité sont nécessaires (taille de l'ensemble S^{val}) et t' la taille de la table courante de c. On y retrouve les complexités respectives $\mathcal{O}(r')$, $\mathcal{O}(r'' \times t')$ (le test de validité d'un tuple dans $STR2$ correspond à $\mathcal{O}(r'')$ au lieu de $\mathcal{O}(r)$ pour STR) et $\mathcal{O}(r' \times d)$ des trois boucles qui constituent $STR2$. Tout comme pour STR, la complexité spatiale pour une même contrainte c dans STR2 est de $\mathcal{O}(n + r \times t)$: la complexité $\mathcal{O}(n + r \times t)$ héritée des structures de STR, associée aux complexités spatiales de $\mathcal{O}(r)$ pour les structures *lastSize*, S^{sup}, et S^{val} (S^{sup} et S^{val} devant être partagées par toutes les contraintes).

2.2.4 Stratégies de recherche

À ce niveau du manuscrit, nous avons présenté les réseaux de contraintes et différentes techniques d'inférence qui permettent de simplifier un réseau par un processus de déduction : la propagation de contraintes.

Il arrive quelques fois que l'inférence appliquée en pré-traitement suffise pour prouver la satisfaisabilité ou l'insatisfaisabilité d'un problème (apparition d'un domaine vide, etc.), mais cela reste relativement rare. Généralement, une phase de recherche est nécessaire pour résoudre un réseau de contraintes. On distingue deux types de recherche : recherche complète et recherche incomplète. Toutefois, nous ne détaillons pas les méthodes de recherche incomplète qui sont hors du contexte de nos contributions.

Méthodes de recherche complète

Dans le cadre des CSP, une recherche complète de type arborescente est guidée généralement par un schéma de branchement consistant à traverser l'espace de recherche, c'est à dire le produit cartésien des domaines, en assignant les variables les unes après les autres jusqu'à l'obtention d'une solution où jusqu'à ce que l'espace de recherche soit totalement parcouru. Nous commencerons dans un premier temps par présenter une approche basique et naïve de recherche complète non optimisée appelée *Générer et Tester*. Ensuite, nous présenterons le modèle BPRA (pour *Branchement Propagation Retour-arrière Apprentissage*) sur lequel sont basés les algorithmes de recherche complète les plus efficaces. Par une utilisation combinée de ses quatre composants, ce modèle propose de mettre en œuvre des techniques pour parcourir efficacement l'espace de recherche (Branchement) tout en le filtrant durant l'exploration (Propagation) et en effectuant des retours en arrière dès l'apparition d'échecs (Retour-arrière). De plus, bon nombre d'informations pourront être enregistrées (Apprentissage) durant la recherche et exploitées par les autres composants de ce modèle.

Algorithme 2.10 : $STR2$

Données : Réseau de contraintes $\mathcal{P} = (X, D, C)$, Contrainte c
Résultat : Ensemble de variables

$S^{sup} \leftarrow \emptyset$
$S^{val} \leftarrow \emptyset$
si $lastAssigned(\mathcal{P}) \in scp(c)$ **alors**
  $S^{val} \leftarrow S^{val} \cup \{lastAssigned(\mathcal{P})\}$
fin
pour chaque $x \in scp(c) \mid x \notin assigned(\mathcal{P})$ **faire**
  $valeursGAC[x] \leftarrow \emptyset$
  $S^{sup} \leftarrow S^{sup} \cup \{x\}$
  si $|dom(x)| \neq lastSize[c][c]$ **alors**
    $S^{val} \leftarrow S^{val} \cup \{x\}$
    $lastSize[c][x] \leftarrow |dom(x)|$
  fin
fin
$i \leftarrow 0$
tant que $i \leq currentLimit[c]$ **faire**
  $index \leftarrow position[c][i]$
  $\tau \leftarrow table[c][index]$
  si $estUnTupleValide(c, \tau, S^{val})$ **alors**
    pour chaque $x \in S^{sup}$ **faire**
      si $\tau[x] \notin valeursGAC[x]$ **alors**
        $valeursGAC[x] \leftarrow valeursGAC[x] \cup \{\tau[x]\}$
        si $|valeursGAC[x]| = |dom(x)|$ **alors**
          $S^{sup} \leftarrow S^{sup} \setminus \{x\}$
        fin
      fin
      $i \leftarrow i + 1$
    fin
  fin
  sinon
    $supprimerTuple(c, i, |assigned(\mathcal{P})|)$
  fin
fin
$X_{reduits} \leftarrow \emptyset$
pour chaque $x \in S^{sup}$ **faire**
  $dom(x) \leftarrow valeursGAC[x]$
  si $dom(x) = \emptyset$ **alors**
    retourner FAILURE
  fin
  $X_{reduits} \leftarrow X_{reduits} \cup \{x\}$
  $lastSize[c][x] \leftarrow |dom(x)|$
fin
retourner $X_{reduits}$

Algorithme 2.11 : $estUnTupleValide$

Données : Contrainte c, Tuple τ, Ensemble de variables S^{val}
Résultat : Booléen

```
1 pour chaque x ∈ S^val faire
2     si τ[x] ∉ dom(x) alors
3         retourner Faux
4     fin
5 fin
6 retourner Vrai
```

Méthode Générer et Tester

Une première méthode (assez naïve) pour la résolution d'une instance CSP est de générer toutes les instanciations complètes possibles et de tester si elles sont solutions du problème, c'est à dire si elles vérifient toutes les contraintes du réseau. Cette approche, aussi connue sous le nom de *Generate-and-test* évalue toutes les possibilités les unes après les autres et n'est donc pas envisageable en pratique. En effet, afin d'obtenir l'ensemble des solutions d'un problème simple tel que les 8-reines, cet algorithme génère et vérifie la validité de $8^8 = 16777216$ instanciations différentes. Néanmoins, ce nombre peut largement être réduit par une analyse de la structure du problème. Ces méthodes sont regroupées dans le modèle BPRA (pour *Branchement Propagation Retour-arrière Apprentissage*).

Le modèle de recherche BPRA

Les algorithmes de résolution complète de CSP les plus efficaces sont basés sur le modèle BPRA [JUSSIEN *et al.* 2002] et [LECOUTRE 2009]. L'efficacité obtenue par la mise en place de ce modèle réside généralement dans une utilisation combinée des différentes techniques proposées par ses composants :

- Branchement : le parcours de l'espace de recherche (branchement binaire, non-binaire, profondeur d'abord, largeur d'abord, etc.) et quelles décisions prendre à chaque étape (heuristiques statiques et dynamiques de choix de variables, valeurs, etc.) ;
- Propagation : mécanismes d'inférence mis en œuvre, les cohérences plus ou moins fortes établies pour filtrer le réseau après la prise de chaque décision et réduire l'espace de recherche ;
- Retour-arrière (aussi nommé *Backtracking* [DAVIS *et al.* 1962] et [GOLOMB & BAUMERT 1965]) : la manière, plus ou moins intelligente, de revenir en arrière lorsqu'un échec est atteint durant la recherche arborescente : dernière décision prise ou décision plus ancienne et réellement responsable de l'échec, etc ;
- Apprentissage : les informations enregistrées durant la propagation (valeurs supprimées) et lorsqu'un échec est rencontré (*nogood*), puis la manière de les exploiter durant le branchement, la propagation et le retour en arrière.

Par la suite, Les différentes techniques de Branchement, de Propagation et de *Backtracking* qui sont généralement utilisées sont détaillées. Toutefois, dans le cadre de cette thèse, les techniques d'Apprentissage utilisées au cours de la phase de résolution ne sont pas mis en avant car, même si il s'agit d'une partie importante dans la résolution des CSP, nos travaux ne sont pas concentrés sur celle-ci, bien que les techniques abordées basées sur les conflits pour la phase de *Backtracking* exploitent des informations apprises au cours du mécanisme de propagation, en l'occurence les explications des différentes suppressions de valeurs (conflits). Il est donc important de souligner que différentes techniques d'apprentissage existent afin d'améliorer la résolution des CSP. Le lecteur intéressé par ce sujet peut se référer notamment aux ouvrages de références suivants : mémorisation de nogoods ([DECHTER 1990a], [FROST &

DECHTER 1994] et [LECOUTRE *et al.* 2007]), gestion des explications de conflits ([PROSSER 1993] et [GINSBERG & MCALLESTER 1994]).

Schéma de branchement

Deux schémas principaux de branchement existent :

— Le branchement binaire (appelé aussi *2-way branching*) ;

— Le branchement non-binaire (appelé aussi *d-way branching*.

À chaque étape d'un branchement binaire, une variable non assignée ainsi qu'une valeur de son domaine sont choisies à chaque étape et l'affectation puis la réfutation de cette valeur pour cette variable sont considérées (*2-way branching* car exactement 2 branches possibles). Concernant le branchement non-binaire, une variable non-assignée est choisie et toutes les assignations de cette variable par les valeurs de son domaine sont considérées (*d-way branching* car d branches possibles pour une variable associée à un domaine de taille d).

À noter que ces deux schémas garantissent une exploration complète de l'espace de recherche. Bien que le branchement non-binaire puisse facilement être simulé par le branchement binaire, c'est deux schémas de branchement ne sont pas équivalents. En effet, [HWANG & MITCHELL 2005] ont montré qu'il existe des instances pour lesquelles utiliser un schéma de branchement non-binaire est exponentiellement moins efficace que d'utiliser un schéma de branchement binaire. C'est naturellement pour cela que seul le branchement binaire est présenté ici.

De plus, plusieurs techniques d'exploration de l'arbre de recherche existent : en profondeur d'abord (*depth-first search*), en largeur d'abord (*brearth-first search*). L'approche présentée par la suite est basée sur une exploration en profondeur d'abord qui représente, par rapport à une exploration en largeur d'abord, un meilleur compromis entre espace mémoire utilisé et efficacité pour trouver (rapidement) une solution.

L'arbre partiel de recherche présenté en figure 2.17 décrit le déroulement d'une recherche de solution avec schéma binaire pour le problème des 4-reines illustré en figure 2.3. La racine de l'arbre représente le réseau de contraintes $\mathcal{P} = (X, D, C)$ initial où aucune assignation n'a encore été réalisée. Chaque branche de l'arbre correspond à des décisions positives et négatives pour chaque $x \in X$.

Définition 2.25 (décisions positives et négatives) *Soient $\mathcal{P} = (X, D, C)$ un réseau de contraintes, $x \in X$ et $a \in dom(x)$. $(X = a)$ (resp. $(X \neq a)$ représente la décision positive (resp. négative). Nous avons alors $\neg(x = a) = (x \neq a)$ et $\neg(x \neq a) = (x = a)$*

Généralement, les branches gauches de l'arbre représentent des décisions positives tandis que les branches droites représentent des décisions négatives. Chaque nœud de l'arbre représente le réseau modifié (et simplifié) par la prise en compte d'une part de l'ensemble des décisions qui ont amené à ce nœud, puis par la propagation des contraintes d'autre part. Une solution du problème est obtenue lorsque toutes les variables sont singletons et que l'instanciation obtenue est globalement cohérente. Dans le cadre d'un branchement binaire, la réfutation d'une valeur pour une variable peut être suivie par l'affectation d'une valeur à une autre variable, ce qui donne de la flexibilité à l'exploration.

Le branchement binaire n'est pas suffisant en lui-même pour optimiser la résolution d'un `CSP`. Encore faut-il prendre les bonnes décisions : dans quel ordre assigner les variables et pour chacune de ces variables, dans quel ordre lui affecter les valeurs de son domaine ? Clairement, les bonnes décisions sont celles qui vont permettre soit de se diriger très rapidement vers une solution, soit de se diriger très rapidement vers un échec et ainsi éviter de parcourir inutilement des zones de l'espace de recherche. Il existe des heuristiques qui guident les choix de variables et de valeurs pour un parcours plus ou moins rapide de l'espace de recherche en fonction du problème considéré.

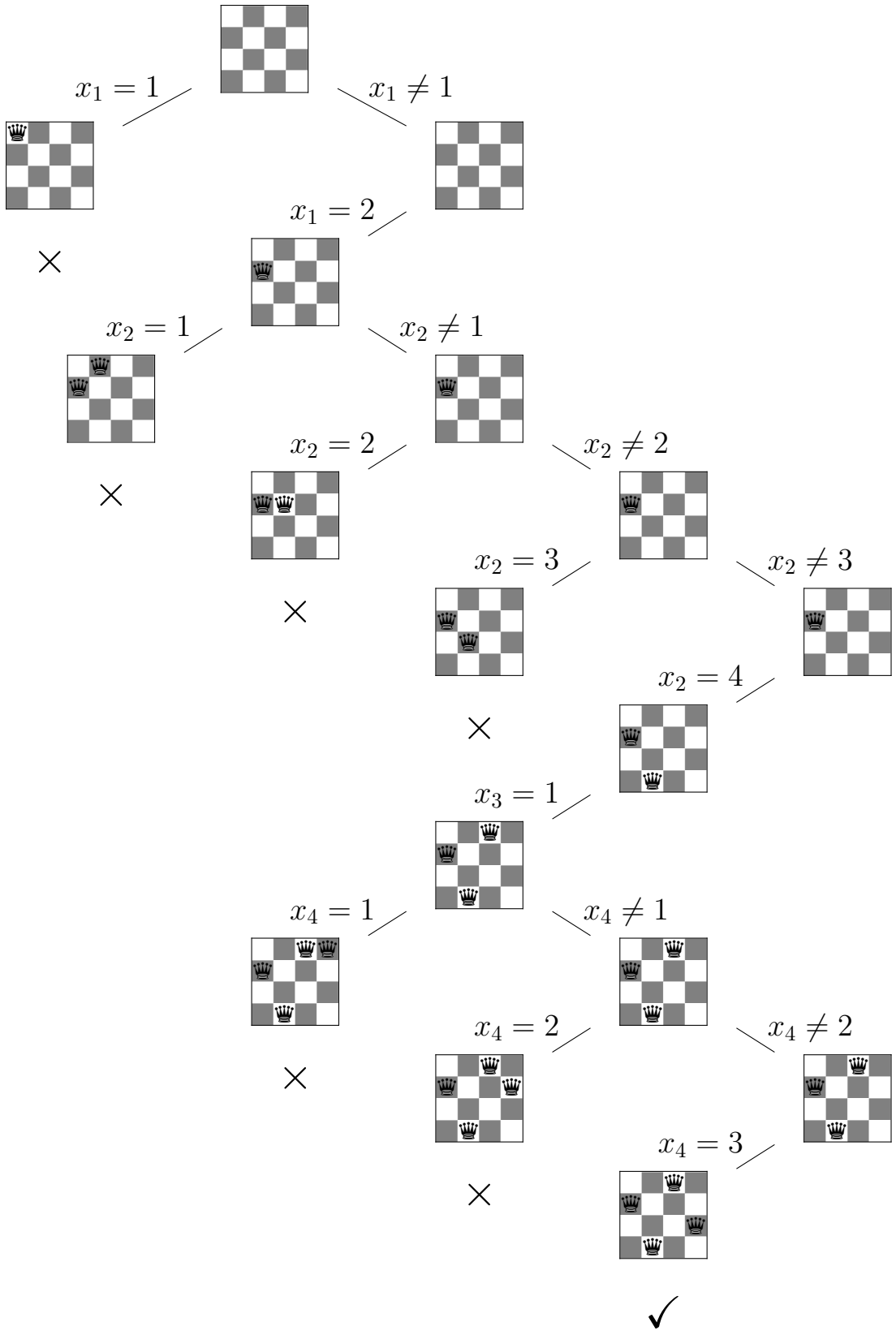


FIGURE 2.17 – Arbre partiel de recherche avec schéma binaire pour le problème des 4-reines (✓ indique une solution au problème et × une instanciation globalement incohérente)

Heuristiques

On distingue trois catégories d'heuristiques de choix de variables :

— statiques ;
— non adaptatives dynamiques ;
— adaptatives dynamiques.

Le type d'heuristique dépend de la nature des informations utilisées pour estimer et évaluer parmi les variables celles permettant d'améliorer les performances de parcours de l'espace de recherche. Cependant, la plupart des heuristiques de choix de variables sont basées sur le principe de *fail-first* émanant du célèbre « Pour réussir, essaies d'abord là où tu as le plus de chance d'échouer » [HARALICK & ELLIOTT 1979]. En d'autres termes, la tendance de ces heuristiques est de choisir d'assigner en priorité les variables ayant le plus de chance de mener à un échec et ainsi ne pas trop s'enfoncer dans l'arbre de recherche. Ce manuscrit décrit brièvement ces différentes catégories.

Les heuristiques statiques, ou catégorie `SVO` (pour *Static Variable Ordering*) sont des heuristiques qui explorent l'état initial du réseau à résoudre pour ordonner les variables. Au cours de toute la recherche, cet ordre reste inchangé malgré les simplifications et modifications du réseau qui peuvent se produire. Voici quelques exemples :

— *lexico* : les variables sont choisies par ordre lexicographique de leurs noms, c'est à dire par exemple que la variable nommée x_i sera choisie avant la variable nommée x_j pour tout $i < j$. L'intérêt de cette heuristique réside uniquement dans la manière dont a été modélisé le problème : si une réflexion particulière a été menée et a abouti à nommer d'abord les variables les plus contraintes du réseau, alors cette heuristique est intéressante. Sinon, elle peut être utilisée en complément d'une heuristique plus pertinente afin de casser l'égalité (tout comme l'heuristique *random* consistant à choisir une variable aléatoirement).
— *deg (maxdeg)* [DECHTER & MEIRI 1989] : les variables sont choisies par ordre décroissant de leur degré initial. Il est naturel de penser que les variables impliquées dans le plus grand nombre de contraintes correspondent aux variables les plus contraintes et intuitivement celles ayant le plus de chance de mener à un échec. C'est tout l'intérêt de cette heuristique.

Les heuristiques (non adaptatives) dynamiques, ou `DVO` (pour *Dynamic Variable Ordering*) sont des heuristiques qui exploitent l'état courant du réseau à résoudre pour ordonner les variables comme par exemple le domaine courant des variables et le degré courant des variables. Le degré courant d'une variable est le nombre de contraintes dans lesquelles elle est impliquée et dont la portée contient au moins une autre variable qui ne soit pas singleton. L'avantage de ces heuristiques face aux heuristiques statiques leurs vient du fait d'exploiter la structure courante du réseau de contraintes et non pas uniquement le réseau initial. En effet, le réseau évolue constamment suite aux réductions des domaines des variables et à la suppression de contraintes universelles diminuant le degré des variables impliquées. En d'autres termes, une variable qui était considérée prioritaire au début de la recherche l'est peut être moins à un autre instant de la recherche. Il est indéniable que prendre des décisions basées sur l'état courant d'un réseau améliore les performances de résolution. Voici quelques exemples :

— *dom (mindom)* [HARALICK & ELLIOTT 1979] : les variables sont choisies par ordre croissant de la taille de leur domaine courant. L'idée est que si une variable contient peu de valeurs dans son domaine, cela signifie que le nombre d'assignations possibles à considérer pour cette variable est faible et qu'ainsi donc un échec éventuel peut être détecté plus tôt ;
— *ddeg* [DECHTER & MEIRI 1994] : les variables sont choisies par ordre décroissant de leur degré dynamique. Le principe est le même que pour l'heuristique statique sauf que la déduction est basée ici sur les degrés courants ;
— *dom/ddeg* [BESSIERE & REGIN 1996] : les variables sont choisies par ordre croissant du ratio entre la taille du domaine courant et le degré dynamique. À noter qu'une variante appelée

dom/deg existe et qu'elle est basée sur le ratio entre le domaine courant et le degré initial ;

— *dom+ddeg* ou *Brelaz* [BRELAZ 1979] : les variables sont choisies par ordre croissant de la taille de leur domaine courant, et en cas d'égalité elles sont choisies par ordre croissant de leur degré courant. Cete heuristique a pour but de casser les égalités de taille de domaines.

Les heuristiques adaptatives sont des heuristiques qui exploitent d'une part la structure courante du réseau et d'autre part les enseignements de ce qui a été fait précédemment : on parle d'adaptation par rapport à l'expérience acquise durant la recherche. Voici quelques exemples :

— *wdeg* [BOUSSEMART *et al.* 2004] : l'idée de cette heuristique est d'associer à chaque contrainte un poids, initialisé à 1, et de l'incrémenter dès que cette contrainte est violée durant la recherche. L'intuition ici est que plus une contrainte est violée, plus elle doit être difficile à satisfaire : plus le poids d'une contrainte est élevée, plus cette contrainte est dure. Ainsi, on va parler de degré pondéré pour les variables : il correspond pour une variable à la somme des poids associés aux contraintes impliquant cette variable et possédant dans sa portée au moins une variable non singleton. L'heuristique consiste donc à considérer les variables par ordre décroissant de leur degré pondéré ;

— *impact* [REFALO 2004] : cette heuristique est basée sur le concept d'impact d'une variable. L'impact d'une variable dans un réseau correspond au nombre de valeurs supprimées dans les domaines des autres variables de ce réseau (en d'autres termes, l'ampleur de la réduction de l'espace de recherche) suite à l'assignation de cette variable et à la propagation des contraintes. Plus précisément, l'impact d'une variable correspond à la somme des impacts de chaque assignation de cette variable par une valeur de son domaine. Les différents impacts mesurés durant la recherche pour une variable sont enregistrés et, au niveau du nœud courant, ce sont les impacts moyens de ces variables qui sont exploités pour le choix de la variable à assigner, c'est à dire les moyennes des différents impacts mesurés pour ces variables durant la recherche jusqu'au nœud courant. Les variables sont choisies par ordre décroissant de leur impact moyen ;

— *last conflict* [LECOUTRE *et al.* 2009] : cette heuristique consiste à s'intéresser aux décisions qui ont conduit à un échec. Plus précisément, le principe est de choisir en priorité la variable ayant mené au dernier échec rencontré. Clairement, l'objectif de cette heuristique est de guider la recherche vers la source des conflits ;

— *dom/wdeg* ([BOUSSEMART *et al.* 2004], [LECOUTRE *et al.* 2004] et [HULUBEI & O'SULLIVAN 2005]) : les variables sont choisies par ordre croissant du ratio entre la taille de leur domaine courant et leur degré pondéré. A l'heure actuelle, cette combinaison d'heuristiques est la plus robuste.

Lorsqu'un schéma de branchement binaire est utilisé, il est nécessaire de sélectionner parmi l'ensemble des valeurs composant le domaine de la variable sélectionnée à l'aide d'une heuristique de choix de valeur. La plupart d'entre elles sont basées sur le principe de *succeed-first* [SMITH 1996]. En d'autres termes, la tendance de ces heuristiques est d'affecter en priorité aux variables les valeurs ayant le plus de chance de mener à une solution. Voici quelques exemples :

— *lexico* : les valeurs sont choisies selon l'ordre dans lequel elles se trouvent dans le domaine courant de la variable. Comme pour les variables, cette heuristique permet d'effectuer des choix pertinents uniquement si la modélisation du problème a été pensé de la sorte, mais généralement elle est utilisée uniquement pour casser les égalités (tout comme l'heuristique *random* consistant à choisir une valeur aléatoirement) ;

— *min-conflits* [MINTON *et al.* 1992] et [FROST & DECHTER 1995] : les valeurs sont choisies par ordre croissant du nombre total de conflits qui leur est associé. Pour une valeur, le nombre total de conflits correspond à la somme du nombre de conflits (valeurs non supports) pour cette variable présentes dans les domaines courants des variables voisines à cette variable ;

— *impact* : les valeurs sont choisies par ordre croissant de leur impact, c'est à dire l'impact de leur

affectation à la variable associée.

Pour conclure sur les heuristiques, il est évident que chacune de ces heuristiques fonctionnera plus ou moins efficacement en fonction du problème traité, les spécificités de chacune d'entre elles et des comparaisons pouvant être consultées dans les différentes publications associées et référencées précédemment.

Techniques de *Backtracking* (*look-back*)

Lors du parcours de l'arbre de recherche, des instanciations partielles incohérentes sont atteintes. Elles ne peuvent être étendues par la moindre variable sans violation d'une contrainte ou par l'apparition d'un domaine vide lorsqu'une cohérence est appliquée sur le réseau courant. Par exemple, si l'on reprend le problème des 4-reines, dans l'arbre de recherche par schéma binaire au moment où on réalise l'assignation $x_1 = 1$, il est alors évident qu'étendre une telle instanciation partielle en solution est impossible. Au lieu de continuer d'avancer, on effectue alors un retour en arrière (*Backtracking*) dans l'arbre afin d'éviter de parcourir toute zone de l'espace de recherche que l'on sait stérile.

Différentes approches dites techniques de *look-back* [DECHTER & FROST 2002] existent et constituent une approche rétrospective visant à améliorer le parcours de l'espace de recherche. D'un point de vue général, ces techniques sont basées sur l'idée « Souviens toi de ce que tu as fait pour éviter de répéter les mêmes erreurs » [HARALICK & ELLIOTT 1979]. On cherche à identifier pour chacune des valeurs supprimées du domaine d'une variable l'explication à l'origine de cette suppression. En effet, l'idée est de déterminer la ou les décisions anciennes déjà prises et qui sont responsables de l'échec rencontré, d'y revenir et de les réparer en prenant d'autres décisions. Ne pas prendre en compte les décisions qui ont amené cette instanciation partielle incohérente n'empêchera pas de retomber dessus une prochaine fois. Nous présentons les trois techniques de *Backtracking* les plus connues.

- Le *Backtracking* standard (ou `SBT` [BITNER & REINGOLD 1975]), appelé aussi retour en arrière chronologique, est l'approche complète la plus utilisée pour la résolution pratique d'un `CSP`. Lors d'un échec (atteinte d'une instanciation partielle incohérente), il consiste à revenir dans l'état précédent la dernière décision prise, considérée alors comme la décision coupable de l'impasse. Les structures du `CSP` modifiées suite à cette décision coupable sont restaurées et on choisit la décision suivante selon le branchement et les heuristiques utilisées. Ainsi dans l'exemple des 4-reines, suite à la découverte du réseau courant incohérent après l'assignation de $x_1 = 1$, la décision suivante adoptée est $(x_1 \neq 1)$. Le `SBT` est limité dans le sens où la dernière décision prise n'est pas forcément la cause de l'échec rencontré, et dans ce cas revenir juste avant cette décision n'empêchera pas de tomber à nouveau dans la même impasse et de faire du *thrashing* (reproduire les mêmes erreurs constamment). Toutefois, puisqu'elle permet d'élaguer certaines branches de l'arbre de recherche, l'algorithme `SBT` est strictement plus efficace que l'algorithme *Generate-and-test*.

Exemple 2.13 *Soit un réseau $\mathcal{P} = (X, D, C)$ défini tel quel $X = \{w, x, y, z\}$ avec $dom(w) = dom(x) = dom(y) = dom(z) = \{i, j\}$ et tel que $C = \{c_1, c_2\}$ avec $scp(c_1) = \{w, z\}$, $scp(c_2) = \{x, z\}$, $rel(c_1) : w \neq z$ et $rel(c_2) : x \neq z$.*

Soit l'instanciation partielle courante $\{w = i, x = j, y = i\}$. Quand l'algorithme de recherche tente d'étendre cette instanciation par l'assignation de la variable z, un échec est rencontré car aucune valeur du domaine de z n'est compatible avec l'instanciation partielle ce qui a pour conséquence $dom(z) = \emptyset$. En effet, l'assignation $z = i$ viole la contrainte c_1 et l'assignation $z = b$ viole la contrainte c_2. Il faut donc effectuer un retour en arrière.

Réaliser la technique `SBT` consisterait à revenir à la dernière décision, c'est à dire à $y = i$. Cependant, si on prend effectivement la nouvelle décision $y \neq i$ et plus précisément $y = j$, on s'aperçoit

que l'on rencontre la même impasse : aucune valeur de $dom(z)$ n'est compatible avec l'instanciation partielle courante $\{w = i, x = j, y = j\}$. Cela est dû au fait que y n'est pas à l'origine de conflits avec la variable z.

- La technique de *Backtracking* basée sur les conflits (CBJ pour *Conflit-directed BackJumping* [PROSSER 1993]), appelée aussi retour en arrière intelligent, consiste donc à s'intéresser plus précisément aux décisions qui ont conduit à cet échec. Pour chaque suppression suite à un conflit de valeur d'un domaine, une explication d'élimination est enregistrée : il s'agit des décisions à l'origine de cette suppression. Lorsqu'un échec est rencontré, l'ensemble des explications d'élimination à l'origine de l'échec est analysé et le retour en arrière est effectué au niveau de la décision la plus récente (la plus profonde dans l'arbre) parmi ces explications. Notons que l'heuristique adaptative *last conflict*, présenté précédemment, simule cette technique de saut en arrière basée sur les conflits.
- La technique de *Backtracking* dynamique (DBT pour *Dynamic Backtracking* [GINSBERG 1993]) exploite également les explications d'élimination de valeurs pour déterminer la décision dite coupable sur laquelle revenir. Cependant, contrairement au saut en arrière basé sur les conflits, la technique DBT supprime la décision coupable tout en conservant les autres décisions qui ont été prises. Tout comme le saut en arrière, les explications sont enregistrées et lorsque l'échec est rencontré, la décision désignée coupable est supprimée (les autres décisions sont conservées).

Techniques de propagation (*look-ahead*)

Suite à la présentation des techniques de *look-back* illustrant le *backtracking* lors de la résolution de CSP, intéressons nous maintenant aux techniques de propagation dites de *look-ahead* [DECHTER 2003] ou approches prospectives qui vont permettre d'améliorer les phases d'exploration de l'arbre de recherche.

Bases sur l'idée de « Prévoir et anticiper le futur afin de réussir dans le présent » [HARALICK & ELLIOTT 1979], ces techniques d'inférence consistent à évaluer pour un réseau les conséquences d'une décision, c'est à dire l'impact d'une instanciation partielle courante sur les domaines des variables non assignées par cette instanciation.

Après une décision, c'est à dire à chaque nœud de l'arbre de recherche, une cohérence plus ou moins forte est établie/maintenue sur le réseau et les valeurs localement incohérentes sont détectées ou supprimées. Ces techniques permettent donc de conserver après filtrage uniquement les valeurs localement cohérentes et ainsi réduire potentiellement la taille de l'espace de recherche et les branches à explorer.

Notons également que le filtrage est effectué durant la recherche après chaque décision, mais il est possible également d'établir une cohérence sur le réseau avant le début de la recherche, on parle de phase de pré-traitement. Le but de cette opération est de simplifier le réseau avant toute exploration par la suppression de toutes les valeurs localement incohérentes et par chance de prouver directement soit la satisfaisabilité du réseau (domaine singleton pour chaque variable) soit son insatisfaisabilité (par l'apparition d'un domaine vide pour une variable).

- La cohérence d'arcs singletons (SAC pour *Singleton Arc Coherence*) proposée par [DEBRUYNE & BESSIÈRE 1997], consiste à effectuer un parcours en largeur de profondeur 1 de l'arbre de recherche. Plus précisément, pour chaque variable x du réseau, les valeurs de $dom(x)$ sont assignées une à une et la cohérence d'arcs est exécutée. Si le réseau obtenu après avoir effectué GAC est incohérent, la valeur est supprimée.

De nombreux algorithmes ont été proposés afin d'établir la singleton arc-cohérence. L'algorithme proposé par [DEBRUYNE & BESSIÈRE 1997], nommé SAC1 a une complexité de $\mathcal{O}(n^2 \times d^2 \times \mathcal{O}(GAC))$.

Il consiste à tester une à une les interprétations de taille 1 et à relancer SAC chaque fois qu'une valeur est supprimée.

Une autre approche, proposée par [BARTÀK & ERBEN 2004], consiste à sauvegarder une liste de support afin d'éviter du travail redondant. [BESSIÈRE *et al.* 2005] propose une approche, nommé SAC-OPT, de complexité $\mathcal{O}(m \times n \times d^3)$ dans le cas de réseau binaire. Cependant, afin d'être optimal, SAC-OPT utilise des structures de données conséquentes $\mathcal{O}(m \times n \times d^2)$. Ainsi dans cette même publication, une version allégée (mais non-optimale) nommée SAC-SDS est proposée afin d'effectuer un compromis au niveau de la mémoire. Finalement, [LECOUTRE & CARDON 2005] propose une approche, nommée SAC3, et qui contrairement aux autres approches ne redémarre pas à chaque suppression de valeurs.

Notons qu'utiliser SAC demande beaucoup de temps au vue de sa complexité, ce qui en fait une méthode généralement utilisée en pré-traitement contrairement aux méthodes que nous présentons maintenant.

- Le *backward checking* (BC), aussi nommé le test de cohérence par l'arrière, est la technique de *look-ahead* de base. Elle consiste tout simplement à vérifier après l'affectation d'une valeur à une variable que l'instanciation partielle courante obtenue suite à cette nouvelle assignation est cohérente, c'est à dire que toutes les contraintes couvertes par cette instanciation sont satisfaites.

Exemple 2.14 *Un exemple d'application est réalisé lors de la résolution du problème des 4-reines dans la figure 2.17 suite à l'assignation de $x_2 = 1$, un test de cohérence est réalisé sur l'instanciation $\{x_1 = 2, x_2 = 1\}$. Une incohérence est détectée car une contrainte est violée suite à l'apparition de 2 reines sur une même diagonale. On revient en arrière et on passe à la décision suivante $x_2 \neq 1$: la valeur 1 est supprimée du domaine de x_2*

- Le *Forward checking* (FC [HARALICK & ELLIOTT 1979]) applique une forme partielle d'arc-cohérence sur le réseau après chaque décision. On parle d'arc-cohérence partielle, ou *partial look-ahead*, dans le sens où seules les variables voisines de la variable qui vient d'être assignée vont être révisées.
- Le maintien de la cohérence d'arc (MAC pour *Maintening (Generalized) Arc-Consistancy* [SABIN & FREUDER 1994]) maintient l'arc-cohérence pendant la recherche. À la différence de FC, l'arc-cohérence va être établie pour l'ensemble du réseau et pas seulement pour les variables voisines de la variable qui vient juste d'être assignée. L'algorithme 2.12 illustre MAC où $\texttt{doGAC}^{str}(\mathcal{P}, x)$ représente la procédure de maintenance d'arc-consistance généralisée d'un réseau de contraintes.

Ce dernier est à l'heure actuelle l'algorithme de résolution du problème CSP le plus efficace. De plus, [LECOUTRE 2011] montre que MAC rencontre les meilleures performances en utilisant l'algorithme STR2 (Algorithme 2.10) pour maintenir GAC.

Exemple 2.15 *La figure 2.18 illustre l'arbre de recherche obtenu par l'application de MAC où les heuristiques de choix de variables et de valeurs sont données par l'ordre lexicographique.*

Comme le montre la figure 2.18, utiliser l'algorithme MAC permet de réduire fortement la taille de l'arbre. La première décision correspond à l'assignation de $x_1 = 1$. Elle est propagée et comme vu lors de l'exemple 2.9 permet de réduire (par inférence) le domaine de x_1 à $\{2, 3, 4\}$. La prochaine décision $x_1 = 2$ appliquée avec arc-cohérence permet de déduire que $x_2 = 4$ (car seule la valeur 4 ne viole pas de contrainte). Puis la propagation par contraintes de l'instantiation partielle obtenue $\{x_1 = 2, x_2 = 4\}$ permet de filtrer les domaines de x_3 et x_4 en les simplifiant à des singletons ($dom(x_3) = \{1\}$ et $dom(x_4) = \{3\}$). L'instanciation globalement cohérente (la solution) est ainsi directement obtenue.

Pour conclure cette section concernant les stratégies de recherche, l'efficacité d'un algorithme de résolution de CSP dépend de la manière de combiner les différentes techniques proposées par les composants du modèle $BPRA$ (*Branchement Propagation Retour-arrière Apprentissage*), plus précisément la manière d'associer techniques de *look-back* et techniques de *look-ahead*.

Algorithme 2.12 : MAC

Données : CSP $\mathcal{P} = (X, D, C)$, Instanciation partielle I
Résultat : Instanciation complète I (solution) ou $\perp$ (non satisfiable)

si $\exists\, x \in X \mid dom(x) = \emptyset$ **alors**
  retourner $\perp$
fin
si $vars(I) = |X|$ **alors**
  retourner I
fin
Sélectionner $x \in X \mid x \notin I$
Sélectionner $v \in dom(x)$
CSP $\mathcal{P}' = (X' \leftarrow X, D' \leftarrow D, C' \leftarrow C \cup \{x = v\})$
si $doGAC^{str}(\mathcal{P}', x) = vrai$ **alors**
  $I' \leftarrow \texttt{MAC}(\mathcal{P}', I \cup \{x = v\})$
  si $I' \neq \perp$ **alors**
    retourner I'
  fin
fin
$dom(x) \leftarrow dom(x) \backslash \{v\}$
retourner *MAC* $(\mathcal{P}, I)$

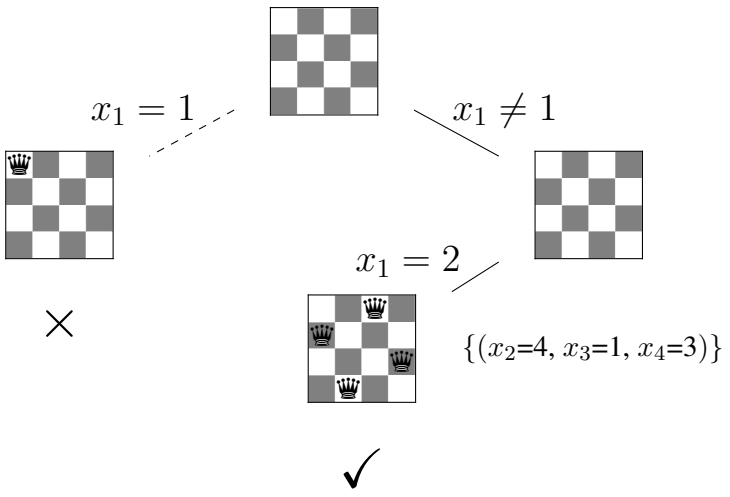


FIGURE 2.18 – Arbre de recherche partiel construit par l'algorithme MAC sur le problème des 4-reines.

Il est intéressant de connaître les associations qui constituent les algorithmes les plus connus (généralement les plus efficaces) pour la résolution des CSP. Ces algorithmes sont résumés dans le tableau 2.2.

Algorithme	***look-ahead***	***look-back***	**Auteur**
BT	BC	SBT	–
CBJ	BC	CBJ	[PROSSER 1993]
DBT	BC	DBT	[GINSBERG & MCALLESTER 1994]
FC	FC	SBT	[HARALICK & ELLIOTT 1979]
MAC	MAC	SBT	[SABIN & FREUDER 1994]

TABLE 2.2 – Différentes associations des techniques de *look-ahead* et de *look-back* présentées.

2.3 Réseaux de contraintes stochastiques (SCSP)

Le cadre CSP permet de modéliser et de résoudre un grand nombre de problèmes. Cependant, certains problèmes demandent plus de souplesse et ne sont pas modélisables par un CSP, c'est à dire uniquement sous la forme d'instanciations strictement autorisées ou strictement interdites par les contraintes. Suite à ce constat, de nombreuses extensions du cadre CSP ont été proposées pour proposer cette flexibilité et permettent d'appréhender divers facteurs : de préférences (CSP flou [FARGIER *et al.* 1993]), de priorités (CSP possibiliste [SCHIEX 1992]), de violations (Max-CSP [LARROSA *et al.* 1999]), de pénalités ou de coûts (CSP pondéré (WCSP) [LARROSA & SCHIEX 2004]), etc.

Il existe un grand nombre de problèmes de décisions dont la notion d'incertitude est fondamentale. Par exemple, la représentation de données au sujet d'événements passés ou futurs étant impossible à déterminer avec exactitude comme les prévisions météorologiques, les préférences des utilisateurs, la demande de produits, etc.

Ainsi, dans le but de modéliser des problèmes de décision combinatoire permettant la prise en compte d'incertidudes et de probabilités, de nombreuses extensions au cadre des CSP ont été proposées comme les MCSP (pour *Mixed Constraint Satisfaction Problem*), les PCSP (pour *Probabilistic Constraint Satisfaction Problem*, les BCSP (pour *Branching Constraint Satisfaction Problem*) et enfin les SCSP (pour *Stochastic Constraint Satisfaction Problem*). Au cours de cette thèse, nous nous intéressons à cette dernière extension inspirée du problème de satisfaction stochastique [LITTMAN *et al.* 2001] car elle permet d'appliquer les meilleures techniques de résolution des CSP à la résolution des problèmes de satisfaction stochastique (propagation de contraintes, etc) mais également l'utilisation des contraintes globales ou des opérateurs arithmétiques, contrairement aux autres modèles. Afin de distinguer les différents modèles, le lecteur pourra se référer à [VERFAILLIE & JUSSIEN 2005].

2.3.1 Définitions et Notations

Le problème de satisfaction de contraintes stochastiques [WALSH 2009] est initié en 2002. La formalisation de cette extension s'appuie sur un réseau de contraintes stochastiques où une distinction est réalisée entre deux types de variables : les variables de décisions et les variables stochastiques.

Une variable stochastique est une variable dont une distribution de probabilités est définie sur l'ensemble des valeurs définissant son domaine.

Définition 2.26 (Variable stochastique) *Soit y une variable stochastique où $dom(y)$ représente le domaine de y. À chaque variable stochastique est associée une distribution de probabilités : il s'agit de l'ensemble des probabilités définit sur chaque valeur v du domaine de y ($v \in dom(y)$) tel que v soit affecté à y.*

On note $P(y = v)$ la probabilité que la valeur v soit affectée à la variable stochastique y et P_y la distribution de probabilité associée à y. Notons que $\sum_{i=1}^{|dom(y)|} P(x = v_i) = 1$

Définition 2.27 (Variable de décision) *Une variable de décision est l'appellation donnée à une variable non stochastique. C'est à dire une variable dont il n'existe pas de distribution de probabilité définie sur son domaine.*

Remarque 2.4 *Afin de marquer la différence entre les symboles utilisés pour désigner les différentes variables, dans la suite de ce manuscrit, nous utiliserons x_s pour désigner une variable stochastique et x_d pour une variable de décision. Si x est utilisé pour désigner une variable, elle peut être de décision ou stochastique.*

La portée d'une contrainte c (toujours nommée $scp(c)$) est définie sur X. La satisfaction d'une contrainte est la même que celle du cadre CSP.

Définition 2.28 (Réseau de Contraintes Stochastiques) *Un réseau de contraintes stochastiques est un 6-tuple (X, Y, D, P, C, θ) tel que :*

- *X est l'ensemble fini et ordonné des n variables du problème ;*
- *Y est le sous-ensemble fini de X des variables stochastiques du problème ($Y \subset X$) ;*
- *D est l'ensemble fini des domaines de X ($\forall x \in X, dom(x) \in D$) ;*
- *P est l'ensemble fini des distributions de probabilités sur les domaines des variables stochastiques composant Y ($\forall x_s \in Y, P_{x_s} \in P$) ;*
- *C est l'ensemble fini des m contraintes du problème portant sur les variables composants X .*
- *θ est un seuil compris entre 0 et 1 inclus ($\theta \in [0, 1]$).*

Remarque 2.5 *Dans la suite de ce manuscrit, nous noterons V l'ensemble des variables de décisions d'un réseau de contraintes stochastiques $\mathcal{P} = (X, Y, D, P, C, \theta)$ tel que $V = X \backslash Y$*

Exemple 2.16 *Soit un réseau de contraintes stochastiques $\mathcal{P} = (X, Y, D, P, C, \theta)$ avec :*

- *$X = \{x_{d1}, x_{s1}, x_{d2}, x_{s2}\}$;*
- *$Y = \{x_{s1}, x_{s2}\}$;*
- *$D = \{dom(x_{d1}), dom(x_{s1}), dom(x_{d2}), dom(x_{s2})\}$ avec $dom(x_{d1}) = dom(x_{s1}) = dom(x_{d2}) = dom(x_{s2}) = \{0, 1, 2\}$;*
- *$P = \{P_{x_{s1}}, P_{x_{s2}}\}$ avec $P_{x_{s1}} = \{P(x_{s1} = 0) = \frac{1}{3}, P(x_{s1} = 1) = \frac{1}{3}, P(x_{s1} = 2) = \frac{1}{3}\}$ et $P_{x_{s2}} = \{P(x_{s2} = 0) = \frac{1}{3}, P(x_{s2} = 1) = \frac{1}{3}, P(x_{s2} = 2) = \frac{1}{3}\}$;*
- *$C = \{c_1, c_2, c_3\}$ avec $scp(c_1) = \{x_{d1}, x_{d2}\}$ où $c_1 : x_{d1} = x_{d2}$, $scp(c_2) = \{x_{d1}, x_{s1}\}$ où $c_2 : x_{d1} + x_{s1} > 1$ et $scp(c_3) = \{x_{d2}, x_{s2}\}$ où $c_2 : x_{d1} + x_{s2} > 1$;*
- *$\theta = \frac{1}{3}$.*

Dans un réseau de contraintes stochastiques, les variables composant X doivent être instanciées dans leur ordre d'apparition dans ce même ensemble. De plus, les variables composants les portées de chaque contrainte $c \in C$ sont triées selon leur ordre d'apparition dans X.

La définition d'une instanciation pour un réseau de contraintes stochastiques est la même que pour un CSP (Définition 2.1). Les assignations des variables stochastiques dans une instanciation sont appelées **scénario** et une probabilité lui est associée.

Définition 2.29 (Probabilité d'un scénario) *Soit I une instanciation d'un réseau de contraintes stochastiques $\mathcal{P} = (X, Y, D, P, C, \theta)$ où $Y = \{x_{s1}, x_{s2}, \ldots, x_{sn}\}$ ($|Y| = sn$). La probabilité associée au scénario de I notée $P(I)$ correspond au produit des probabilités d'assignation des valeurs aux variables stochastiques qui le compose. Formellement, elle est définie par : $\prod_{i=1}^{sn} P(x_{si} = v_i)$.*

Définition 2.30 (Politique d'un réseau de contraintes stochastiques) *Soit un réseau de contraintes stochastiques $\mathcal{P} = (X, Y, D, P, C, \theta)$. Une politique est un arbre où :*

- *Chaque nœud est étiqueté par une variable composant X dont la racine est la première variable apparaissant dans X et se terminant par la dernière variable de X. Les nœuds étiquetés par une variable de décision n'ont qu'un seul fils et les nœuds étiquetés par une variable stochastique ont un fils pour chaque valeur composant le domaine de cette variable ;*
- *Chaque arête est étiquetée par la valeur assignée à la variable étiquetant le nœud qui la précède ;*
- *Chaque feuille correspond à l'instanciation composée des valeurs assignées aux variables sur le chemin entre la feuille et la racine. Elle est étiquetée par la valeur 1 si l'instanciation associée satisfait toutes les contraintes de C, sinon elle est étiquetée par 0.*

Notons qu'une politique π peut (comme une instanciation) être **partielle**, on note $vars(\pi)$ les variables instanciées de V du réseau de contraintes stochastiques. Une politique π **couvre** une contrainte c si et seulement si $scp(c) \in vars(\pi)$.

La **satisfaction** d'une politique est définie comme la somme de chaque valeur étiquetant chaque feuille qui la compose pondérée par la probabilité du scénario correspondant à la feuille. Une politique π d'un réseau de contraintes stochastiques $\mathcal{P} = (X, Y, D, P, C, \theta)$ est dite **politique solution** si sa satisfaction est supérieure ou égale au seuil θ satisfaisant par la même occasion l'ensemble des contraintes C. L'ensemble des politiques solutions d'un réseau de contraintes stochastiques $\mathcal{P}$ est noté *sols($\mathcal{P}$)*.

Définition 2.31 (Problème de satisfaction de contraintes stochastiques (`SCSP`)) *Un problème de satisfaction de contraintes stochastiques (`SCSP` pour Stochastic Constraint Satisfaction Problem) est le problème de décision qui consiste à savoir si il existe au moins une politique satisfaisant le réseau de contraintes stochastiques associé. Par abus de langage, un réseau de contraintes stochastiques peut aussi être appelé `SCSP`.*

Exemple 2.17 *Reprenons l'exemple 2.16, la Figure 2.19 représente deux politiques pour le problème `SCSP` correspondant. La politique de gauche représente la décision d'assigner la valeur 0 aux deux variables de décisions x_{d1} et x_{d2} et celle de droite la décision d'assigner la valeur 1 à ces mêmes variables. Dans les deux politiques, la contrainte $c_1 : x_{d1} = x_{d2}$ portant uniquement sur les variables de décisions est satisfaite.*

Dans chaque politique, les neuf chemins entre chaque feuille et la racine représentent les neuf scénarios possibles. Par exemple, le scénario où la valeur 0 est assignée aux deux variables stochastiques x_{s1} et x_{s2} est le chemin entre la feuille la plus à gauche et la racine de chaque politique. Ce scénario ne satisfait pas les contraintes $c_2 : x_{d1} + x_{s1} > 1$ et $c_3 : x_{d2} + x_{s2} > 1$, la feuille correspondante est alors étiquetée par la valeur 0. Contrairement au scénario représenté par le chemin entre la feuille la plus à droite et la racine où la valeur 2 est assignée aux deux variables stochastiques qui est étiquetée par la valeur 1.

La satisfaction associée à chaque politique est égale à la somme de chacune des valeurs étiquetant les feuilles pondérées par la probabilité du scénario qui correspond. Dans cet exemple, la probabilité est uniforme pour chaque scénario, elle est correspond à $\frac{1}{3} \times \frac{1}{3} = \frac{1}{9}$. Ainsi la satisfaction s_g de la politique de gauche est $\frac{1}{9}$ et la satisfaction s_d de la politique de droite est $\frac{4}{9}$.

Pour une valeur de seuil $\theta = \frac{1}{3}$. on a alors $s_g < \theta$ signifiant que la politique de gauche ne satisfait pas le `SCSP` correspondant. Toutefois, nous avons $s_d > \theta$ signifiant que la politique de droite est une

solution pour le problème. Notons que pour ce problème, ce n'est pas la politique solution optimale car si on assigne la valeur 2 *aux deux variables de décisions, quelque soit la valeur des variables stochastiques, le* SCSP *est satisfait. La satisfaction de la politique solution optimale est ici de* 1.

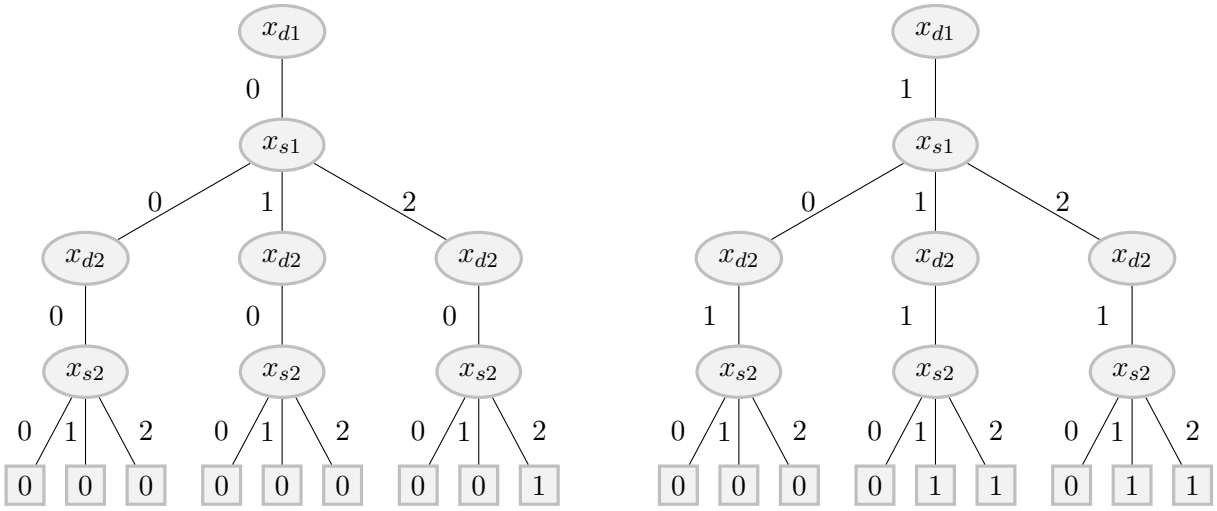


FIGURE 2.19 – Deux politiques pour le SCSP de l'exemple 2.16.

Évidemment déterminer si un problème de satisfaction de contraintes stochastiques est satisfaisable possède une complexité plus élevée qu'un CSP.

Propriété 2.5 *Décider si un problème de satisfaction de contraintes stochastiques est satisfaisable est un problème* PSPACE-complet *[*TARIM et al. *2006].*

Remarque 2.6 *Comme le montre [*WALSH *2009], il est possible de caractériser le problème* SCSP *dans une sous classe de la classe* PSPACE *désignée par NP^{PP}. Elle correspond à la classe de problèmes décidés en temps non polynomial par une machine de Turing non-déterministe en faisant appel à un oracle capable de vérifier en un pas de calcul si la décision est correcte en temps polynomialement probabiliste. Notons que $NP \subseteq PP \subseteq NP^{PP} \subseteq PSPACE$*

Il est possible de représenter un réseau de contraintes stochastiques par un m-SCSP (un SCSP à m niveaux) où chaque niveau représente un sous-ensemble de X nommé $X_i = V_i \cup Y_i$ tel que $X = \bigcup_{i=1}^{m} X_i$. Dit autrement $X = \{V_1, Y_1, V_2, Y_2, \ldots, V_m, Y_m\}$.

Remarque 2.7 *Notons que cette modélisation n'est pas restrictive même si il est possible que l'ensemble X des variables composant un* SCSP *commence par des variables stochastiques ($V_1 = \emptyset$) et se termine par des variables de décisions ($Y_m = \emptyset$).*

Exemple 2.18 *L'exemple 2.16 correspond à un* 2-SCSP *où $X_1 = \{x_{d1}, x_{s1}\} et\ X_2 = \{x_{d2}, x_{s2}\}$.*

Notons que le SCSP à un seul niveau (1-SCSP) est nommé μSCSP (pour micro-SCSP) dans ce manuscrit.

Définition 2.32 (micro-SCSP (μSCSP)) *Un micro-*SCSP *est un* SCSP *$\mathcal{P} = (X, Y, D, C, P, \theta)$ dont l'ensemble X composant les n variables est ordonné strictement par l'ensemble des dn variables de décisions composant V puis par l'ensemble des sn variables stochastiques composant Y.*

Dit autrement, si $V = \{x_{d1}, x_{d2}, \ldots, x_{dn}\}$ et $Y = \{x_{s1}, x_{s2}, \ldots, x_{sn}\}$ alors $X = V \cup Y = \{x_{d1}, x_{d2}, \ldots, x_{dn}, x_{s1}, x_{s2}, \ldots, x_{sn}\}$.

Un μSCSP peut être satisfait si il existe un ensemble d'assignations pour les variables de décisions tel qu'il existe des scénarios possibles dont la somme des probabilités de chaque scénario est supérieure ou égale au seuil attendu.

Si on étend la résolution d'un μSCSP à un m-SCSP, on peut comprendre que la résolution d'un tel problème est de déterminer un ensemble d'assignations pour les variables de décisions composant le premier niveau (μSCSP_1) où soit un ensemble de valeurs aléatoires données pour les variables stochastiques, il est possible de déterminer un ensemble d'assignations pour les variables de décisions composant le second niveau (μSCSP_2), etc ...jusqu'au dernier niveau (μSCSP_m), tel qu'il existe des scénarios possibles dont la somme des probabilités de chaque scénario est supérieure ou égale au seuil θ attendu.

Il est important de préciser que les contraintes d'un SCSP porte sur l'ensemble des différents niveaux le composant, il n'est donc pas possible de décomposer l'ensemble des contraintes C comme il est possible de le faire pour l'ensemble X des variables.

Dans la suite de cette section, nous présentons une technique de *look-back* et une technique de *look-ahead* adaptées du cadre CSP permettant de décider si un SCSP est satisfaisable.

2.3.2 Algorithmes de résolutions

Les algorithmes de résolutions présentés ci-après nécessitent de redéfinir la cohérence d'une valeur pour un SCSP.

Définition 2.33 (Cohérence d'une valeur) *Soit un SCSP $\mathcal{P} = (X, Y, D, P, C, \theta)$ et x une variable de X. Une valeur $v \in dom(x)$ est dite cohérente si et seulement si il existe au moins une politique solution dans laquelle l'assignation $x = v$ est inclus.*

Intuitivement, pour une variable x, une valeur v est incohérente si l'assignation $x = v$ ne peut participer à aucune politique solution.

Le *Backtracking* standard, la technique la plus utilisée de *look-back* du cadre CSP a été adaptée pour les réseaux de contraintes stochastiques [WALSH 2009]. SSBT pour *Standard Stochastic Backtracking* est illustré récursivement par l'algorithme 2.13. Elle représente une exploration en profondeur de l'arbre de recherche correspondant au SCSP afin de déterminer si il existe une politique π dont la satisfaction associée s est supérieure à θ et de la retourner.

Cette technique explore chaque couple variable-valeur (x_i, v) dans leur ordre d'apparition dans X. L'algorithme 2.14 d'initialisation de SSBT nommé $init_{SSBT}$ initialise i à 1 représentant la première variable de X, la politique π à l'ensemble vide et $\theta_{courant}$ à θ, cette dernière valeur est utilisée pour les appels récursifs de SBT.

Ainsi pour chaque couple-valeur (x_i, v), si l'assignation correspondante $(x_i = v)$ est cohérente pour l'ensemble des contraintes C (estCohérent$(x_i = v, C)$ retourne vrai si il existe un tuple autorisé dans chaque contrainte $c \in C$ qui comprend l'assignation $(x_i = v)$), l'algorithme adopte un comportement différent selon le type de variable que représente x_i :

- Dans le cas où x_i est une variable stochastique, SSBT est appelé récursivement afin de déterminer une politique satisfaisant le sous problème dont la variable x_i courante vient d'être assignée (on incrémente i de 1) et dont le seuil θ attendu pour ce nouveau SCSP est instancié à $\frac{\theta_{courant} - s}{p}$ correspondant au seuil nécessaire pour satisfaire le SCSP où la variable x_i est satisfaite. Suite à cet appel, la satisfaction courante est incrémentée par la satisfaction retournée par l'appel de SSBT pondérée par la probabilité de l'assignation courante ($P(x_i = v)$). Enfin, si la satisfaction courante est supérieure au seuil courant on retourne la politique et sa satisfaction associée.
- Dans le cas où x_i est une variable de décision, l'assignation $\{(x_i = v)\}$ est ajoutée à la politique π (ligne 16). Puis $SSBT$ est appelé en incrémentant i de 1 afin d'explorer la variable suivante dans X. Finalement, le maximum entre la satisfaction courante et la satisfaction retournée par SSBT

est comparée au seuil courant. Si le maximum est plus grand, alors la politique est retournée sinon, on supprime l'assignation $\{(x_i = v)\}$ de la politique π (ligne 23) provoquant un retour en arrière.

SSBT se termine quand la dernière variable de X est assignée. Finalement, après l'appel de SSBT dans l'Algorithme 2.14 $init\_SSBT$, on verifie que la polique retournée possède autant d'assignations que de variables de décisions (ligne 2), si c'est le cas, une politique solution a été trouvée, sinon on retourne FAILURE, indiquant que le SCSP n'est satisfaisable.

Algorithme 2.13 : $SSBT$

Données : SCSP $\mathcal{P} = (X, Y, D, P, C, \theta)$, une politique π, un seuil $\theta_{courant}$, un entier i
Résultat : Une politique π, une satisfaction s

1 **si** $i \geq |X|$ **alors**
2 **retourner** $(\pi, 1)$
3 **fin**
4 $s \leftarrow 0$
5 **pour chaque** $v \in dom(x_i)$ **faire**
6 **si** *estCohérent*$(x_i = v, C)$ **alors**
7 **si** $x_i \in Y$ **alors**
8 $p \leftarrow P(x_i = v)$
9 $(\pi, s_t) \leftarrow SSBT(\mathcal{P}, \pi, \frac{\theta_{courant} - s}{p}, i + 1)$
10 $s \leftarrow s + p \times s_t$
11 **si** $s \geq \theta_{courant}$ **alors**
12 **retourner** (π, s)
13 **fin**
14 **fin**
15 **sinon**
16 $\pi \leftarrow \pi \cup \{(x_i = v)\}$
17 $(\pi, s_t) \leftarrow SSBT(\mathcal{P}, \pi, \theta_{courant}, i + 1)$
18 $s \leftarrow max(s, s_t)$
19 **si** $s \geq \theta_{courant}$ **alors**
20 **retourner** (π, s)
21 **fin**
22 **sinon**
23 $\pi \leftarrow \pi \backslash \{(x_i = v)\}$
24 **fin**
25 **fin**
26 **fin**
27 **fin**
28 **retourner** (π, s)

Notons que les variables devant être instanciées dans leur ordre d'apparition dans X, il n'est donc pas possible d'appliquer d'heuristique sur l'ordre des variables dans un SCSP (ce qui n'est pas le cas d'un μSCSP où il est possible de réordonner le sous-ensemble V des variables de décisions). Toutefois, les heuristiques de valeurs peuvent quant à elle être utilisées pour optimiser la recherche.

Comme vu dans la section détaillant les techniques de recherche dans le cadre des CSP, suite aux techniques de retour en arrière (*look-back*), viennent les techniques de propagation (*look-ahead*).

[WALSH 2009] et [BALAFOUTIS & STERGIOU 2006] adaptent la technique de *Forward Checking*

Algorithme 2.14 : $init\_SSBT$

Données : SCSP $\mathcal{P} = (X, Y, D, P, C, \theta)$
Résultat : Une politique π
1 $(\pi, s) \leftarrow SSBT(\mathcal{P}, \emptyset, \theta, 1)$
2 **si** $|\pi| \neq |V|$ **alors**
3 | **retourner** *FAILURE*
4 **fin**
5 **retourner** π

au cadre SCSP illustré récursivement dans l'Algorithme 2.15 et nommé dans ce manuscrit SFC pour *Stochastic Forward Checking*.

Comme pour SBT, l'algorithme SFC explore récursivement chaque couple variable-valeur (x_i, v_j) dans leur ordre d'apparition dans X. De ce fait, SFC a besoin de l'Algorithme 2.16 $init_{SFC}$ pour initialiser ses paramètres d'entrée.

SFC est proche de SBT algorithmiquement, nous détaillerons donc ici uniquement les différences entre les deux algorithmes.

Avant chaque assignation d'une variable, SFC utilise l'inférence (via l'Algorithme check 2.17) sur les variables assignées et filtre les valeurs des domaines des variables qui ne sont pas encore assignées et qui provoqueront l'apparition d'instanciations localement incohérentes (*nogood*). Dit autrement, ils suppriment les valeurs des domaines des variables non assignées ne permettant pas d'étendre l'instanciation courante vers une solution. Pour cela SFC utilise différents composants :

— un tableau $prune(i, j)$ permettant de stocker la profondeur dans l'arbre où chaque valeur $v_j \in dom(x_i)$ a été supprimée du domaine par SFC ;
— À chaque variable stochastique x_{si}, on associe une valeur q_i indiquant la probabilité que les valeurs qui ne sont pas encore supprimées dans $dom(x_{si})$ peuvent contribuer à une solution du problème. Chaque q_i est initialisée à 1 ;
— Les seuils θ_{bas} et θ_{haut} sont initialement instanciés à la valeur de seuil θ du problème SCSP à résoudre. Ces deux valeurs sont utilisées pour filtrer l'arbre de recherche.

Supposons que s_i correspond à la satisfaction qu'apporte l'assignation de v à la variable stochastique x_{si} et s la satisfaction de l'instanciation courante. Les autres valeurs de cette variable peuvent donc être ignorées, ainsi on a $s + p \times s_i \geq \theta$ où $p = P(x_{si} = v)$. Ce qui justifie le seuil « haut » θ_{haut} et sa mise à jour à chaque appel de SFC (ligne 11) quand une valeur est assignée à une variable stochastique. De même, il est impossible de trouver une politique solution pour le SCSP si $s + p \times s_i + q_i \leq \theta$. Ceci justifie le seuil « bas » θ_{bas} et sa mise à jour à chaque appel de SFC à la même ligne.

Maintenant supposons que s_i correspond à la satisfaction qu'apporte l'assignation de la variable de décision x_i. Si cette valeur est supérieure à θ_{bas} alors toutes les autres valeurs à assigner doivent être supérieures à s_i pour faire partie d'une politique avec une meilleure satisfaction. Il est donc possible de mettre à jour θ_{bas} par $(max(s, \theta_{bas})$ à chaque appel de SFC quand une variable de décision est assignée (ligne 23).

La procédure $restore(i)$ est appelée par SFC quand une instanciation partielle est rejetée et quand un retour en arrière est réalisé afin de restaurer les valeurs des domaines des variables non assignées qui ont été supprimées par inférence précédemment et pour ré-incrémenter q_i par la probabilité correspondant aux valeurs supprimées précédemment pour chaque variable stochastique.

La procédure $check$ pour un assignement (x_i, v_j) renvoie faux si le domaine d'une variable est vide (dv = vrai) ou si il y a trop de valeurs supprimées par inférence dans le domaine d'une variable pour que sa satisfaction maximum ms (pondérée par la probabilité $P(x_i = v_j)$ si c'est une variable stochastique)

Algorithme 2.15 : SFC

Données : SCSP $\mathcal{P} = (X, Y, D, P, C, \theta)$, une politique π, un seuil bas θ_{bas}, un seuil haut θ_{haut},un entier i
Résultat : Politique π, satisfaction s

si $i \geq |X|$ **alors**
  retourner $(\pi, 1)$
fin
$s \leftarrow 0$
pour chaque $v_j \in dom(x_i)$ **faire**
  si $prune(i, j) = 0$ **alors**
    si $check(x_i \leftarrow v_j, q_i, \theta_{bas}, s)$ **alors**
      si $x_i \in Y$ **alors**
        $p \leftarrow P(x_i = v_j)$
        $q_i \leftarrow q_i - p$
        $(\pi, s_t) \leftarrow SFC(\mathcal{P}, \pi, \frac{\theta_{bas}-s}{p}, \frac{\theta_{haut}-s}{p}, i+1)$
        $s \leftarrow s + p \times s_t$
        $restore(i)$
        si $(s + q_i) < \theta_{bas}$ **alors**
          retourner (π, s)
        fin
        si $s \geq \theta_{haut}$ **alors**
          retourner (π, s)
        fin
      fin
      sinon
        $\pi \leftarrow \pi \cup \{(x_i = v_j)\}$
        $(\pi, s_t) \leftarrow SFC(\mathcal{P}, \pi, max(s, \theta_{bas}), \theta_{haut}, i+1)$
        $s \leftarrow max(s, s_t)$
        $restore(i)$
        si $s \geq \theta_{haut}$ **alors**
          retourner (π, s)
        fin
        sinon
          $\pi \leftarrow \pi \backslash \{(x_i = v)\}$
        fin
      fin
    fin
    sinon
      $restore(i)$
    fin
  fin
fin
retourner (π, s)

Algorithme 2.16 : $init\_SFC$

Données : SCSP $\mathcal{P} = (X, Y, D, P, C, \theta)$
Résultat : Une politique π
1 $(\pi, s) \leftarrow SFC(\mathcal{P}, \emptyset, \theta, \theta, 1)$
2 **si** $|\pi| \neq |V|$ **alors**
3 | **retourner** *FAILURE*
4 **fin**
5 **retourner** π

soit suffisante pour trouver une politique solution au problème. La satisfaction maximum ms d'une valeur v_j à la variable courante x_i à assigner est égale à $\prod_{s=i+1}^{n} \sum_{t=1}^{|dom(x_s)|} P(x_s = v_t)$ avec $s > i$. Dit autrement, ms correspond au produit de toutes les sommes des probabilités des valeurs restantes des domaines de toutes les variables non (encore) assignées.

La définition de la cohérence pour un problème SCSP (Définition 2.33) impose que pour déterminer l'incohérence d'une valeur, il est nécessaire de trouver l'ensemble des politiques solutions d'un SCSP. Ainsi [BALAFOUTIS & STERGIOU 2006] et [TARIM *et al.* 2006] proposent des algorithmes incomplets en pratique permettant de décider si un problème SCSP est satisfaisable ou non plus efficacement.

Les travaux réalisés au cours de cette thèse nécessitent des algorithmes de résolution complets en pratique, toutefois une (courte) présentation des algorithmes [BALAFOUTIS & STERGIOU 2006] et [TARIM *et al.* 2006] est réalisée dans la sous-section suivante justifiant de leurs incomplétudes.

2.3.3 Quelques algorithmes incomplets

L'arc-cohérence est un concept important des CSP comme vu en section 2.2.2 à la base de la propagation par contraintes. [BALAFOUTIS & STERGIOU 2006] introduit cette même notion à l'extension stochastique des réseaux de contraintes.

La définition de l'arc-cohérence et des techniques de filtrage associées aident fortement les stratégies de recherche par un raisonnement local. Toutefois avant d'introduire cette notion, il est nécessaire de rappeler quelques notations dans le cadre SCSP.

Soit un SCSP $\mathcal{P} = (X, Y, D, P, C, \theta)$, nous notons $\mathcal{P}_{c_j} = (X, Y, D, P, c_j, \theta)$ le sous-problème SCSP où seule la contrainte $c_j \in C$ est considérée. Rappelons que $\tau[x_i]$ désigne la valeur de x_i dans le tuple τ.

Définition 2.34 (arc-cohérence (AC)) *Soit un SCSP $\mathcal{P} = (X, Y, D, P, C, \theta)$, x_d une variable de décision de V et x_s une variable stochastique de Y. Une valeur $v \in dom(x_d)$ est dite arc-cohérente si et seulement si pour chaque variable $c_j \in C$, v est cohérente dans $\mathcal{P}_{c_j}$. Une valeur $v \in dom(x_s)$ est dite arc-cohérente si et seulement si, pour chaque contrainte $c_j \in C$, il existe un tuple valide τ autorisé par $rel(c_j)$. $\mathcal{P}$ est dit arc-cohérent si et seulement si toutes les valeurs de toutes les variables de X sont arc-cohérentes.*

Notons que la définition de l'arc-cohérence pour une valeur dépend du type de la variable à laquelle elle appartient. En effet, la définition de l'arc-cohérence pour une valeur assignée à une variable stochastique est identique celle du cadre CSP alors que la définition de l'arc-cohérence pour une valeur assignée à une variable de décision généralise celle du cadre CSP. On comprend alors que décider si un problème SCSP donné est arc-cohérent est bien plus difficile qu'un CSP.

[BALAFOUTIS & STERGIOU 2006] introduit un algorithme incomplet nommé *Stochastic_AC* permettant d'appliquer en partie l'arc-cohérence sur un réseau stochastique. De plus, cette méthode rencontre de mauvaise performance en étant maintenue via l'algorithme MAC [SABIN & FREUDER 1994]

Algorithme 2.17 : *check*

Données : Assignation $x_i = v_j$, Réel q_i, Seuil bas θ_{bas}, Satisfaction s
Résultat : Booléen

```
1  q_i ← q_i - P(x_i = v_j)
2  pour chaque k ∈ [i+1; n] faire
3      dv ← vrai
4      pour chaque v_l ∈ dom(x_k) faire
5          si prune(k, l) = 0 alors
6              si incohérent(x_i = v_j, x_k = v_l) alors
7                  prune(k, l) ← i
8                  si x_k ∈ S alors
9                      ms ← ∏_{s=i+1}^{n} Σ_{t=1}^{|dom(x_s)|} P(x_s = v_t)
10                     si x_i ∈ S alors
11                         si (ms × P(x_i = v_j) + s + q_i) < θ_bas alors
12                             retourner Faux
13                         fin
14                     fin
15                     sinon
16                         si ms < θ_bas alors
17                             retourner Faux
18                         fin
19                     fin
20                 fin
21             fin
22             sinon
23                 dv ← faux
24             fin
25         fin
26     fin
27     si dv = vrai alors
28         retourner Faux
29     fin
30 fin
31 retourner Vrai
```

Algorithme 2.18 : $restore$

```
Données : Entier i
pour chaque j ∈ [i + 1; n] faire
    pour chaque v_k ∈ dom(x) faire
        si prune(j, k) = i alors
            prune(j, k) = 0
            si x_j ∈ S alors
                q_j ← q_j + P(x_j = d_k)
            fin
        fin
    fin
fin
```

face à SFC et ne peut donc être utilisée qu'en phase de pré-traitement où même ici l'aide qu'il apporte à la résolution n'est pas significative. Pour plus d'informations quant à son implémentation, le lecteur pourra se référer à [BALAFOUTIS & STERGIOU 2006].

[TARIM *et al.* 2006] propose une approche étendue des SCSP nommée *Stochastic OPL* basée sur les scénarios où une distinction est réalisée entre deux types de contraintes : les contraintes de chance et les contraintes dures.

Définition 2.35 (Contrainte de chance (*chance constraint*)) *Soit un SCSP* $\mathcal{P} = (X, Y, D, P, C, \theta)$ *et une contrainte* $c \in C$. *c est dit contrainte de chance si* $\exists x_s \in scp(c) \mid x_s \in Y$.

Dit autrement, une contrainte c_h de chance est une contrainte qui porte sur au moins une variable stochastique. θ_h est un seuil de probabilités associé à la contrainte c_h tel qu'il existe une fraction de scénarios supérieure à θ_h où c_h n'est pas violée.

Définition 2.36 (Contrainte dure (*hard constraint*)) *Soit un SCSP* $\mathcal{P} = (X, Y, D, P, C, \theta)$ *et une contrainte* $c \in C$. *c est dit contrainte dure si* $\exists! x_s \in scp(c) \mid x_s \in Y$.

Notons qu'une contrainte de chance où $c_h = 1$ est équivalent à une contrainte dure.

Cette approche basée sur les scénarios autorise de multiples contraintes de chance c_h dans un même SCSP et θ peut être divisé en un ensemble de seuils de probabilités θ_h. Un SCSP classique est équivalent à un SCSP basé sur les scénarios si l'ensemble des seuils de probabilités θ_h associé à chaque contrainte est identique.

Cette approche est basée sur la génération d'un arbre de scénarios qui intègre toutes les assignations possibles des variables stochastiques et où chaque niveau de l'arbre correspond à un niveau k du m-SCSP correspondant au problème. L'objectif étant de résoudre chaque scénario comme un CSP où les variables stochastiques sont remplacées par des constantes correspondant aux valeurs prises par les variables stochastiques dans le scénario. Les seules variables restantes étant les variables de décisions, il est possible de le résoudre comme un réseau de contraintes classiques bien que certaines variables de décisions appartiennent à plusieurs niveaux de l'arbre de scénarios généré. Son intérêt est de ré-utiliser l'ensemble des solveurs existants de CSP qui permettent de combiner la résolution des problèmes de satisfaction de réseaux de contraintes à l'optimisation linéaire stochastique en nombres entiers ([SHAPIRO *et al.* 2014] et [L. *et al.* 1998]). Dit autrement, cette approche est une extension du langage d'optimisation du modèle de contraintes (OPL) [VAN HENTENRYCK *et al.* 1999].

Toutefois, le nombre de scénarios grandit exponentiellement selon le nombre de niveaux composant le m-SCSP à résoudre et pour que cette approche soit utilisable en pratique, il est nécessaire d'utiliser une technique de réduction du nombre de scénarios [DUPACOVÁ *et al.* 2000].

Cette technique détermine un sous-ensemble de scénarios bien plus petit que celui d'origine et effectue une redistribution de la distribution de probabilités P du problème d'origine sur ce sous-ensemble. Malheureusement, l'utilisation de cette technique ne préserve pas complètement l'ensemble des politiques solutions d'un SCSP. Ainsi, l'approche *Stochastic OPL*, bien qu'elle soit bien plus efficace que SFC, est incomplète en pratique.

2.4 Conclusion

Le problème de satisfaction de réseaux de contraintes fait l'objet de nombreuses études depuis de nombreuses années. Des avancées significatives dans la résolution pratique de ce problème ont ainsi pu être observées, à la fois dans un réseaux de contraintes classiques et plus récemment pour un réseau de contraintes stochastiques. Dans la deuxième partie de ce manuscrit, nous proposons un modèle SCSP pour le contexte GGP et un programme-joueur basé sur ce modèle.

Deuxième partie

Contributions

Chapitre 3

Modélisation par contraintes de jeux GDL

Sommaire

3.1 Équivalence entre modèles 106
3.1.1 *Ground* instance GDL propositionnel 106
3.1.2 P-GDL $\Longleftrightarrow$ SCSP 107
3.2 De GDL à SCSP : Exemple 108
3.2.1 Extraction des variables 109
3.2.2 Extraction des domaines et des distributions de probabilités 109
3.2.3 Extraction des contraintes 109
3.3 Extension du modèle aux jeux à information partagée 112
3.3.1 Fragment PSPACE de GDL-II 112
3.3.2 De GDL-II à SCSP 114
3.4 Expérimentations 115
3.4.1 Jeux GDL 115
3.4.2 Jeux GDL-II 117
3.5 Modélisation des stratégies 122
3.6 Conclusion 123

Par son approche déclarative de la résolution de problèmes combinatoires, la programmation par contraintes offre un cadre prometteur pour relever le défi imposé par le *General Game Playing*. Au cours de ces dernières années certains formalismes adaptés à la représentation et à la résolution de jeux ont fait leurs apparitions parmi lesquels on peut citer les QCSP [BÖRNER *et al.* 2003], les *strategic* CSP [BESSIERE & VERGER 2006] ou encore les *constraint games* [NGUYEN & LALLOUET 2014]. Cependant, la plupart de ces formalismes sont restreints aux jeux à information complète et certaine impliquent qu'à chaque instant les joueurs aient connaissance de l'état du jeu et que les actions de chaque joueur soient déterministes. Suite à ce constat, ces modèles ne sont pas adaptés au cadre GGP.

L'objectif de ce chapitre est d'aborder le problème GGP d'un point de vue original, fondé sur les réseaux de contraintes stochastiques (SCSP). En effet, les SCSP représentent un outil puissant permettant non seulement de modéliser et de résoudre des jeux de stratégie mais également de modéliser l'intervention du hasard.

De plus, dans [WALSH 2009], les problèmes de satisfactions de réseaux de contraintes stochastiques (SCSP) sont montrés étroitement liés aux problèmes de décision Markovien (MDP pour *Markov Decision Problem*). En effet, comme pour un SCSP, la solution d'un MDP est une politique (souvent nommée

politique stochastique). Dans un MDP, une politique spécifie la meilleure action à prendre dans chaque état et la relie intrinsèquement à une politique solution d'un SCSP. Les MDP sont couramment utilisés pour décrire les casses-têtes (jeu à 1 joueur) et les jeux de Markov détaillés en section 1.1.3 représentent une extension des problèmes de décision Markovien englobant les jeux multi-joueurs. Comme décrit en section 1.2.1, un programme GDL peut se formaliser par un jeu de Markov justifiant de la proximité des deux modèles.

Dans un premier temps en section 1, nous montrons l'équivalence entre un programme GDL propositionnel et un réseau de contraintes stochastiques. Ensuite dans la section suivante, nous détaillons sur un exemple la traduction d'un programme GDL en SCSP. En section 3, nous montrons que notre traduction vers un modèle SCSP peut être étendue aux jeux à information partagée décrites par un fragment de GDL-II montré dans PSPACE. La section 4 valide expérimentalement notre processus de traduction sur tout un panel de jeux GDL et GDL-II montrant la compatibilité temporelle d'une telle traduction dans le cadre de compétitions GGP. Finalement, la section suivante envisage la modélisation de stratégies dans le but de « résoudre » un jeu sous la forme d'un SCSP avant de conclure ce chapitre.

Les différents travaux présentés dans ce chapitre ont fait l'objet de plusieurs publications : [KORICHE *et al.* 2014, KORICHE *et al.* 2016a, KORICHE *et al.* 2015a, KORICHE *et al.* 2016b].

3.1 Équivalence entre modèles

La traduction présentée dans cette section de GDL à SCSP est envisagée pour les jeux à information imparfaite où un joueur environnement est décrit par le mot-clé $random$ introduit en GDL-II. La sémantique d'un jeu GDL + *random* est détaillée dans la définition 1.32.

3.1.1 *Ground* instance GDL propositionnel

Comme GDL est connu pour être Turing complet (même pour les jeux GDL valides) [SAFFIDINE 2014], nous nous intéressons ici uniquement aux programmes GDL propositionnels après avoir appliqué une phase de *grounding* (voir définition 1.38), noté P-GDL. En effet, de tels jeux possèdent deux propriétés additionnelles :

1. le nombre de *ground* instances de fonctions de termes et la taille des clauses sont bornés ;
2. Le graphe de dépendance associé au programme GDL est acyclique.

Dans ce fragment, tout programme P-GDL G peut être réécrit en une *ground* instance G_g équivalente de taille polynomiale $|G|$. Il est donc possible d'appliquer une phase de *grounding* à toutes les instances inclues dans ce fragment. Les *ground* termes de G_g peuvent être partitionnés en un ensemble $J^* = \{1, 2, \ldots, k, k+1\}$ de joueurs (où $k+1$ représente le joueur environnement), un ensemble F de fluents, un ensemble A d'actions et un ensemble U de constantes d'utilité.

De même, les règles de $\mathtt{G}_g$ peuvent être partitionnées en six catégories :

$$\begin{aligned}
&\mathtt{role}(j)\\
&\mathtt{init}(f)\\
&\mathtt{terminal} \ \mathtt{:-} \ \mathtt{L}_1, \cdots, \mathtt{L}_q\\
&\mathtt{legal}(i,a) \ \mathtt{:-} \ \mathtt{L}_1, \cdots, \mathtt{L}_{q'}\\
&\mathtt{next}(f) \ \mathtt{:-} \ \mathtt{does}(1,a_1), \cdots, \mathtt{does}(k+1,a_{k+1})\\
&\mathtt{goal}(i,\nu) \ \mathtt{:-} \ \mathtt{L}_1, \cdots, \mathtt{L}_{q''}
\end{aligned}$$

où $j \in J^*$, $\nu \in U$, $f \in F$ et $a_1, \ldots, a_{k+1} \in A$. De plus, chaque $\mathtt{L}_j$ est un atome $\mathtt{A}_j$ ou sa négation $\mathtt{not}\ \mathtt{A}_j$, où $\mathtt{A}_j$ est une expression de la forme $\mathtt{true}(f)$ ou $f_1 \neq f_2$.

La complétion de *Clark* [CLARK 1978] de $\mathtt{G}_g$ est l'ensemble des formules propositionnelles $\mathtt{G}_c$, où
— chaque symbole `not` dans $\mathtt{G}_g$ est remplacé par $\neg$;
— chaque atome `A` en tête de règle est associé avec son ensemble $\mathtt{B}_\mathtt{A}$ de corps de règles, c'est à dire

$$\mathtt{B}_\mathtt{A} = \{\mathtt{L}_1 \wedge \ldots \wedge \mathtt{L}_q \mid \mathtt{A} \ \mathtt{:-} \ \mathtt{L}_1, \ldots, \mathtt{L}_q \in \mathtt{G}_g\}$$

— pour chaque atome `A`, toutes les règles d'en-tête `A` sont remplacées par une unique formule $\mathtt{A} \leftrightarrow \bigvee \mathtt{B}_\mathtt{A}$.

Pour un atome `A`, nous utilisons $\mathtt{G}_g \vdash \mathtt{A}$ pour désigner que `A` est à l'intersection de tous les ensembles résultants de $\mathtt{G}_g$, et $\mathtt{G}_c \models \mathtt{A}$ pour indiquer que $\mathtt{G}_c \wedge \neg\mathtt{A}$ est insatisfaisable. Ainsi selon le corollaire 4.21 de [BEN-ELIYAHU & DECHTER 1994], si $\mathtt{G}_g$ est associé à un graphe de dépendance acyclique, nous avons $\mathtt{G}_g \vdash \mathtt{A}$ si et seulement si $\mathtt{G}_c \models \mathtt{A}$, c'est à dire que, $\mathtt{G}_g$ et $\mathtt{G}_c$ sont logiquement équivalent.

Dans la partie suivante, nous montrons l'équivalence entre un `SCSP` à T niveaux est le programme propositionnel `P-GDL` correspondant.

3.1.2 P-GDL $\Longleftrightarrow$ SCSP

Soit G_g le *ground* programme du jeu `P-GDL` G et G_c la complétion de Clark de G_g. Notons $P_{G,T} = (X, Y, D, C, P, \theta)$ le `SCSP` à T niveaux obtenu à partir de la traduction du programme `P-GDL` G. À partir d'un *ground* `P-GDL` les composants de $P_{G,T}$ sont décrits comme suit :
— $X = (V_1, Y_1, \ldots, V_T, Y_T)$ où V_t inclue un ensemble de variables booléennes $\{f_{i,t}\}_{i=1}^{n}$ chacune associée à un fluent en particulier de F, une variable booléenne ω_t identifiant si un état est terminal ou non, un ensemble de variables $\{a_{j,t}\}_{j=1}^{k}$ de domaine A chacune associée à un joueur j et un ensemble de variables $\{u_{j,t}\}_{j=1}^{k}$ de domaine U, chacune associée à un joueur j. Chaque Y_t inclue une unique variable $a_{k+1,t}$ de domaine A associée au joueur environnement ;
— $C = C_1 \cup \ldots \cup C_T$ tel que C_1 inclue un ensemble de contraintes modélisant les clauses unaires sur *init* dans G_c et pour chaque t chaque règle $A \leftrightarrow \bigvee \mathtt{B}_\mathtt{A}$ dans G_c est assignée à une contrainte $c_{A,t}$ de C_t modélisant cette règle. Les contraintes sur *legal* pour le même joueur j sont fusionnées en une unique contrainte $\mathit{legal}_{j,t}$;
— $P = \{P_1, \ldots P_t\}$ avec pour chaque t, la portée de P_t est la portée $\{f_{i,t}\}_{i=1}^{m}$ de $\mathit{legal}_{k+1,t}\,(a_{k+1,t})$; et pour la portée de chaque instanciation $\boldsymbol{v}$, $P_t(\cdot \mid \boldsymbol{v})$ est la distribution uniforme sur $\{a \in A \mid \mathtt{legal}_{k+1,t}(a, \boldsymbol{v}) = 1\}$.
— $\theta = 1$.

Pour un niveau t, soit ϑ_t la fonction encodant tout état $s \in 2^F$ tel que $\vartheta_t(s) = \{(f_{i,t}, 1) \mid f_i \in s\} \cup \{(f_{i,t}, k+1) \mid f_i \notin s\}$, et tout vecteur d'actions $\boldsymbol{a} \in A^k$ dans $\vartheta_t(\boldsymbol{a}) = \{(p_{j,t}, a_j)\}_{j=1}^{k}$. Nous désignons le jeu stochastique d'horizon T $(s_0, S_{ter}, L, P, \boldsymbol{u})$ par $\mathcal{G}(\mathcal{P}_{\mathtt{G},T})$ tel que

$f_j \in s_0$ ssi $\{(f_{i,0}, 0)\}$ globalement cohérent;
$s \in S_{ter}$ ssi $\exists t$ tel que $\vartheta_t(s) \cup \{(\omega_t, 1)\}$ est globalement cohérent;
$a \in L_j(s)$ ssi $\vartheta_0(s)$ est globalement cohérent implique $\vartheta_0(s) \cup \{(a_{j,0}, a)\}$ est globalement cohérent;
$u_j(s) = \nu$ ssi $\exists t$ tel que $\vartheta_t(s) \cup \{(u_{j,t}, \nu)\}$ est globalement cohérent;
$P(s' \mid s, \boldsymbol{a}) = \frac{1}{L_{k+1}(s)}$ si $\exists a_{k+1} \in A$ tel que
$\vartheta_0(s) \cup \{(a_{k+1,0}, a_{k+1})\} \cup \vartheta_0(\boldsymbol{a}) \cup \vartheta_1(s')$ est globalement cohérent, et $P(s' \mid s, \boldsymbol{a}) = 0$ sinon.

Proposition 3.1 *Soit le programme `P-GDL` G et soit T un entier positive tel que $\mathcal{G}(\mathtt{G})$ est un jeu à horizon T. Alors $\mathcal{G}(\mathtt{G}) = \mathcal{G}(\mathcal{P}_{\mathtt{G},T})$.*

Preuve *Soit $\mathcal{G}(\mathtt{G}) = (s_0, S_{ter}, L, P, \boldsymbol{u})$ le jeu stochastique de $\mathtt{G}$ et supposons que $\mathcal{G}(\mathtt{G})$ est un jeu d'horizon T. Soit $\mathcal{T} : 2^F \times A^k \to 2^F$ l'association donnée par $s' \in \mathcal{T}(s, \boldsymbol{a})$ si et seulement si $(s, \boldsymbol{a}, s')$ est inclue dans certaines séquences de jeu de $\mathcal{G}(\mathtt{G})$. En nous basant sur $\mathcal{G}(\mathtt{G})$, nous savons que*

$$
\begin{aligned}
f_i \in s_0 \textit{ ssi } \mathtt{G}_g \vdash \mathtt{init}(f_i),\\
s \in S_{ter} \textit{ ssi } \mathtt{G}_g \cup \mathtt{true}(s) \vdash \mathtt{terminal},\\
a \in L_j(s) \textit{ ssi } \mathtt{G}_g \cup \mathtt{true}(s) \vdash \mathtt{legal}(j, a),\\
u_j(s) = \nu \textit{ ssi } \mathtt{G}_g \cup \mathtt{true}(s) \vdash \mathtt{goal}(j, \nu),\\
s' \in \mathcal{T}(s, \boldsymbol{a}) \textit{ ssi } \exists\, a_{k+1} \in A \textit{ tel que } \mathtt{G}_g \cup \mathtt{true}(s) \cup \mathtt{does}(a_{k+1}, \boldsymbol{a}) \vdash \mathtt{true}(s')
\end{aligned}
$$

avec $\mathtt{does}(a_{k+1}, \boldsymbol{a})$ l'ensemble des ground termes $\{\mathtt{does}(j, a)\}_{j=1}^{k+1}$. Clairement, $\mathcal{G}(\mathtt{G}) = \mathcal{G}(\mathtt{G}_g)$. De plus, si le graphe de dépendance de $\mathtt{G}$ est acyclique, toutes les équivalences ci-dessus restent valables suite à la complétion de Clark $\mathtt{G}_c$ en remplaçant simplement l'opérateur $\vdash$ par l'opérateur $\models$ et chaque ensemble $\mathtt{true}(s)$ par $\mathtt{lit}(s) = \mathtt{true}(s) \cup \{\neg f \mid f \notin s\}$. Par conséquent, $\mathcal{G}(\mathtt{G}_g) = \mathcal{G}(\mathtt{G}_c)$.

Ainsi, en utilisant la construction de $\mathcal{P}_{\mathtt{G},T}$, nous savons que $\mathtt{G}_c \models \mathtt{init}(f_i)$ si et seulement si $\{(f_i, 0)\}$ est globalement cohérent, dès que $\mathcal{P}_{\mathtt{G},T}$ inclue toutes les contraintes modélisant les clauses unaires $\mathtt{init}$ dans $\mathtt{G}_c$.

De plus, suite au fait que la contrainte $\mathtt{terminal}_t$ modélise la règle $\mathtt{terminal} \leftrightarrow \mathtt{B}_{\mathtt{terminal}}$, nous savons que $\mathtt{G}_c \cup \mathtt{lit}(s) \models \mathtt{terminal}$ si et seulement si $\vartheta_t(s) \cup \{(\omega_t, 1)\}$ est globalement cohérent pour t car $\mathcal{G}(\mathtt{G}_c)$ est un jeu à horizon T. Les autres cas $\mathtt{legal}(j, a)$ et $\mathtt{goal}(j, \nu)$ sont traités similairement. Pour le dernier cas, sachant que la contrainte $\mathtt{next}_t$ modélise les règles $\mathtt{next}(f_i) \leftrightarrow \mathtt{B}_{\mathtt{next}(f_i)}$, nous savons que $\mathtt{G}_c \cup \mathtt{lit}(s) \cup \mathtt{does}(\boldsymbol{a}, a_{k+1}) \models \mathtt{true}(s')$ si et seulement si $\vartheta_t(s) \cup \vartheta_t(\boldsymbol{a}) \cup \{(p_{k+1,t}, a_{k+1})\} \cup \vartheta_{t+1}(s')$ est globalement cohérent.

Finalement, comme les actions légales de $k+1$ dans $\mathcal{G}(\mathtt{G}_c)$ et $\mathcal{G}(\mathcal{P}_{\mathtt{G},T})$ sont égales, il s'en suit que la fonction de transition probabiliste dans $\mathcal{G}(\mathtt{G}_c)$ et $\mathcal{G}(\mathcal{P}_{\mathtt{G},T})$ sont égales. Par conséquent, $\mathcal{G}(\mathtt{G}_c) = \mathcal{G}(\mathcal{P}_{\mathtt{G},T})$, comme voulu. □

Dit autrement, si $\mathtt{G}$ est un programme P-GDL, alors $\mathtt{G}$ et son SCSP à T niveaux modélisant $\mathcal{P}_{\mathtt{G},T}$ décrivent le même jeu stochastique, suite au fait que tout état terminal du jeu peut être atteint après au plus T actions.

Remarque 3.1 *Dans la suite de ce manuscrit, tout jeu GDL appartient au fragment proposition P-GDL.*

Au cours de la partie suivante, nous exposons les différentes étapes nécessaires pour réaliser la traduction d'un programme GDL en SCSP.

3.2 De GDL à SCSP : Exemple

Pour rappel, un jeu GDL possède un horizon T si chaque état terminal qui le compose est atteignable après au plus T tours, ainsi les pas $t \in \{0, \ldots, T\}$ capturent les tours de jeu. On note GDL_T l'ensemble de tous les programmes GDL dont le jeu de Markov associé est acyclique, à profondeur au plus T.

La traduction proposée permet d'obtenir un modèle SCSP défini par le 6-tuple (X, Y, D, P, C, θ) à partir d'un programme $\mathtt{GDL}_T$ G, en construisant les ensembles de variables X, de domaines D et de contraintes C, ainsi que les distributions de probabilités P correspondantes. Nous illustrons cette traduction sur le jeu GDL du *Matching Pennies* illustré dans l'exemple 1.1 en associant un $\mu\mathtt{SCSP}_t$ à chaque temps t en trois étapes que nous allons examiner.

3.2.1 Extraction des variables

À chaque μSCSP$_t$, l'ensemble des variables qui le compose est décrit sous la forme $X = (\{F_t\}, \omega_t, U_t, \{A_t\}, A_{k+1,t}, \{F_{t+1}\})$ où chaque composant est décrit dans ce qui suit.

Tout d'abord, ω_t représenté par une variable booléenne dénommée $terminal_t$ indique si le jeu a atteint un état terminal à l'instant courant t. Suite à cela, U_t représente l'ensemble des valeurs de scores possibles pour chaque joueur, on associe une variable $score_{k,t}$ pour chaque joueur k du programme GDL.

$\{F_t\}$ et $\{F_{t+1}\}$ définissent l'ensemble des fluents décrivant le jeu au tour t et $t+1$. Ainsi à chaque prédicat de la forme $F(X_1, \ldots, X_k, Z)$, qui n'est pas un mot-clé de GDL on associe une variable $f_t \in F_t$ et une variable $f_{t+1} \in F_{t+1}$.

$\{A_{1,t}, \ldots, A_{k,t}\}$ est l'ensemble des variables de décisions décrivant les actions possibles de chaque joueur k et $A_{k+1,t}$ est la variable stochastique représentant les actions possibles de l'environnement $k+1$ si celui-ci est impliqué dans le jeu GDL. On représente ces variables par des variables du type $action_{k,t}$.

Pour finir, comme le nombre de fluents peut varier à chaque tour de jeu, il est nécessaire de définir leur nombre en fonction du temps t. Ainsi, si t = 0, une variable correspondante à chaque fluent induit par les règles *init* est générée sinon si $t \neq 0$, nous générons une variable pour chaque fluent résultant de chaque $f_{t+1} \in F_{t+1}$.

Exemple 3.1 *Reprenons le jeu du Matching Pennies dont le programme GDL correspondant est illustré dans la figure 1.4. Les variables correspondantes au programme GDL à tout temps t sont ordonnées dans X comme suit : $piece_t$, $terminal_t$, $score_{j1,t}$, $score_{j2,t}$,$action_{j1,t}$, $action_{j2,t}$, $piece_{t+1}$.*

Ici, le jeu se décompose en deux temps. À l'instant $t = 0$ et suite aux règles init, deux variables $piece_{1,0}$ et $piece_{2,0}$ sont générées. De même à l'instant suivant, deux nouvelles variables sont générées : $piece_{1,1}$ et $piece_{2,1}$.

Suite à l'extraction des variables, vient l'extraction des domaines et des distributions de probabilités associées aux variables.

3.2.2 Extraction des domaines et des distributions de probabilités

Concernant la variable $terminal_t$, le domaine correspondant est intuitivement lié à la génération de la variable. À savoir, le domaine de la variable $terminal_t$ est booléen.

Toutefois le domaine de chaque variable fluent f est donné par l'ensemble des combinaisons des termes $o_1, \ldots, o_n$ pouvant instancier chaque atome d'arité n de la forme $F(o_1, \ldots, o_n)$. L'extraction des domaines de variables d'action s'effectue de manière analogue, en identifiant au préalable, à partir de la relation $legal(j, A)$, les atomes A qui participent à la construction du domaine $action_{j,t}$. De même pour le domaine de chaque variable $score_{j,t}$ il est déduit de tout terme U pouvant apparaître dans chaque prédicat $goal(J, U)$.

Exemple 3.2 *Toujours sur le Matching Pennies présenté en figure 1.4. Suite aux phases d'extraction décrites ci-dessus, les variables et les domaines du SCSP obtenu sont donnés dans le tableau 3.1.*

Concernant les jeux impliquant un environnement, la distribution de probabilités de $A_{k+1,t}$ est uniforme sur le domaine de la variable à chaque instant t.

Finalement, la dernière étape consiste à extraire les contraintes du problème SCSP correspondant.

3.2.3 Extraction des contraintes

Pour chaque règle R du programme GDL$_T$ et à chaque instant t, nous associons une contrainte en extension $C_{R,t}$. Pour commencer afin de définir la portée de chaque contrainte, nous spécifions une règle

Variables	Domaines
$piece_t$	$\{j1\_pile, j1\_face, j1\_inconnu, j2\_pile, j2\_face, j2\_inconnu\}$
$terminal_t$	$\{vrai, faux\}$
$score_{j1,t}$	$\{0, 100\}$
$score_{j2,t}$	$\{0, 100\}$
$action_{j1,t}$	$\{retourne\_pile, retourne\_face\}$
$action_{j2,t}$	$\{retourne\_pile, retourne\_face\}$
$piece_{t+1}$	$\{j1\_pile, j1\_face, j1\_inconnu, j2\_pile, j2\_face, j2\_inconnu\}$

TABLE 3.1 – L'ensemble des variables et leurs domaines extraits du matching pennies en GDL

de réécriture en fonction de chaque prédicat clé utilisé dans chacune des règles. La table 3.2 indique ces règles de réécriture.

Prédicat GDL	Portée de la contrainte
$init(f(\ldots))$	$\{f_0\} \in scp(C_0)$
$true(f(\ldots))$	$\{f_t\} \in scp(C_t)$
$does(j, a(\ldots))$	$\{action_{j,t}\} \in scp(C_t)$
$legal(j, a(\ldots))$	$\{action_{j,t}\} \in scp(C_t)$
$next(f(\ldots))$	$\{f_{t+1}\} \in scp(C_t)$
$goal(j, N)$	$\{score_{j,t}\} \in scp(C_t)$
$terminal$	$\{terminal_t\} \in scp(C_t)$

TABLE 3.2 – L'ensemble des variables et leurs domaines extraits du matching pennies en GDL

Notons que pour toute règle *goal*, nous ajoutons la variable $terminal_t$ à la portée de la contrainte correspondante afin de garantir qu'un joueur ne puisse obtenir un score différent de 0 que dans le cas où $terminal_t$ est égale à $vrai$.

La seconde étape est de définir la relation de chaque contrainte. Chaque combinaison d'atomes de chaque *ground* règle correspond à la relation de chaque contrainte. Afin de diminuer le nombre de contraintes, nous fusionnons toutes les contraintes de même portée.

Exemple 3.3 *Reprenons de nouveau le jeu du Matching Pennies illustré en GDL en figure 1.4. La phase d'extraction illustrée ici permet d'obtenir les contraintes du SCSP correspondant. Les contraintes des deux temps ($t = 0$ et $t = 1$) sont données dans le tableau 3.3 où les notations suivantes sont utilisées afin de simplifier l'écriture :*

- P = *pile ;*
- F = *face ;*
- I = *inconnu ;*
- V = *vrai ;*
- W = *faux ;*
- ter = *terminal ;*
- $*$ = *toutes valeurs possibles.*

Au cours de la section suivante, nous montrons qu'il est possible d'étendre notre traduction vers un

Contraintes : {**Portée**}	**Relation**
Contraintes *init*	
$c_1 : \{piece_{1,0}\}$	$\{j1\_I\}$
$c_2 : \{piece_{2,0}\}$	$\{j2\_I\}$
Contraintes *legal*	
$c_3 : \{action_{j1,0}, piece_{1,0}\}$	$\{(retourne\_P, j1\_I), (retourne\_F), j1\_I)\}$
$c_4 : \{action_{j2,0}, piece_{2,0}\}$	$\{(retourne\_P, j2\_I), (retourne\_F, j2\_I)\}$
Contraintes *next*	
$c_5 : \{piece_{1,1}, action_{j1,0}\}$	$\{(j1\_P, retourne\_P), (j1\_F, retourne\_F)\}$
$c_6 : \{piece_{2,1}, action_{j2,0}\}$	$\{(j2\_P, retourne\_P), (j2\_F, retourne\_F)\}$
Contraintes *terminal*	
$c_7 : \{ter_0, piece_{1,0}, piece_{2,0}\}$	$\{(V, j1\_P, j2\_P), (V, j1\_P, j2\_F), (V, j1\_F), j2\_P)$ $, (V, j1\_F, j2\_F), (W, j1\_I, *), (W, *, j2\_I)\}$
$c_8 : \{ter_1, piece_{1,1}, piece_{2,1}\}$	$\{(V, j1\_P, j2\_P), (V, j1\_P, j2\_F), (V, j1\_F, j2\_P)$ $, (V, j1\_F, j2\_F), (W, j1\_I, *), (W, *, j2\_I)\}$
Contraintes *goal*	
$c_9 : \{score_{j1,0}, piece_{1,0}, piece_{2,0}, ter_0\}$	$\{(100, j1\_P, j2\_P, V), (100, j1\_F, j2\_F, V)$ $(0, *, *, W), (0, j1\_F, j2\_P, V), (0, j1\_P, j2\_F, V)$ $, (0, j1\_I, *, V), (0, *, j2\_I, V)\}$
$c_{10} : \{score_{j2,0}, piece_{1,0}, piece_{2,0}, ter_0\}$	$\{(100, j1\_F, j2\_P, V), (100, j1\_P, j2\_F, V)$ $(0, *, *, W), (0, j1\_P, j2\_P, V), (0, j1\_F, j2\_F, V)$ $, (0, j1\_I, *, V), (0, *, j2\_I, V)\}$
$c_{11} : \{score_{j1,1}, piece_{1,1}, piece_{2,1}, ter_1\}$	$\{(100, j1\_P, j2\_P, V), (100, j1\_F, j2\_F, V)$ $(0, *, *, W), (0, j1\_F, j2\_P, V), (0, j1\_P, j2\_F, V)$ $, (0, j1\_I, *, V), (0, *, j2\_I, V)\}$
$c_{12} : \{score_{j2,1}, piece_{1,1}, piece_{2,1}, ter_1\}$	$\{(100, j1\_F, j2\_P, V), (100, j1\_P, j2\_F, V)$ $(0, *, *, W), (0, j1\_P, j2\_P, V), (0, j1\_F, j2\_F, V)$ $, (0, j1\_I, *, V), (0, *, j2\_I, V)\}$

TABLE 3.3 – L'ensemble des contraintes extraites du *matching pennies* en GDL

problème SCSP à un fragment plus important du langage GDL-II.

3.3 Extension du modèle aux jeux à information partagée

Comme vu dans la section 1.2.3, le *game description language with incomplete information*, abrégé GDL-II est assez expressif pour représenter les jeux stochastiques multi-agents avec observations partielles. Malheureusement, une telle expressivité n'est pas possible sans un prix : le problème consistant à trouver une stratégie gagnante est $NExp^{NP}$-DIFFICILE, une classe de complexité qui est bien au-delà de la portée des *solvers* actuels. C'est pourquoi dans la partie suivante, nous identifions un fragment de GDL-II PSPACE-COMPLET englobant les jeux où les agents partagent les mêmes observations (partielles) dénommés jeux à information partagée.

3.3.1 Fragment PSPACE de GDL-II

Un **jeu à information partagée** représente tout jeu stochastique à information partielle G dans lequel à chaque tour tous les joueurs partagent le même état de croyance, c'est à dire $B_1(s) = \dots B_k(s)$ pour tout état $s \in S$. Afin de simplifier l'écriture, $B(s)$ est utilisé pour désigner l'état de croyance commun dans s. Clairement tout jeu à information partagée peut être converti dans un jeu stochastique complètement observable, en remplaçant la fonction de transition P et la fonction de croyance B par une nouvelle fonction de croyance $Q : S \times A^k \hookrightarrow \Delta_s$ pour tout $\boldsymbol{a} \in L(s)$ défini ainsi :

$$Q(s, \boldsymbol{a})(s') = \prod_{t \in S} P(s, \boldsymbol{a})(t) \cdot B(t)$$

En d'autres mots, $Q(s, \boldsymbol{a})(s')$ est la probabilité d'observer s' après avoir effectué la séquence d'actions $\boldsymbol{a}$ dans l'état s. Étant donné que pour tout jeu G stochastique à information partagée, une politique jointe de G est une fonction $\pi : S \to A^k$ où $pi_j(s)$ représente la politique du joueur j, et $\pi{-}j(s)$ est la politique jointe des autres joueurs. Soit un vecteur de seuil $\boldsymbol{\theta} \in [0, 1]^k$, nous désignons par π une politique gagnante pour le joueur j si le score attendu de j dans π est plus grand que θ_j.

Basé sur la notion d'information partagée, nous pouvons maintenant examiner quelques restrictions des programmes GDL-II garantissant ensemble que la recherche d'une politique gagnante est dans PSPACE. Rappelons, qu'un programme GDL-II G possède un nombre de tours borné si le nombre de termes de l'univers de *Herbrand* de G est polynomial sur $|G|$. Par conséquent, si G est borné et que chaque règle de G possède un nombre constant de variables, alors G est propositionnel.

Pour finir, G est à information partagée si pour chaque joueur j, chaque fluent f, chaque état s et chaque séquence d'actions $\boldsymbol{a}$, si on a $G \cup s^{true} \cup \boldsymbol{a}^{does} \models sees(j, f)$ alors $G \cup s^{true} \cup \boldsymbol{a}^{does} \models sees(q, f)$ pour tout joueur $q \in [k]$.

Théorème 3.1 *Soit $G_T \subseteq$ GDL-II le fragment propositionnel des jeux à information partagée d'horizon T. Alors la recherche d'une politique gagnante dans G_T est* PSPACE-COMPLET.

Preuve *Sachant que QBF sur CNF, noté QBF[CNF] est* PSPACE-DIFFICILE *et que la classe* PSPACE *est fermée par complémentarité, nous savons que QBF sur DNF, noté QBF[DNF] est* PSPACE-DIFFICILE. *Montrons qu'il est possible d'encoder une formule QBF[DNF] en jeu GDL propositionnel. Soit une formule QBF[DNF] du type :*

$$\exists x_1 \forall y_2 ... \exists x_{n-1} \forall y_n T_1 \vee T_2 \vee ... \vee T_m$$

où T_i représente une DNF par une conjonction de littéraux sur l'ensemble $Z = \{x_1, \dots, x_{n-1}\} \cup \{y_2, \dots, y_n\}$ et n est pair. L'ensemble des fluents F est alors égale à $Z \cup \{\overline{z} \mid z \in Z\}$ représentant

l'ensemble des littéraux propositionnels sur Z. Ajoutons $\{d_1, \ldots, d_n, e\}$ à l'ensemble F où d_i représente la profondeur i de chaque littéral x_i et y_i et e indiquant si un état est terminal. Un programme GDL encodant une formule de type QBF[DNF] et comprenant deux joueurs $\exists$ et $\forall$ est :

```
% rôles
role(∃).
role(∀).
% état initial
init(d_1).
% actions légales pour tous les i impairs
legal(∃, x_i) ← true(d_i).
legal(∃, x̄_i) ← true(d_i).
legal(∀, noop) ← true(d_i).
% actions légales pour tous les i pairs
legal(∀, y_i) ← true(d_i).
legal(∀, ȳ_i) ← true(d_i).
legal(∃, noop) ← true(d_i).
% mis à jour de l'état du jeu
next(z_i) ← does(P, z_i).
next(z̄_i) ← does(P, z̄_i).
next(d_{i+1}) ← does(P, A), true(d_i).
% états terminaux
terminal ← true(d_n).
e ← true(T_1).
e ← true(T_2).
...
e ← true(T_m).
goal(∃, 100) ← e.
goal(∃, 0) ← not(e).
goal(∀, 100) ← not(e).
goal(∀, 0) ← e.
```

Ici, à l'état initial seul le fluent d_1 est vrai. L'objectif du joueur $\exists$ est de vérifier la DNF contrairement à celui de $\forall$ qui est de falsifier cette même DNF. Au cours du jeu, le fluent z_i est vrai ou faux en fonction de la décision du joueur à la profondeur i. L'évolution du jeu est donc possible jusqu'à un état terminal identifié dès que la profondeur n est atteinte.

Soit st une stratégie pure stationnaire gagnante pour le joueur $\exists$. Alors l'arbre d'assignations associé à cette stratégie satisfait la formule QBF[DNF]. Inversement soit un arbre satisfaisant la formule, alors la stratégie associée à cet arbre (c'est à dire que chaque nœud est traduit en fonction des actions choisies pour le joueur donné) est une stratégie pure stationnaire gagnante dans le programme GDL. Par conséquent, trouver une stratégie pure stationnaire dans GDL propositionnel est PSPACE-DIFFICILE *impliquant que la recherche d'une politique gagnante dans le fragment propositionnel des jeux à information partagée d'horizon T est* PSPACE-DIFFICILE.

Montrons maintenant la PSPACE *complétude. Pour tout jeu G_T fini et dont la profondeur est limitée par l'horizon T, une politique gagnante peut être déterminée en $f(|G|)$ temps et en $g(|G|)$ espaces en utilisant une machine de Turing alternativement stochastique, c'est à dire une machine de Turing incluant des états stochastiques pour le joueur environnement, des états existentiels pour le joueur j dont la politique est gagnante et des états universels pour tous les autres joueurs.*

Comme G est propositionnel, le nombre de fluents et le nombre d'actions sont polynomiales sur $|G|$. Ensemble, ces deux faits impliquent que $g(|G|)$ est polynomiale.

À chaque état du jeu, la machine de Turing alternativement stochastique peut déterminer une sé-

quence d'actions à l'aide des états existentiels. Comme $k \leq |G|$, le prochain état peut être construit en utilisant un nombre polynomial d'états universels et d'états stochastiques. Par conséquent, en ajoutant le fait que cette machine de Turing peut trouver un état terminal après au plus T tours implique que $f(|G|)$ est aussi polynomiale.

Enfin, comme toute machine de Turing alternativement stochastique utilisant un nombre de temps et d'espaces polynomial peut être simulée en NPSPACE *[*BONNET *&* SAFFIDINE *2014] et suite au théorème de Savitch indiquant que* NPSPACE = PSPACE*, il s'en suit que G_T est dans* PSPACE*, ce qui confirme le théorème.* □

Rappelons que la recherche d'une politique solution dans un SCSP est PSPACE-COMPLET. Le problème reste PSPACE pour T-niveau à k joueurs car chaque niveau du problème est dans NP^{PP}. C'est pourquoi, il nous est possible d'étendre notre processus de traduction afin de permettre d'englober GDL-II.

3.3.2 De GDL-II à SCSP

La traduction d'un programme GDL-II $G \in G_T$ à k joueurs vers un SCSP à T niveaux est proche de la traduction d'un programme GDL classique.

Tout d'abord, nous extrayons les différents éléments composant chaque niveau du SCSP correspondant. La génération des différentes variables correspond toujours à $X = (\{F_t\}, \omega_t, U_t, \{A_t\}, A_{k+1,t}, \{F_{t+1}\})$. $\{F_t\}, \omega_t, U_t, \{A_t\}$ et $A_{k+1,t}$ sont identiques. Toutefois, $\{F_{t+1}\}$ correspond à un ensemble de variables stochastiques dont la distribution de probabilité représentera la fonction de croyance de l'état suivant. De même, afin de conserver la propriété d'un μSCSP permettant de dissocier l'ensemble ordonné des variables X en deux sous-ensembles distincts représentant les variables de décisions (ensemble V) et les variables stochastiques (ensemble Y), $A_{k+1,t}$ représentant la variable d'action du joueur environnement doit toujours être instanciée après $\{A_t\}$, qui pour rappel, représente les combinaisons d'actions des k joueurs.

L'extraction des domaines et des contraintes sont identiques au processus précédent car GDL et GDL-II propose la même sémantique et syntaxe. Mais, afin de modéliser les jeux à information incomplète, GDL-II propose un nouveau genre de règles décrit par le mot-clé *sees* représentant les informations perçues par chaque joueur en fonction de l'état du jeu.

Au cours du processus de traduction, les règles *sees* sont utilisées pour exprimer la table de probabilités conditionnelles $P(f_{t+1}|\boldsymbol{f}, \boldsymbol{a})$ de chaque variable stochastique composant $\{F_{t+1}\}$. Tout terme fluent non perçu est associé à une distribution de probabilités uniforme sur l'ensemble des termes constants pouvant l'instancier. Notons que la distribution de probabilités $P(a_{k+1}|\boldsymbol{f})$ de la variable stochastique $A_{k+1,t}$ reste uniforme sur l'ensemble des actions légales de l'environnement à chaque tour.

En répétant ce processus T fois, nous obtenons alors un SCSP à T niveaux modélisant le jeu GDL-II d'horizon T. Pour finir, le seuil θ est fixé à 1 afin de garantir que le SCSP doit être satisfait par une politique solution.

Exemple 3.4 *La figure 3.1 illustre un $\mu SCSP_t$ correspondant au programme GDL-II au temps t du « Hidden Matching Pennies », illustré en figure 1.6. Notons que les variables $action_{random,t}$ et $action_{j1,t}$ sont assimilées respectivement à $choisit_{random,t}$ et $retourne_{j1,t}$ pour des raisons de lisibilité afin de ne pas répéter le nom de l'action dans le domaine de chacune de ces variables. De même, la variable $terminal_t$ n'est pas exprimée dans les relations composant les contraintes goal afin de clarifier la lecture. Rappelons que si $terminal_t$ est faux alors quelque soit les valeurs des variables fluents $score_t$ est égale à 0. Les notations suivantes sont utilisées :*

— $P = pile$ *;*
— $F = face$ *;*

— I = $inconnu$;
— V = $vrai$;
— W = $faux$.

Premièrement, les deux variables $terminal_t$ et $score_{j1,t}$ représentent respectivement la fin du jeu et les scores possibles de $j1$. Ces variables sont extraites des mots-clés $terminal$ et $goal(J, S)$. Le domaine associé à $terminal_t$ est booléen et celui de $score_{j1,t}$ correspond aux différentes valeurs S possiblement impliquées dans $goal(J, S)$.

Grâce au mot-clé $role$, deux variables $control_t$ et $control_{t+1}$ déterminent le joueur possédant la « main » au tour t et $t + 1$. Le domaine de ces variables correspond à l'ensemble des joueurs défini par le mot-clé $role$.

Les états du jeu au tour t et $t + 1$ sont représentés par l'ensemble des variables dérivées de par le mot-clé next. Le domaine de ces variables correspond aux combinaisons des valeurs des différents fluents C, $C1$ et $C2$.

Puis, on utilise le mot-clé legal pour générer les variables $choisit_t$ et $retourne_t$. Notons que $choisit_t$ est une variable stochastique car elle correspond à l'action du joueur environnement ($random$). La seconde variable stochastique est $piece_{t+1}$. La distribution de probabilités associée à $piece_{t+1}$ est directement associée à la règle sees ($sees(j1, pieces(C1, \_)) \leftarrow does(random, choisit(C1, C2))$) indiquant que $J1$ ne perçoit que la première pièce au moment où $random$ choisit les deux cotés de ses pièces.

Une contrainte terminale relie la variable $terminal_t$ aux deux variables représentant les pièces. Le jeu se termine quand toutes les pièces ne sont plus assignées à la valeur « inconnu ». De la même manière, la contrainte goal relie les différents cotés de chacune des pièces aux scores associés à $j1$. Finalement les contraintes next permet au jeu de passer d'un état t à un autre état $t + 1$ dépendant de l'action choisie par le joueur (variable $retourne_t$) et par l'action réalisée par $random$ (variable $choisit_t$).

Au cours de la section suivante, nous amenons à montrer expérimentalement que notre processus de traduction de GDL (ou GDL-II) vers SCSP est compatible temporellement dans le cadre d'une compétition GGP.

3.4 Expérimentations

Dans ce cadre, nous présentons une série de résultats expérimentaux réalisés sur un cluster d'Intel Xeon E5-2643 CPU 3.4 GHz possédant 32 Gb de mémoire RAM sous Linux. Le processus de traduction a été implémenté en C++ et n'utilise aucun outil externe.

3.4.1 Jeux GDL

Afin de démontrer que la génération de chaque μSCSP représentatif d'un tour de jeu GDL est compatible avec les actuelles compétitions GGP orchestrées autour de jeux à information complète où le temps de pré-traitement (*startclock*) est généralement supérieur à 60 secondes, nous avons réalisé la traduction des 150 instances GDL proposées sur le serveur Tiltyard [SCHREIBER 2014] dans leur dernière version et qui ont été jouées au moins une fois dans la limite de 60 secondes.

Les tableaux 3.4, 3.5, 3.6 et 3.7 rapportent les différents résultats obtenus. La première colonne (Jeu) représente le nom de l'instance. La seconde colonne (Horizon) indique la valeur de t maximal pour tout $\mu SCSP_t$ généré par notre processus de compilation dans le temps accordé. La valeur obtenue est toujours supérieure ou égale au nombre maximal de tours joués sur le serveur Tiltyard pour l'ensemble des instances GDL garantissant en conséquence la possibilité d'atteindre un état terminal dans les temps. Les colonnes #vars, MaxDom et #conts représentent respectivement, le nombre de variables générées

Variable	Domaine
$\texttt{piece}_t$	$\{\texttt{F}, \texttt{P}, \texttt{I}\}$
$\texttt{pieces}_t$	$\{\texttt{FF}, \texttt{PP}, \texttt{FP}, \texttt{PF}, \texttt{II}\}$
$\texttt{control}_t$	$\{\texttt{j1}, \texttt{random}\}$
$\texttt{terminal}_t$	$\{\texttt{V}, \texttt{W}\}$
$\texttt{score}_{j1,t}$	$\{0, 50, 100\}$
$\texttt{choisit}_{random,t}$	$\{\texttt{FF}, \texttt{PP}, \texttt{FP}, \texttt{PF}, \texttt{noop}\}$
$\texttt{retourne}_{j1,t}$	$\{\texttt{F}, \texttt{P}, \texttt{noop}\}$
$\texttt{piece}_{t+1}$	$\{\texttt{F}, \texttt{P}, \texttt{I}\}$
$\texttt{piece}_{t+1}$	$\{\texttt{FF}, \texttt{PP}, \texttt{FP}, \texttt{PF}, \texttt{II}\}$
$\texttt{pieces}_{t+1}$	$\{\texttt{j1}, \texttt{random}\}$

Variables et domaines

$\texttt{piece}_t$	$\texttt{pieces}_t$	$\texttt{terminal}_t$
F	FF	T
F	FP	T
⋮	⋮	⋮
P	PP	T
F	FI	W
F	IP	W
⋮	⋮	⋮
I	II	W

Contraintes *terminal*

$\texttt{piece}_t$	$\texttt{pieces}_t$	$\texttt{score}_{j1,t}$
F	FF	100
P	PP	100
F	FP	50
F	PF	50
P	FP	50
P	PF	50
F	PP	0
P	FF	0

Contraintes *goal*

$\texttt{control}_t$	$\texttt{retourne}_{j1,t}$
j1	F
j1	P
random	noop

Contraintes *legal* de $j1$

$\texttt{control}_t$	$\texttt{control}_{t+1}$
random	j1

$\texttt{pieces}_t$	$\texttt{choisit}_{random,t}$	$\texttt{pieces}_{t+1}$
FF	noop	FF
PP	noop	PP
FP	noop	FP
PF	noop	PF
II	FF	FF
II	PP	PP
II	FP	FP
II	PF	PF

$\texttt{piece}_t$	$\texttt{retourne}_{j1,t}$	$\texttt{piece}_{t+1}$
I	noop	I
I	F	F
I	P	P

Contraintes *next*

$\texttt{control}_t$	FF	FP	PF	PP	noop
random	$1/4$	$1/4$	$1/4$	$1/4$	0
j1	0	0	0	0	1

Distribution de probabilité de $choisit_{random,t}$

$\texttt{sees}_t$	FF	FP	PF	PP
$\texttt{pieces}_{t+1}(\texttt{F}, \_)$	$1/2$	$1/2$	0	0
$\texttt{pieces}_{t+1}(\texttt{P}, \_)$	0	0	$1/2$	$1/2$

Distribution de probabilité de $pieces_t$ **en fonction des « sees » de** $j1$

FIGURE 3.1 – μSCSP encodant le « Hidden Matching Pennies » décrit en GDL-II en figure 1.6.

pour chaque μSCSP$_t$, la taille du plus grand domaine et le nombre de contraintes de chaque μSCSP$_t$. Finalement la dernière colonne (Temps) visualise le nombre de secondes nécessaires pour générer le μSCSP représentatif de chaque tour de jeu du programme GDL.

Jeux à un joueur

Le tableau 3.4 se concentre uniquement sur les jeux à un joueur du serveur Tiltyard. Tout d'abord, notons que chaque jeu est traduit en μSCSP en moins de 5 secondes excepté pour le *31 Queens Puzzle lg* représentatif du problème des 31 reines, impliquant une instance avec une taille plus importante, où $5,42$ secondes sont nécessaires.

Chaque μSCSP obtenu implique moins de 200 variables sauf pour les instances impliquant un nombre de fluents plus important notamment pour les instances du problème des reines impliquant plus de 12 reines. Le nombre de contraintes associées à chaque μSCSP est compris entre 5 et 29 contraintes, un nombre peu important de contraintes simplifiant ainsi leurs résolutions.

Jeux multijoueurs

Les tableaux 3.5, 3.6 et 3.7 se focalisent sur les instances multijoueurs du serveur Tiltyard. Comme pour les jeux impliquant un seul joueur, toutes les instances sont traduites avec un temps compatible avec le temps de pré-traitement des compétitions. Toutefois les instances de type *Checkers* (les dames) nécessitent un temps supérieur à 30 secondes, s'expliquant par des domaines beaucoup plus importants.

Les SCSPs à un niveau obtenus sont composés d'au plus 300 variables excepté pour l'instance GDL intitulé *Connect Four Larger* décrivant un Puissance 4 de taille 20×20 où chaque case du jeu est représentée par un fluent générant ainsi un μSCSP possédant plus de 800 variables. Généralement le domaine maximal pour chaque instance est satisfaisant (< 500) sauf pour les jeux construits avec des prédicats impliquant un nombre de combinaisons de termes important impliquant des domaines de grandes tailles. Tout comme pour les jeux à un joueur, le nombre de contraintes associées à chaque μSCSP est peu important (< 50).

Afin de permettre une résolution des instances, il est nécessaire d'atteindre les états terminaux afin d'obtenir les scores de chaque joueur. Ainsi, l'horizon atteinte pour chaque instance par notre processus surpasse ou égalise aisément le nombre moyen de tours joués sur le serveur Tiltyard indiquant la possibilité d'atteindre rapidement les différents états terminaux du jeu lors d'un match sur le serveur.

3.4.2 Jeux GDL-II

Nos résultats concernant la traduction de jeux GDL-II en SCSP sont résumés dans le tableau 3.8. Ce dernier tableau est réparti en 3 parties regroupant 15 jeux GDL-II respectivement répartis : (1) par les jeux à information imparfaite (impliquant uniquement un joueur environnement mais à information complète) ; (2) par les jeux à un joueur à information incomplète ; (3) par les jeux multijoueurs à information partagée.

La majorité des instances GDL-II proviennent du serveur Dresden [23]. Le lecteur pourra se référer au site boardgamegeek.com pour obtenir plus d'informations sur chacun des jeux expérimentés.

Le temps de pré-traitement accordé lors des quelques compétitions GGP à information incomplète en GDL-II est semblable aux compétitions à information complète à savoir supérieur à 60 secondes.

Pour l'ensemble des instances des trois catégories, le temps nécessaire de traduction est inférieur au temps de pré-traitement alloué lors de matchs GGP. Notons que le *Backgammon*, le *Can't Stop* et le *Yahtzee* demandent des temps plus importants que les autres instances (entre 35 et 50 secondes).

23. Serveur Dresden : http://ggpserver.general-game-playing.de/

Jeu	Horizon	#vars	MaxDom	#conts	Temps (s)
6 Queens Puzzle ug	6	77	72	6	0.23
8 Queens Puzzle lg	8	133	128	7	1.02
8 Queens Puzzle ug	8	133	128	6	1.01
12 Queens Puzzle ug	12	293	288	6	1.52
16 Queens Puzzle ug	16	517	512	6	3.01
31 Queens Puzzle lg	31	1927	1922	7	5.42
Asteroids	50	15	20	27	1.71
Cell Puzzle	30	55	50	11	2.24
Chinese Checkers 1P	40	79	1370	8	4.49
Coins	5	21	64	7	0.08
Eight Puzzle	60	23	81	8	0.41
European Peg Jumping	35	79	2401	11	3.14
Futoshiki 4x4	15	37	65	9	1.68
Futoshiki 5x5	23	55	126	9	1.75
Futoshiki 6x6	33	77	217	9	1.83
Hidato (19 hexes)	12	43	362	8	1.70
Hidato (37 hexes)	26	79	1370	8	3.42
Knight's Tour	30	65	900	7	1.94
Knight's Tour Large	81	167	6561	7	3.76
Lights On	20	21	20	10	0.03
Lights On Parallel	20	37	80	10	0.16
Lights On Simult-4	20	37	65536	13	1.21
Lights On Simultaneous	20	37	256	11	1.97
Lights On Simultaneous II	234	35	256	14	2.60
Lights Out	20	55	50	7	0.18
Max knights	33	135	512	9	1.36
Maze	10	9	10	10	0.12
Mine clearing (small)	49	30	650	29	2.03
Nonogram 5x5	13	29	25	5	0.26
Nonogram 10x10	54	111	100	5	1.48
Peg Jumping	31	71	2401	11	1.48
Rubik's Cube	50	85	484	16	2.42
Snake Parallel	20	23	800	11	1.04
Solitaire Chinese Checkers	55	43	1369	11	1.06
Sudoku Grade 1	50	165	810	8	1.39
Sudoku Grade 2	53	165	810	8	1.38
Sudoku Grade 3	54	165	810	8	1.41
Sudoku Grade 4	54	165	810	8	1.36
Sudoku Grade 5	53	165	810	8	1.39
Sudoku Grade 6E	64	165	810	8	1.36
Sudoku Grade 6H	64	165	810	8	1.40
Untwisty Complex 2	8	21	8	24	0.09

TABLE 3.4 – Traduction de jeux GDL à un joueur vers SCSP

Jeu	Horizon	#vars	MaxDom	#conts	Temps (s)
Amazons 8x8	112	134	4161	10	8.63
Amazons 10x10	184	206	10101	10	10.31
Amazons Suicide 10x10	184	206	10101	10	10.48
Amazons Torus 10x10	184	206	10101	10	10.51
Battle	30	90	6561	10	1.36
Battlebrushes 4P	20	146	325	12	4.21
Beat-Mania	60	56	90	12	1.15
Bidding Tic-Tac-Toe	18	17	30	15	0.67
Bidding Tic-Tac-Toe w/ 10 coins	18	17	30	15	0.69
Blocker	10	16	48	8	0.28
Bomberman	50	138	240	16	1.18
Breakthrough	207	70	4097	8	2.64
Breakthrough small	53	54	1297	8	1.16
Breakthrough small with holes	53	54	1297	9	1.16
Breakthrough with holes	207	70	4097	9	2.64
Breakthrough with walls	207	70	4097	8	2.60
Breakthrough suicide	207	70	4097	8	2.65
Breakthrough suicide small	53	54	1297	8	1.15
Cephalopod micro	51	24	603	8	1.23
Checkers	102	140	68173823	34	36.21
Checkers must-jump	60	140	68173823	34	36.82
Checkers on a Barrel without kings	100	60	8193	16	4.79
Checkers on a Cylinder, Must-jump	50	140	68173823	34	37.01
Checkers small	102	108	21685249	34	31.15
Checkers tiny	102	76	4329473	34	26.12
Checkers with modified goals	100	140	68173823	34	36.67
Chess	201	152	49153	49	12.16
Chicken Tic-Tac-Toe	9	24	27	12	0.31
Chinese Checkers v1 2P	40	20	401	9	2.16
Chinese Checkers v1 3P	60	27	401	11	2.27
Chinese Checkers v1 4P	60	46	401	18	2.43
Chinese Checkers v1 6P	61	48	401	17	2.49
Chinook	41	140	8193	27	4.16
Cit-Tac-Eot	25	56	75	16	1.43
Connect Five	64	134	192	7	1.33
Connect Four	48	102	96	9	0.86
Connect Four 3P	48	103	144	11	1.01
Connect Four 9x6	54	198	108	9	1.52
Connect Four Large	108	218	216	9	1.84
Connect Four Larger	400	804	800	9	4.72
Connect Four Simultanenous	48	200	192	9	2.06
Connect Four Suicide	48	102	96	9	0.86
Copolymer 4-length	42	88	280	15	2.32

TABLE 3.5 – Traduction de jeux GDL multijoueurs (A à C) vers SCSP

Jeu	Horizon	#vars	MaxDom	#conts	Temps (s)
Dollar Auction	32	8	42	12	0.21
Dots and Boxes	60	130	1297	16	1.64
Dots and Boxes Suicide	60	130	1297	16	1.65
Dual Connect 4	96	200	96	14	3.56
English Draughts	158	54	6145	18	5.74
Escort-Latch breakthrough	61	44	4097	13	3.88
Free for all 2P	30	20	2402	11	4.19
Free for all 3P	30	21	2402	11	4.21
Free for all 4P	30	22	2402	11	4.25
Free for all zero sum	30	20	2402	11	4.17
Ghost Maze	50	78	256	15	1.24
Golden Rectangle	56	118	224	14	2.21
Guess two thirds of the average 2P	1	6	400	5	0.08
Guess two thirds of the average 4P	1	8	400	7	0.08
Guess two thirds of the average 6P	1	10	400	9	0.09
Hex	81	170	162	14	1.43
Hex with pie	81	176	162	18	1.45
Iterated Chicken	20	7	101	13	0.78
Iterated Coordination	20	7	101	11	0.87
Iterated Prisoner's Dilemma	20	7	101	13	1.13
Iterated Stag Hunt	20	7	101	13	1.16
Iterated Tinfoll Hat Game	10	7	101	6	0.75
Iterated Ultimatum Game	20	9	101	15	1.20
Kalah 5x2x3	116	34	360	29	1.94
Kalah 6x2x4	234	40	577	29	2.06
Knight Fight	53	206	803	13	2.08
Knight Through	69	70	257	8	1.63
Majorities	55	308	378	17	2.68
Nine-Board Tic-Tac-Toe	81	170	163	9	1.86
Number Tic-Tac-Toe	9	24	90	9	0.84
Pacman 2P	100	164	144	15	1.71
Pacman 3P	100	166	144	14	1.72
Pawn Whopping	96	38	257	9	3.41
Pawn-to-Queen	17	134	257	15	1.84
Pentago	72	78	72	12	0.69
Pentago Suicide	72	78	72	12	0.72
Quarto	32	70	256	13	1.62
Quarto Suicide	32	70	256	13	1.62
Qyshinsu	50	56	121	24	1.61
Reversi	60	134	101	8	1.03
Reversi Suicide	60	134	101	8	1.06

TABLE 3.6 – Traduction de jeux GDL multijoueurs (D à R) vers SCSP

Jeu	Horizon	#vars	MaxDom	#conts	Temps (s)
Sheep and Wolf	48	134	577	11	2.49
Shmup	128	137	512	10	3.06
Skirmish variant	101	76	768	28	4.62
Skirmish zero-sum	101	76	768	28	4.69
Snake 2P	8	48	144	16	1.68
Snake Assemblit	40	44	108	21	2.31
Speed Chess	201	76	24581	26	8.04
The Centipede Game	19	10	20	17	0.73
Tic-Tac-Chess 2P	31	28	2402	8	1.97
Tic-Tac-Chess 3P	31	29	2402	9	2.01
Tic-Tac-Chess 4P	33	30	2402	10	2.04
Tic-Tac-Heaven	108	224	186	15	2.15
Tic-Tac-Toe	9	24	27	10	0.15
Tic-Tac-Toe 3P	25	57	75	11	1.30
Tic-Tac-Toe 9 boards with pie	81	174	164	13	1.63
Tic-Tac-Toe Large	25	56	50	8	1.26
Tic-Tac-Toe LargeSuicide	25	56	50	8	1.26
Tic-Tac-Toe Parallel	9	42	90	14	1.35
Tic-Tac-Toe Serial	18	44	27	22	0.46
Tic-Tic-Toe	6	24	27	10	0.82
Tron 10x10	50	296	432	10	1.89
TTCC4 2P	101	30	124856	20	7.21
War of Attrition	5	10	100	7	1.09
With Conviction !	25	56	150	13	1.41

TABLE 3.7 – Traduction de jeux GDL multijoueurs (T à Z) vers SCSP

Jeu	Horizon	#vars	MaxDom	#conts	Temps (s)
Jeux à information imparfaite					
Backgammon	400	113	5200	22	38.9
Can't Stop	240	148	67412	49	44.6
Kaseklau	45	27	62	55	2.49
Pickomino	400	54	144	121	4.84
Yahtzee	50	18	30	8862	47.8
Jeux à un joueur à information incomplète					
GuessDice	1	6	6	4	0.01
Mastermind	9	144	256	24	2.58
Monty Hall	4	8	7	14	0.07
Vaccum Cleaner Random	200	34	400	24	2.51
Wumpus	25	26	100	19	1.08
Jeux multijoueurs à information partagée					
Pacman 3P	100	178	192	14	1.83
Schnappt Hubi	100	96	412	23	3.40
Sheep & Wolf	48	136	577	11	2.91
TicTacToe Latent Random 9x9	81	60	243	24	2.21
War (card game)	546	13	2704	25	8.12

TABLE 3.8 – Traduction de jeux GDL-II vers SCSP

Ces derniers s'expliquent par des domaines plus importants et impliquent un plus grand nombre de contraintes, notamment pour le *Yahtzee*.

Chaque μSCSP possède un nombre de variables toujours inférieur à 200. Toutefois, notons qu'en dehors des jeux à information imparfaite (n'impliquant qu'une variable stochastique représentative des actions de l'environnement à chaque tour), les μSCSP générés proposent un nombre de variables stochastiques bien plus important indiquant une complexité de résolution plus élevée. Toutefois, le domaine maximal n'est pas très important dans la plupart des cas excepté pour le *Can't Stop* qui possède des prédicats avec un grand nombre de combinaisons de termes. Pour finir, le nombre de contraintes reste de la même importance que pour les jeux GDL classiques à part pour le *Pickomino* et le *Yathzee* décris par beaucoup plus de règles logiques et de ce fait générant plus de contraintes.

Ainsi, suite à ces expérimentations, il est possible d'envisager la modélisation de stratégies lors de compétitions GGP à l'aide des SCSP générés. Ceci est l'objet de la section suivante.

3.5 Modélisation des stratégies

Il est important de garder en tête qu'une politique solution π d'un réseau de contraintes stochastiques N ne garantit pas d'être une « stratégie dominante » pour certains des joueurs [SHOHAM & LEYTON-BROWN 2009]. En fait, la notion de politique mise en avant dans la résolution d'un SCSP équivalent à un programme GDL n'est pas équivalente à la notion d'arbre de jeu dans laquelle l'ensemble des nœuds de décisions auraient été partitionnés en k sous-ensembles, chacun associé à un joueur spécifique. Au lieu de réaliser cette équivalence, l'objectif globale de la satisfaction de réseaux de contraintes stochastiques est de déterminer une assignation aux nœuds de décisions pour lesquels l'utilité attendue résultante correspond au critère de seuil θ.

Sur la base du processus de traduction, le SCSP obtenu encode uniquement les règles du jeu GDL.

Dans le but de permettre « la résolution » d'un jeu, il est nécessaire de définir les solutions du SCSP d'un point de vue déclaratif. En somme, les solutions (ou dans le contexte des jeux, les stratégies) sont capturées par l'utilisation de deux opérateurs $\oplus$ et $\otimes$ sur l'ensemble des contraintes pour les relier.

Par la suite, la notation $\boldsymbol{a}_{-(k+1)} \in \{A_t\}$ comme abréviation de $(a_1, \ldots, a_t) \in A_{1,t} \times \ldots \times A_{k,t}$ afin de spécifier la séquence d'actions des k joueurs (excluant donc l'environnement $k+1$). Par extension $\boldsymbol{a}_{-(k+1),j} \in \{A_t\}_{-j}$ dénote une séquence d'actions $(a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_k)$ de k-1 joueurs excluant le joueur j et l'environnement. De même, la notation sur $\boldsymbol{f} \in \{F_t\}$.

Afin d'incorporer les stratégies des joueurs, chaque μSCSP fractionnant le SCSP généré est enrichi de plusieurs contraintes. La première contrainte $u_{k+1,t} : \{\{F_t\}, \{A_t\}, U_{k+1}\}$ associe à chaque joueur $j \in k$ dans chaque ensemble de fluents $f \in \{F_t\}$ et à chaque combinaison d'action $\boldsymbol{a}_{-(k+1)} \in \{A_t\}$, la valeur $u_{k+1,t}(j, \boldsymbol{f}, \boldsymbol{a}_{-(k+1)})$ dans U_{k+1} donnée par :

$$u_{k+1,t}(j, \boldsymbol{f}, \boldsymbol{a}_{-(k+1)}) = \sum_{a_{k+1} \in A_{k+1,t}} \sum_{\boldsymbol{f'} \in \{F_{t+1}\}} P(a_{k+1} \mid \boldsymbol{f}) P(\boldsymbol{f'} \mid \boldsymbol{f}, \boldsymbol{a})\ u^*_{j,t+1}(\boldsymbol{f'})$$

où $\boldsymbol{a} = (\boldsymbol{a}_{-(k+1)}, a_{k+1})$ est la dernière combinaison d'actions de tous les $k+1$ joueurs (incluant l'environnement $k+1$), et $u^*_{t+1,j}(\boldsymbol{f'})$ représente l'utilité du joueur j à l'état $\boldsymbol{f'}$. Intuitivement, $u_{k+1,t}(j, \boldsymbol{f}, \boldsymbol{a}_{-(k+1)})$ correspond à l'utilité attendue du joueur j quand le vecteur d'actions $\boldsymbol{a}_{-(k+1)}$ des k joueurs est appliquée à l'état $\boldsymbol{f}$.

Sur la base de cette valeur, nous pouvons définir l'utilité de chaque action de chacun des joueurs. Dans ce but, nous associons à chaque joueur j une contrainte $u_{j,t} : \{\{F_t\}, A_j, U_j\}$. Cette contrainte met en relation chaque état décrit par $\boldsymbol{f} \in \{F_t\}$ et chaque action $a_j \in \{A_j\}$ à la valeur $u_{j,t}(\boldsymbol{f}, a_j)$ dans U_j donnée par

$$u_{j,t}(\boldsymbol{f}, a_j) = \bigoplus_{\boldsymbol{a}_{-\{j,(k+1)\}} \in \{A_t\}_{-j}} u_{k+1,t}(j, \boldsymbol{f}, \boldsymbol{a}_{-(k+1)})$$

Symétriquement, nous pouvons capturer l'utilité de chaque joueur dans chaque état. Afin de réaliser cela, nous associons à chaque joueur j, une contrainte $u^*_{j,t} : \{\{F_t\}, U^*_j\}$. Cette contrainte représente chaque état décrit par $\boldsymbol{f} \in \{F_t\}$ à une valeur $u^*_{j,t}(\boldsymbol{f})$ dans U^*_j définie par la condition suivante : si $\boldsymbol{f}$ est un état terminal, alors $u^*_{j,t}(\boldsymbol{f})$ correspond au score de j dans $\boldsymbol{f}$ spécifié par les règles *goal*. Sinon,

$$u^*_{j,t}(\boldsymbol{f}) = \bigotimes_{a_j \in A_{j,t}} u_{j,t}(\boldsymbol{f}, a_j)$$

Enfin, l'action optimale est capturée en ajoutant simplement la contrainte d'égalité $u^*_{j,t}(\boldsymbol{f}) = u_{j,t}(\boldsymbol{f}, a_j) : \{U^*_j, U_j\}$. Ces égalités permettent de filtrer chaque action non optimale du joueur.

Remarque 3.2 *Notons que d'autres types de solutions (ou de stratégies) peuvent être définies sur les opérateurs $\oplus$ et $\otimes$. Dans la suite de ce manuscrit et plus particulièrement dans les chapitres suivants illustrant un programme-joueur basé sur la résolution d'une succession de μSCSP, nous utilisons la stratégie standard minmax (voir définition 1.11) pour tous les joueurs donnée par $\otimes$ = max et $\oplus$ = min.*

3.6 Conclusion

Au cours de ce chapitre, le cadre stochastique des réseaux de contraintes et le formalisme GDL sont mis en relation au travers d'une sémantique originale basée sur les jeux de Markov. Suite à la proximité des deux modèles, nous avons proposé un processus de traduction de tout jeux GDL à information imparfaite à un temps quelconque t en problème de satisfaction de contraintes stochastiques à un niveau. Nous

montrons qu'en répétant ce processus autant de fois que le jeu GDL possède de tours, nous obtenons un réseau de contraintes stochastiques équivalent au jeu GDL.

Puis, nous identifions un fragment PSPACE de GDL-II représentant l'ensemble des jeux à information partagée et proposons une extension de notre processus de traduction à ce fragment. Dans la partie suivante, nous montrons expérimentalement la compatibilité temporelle de la génération d'un SCSP équivalent au jeu GDL avec les temps de pré-traitement accordés lors des compétitions GGP.

Enfin, suite à cette compatibilité, nous proposons la modélisation de différents types de stratégies en tant que solutions du SCSP obtenu en enrichissant ce dernier de quelques contraintes supplémentaires permettant de capturer l'action optimale d'un joueur j.

Plusieurs perspectives s'ouvrent naturellement à ce travail. Dans un premier temps, il serait intéressant d'étendre cette modélisation à l'ensemble des jeux GDL-II. En effet, différentes modélisations plus génériques que les réseaux de contraintes stochastiques peuvent être envisagées et leur étude couplée à des tests expérimentaux seront l'objet de recherches futures. Naturellement, la modélisation de jeux GDL en réseaux de contraintes ouvre la voie à la recherche afin d'exploiter les techniques de résolutions adaptées à ce modèle afin de proposer un programme générique pour le *General Game Playing* adapté au cadre des compétitions. Ce dernier travail est le sujet qui nous intéresse dans le chapitre suivant.

Chapitre 4

Une approche GGP dirigée par les contraintes

Sommaire

4.1 Un fragment SCSP pour GDL . . . 126
4.2 MAC-UCB . . . 127
4.2.1 Phase de pré-traitement . . . 127
4.2.2 Phase de résolution : MAC . . . 128
4.2.3 Phase de simulation : UCB . . . 131
4.3 MAC-UCB-II . . . 132
4.4 Évaluations expérimentales . . . 134
4.4.1 Dilemme exploration/exploitation . . . 134
4.4.2 MAC-UCB Vs. FC-UCB . . . 135
4.4.3 MAC-UCB face aux meilleures approches GDL . . . 136
4.4.4 MAC-UCB-II face aux meilleures approches GDL-II . . . 140
4.5 WoodStock : un programme-joueur dirigé par les contraintes . . . 144
4.5.1 Protocole de communication . . . 145
4.5.2 Phase de traduction . . . 145
4.5.3 Phase de résolution . . . 146
4.5.4 Phase de simulation . . . 147
4.6 WoodStock en compétition . . . 147
4.6.1 *Tiltyard* Open 2015 . . . 147
4.6.2 Tournoi continu GGP . . . 149
4.7 Conclusion . . . 150

Comme vu dans le chapitre 1, les algorithmes GGP incluent, parmi d'autres, la programmation logique, les ASP, la construction automatique de fonctions d'évaluations et les méthodes de type Monte Carlo. Au cours de ce nouveau chapitre, nous nous focalisons sur le formalisme des réseaux de contraintes stochastiques utilisé pour la prise de décisions séquentielles, dont l'effet incertain des actions est modélisé par une distribution de probabilités.

De ce point de vue, nous examinons un fragment du problème de satisfaction des réseaux de contraintes stochastiques capturant un fragment du formalisme GDL-II représentatif des jeux à information partagée comprenant notamment les jeux à information complète et les jeux à information imparfaite. L'intérêt

principal des SCSP représentant cette classe de jeux est qu'il peut être décomposé en séquence de μSCSP (aussi dénommé SCSP à un niveau). Basée sur cette décomposition, nous proposons un algorithme de décisions séquentielles, dénommé MAC−UCB combinant l'algorithme MAC (pour *Maintening Arc Consistency*) permettant de résoudre chaque μSCSP dans la séquence et la méthode des bandits à plusieurs bras UCB (pour *Upper Confidence Bound*) afin d'approximer l'utilité attendue des stratégies.

Nous montrons qu'en pratique MAC−UCB domine FC−UCB, une variante où l'algorithme MAC est remplacé par la méthode FC (pour *Forward Checking*) adaptée au cadre des SCSP mais également les meilleures approches GGP actuelles au travers de nombreuses expérimentations sur l'ensemble des jeux du serveur Tiltyard et sur 15 jeux GDL−II représentatifs du fragment en question.

Le chapitre est organisé comme suit : Tout d'abord, nous identifions le fragment SCSP obtenu suite à la traduction d'un jeu GDL en un SCSP équivalent et nous proposons une méthode itérative exploitant efficacement ce fragment en permettant la résolution de chaque μSCSP. Par la suite, nous mettons en avant l'algorithme MAC−UCB proposant une résolution adaptée aux jeux GDL à information imparfaite. En section 3, nous montrons que MAC−UCB peut être étendue à la résolution des SCSP traduits de jeux GDL−II dans une seconde version dénommée MAC−UCB−II. La section 4 confirme la performance de MAC−UCB et de MAC−UCB−II face à diverses approches GGP. Finalement en section 5, nous proposons un programme joueur générique, dénommé WoodStock implémentant MAC−UCB dans le cadre de compétitions GGP et qui après 2.000 matchs, a obtenu la place de leader dans la compétition continue.

La plupart des résultats présentés ici, ont été publiés à différentes occasions : [KORICHE *et al.* 2016c, KORICHE *et al.* 2016b, KORICHE *et al.* 2015b]

4.1 Un fragment SCSP pour GDL

D'un point de vue théorie des jeux, les réseaux de contraintes stochastiques étudiés dans [HNICH *et al.* 2009, TARIM *et al.* 2006, WALSH 2009] englobent les jeux stochastiques à un joueur, dans lesquels l'environnement (défini via les variables stochastiques) est « inconscient » : lors de chaque tour du jeu, la distribution de probabilités sur les actions de l'environnement est indépendante de la description de l'état courant. Au cours du chapitre précédent, nous avons proposé une modélisation d'un programme GDL (resp. GDL−II) en un réseau de contraintes stochastiques englobant les jeux multi-joueurs à information imparfaite (resp. à information partagée).

Empruntant la terminologie de [WALSH 2009], un niveau (de décision) dans un SCSP est un tuple de variables (V_t, Y_t), où V_t est un sous-ensemble comprenant les variables de décisions et Y_t est un sous-ensemble incluant les variables stochastiques, de plus les variables de décisions apparaissent avant les variables stochastiques.

Définition 4.1 (T-niveaux SCSP à *k* joueurs) *Un T-niveaux SCSP à k joueurs est un SCSP à k joueurs $\mathcal{P}$ = (X,Y,D,C,P,θ) dans lequel X peut être partitionné en T niveaux, où $X = ((V_0, Y_0), \ldots, (V_T, Y_T))$ avec $\{V_t\}_{t=0}^{T}$ est une partition de $X \setminus Y$, $\{Y_t\}_{t=0}^{T}$ est une partition de Y, et $scp(x_{s,i}) \subseteq V_t$ pour chaque $x_{s,t} \in Y_t$ et chaque $t \in T$. Si $T = 0$, N est un SCSP à un niveau, noté μSCSP.*

D'un point de vue computationnelle, satisfaire un problème SCSP à T-niveaux et à k-joueurs est PSPACE-dur, car il inclut les T-niveaux SCSP à un joueur comme un cas particulier [WALSH 2009]. D'autre part, la complexité d'un μSCSP à k joueurs est seulement NP^{PP}. Cela résulte de la NP^{PP}-dureté d'un μSCSP à un joueur [WALSH 2009] et du fait que, pour un SCSP à un niveau, les nœuds de décisions d'une politique solution π peuvent être déterminés en temps non polynomial (NP), et que l'utilité attendue de π peut être vérifiée en temps polynomialement probabiliste (PP).

Il est important de préciser que le réseau de contraintes stochastiques à T-niveaux encodant le jeu

GDL peut être décomposé en une séquence de $(\mu\text{SCSP}_0, \ldots, \mu\text{SCSP}_T)$ de réseau de contraintes stochastiques à un niveau.

Spécifiquement, soit $\mathcal{P} = (X, Y, D, C, P, \theta)$ le SCSP associé à un programme GDL, où les variables ordonnées de X sont partitionnées en T niveaux $(X_0, \ldots, X_t)$, avec $X_t = (\{F_t\}, \omega_t, U_t, \{A_t\}, A_{k+1,t})$. Puis, chaque μSCSP_t dans la séquence est un tuple $(X_t, Y_t, D_t, C_t, P_t, \theta)$, avec $X_t = (\{F_t\}, \omega_t, U_t, \{A_t\}, A_{k+1,t}, \{F_t + 1\})$, Y_t est la restriction de Y à la variable stochastique $A_{k+1,t}$ modélisant l'action de l'environnement au temps t, D_t et C_t sont les restrictions de D et C sur les variables dans X_t, P_t est la restriction de P sur la distribution de probabilités de $A_{k+1,t}$. Bien que $A_{k+1,t}$ est suivie par l'ensemble de variables de décisions $\{F_t + 1\}$) ordonnées dans X_t, de telles variables expriment les fluents pour lesquelles leurs valeurs sont propagées par les contraintes *next* une fois que les variables d'actions précédentes $\{a_{0,t}, \ldots, a_{k,t}, A_{k+1,t}\}$ ont été instanciées.

Par construction, la décomposition décrite ci-dessus en μSCSP induit une partition de l'ensemble des contraintes C de $\mathcal{P}$ sur les contraintes composant C_t, chacune associée au μSCSP_t auquel elle appartient. En outre, comme tout programme GDL bien formé G représente un arbre de jeu stochastiquement structuré, chaque état de jeu peut atteindre un état terminal. De par l'équivalence de $\mathcal{P}$ encodant G, ceci implique que toute instanciation I cohérente dans la sous-séquence $(\mu\text{SCSP}_0, \ldots, \mu\text{SCSP}_t)$ garantit d'être globalement cohérente pour $\mathcal{P}$.

Dit autrement, le réseau de contraintes stochastiques d'un programme GDL à horizon T peut être décomposé en μSCSP_t plus simple à résoudre, chacun associé à un sous-ensemble distinct de contraintes. Une telle décomposition encourage naturellement la résolution d'un programme GDL séquentiellement, en résolvant itérativement chaque μSCSP_t dans la séquence.

4.2 MAC−UCB

Basé sur le fragment des SCSP illustré dans la section précédente, nous proposons maintenant notre technique de résolution MAC−UCB. Comme indiqué auparavant, le réseau de contraintes stochastiques d'un programme GDL est une séquence de μSCSPs, chacun associé à un tour de jeu. Pour chaque $\mu\text{SCSP}_t \in \{0, \ldots, T\}$, MAC−UCB recherche l'ensemble des politiques solutions en décomposant le problème en deux parties : un CSP classique et un μSCSP (plus petit que l'original). La première partie est résolue à l'aide de l'algorithme MAC et la seconde partie grâce à l'algorithme SFC dédié au cadre SCSP. Par la suite, une série d'échantillonnages avec borne de confiance est réalisée pour simuler l'utilité attendue de chaque politique solution de chaque μSCSP_t.

4.2.1 Phase de pré-traitement

Avant d'examiner la résolution d'un μSCSP, nous appliquons quelques techniques de pré-traitement afin de faciliter le processus de résolution suivant.

Dans un premier temps, toutes les contraintes portant exactement sur les mêmes variables sont fusionnées. Soit un μSCSP $\mathcal{P}$ et deux contraintes c_i et c_j de $\mathcal{P}$ tel que $scp(c_i) = scp(c_j)$. Ces deux contraintes sont alors supprimées et remplacées par une unique contrainte c_k tel que $rel(c_k) = rel(c_i) \cap rel(c_j)$ et $scp(c_k) = scp(c_i) = scp(c_j)$. Cette étape est importante car lors du processus de traduction chaque règle logique est associée à une contrainte. De ce fait, si le programme GDL comporte une règle logique pour décrire chaque évolution du jeu (par le prédicat *next*) pour chaque combinaison de terme constant appartenant aux mêmes termes variables, une contrainte sera générée pour chacune de ces combinaisons avec la même portée. Cette étape réduit alors le μSCSP original en réseau de contraintes où chacune des contraintes le composant possède une portée différente des autres contraintes du réseau.

La seconde étape est de supprimer toutes les contraintes unaires en projetant chacune de leur relation

sur le domaine de chaque variable contrainte par celles-ci limitant ces valeurs autorisées aux tuples de la relation associée. Comme la génération du μSCSP au temps 0 intègre une contrainte unaire pour chaque occurrence de règle construite à l'aide du prédicat *init* et que la génération du μSCSP au temps $t+1$ est composée de contraintes unaires correspondant à chaque solution du μSCSP au temps t, cette étape est primordiale. En effet, en projetant les relations de ces contraintes, dans un premier temps, elle permet de diminuer le nombre de contraintes et dans un second de temps de diminuer fortement la taille des domaines des variables restantes.

Par la suite, nous supprimons toutes les variables universelles. Rappelons qu'une variable est dite universelle dans une contrainte c si quelque soit la valeur assignée, c est toujours satisfaite. Formellement, soit une contrainte c, une variable $x \in scp_c$ est dite universelle si $|rel(c)|$ est égale au produit de $|dom(x)|$ et du nombre de tuples de la relation associée à $scp(c_i)$, où $scp(c_i) = \{scp(c) \,/\, \{x\}\}$. De telles variables (induites par la traduction d'un jeu GDL en SCSP) sont supprimées de la portée de ces contraintes.

Finalement, la dernière technique de pré-traitement utilisée exploite la propriété induite par SAC (*Singleton Arc Consistency*). Pour rappel, un réseau de contraintes $\mathcal{P}$ est singleton arc-cohérent si chaque valeur (de chaque variable) de $\mathcal{P}$ est singleton arc-cohérente. Une valeur est singleton arc-cohérente si quand elle est assignée à la variable qui lui correspond, elle n'induit pas un réseau arc-incohérent. Ainsi en appliquant cette dernière propriété, les valeurs incohérentes sont supprimées du domaine de chaque variable.

Toutes ces techniques sont appliquées sur chaque μSCSP, excepté sur μSCSP_T où les utilités peuvent être obtenues. Notons que SAC est appliquée uniquement sur la partie dure (partie CSP) extraite de chaque μSCSP. La modélisation de la partie dure est traitée dans la partie suivante.

4.2.2 Phase de résolution : MAC

Après la réalisation de toutes les techniques de pré-traitement, l'objectif de la phase de résolution est d'énumérer l'ensemble des politiques solutions du μSCSP, dont certaines peuvent mener à une solution optimale.

Comme indiqué en section 2.3.2, la meilleure méthode de résolution complète d'un μSCSP est l'algorithme de *Forward Checking* (SFC) adapté à ce cadre. Malheureusement, dans le cadre d'un μSCSP impliquant un nombre de contraintes important, SFC n'est pas assez efficace suite à sa faible capacité de filtrage. Par conséquent, nous proposons ici une technique de résolution adaptée au SCSP à un niveau.

Nous commençons par séparer le μSCSP $\mathcal{P}$ en un CSP $\mathcal{P}'$ modélisant la **partie dure** composée uniquement des contraintes dures de $\mathcal{P}$ (voir algorithme 4.1) et en un μSCSP $\mathcal{P}''$ modélisation la **partie chance** contenant uniquement les contraintes de chances de $\mathcal{P}$ (voir algorithme 4.2). Les politiques solutions du μSCSP sont alors identifiées en combinant les solutions du CSP $\mathcal{P}'$ avec les solutions du μSCSP $\mathcal{P}''$.

Tout d'abord, examinons la résolution de $\mathcal{P}''$. Pour des instances GDL décrivant des jeux à information imparfaite, $\mathcal{P}''$ inclue au plus une unique contrainte (stochastique) capturant la règle de transition impliquée par le joueur environnement (*random*). Par conséquent, $\mathcal{P}''$ peut être facilement résolue par SFC adapté au cadre des SCSP à un niveau. Par la suite, l'ensemble de politiques solutions obtenues par SFC sur $\mathcal{P}''$ est encodé par une contrainte dure c_f, dénommée **contrainte de faisabilité**, où $scp(c_f)$ contient l'ensemble des variables de décisions de $\mathcal{P}''$ et $rel(c_f)$ est l'ensemble des tuples correspondant aux assignations des variables de décisions qui font parties d'au moins une politique solution de $\mathcal{P}''$. Formellement, soit $sols(\mathcal{P}'')$ l'ensemble des politiques solutions de P'' et $q = |sols(\mathcal{P}'')|$ alors :

— $scp(c_f) = vars(\pi)$ tel que $\pi \in sols(\mathcal{P}'')$;

— $rel(c_f) = (\tau_0, \ldots, \tau_i, \ldots, \tau_q)$ tel que $\tau_i \in \pi_i \in sols(\mathcal{P}'')$.

Désormais, intéressons nous à la résolution du CSP $\mathcal{P}'$. Ici, l'algorithme classique MAC est utilisé pour énumérer toutes les solutions de $\mathcal{P}'$. Dans le but de tenir compte des solutions identifiées dans $\mathcal{P}''$,

Algorithme 4.1 : *partie_dure*

Données : μSCSP $\mathcal{P} = (X, Y, D, C, P, \theta)$
Résultat : CSP $\mathcal{P}' = (X', D', C')$
$\mathcal{P}' = (X' \leftarrow \emptyset, D' \leftarrow \emptyset, C' \leftarrow \emptyset)$
pour $c \in C$ **faire**
  si $\nexists x_s \in scp(c) \mid x_s \in Y$ **alors**
    $C' \leftarrow C' \cup \{c\}$
    pour $x_d \in scp(c)$ **faire**
      $X' \leftarrow X' \cup \{x_d\}$
      $D' \leftarrow D' \cup \{dom(x_d)\}$
    fin
  fin
fin
retourner $\mathcal{P}'$

Algorithme 4.2 : *partie_chance*

Données : μSCSP $\mathcal{P} = (X, Y, D, C, P, \theta)$
Résultat : μSCSP $\mathcal{P}'' = (X'', Y'', D'', C'', P'', \theta)$
μSCSP $\mathcal{P}'' = (X'' \leftarrow \emptyset, Y'' \leftarrow \emptyset, D'' \leftarrow \emptyset, C'' \leftarrow \emptyset, P'' \leftarrow \emptyset, \theta)$
pour $c \in C$ **faire**
  si $\exists x_s \in scp(c) \mid x_s \in Y$ **alors**
    $C'' \leftarrow C'' \cup \{c\}$
    pour $x \in scp(c)$ **faire**
      si $x \in Y$ **alors**
        $Y'' \leftarrow Y'' \cup \{x\}$
        $P'' \leftarrow P'' \cup \{P_x\}$
      fin
      sinon
        $X'' \leftarrow X'' \cup \{x\}$
      fin
      $D'' \leftarrow D'' \cup \{dom(x)\}$
    fin
  fin
fin
retourner $\mathcal{P}''$

nous ajoutons simplement la contrainte de faisabilité c_f aux contraintes C' de $\mathcal{P}'$. Les solutions obtenues par MAC sur $P' = (X', D', C' \cup \{c_f\})$ sont identiques à l'ensemble des solutions du μSCSP P original.

MAC exploite la propriété de consistance d'arc avec pour objectif de filtrer efficacement les politiques non satisfaisantes du CSP $\mathcal{P}'$. Notons également qu'il est également possible de traiter d'abord la résolution du CSP $\mathcal{P}'$ (sans la contrainte de faisabilité) pour ensuite traiter le μSCSP $\mathcal{P}''$. Cependant suite à la petite taille de $\mathcal{P}''$, il est plus efficace en pratique de résoudre d'abord $\mathcal{P}''$ avant de s'attarder sur le problème plus imposant que représente $\mathcal{P}'$.

Exemple 4.1 *Nous illustrons la résolution d'un μSCSP $\mathcal{P} = (X, Y, D, C, P, \theta)$ où :*
- $X = \{x_1, x_2, y\}$ *;*
- $Y = \{y\}$ *;*
- $D = \{dom(x_1), dom(x_2), dom(y)\}$ *avec* $dom(x_1) = \{1, 2, 3\}$ *et* $dom(x_2) = dom(y) = \{0, 1, 2\}$ *;*
- $C = \{c_1, c_2, c_3\}$ *où chaque contrainte est détaillée dans la figure 4.1 ;*
- $P = \{P_y\}$ *avec* $P_y = \{P(y = 0) = \frac{1}{3}, P(y = 1) = \frac{1}{3}, P(y = 2) = \frac{1}{3}\}$
- $\theta = \frac{3}{4}$.

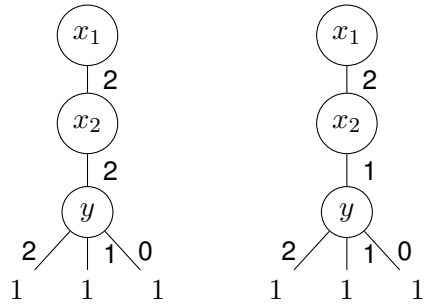
$$c_1(x_1, y) = \begin{cases} 1 & \text{si } x_1 + y > 1 \\ 0 & \text{sinon} \end{cases}$$

$$c_2(y, x_2) = \begin{cases} 1 & \text{si } y + x_2 \geq 1 \\ 0 & \text{sinon} \end{cases}$$

$$c_3(x_1, x_2) = \begin{cases} 0 & \text{si } x_1 = x_2 \\ -\infty & \text{sinon} \end{cases}$$

FIGURE 4.1 – Les contraintes du μSCSP $\mathcal{P}$

La première étape est de dissocier les contraintes stochastiques afin de construire le μSCSP $\mathcal{P}''$. Ici, les seules contraintes stochastiques sont c_1 et c_2. Le μSCSP $\mathcal{P}''$ est alors restreint à ces deux contraintes. Pour le problème $\mathcal{P}''$, SFC retourne deux politiques solutions π_1 et π_2 décrites dans la figure 4.2.



(a) π_1 (b) π_2

FIGURE 4.2 – Les solutions du μSCSP $\mathcal{P}''$.

Suite à la résolution de $\mathcal{P}''$, nous pouvons construire la contrainte de faisabilité c_f qui sera ajoutée à $\mathcal{P}'$. c_f est défini par la portée $scp(c_f) = \{x_1, x_2\}$ et la relation $rel_{c_f} = \{(2, 1), (2, 2)\}$. Le problème $\mathcal{P}'$ associé à la partie CSP de $\mathcal{P}$ est alors défini par l'ensemble de variables (de décisions) $\{x_1, x_2\}$ (associées à leur domaine respectif) et aux contraintes $\{c_3, c_f\}$.

L'algorithme MAC *retourne pour ce problème une seule et unique solution* $(x_1 = 2\,;\, x_2 = 2)$ *correspondant à la politique* π_1*. Cette politique solution est l'unique solution de* $\mathcal{P}$*.*

4.2.3 Phase de simulation : UCB

Rappelons dans un premier temps que tout programme GDL est un jeu séquentiel fini et que les utilités (les scores) de chaque joueur ne sont accessibles que dans un état terminal. De plus, de manière générale, les instances de jeu utilisées en pratique impliquent un arbre de jeu possédant une profondeur et une largeur importante. MAC, à lui seul, ne peut donc pas résoudre l'ensemble de l'arbre de jeu en accord avec le temps délimité dans le cadre d'un match GGP.

C'est pourquoi, afin de répondre à ce problème, il est nécessaire de simuler les prochains états de l'arbre de jeu de tout état non-terminal afin d'estimer l'utilité des solutions qu'apporte MAC à chaque μSCSP$_t$. Dans ce but, nous utilisons une technique de bandits multi-bras dirigée par la propriété UCB (pour *Upper Confidence Bound*) en considérant chaque politique solution du μSCSP$_t$ comme un « bras ». À partir d'une politique partielle du SCSP équivalent au jeu GDL associée à une politique solution du μSCSP$_t$, nous simulons uniformément et aléatoirement toutes les coups possibles de $t+1$ à $T-1$.

La « meilleure » solution du μSCSP$_t$ est celle qui maximise $\bar{u}_i + \sqrt{\frac{2\ln n}{n_i}}$, où $\bar{u}_i$ est le score moyen de la politique solution i, n_i est le nombre de fois où i a été simulé jusque là, et n est le nombre total de simulations réalisées. La résolution du prochain problème est décidée en instanciant μSCSP$_{t+1}$ par les valeurs de la meilleure politique solution estimée à partir de μSCSP$_t$.

L'algorithme 4.3 détaille le déroulement de MAC-UCB. Le *template* μSCSP$_t$ correspond au réseau de contraintes stochastiques à un niveau issue de la traduction du jeu GDL correspondant. Le second paramètre μSCSPs est une liste de réseaux stochastiques à un niveau (μSCSP$_i$) ordonnée de manière décroissante en fonction de l'utilité attendue (v_i) de chaque réseau. À l'état initial ($t = 0$), μSCSPs n'est composée que d'un seul réseau de contraintes stochastiques à un niveau correspondant au *template* μSCSP$_t$ et où les contraintes modélisant les règles « init » du jeu GDL correspondant sont ajoutées. Toutefois lors de l'appel de MAC-UCB à d'autres temps $t \neq 0$, μSCSPs ne contient que les réseaux de contraintes stochastiques à un niveau de temps supérieur à t associés à leurs utilités calculées précédemment.

MAC-UCB résout itérativement chaque μSCSP de la liste μSCSPs tant qu'il reste du temps (ligne 1). À chaque itération, le réseau $\mathcal{P}$ choisit pour être résolu est le premier élément de la liste μSCSPs correspondant au réseau associé à la plus grande utilité attendue et il est supprimé de la liste μSCSPs (ligne 2).

La résolution de $\mathcal{P}$ commence par la génération du CSP $\mathcal{P}'$ correspondant à la partie dure (ligne 3) et par la génération du μSCSP $\mathcal{P}''$ représentant la partie chance (ligne 4) à l'aide des algorithmes 4.1 et 4.2. $\mathcal{P}''$ est résolu par l'algorithme de *Forward Checking* adapté au cadre SCSP (algorithme 2.15) (ligne 5). Les lignes 6 à 13 illustrent la génération de la contrainte de faisabilité c_f à l'aide des solutions ($sols(\mathcal{P}'')$) du réseau $\mathcal{P}''$. c_f est ajoutée aux contraintes du réseau P' (ligne 14) et il est résolu par l'algorithme MAC (ligne 15).

Les lignes 16 à 25 permettent de générer et d'ajouter les réseaux $muSCSP_{t+1}$ à la liste μSCSPs afin de les résoudre aux prochaines itérations. La génération de chaque réseau au temps $t+1$ est effectuée à partir de toute solution obtenue qui ne correspond pas à un état terminal (ligne 17) détecté à l'aide de la variable ω (qui pour rappel indique si l'état correspondant est terminal ou non). Le réseau μSCSP$_i$ généré pour chaque solution τ_i est initialisé à l'aide du *template* μSCSP$_t$ (ligne 18). Puis pour chaque instanciation d'une variable f_{t+1} (correspondant au fluent au temps suivant $t+1$) du tuple solution τ_i, nous ajoutons une contrainte unaire restreignant le domaine de chaque variable f_t (correspondant au même fluent au temps t) du nouveau réseau μSCSP$_i$ au singleton $\{\tau_i[f_{t+1}]\}$ (lignes 19 et 20). L'utilité

attendue v_i correspondant au réseau $\mu\texttt{SCSP}_i$ est générée à l'aide de la technique UCB (ligne 22). Finalement, le couple $(\mu\texttt{SCSP}_i, v_i)$ est inséré correctement dans la liste $\mu\texttt{SCSP}s$ en fonction de son utilité v_i (ligne 23).

Algorithme 4.3 : MAC-UCB

Données : Réseau *template* $\mu\texttt{SCSP}_t = (X_t, Y_t, D_t, C_t, P_t, 1)$, Liste ordonnée de réseaux stochastiques $\mu\texttt{SCSP}s = \{(\mu\texttt{SCSP}_0, v_0), \ldots, (\mu\texttt{SCSP}_n, v_n)\}$ avec $v_0 \geq \ldots \geq v_n$

1 **tant que** $\neg$ *timeout* **faire**
2 Sélectionner et supprimer le premier réseau $\mathcal{P}$ de $\mu\texttt{SCSP}s$
3 CSP $\mathcal{P}' = (X', D', C') \leftarrow partie\_dure(\mathcal{P})$
4 μSCSP $\mathcal{P}'' \leftarrow partie\_chance(\mathcal{P})$
5 $sols(\mathcal{P}'') \leftarrow \texttt{SFC}(\mathcal{P}'')$
6 Contrainte c_f = $(scp(c_f) \leftarrow \emptyset, rel(c_f) \leftarrow \emptyset)$
7 $scp(c_f) \leftarrow vars(\pi) \mid \pi \in sols(\mathcal{P}'')$
8 **pour chaque** $\pi_k \in sols(\mathcal{P}'')$ **faire**
9 **pour chaque** $(x = v) \in \pi_k \mid x \in V_t''$ **faire**
10 $\tau_k[x] \leftarrow v$
11 **fin**
12 $rel(c_f) \leftarrow rel(c_f) \cup \tau_k$
13 **fin**
14 $\mathcal{P}' = (X', D', C' \cup \{c_f\})$
15 $sols(\mathcal{P}') \leftarrow \texttt{MAC}(\mathcal{P}')$
16 **pour chaque** $\tau_i \in sols(\mathcal{P}')$ **faire**
17 **si** $\tau_i[\omega] = 0$ **alors**
18 $\mu\texttt{SCSP}_i = (X_i \leftarrow X_t, Y_i \leftarrow Y_t, D_i \leftarrow D_t, C_i \leftarrow C_t, P_i \leftarrow P_t, 1)$
19 **pour chaque** $\tau_i[f_{t+1}] \mid f_{t+1} \in X'$ **faire**
20 $C_i \leftarrow C_i \cup \{(f_t = \tau_i[f_{t+1}])\} \mid f_t \in X_i$
21 **fin**
22 $v_i \leftarrow \texttt{UCB}(\mu\texttt{SCSP}_i)$
23 $\mu\texttt{SCSP}s \leftarrow \mu\texttt{SCSP}s \cup \{(\mu\texttt{SCSP}_i, v_i)\}$ // en accord avec l'ordre de $\mu\texttt{SCSP}s$
24 **fin**
25 **fin**
26 **fin**

4.3 MAC-UCB-II

Comme illustré lors du chapitre précédent, notre processus de traduction peut aisément être étendu au cadre GDL-II et plus particulièrement au cadre des jeux à information partagée qu'exprime un fragment PSPACE de GDL-II. La question qui se pose naturellement est de savoir si MAC-UCB peut être étendu directement en MAC-UCB-II sur le SCSP obtenu via le processus de traduction de GDL-II à SCSP.

Afin de répondre à cette question, il est nécessaire de vérifier si le fragment de SCSP identifié suite à la traduction de GDL en SCSP est conservé par la traduction de GDL-II en SCSP. Sachant que le processus de traduction génère toujours un $\mu\texttt{SCSP}_t$ pour chaque tour t de jeu impliquant un sous-ensemble distinct de contraintes de C, il suffit de s'intéresser à la décomposition de l'ensemble des variables afin de vérifier que l'ordonnancement des variables respecte toujours l'ordre imposé par un

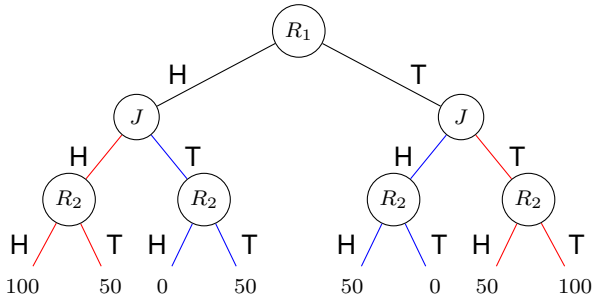


FIGURE 4.3 – L'arbre de recherche obtenu par MAC−UCB−II sur le jeu du « hidden matching pennies » .

SCSP à un niveau à savoir, le sous-ensemble des variables de décisions avant le sous-ensemble des variables stochastiques.

Chaque $\mu\texttt{SCSP}_t$ dans la séquence est un tuple $(X_t, Y_t, D_t, C_t, P_t, \theta)$, avec $X_t = (\{F_t\}, \omega_t, U_t, \{A_t\}, A_{k+1,t}, \{F_t + 1\})$, D_t et C_t restent inchangés face au $\mu\texttt{SCSP}_t$ générés lors du processus de traduction de GDL à SCSP. Toutefois Y_t et P_t sont modifiés. Y_t est la restriction de Y à la variable stochastique $A_{k+1,t}$ modélisant l'action de l'environnement au temps t et aux variables $\{F_t + 1\}$ modélisant l'état suivant. Quant à P_t, il est décrit comme la restriction de P sur les distributions de probabilités de $A_{k+1,t}$ et de l'ensemble des variables $\{F_t + 1\}$.

Il est donc clair que la décomposition de X_t en deux sous-ensembles ordonnés V_t modélisant les variables de décisions et Y_t modélisant les variables stochastiques est tout à fait envisageable avec $V_t = (\{F_t\}, \omega_t, U_t, \{A_t\})$ et $Y_t = (A_{k+1,t}, \{F_t + 1\})$. Le fragment SCSP identifié suite au processus de traduction est donc conservé et MAC−UCB−II peut être utilisé.

Remarque 4.1 *Bien que MAC−UCB et MAC−UCB−II soient exactement les mêmes algorithmes, Il est toutefois nécessaire de connaître par avance si le formalisme utilisé est GDL ou GDL−II afin que la traduction du programme génère un SCSP équivalent suite à l'ajout du mot-clé sees modifiant la nature stochastique des variables représentant chaque fluent du jeu. De plus, les axes de recherches pour les jeux à information complète et incomplète étant souvent très différents, il est préférable de dissocier notre approche en deux versions pour ces deux classes de jeux afin de prédire de futurs travaux adaptés uniquement à l'une ou l'autre de ces méthodes.*

Exemple 4.2 *La figure 4.3 illustre l'arbre de recherche obtenu directement par le modèle SCSP construit par MAC−UCB−II. Afin de simplifier les notations, nous avons renommé la première pièce de l'environnement R_1, sa seconde R_2 et la pièce du joueur par J. MAC−UCB−II obtient une utilité attendue moyenne de 75 pour le sous-arbre rouge et de 25 pour le sous-arbre bleu. Par conséquent MAC−UCB−II joue toujours de manière à orienter le jeu vers le sous-arbre rouge dépendant de la première pièce révélé par le joueur environnement. Nous pouvons facilement remarquer que la stratégie suivie par MAC−UCB−II (afin de garantir le meilleur score) est de jouer le même côté que la pièce révélée par le joueur environnement. Notons que ce jeu implique un arbre de recherche de petite taille, mais que dans le cas général, les arbres de recherches sont bien plus imposants et MAC−UCB−II ne peut pas complètement l'explorer au cours du temps alloué, dans ce cas comme pour MAC−UCB, UCB est nécessaire afin de simuler l'utilité attendue des états non terminaux.*

Notons juste que le nombre de variables stochastiques étant beaucoup plus important dans un SCSP traduit d'un jeu à information imparfaite à un jeu à information incomplète, MAC−UCB−II se confrontera à des instances bien plus complexes à résoudre suite à la taille des politiques solutions exponentiellement plus grandes.

4.4 Évaluations expérimentales

Au cours de cette section, nous illustrons quelques résultats expérimentaux conduits sur un cluster d'Intel Xeon E5-2643 CPU 3.4 GHz possédant 32 Gb de mémoire RAM sous Linux. MAC−UCB a été implémenté en C++ et n'utilise aucun outil externe.

Dans un premier temps, intéressons nous au dilemme exploration/exploitation rencontré par MAC−UCB et MAC−UCB−II.

4.4.1 Dilemme exploration/exploitation

Dans cette section, pour répondre au dilemme exploration/exploitation, nous cherchons expérimentalement à établir un ratio entre la phase de résolution (MAC) et la phase de simulation (UCB) afin d'obtenir les meilleures performances de MAC−UCB en pratique.

Pour cela, dans un premier temps, nous avons implémenté l'algorithme de l'état de l'art en GGP, à savoir UCT en suivant les spécifications décrites dans sa version multi-joueurs [STURTEVANT 2008]. De plus, dans un souci d'équité, nous avons permis à l'algorithme UCT d'utiliser une mémoire cache lui permettant de connaître par avance si un sous-arbre a été exploré. Puis nous l'avons confronté à MAC−UCB et à MAC−UCB−II en faisant varier le ratio résolution/estimation correspondant au temps en pourcentage alloué à chacune des parties de l'algorithme. Nos expérimentations ont été réalisées, pour MAC−UCB sur l'ensemble des 150 jeux GDL de Tiltyard dans leur dernière version et qui ont été joués au moins une fois et pour MAC−UCB−II sur les 15 mêmes jeux GDL−II représentatif du fragment PSPACE de GDL−II identifié dans le chapitre précédent. Nous avons alloué 60 secondes de temps de pré-traitement ($startclock$) et 15 de temps de délibération ($playclock$).

La figure 4.4 répertorie l'évolution du score moyen obtenu par MAC−UCB et par MAC−UCB−II face à UCT sur l'ensemble des jeux expérimentés respectivement sur le graphique de gauche et sur le graphique de droite. L'axe horizontal représente le ratio de temps alloué à la résolution et l'axe vertical représente le score moyen obtenu par MAC−UCB.

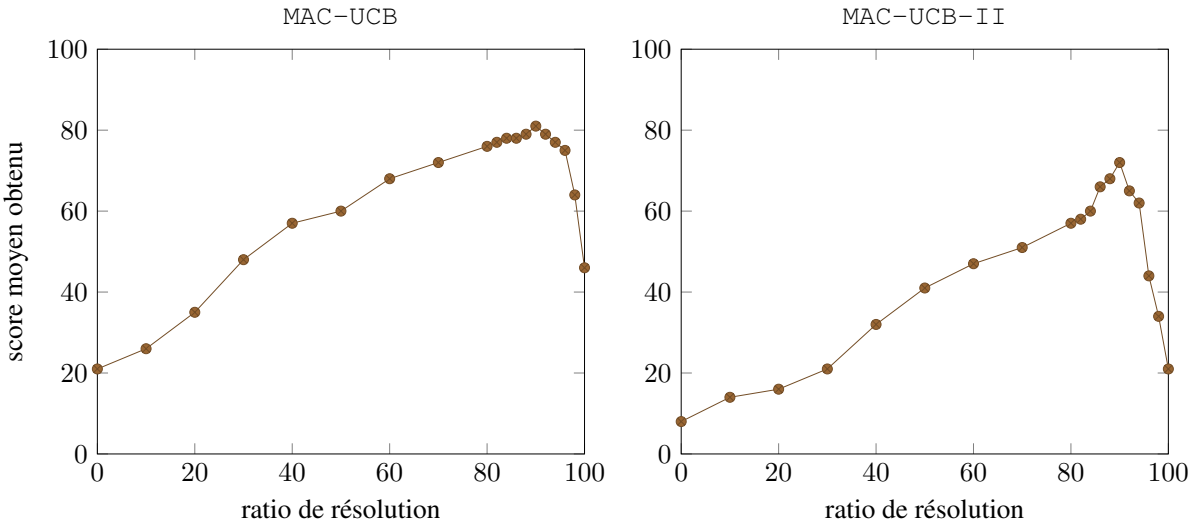


FIGURE 4.4 – Analyse de Sensibilité de MAC−UCB et de MAC−UCB−II.

Suite à ces résultats, il est aisé de remarquer que les meilleures performances sont rencontrées par MAC−UCB et MAC−UCB−II en allouant 90% du temps à la résolution et 10% du temps à la simulation.

Nos prochaines expérimentations s'intéressent à montrer qu'en pratique MAC−UCB obtient de meilleurs résultats que FC−UCB.

4.4.2 MAC−UCB Vs. FC−UCB

Afin de montrer qu'en pratique résoudre chaque μSCSP en décomposant ce dernier en CSP résolu par l'algorithme MAC et en μSCSP plus petit résolu par FC est plus efficace que résoudre directement chaque μSCSP par FC, nous avons réalisé 1.800.000 matchs au travers de 15 jeux GDL différents. Nous avons développé chacun de ces jeux dans le but d'obtenir la plus grande diversité possible (impliquant 2 à 3 joueurs, coopératif ou compétitif, avec ou sans joueur environnement, de plateau ou non, simultané ou non, ...).

Pour chaque jeu, un joueur suit la stratégie déduit par FC−UCB, MAC−UCB ou UCT. Au cours de chaque match le temps de pré-traitement est de 100 secondes et 5.000 matchs sont réalisés en variant le temps de délibération entre les valeurs suivantes : $1s$, $5s$, $10s$, $20s$, $30s$, $40s$, $50s$, $60s$.

Le tableau 4.1 rapporte le pourcentage de victoires obtenues par MAC−UCB (ou de FC−UCB quand MAC−UCB n'est pas utilisé) pour chaque jeu en allouant 30 secondes de temps de délibération. L'écart-type (σ) est également indiqué.

Jeu	**MAC−UCB vs. UCT**	σ	**MAC−UCB vs. FC−UCB**	σ	**FC−UCB vs. UCT**	σ
Awale	56.7 %	1.63 %	77.7 %	1.92 %	43.2 %	2.34 %
Backgammon	68.2 %	5.49 %	84.8 %	6.36 %	47.3 %	5.87 %
Bomberman	65.4 %	6.32 %	75.7 %	5.34 %	58.4 %	5.46 %
Can't Stop	71.7 %	6.43 %	65.9 %	4.87 %	54.7 %	5.34 %
Checkers	61.2 %	2.12 %	76.9 %	1.61 %	57.5 %	1.43 %
Chess	53.8 %	1.75 %	68.5 %	1.76 %	39.4 %	1.75 %
Chinese checkers	55.4 %	8.24 %	78.1 %	7.23 %	32.7 %	6.51 %
Hex	69.7 %	2.45 %	73.2 %	3.24 %	55.3 %	3.12 %
Kaseklau	71.4 %	6.56 %	58.9 %	7.87 %	68.3 %	7.34 %
Orchard	70.2 %	3.41 %	70.2 %	3.45 %	70.0 %	2.45 %
Othello	79.3 %	1.41 %	75.1 %	0.89 %	61.3 %	1.14 %
Pacman	69.1 %	2.73 %	74.1 %	3.12 %	61.8 %	3.78 %
Pickomino	63.4 %	4.89 %	65.9 %	6.10 %	52.1 %	5.34 %
Tictactoe	65.7 %	0.89 %	51.8 %	0.76 %	64.9 %	0.73 %
Yathzee	71.2 %	5.48 %	74.0 %	5.12 %	58.6 %	5.34 %

TABLE 4.1 – Resultats sur plusieurs jeux GDL où $playclock = 30s$

Pour l'ensemble des jeux, MAC−UCB surpasse statistiquement UCT et FC−UCB et le phénomène s'amplifie avec un temps de délibération plus important. La partie gauche du tableau (MAC−UCB vs. UCT) montre que MAC−UCB est particulièrement performant sur les jeux à information imparfaite. En effet, sur le Backgammon, le Bomberman, el Can't Stop, le Kaseklau, le Pacman, le Pickomino et le Yathzee, MAC−UCB gagne avec un nombre de victoires moyen supérieur à 70%. Concernant les jeux à information complète, l'écart-type est plus faible car MAC−UCB ne peut être avantagé par le filtrage stochastique que la partie UCB lui offre via l'exploration complète de sous-arbres du jeu.

Il est important de souligner la spécificité de deux jeux : le `Chinese Checkers` et l'`Orchard`. Pour le premier, impliquant trois joueurs, l'un est controlé par `MAC-UCB` ou `FC-UCB` et les deux autres par `UCT` ou `FC-UCB`. Nous observons ici que même si `MAC-UCB` gagne avec une moyenne de 55% contre deux joueurs `UCT`, l'écart type est important ($> 8\%$). C'est pourquoi, il ne nous est pas possible de confirmer que `MAC-UCB` soit assez efficace pour surpasser les deux adversaires `UCT`. Concernant l'`Orchard` impliquant un jeu coopératif, nous présentons le nombre de victoires moyen pour chaque joueur. Ici, la performance des algorithmes `MAC-UCB` et `UCT` est d'environ 70%. Notons au passage que ce score est maximal si la stratégie optimale est suivie.

Pour la partie centrale du tableau (`MAC-UCB` vs. `FC-UCB`), il est clair que `MAC-UCB` domine `FC-UCB`. Ainsi, la technique de filtrage plus intense utilisée par `MAC-UCB` porte ses fruits : la propriété d'arc cohérence maintenue par cet algorithme possède un impact significatif sur l'efficacité de résolution de chaque μ`SCSP`. Finalement, la partie droite du tableau (`FC-UCB` vs. `UCT`) nous indique que bien que `FC-UCB` surpasse `UCT` sur quelques jeux possédant peu de contraintes et de variables, `UCT` reste victorieux dans la majorité des cas. Par conséquent, à la lumière de ces résultats, l'efficacité de `MAC` (couplée à `UCB`) est cruciale afin de rapidement déterminer les politiques solutions.

4.4.3 `MAC-UCB` face aux meilleures approches `GDL`

Évolution du nombre de victoires

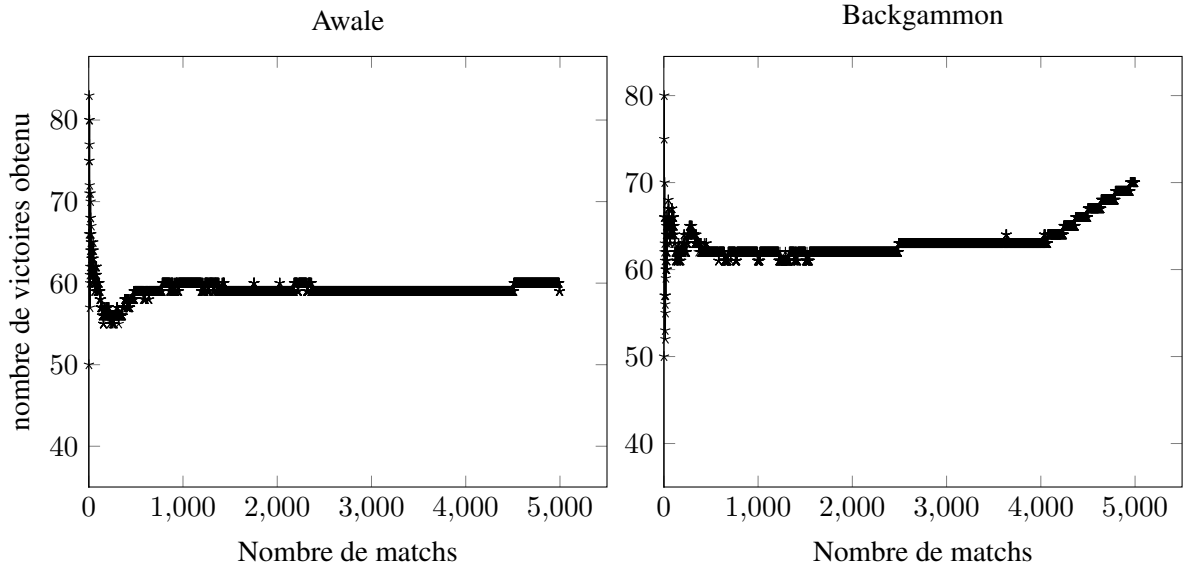


FIGURE 4.5 – Évolution du nombre de victoires de `MAC-UCB` contre `UCT`, avec $playclock = 30s$

La figure 4.5 rapporte l'évolution du nombre de victoires de `MAC-UCB` contre `UCT` où 30 secondes de temps délibération sont allouées au travers de 5.000 matchs différents. La figure de gauche se concentre sur l'`Awale`, un jeu à information complète, ici l'évolution est globalement constante et implique un écart-type de 1.41%. En effet, dans les jeux à information complète, le filtrage est uniquement réalisé par `MAC` sur la partie `CSP` du problème. Cependant la figure de droite se focalise sur le `Backgammon`, un jeu à information imparfaite où la performance de `MAC-UCB` est meilleure et augmente significativement avec le nombre de matchs réalisés. Ceci peut être expliqué par le nombre croissant de sous-arbres complètement explorés via `UCB`, dont la mémoire cache est exploitée de plus en plus au fil des matchs.

Le même phénomène peut être observé pour les autres jeux stochastiques.

Game	1 s	5 s	10 s	20 s	30 s	40 s	50 s	60 s
Awale	38.6 %	43.3 %	47.4 %	**51.6** %	56.7 %	57.9 %	61.3 %	63.0 %
Backgammon	**53**.1 %	58.1 %	61.4 %	65.8 %	68.2 %	71.3 %	77.0 %	79.6 %
Bomberman	**58.3** %	60.0 %	60.2 %	64.7 %	65.4 %	70.1 %	72.4 %	75.6 %
Can't Stop	**50.3** %	54.7 %	59.9 %	62.2 %	71.7 %	74.9 %	76.3 %	78.9 %
Checkers	43.4 %	49.6 %	**51.4** %	56.8 %	61.2 %	67.2 %	71.3 %	75.7 %
Chess	34.0 %	39.3 %	46.1 %	49.9 %	**53.8** %	56.4 %	57.6 %	60.8 %
Chinese Checkers	27.4 %	35.5 %	43.7 %	**50.7** %	55.4 %	59.1 %	63.2 %	65.4 %
Hex	**54.7** %	56.2 %	58.5 %	67.4 %	69.7 %	71.4 %	71.9 %	72.5 %
Kaseklau	**63.6** %	65.2 %	68.3 %	70.2 %	71.4 %	73.2 %	74.7 %	75.1 %
Orchard	**65.3** %	68.5 %	69.9 %	70.2 %	70.2 %	70.1 %	70.2 %	70.2 %
Othello	**61.9** %	64.0 %	70.6 %	75.8 %	79.3 %	82.0 %	84.2 %	84.9 %
Pacman	**64.4** %	66.2 %	67.1 %	67.9 %	69.1 %	69.4 %	69.8 %	70.5 %
Pickomino	**52.4** %	55.3 %	58.0 %	61.1 %	63.4 %	65.8 %	66.1 %	68.6 %
Tic-Tac-Toe	**63.6** %	64.4 %	64.9 %	65.4 %	65.7 %	65.9 %	65.8 %	65.4 %
Yathzee	43.5 %	**50.1** %	53.7 %	64.3 %	71.2 %	75.2 %	77.1 %	78.9 %

TABLE 4.2 – Les victoires de MAC−UCB vs. UCT avec différents $playclock$ sur 5.000 matchs

MAC−UCB avec différents $playclock$

Le tableau 4.2 décrit le pourcentage de victoires de MAC−UCB contre UCT avec différents temps de délibérations, l'écart de performance entre MAC−UCB et UCT augmente avec l'augmentation de ce temps. Cet écart peut être expliqué par la capacité de MAC à résoudre un nombre important de μSCSPs composant le SCSP équivalent, lui permettant ainsi de simplifier la tâche d'exploration que réalise UCB. Pour le Tic−Tac−Toe impliquant l'horizon le plus faible, l'ensemble de l'arbre de jeu est exploré dès que $playclock \geq 5s$. De même pour de petits jeux comme le Bomberman ou le Pacman, MAC−UCB gagne aisément dès que $playclock \geq 1s$. De manière opposée, pour des jeux associés à un arbre de recherche plus important comme l'Awale, le Chess ou le Chinese Checkers, MAC−UCB est vaincu par UCT quand le temps de délibération est trop faible (typiquement quand il est inférieur à $10s$). Toutefois, MAC−UCB obtient de meilleurs résultats pour les jeux stochastiques en exploitant le filtrage stochastique qu'induit UCB. Notamment pour l'Orchard, la stratégie optimale est atteinte en utilisant que $20s$ de temps de délibération.

Mise en pratique

Enfin, afin de confronter MAC−UCB aux dernières approches proposées dans le cadre GGP apportant les meilleures performances, nous avons réalisé une compétition entre différents programmes-joueurs basés sur ces approches. Le premier d'entre eux est le même que précédemment est correspond à l'algorithme de l'état de l'art de GGP pour les jeux à information complète : UCT. Le second programme-joueur est SANCHO dans sa version 1.61c [24] élaboré par S. Draper et A. Rose et champion international

24. SANCHO 1.61c : http://sanchoggp.github.io/sancho-ggp/

de la compétition GGP de 2014 en utilisant un ensemble de techniques MCTS via un réseau de *propnet*. Le troisième programme suit l'algorithme GRAVE (algorithme 1.2 [CAZENAVE 2015]) qui pour rappel implémente une généralisation de la méthode RAVE adaptée au cadre GGP. Finalement, le dernier programme-joueur auquel, MAC−UCB est confronté se base sur l'algorithme CFR [SHAFIEI *et al.* 2009], une implémentation de la technique *CounterFactual Regret* utilisée dans les jeux à information partielle et détaillée dans le chapitre 1.

Remarque 4.2 *Il est important d'indiquer que nous n'avons pas comparer MAC−UCB (et plus particulièrement MAC−UCB−II) à d'autres approches adaptées pour GDL−II, car mis à part CFR, il ne s'agit que de programmes-joueurs indisponibles librement. Concernant l'approche HyperPlay-II décrite dans le chapitre 2 (qui semble être la meilleure approche GDL−II actuellement), celle-ci ne propose pas une implémentation accessible via la publication associée et ne pouvait être disponible lors de nos travaux pas ses auteurs. L'approche CFR est alors considérée comme la meilleure approche GDL−II disponible.*

Nos expérimentations se sont portées sur le même ensemble de jeux que dans le chapitre précédent portant sur l'ensemble des jeux Tiltyard dans leurs dernières versions et qui ont été joués au moins une fois (à savoir 42 jeux à un joueur et 109 jeux multijoueurs). Pour l'ensemble des jeux GDL, 100 matchs sont réalisés entre MAC−UCB et chaque algorithme pour un total de 184.600 où lors de chaque match, le temps de prétraitement et de délibération est fixé à la dernière compétition GGP en date : $startclock$ = $180s$ et $playclock$ = $15s$. Finalement, afin de garantir l'équité, les rôles que représentent chaque approche GGP au cours de chaque match sont échangés après chaque match.

Résultats sur des jeux à un joueur

Le tableau 4.3 rapporte le score maximal atteint par chaque algorithme sur chacun des jeux à un joueur. Ici les valeurs indiquées en rouge indiquent que l'algorithme n'a pas atteint le score optimal que propose le jeu.

Dans une grande partie de ces instances, MAC−UCB et les quatre autres programmes-génériques atteignent le score optimal en résolvant complètement chaque puzzle. Toutefois sur certains jeux proposant un arbre de recherche très important, comme les problèmes des reines impliquant 16 ou 31 d'entre elles, le *Hidato* à 37 hexagones, le *nonogram* de taille 10×10 ou encore le *Rubik's Cube*, aucun algorithme ne réussit à atteindre le score optimal (de 100) dans chacun de ces jeux.

MAC−UCB réussit à compléter certains puzzles que d'autres approches ne parviennent pas à résoudre comme le problème des 12 reines (où UCT et SANCHO obtiennent 0), le *Futoshiki* de taille 6×6, le jeu des dames chinoises en solitaire (où UCT et CFR n'obtiennent pas le score maximal) mais également sur certains instances de *Sudoku* de grade élevé. De plus, même si MAC−UCB ne réussit pas à le résoudre, il obtient le meilleur score sur le jeu du *Rubik's Cube* (égalisant GRAVE) par contre, il ne réussit pas atteindre le score maximal (de 84) sur l'instance *Mine Clearing (small)* bien que GRAVE y parvienne.

Résultats de MAC−UCB sur des jeux multijoueurs

Les tableaux 4.4, 4.5 et 4.6 répertorient le score moyen obtenu par MAC−UCB contre chacun des autres programmes-joueurs. Par exemple, la ligne 2, colonne 2 du tableau 4.4 indique que MAC−UCB a remporté sur les 200 matchs réalisés contre UCT sur le jeu Amazons de taille 10×10 un score moyen de 87.0. Toutes les valeurs indiquées en rouge dans ces trois tableaux correspondent à l'obtention par MAC−UCB d'un score moyen inférieur à la moyenne à savoir au score optimal divisé par le nombre de joueurs (exemple : pour un jeu à 3 joueurs proposant un score optimal de 100, si MAC−UCB obtient moins de $\frac{100}{3} \sim 33.3\%$, la valeur est indiquée en rouge).

Jeu	MAC-UCB	UCT	CFR	GRAVE	Sancho
6 Queens Puzzle ug	100	100	100	100	100
8 Queens Puzzle lg	100	100	100	100	100
8 Queens Puzzle ug	100	100	100	100	100
12 Queens Puzzle ug	100	0	100	100	0
16 Queens Puzzle ug	0	0	0	0	0
31 Queens Puzzle lg	0	0	0	0	0
Asteroids	100	100	100	100	100
Cell Puzzle	100	100	100	100	100
Chinese Checkers 1P	100	100	100	100	100
Coins	100	100	100	100	100
Eight Puzzle	100	100	100	100	100
European Peg Jumping	90	90	90	90	90
Futoshiki 4x4	100	100	100	100	100
Futoshiki 5x5	100	100	100	100	100
Futoshiki 6x6	100	0	0	0	0
Hidato (19 hexes)	100	100	100	100	100
Hidato (37 hexes)	0	0	0	0	0
Knight's Tour	100	100	100	100	100
Knight's Tour Large	100	100	100	100	100
Lights On	100	100	100	100	100
Lights On Parallel	100	100	100	100	100
Lights On Simult-4	100	100	100	100	100
Lights On Simultaneous	100	100	100	100	100
Lights On Simultaneous II	100	100	100	100	100
Lights Out	100	100	100	100	100
Max knights	100	100	100	100	100
Maze	100	100	100	100	100
Mine clearing (small)	78	78	78	84	78
Nonogram 5x5	100	100	100	100	100
Nonogram 10x10	0	0	0	0	0
Peg Jumping	100	100	100	100	100
Rubik's Cube	50	35	30	50	45
Snake Parallel	100	100	100	100	100
Solitaire Chinese Checkers	100	70	90	100	90
Sudoku Grade 1	100	100	100	100	100
Sudoku Grade 2	100	100	100	100	100
Sudoku Grade 3	100	100	100	100	100
Sudoku Grade 4	100	100	100	100	100
Sudoku Grade 5	100	0	100	100	100
Sudoku Grade 6E	100	0	100	100	100
Sudoku Grade 6H	100	0	0	100	100
Untwisty Complex 2	100	100	100	100	100

TABLE 4.3 – Résultats sur chaque jeu GDL à un joueur

Majoritairement MAC-UCB surpasse l'ensemble des autres approches expérimentées et remporte un score supérieur à la moyenne. Particulièrement contre UCT et CFR où le score obtenu est parfois supérieur à 80% (ex : reversi et Skirmish variant) voire 90% (ex : *Beat-Mania*). Quelques exceptions apparaissent tout de même, comme par exemple contre CFR lors des instances checkers small/tiny ou le Tic-Tac-Toe [large] et contre UCT sur le Chicken Tic-Tac-Toe ou le Snake 2P. Toutefois, ces instances impliquent pour la plupart un arbre de recherche peu important et le score moyen obtenu par MAC-UCB est proche de la moyenne (50) ce qui indique que les deux algorithmes ont certainement explorés entièrement l'arbre dans la plupart des cas.

Sancho et GRAVE semblent plus performant que UCT et CFR contre MAC-UCB mais restent toutefois moins compétitifs que notre algorithme. Certains instances démontrent la supériorité notable de MAC-UCB sur ces deux approches comme sur le Knight Fight ou le Connect Four Larger contre Sancho et sur le Speed Chess contre GRAVE. Quelques instances font tout de même défaut à MAC-UCB comme par exemple le Amazons Torus 10×10 contre GRAVE avec un score moyen de 46.1% et *Breakthrough suicide* contre Sancho avec 48.1% et Sheep and Wolf avec 47.2%.

Notons également les instances Iterated Prisonner's Dilemma et The Centipede Game où MAC-UCB obtient un score toujours inférieur à la moyenne, ceci s'explique par le simple fait que ces jeux ne sont pas à somme nulle et que le score obtenu est semblable à celui des autres approches car ils impliquent des arbres également faibles. De même les trois instances Guess two thirds of the average impliquant entre 2 à 6 joueurs ne sont pas significatives car elles demandent simplement en un tour de deviner la valeur la plus proche d'une autre valeur sans réelle stratégie envisageable.

4.4.4 MAC-UCB-II face aux meilleures approches GDL-II

Intéressons nous maintenant aux performances de MAC-UCB-II contre toutes les approches énoncées précédemment sur le même ensemble de 15 jeux GDL-II traduits en SCSP dans le chapitre précédent. Les mêmes temps de pré-traitements et de délibérations sont utilisés ($startclock$ = $180s$ et $playclock$ = $15s$) mais 1.000 matchs par jeu (pour un total de 165.000 matchs) sont réalisés entre MAC-UCB-II et chaque approche afin de circonvenir à la notion de chance pouvant apparaître dans les jeux impliquant un environnement.

Notons que Sancho n'est pas adapté aux jeux GDL-II, c'est pourquoi sur les jeux à information imparfaite impliquant un environnement, il considère ce dernier comme un joueur à part en entière et ne fonctionne pas pour les jeux à information incomplète.

Résultats sur des puzzles à information incomplète

Le tableau 4.7 rapporte les résultats des différentes approches sur les jeux à information incomplète n'impliquant qu'un seul joueur. Pour ces jeux, MAC-UCB-II est la meilleure approche sur l'ensemble des jeux sauf sur les instances *GuessDice* et *Wumpus*. Cependant l'instance *GuessDice* n'est pas significative car il n'existe aucune stratégie permettant de gagner, le jeu consistant uniquement à deviner la valeur du dé lancé par l'environnement en un seul tour. Notons également que CFR et MAC-UCB-II obtiennent un score similaire sur le *MasterMind*.

Résultats sur des jeux multijoueurs à information imparfaite

Le tableau 4.8 rapporte les résultats des différentes approches sur les jeux multijoueurs à information imparfaite. Les quatre approches expérimentées ici sont vaincues par MAC-UCB-II avec un score moyen supérieur à 60% contre SANCHO, UCT et CFR et avec 50% contre GRAVE.

Jeu	UCT	CFR	GRAVE	Sancho
Amazons 8x8	85.2	80.3	56.0	58.6
Amazons 10x10	87.0	81.3	53.6	57.4
Amazons Suicide 10x10	69.1	68.3	71.1	72.4
Amazons Torus 10x10	62.7	73.3	46.1	52.6
Battle	91.4	75.9	71.1	68.4
Battlebrushes 4P	55.1	51.2	48.8	49.0
Beat-Mania	100	95.9	85.7	74.6
Bidding Tic-Tac-Toe	77.1	64.0	66.4	58.8
Bidding Tic-Tac-Toe w/ 10 coins	80.6	58.0	62.3	61.0
Blocker	76.1	72.0	56.1	48.2
Bomberman	58.3	55.1	56.2	51.4
Breakthrough	94.3	86.7	78.1	72.3
Breakthrough small	54.1	54.0	51.9	49.4
Breakthrough small with holes	53.7	54.4	50.3	52.1
Breakthrough with holes	91.1	84.9	79.4	70.3
Breakthrough with walls	92.0	85.7	78.1	71.3
Breakthrough suicide	68.8	72.3	54.4	48.1
Breakthrough suicide small	53.4	54.1	47.0	49.2
Cephalopod micro	63.3	55.4	62.8	61.1
Checkers	88.4	76.6	73.1	67.4
Checkers must-jump	94.2	76.8	70.2	69.1
Checkers on a Barrel without kings	94.1	78.7	75.2	68.2
Checkers on a Cylinder, Must-jump	90.1	83.1	77.3	65.1
Checkers small	55.1	48.5	50.2	51.6
Checkers tiny	54.6	48.4	49.3	50.1
Checkers with modified goals	89.3	77.8	72.4	68.3
Chess	58.1	82.1	70.8	53.4
Chicken Tic-Tac-Toe	49.2	50.7	50.3	49.8
Chinese Checkers v1 2P	60.8	63.4	62.7	56.4
Chinese Checkers v1 3P	56.2	42.8	43.6	40.2
Chinese Checkers v1 4P	49.6	40.7	37.8	36.4
Chinese Checkers v1 6P	49.4	42.8	46.3	40.7
Chinook	81.2	74.6	75.3	68.9
Cit-Tac-Eot	62.0	51.2	47.4	51.4
Connect Five	67.2	60.1	63.8	64.9
Connect Four	65.0	60.4	62.1	61.2
Connect Four 3P	74.1	64.2	57.5	51.2
Connect Four 9x6	89.1	69.5	73.4	56.2
Connect Four Large	80.4	76.2	64.4	68.4
Connect Four Larger	76.3	83.2	50.8	72.4
Connect Four Simultanenous	76.4	62.4	64.9	67.4
Connect Four Suicide	67.4	62.4	52.9	56.8
Copolymer 4-length	75.1	62.4	54.8	56.2

TABLE 4.4 – Résultats de MAC-UCB sur chaque jeu GDL (A à C)

Jeu	UCT	CFR	GRAVE	Sancho
Dollar Auction	82.5	82.3	82.4	82.4
Dots and Boxes	78.2	79.2	82.3	69.3
Dots and Boxes Suicide	69.3	71.4	81.6	56.4
Dual Connect 4	62.3	65.6	61.4	63.2
English Draughts	69.1	78.1	55.3	51.4
Escort-Latch breakthrough	66.8	61.3	64.5	62.3
Free for all 2P	59.4	54.2	53.1	54.1
Free for all 3P	48.7	47.6	45.6	46.6
Free for all 4P	52.3	51.6	53.2	48.6
Free for all zero sum	67.5	63.9	61.4	58.4
Ghost Maze	74.6	67.7	69.1	70.1
Golden Rectangle	80.2	74.6	71.5	74.2
Guess two thirds of the average 2P	49.5	50.4	50.2	49.8
Guess two thirds of the average 4P	24.9	25.1	25.6	24.3
Guess two thirds of the average 6P	16.2	16.9	17.1	16.6
Hex	81.4	76.4	63.9	64.2
Hex with pie	84.5	77.2	75.4	79.2
Iterated Chicken	71.30	64.5	67.3	66.2
Iterated Coordination	73.5	70.1	71.3	68.4
Iterated Prisoner's Dilemma	43.2	46.3	42.3	39.2
Iterated Stag Hunt	70.6	68.5	69.1	65.2
Iterated Tinfoll Hat Game	69.4	68.5	64.2	65.0
Iterated Ultimatum Game	56.4	60.3	62.3	58.4
Kalah 5x2x3	84.6	85.2	84.2	81.0
Kalah 6x2x4	82.1	84.2	86.1	82.6
Knight Fight	84.2	76.1	78.1	76.2
Knight Through	87.2	76.2	77.2	77.1
Majorities	84.2	79.5	72.1	74.6
Nine-Board Tic-Tac-Toe	53.5	55.1	51.9	53.6
Number Tic-Tac-Toe	51.2	49.6	49.2	50.1
Pacman 2P	74.6	72.5	70.6	73.6
Pacman 3P	72.4	75.6	71.3	71.3
Pawn Whopping	74.1	73.1	70.2	64.6
Pawn-to-Queen	61.2	54.2	61.4	58.2
Pentago	86.2	81.3	79.4	67.4
Pentago Suicide	73.4	64.3	68.1	61.3
Quarto	73.4	65.4	71.4	74.1
Quarto Suicide	61.4	58.4	63.2	63.7
Qyshinsu	80.3	74.6	70.3	72.0
Reversi	80.3	86.2	51.3	54.2
Reversi Suicide	86.4	85.7	57.4	53.2

TABLE 4.5 – Résultats de MAC-UCB sur chaque jeu GDL (D à R)

Jeu	UCT	CFR	GRAVE	Sancho
Sheep and Wolf	59.5	53.4	52.1	47.2
Shmup	75.1	66.2	50.8	51.4
Skirmish variant	89.5	86.4	85.2	81.3
Skirmish zero-sum	63.4	77.0	63.1	59.6
Snake 2P	49.8	50.3	50.1	49.7
Snake Assemblit	74.1	70.2	64.2	64.3
Speed Chess	89.4	84.2	87.1	86.3
The Centipede Game	12.3	13.6	20.1	6.4
Tic-Tac-Chess 2P	56.1	48.5	53.6	50.2
Tic-Tac-Chess 3P	39.6	36.4	31.1	38.4
Tic-Tac-Chess 4P	74.2	67.2	69.1	70.2
Tic-Tac-Heaven	63.4	68.4	69.1	64.3
Tic-Tac-Toe	50.3	49.7	50.1	50.6
Tic-Tac-Toe 3P	54.6	54.1	53.7	54.1
Tic-Tac-Toe 9 boards with pie	81.4	76.4	73.2	75.4
Tic-Tac-Toe Large	50.7	49.3	49.7	49.2
Tic-Tac-Toe LargeSuicide	51.3	50.4	49.2	49.6
Tic-Tac-Toe Parallel	50.0	50.0	50.0	50.0
Tic-Tac-Toe Serial	50.0	50.0	50.0	50.0
Tic-Tic-Toe	50.0	50.0	50.0	50.0
Tron 10x10	81.3	70.1	68.4	78.5
TTCC4 2P	79.4	82.7	50.1	53.9
War of Attrition	57.5	55.4	48.0	48.4
With Conviction !	50.0	50.0	50.0	50.0

TABLE 4.6 – Résultats de MAC-UCB sur chaque jeu GDL (T à Z)

Jeu	MAC-UCB-II	UCT	CFR	GRAVE
GuessDice	15.3	15.4	16.2	16.2
MasterMind	67.5	53.2	67.4	61.3
Monty Hall	66.1	62.5	63.6	63.7
Vaccum Cleaner Random	63.5	34.9	47.4	57.8
Wumpus	33.5	39.8	43.7	53.3

TABLE 4.7 – Résultats de MAC-UCB-II sur les jeux GDL-II à un joueur

Jeu	UCT	CFR	GRAVE	Sancho
Backgammon	70.5	66.7	54.6	97.6
Can't Stop	73.1	68.1	62.7	94.3
Kaseklau	73.9	73.4	57.3	83.3
Pickomino	66.2	61.6	59.4	74.1
Yahtzee	76.1	73.1	52.8	73.0

TABLE 4.8 – Résultats de MAC-UCB-II sur les jeux GDL-II à information imparfaite

Jeu	UCT	CFR	GRAVE
Pacman	59.2	60.1	56.4
Schnappt Hubi	71.0	56.4	56.3
Sheep & Wolf	67.8	57.1	55.1
Tic-Tac-Toe Latent Random 10x10	81.4	80.1	69.5
War (card game)	72.1	69.6	66.5

TABLE 4.9 – Résultats de MAC-UCB-II sur les jeux GDL-II à information partagée

Jeux à information partagée

Finalement, les derniers jeux expérimentés correspondent aux jeux à information partagée. Par exemple pour l'instance *Tic-Tac-Toe Latent Random* 10×10 impliquant 2 joueurs et un environnement. Dans ce jeu, l'environnement place une croix ou un rond sur l'une des cases libres à son tour et les joueurs ne peuvent observer les conséquences des actions de l'environnement qu'au moment où l'un des joueurs essayent de placer son symbole sur l'une des cases marquées par ce dernier. *Schnappt Hubi* et *Sheep & Wolf* sont des jeux coopératifs, où l'ensemble des joueurs partagent leurs observations dans le but de vaincre l'environnement.

Dans cette classe de jeux, MAC-UCB-II gagne avec en moyenne un score supérieur à 55% contre chaque opposant et avec un score moyen proche (voir supérieur) de 70% au *Tic-Tac-Toe Latent Random* 10×10 et au *War (card game)*.

Au cours de la prochaine section, nous cherchons à confirmer ces résultats dans le cadre de compétitions GGP par l'implémentation de MAC-UCB via un programme joueur générique dénommé WoodStock.

4.5 WoodStock : un programme-joueur dirigé par les contraintes

Au cours des sections précédentes, une nouvelle approche GGP utilisant le paradigme de résolution de réseaux de contraintes stochastiques, dénommé MAC-UCB, a été mise en avant. Expérimentalement, cette approche a été montrée efficace et surpassant de nombreuses autres approches GGP. Toutefois, afin de confirmer l'efficacité de cette méthode, il est nécessaire de la confronter aux autres programmes composant les compétitions GGP.

Remarque 4.3 *Rappelons qu'actuellement les seules compétitions organisées se focalisent sur le formalisme GDL bien que WoodStock permet également la résolution de jeux à information partagée en GDL-II.*

Pour cela, nous avons réalisé le programme joueur générique WoodStock (*With Our Own Developper STOchastic Constraint toolKit*) implémentant l'algorithme MAC-UCB et nous l'avons soumis à la dernière compétition GGP, la *Tiltyard Open* 2015 mais également au tournoi en continu que le serveur Tiltyard propose depuis 2012 et qui regroupe plus de 1.000 programmes joueurs différents se confrontant sur l'ensemble des jeux que propose le serveur.

Par la suite, nous mettons en avant notre programme-joueur que dans le cadre GDL, cette version étant à l'origine des résultats obtenus en compétition.

4.5.1 Protocole de communication

Comme vu dans le chapitre 1, afin de permettre la communication entre un programme-joueur et le *game manager* un protocole de communication dénommé GCL est utilisé lors des compétitions GGP.

À chaque tour, afin que WoodStock puisse prendre connaissance de l'état courant et lui permettre de communiquer l'action qu'il souhaite réaliser, nous avons développé un joueur réseau basé sur le protocole GCL intitulé spy-ggp [25] disponible sous double Licence GNULGPL3 / CeCILL-C. Celui-ci est réalisé via la bibliothèque Python standard et nécessite uniquement l'interprète Python3 pour fonctionner. Il peut être lié à n'importe quelle bibliothèque dynamique (.so, .dll, .dylib en fonction du système utilisé) et modélisant la partie « stratégique » du joueur. Cette bibliothèque peut être écrite dans un autre langage (C, C++, C#, Go, OCaml, ...). Il est très flexible et depuis sa première utilisation en compétition, ce programme s'est montré particulièrement robuste.

Le langage C++ a été choisi pour implémenter la partie « stratégique » de WoodStock dans le but de profiter du cadre de la programmation orienté objet (POO). De plus, l'utilisation de la POO permet de réaliser un programme modulaire qu'il est aisé de maintenir et d'améliorer au fil des futures recherches qu'impliquent nos travaux actuels. Finalement, C++ assure une capacité de calcul bien plus performante que les autres langages objets.

La partie stratégique de WoodStock se compose naturellement de trois parties :

— Une phase de traduction ;
— Une phase de résolution ;
— Une phase de simulation.

4.5.2 Phase de traduction

Pendant le temps de pré-traitement alloué (*start clock*), WoodStock adapte le processus de traduction d'un programme GDL impliquant k joueurs en un SCSP dans le but que ce dernier puisse être utilisé le plus efficacement en compétition. Tout d'abord, suite à la réception du programme GDL, WoodStock génére un *template* dénommé μSCSP_t représentatif d'un tour quelconque du jeu GDL. Ce patron est une traduction du programme GDL sans y inclure les règles construites via le prédicat « init » modélisant l'état initial du jeu.

Dans un premier temps, nous appliquons la méthode de Lifschitz et Yang [LIFSCHITZ & YANG 2011] sur le programme GDL afin d'éliminer les fonctions et de permettre une traduction plus aisée. Enfin, WoodStock génère chaque variable du programme SCSP : une variable $terminal$ modélisant si un état est terminal ou non, k variables $score_j$ afin de modéliser le score de chaque joueur j, k variables $action_j$ pour modéliser l'action de chaque joueur j et deux variables pour chaque fluent du jeu, respectivement une au temps courant t et une seconde au temps suivant $t+1$.

La prochaine étape est l'extraction des domaines. Le domaine de $terminal$ est booléen et le domaine de chaque variable $score_j$ est modélisé par la valeur entière apparaissant dans chaque règle « goal » correspondant à chaque joueur j. Finalement pour chaque variable $action_j$ et pour chaque variable modéli-

25. spy-ggp : https://github.com/syllag/spy-ggp

sant un fluent au temps t ou $t+1$, WoodStock exploite les conditions d'une compétition GGP actuelle imposant l'utilisant des règles « input » et « base » composant un programme GDL. Ces derniers permettant de générer rapidement l'univers de Herbrand, il n'est pas nécessaire de calculer l'ensemble des combinaisons des constantes pour obtenir le domaine de chaque variable « fluent ».

La dernière étape est la génération de chaque contrainte. WoodStock commence par créer une contrainte pour chaque règle GDL en associant à chaque contrainte sa portée correspondante suite aux règles de réécriture détaillées dans le tableau 3.2. Quant aux relations des contraintes, elles sont restreintes uniquement aux termes présents dans chaque règle GDL correspondante. La présence des mot-clés « distinct » et « not » sont utilisés pour effectuer un premier filtrage de chaque relation. Les contraintes obtenues sont en extension et modélisées par des contraintes tables.

WoodStock utilise XCSP 3.0 [26], un format basé sur XML pour représenter les instances obtenues dans le but de fournir de nouvelles instances SCSP et d'en extraire facilement la partie CSP.

Suite à la génération du *template* μSCSP_t, quelques méthodes de pré-traitement ne demandant que très peu de temps y sont appliquées afin de rendre la phase de résolution plus efficace. Les contraintes possédant la même portée sont fusionnées en une seule contrainte en unifiant les relations. Puis, toutes les variables universelles du réseau sont supprimées.

Afin d'obtenir un μSCSP_t à un temps donné t, WoodStock utilise un injecteur. Au temps $t = 0$ représentant l'état initial, l'injecteur correspond à un ensemble de contraintes unaires générées à l'aide des règles « init ». À tout autre temps $t \neq 0$, l'injecteur est construit avec les solutions du micro SCSP correspondant à l'état à l'origine du nouvel état. Grâce à celui-ci, il est possible de projeter la relation de chaque contrainte unaire ajoutée sur le domaine de chaque variable apparaissant dans la portée sur l'ensemble du réseau de contrainte en le restreignant à la seule valeur autorisée par le tuple de la contrainte associée.

Le μSCSP obtenu est alors assez petit pour appliquer l'algorithme SAC sur ce dernier. Suite à cela, les valeurs inconsistantes sont supprimées du domaine des variables. C'est pourquoi, seules les actions légales représentées par les valeurs consistantes des variables $action_j$ de chaque joueur j sont conservées et garantissent la légalité des actions jouées par WoodStock au cours de l'ensemble du jeu.

4.5.3 Phase de résolution

WoodStock nécessite l'utilisation de l'algorithme MAC pour résoudre la partie classique et l'utilisation d'un algorithme stochastique pour la seconde partie. SFC (voir algorithme 2.15) est utilisé pour résoudre la seconde partie. Enfin, l'avantage principal sous-jacent de MAC−UCB est l'utilisation de technique de filtrage sur des contraintes non binaires.

Par conséquent, afin de vérifier la véracité de l'efficacité de MAC−UCB sur l'état de l'art de GGP en compétition, nous avons choisi d'implémenter STR présenté dans l'algorithme 2.6 en associant à chaque contrainte, les quatre structures nécessaires à son implémentation (position[c], currentLimit[c], LevelLimits[c] et valeursGAC). De plus, l'heuristique *dom/ddeg* est utilisé afin de choisir les variables à résoudre par ordre croissant selon le ratio entre la taille du domaine courant et le degré dynamique des variables.

Remarque 4.4 *Cette méthode étant rapide et simple à implémenter, il nous a été possible de rapidement participer aux compétitions GGP sans rencontrer de difficultés à ce niveau. Notons que comme vu dans le chapitre 2, cet algorithme propose des versions optimisées STR2 [*LECOUTRE *2011] et STR3 [*C. et al. *2015] mais imposant une difficulté supplémentaire impliquant un temps d'implémentation plus important. Ces deux algorithmes pourront faire l'objet de futures améliorations pour WoodStock.*

26. XCSP 3.0 : xcsp.org

4.5.4 Phase de simulation

Afin d'être efficace l'algorithme de résolution itératif utilisé (MAC−UCB) doit choisir le plus judicieusement possible le prochain SCSP à un niveau à résoudre. Dans ce but, WoodStock nécessite de réaliser des simulations pour évaluer l'utilité attendue de chaque μSCSP dont la variable $terminal_t$ est assignée à la valeur *fausse* à l'aide de la méthode UCB. Le principal problème est d'atteindre rapidement un état terminal sans résoudre chaque μSCSP rencontré. Heureusement, grâce à la propriété SAC que nous appliquons sur chaque μSCSP_t à chaque temps t, nous obtenons les coups légaux à chaque tour rapidement. Par conséquent, WoodStock peut choisir aléatoirement une séquence d'actions jusqu'à atteindre un état terminal identifié par l'assignation de la valeur $true$ à la variable $terminal_t$.

Comme vu dans la partie expérimentale de MAC−UCB lors d'une analyse de sensibilité entre la partie résolution et la partie simulation, le ratio présentant les meilleurs résultats est de 90% de résolution et 10% de simulation.

Afin d'améliorer l'efficacité des techniques de filtrage de WoodStock une table de hachage de Zobrist [ZOBRIST 1990] est utilisée pour stocker tous les états déjà explorés. En combinant les état terminaux déjà atteints avec le nombre de coups légaux de chaque état, il est possible de déterminer si un sous-arbre a été complètement exploré mais également d'éviter de simuler deux fois une même succession de coups légaux.

Finalement, le mouvement sélectionné par WoodStock correspond à l'action proposant la meilleure utilité espérée dans l'état courant. Notons que pour envisager tout problème de communication pouvant provoquer l'apparition d'un *timeout*, 2 secondes sont conservées pour sélectionner l'action et garantir son envoi au *game manager*.

Au cours de la section suivante, nous présentons les résultats de WoodStock au cours de sa participation à la dernière compétition GGP mais également au cours de la compétition en continue que propose le serveur *Tiltyard*.

4.6 WoodStock en compétition

Actuellement en compétition, WoodStock fonctionne sur un Intel Core i7-4770L CPU 3.50 Ghz associé à 32 Gb de RAM sous Linux. La première participation de WoodStock fut lors de la dernière compétition GGP dénommé *Tiltyard Open* en décembre 2015 organisée par *Sam Schreiber* et *Alex Landau* sur le site Tiltyard [27].

4.6.1 *Tiltyard* Open 2015

Contexte

Cette compétition est réalisée en deux étapes. La première représente la phase de qualification où le temps de pré-traitement est fixé à 120 secondes et la temps de délibération à 15 secondes, la seconde étape représentant la finale propose un temps de pré-traitement de 180 secondes et le même temps de délibération.

Les participants de la phase de qualification jouent sur 10 nouveaux jeux au cours d'un tournoi suisse ; la plupart d'entre eux sont réalisés plusieurs fois. Ils incluent des jeux à 1, 2, ou 4 joueurs et plusieurs d'entre eux sont simultanés.

Les programmes-joueurs participent automatiquement à chaque match sur la base des résultats de ces précédents matchs. Premièrement, les matchs sont réalisés entre joueurs possédant un score courant similaire sur le jeu courant et deuxièmement entre joueurs possédant un score courant similaire sur

27. www.ggp.org/view/tiltyard/matches/#tiltyard_open_20151204_2

l'ensemble de la compétition. À cela s'ajoute un facteur permettant de réduire les matchs répétés entre les mêmes joueurs.

La table 4.10 indique les informations disponibles sur les 10 jeux proposés lors de la phase de qualification associée à un identifiant unique afin de conserver leurs anonymats. 1 à 4 joueurs sont impliqués dans le nombre de matchs prédéterminés pour chaque jeu. Chaque programme-joueur victorieux obtient un nombre de points égal au score qu'il a obtenu à l'issu du match pondéré par différents poids associés à chaque jeu.

Id	#Joueurs	#Matchs	Poids
1	1	1	2
2	2	6	0.5
3	1	2	1
4	2	6	0.5
5	1	1	2
6	2	6	0.5
7	4	4	0.5
8	2	3	1
9	4	4	0.5
10	2	3	1

TABLE 4.10 – Les informations disponibles pour chaque jeu de la phase de qualification

Nous pouvons noter que la phase de qualification avantage les jeux à deux joueurs et que le score maximal pouvant être atteint est de $2,500$.

La finale inclue les quatre programmes-joueurs obtenant les meilleurs scores lors des qualifications et propose un système d'élimination sur plusieurs matchs réalisés par tour.

Résultats de `WoodStock` à l'issue de la compétition

Neuf programmes listés dans la table 4.11 ont participé à la *2015 Tiltyard Open*. Trois d'entre eux, fonctionnant avec un réseau de propositions (*propnet*) et des méthodes Monte Carlo, sont les derniers champions `GGP` de ces dernières années. La même table montre le score obtenu par chaque programme-joueur générique à l'issue de la phase de qualification.

Au cours de cette étape, `WoodStock` a fini second avec un score (1366.250 points) plus élevé que les derniers champions `GGP`. Notons que `Galvanise` et `Sancho`, les deux derniers champions ne sont pas qualifiés car ils ont perdu de nombreux points contre `WoodStock` et `LeJoueur`. `LeJoueur` [MÉHAT & CAZENAVE 2011a] est une implémentation en parallèle de l'ancien champion `GGP Ary` permettant de réaliser de nombreuses simulations. Ce dernier a obtenu la première place de la qualification en utilisant $3,000$ simulateurs et fonctionnant sur un cluster composé de 48 threads. Notons que pour le moment, `WoodStock` fonctionne sur un seul thread, par conséquent, les deux infrastructures ne sont pas similaires et explique pourquoi `LeJoueur` a obtenu un meilleur score.

Pour sa première participation, `WoodStock` s'est qualifié pour la finale au côté de `LeJoueur`, `GreenShell` et `QFWFQ`. Malheureusement, au cours de cette phase, `WoodStock` a rencontré une erreur sur le premier jeu `GDL` et il était dans l'impossibilité d'envoyer une action au *game manager*.

28. Il est nécessaire de mentionner que `GreenShell` est le nouveau pseudonyme utilisé par `TurboTurtle`, le champion 2011 et 2013 de `GGP`.

Rang	GGP player	Score final
1	LeJoueur	1783.750
2	WoodStock	1366.250
3	GreenShell [28]	1351.000
4	QFWFQ	1245.875
5	Sancho	1213.500
6	SteadyEddie	1201.875
7	Galvanise	1185.250
8	Modest	1129.000
9	MonkSaki	1037.813

TABLE 4.11 – Les scores finaux suite à la phase de qualification

Cette erreur était dûe à un problème dans son implémentation. Par conséquent, ces mouvements ont été remplacés par des actions aléatoires et il a perdu les deux premiers matchs. Sur les trois matchs suivants, WoodStock n'a rencontré aucun problème et il les a remportés pour finalement obtenir la troisième position de la compétition GGP. Le tableau 4.12 indique les rangs finaux.

Rang	GGP player
1	LeJoueur
2	GreenShell
3	WoodStock
4	QFWFQ

TABLE 4.12 – Le classement final de la *2015 Open Tiltyard*

4.6.2 Tournoi continu GGP

Suite à sa première participation en compétition GGP, l'erreur d'implémentation rencontrée par WoodStock a été corrigée et il a pu participé à la compétition continu de *General Game Playing* où l'ensemble des programmes-joueurs sont représentés (à ce jour, plus de $1,000$ programmes) jouant sur l'ensemble des jeux que propose le serveur Tiltyard. Ce type de compétition est naturellement plus complexe car le temps de pré-traitement et le temps de délibération sont choisis aléatoirement entre deux matchs et aucun programme-joueur ne peut donc se calibrer en fonction de ce paramètre.

Depuis mars 2016 et après près de $2,000$ matchs, WoodStock est le leader du tournoi et a détrôné Sancho qui était leader de cette compétition depuis plus de trois ans. À ce jour, WoodStock obtient un score moyen supérieur à 76 et le tableau 4.13 indique le classement actuel du tournoi continu GGP.

Les matchs de WoodStock sont accessibles en direct à l'adresse suivante : www.ggp.org/view/tiltyard/players/WoodStock. Il est notamment possible de retrouver le score moyen obtenu par WoodStock sur chaque jeu GDL.

Rang	GGP player	Score Agon
1	WoodStock	225.4
2	Sancho	220.63
3	Galvanise	214.4
4	LabThree	191.71
5	Coursera Quixote	169.77
6	SgianDubh	163.92
7	CloudKingdom	162.23
8	SteadyEddie	159.53
9	Alloy	133.5
10	LeJoueur	120.26

TABLE 4.13 – Le classement actuel (datant du 15 juillet 2016) du tournoi continu GGP

4.7 Conclusion

Au cours de ce travail, nous avons identifié un fragment de la satisfaction de réseaux de contraintes stochastiques correspondant à un fragment du formalisme GDL-II, démontré traitable de part sa complexité (PSPACE), englobant les jeux à information complète, les jeux à information imparfaite et les jeux à information partagée. Basé sur ce fragment, nous proposons un algorithme MAC-UCB pour les jeux GDL et MAC-UCB-II pour les jeux GDL-II permettant d'obtenir efficacement les politiques solutions en combinant une méthode de résolution de réseaux de contraintes n-aire classique (MAC) et une méthode d'échantillonnages de type Monte Carlo (UCB).

Expérimenté dans un contexte GGP sur de nombreux jeux GDL et GDL-II, tout d'abord sur différents temps de délibérations, puis sur l'ensemble des jeux GDL que propose le serveur *Tiltyard* contre les dernières approches GGP proposant les meilleures performances et composant l'état de l'art de GGP, MAC-UCB s'avère dans la plupart des cas, être plus performant que ces dernières que ce soit en GDL ou en GDL-II (à information partagée).

Finalement, afin de confirmer l'efficacité de cette nouvelle approche, nous proposons un nouveau type de programme-joueur dénommé WoodStock dirigé par la satisfaction de réseaux de contraintes via notre algorithme MAC-UCB. Dans le cadre de la compétition *Open Tiltyard* 2015, il obtient la seconde position en phase de qualification et la troisième place en finale. De même, sur le même serveur, WoodStock participe à la compétition continue GGP depuis décembre 2015 et maintient sa place de leader depuis mars 2016 en détrônant Sancho qui l'a possédé depuis trois ans.

Ce travail ouvre la voie à de nombreuses perspectives de recherche en apportant le monde des contraintes au cadre GGP. Tout d'abord, WoodStock implémente MAC-UCB dans un contexte orienté objet et peut être facilement amélioré de par une étude plus poussée des différentes techniques de résolution pouvant substituer l'algorithme STR qu'il utilise actuellement ou encore en renforçant les techniques de pré-traitement et les heuristiques utilisées. De même WoodStock est actuellement séquentiel et ne peut se comparer aux programmes-joueurs génériques implémentés en parallèle, l'étude d'une résolution parallèle des instances μSCSP est une piste de recherche en cours de travaux en plus de la parallélisation de la partie simulation qu'induit UCB.

Un autre intérêt de ce travail est d'établir une borne de confiance plus adaptée qu'UCB aux problèmes de jeux (souvent compétitifs) afin d'obtenir de meilleures heuristiques ou encore d'exploiter la structure graphique induite par un réseau de contraintes afin de déterminer certaines propriétés de jeux et de les

exploiter pour détecter automatiquement de nouvelles heuristiques.

Cette dernière piste peut par exemple permettre de détecter des symétries de jeux ou de stratégies dans les instances auquels nos travaux se confrontent en établissant le lien entre la notion de symétries dans le cadre `CSP`/`SCSP` et le cadre des jeux. Cette dernière notion est l'objet du chapitre suivant.

Chapitre 5

Détection de symétries par les contraintes pour le General Game Playing

Sommaire

5.1 Détection de symétries dans le *General Game Playing* 155
5.1.1 Jeux et symétries 155
5.1.2 Méthode des graphes de règles (méthode RG) 157
5.2 Détection de symétries dans la programmation par contraintes 158
5.2.1 Deux types de symétries 158
5.2.2 Exemple d'application 160
5.3 Symétries de jeux GDL et symétries dans un SCSP 163
5.4 MAC-UCB-SYM 168
5.4.1 Détection en pratique des symétries 168
5.4.2 Profondeur des stratégies minimax 168
5.5 Résultats expérimentaux 169
5.5.1 Détection des symétries entre la méthode RG et les contraintes 169
5.5.2 Étude de la détection de symétries par la méthode RG 170
5.5.3 MAC-UCB-SYM dans le cadre GDL 172
5.5.4 MAC-UCB-II-SYM dans le cadre GDL-II 174
5.5.5 WoodStock et MAC-UCB-SYM, Champion GGP 2016 178
5.6 Conclusion 180

Conceptuellement, tout jeu (déterministe ou stochastique) à horizon fini impliquant deux joueurs et à somme nulle possède une fonction minimax optimale spécifiant le résultat attendu d'un jeu, pour tout état possible, sous la condition que l'ensemble des joueurs jouent parfaitement.

En théorie de tels jeux peuvent être résolus en calculant récursivement la fonction d'utilité dans un arbre de recherche de taille l^d, où l est la largeur du jeu (le nombre d'actions légales par état) et d est la profondeur du jeu (le nombre de mouvements nécessaires pour atteindre un état terminal). Pourtant, même pour des jeux d'une taille modérée, une recherche exhaustive n'est pas envisageable en pratique lors des tournois de *General Game Playing*, dû au temps de délibération très cours imposé à chaque joueur pour déterminer sa prochaine action.

C'est pour cette raison, que l'un des challenges de GGP est de concevoir des techniques génériques d'inférence et d'apprentissage pour réduire efficacement l'espace de recherche des jeux, dont les règles ne sont fournies qu'au début du jeu.

À cette fin, la détection de symétries est une approche d'inférence bien connue en Intelligence Artificielle pour accroître la résolution de tels problèmes combinatoires, en transférant les connaissances apprises en des régions équivalentes de l'espace de recherche. Les jeux représentent un exemple notable car ils impliquent typiquement de nombreux états équivalents et des actions équivalentes. De telles similarités peuvent être exploitées pour transférer la fonction d'utilité entre plusieurs nœuds de l'arbre de recherche : la largeur l de l'arbre peut être réduite en exploitant les actions menant à des résultats attendus similaires et la profondeur d de l'arbre peut également être réduite en reconnaissant des états associés à des valeurs similaires.

Ainsi la plupart des méthodes de détections de symétries sont réalisées en utilisant des algorithmes basés sur des automorphismes de graphes [DARGA *et al.* 2008, MCKAY & PIPERNO 2014]. Le composant principal pour mettre en évidence les symétries de jeux est une structure graphique sur laquelle un groupe de permutation est induit. Idéalement, ce groupe de permutation devrait être aussi près que possible du groupe des symétries minimax, préservant par la même occasion les politiques optimales issues du jeu.

Dans le cadre du *General Game Playing*, les approches de détection de symétries composant l'état de l'art reposent sur un graphe de règles annoté RG (pour *rule graph*) associé à la représentation GDL d'un jeu [KUHLMANN & STONE 2007, SCHIFFEL 2010]. Intuitivement, un graphe de règles contient des nœuds pour représenter les prédicats, les fonctions et les variables du programme GDL, connectés par des arêtes dans le but de capturer la structure des règles. Une fois, le graphe de règles défini en n'utilisant que la représentation GDL en entrée, les méthodes de détections de symétries basées sur ces règles sont applicables sur divers algorithmes de résolution de jeux génériques. Toutefois, le graphe de règles et son groupe de permutations déduit sont fortement dépendants de la syntaxe utilisée pour représenter les jeux. En effet, bien que syntaxiquement différentes règles peuvent posséder la même sémantique, la méthode RG basé sur différentes représentations GDL du même jeu peuvent mener à la détection de différentes symétries. Comme illustré dans [SCHIFFEL 2010], même en ajoutant une seule règle tautologique à la description GDL du jeu, il se peut que plusieurs symétries ne soient pas détectées. De plus, différents termes permettant de décrire des fluents équivalents ou des actions équivalentes peuvent aussi empêcher à certaines symétries d'être détectées. Par conséquent, seul un sous-ensemble restreint de symétries de jeux peut être reconnu à partir de cette méthode, à cause du biais syntaxique imposé par GDL.

Dans ce chapitre, nous proposons une approche alternative pour détecter les symétries dans les jeux stochastiques inspirée des techniques de programmation par contraintes. Comme vu dans le chapitre précédent, le SCSP obtenu d'un programme GDL décrivant un jeu stochastique et les stratégies minimax associées peut être décomposé en une séquence de μSCSP, spécifiant chacun un tour de jeu. Sur la base de cette décomposition, notre méthode de détection de symétries utilise la micro-structure complémentaire associée aux SCSPs à un niveau afin de détecter les symétries de contraintes formant un sous-groupe des symétries minimax. De plus, l'utilisation des micro-structures complémentaires peut aussi être exploité pour détecter les symétries en approximant les stratégies minimax en combinant l'utilisation de techniques de propagations par contraintes standards et l'exploration par bandit.

Nous fournissons une évaluation expérimentale de cette approche en utilisant l'algorithme MAC-UCB décrit dans le chapitre 4 pour résoudre chaque μSCSP. Nous démontrons sur l'ensemble des instances GDL jouées sur le serveur Tiltyard et sur 5 jeux GDL-II avec information complète que MAC-UCB-SYM, une version de MAC-UCB couplée à notre approche de détection de symétries, surpasse significativement les algorithmes standards MCTS couplés à la détection de symétries basée sur un graphe de règles. Pour finir, WoodStock, le programme-joueur implémentant MAC-UCB a été étendu à MAC-UCB-SYM lors de la compétition internationale de *General Game Playing* 2016. Ce dernier a terminé premier de la

compétition devenant champion international.

Ce chapitre s'organise comme suit. Dans une première section, nous définissons la notion de symétries dans un jeu stochastique et introduisons la méthode RG de détection de symétries et ses limites. Par la suite en section 2, nous rappelons les méthodes de détections de symétries utilisées dans la programmation par contraintes. Au cours de la section 3, nous établissons le lien entre symétries de contraintes et symétries de jeux. Le cadre pratique utilisé par MAC-UCB-SYM est mis en avant dans la section 4. Finalement, la section 5 expose l'ensemble des résultats expérimentaux réalisés et la mise en pratique de MAC-UCB-SYM par WoodStock lors de la compétition internationale de *General Game Playing* 2016 (IGGPC'16) lui permettant de devenir le champion GGP 2016.

5.1 Détection de symétries dans le *General Game Playing*

5.1.1 Jeux et symétries

Reprenons la sémantique d'un jeu GDL + *random* illustrée dans dans la définition 1.32. Rappelons toutefois que nous nous intéressons uniquement aux jeux stochastiques à information complète et à horizon fini. La sémantique d'un jeu stochastique G est décrite dans la définition 1.18. L'ensemble de tous les états dans G atteignables depuis s_0 sont désignés par S_G.

Une stratégie pure stationnaire (voir définition 1.21) associe des états à des actions. Au cours de ce chapitre, nous nous intéressons particulièrement aux stratégies minimax pour lesquelles la fonction de valeur associée à chaque joueur j est :

$$\mathcal{V}_j^*(s) = \begin{cases} \max_{a_j} \min_{\boldsymbol{a}_{-j}} Q_j^*(s, \boldsymbol{a}) & \text{si } s \notin S_{ter} \\ u_j(s) & \text{sinon} \end{cases} \tag{5.1}$$

où $Q_j^*(s, \boldsymbol{a}) = \sum_{s' \in S} P(s' \,|\, s, \boldsymbol{a}) \mathcal{V}_j^*(s')$ et où a_j est dans $L_j(s)$ et $\boldsymbol{a}_{-j}$ est compris dans la projection de $\boldsymbol{L}(s)$ sur $J \setminus \{j\}$.

La stratégie *minimax* est connue pour être optimale pour la classe importante des jeux stochastiques à deux joueurs tour par tour et à somme nulle [CONDON 1992] donnée par $k = 2$, $u_1(s) = -u_2(s)$ pour tout état terminal $s \in S_{ter}$ et $L_1(s) = \{\top\}$ ou $L_2(s) = \{\top\}$ pour chaque état $s \in S_G$, où $\top$ est l'action *noop* (sans effet sur l'état du jeu).

En pratique, il est généralement difficile d'évaluer la fonction de valeur minimax. Une approche usuelle pour le permettre et d'approximer $\mathcal{V}_j^*$ par une valeur minimax limitée en profondeur. Une fois la profondeur d atteinte, une fonction heuristique d'évaluation est appliquée [LANCTOT *et al.* 2013].

Soit une fonction heuristique $\hat{\mathcal{V}}_j : S \to \mathbb{R}$ tel que $\hat{\mathcal{V}}_j(s) = u_j(s)$ quand $s \in S_{ter}$, la stratégie minimax bornée par d est spécifiée par :

$$\mathcal{V}_j^d(s) = \begin{cases} \max_{a_j} \min_{\boldsymbol{a}_{-j}} Q_j^d(s, \boldsymbol{a}) & \text{si } d > 0 \text{ et } s \notin S_{ter} \\ \hat{\mathcal{V}}_j(s) & \text{sinon} \end{cases}$$
$$\text{où } Q_j^d(s, \boldsymbol{a}) = \sum_{s' \in S} P(s' \,|\, s, \boldsymbol{a}) \mathcal{V}_j^{d-1}(s'). \tag{5.2}$$

Clairement, $\mathcal{V}_j^d(s) = \mathcal{V}_j^*(s)$ dès que $d = T$ pour un jeu d'horizon T. La qualité de $\mathcal{V}_j^d(s)$ dépend à la fois de la fonction heuristique $\hat{\mathcal{V}}_j$ et de la profondeur d choisie.

Intuitivement, une symétrie définie sur un ensemble O d'objets est une permutation σ sur O qui ne modifie pas certaines propriétés de ces objets. Toute collection de symétries préservant la même propriété sur O forme un groupe de symétries.

Définition 5.1 (Groupe de symétries) *Un groupe de symétries est un tuple (X,Θ) tel que X est un ensemble d'objets et Θ une bijonction où :*

- *Θ autorise l'associativité : $\forall i,j,k \in X\ (i\Theta j)\Theta k = i\Theta(j\Theta k)$;*
- *Θ possède un élément neutre ϵ tel que : $\forall i \in X\ i\Theta\epsilon = \epsilon\Theta i = i$;*
- *Θ autorise la bijonction réciproque : $\forall i \in X\ i\Theta i^{-1} = i^{-1}\Theta i = \epsilon$.*

Le groupe obtenu en particulier dépend de la propriété préservée par l'ensemble des symétries en question. Deux types principaux de symétries de jeu sont distingués dans ce chapitre : les symétries de structures préservant la structure d'un jeu et les symétries minimax préservant les stratégies minimax des joueurs interagissant dans un jeu. Formellement, soit $G(R) = (s_0, S_{ter}, L, P, \boldsymbol{u})$ un jeu stochastique défini sur l'ensemble J des joueurs, l'ensemble des fluents F, l'ensemble des actions possibles A décrit par le jeu `GDL` R. Une symétrie de structure de G est une fonction bijective θ sur $J \cup F \cup A$ tel que pour tout $j \in J, f \in F, s, s' \subseteq F$ et $\boldsymbol{a} \in A^k$,

$$\sigma(j) = j, \sigma(a) \in A, \sigma(f) \in F \tag{5.3a}$$
$$\sigma(f) \in s_0 \text{ ssi } f \in s_0 \tag{5.3b}$$
$$\sigma(s) \in S_{ter} \text{ ssi } s \in S_{ter} \tag{5.3c}$$
$$\sigma(a) \in L_j(\sigma(s)) \text{ ssi } a \in L_j(s) \tag{5.3d}$$
$$P(\sigma(s') \mid \sigma(s), \sigma(\boldsymbol{a})) = P(s' \mid s, \boldsymbol{a}) \tag{5.3e}$$
$$u_j(\sigma(s)) = u_j(s) \tag{5.3f}$$

où $\sigma(s) = \{\sigma(f) \mid f \in s\}$, et $\sigma(\boldsymbol{a}) = (\sigma(a_1), \ldots, \sigma(a_k))$. La condition (5.3a) indique que les symétries de jeux doivent préserver les types d'objets et les conditions (5.3b à 5.3f) affirment que les symétries doivent aussi préserver l'état initial, les états terminaux, les actions légales, les transitions de probabilités et les utilités de chacun des joueurs.

Une stratégie minimax est une fonction bijective θ sur $J \cup F \cup A$ satisfaisant les conditions (5.3a à 5.3c) ainsi que

$$\mathcal{V}^*_j(\sigma(s)) = \mathcal{V}^*_j(s) \tag{5.4}$$

pour tout $j \in J$ et $s \in S$. Par extension, les stratégies minimax de profondeur d sont définies en remplaçant la condition (5.4) par

$$\mathcal{V}^d_j(\sigma(s)) = \mathcal{V}^d_j(s) \tag{5.5}$$

Notons que toute symétrie de structure est une symétrie minimax, car les conditions (5.3d à 5.3f) impliquent que deux états possédant des sous-jeux équivalents doivent posséder la même valeur minimax. Toutefois le contraire n'est pas vrai.

Exemple 5.1 *Soit σ une stratégie minimax, s et s' deux états avec $\sigma(s) = s'$ mais $|\boldsymbol{L}(s)| \neq |\boldsymbol{L}(s')|$ alors aucune symmétrie de structure ne peut associer s dans s' à cause de la condition (5.3d).*

Pourtant, les symétries de structures et les symétries minimax de profondeur d sont incomparables (sauf pour $d = T$ dans les jeux à horizon T). En effet, même si deux états distincts s et s' possèdent des sous jeux équivalents, on a $\hat{\mathcal{V}}_j(s) \neq \hat{\mathcal{V}}_j(s')$ dès que $\hat{\mathcal{V}}_j$ est appliquée, par exemple, une fonction heuristique qui estime la valeur des états par échantillonnage .

Au cours de la partie suivante, nous mettons en avant la détection de symétries dans le cadre du *General Game Playing* via la méthode des graphes de règles.

5.1.2 Méthode des graphes de règles (méthode RG)

Sans lien avec la détection de symétries dans les jeux, [KUHLMANN & STONE 2007] décrit une modélisation d'un programme GDL sous la forme de « graphes de règles » tel que deux graphes de règles soient isomorphes si et seulement si les programmes GDL décrivant ces jeux soient identiques par un simple processus de renommage de variables et de constantes qui ne soient pas clés. Fondamentalement, un graphe de règles contient des sommets pour l'ensemble des prédicats, des fonctions, des constantes et des variables présentes dans la description GDL du jeu et les connecte entre eux afin de faire correspondre les structures composantes les règles GDL utilisées. Les nœuds d'un graphe de règles sont colorés tels que les isomorphismes ne peuvent faire correspondre que des constantes vers d'autres constantes, des variables vers d'autres variables.

[SCHIFFEL 2010] propose d'exploiter les graphes de règles pour détecter les symétries présentes dans les programmes GDL. Pour rappel, nous nous intéressons au fragment propositionnel des jeux GDL décrit dans le chapitre 3 et noté P−GDL. Soit un *ground* P−GDL R_g. Les *ground* termes de R_g peuvent être partitionnés en un ensemble $J^* = \{1, 2, \ldots, k, k+1\}$ de joueurs (où $k+1$ représente le joueur environnement), un ensemble F de fluents, un ensemble A d'actions et un ensemble U de constantes d'utilité.

Pour un programme P−GDL, nous utilisons une version simplifiée des graphes de règles. Formellement, le *ground* graphe de règles $\mathbb{G}_R$ d'un programme P−GDL R est le graphe coloré (N, E, χ) sur R_g tel que

- chaque *ground* terme $\mathtt{t} \in J^* \cup F \cup A \cup U$ est associé à un noeud distinct $n_\mathtt{t}$; si $\mathtt{t}$ est un joueur $j \in J^*$, alors $\chi(n_\mathtt{t}) = j$, si $\mathtt{t}$ est une valeur d'utilité $\nu \in U$, alors $\chi(n_\mathtt{t}) = \nu$, si $\mathtt{t}$ est un fluent de F, alors $\chi(n_\mathtt{t}) = \mathtt{fluent}$, et si $\mathtt{t}$ est une action de A, alors $\chi(n_\mathtt{t}) = \mathtt{action}$;
- chaque atome $\mathtt{A}$ de la forme $\mathtt{p}pred(\mathtt{t}_1, \ldots, \mathtt{t}_q)$ est associé à un noeud distinct $n_\mathtt{A}$ de la couleur $\chi(n_\mathtt{A}) = \mathtt{p}pred$, et à une clique sur $\{n_\mathtt{A}, n_{\mathtt{t}_1}, \cdots, n_{\mathtt{t}_{pred}}\}$;
- chaque littéral $\mathtt{L}$ de la forme $\mathtt{not\ A}$ est associé à un noeud distinct $n_\mathtt{L}$ avec $\chi(n_\mathtt{L}) = \mathtt{not}$, et à une arête $(n_\mathtt{L}, n_{pred})$;
- chaque règle $\mathtt{r}$ de la forme $\mathtt{A\ :-\ L}_1, \ldots, \mathtt{L}_q$ est associé à un noeud distinct $n_\mathtt{r}$ de couleur $\chi(n_\mathtt{r}) = \mathtt{rule}$, et à une clique sur $\{n_\mathtt{r}, n_\mathtt{A}, n_{\mathtt{L}_1}, \ldots, n_{\mathtt{L}_q}\}$.

Proposition 5.1 *Soit R_g un ground programme P−GDL et $\mathbb{G}$ son ground graphe de règles. Alors, tout automorphisme de $\mathbb{G}$ est une symétrie de structure de $G(R_g)$*

Preuve *Soit $\mathbb{G} = (N, E, \chi)$ le graphe coloré de règles de $\mathtt{R}_g$, et soit σ un automorphisme de $\mathbb{G}$. Pour un ground terme $\mathtt{t}$, soit $\sigma(\mathtt{t})$ le terme $\mathtt{t}'$ tel que $n_{\mathtt{t}'} = \sigma(n_\mathtt{t})$. Pour un atome $\mathtt{A}$ de la forme $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_q)$, soit $\sigma(\mathtt{A})$ l'atome $\mathtt{p}(\sigma(\mathtt{t}_1), \ldots, \sigma(\mathtt{t}_q))$. Notons que le nom de l'atome $\mathtt{p}$ est préservé par χ. Les extensions aux littéraux $\sigma(\mathtt{L})$ et aux règles $\sigma(\mathtt{r})$ sont définies similairement. Il est possible d'affirmer que $\sigma(\mathtt{R}_g) = \mathtt{R}_g$, dès que $\sigma(\mathtt{r}) \in \mathtt{R}_g$ si et seulement si $\mathtt{r} \in \mathtt{R}_g$. Basé sur cet invariant, (5.3a) se base sur la coloration χ et (5.3b) sur la sémantique des états initiaux. De plus, si $s \in S_{ter}$ alors $\mathtt{R}_g \cup \mathtt{true}(s) \vdash \mathtt{terminal}$, qui est équivalent à $\sigma(\mathtt{R}_g) \cup \mathtt{true}(\sigma(s)) \vdash \sigma(\mathtt{terminal})$ dès que $\mathtt{true}$ et $\mathtt{terminal}$ sont conservés par χ. Par conséquent, $\sigma(s) \in S_{ter}$. Les autres conditions (5.3d à 5.3f) sont démontrées similairement. Notamment, la démonstration de la condition (5.3e) vient du fait de $|L_{k+1}(\sigma(s))| = |L_{k+1}(s)|$ et de $\sigma(s') \in \rho(\sigma(s), \sigma(\boldsymbol{a}))$ si et seulement si $s' \in \rho(s, \boldsymbol{a})$.*

Afin de capturer le plus de symétries possibles, l'utilisation du graphe de règles du programme GDL sans les règles « init » décrivant l'état initial est construit. L'approche de détection de symétries via les graphes de règles propose l'utilisation de tables de transpositions pour détecter des états symétriques avant d'étendre l'arbre de recherche d'un programme GDL afin d'atteindre les états suivants. Ainsi, si

un état symétrique à l'état venant d'être atteint est déjà présent dans la table de transposition, il suffit de générer le sous-arbre correspondant à cet état via les informations stockées dans la table de transposition.

Cette méthode permet d'améliorer la vitesse d'expansion de l'arbre de recherche nécessaire à toute méthode de résolution utilisée par un programme-joueur générique pour quelques jeux GDL mais dans le cas où le nombre des symétries est trop important ou dans celui où la taille des règles *next* et *legal* est trop grand, celui-ci provoque l'effet inverse en ralentissant la recherche dans l'arbre de recherche.

En effet, le principal défaut de cette méthode est sa proximité syntaxique avec le programme GDL, tout d'abord comme dit dans le paragraphe ci-dessus, si celui-ci est important, la recherche de symétries est lente et prend du temps à générer. Le problème étant intuitivement lié au fait que cette approche n'autorise pas de faire correspondre des termes arbitraires vers d'autres termes. Par exemple, prenons deux actions a et a' avec les mêmes préconditions et effets mais que le programme GDL utilisent les règles suivantes pour l'action a : (i) $legal(j, a) \leftarrow A$ (ii) $legal(j, a) \leftarrow not\ A$ et pour l'action a' un seul fait $legal(j, a')$. Dans ce cas, aucun automorphisme dans G_R ne peut permuter a et a' car les voisins de n_a et n'_a ne sont pas équivalents.

Afin de circonvenir à ce problème, [SCHKUFZA *et al.* 2008] propose de transformer le programme GDL par des propositions avant de générer le graphe de règles, mais cette méthode n'est possible que pour les petits jeux car la représentation des règles générées est exponentiellement plus grande que les règles d'origine provoquant également un calcul des automorphismes du graphe très important.

De plus, comme indiqué par [SCHIFFEL 2010], seules les symétries induites par la description GDL d'un jeu sont détectées. Par conséquent, les symétries d'un même jeu décrit différemment par plusieurs programmes GDL ne mènent pas au même ensemble de symétries.

Pour toutes ces raisons, cette méthode est peu utilisée dans le cadre pratique (en compétition GGP). Ainsi au cours de ce chapitre, nous proposons d'exploiter la détection de symétries via le réseau de contraintes stochastiques généré par notre processus de traduction pour détecter l'ensemble des symétries et permettre leurs exploitation efficacement. Mais avant de présenter notre approche, il est nécessaire de définir la notion de symétrie dans le cadre de la programmation par contraintes.

5.2 Détection de symétries dans la programmation par contraintes

De nombreuses définitions des symétries dans le cadre des contraintes existent dans la littérature. Cette notion est issue du calcul propositionnel où [AGUIRRE 1991] et [CRAWFORD *et al.* 1996] définissent une permutation σ sur les variables d'un ensemble de clauses W en forme conjonctive normale (CNF) comme une symétrie si $W = \sigma(W)$. [BENHAMOU & SAIS 1992] généralise cette définition à une permutation σ définie sur un ensemble de littéraux préservant l'ensemble des clauses. Plus généralement [BROWN *et al.* 1988] définit une symétrie comme une permutation des variables d'un problème qui préserve l'ensemble des solutions et [BACKOFEN & WILL 2002] l'étend au cadre des CSP comme une fonction bijective de l'ensemble des solutions. Finalement, la symétrie devient une notion fondamentale pour la résolution des réseaux de contraintes.

5.2.1 Deux types de symétries

[COHEN *et al.* 2006] définit la notion de symétrie dans un CSP en deux définitions distinctes.

Définition 5.2 (Symétrie de solutions (*solution symmetry*)) *Soit un CSP $\mathcal{P} = (X, D, C)$. Une symétrie de solutions de $\mathcal{P}$ est une permutation sur l'ensemble $X \times D$ qui préserve l'ensemble des solutions de $\mathcal{P}$ ($sols(\mathcal{P})$).*

En d'autres termes, une symétrie de solution est une bijonction définie sur l'ensemble des paires variables-valeurs qui préservent l'ensemble des solutions. Notons que des symétries sur les valeurs ou sur les variables sont des cas particuliers des symétries sur les paires variables-valeurs.

Définition 5.3 (Symétrie de contraintes (*constraint symmetry*)) *Soit un* CSP $\mathcal{P} = (X, D, C)$*, une symétrie de contraintes est un automorphisme de la micro-structure complémentaire de* $\mathcal{P}$*.*

Remarque 5.1 *Rappelons qu'un automorphisme est une application linéaire bijective d'un espace dans lui-même. Ainsi, appliqué à un hypergraphe, il s'agit d'une bijection des sommets préservant les arêtes de l'hypergraphe.*

Exemple 5.2 *Soit le* CSP $\mathcal{P} = (X, D, C)$ *où* $X = \{w, x, y, z\}$ *avec* $dom(w) = dom(x) = dom(y) = dom(z) = \{0, 1\}$ *et* $C = \{c_1 : x + y + z = 0, c_2 : w + y = 1, c_3 : w + z = 0\}$ *avec 1+1=0.*

c_1 *possède quatre instanciations globalement incohérentes de taille 3 :*

— $c_1^{I_1} : \{(x, 1), (y, 0), (z, 0)\}$ *;*
— $c_1^{I_2} : \{(x, 0), (y, 1), (z, 0)\}$ *;*
— $c_1^{I_3} : \{(x, 0), (y, 0), (z, 1)\}$ *;*
— $c_1^{I_4} : \{(x, 1), (y, 1), (z, 1)\}$.

Les contraintes c_2 *et* c_3 *ont quant à eux quatre instanciations globalement incohérentes de taille 2 :* $\{(w, 1), (z, 0)\}$, $\{(w, 0), (z, 1)\}$, $\{(w, 1), (y, 1)\}$ *et* $\{(w, 0), (y, 0)\}$.

Enfin, chaque variable de par son domaine provoque l'apparition d'une instanciation globalement incohérente de taille 2 : $\{(w, 0), (w, 1)\}$, $\{(x, 0), (x, 1)\}$, $\{(y, 0), (y, 1)\}$, $\{(z, 0), (z, 1)\}$. *La micro-structure complémentaire de* P *est représentée en Figure 5.1.*

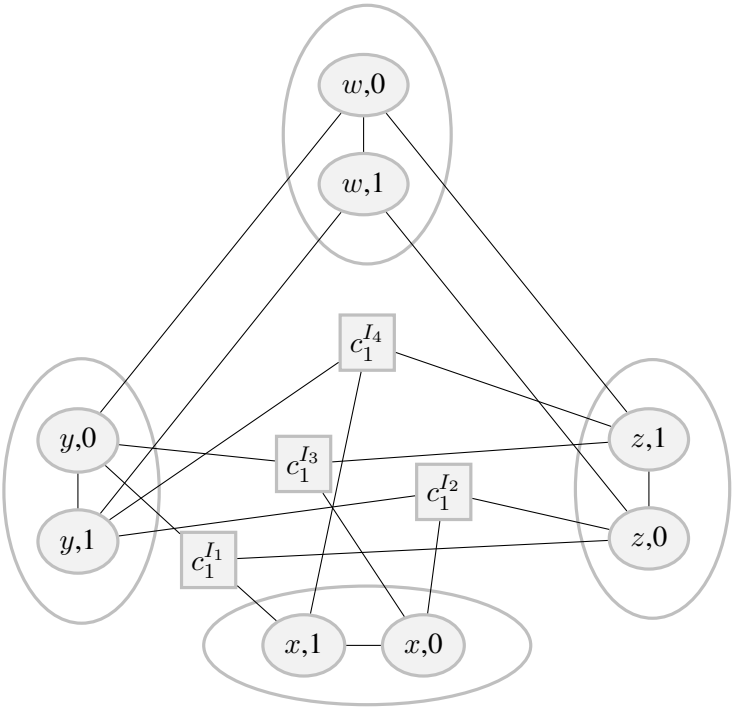


FIGURE 5.1 – micro-structure complémentaire d'un CSP $\mathcal{P}$

Les automorphismes de cet hypergraphe sont représentés par ces permutations de sommet en cycle :

— $\{(\langle w, 0\rangle\langle w, 1\rangle)\ (\langle y, 0\rangle\langle y, 1\rangle)\ (\langle z, 0\rangle\langle z, 1\rangle)$ *;*

— $\{(\langle w,0\rangle\langle w,1\rangle)\ (\langle y,0\rangle\langle z,0\rangle)\ (\langle y,1\rangle\langle z,1\rangle)$;
— $\{(\langle y,0\rangle\langle z,1\rangle)\ (\langle y,1\rangle\langle z,0\rangle)$.

Par exemple, pour le premier automorphisme, si on permute $(w,0)$ *avec* $(w,1)$ *alors on doit permuter simultanément* $(y,0)$ *avec* $(y,1)$ *et* $(z,0)$ *avec* $(z,1)$ *mais on ne modifie pas* $(x,0)$ *et* $(x,1)$. *Comme indiqué dans la définition 5.3, les symétries de contraintes correspondent aux automorphismes générés par ces permutations. Toutefois, nous avons* $sols(\mathcal{P}) = \{\ \{(w,0),(x,1),(y,1),(z,0)\},\ \{(w,1),(x,1),(y,0),(z,1)\}\ \}$. *Par conséquent, La permutation* $\{(\langle w,0\rangle\langle z,0\rangle\langle y,1\rangle)$ *permettant de permuter* $(w,0)$ *en* $(z,0)$, $(z,0)$ *en* $(y,1)$ *et enfin* $(y,1)$ *en* $(w,0)$ *préserve l'ensemble des solutions, il s'agit donc d'une symétrie de solutions comme indiquée dans la Définition 5.2.*

Comme vu dans la section précédente, la particularité d'un groupe de symétries dépend des propriétés préservées par ce groupe. Les symétries de solutions préservent la notion d'instanciation globalement cohérente (les solutions) dans un CSP alors que les symétries de contraintes ne préservent que les instanciations localement cohérentes (satisfaisant un sous-ensemble de contraintes).

Propriété 5.1 *Le groupe des symétries de contraintes d'un CSP* $\mathcal{P}$ *est un sous-groupe du groupe des symétries de solutions de* $\mathcal{P}$. *Dit autrement, toute symétrie de contrainte est une symétrie de solution.*

Bien que, les deux types de symétries semblent très différentes de par leurs définitions, elles sont en faites très proches. En effet, toutes symétries de contraintes préservent également les solutions [COHEN *et al.* 2006].

Il est alors possible de considérer le groupe des symétries de solutions comme un groupe d'automorphismes d'un hypergraphe particulier.

Définition 5.4 (*k-nogood* hypergraphe) *Soit un CSP* $\mathcal{P} = (X, D, C)$. *Le k-nogood hypergraphe de* $\mathcal{P}$ *est un hypergraphe dont l'ensemble des sommets est* $X \times D$ *et dont l'ensemble des hyperarêtes est l'ensemble des k-nogoods de taille* m *(variables) tel que* $m \leq k$.

Soit un CSP $\mathcal{P} = (X, D, C)$ et sa micro-structure complémentaire associée définit par un hypergraphe $\mathcal{H} = (N, E)$. Un *k-nogood* hypergraphe $\mathcal{H}_k = (N_k, E_k)$ de $\mathcal{P}$ possède les mêmes sommets que $\mathcal{H}$ ($N = N_k$) et toutes les hyperarêtes de $\mathcal{H}$ sont inclues dans $\mathcal{H}_k$ ($E \subset E_k$). Les hyperarêtes supplémentaires de $\mathcal{H}_k$ représentent les assignements partiels localement incohérents de k variables (les *k-nogoods*).

Propriété 5.2 *Pour tout CSP* $\mathcal{P}$ *d'arité* k, *le groupe de toutes les symétries de solutions de* $\mathcal{P}$ *équivaut au groupe des automorphismes du k-nogood hypergraphe de* $\mathcal{P}$.

De ce fait, pour obtenir les symétries de solutions d'un CSP d'arité k, il suffit de considérer les automorphismes d'un hypergraphe obtenu en ajoutant à la micro-structure complémentaire de $\mathcal{P}$ l'ensemble des hyperarêtes correspondant aux *nogoods* d'arité k ou inférieure.

Ainsi parfois il est nécessaire de considérer l'ensemble des *nogoods* d'arité k pour obtenir l'ensemble des solutions (par exemple si un CSP n'a pas de solution). Par contre, il est à noter qu'il est uniquement nécessaire de considérer les *nogoods* portant sur k variables au plus (par exemple, sur un CSP binaire, seuls les *nogoods* portant sur une ou deux variables sont nécessaires).

5.2.2 Exemple d'application

Au cours de cette sous-section, nous allons appliquer les symétries de solutions et de contraintes au problème des n-reines d'arité 2 illustré dans la sous-section 2.1.2. Ce problème se présente sous la forme d'un échiquier et par conséquent il possède huit symétries géométriques : une symétrie axiale par

rapport à l'horizontale, une par rapport à la verticale, deux par rapport aux diagonales et des symétries par rapport à la rotation à 90°, 180°, 270°, 360°.

Pour rappel, le CSP correspondant au problème des n-reines modélise les reines par n variables correspondant aux colonnes et dont les valeurs associées correspondent aux lignes. Ainsi la micro-structure complémentaire du problème des n-reines composée des tuples variables-valeurs interdits modélise les symétries entre lignes et colonnes. Par conséquent, chaque symétrie géométrique de l'échiquier est représentée par une symétrie de contrainte (un automorphisme de la micro-structure complémentaire) comme indiqué dans la Définition 5.3.

Clairement, l'ensemble des solutions du CSP correspondant au problème des n-reines est invariant en appliquant chaque symétrie géométrique. De ce fait et selon la définition 5.2, chaque symétrie géométrique est une symétrie de solution pour le CSP correspondant. Toutefois, il est possible de déterminer plus de symétries de solutions pour le CSP que les huit géométriques comme nous allons le voir pour différents sous-problèmes des n-reines pour différentes valeurs de n. Notons que le CSP est binaire et que pour cet exemple, nous pouvons parler de graphe au lieu d'hypergraphe et que seuls les *1-nogoods* et les *2-nogoods* sont nécessaires.

- Le problème des 3-reines n'a pas de solution. Comme indiqué dans la propriété 5.2, le *2-nogood* hypergraphe est le graphe complet des neuf sommets le composant, ainsi chaque permutation des sommets du graphe est un automorphisme ;
- Le problème des 4-reines illustré par son graphe de comptabilité en figure 2.3 n'a que deux solutions illustrées en figure 2.4. Il est possible de remarquer que ces deux solutions sont liées par une symétrie axiale sur la diagonale (sud-ouest ; nord-est). Comme les automorphismes d'un graphe sont les mêmes que les automorphismes de son complément, nous pouvons considérer le complément du *2-good* hypergraphe comme les hyperarêtes représentant les assignations des paires variable-valeur qui sont autorisées par ces solutions.
  La figure 5.2 illustre ce graphe où chaque sommet représente une paire variable-valeur sur un échiquier. La position de chaque paire variable-valeur sur cet échiquier correspond à la position d'une reine sur ce dernier. Chaque solution est décrite comme une clique de taille 4 sur ce graphe. Les automorphismes de ce graphe sont : les permutations sur les sommets d'une même clique, les permutations entre les sommets d'une clique avec les sommets de la seconde clique et les sommets isolés (qui ne sont pas compris dans une clique) représentent les *1-nogoods*. De plus, il est possible de combiner ces permutations 2 par 2 afin d'en obtenir de nouvelles. On dénombre un total de $4! \times 4! \times 2 \times 8! = 46,448,640$ automorphismes. Par conséquent, le problème des 4-reines possèdent dont $46,448,640$ symétries de solutions.

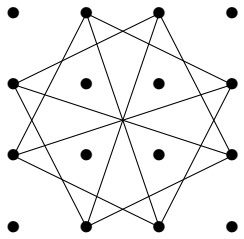


FIGURE 5.2 – Le complément du *2-nogood* hypergraphe du problème des 4-reines.

- Le problème des 5-reines possède dix solutions comme décrites dans la figure 5.3. Ces solutions sont réparties en deux classes d'équivalences par les symétries géométriques de l'échiquier. Elles transforment chaque solution en une autre solution de la même classe d'équivalence. Dans la figure, les huit premières solutions font partie de la même classe d'équivalence et les deux

dernières à la seconde classe d'équivalence.
Chaque case de l'échiquier possède au moins une reine sur l'une des dix solutions, par conséquent le CSP ne possède pas de *1-nogood*. Par contre, il existe certains couples de cases, où deux reines ne peuvent toujours pas être placées correspondant à des *2-nogoods*. Certaines directement déductibles des contraintes binaires et 40 autres par inférence comme la paire de variable-valeur $((x_1, 5), (x_3, 2))$ qui empêche toute assignation de x_2. Ainsi en ajoutant ces 40 arêtes à la micro-structure complémentaire, nous obtenons le *2-nogood* hypergraphe du problème des 5-reines. À l'aide du logiciel NAUTY [McKay & Piperno 2014], il est possible de déterminer les automorphismes de ce graphe. On en dénombre $28,800$ c'est à dire autant de symétries de solutions. Contrairement au problème des 4-reines, ces symétries de solutions ne sont pas visuellement descriptibles par les symétries de géométrie.

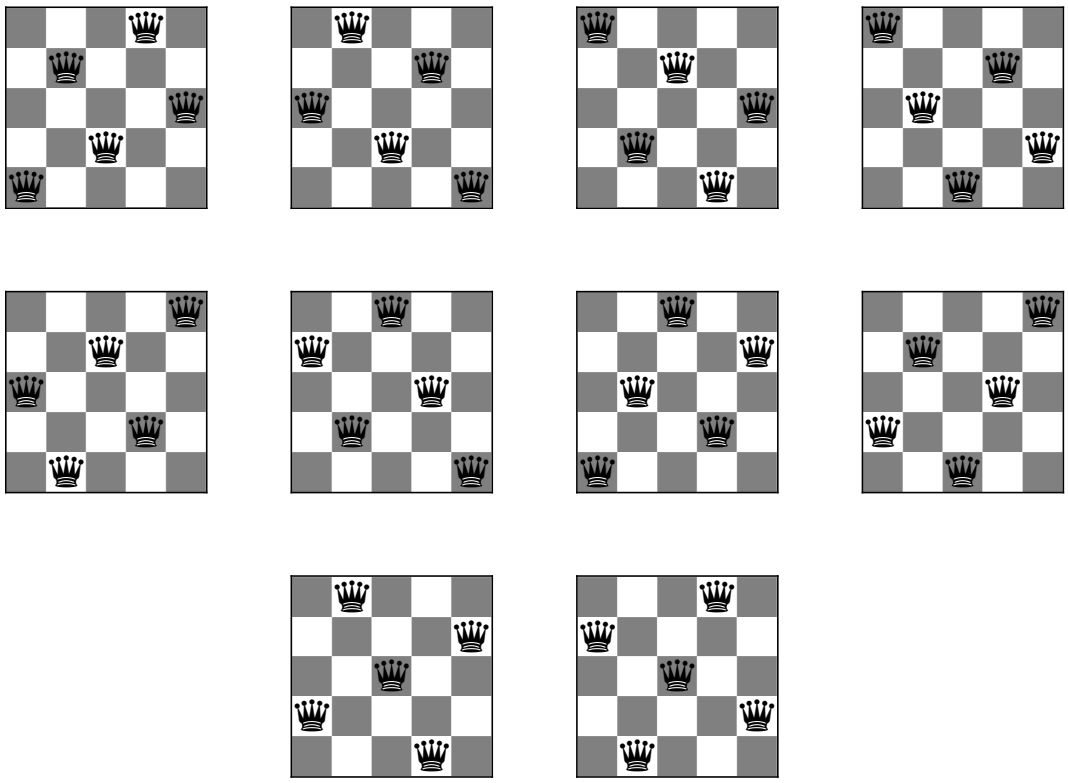


FIGURE 5.3 – Les 10 solutions du problème des 5-reines.

- Pour le problème des 6-reines qui ne possède que quatre solutions, le nombre de *1-nogood* est important ce qui implique un nombre très important d'arêtes à ajouter à la micro-structure complémentaire. Le nombre d'automorphismes qui en découle est de la même façon important comme le montre le tableau 5.1.
- Au delà de 6 reines ($n \geq 7$), le nombre d'automorphisme est toujours égal à 8. En effet, le nombre de symétrie de solutions devient égal au nombre de symétrie géométrique car la micro-structure complémentaire devient identique au *2-nogood* hypergraphe.

Au cours des deux sections suivantes, nous exposons les relations entre les symétries dans le cadre de la programmation par contraintes et les symétries de jeux. Nous commençons par exploiter la notion de symétrie de contraintes

n	Symétries de solutions
3	$9! = 362,880$
4	$4! \times 4! \times 2 \times 8! = 46,448,640$
5	$28,800$
6	$3,089,428,805,320,704,000,000$
$n \geq 7$	8

TABLE 5.1 – Nombre de symétrie de solutions pour le problème des n-reines pour différents n.

5.3 Symétries de jeux GDL et symétries dans un SCSP

Basé sur la section précédente, nous pouvons étendre la notion de microstructure complémentaire aux réseaux de contraintes stochastiques obtenus après traduction d'un jeu P-GDL. Soit un SCSP $\mathcal{P}$, la microstructure complémentaire de $\mathcal{P}$ est l'hypergraphe $\mathcal{H} = (N, E)$, où $N = \{(x, v) \mid x \in X, v \in D(x)\}$, et E l'union de toutes les contraintes $c \in C$ des instanciations sur $scp(c)$ interdites par c, c'est à dire, $E = \bigcup_{c \in C} \{\boldsymbol{v} \in D(scp(c)) \mid c(\boldsymbol{v}) = 0\}$. Chaque automorphisme de $\mathcal{H}$ est une symétrie de contraintes de $\mathcal{P}$.

Notons $\mathcal{P}_{R,T} = (X, Y, D, C, P, \theta)$ le SCSP à T niveaux obtenu à partir de la traduction du programme P-GDL R. Par extension, la microstructure complémentaire de $\mathcal{P}_{R,T}$ est l'hypergraphe coloré $\mathcal{H} = (N, E, \chi)$, où N et E sont définis comme précédemment et χ associe chaque nœud (x, v) dans N comme suit : si x est une variable terminale ω_t alors $\chi(x) =$ $terminal$, si x est une variable fluent $f_{i,t}$, si x est une variable joueur $z_{j,t}$ alors $\chi(x) = j$, si x est une variable stochastique y_t alors $\chi(x) = 0$ et si x est une variable d'utilité $r_{j,t}$ alors $\chi(x) = v$.

Proposition 5.2 *Soit le programme P-GDL R et T un entier positif. Alors toute symétrie de contraintes de $\mathcal{P}_{R,T}$ est une symétrie de structure de $G(\mathcal{P}_{R,T})$*

Preuve *Soit $\mathcal{H} = (N, E, \phi)$ la micro-structure complémentaire de $\mathcal{P}_{R,T}$ et σ l'automorphisme de $\mathcal{H}$. Soit $G = (s_0, S_{ter}, L, P, \boldsymbol{u})$ le jeu d'horizon T de $\mathcal{P}_{R,T}$ et $\boldsymbol{h} = (s_0, \boldsymbol{a}_0, s_1, \ldots, s_{T-1}, \boldsymbol{a}_{T-1}, s_T)$ une sous-séquence de séquences de jeux de G, où $s_T \in S_{ter}$. Enfin, Soit $P(\boldsymbol{h} = \prod_{t=1}^{T} P(s_t \mid s_{t-1}, \boldsymbol{a}_t)$ et $u_j(\boldsymbol{h}) = u_j(s_T)$ pour chaque joueur $j \in \{1, \ldots, k\}$.*

Considérons la séquence

$$\vartheta(\boldsymbol{h}) = \bigcup_{t=1}^{T} \vartheta_t(s_t) \cup \bigcup_{t=1}^{T-1} \vartheta_t(\boldsymbol{a}_t)$$

Clairement, $P(\vartheta(\boldsymbol{h})) = P(\boldsymbol{h})$ et $\vartheta(\boldsymbol{h}) \cup \{(u_{j,T}, u_j(\boldsymbol{h}))\}_{j=1}^{k}$ est globalement cohérent. Ainsi, une condition suffisante pour établir que σ satisfait (5.3a à 5.3f) est de prouver que la séquence

$$\sigma(\vartheta(\boldsymbol{h})) = \bigcup_{t=1}^{T} \sigma(\vartheta_t(s_t)) \cup \bigcup_{t=1}^{T-1} \sigma(\vartheta_t(\boldsymbol{a}_t))$$

satisfait les conditions suivantes : (i) $P(\vartheta(\boldsymbol{h})) = P(\boldsymbol{h})$ (ii) $\vartheta(\boldsymbol{h}) \cup \{(u_{j,T}, u_j(\boldsymbol{h}))\}_{j=1}^{k}$ est globalement cohérent.

Il est possible de remarquer que les joueurs et les utilités sont conservés par la coloration χ. Par conséquent, nous avons $\{\sigma((u_{j,T}, u_j(\boldsymbol{h}))\}_{j=1}^{k} = \{(u_{j,T}, u_j(\boldsymbol{h}))\}_{j=1}^{k}$. Comme les variables fluents et

les variables d'actions sont aussi préservées par χ, la séquence $\sigma(\vartheta(\boldsymbol{h}))$ inclue exactement T états encodant $\sigma(\vartheta_t(s_t))$ et $T-1$ actions encodant $\sigma(\vartheta_t(\boldsymbol{a}_t))$. De plus, comme les variables fluents initiaux et les variables terminal sont également préservées par χ, nous savons que $\sigma(\vartheta_1(s_1)) \in \prod_{i=1}^{n} D(f_{i,1})$ et $\sigma(\vartheta_T(s_T)) \in \prod_{i=1}^{n} D(f_{i,T})$. Finalement, comme σ est un automorphisme de $\mathcal{H}$, il préserve également tous les tuples interdits par l'ensemble des contraintes et de manière équivalente il ne modifie pas les tuples autorisés. Donc, $\vartheta(\boldsymbol{h}) \cup \{(u_{j,T}, u_j(\boldsymbol{h}))\}_{j=1}^{k}$ est globalement cohérent si et seulement si $\sigma(\vartheta(\boldsymbol{h})) \cup \{\sigma((u_{j,T}, u_j(\boldsymbol{h}))\}_{j=1}^{k}$ est globalement cohérent.

Pour conclure cette preuve, nous savons que pour chaque t, les tuples autorisés de $legal_{k+1,t}$ sont préservés par $\mathcal{H}$. Donc,

$$P_t(\cdot \mid \vartheta(s_t), \vartheta(\boldsymbol{a}_t)) = P_t(\cdot \mid \sigma(\vartheta(s_t)), \sigma(\vartheta(\boldsymbol{a}_t)))$$

pour chaque distribution de probabilités conditionnelles P_t. Par conséquent, $P(\sigma(\vartheta(\boldsymbol{h}))) = P(\boldsymbol{h})$.

Il est important de garder en tête que tout politique solution de $\mathcal{P}_{R,T}$ n'encode aucun concept de solution du jeu. En effet, toute politique solution π de $\mathcal{P}_{R,T}$ est juste une action légale qui satisfait les règles du jeu. Comme détaillé dans le chapitre 3 en section 3.4, la traduction d'un programme `GDL` permet d'encoder sous la forme d'un $SCSP$ uniquement les règles du jeu `GDL`. Par conséquent, il est logique que la détection des symétries de contraintes via la micro-structure complément de ce `SCSP` n'englobe pas les symétries de stratégies.

Soit $\mathcal{P}^*_{R,T}$ l'extension de $\mathcal{P}_{R,T}$ aux contraintes $C^* = (C^*_1, \ldots, C^*_T)$, où $C^*_t = C_t \cup \{\mathcal{V}^*_{j,t}, Q^*_{j,t}, \texttt{eq}_{j,t}\}_{j=1}^{k}$. Les portées de $\mathcal{V}^*_{j,t}$ et $Q^*_{j,t}$ sont $\{f_{i,t}\}_{i=1}^{n}$ et $\{f_{i,t}\}_{i=1}^{n} \cup \{a_{j,t}\}_{j=1}^{k}$, respectivement. Basé sur la stratégie minimax (5.1), $\mathcal{V}^*_{j,t}$ associe tout état s à $\max_{a_j} \min_{\boldsymbol{a}_{-j}} Q^*_{j,t}(s, \boldsymbol{a})$ si $\texttt{terminal}_t(s) = 0$ et à $goal_{j,t}(s)$ sinon.

$Q^*_{j,t}$ associe tout état s et toute combinaison d'actions $\boldsymbol{a}$ à $\sum_{a_{k+1}} P_t(a_{k+1} \mid s, \boldsymbol{a}) \mathcal{V}^*_{j,t+1}(s)$, où a_{k+1} est compris dans $D(p_{k+1,t})$ tel que $\texttt{legal}_{k+1,t}(a_{k+1}, s) = 1$. Finalement, la contrainte $eq_{j,t}$ est utilisée pour s'assurer que $Q^*_{j,t}(s, \boldsymbol{a})$ et $\mathcal{V}^*_{j,t}(s)$ sont égales, ce qui implique que seules les actions optimales du joueur j sont gardées dans les instanciations globalement cohérentes $(s, \boldsymbol{a})$. Le réseau $\mathcal{P}^d_{R,T}$ est défini de manière analogue en utilisant les stratégies minimax de profondeur d (5.2).

La micro-structure complémentaire de $\mathcal{P}^*_{R,T}$ et $\mathcal{P}^d_{R_T}$ sont définis comme indiqué ci-dessus.

Proposition 5.3 *Soit le programme `P-GDL` R et soit $T \geq 0$. Alors, toute symétrie de contraintes de $\mathcal{P}^*_{R,T}$ (resp. $\mathcal{P}^d_{R,T}$) est une symétrie minimax (resp. minimax de profondeur d) de $G(\mathcal{P}_{R,t})$.*

Preuve *Soit $\mathcal{H}^* = (N, E, \chi)$ la micro-structure complémentaire de $\mathcal{P}^*_{R,T}$ et σ un automorphisme de $\mathcal{H}^*$. Soit $G = (s_0, S_{ter}, L, P, \boldsymbol{u})$ le jeu d'horizon T de $\mathcal{P}_{R,T}$ et $\boldsymbol{h} = (s_0, \boldsymbol{a}_0, s_1, \ldots, s_{T-1}, \boldsymbol{a}_{T-1}, s_T)$ une sous-séquence finie de séquences de jeux de G, avec un état terminal s_T, les probabilités $P(\boldsymbol{h})$ et les utilités $u_j(\boldsymbol{h})$ définis comme précédemment. Soit $\vartheta(\boldsymbol{h})$ la représentation de $\boldsymbol{h}$ et $\sigma(\vartheta(\boldsymbol{h}))$ sa permutation. Suite à la proposition 5.3, nous savons que σ satisfait (5.3a à 5.3c). Maintenant, si $\boldsymbol{h}$ satisfait les égalités $Q^*(s_t, \boldsymbol{a}_t) = \mathcal{V}^*(s_t)$ pour chaque $t \in \{0, \cdots, T-1\}$, nous savons que $\vartheta(\boldsymbol{h})$ doit aussi satisfaire les égalités $Q^*(\vartheta(s_t), \vartheta(\boldsymbol{a}_t)) = \mathcal{V}^*(\vartheta(s_t))$ dans C^*_t. Ainsi, comme σ est un automorphisme de $\mathcal{H}$, il s'en suit que $\sigma(\vartheta(\boldsymbol{h}))$ satisfait $Q^*(\sigma(\vartheta(s_t)), \sigma(\vartheta(\boldsymbol{a}_t))) = \mathcal{V}^*(\sigma(\vartheta(s_t)))$. Par conséquent, la séquence $\sigma(\vartheta(\boldsymbol{h}))$ satisfait (5.4) comme souhaité. Une stratégie similaire s'applique pour $\mathcal{P}^d_{R,T}$.*

Le résultat précèdent est théoriquement très intéressant mais peut être difficilement implémenté suite à la taille importante de la micro-structure complémentaire du `SCSP` à T niveaux. Bien heureusement, lors du chapitre 4, nous montrons que le `SCSP` obtenu peut être résolu itérativement de par la notion de décomposabilité du `SCSP` en μ`SCSP` représentant chaque tour de jeu du programme `GDL`. Par conséquent, $\mathcal{P}^*_{R,T}$ (et $\mathcal{P}^d_{R,T}$) peut être décomposé en μ`SCSP`s, chacun associé à une micro-structure complémentaire restreinte, sur lesquelles les automorphismes peuvent être détectés plus efficacement.

Pour un tour $t \in \{0, \ldots, T\}$, le μSCSP$_t$ de $\mathcal{P}^*_{R,T}$ est un tuple $\mathcal{O}^*_{\mathtt{R},t} = (X_t, Y_t, D, C^*_t, P_t, \theta)$ où $O^*_{\mathtt{R},t}$ est la restriction de $\mathcal{P}^*_{R,T}$ avec $X_t = (V_t, Y_t)$. La micro-structure complémentaire de $\mathcal{O}^*_{\mathtt{R},t}$ est simplement la restriction de la micro-structure complémentaire de $\mathcal{P}^*_{R,T}$ pour l'ensemble des nœuds $N_t = \{(x, v) \mid x \in X_t, v \in D(x)\}$. Le μSCSP$_t$ de $\mathcal{P}^p_{R,T}$ et sa micro-structure complémentaire sont définis similairement.

Conjecture 5.1 *Soit un programme* P-GDL *et soit $T \geq 0$. Alors pour tout $t \in \{0, \ldots, T\}$, chaque symétrie de contraintes du $\mu SCSP_t$ de $\mathcal{O}^*_{\mathtt{R},t}$ (resp. $\mathcal{O}^d_{\mathtt{R},t}$) peut être étendue à une symétrie minimax de $G(\mathcal{P}^*_{R,t})$ (resp. minimax de profondeur d de $G(\mathcal{P}^d_{R,t})$).*

En supposant que la conjecture 5.1 soit vraie, l'intérêt pratique de notre cadre est que les symétries de contraintes de μSCSPs peuvent être exploitées pour réduire l'espace de recherche d'un politique solution dans le réseau. En effet, si à partir d'un certain niveau t, nous trouvons un automorphisme σ de $\mathcal{O}^d_{\mathtt{R},t}$, alors en couplant la proposition 5.3 et la conjecture 5.1, nous savons que toute action $a'_t = \sigma(a_t)$ est garantie d'être une stratégie minimax optimale de profondeur d à condition que a_t est une stratégie minimax optimale de profondeur d.

La génération automatique de μSCSP similaire à un μSCSP $\mathcal{P}$ déjà résolu permettrait d'exploiter l'utilité calculé de $\mathcal{P}$. Par exemple, dans le jeu du Tic-Tac-Toe, dès que l'adversaire vient à marquer la case centrale de son symbole, marquer n'importe quelle case située dans les coins du plateau mène à une situation similaire. Au cours de notre approche, nous proposons de détecter et de stocker l'ensemble des μSCSPs symétriques afin d'en bénéficier dès que l'un des membres du groupe de symétries a été atteint. Expliquons cette idée de par l'exemple suivant.

Exemple 5.3 *Considérons une variante simplifié du Tic-Tac-Toe 2x2 opposant un programme-joueur représentant le symbole O à un environnement représentant le symbole $\times$. Dans cette variante, seule la composition d'une ligne ou d'une colonne du même symbole permet au joueur de remporter la victoire (la diagonale ne le permettant donc pas). Supposons qu'à l'état initial, le symbole $\times$ marque la case $(1, 1)$ du plateau et que le contrôle est donné au joueur O. La figure 5.4 représente l'instance GDL décrivant ce jeu.*

La figure 5.5 représente l'arbre de recherche jusqu'au tour $t = 1$. Dans notre représentation par réseau de contraintes stochastiques à un niveau, chaque état i au tour t est représenté par un $\mu SCSP^i_t$.

En recherchant les automorphismes des micro-structures compléments de chaque $\mu SCSP$ au temps t, nous obtenons un seul et unique groupe d'automorphismes constituants les états symétriques $\mu SCSP^1_1$, $\mu SCSP^2_1$ et $\mu SCSP^3_1$. Ce résultat est concordant avec la réalité. En effet, ces trois états sont symétriques géométriquement par rapport à l'action choisie par le joueur représentant le symbole O.

Comme vu dans l'exemple précédent, la détection d'automorphismes dans la micro-structure complément des μSCSP correspondant à un tour du programme GDL permet de détecter l'ensemble des symétries de la structure du jeu. Contrairement à la détection de symétries par le graphe de règles d'un programme GDL, notre approche n'est pas dépendante syntaxiquement des règles GDL et ne se limite pas aux symétries induites de la description d'un jeu mais aux symétries du jeu lui-même.

Toutefois, l'utilisation unique des symétries de structure ne permet pas d'englober l'ensemble des symétries du jeu. En effet, ces symétries ne permettent pas de prendre en considération les utilités attendues de chaque état non-terminal toutes égales à 0 dans un programme GDL. Afin d'affiner l'utilité attendue de chaque état terminal, il est nécessaire de la calculer via les états terminaux composant le sous-arbre de chaque état non-terminal et de prendre en considération les solutions du SCSP équivalent au programme GDL.

```
role(oplayer). role(random).

symbole(random,x). symbole(oplayer,o).
index(1). index(2).

base(cell(L,C,S)) ← index(L), index(C), symbole(J,S).
base(cell(L,C,b)) ← index(L), index(C).
input(J,noop) ← role(J).
input(J,mark(L,C)) ← role(J), index(L), index(C).

init(cell(1,1,x)). init(cell(1,2,b)).
init(cell(2,1,b)). init(cell(2,2,b)).
init(control(oplayer)).

legal(J,noop) ← role(J), not(true(control(J))).
legal(J,mark(L,C)) ← role(J), true(cell(L,C,b)).

next(control(J2)) ← role(J2), true(control(J1)), distinct(J1,J2).
next(cell(L,C,S)) ← does(J,mark(L,C)), symbole(J,S).
next(cell(L1,C1,S)) ← true(cell(L1,C1,S), does(J,mark(L2,C2)), distinct(L1,L2),
distinct(C1,C2).

jouable ← true(cell(L,C,b)).
colonne ← true(cell(1,C,S)), true(cell(2,C,S)).
ligne ← true(cell(L,1,S)), true(cell(L,2,S)).

terminal ← not(jouable).
terminal ← ligne.
terminal ← colonne.

goal(J,100) ← true(cell(1,C,S)),true(cell(2,C,S)) symbole(J,S).
goal(J,100) ← true(cell(L,1,S)),true(cell(L,2,S)) symbole(J,S).
goal(J,50) ← not(jouable), not(ligne), not(colonne).
```

FIGURE 5.4 – Le programme GDL correspondant au jeu Tic-Tac-Toe 2x2

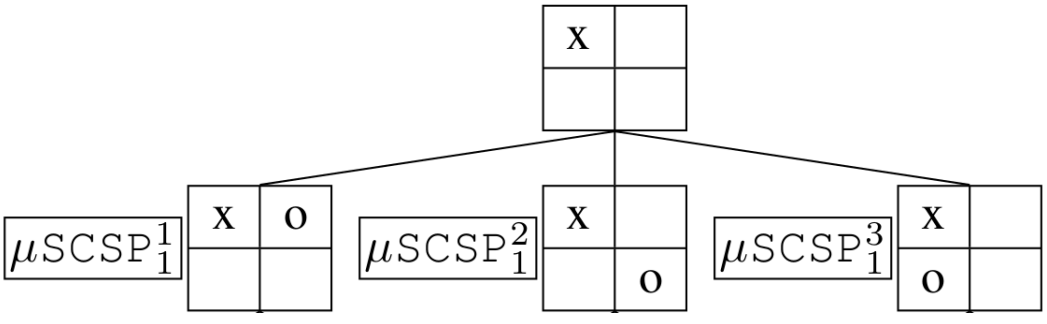


FIGURE 5.5 – L'arbre de recherche du Tic-Tac-Toe 2x2 au tour $t = 1$

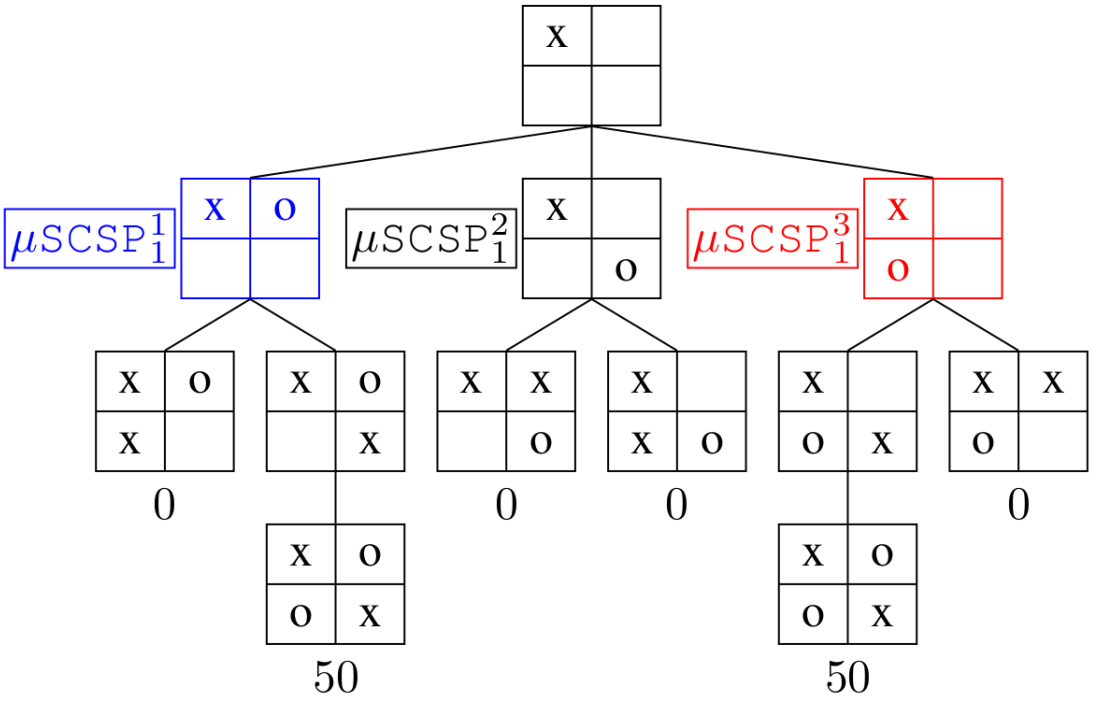


FIGURE 5.6 – L'arbre de recherche (complet) du `Tic-Tac-Toe 2x2`

Exemple 5.4 *Reprenons l'exemple précédent concentré sur la variante du* `Tic-Tac-Toe 2x2`*. La figure 5.6 représente l'arbre de recherche du jeu dans son intégralité. Les valeurs présentes sous les feuilles représentent les scores obtenus par le joueur O que le programme-joueur représente.*

Si on ajoute la notion d'utilité attendue à chaque état représenté par chaque μSCSP au temps $t = 1$, on obtient une utilité moyenne de 25 pour les deux $\mu SCSP_1^1$ (en bleu) et $\mu SCSP_1^3$ (en rouge) et une utilité moyenne de 0 pour le $\mu SCSP_1^2$ (en noir). On peut donc déduire que le second μSCSP au temps $t = 1$ n'est pas symétrique stratégiquement (par une stratégie minimax) au premier et au troisième μSCSP au tour $t = 1$.

L'intérêt de détecter l'ensemble des symétries du jeu via le réseau de contraintes stochastiques associé permet de n'explorer que le sous-arbre d'un seul membre du groupe de symétries pour déduire celui des autres membres, en effet dans l'exemple présenté ci-dessus, on peut remarquer que le groupe de symétries composé de l'état bleu et rouge, sont symétriques de par les conséquences des actions $mark(1, 2)$ ou $mark(2, 1)$ du joueur symbolisé par O. Ainsi, si l'état bleu est exploré en premier, l'ensemble du sous-arbre peut être généré pour l'état rouge sans explorer celui-ci permettant une exploration plus rapide.

Toutefois, dans le cas pratique, les arbres de recherche possèdent, comme nous l'avons vu précédemment, une largeur et une profondeur bien plus importante. C'est pourquoi dans la section suivante, nous proposons `MAC-UCB-SYM`, une extension de notre approche `MAC-UCB` associée à la détection de symétries minimax de profondeur d estimant l'utilité attendue via les simulations `UCB` pour chaque état non-terminal.

5.4 MAC−UCB−SYM

5.4.1 Détection en pratique des symétries

Dans un premier temps, toute instance GDL R est traduite en *ground* instance R_c à l'aide de la technique de complétude de Clark. Puis R_c est alors convertie en SCSP $\mathcal{P}_{R,T}$ à T niveaux comme décrit dans le chapitre 3.

Afin de détecter les symétries, MAC−UCB−SYM doit générer la micro-structure complémentaire $\mathcal{H}_t$ de chaque μSCSP_t composant $P_{R,T}$. Pour cela, il commence par générer la micro-structure complémentaire commune à chaque μSCSP_t issue de la traduction du programme GDL à tout temps t (sans les règles *init*). De ce fait pour obtenir $\mathcal{H}_t$ à un temps t donné, il suffit pour le temps $t = 0$ d'ajouter les arêtes correspondantes aux contraintes issues des règles *init* et pour $t \neq 0$ les arêtes correspondantes aux contraintes générées à partir des solutions de μSCSP_{t-1}. Les micro-structures obtenues permettent de capturer graphiquement les règles d'un jeu GDL R à chaque temps t. MAC−UCB−SYM représente chaque micro-structure complémentaire par un hypergraphe pouvant aussi être considéré comme un graphe bipartie : La première partie représentant l'ensemble des littéraux (couples variables-valeurs) et la seconde partie modélisant l'ensemble des tuples interdits composants toutes les contraintes de $\mathcal{P}$. Une arête (l, τ) de ce graphe correspond à l'occurrence d'un littéral l dans un tuple τ d'une contrainte.

Notons que pour chaque μSCSP_t composant $P_{R,T}$, l'ensemble des contraintes est enrichie de nouvelles contraintes permettant de modéliser les stratégies minimax comme décrit précédemment. Ainsi, chaque μSCSP_t $\mathcal{P}$ est enrichi d'une contrainte de portée $(\{F_t\}, \{A_t\}, A_{k+1,t})$ et de relation formée par l'ensemble des tuples de la forme $(s, \boldsymbol{a})$, s étant défini sur $\{f_t\}$ et $\boldsymbol{a}$ représentant une combinaisons d'actions sur $\{A_t\}$ et $A_{k+1,t}$. Chaque tuple $(s, \boldsymbol{a})$ est vu comme un $nogood$ si toutes les politiques débutant par s avec le vecteur d'action $\boldsymbol{a}$ est prédite par UCB comme une politique non solution. Dit autrement, l'utilité attendue simulée par UCB est inférieure au seuil θ.

Sur la base de chaque μSCSP $\mathcal{P}$ enrichi de tout $nogood$ détecté au fil de la résolution du problème, MAC−UCB−SYM déduit les automorphismes de la micro-structure complément de $mathcal P$. À partir de ce graphe, MAC−UCB−SYM calcule l'ensemble des automorphismes et génère les μSCSPs symétriques. Chacun de ces μSCSP est stocké et associé à leur utilité attendue dans une table de hachage de *Zobrist* [ZOBRIST 1990] où chaque valeur assignée dans chaque μSCSP correspond à une valeur de hachage générée aléatoirement. Le nombre de *Zobrist* alors utilisé pour retrouver un μSCSP dans cette table est alors le XOR des nombres aléatoires associés à chaque couple variable-valeur assignée garantissant une probabilité de collision très faible. Ainsi avant de résoudre un μSCSP, MAC−UCB−SYM vérifie si il ne correspond pas à un μSCSP symétrique déjà rencontré. Si c'est le cas, l'utilité attendue associée est directement obtenue sans avoir besoin de le résoudre ou de simuler les état suivants.

Finalement, la détection de symétries utilisée pour MAC−UCB−SYM étant uniquement basée sur la micro-structure complément et sur le seuil θ, cette technique est également tout à fait envisageable pour améliorer les performances de MAC−UCB−II dans le cadre des jeux à information incomplète.

5.4.2 Profondeur des stratégies minimax

MAC−UCB−SYM a pour objectif de trouver les stratégies minimax de profondeur d, représentées par les politiques solutions du SCSP $\mathcal{P}^d_{R,T}$. Le choix de la profondeur d est déterminé par le dilemme exploration/exploitation mais également par le temps dédié à la détection de symétries. Pour MAC−UCB−SYM, 45% du temps délibération est dédié à l'exploitation (MAC), 30% à l'exploration (UCB) et 25% à la recherche de symétries (SYM).

Dans le but de justifier ces ratios, la figure 5.7 représente l'analyse de sensibilité moyenne sur l'ensemble des jeux pratiqués lors de nos expérimentations par MAC−UCB−SYM. Le ratio correspondant à

chaque composant a été varié entre 20% et 80%. L'axe x des abscisses correspond au temps alloué à la partie résolution (MAC), l'axe y des ordonnées correspond au temps alloué à la détection de symétries (SYM) et l'axe z correspond au score moyen obtenu par MAC−UCB−SYM.

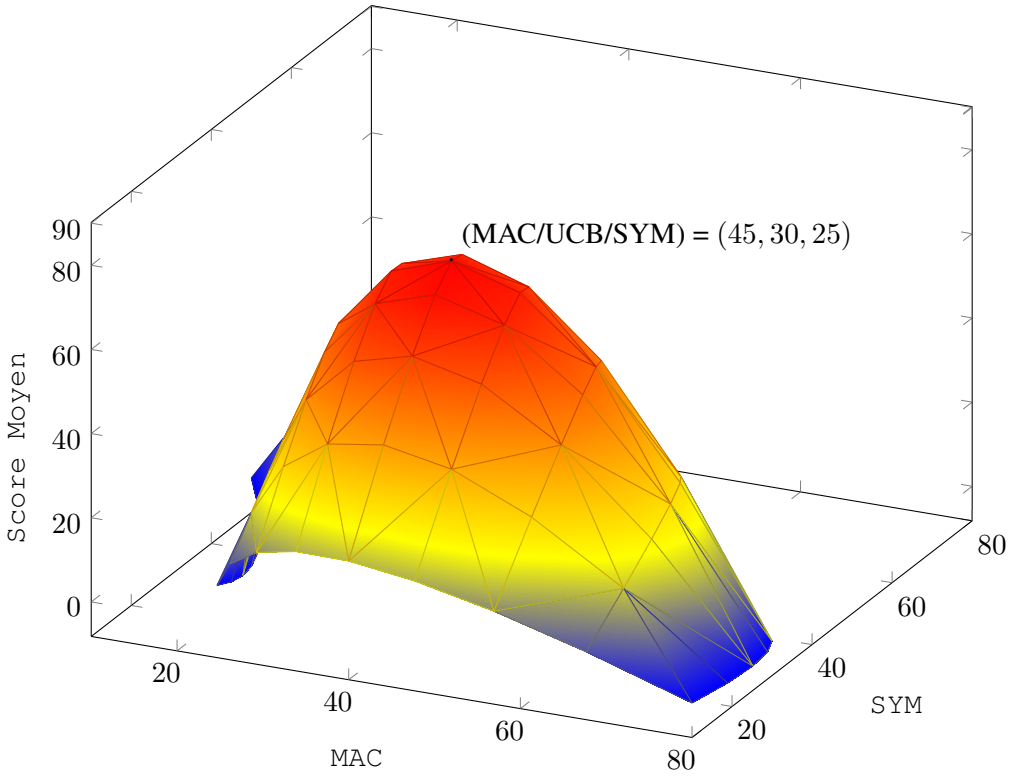


FIGURE 5.7 – Analyse de sensibilité de MAC−UCB−SYM.

5.5 Résultats expérimentaux

Afin de mettre en avant expérimentalement notre algorithme, nous présentons maintenant une série de résultats expérimentaux conduits sur un cluster Intel Xeon E5-2643 CPU 3.3 GHz associé à 64 GB de mémoire RAM sous Linux. MAC−UCB−SYM a été implémenté en C++ et nous utilisons NAUTY [McKAY & PIPERNO 2014] pour réaliser la détection de symétries.

5.5.1 Détection des symétries entre la méthode RG et les contraintes

Avant tout, avec pour objectif d'illustrer l'efficacité de la détection de symétries via les réseaux de contraintes face à la méthode RG utilisant un graphe de règles, dans la table 5.2 nous étudions le nombre de groupes de symétries détectées en 15 minutes par les deux méthodes pour différentes valeurs de n sur le puzzle GDL correspondant au problème des n-reines, un problème usuel dans le cadre de la détection de symétries comme illustré dans la section 1.

Remarque 5.2 *Il est important de rappeler au lecteur que les réseaux de contraintes traduits de par les programmes GDL décrivant les puzzles correspondant aux problèmes des n-reines étudiés ici différent des CSP utilisés dans la section 2 permettant de représenter ces mêmes problèmes. C'est pourquoi le*

nombre de groupe de symétries décrit ci-après ne correspond pas forcément au nombre de symétries de solutions décrit dans la section 2.

Problème des n-reines	**Méthode RG**	**MAC-UCB-SYM**
5	2	2
6	1	1
8	7	12
12	131	1,787
16	131	1,846,955
31	0	842,610,101,794,437

TABLE 5.2 – Le nombre de groupes de symétries détectées pour différentes tailles du problème des n-reines en utilisant la méthode RG et MAC-UCB-SYM en 15 minutes

Il est évident que la méthode RG détecte moins de groupes de symétries que MAC-UCB-SYM. Dès le problème usuel des n-reines impliquant 8 reines, notre approche détecte 12 groupes de symétries alors que la méthode RG ne parvient qu'à en identifier 7 d'entre elles. De plus dans le jeu des n-reines, plus n grandit, plus le nombre de symétries augmente. Cependant, RG ne détecte jamais plus de 131 symétries contrairement à la méthode SCSP dont le nombre de symétries détectées augmente considérablement. Pour finir, pour la plus grande instance étudiée impliquant 31 reines, la méthode RG ne possède pas assez de temps pour détecter la moindre symétrie face à MAC-UCB-SYM qui en détecte un nombre très important.

Concernant les expérimentations suivantes, nous avons confronté MAC-UCB-SYM à son homologue MAC-UCB n'utilisant pas la détection de symétries mais également face aux même approches GGP que dans le chapitre précédent (UCT, CFR et GRAVE) dans leur version classique et dans une version utilisant la détection de symétries de par la méthode RG (UCT-SYM, CFR-SYM et GRAVE-SYM). Finalement, l'un des derniers champions GGP déjà illustré dans le chapitre précédent (Sancho) est également confronté à MAC-UCB-SYM.

5.5.2 Étude de la détection de symétries par la méthode RG

Au cours de cette partie, nous étudions l'apport de la détection de symétries via la méthode RG [SCHIFFEL 2010] aux différentes approches GGP expérimentées dans le cadre d'une compétition GGP. Les tableaux 5.3 et 5.4 reprennent les résultats des confrontations réalisées entre MAC-UCB-SYM et chacune des approches GGP contre elle-même associée à la détection de symétries via la méthode RG. Chaque tableau indique le nombre moyen de victoires pour chaque joueur GGP classique contre sa version à base de symétries. Le tableau 5.3 se concentre sur 20 jeux GDL-I présentant de nombreuses symétries alors que le tableau 5.4 se compose de 5 instances GDL-II représentatif de jeux à information imparfaite. Pour chaque jeu GDL-I, 300 matchs sont réalisés et pour chaque jeu GDL-II 1.000 matchs pour un total de 44.000 matchs différents. Les mêmes temps de pré-traitement et de délibération sont utilisés que lors de la finale de la compétition *Open Tiltyard* 2015 à savoir : $startclock$ = $180s$ et $playclock$ = $15s$.

Pour quelques jeux comme le Quarto ou encore le Shmup, la détection de symétries via la méthode RG aide l'exploration de l'arbre de recherche associé à chacun de ces jeux, c'est pourquoi GRAVE, CFR et particulièrement UCT sont vaincus par leur version associée à la recherche de symétries. Mais clairement, pour des jeux plus imposants, comme par exemple le Amazons Torus, les échecs (Chess), ou le Pickomino, chaque approche GGP obtient de bien meilleurs résultats qu'en les associant à la détection

Game	MAC-UCB-SYM	UCT	GRAVE	CFR
Amazons Torus 10x10	98	95	96	95
Breakthrough Suicide	94	46	84	62
Chess	100	85	100	96
Connect Four 20x20	98	90	94	92
Connect Four Simultaneous	100	97	100	96
Copolymer 4-length	87	90	92	93
Dots and Boxes Suicide	84	42	56	49
English Draughts	98	71	87	90
Free For All 2P	64	58	69	65
Hex	96	52	86	55
Knight Through	89	42	53	52
Majorities	95	54	83	73
Pentago	98	55	57	54
Quarto	85	31	54	47
Reversi Suicide	92	76	81	86
Sheep and Wolf	78	54	61	76
Shmup	74	38	42	43
Skirmish zero-sum	100	97	100	100
TicTac Chess 2P	94	95	94	97
TTCC4 2P	93	91	95	93

TABLE 5.3 – Les résultats de chaque joueur GGP contre lui-même associé à la détection de symétries via la méthode RG sur des jeux à information complète

Jeu	MAC-UCB-SYM	UCT	GRAVE	CFR
Backgammon	95.3	53.7	79.2	86.2
Can't Stop	91.9	48.2	56.2	61.4
Kaseklau	51.4	31.3	41.3	43.4
Pickomino	83.7	78.4	93.4	94.8
Yahtzee	72.1	43.1	46.4	46.9

TABLE 5.4 – Les résultats de chaque joueur GGP contre lui-même associé à la détection de symétries via la méthode RG sur des jeux à information imparfaite

de symétries via la méthode RG (un exemple notable est le Skirmish zero-sum où CFR et GRAVE gagnent dans 100% des cas). Ceci s'explique par le nombre d'automorphismes très faible (voir nul) détecté par RG, c'est pourquoi UCT, GRAVE ou CFR jouent de manière bien plus stratégique grâce au temps supplémentaire utilisé pour explorer l'arbre de recherche plutôt que rechercher inutilement des symétries via RG.

Finalement, MAC-UCB-SYM surpasse en tout point sa version associée à la détection de symétries par la méthode RG pour l'ensemble des jeux avec souvent un score supérieur à 90% excepté pour le Kaseklau où le nombre de symétries détecté pour ces deux méthodes est similaire.

5.5.3 MAC-UCB-SYM dans le cadre GDL

Au cours de cette section, nous exposons les résultats de MAC-UCB-SYM sur chaque instance GDL contre chaque approche GGP dans sa version classique et dans sa version associée à la recherche de symétries par la méthode RG (sauf pour Sancho uniquement dans sa version classique). Nous avons réalisé 300 matchs par instance présente sur le serveur *Tiltyard* dans leurs dernières version et qui ont été jouées au moins une fois. Les temps utilisés sont les mêmes que précédemment ($startclock$ = $180s$ et $playclock$ = $15s$). Finalement, afin de garantir l'équité, les rôles que représentent chaque approche GGP au cours de chaque match sont échangés après chaque match.

Au vue des résultats obtenus par les autres approches GGP associées à la recherche de symétries par la méthode RG, afin de favoriser la meilleure approche entre la version classique et la version utilisant la méthode RG, les tableaux 5.5, 5.6, 5.7 et 5.8 indiquent les moins bons résultats moyens obtenus par MAC-UCB-SYM contre l'une ou autre des deux versions de chaque approche GGP, noté par exemple pour l'approche UCT : UCT*.

Résultats de MAC-UCB-SYM sur les jeux GDL à un joueur

Le tableau 5.5 se concentre uniquement sur les résultats rencontrés par chaque approche GGP et par MAC-UCB-SYM sur les instances GDL à un joueur.

Tout d'abord mettons en avant que UCT, CFR et GRAVE obtiennent exactement les mêmes résultats avec ou sans recherche de symétries, la méthode RG n'apporte donc aucune amélioration. Par contre, MAC-UCB associé à la recherche de symétries (MAC-UCB-SYM) présente une nette amélioration. Il obtient le score optimal pour l'ensemble des instances GDL excepté pour le Rubik's Cube où toutefois, il augmente indéniablement son score face à la version MAC-UCB classique. Notamment dans les puzzles associés à un arbre de recherche très important comme le 31 Queens Puzzle lg ou le Hidato 37 hexes MAC-UCB-SYM détecte un grand nombre de symétries lui permettant d'explorer l'arbre de recherche beaucoup plus rapidement et d'atteindre une solution satisfaisante dans le temps imparti. En moyenne 75% de l'arbre de recherche n'est pas exploré par MAC-UCB-SYM face à MAC-UCB sur les instances à un joueur.

Résultats de MAC-UCB-SYM sur les jeux GDL multi-joueurs

Les tableaux 5.6, 5.7 et 5.8 rapportent les résultats de MAC-UCB-SYM face à chaque approche GGP. La lecture des données se fait identiquement aux expérimentations du chapitre précédent portant sur MAC-UCB, par exemple dans le tableau 5.6, la ligne 2 colonne 2 indique que MAC-UCB-SYM obtient un score moyen de 95.3 dans le pire des cas face à UCT avec ou sans recherche de symétries via la méthode RG.

Pour l'ensemble des instances GDL du serveur *Tiltyard*, MAC-UCB-SYM surpasse clairement sa version sans recherche de symétries toujours avec un score moyen plus grand que 50. De plus pour les

Jeu	MAC-UCB-SYM	UCT*	CFR*	GRAVE*	Sancho
6 Queens Puzzle ug	100	100	100	100	100
8 Queens Puzzle lg	100	100	100	100	100
8 Queens Puzzle ug	100	100	100	100	100
12 Queens Puzzle ug	100	0	100	100	0
16 Queens Puzzle ug	100	0	0	0	0
31 Queens Puzzle lg	100	0	0	0	0
Asteroids	100	100	100	100	100
Cell Puzzle	100	100	100	100	100
Chinese Checkers 1P	100	100	100	100	100
Coins	100	100	100	100	100
Eight Puzzle	100	100	100	100	100
European Peg Jumping	90	90	90	90	90
Futoshiki 4x4	100	100	100	100	100
Futoshiki 5x5	100	100	100	100	100
Futoshiki 6x6	100	0	100	0	0
Hidato (19 hexes)	100	100	100	100	100
Hidato (37 hexes)	100	0	0	0	0
Knight's Tour	100	100	100	100	100
Knight's Tour Large	100	100	100	100	100
Lights On	100	100	100	100	100
Lights On Parallel	100	100	100	100	100
Lights On Simult-4	100	100	100	100	100
Lights On Simultaneous	100	100	100	100	100
Lights On Simultaneous II	100	100	100	100	100
Lights Out	100	100	100	100	100
Max knights	100	100	100	100	100
Maze	100	100	100	100	100
Mine clearing (small)	84	78	78	84	78
Nonogram 5x5	100	100	100	100	100
Nonogram 10x10	100	0	0	0	0
Peg Jumping	100	100	100	100	100
Rubik's Cube	80	35	50	50	45
Snake Parallel	100	100	100	100	100
Solitaire Chinese Checkers	100	70	90	100	90
Sudoku Grade 1	100	100	100	100	100
Sudoku Grade 2	100	100	100	100	100
Sudoku Grade 3	100	100	100	100	100
Sudoku Grade 4	100	100	100	100	100
Sudoku Grade 5	100	0	100	100	100
Sudoku Grade 6E	100	0	100	100	100
Sudoku Grade 6H	100	0	0	100	100
Untwisty Complex 2	100	100	100	100	100

TABLE 5.5 – Résultats de MAC-UCB-SYM sur chaque jeu GDL à un joueur

jeux possédants un arbre de recherche de profondeur importante (par exemple, l'`Amazons Torus 10x10`), un score supérieur à 80. En effet, ceci peut s'expliquer par l'amélioration significative de plus de 80% vis à vis du nombre de nœuds explorés par `MAC-UCB-SYM` face à son antécédent sur les plus gros jeux, indiquant que la détection de symétries est inestimable pour de tels jeux.

Les quelques exceptions peuvent s'expliquer d'elles-mêmes. Soit par le fait que les jeux ne sont pas significatifs, car il n'existe pas de meilleures stratégies que de jouer aléatoirement (`Guess two thirds of the average` ou `The Centipede Game` par exemple), soit par le fait que l'arbre de recherche est très petit (`Tic-Tac-Toe` ou `Shmup`) ou encore que le nombre de symétries est faible voir nul expliquant une faible amélioration vis à vis de `MAC-UCB` (`Free for all`, `War of Attrition`). Pour ces différentes instances `MAC-UCB-SYM` obtient un score moyen semblable à son prédécesseur ce qui, par la même occasion, indique que l'exploration de symétries même dans le cas où celles-ci sont inexistantes ne provoque pas une baisse trop importante de la stratégie obtenue par l'algorithme.

`UCT` et `CFR`, quant à eux, sont vaincus par `MAC-UCB-SYM`, avec parfois un score moyen de 100 (ex : `Breakthrough` ou `Hex`). `Sancho`, le champion de la compétition `GGP` international 2014, est en moyenne moins compétitif que `MAC-UCB-SYM` spécialement pour des jeux impliquants un arbre de recherche très large. Citons quelques exemples : `Amazons Torus 10x10` (81.0), `Chess` (87.2), `Connect Four Larger` (95.9) ou encore `Skirmish zero-sum` (67.4). Enfin, `GRAVE` est le meilleur rival de `MAC-UCB-SYM`, mais notre approche obtient un score moyen plus important que lui pour les jeux à information complète.

5.5.4 `MAC-UCB-II-SYM` dans le cadre `GDL-II`

Résultats de `MAC-UCB-II-SYM` sur les jeux `GDL-II` à un joueur

Le tableau 5.9 rapporte les résultats de `MAC-UCB-II-SYM` et des autres approches `GGP` sur les 5 jeux `GDL-II` à un joueur expérimentés dans ce manuscrit.

Ici seules les instances `Vaccum Cleaner Random` et `Wumpus` sont assez grandes pour exploiter la détection de symétries. `MAC-UCB-II-SYM` est l'approche `GGP` la plus performante et elle réalise une réelle amélioration sur `MAC-UCB-II`. Cependant, le `MasterMind` et le `Monty Hall` ne possédant qu'une très faible profondeur d'arbre et très peu de symétries détectées, `MAC-UCB-II-SYM` n'obtient pas un score plus élevé que `MAC-UCB-II` ou `CFR`. Notons finalement qu'aucune symétrie, n'est détectée dans le jeu `GuessDice` (où il faut en un tour deviner la valeur d'un dé), les résultats sur cette instance sont donc très peu significatifs.

Résultats de `MAC-UCB-II-SYM` sur les jeux `GDL-II` à information imparfaite

Le tableau 5.10 indique les résultats de `MAC-UCB-II-SYM` face aux autres approches `GGP` sur les 5 jeux `GDL-II` à information imparfaite.

Pour l'ensemble des instances, les quatre adversaires de `MAC-UCB-II-SYM` sont vaincus avec un score supérieur à 70 contre `Sancho`, `UCT` et `CFR` et supérieur à 55 contre `GRAVE`. En effet, dans les jeux à information imparfaite, il est typique de rencontrer de nombreuses symétries sur les états probabilistes générés par la conséquence des actions du joueur environnement, ce qui explique la performance de `MAC-UCB-II-SYM` qui détecte un grand nombre d'entre elles au cours de ces matchs. Un exemple remarquable est le `Yahtzee`, où une amélioration d'exploration de l'arbre de recherche de plus de 90% est observée (contre `MAC-UCB-II`) à l'aide de la détection de symétries ce qui provoque les très bons résultats que `MAC-UCB-II-SYM` rencontrent face aux autres programmes-joueurs.

Jeu	MAC-UCB*	UCT*	CFR*	GRAVE*	Sancho
Amazons 8x8	78.3	90.1	94.1	72.3	70.3
Amazons 10x10	76.5	95.3	98.2	74.3	71.4
Amazons Suicide 10x10	86.4	96.4	97.4	79.1	80.4
Amazons Torus 10x10	81.2	96.4	94.9	76.3	81.0
Battle	91.3	100	86.4	80.6	76.8
Battlebrushes 4P	46.5	53.1	50.2	48.6	46.2
Beat-Mania	81.3	100	98.4	86.4	78.3
Bidding Tic-Tac-Toe	69.5	86.1	73.4	75.9	67.8
Bidding Tic-Tac-Toe w/ 10 coins	72.4	91.1	75.1	78.8	72.6
Blocker	82.4	76.3	85.4	64.3	61.8
Bomberman	69.4	61.4	59.7	60.3	59.7
Breakthrough	85.3	100	100	93.5	86.4
Breakthrough small	78.6	97.6	84.2	66.4	60.4
Breakthrough small with holes	79.4	97.4	85.5	69.3	61.3
Breakthrough with holes	85.7	100	100	93.2	86.4
Breakthrough with walls	86.1	100	100	93.6	87.0
Breakthrough suicide	92.1	93.5	95.4	57.8	78.3
Breakthrough suicide small	86.4	100	100	60.2	80.9
Cephalopod micro	61.9	68.2	59.4	67.8	66.7
Checkers	85.3	90.4	81.3	79.6	73.8
Checkers must-jump	84.9	90.3	82.1	78.6	71.4
Checkers on a Barrel without kings	90.3	91.6	87.4	90.3	83.1
Checkers on a Cylinder, Must-jump	87.6	98.4	93.4	81.6	72.0
Checkers small	53.4	55.7	50.1	50.5	51.8
Checkers tiny	50.9	54.6	50.2	49.8	50.2
Checkers with modified goals	86.4	94.3	83.5	80.4	75.1
Chess	77.4	94.1	89.8	86.4	87.2
Chicken Tic-Tac-Toe	52.1	50.1	50.8	50.4	50.3
Chinese Checkers v1 2P	85.1	86.8	84.2	73.5	68.9
Chinese Checkers v1 3P	82.1	74.2	62.7	67.1	59.1
Chinese Checkers v1 4P	81.4	59.7	53.4	50.2	55.1
Chinese Checkers v1 6P	80.3	61.3	52.1	50.3	54.2
Chinook	86.2	93.2	79.8	80.1	73.2
Cit-Tac-Eot	53.1	50.8	51.3	50.7	51.9
Connect Five	78.4	69.1	75.3	74.1	74.1
Connect Four	75.7	68.9	75.0	73.8	73.5
Connect Four 3P	76.4	80.0	72.3	59.2	58.0
Connect Four 9x6	82.1	96.4	79.4	59.8	68.1
Connect Four Large	83.0	87.1	83.0	60.1	70.2
Connect Four Larger	86.1	97.9	96.1	62.3	95.9
Connect Four Simultanenous	73.5	89.0	64.3	74.2	80.8
Connect Four Suicide	78.2	70.1	68.5	68.4	63.1
Copolymer 4-length	74.0	87.0	87.6	80.3	76.2

TABLE 5.6 – Résultats de MAC-UCB-SYM sur chaque jeu GDL (A à C).

Jeu	MAC-UCB*	UCT*	CFR*	GRAVE*	Sancho
Dollar Auction	53.4	82.8	82.6	82.6	82.5
Dots and Boxes	71.2	79.2	80.1	81.8	70.1
Dots and Boxes Suicide	62.9	82.1	80.2	69.2	85.8
Dual Connect 4	78.6	71.6	77.2	75.1	74.8
English Draughts	83.7	97.6	98.8	66.1	58.4
Escort-Latch breakthrough	85.7	97.5	98.6	94.0	87.1
Free for all 2P	52.1	80.6	81.8	59.4	70.9
Free for all 3P	51.6	78.1	80.2	57.1	68.6
Free for all 4P	52.4	79.1	80.2	57.4	67.1
Free for all zero sum	54.2	81.8	82.0	60.1	72.1
Ghost Maze	60.1	80.4	73.1	72.4	71.4
Golden Rectangle	74.1	84.1	73.7	72.4	69.1
Guess two thirds of the average 2P	49.2	49.7	50.1	50.0	49.6
Guess two thirds of the average 4P	25.1	24.6	25.0	24.8	24.7
Guess two thirds of the average 6P	16.4	16.3	16.8	17.0	16.4
Hex	83.8	100	100	72.3	79.1
Hex with pie	83.1	100	72.1	100	78.2
Iterated Chicken	51.1	70.0	62.1	66.5	65.0
Iterated Coordination	61.2	75.5	73.1	74.3	70.4
Iterated Prisoner's Dilemma	51.3	50.3	48.1	45.4	42.6
Iterated Stag Hunt	53.4	71.2	67.4	66.7	62.1
Iterated Tinfoll Hat Game	53.1	70.1	68.3	65.4	66.7
Iterated Ultimatum Game	54.1	57.0	62.4	63.7	59.7
Kalah 5x2x3	82.0	94.1	94.1	90.1	87.4
Kalah 6x2x4	83.1	95.7	95.0	91.4	88.1
Knight Fight	74.5	86.1	78.7	79.3	78.4
Knight Through	81.2	89.7	90.1	80.2	86.6
Majorities	83.2	97.1	93.5	80.4	83.8
Nine-Board Tic-Tac-Toe	74.1	81.2	83.4	80.1	69.1
Number Tic-Tac-Toe	51.1	51.4	50.1	50.0	50.3
Pacman 2P	64.1	77.1	75.0	72.6	74.2
Pacman 3P	62.3	73.1	76.4	72.4	72.4
Pawn Whopping	63.4	75.1	74.7	73.4	66.4
Pawn-to-Queen	61.3	63.9	56.4	61.9	60.1
Pentago	52.3	64.4	72.8	59.4	52.1
Pentago Suicide	53.1	64.5	72.6	58.1	53.1
Quarto	53.1	64.1	71.1	63.0	56.1
Quarto Suicide	53.0	63.8	70.3	63.2	55.7
Qyshinsu	68.1	94.1	86.4	81.4	78.1
Reversi	84.1	90.1	91.4	72.1	64.1
Reversi Suicide	71.4	100	100	62.4	56.8

TABLE 5.7 – Résultats de MAC-UCB-SYM sur chaque jeu GDL (D à R).

Jeu	MAC-UCB*	UCT*	CFR*	GRAVE*	Sancho
Sheep and Wolf	74.1	94.2	93.1	61.7	55.3
Shmup	56.3	62.4	75.7	50.4	50.9
Skirmish variant	82.4	97.7	98.6	86.4	84.1
Skirmish zero-sum	84.2	98.4	99.1	79.4	67.4
Snake 2P	54.1	56.1	54.2	51.3	50.8
Snake Assemblit	62.4	76.4	71.9	65.7	66.0
Speed Chess	79.8	95.2	90.2	86.7	87.6
The Centipede Game	16.4	13.4	13.9	16.1	13.2
Tic-Tac-Chess 2P	94.6	97.6	94.3	74.4	85.3
Tic-Tac-Chess 3P	78.4	80.1	78.4	50.4	43.7
Tic-Tac-Chess 4P	83.4	76.4	73.1	68.6	61.3
Tic-Tac-Heaven	59.3	66.3	70.2	70.8	67.3
Tic-Tac-Toe	50.0	50.1	50.0	50.0	50.2
Tic-Tac-Toe 3P	43.4	54.7	53.9	53.9	54.3
Tic-Tac-Toe 9 boards with pie	72.1	85.9	84.2	80.0	72.3
Tic-Tac-Toe Large	51.4	51.2	50.6	50.4	50.4
Tic-Tac-Toe LargeSuicide	52.1	51.7	51.1	50.6	51.2
Tic-Tac-Toe Parallel	50.0	50.0	50.0	50.0	50.0
Tic-Tac-Toe Serial	50.0	50.0	50.0	50.0	50.0
Tic-Tic-Toe	50.0	50.0	50.0	50.0	50.0
Tron 10x10	63.4	88.2	76.1	70.3	79.1
TTCC4 2P	81.7	97.2	97.6	60.4	67.6
War of Attrition	52.4	57.6	55.7	48.2	48.9
With Conviction!	50.0	50.0	50.0	50.0	50.0

TABLE 5.8 – Résultats de MAC-UCB-SYM sur chaque jeu GDL (T à Z).

Jeu	MAC-UCB-II-SYM	MAC-UCB-II*	UCT*	CFR*	GRAVE*
GuessDice	15.6	15.0	15.7	16.0	16.5
MasterMind	63.9	67.8	53.8	68.1	60.1
Monty Hall	65.1	65.2	62.5	63.1	64.3
Vaccum Cleaner Random	74.8	61.5	34	46	58.8
Wumpus	71.2	32.1	40.1	44.1	51.2

TABLE 5.9 – Résultats de MAC-UCB-II-SYM sur les jeux GDL-II à un joueur.

Jeu	MAC-UCB-II*	UCT*	CFR*	GRAVE*	Sancho
Backgammon	91.2	97.2	94.1	68.2	100
Can't Stop	88.5	93.5	88.5	92.1	100
Kaseklau	72.1	60.2	69.4	56.3	86.4
Pickomino	74.6	73.7	71.1	86.4	91.4
Yahtzee	86.7	81.9	89.4	59.1	90.4

TABLE 5.10 – Résultats de MAC-UCB-II-SYM sur les jeux GDL-II à information imparfaite.

Résultats de MAC-UCB-II-SYM sur les jeux GDL-II à information partagée

Enfin, le tableau 5.11 répertorie les résultats de MAC-UCB-II-SYM face aux autres approches GGP sur les 5 derniers instances GDL-II membres des jeux à information partagée.

Jeu	MAC-UCB-II*	UCT*	CFR*	GRAVE*
Pacman	62.3	61.2	64.2	60.4
Schnappt Hubi	79.3	84.4	76.2	74.7
Sheep & Wolf	84.1	90.2	74.3	73.2
Tic-Tac-Toe Latent Random 10x10	84.5	95.1	91.7	84.6
War (card game)	63.2	71.3	68.4	62.4

TABLE 5.11 – Résultats de MAC-UCB-II-SYM sur les jeux GDL-II à information partagée.

Ici, en dehors de Pacman qui ne rencontre que 20% d'amélioration face à MAC-UCB, il est possible d'observer que la détection de symétries pour les POSGs est également payante, caractérisée par une réduction substantielle de l'arbre de recherche. En effet, MAC-UCB-SYM remporte l'ensemble des matchs en moyenne avec un score supérieur à 60 contre chaque adversaire, et particulièrement plus, avec au minimum un score 84 sur le Tic-Tac-Toe Latent Random 10x10.

5.5.5 WoodStock et MAC-UCB-SYM, Champion GGP 2016

Lors de la dernière compétition internationale organisée par l'université de *Stanford* (IGGPC'16), nous avons pu mettre en pratique MAC-UCB-SYM par l'intermédiaire de WoodStock, notre programme-joueur (voir section 4.5). La compétition 2016 a regroupé 10 concurrents comprenant les champions GGP 2011, 2013, 2014 et 2015 : Alloy, DROP-TABLE-TEAMS, Galvanise, General, MaastPlayer, QFWFQ, Sancho, SteadyEddie, TurboTurtle et WoodStock. Ils se sont affrontés au travers de nombreux jeux à un ou deux joueurs au cours de deux journées. Tous les jeux à deux joueurs étaient garanties d'être des jeux à somme nulle. La plupart des jeux GDL était nouveau et mis en pratique pour la première fois en compétition. 120 secondes de temps de pré-traitement (*StartClock*) et 30 secondes de temps de délibération (*PlayClock*) ont été allouées.

Jour 1

Lors de la première journée, deux groupes regroupant chacun 5 joueurs ont été réalisé arbitrairement. Le classement final de chaque groupe est obtenu à la suite du calcul de la somme pondérée des scores de chaque joueur. Les deux joueurs obtenant les scores maximaux sont qualifiés au sein de la « poule des vainqueurs » et les deux suivants au sein de la « poule des vaincus » pour la seconde journée. Les scores obtenus par WoodStock sur chaque jeu à l'issue de la première journée sont présentés dans le tableau 5.12. WoodStock a obtenu 1.200 points suite à cette première journée et il fut le seul programme-joueur à obtenir le score maximum sur l'ensemble des puzzles. Ainsi, il fut qualifié au sein de la « poule des vainqueurs » pour la seconde journée au coté de DROP-TABLE-TEAMS, Galvanise et TurboTurtle.

Jour 2

La seconde journée s'est déroulée sous la forme de duels remportés par le premier programme obtenant 3 victoires (excepté pour la finale où 4 victoires sont nécessaires). Le tableau 5.13 répertorie l'ensemble des matchs joués par WoodStock. Au cours des quarts de finale, WoodStock a affronté

Jeu	#Joueurs	WoodStock	Poids
Vectorracer6	1	100	1
EightPuzzle	1	100	1
HunterBig	1	100	1
UntwistyComplex	1	100	0.5
Jointbuttonsandlights25	1	100	0.5
Sudoku	1	100	0.5
Bandl3	1	100	0.33
bandl7	1	100	0.33
bandl10	1	100	0.33
Majorities	2	100 / 300	1
Battleofnumbers	2	150 / 300	1
Jointconnectfour77	2	100 / 300	1
Breakthrough	2	300 / 300	1

TABLE 5.12 – WoodStock- IGGPC 2016 - Jour 1.

DROP-TABLE-TEAMS et a obtenu 3 victoires et 2 défaites lui permettant d'atteindre les demis finales face à TurboTurtle. Au cours de ces dernières, WoodStock n'a rencontré aucune défaite et a obtenu 3 victoires. Finalement, lors de la finale l'opposant à Galvanise, WoodStock est devenu Champion GGP à la suite de 2 défaites et de 4 victoires.

Une chose importante à noter et que la détection de symétries a été primordiale au cours de la compétition. Un exemple notable est le dernier jeu pratiqué par WoodStock. Lors des deux matchs de Jointconnectfour, un grand nombre de symétries détectées ont permis de réduire fortement l'espace de recherche et d'explorer complètement l'arbre de recherche au bout d'une trentaine de coups joués contrairement à son opposant.

Jeu	WoodStock	Adversaire
Quart de finale contre DROP-TABLE-TEAMS		
Hexawesome	100	0
Reflectconnect6	30	70
Connectfour77	100	0
Connectfour77	0	100
Majorities	100	0
Demi finale contre TurboTurtle		
Battleofnumbersbig	100	0
Platformjumpers	100	0
Jointconnectfour77	100	0
Finale contre Galvanise		
Reversi	0	100
Reversi	0	100
Skirmish	100	0
Skirmish	100	0
Jointconnectfour	100	0
Jointconnectfour	100	0

TABLE 5.13 – WoodStock- IGGPC 2016 - Jour 2.

5.6 Conclusion

Dans ce chapitre, nous proposons une alternative à la détection de symétries dans le cadre du *General Game Playing*. Tout d'abord nous mettons théoriquement en relation la notion de symétries de jeux et la notions de symétries dans la programmation par contraintes. Suite à l'ajout de contraintes modélisant les stratégies minimax des différents joueurs au réseau de contraintes stochastiques obtenu après traduction d'un jeu GDL, nous montrons que toute symétrie de contraintes de ce réseau est une symétrie minimax du jeu GDL. Enfin, nous proposons d'étendre la notion de symétrie de contraintes d'un $\mu\texttt{SCSP}_t$ $\mathcal{P}$ à une symétrie de contraintes d'un $\texttt{SCSP}_T$ à T niveaux (avec $t \in T$) où le niveau t est décrit par $\mathcal{P}$. Notre approche, MAC-UCB-SYM, une variante de MAC-UCB, met en pratique la détection de symétries minimax via l'ajout de *nogoods* successivement au cours de la résolution. Nous montrons que MAC-UCB-SYM surpasse son antécédent et l'ensemble des approches GGP (associées ou non à la détection de symétries de par la méthode RG) composant l'état de l'art sur un grand nombre de jeux GDL et GDL-II. Par conséquent, la détection de symétries est montré primordiale en offrant une amélioration conséquente pour la résolution des jeux à information complète, imparfaite ou partagée. Finalement, notre programme-joueur, WoodStock, est devenu champion GGP 2016 à l'aide de MAC-UCB-SYM lors de la compétition internationale organisée par l'université de *Stanford*.

À la lumière de ces résultats, un axe de recherche est l'étude théorique de la détection de symétries pour des algorithmes de type Monte Carlo comme UCT, UCB ou GRAVE. Un autre axe de recherche découlant de ces travaux est la décomposition de jeux en sous-jeux afin de les résoudre parallèlement ou encore d'exploiter les symétries afin d'établir automatiquement des heuristiques et d'analyser les jeux GDL.

Conclusion

Ce manuscrit détaille nos différentes contributions en proposant la programmation par contraintes stochastiques comme une nouvelle approche pour le *General Game Playing*.

La première phase de ce travail a consisté à concevoir une modélisation par contraintes stochastiques équivalente à un jeu GDL. À cette fin, nous avons proposé une sémantique originale basée sur les jeux de *Markov* au formalisme GDL et nous avons décrit un processus de traduction d'un jeu GDL à tout temps t en un réseau de contraintes stochastiques à un niveau. En répétant ce processus, nous avons montré qu'un SCSP équivalent au jeu GDL original est obtenu. De plus, après avoir identifié un fragment praticable des jeux GDL-II englobant les jeux à information partagée, nous avons démontré qu'il est possible d'étendre notre processus de traduction à ce fragment permettant ainsi leurs résolutions. Puis via une série d'expérimentations sur l'ensemble des jeux GDL utilisés lors de la compétition continue du serveur *Tiltyard* ainsi que sur une quinzaine de jeux GDL-II représentatif du fragment étudié, nous avons mis en évidence que la traduction vers un SCSP est compatible avec les conditions temporelles imposées par les compétitions internationales GGP et que les réseaux obtenus sont d'une taille raisonnable, permettant d'envisager leurs résolutions. Finalement, nous avons montré que les stratégies de jeux sont représentables par un ensemble de contraintes additionnelles au réseau de contraintes stochastiques obtenu par traduction d'un programme GDL.

L'établissement d'un formalisme par contraintes pour le *General Game Playing* ouvre la voie à de nombreuses possibilités exploitant l'ensemble des techniques de résolutions associées. Ainsi, dans une seconde partie, nous avons proposé un algorithme itératif par niveau permettant la résolution du SCSP obtenu, dénommé MAC-UCB, en exploitant un fragment SCSP permettant l'utilisation de techniques de résolutions de CSP classiques. Ainsi, en combinant l'algorithme MAC et la méthode Monte Carlo UCB, notre approche a ouvert la voie à la programmation par contraintes pour les compétitions GGP. De plus, cette approche permet également la résolution de SCSP générés à partir de jeux GDL-II à information partagée suite à la conservation du fragment SCSP nécessaire à son application.

Afin d'étudier notre algorithme face aux autres approches établissant l'état de l'art du *General Game Playing*, nous avons réalisé de nombreuses expérimentations au travers de plusieurs millions de matchs indiquant la compétitivité de MAC-UCB sur les jeux à information complète, imparfaite ou partagée. Finalement, WoodStock, notre programme-joueur générique a été implémenté et permet l'application de MAC-UCB lors des compétitions GGP. Ainsi, depuis mars 2016 et jusqu'à ce jour, ce dernier se positionne comme leader de la compétition continue de *General Game Playing* tout en offrant de nombreuses perspectives d'évolution.

L'une d'entre elles a été d'exploiter les symétries de jeux. Ainsi, nos investigations se sont portées sur la mise en relation de la détection de symétries dans le cadre des réseaux de contraintes et des symétries de jeux. Tout d'abord, nous avons montré l'équivalence entre la notion de symétries de contraintes et de symétries de structure de jeux. Puis, suite à l'ajout de contraintes représentatives des stratégies minimax du jeu, nous avons mis en avant que la détection de symétries de contraintes dans le SCSP obtenu permet la détection des symétries minimax du jeu.

Suite à ces résultats théoriques, nous avons proposé une extension à notre algorithme couplant MAC−UCB à notre processus de détection de symétries, dénommée MAC−UCB−SYM. Grâce à l'ajout systématique d'instanciations globalement incohérentes à la micro-structure complément associée à chaque réseau de contraintes stochastiques, il permet d'affiner la détection de nouvelles symétries au cours du jeu réduisant ainsi l'espace de recherche associé à l'arbre de jeu. À l'aide d'un grand nombre de matchs expérimentaux entre la meilleure méthode GGP de détection de symétries associée aux approches GGP les plus performantes et notre nouvelle approche MAC−UCB−SYM, nous avons montré que cette dernière les surclassent toutes aisément. Finalement, à l'aide de MAC−UCB−SYM, WoodStock s'impose comme le nouveau leader GGP suite à sa première participation à la compétition GGP internationale 2016 (IGGPC 2016). Ce constat permet notamment d'indiquer que la détection de symétries est primordiale dans la « résolution » de jeux et envisage l'étude théorique de détection de symétries pour les algorithmes tel que UCT, UCB ou GRAVE.

Ces différentes contributions pavent le chemin à de nombreuses perspectives. Tout d'abord, naturellement, une étude plus poussée de l'ensemble des techniques existantes dans le cadre des réseaux de contraintes pour résoudre la partie décisionnelle des SCSP générés afin d'optimiser WoodStock et d'obtenir de meilleures performances. Dans le même axe de recherche, l'étude de techniques de parallélisation adaptées au fragment SCSP identifié associé à la parallélisation des méthodes d'échantillonnages est une voie non négligeable afin de permettre la comparaison de WoodStock aux programmes-joueurs parallèles.

Une autre voie d'optimisation serait l'étude d'une autre méthode d'échantillonnage plus adaptée que la méthode UCB utilisée actuellement. Une voie en cours d'étude est l'exploitation des bandits combinatoires qui semblent offrir de meilleures performances. Une perspective différente consiste à exploiter de nouveau la structure graphique associée au réseau de contraintes afin d'identifier automatiquement des propriétés de jeux qui pourraient être utilisées comme heuristiques de recherche dans l'arbre de jeu.

En outre, l'utilisation de la représentation graphique d'un CSP peut représenter une piste intéressante à exploiter pour réaliser l'analyse d'instances GDL. Une application possible serait d'identifier automatiquement si une instance GDL est bien formée ou non via la détection de cycles ou encore de décomposer un jeu en sous-jeux à travers la détection de cliques. Des travaux futurs seront conduits dans ce sens.

Finalement, récemment deux nouveaux formalismes VGDL [SCHAUL 2013] (pour *Video Game Definition Language*) et GDL−III [THIELSCHER 2016] (pour *Game Description Language with Imperfect Information and Introspection*) offrent un nouveau cadre prometteur à l'intelligence artificielle généraliste dans un contexte temps réel pour le premier et à la représentation des jeux épistémiques pour le second. L'étude d'un modèle à base de contraintes est une piste à envisager, tout comme l'application de notre algorithme aux applications robotiques qu'offre le *General Game Playing*.

Table des figures

Chapitre 1 : *General Game Playing*

1.1 Un arbre de recherche d'un jeu à un joueur. . . . 14
1.2 Un arbre de recherche d'un jeu à deux joueurs. . . . 14
1.3 Un arbre de recherche d'un jeu à information incomplète et imparfaite. . . . 16
1.4 Le programme GDL correspondant au jeu « Matching Pennies » . . . 23
1.5 Diagramme de l'environnement d'une compétition de GGP . . . 25
1.6 Le programme GDL-II correspondant au jeu « Hidden Matching Pennies » . . . 29
1.7 Exploration d'un arbre de recherche avec Minimax. . . . 33
1.8 Exploration d'un arbre de recherche avec Négamax. . . . 34
1.9 Exploration d'un arbre de recherche avec Alpha-Beta. . . . 34
1.10 Les quatres étapes des méthodes de type MCTS réalisées en cycle . . . 38
1.11 Parallélisation au niveau de l'arbre des techniques de MCTS . . . 43
1.12 Parallélisation au niveau de la racine des techniques de MCTS . . . 44
1.13 Parallélisation au niveau des feuilles des techniques de MCTS . . . 46
1.14 Un réseau de propositions $propnet$ . . . 49
1.15 Un $propnet$ à l'état initial . . . 51
1.16 Un $propnet$ après les actions . . . 51
1.17 Propagation de signal . . . 52
1.18 État suivant après propagation . . . 52

Chapitre 2 : Programmation par contraintes

2.1 Graphe de contraintes d'un CSP . . . 60
2.2 Microstructure de compatibilité d'un réseau de contraintes . . . 61
2.3 Graphe de contraintes du problème des 4-reines . . . 62
2.4 Les deux solutions du problème des 4-reines. . . . 62
2.5 Suppression de valeurs par cohérence de nœud. . . . 64
2.6 Un réseau non arc-cohérent . . . 66
2.7 On applique l'inférence sur la contrainte c_1 . . . 67
2.8 Un réseau $\mathcal{P}$ toujours non arc-cohérent . . . 67
2.9 On applique l'inférence sur la contrainte c_2 et c_3 . . . 67
2.10 $\phi(\mathcal{P})$ est un réseau arc-cohérent. . . . 67
2.11 Les structures STR initialisées pour la contrainte c . . . 76
2.12 Inversion dans *position* des tuples aux positions 0 et 6, puis mis à jour de *levelLimits* à 6 pour le niveau 1 . . . 77
2.13 *currentLimit* est décrémenté. La table courante ne contient plus que 6 tuples. . . . 77

2.14 STR appliqué à la contrainte c après la décision $y = c$ 77
2.15 STR appliqué à la contrainte c après la propagation de $x \neq i$ depuis une autre contrainte. La valeur (z, j) n'ayant plus de support, elle est supprimée de $dom(z)$ 78
2.16 État des structures de STR pour la contrainte c après la restauration conséquente au retour en arrière au niveau 1 78
2.17 Arbre partiel de recherche avec schéma binaire pour le problème des 4-reines (✓ indique une solution au problème et × une instanciation globalement incohérente) 83
2.18 Arbre de recherche partiel construit par l'algorithme MAC sur le problème des 4-reines. . 89
2.19 Deux politiques pour le SCSP de l'exemple 2.16. 93

Chapitre 3 : Modélisation par contraintes de jeux GDL
3.1 μSCSP encodant le « Hidden Matching Pennies » décrit en GDL-II en figure 1.6. . . . 116

Chapitre 4 : Une approche GGP dirigée par les contraintes
4.1 Les contraintes du μSCSP $\mathcal{P}$ 130
4.2 Les solutions du μSCSP $\mathcal{P}''$. 130
4.3 L'arbre de recherche obtenu par MAC-UCB-II sur le jeu du « hidden matching pennies » .133
4.4 Analyse de Sensibilité de MAC-UCB et de MAC-UCB-II. 134
4.5 Évolution du nombre de victoires de MAC-UCB contre UCT, avec $playclock = 30s$ . . . 136

Chapitre 5 : Détection de symétries par les contraintes pour le General Game Playing
5.1 micro-structure complémentaire d'un CSP $\mathcal{P}$ 159
5.2 Le complément du *2-nogood* hypergraphe du problème des 4-reines. 161
5.3 Les 10 solutions du problème des 5-reines. 162
5.4 Le programme GDL correspondant au jeu Tic-Tac-Toe 2x2 166
5.5 L'arbre de recherche du Tic-Tac-Toe 2x2 au tour $t = 1$ 166
5.6 L'arbre de recherche (complet) du Tic-Tac-Toe 2x2 167
5.7 Analyse de sensibilité de MAC-UCB-SYM. 169

Liste des tableaux

1 Programmes-joueurs à renommée mondiale suite à leurs victoires sur les champions humains. 2

Chapitre 1 : *General Game Playing*
1.1 Matrice de gain du *Matching Pennies* 9
1.2 Équilibre de Nash 11
1.3 Mots-clés de GDL 22
1.4 Les champions de *General Game Playing* 26
1.5 Mots-clés spécifiques à GDL-II 27

Chapitre 2 : Programmation par contraintes
2.1 Algorithmes établissant la cohérence d'arc et leur complexité. 68
2.2 Différentes associations des techniques de *look-ahead* et de *look-back* présentées. 90

Chapitre 3 : Modélisation par contraintes de jeux GDL
3.1 L'ensemble des variables et leurs domaines extraits du matching pennies en GDL 110
3.2 L'ensemble des variables et leurs domaines extraits du matching pennies en GDL 110
3.3 L'ensemble des contraintes extraites du *matching pennies* en GDL 111
3.4 Traduction de jeux GDL à un joueur vers SCSP 118
3.5 Traduction de jeux GDL multijoueurs (A à C) vers SCSP 119
3.6 Traduction de jeux GDL multijoueurs (D à R) vers SCSP 120
3.7 Traduction de jeux GDL multijoueurs (T à Z) vers SCSP 121
3.8 Traduction de jeux GDL-II vers SCSP 122

Chapitre 4 : Une approche GGP dirigée par les contraintes
4.1 Resultats sur plusieurs jeux GDL où $playclock = 30s$ 135
4.2 Les victoires de MAC-UCB vs. UCT avec différents $playclock$ sur 5.000 matchs 137
4.3 Résultats sur chaque jeu GDL à un joueur 139
4.4 Résultats de MAC-UCB sur chaque jeu GDL (A à C) 141
4.5 Résultats de MAC-UCB sur chaque jeu GDL (D à R) 142
4.6 Résultats de MAC-UCB sur chaque jeu GDL (T à Z) 143
4.7 Résultats de MAC-UCB-II sur les jeux GDL-II à un joueur 143
4.8 Résultats de MAC-UCB-II sur les jeux GDL-II à information imparfaite 144
4.9 Résultats de MAC-UCB-II sur les jeux GDL-II à information partagée 144
4.10 Les informations disponibles pour chaque jeu de la phase de qualification 148

4.11 Les scores finaux suite à la phase de qualification 149
4.12 Le classement final de la *2015 Open Tiltyard* 149
4.13 Le classement actuel (datant du 15 juillet 2016) du tournoi continu GGP 150

Chapitre 5 : Détection de symétries par les contraintes pour le General Game Playing
5.1 Nombre de symétrie de solutions pour le problème des n-reines pour différents n. 163
5.2 Le nombre de groupes de symétries détectées pour différentes tailles du problème des n-reines en utilisant la méthode RG et MAC−UCB−SYM en 15 minutes 170
5.3 Les résultats de chaque joueur GGP contre lui-même associé à la détection de symétries via la méthode RG sur des jeux à information complète 171
5.4 Les résultats de chaque joueur GGP contre lui-même associé à la détection de symétries via la méthode RG sur des jeux à information imparfaite 171
5.5 Résultats de MAC−UCB−SYM sur chaque jeu GDL à un joueur 173
5.6 Résultats de MAC−UCB−SYM sur chaque jeu GDL (A à C). 175
5.7 Résultats de MAC−UCB−SYM sur chaque jeu GDL (D à R). 176
5.8 Résultats de MAC−UCB−SYM sur chaque jeu GDL (T à Z). 177
5.9 Résultats de MAC−UCB−II−SYM sur les jeux GDL−II à un joueur. 177
5.10 Résultats de MAC−UCB−II−SYM sur les jeux GDL−II à information imparfaite. 177
5.11 Résultats de MAC−UCB−II−SYM sur les jeux GDL−II à information partagée. 178
5.12 WoodStock- IGGPC 2016 - Jour 1. 179
5.13 WoodStock- IGGPC 2016 - Jour 2. 179

Bibliographie

[AGUIRRE 1991] A. S. M. AGUIRRE. « How to Use Symmetries in Boolean Constraint Solving ». Dans *WCLP*, pages 287–306, 1991.

[ALOUL *et al.* 2004] F. A. ALOUL, A. RAMANI, I. L. MARKOV, et K. A. SAKALLAH. « ShatterPB : Symmetry-breaking for pseudo-Boolean Formulas ». Dans *ASP-DAC '04*, pages 883–886. IEEE Press, 2004.

[ARANGÚ *et al.* 2010] M. ARANGÚ, M. A. SALIDO, et F. BARBER. « AC2001-OP : An Arc-Consistency Algorithm for Constraint Satisfaction Problems ». Dans *Trends in Applied Intelligent Systems*, pages 219–228, 2010.

[ARNESON *et al.* 2010] B. ARNESON, R. B. HAYWARD, et P. HENDERSON. « Solving Hex : Beyond Humans ». Dans *CG'10*, pages 1–10, 2010.

[AZARIA & SIPPER 2005] Y. AZARIA et M. SIPPER. « Using GP-Gammon : Using Genetic Programming to Evolve Backgammon Players ». Dans Maarten KEIJZER, Andrea TETTAMANZI, Pierre COLLET, Jano I. VAN HEMERT, et Marco TOMASSINI, éditeurs, *EuroGP'05*, volume 3447 de *Lecture Notes in Computer Science*, pages 132–142. Springer, 2005.

[BACKOFEN & WILL 2002] R. BACKOFEN et S. WILL. « Excluding Symmetries in Constraint-Based Search ». *Constraints*, 7(3) :333–349, 2002.

[BALAFOUTIS & STERGIOU 2006] T. BALAFOUTIS et K. STERGIOU. « Algorithms for Stochastic CSPs ». Dans *CP'06*, pages 44–58, 2006.

[BARTÀK & ERBEN 2004] R. BARTÀK et R. ERBEN. « A New Algorithm for Singleton Arc Consistency. ». Dans Valerie BARR et Zdravko MARKOV, éditeurs, *FLAIRS Conference*, pages 257–262. AAAI Press, 2004.

[BEN-ELIYAHU & DECHTER 1994] R. BEN-ELIYAHU et R. DECHTER. « Propositional Semantics for Disjunctive Logic Programs ». *Annals of Mathematics and Artificial Intelligence*, 12 :53–87, 1994.

[BENHAMOU & SAIS 1992] B. BENHAMOU et L. SAIS. « Theoretical Study of Symmetries in Propositional Calculus and Applications ». Dans *CADE'11*, pages 281–294, 1992.

[BENHAMOU 1994] B. BENHAMOU. « Study of symmetry in Constraint Satisfaction Problems ». Dans *CP'94*, pages 246–254, 1994.

[BESSIERE & DEBRUYNE 2005] C. BESSIERE et R. DEBRUYNE. « Optimal and Suboptimal Singleton Arc Consistency Algorithms ». Dans *IJCAI'05*, pages 54–59, 2005.

[BESSIERE & REGIN 1996] C. BESSIERE et J. C. REGIN. « MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems ». Dans *CP'96*, pages 61–75, 1996.

[BESSIERE & VERGER 2006] C. BESSIERE et G. VERGER. « Strategic constraint satisfaction problems ». Dans *CP'06 Workshop on Modelling and Reformulation*, pages 17–29, 2006.

[BESSIERE 2006] C. BESSIERE. « Constraint propagation ». Dans *Handbook of Constraint Programming*, pages 29–83. Elsevier, 2006.

[BESSIÈRE & FREUDER 1999] C. BESSIÈRE et E. C. FREUDER. « Using Constraint Metaknowledge to Reduce Arc Consistency Computation ». *Artificial Intelligence*, 107 :125–148, 1999.

[BESSIÈRE *et al.* 2005] C. BESSIÈRE, J.-C. RÉGIN, R. H. C. YAP, et Y. ZHANG. « An optimal coarse-grained arc consistency algorithm. ». *Artificial Intelligence*, 165(2) :165–185, 2005.

[BESSIÈRE 1994] C. BESSIÈRE. « Arc-Consistency and Arc-Consistency Again ». *Artificial Intelligence*, 65 :179–190, 1994.

[BITNER & REINGOLD 1975] J. R. BITNER et E. M. REINGOLD. « Backtrack Programming Techniques ». *Communications of the ACM*, 18(11) :651–656, 1975.

[BJORNSSON & FINNSSON 2009] Y. BJORNSSON et H. FINNSSON. « CadiaPlayer : A Simulation-Based General Game Player ». *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1) :4–15, 2009.

[BLACK 1999] P. E. BLACK. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[BONNET & SAFFIDINE 2014] E. BONNET et A. SAFFIDINE. « On the complexity of general game playing ». Dans *CGW'14*, pages 90–104, 2014.

[BOURKI *et al.* 2010] A. BOURKI, G. CHASLOT, M. COULM, V. DANJEAN, H. DOGHMEN, T. HÉRAULT, J.-B. HOOCK, A. RIMMEL, F. TEYTAUD, O. TEYTAUD, P. VAYSSIÈRE, et Z. YU. « Scalability and Parallelization of Monte-Carlo Tree Search ». Dans *CG'10*, 2010.

[BOUSSEMART *et al.* 2004] F. BOUSSEMART, F. HEMERY, C. LECOUTRE, et L. SAIS. « Boosting Systematic Search by Weighting Constraints ». Dans *ECAI'04*, pages 146–150, 2004.

[BOUZY & HELMSTETTER 2003] B. BOUZY et B. HELMSTETTER. « Monte-Carlo Go Developments ». Dans *ACG'03*, pages 159–174, 2003.

[BOWLING *et al.* 2015] M. BOWLING, N. BURCH, M. JOHANSON, et O. TAMMELIN. « Heads-up Limit Hold'em Poker is Solved ». *Science*, 347(6218) :145–149, 2015.

[BRELAZ 1979] D. BRELAZ. « New Methods to Color Vertices of a Graph ». *Communications of the ACM*, 22(4) :251–256, 1979.

[BROWN *et al.* 1988] C. A. BROWN, L. FINKELSTEIN, et P. W. PURDOM. « Backtrack Searching in the Presence of Symmetry. ». Dans Teo MORA, éditeur, *AAECC*, volume 357 de *Lecture Notes in Computer Science*, pages 99–110. Springer, 1988.

[BROWNE *et al.* 2012] C. BROWNE, E. POWLEY, et S. TAVENER. « A survey of monte carlo tree search methods ». *Intelligence and AI*, 4(1) :1–49, 2012.

[BURCH *et al.* 2012] N. BURCH, M. LANCTOT, D. SZAFRON, et R. GIBSON. Efficient Monte Carlo Counterfactual Regret Minimization in Games with Many Player Actions. Dans F. PEREIRA, C. J. C. BURGES, L. BOTTOU, et K. Q. WEINBERGER, éditeurs, *Advances in Neural Information Processing Systems 25*, pages 1880–1888. Curran Associates, Inc., 2012.

[BURO 2002] M. BURO. « The evolution of strong othello programs ». Dans *IFIP'02 - WorkShop of IWEC'02*, pages 81–88, 2002.

[BUSCEMI & MONTANARI 2008] M. G. BUSCEMI et U. MONTANARI. « A Survey of Constraint-based Programming Paradigms ». *Computer Science Review*, 2(3) :137–141, 2008.

[BÖRNER *et al.* 2003] F. BÖRNER, A. BULATOV, P. JEAVONS, et A. KROKHIN. Quantified Constraints : Algorithms and Complexity. Dans Matthias BAAZ et JohannA. MAKOWSKY, éditeurs, *Computer Science Logic*, volume 2803, pages 58–70. Springer Berlin Heidelberg, 2003.

[C. *et al.* 2015] Lecoutre C., Likitvivatanavong C., et R. H. C. YAP. « STR3 : A path-optimal filtering algorithm for table constraints ». *Artif. Intell.*, 220 :1–27, 2015.

[CAMPBELL *et al.* 2002] Murray CAMPBELL, A.Joseph Hoane JR., et Feng hsiung HSU. « Deep Blue ». *Artificial Intelligence*, 134(1–2) :57 – 83, 2002.

[CAZENAVE & JOUANDEAU 2007] T. CAZENAVE et N. JOUANDEAU. « On the Parallelization of UCT ». Dans *CGW'07*, pages 93–101, 2007.

[CAZENAVE & JOUANDEAU 2009] T. CAZENAVE et N. JOUANDEAU. « Parallel Nested Monte-Carlo search ». Dans *IPDPS'09*, pages 1–6, 2009.

[CAZENAVE *et al.* 2016] T. CAZENAVE, A. SAFFIDINE, M. J. SCHOFIELD, et M. THIELSCHER. « Nested Monte Carlo Search for Two-Player Games ». Dans *AAAI'16*, pages 687–693, 2016.

[CAZENAVE 2006] T. CAZENAVE. « A Phantom-Go Program ». Dans *ACG'05*, pages 120–125, 2006.

[CAZENAVE 2009] T. CAZENAVE. « Nested Monte-Carlo Search ». Dans *IJCAI'09*, pages 456–461, 2009.

[CAZENAVE 2015] T. CAZENAVE. « Generalized Rapid Action Value Estimation ». Dans *IJCAI'15*, pages 754–760, 2015.

[CEREXHE *et al.* 2013] T. J. CEREXHE, O. SABUNCU, et M. THIELSCHER. « Evaluating Answer Set Clause Learning for General Game Playing ». Dans *LPNMR'13*, volume 8148 de *Lecture Notes in Computer Science*, pages 219–232. Springer, 2013.

[CERNY 2014] J. CERNY. « *Playing General Imperfect-Information Games Using Game-Theoretic Algorithms* ». PhD thesis, Czech Technical University, 2014.

[CHASLOT *et al.* 2008] G. CHASLOT, M. H. M. WINANDS, et H. J. v. d. HERIK. « Parallel Monte-Carlo Tree Search ». Dans *CG'08*, pages 60–71, 2008.

[CHASLOT 2010] G. CHASLOT. « *Monte-Carlo Tree Search* ». PhD thesis, Maastricht University, 2010.

[CHMEISS *et al.* 2003] A. CHMEISS, P. JÉGOU, et L. KEDDAR. « On a generalization of triangulated graphs for domains decomposition of CSPs ». Dans *IJCAI'03*, pages 203–208, 2003.

[CIANCARINI & FAVINI 2010] P. CIANCARINI et G. P. FAVINI. « Monte Carlo tree search in Kriegspiel ». *Artificial Intelligence*, 174(11) :670–684, 2010.

[CLARK 1978] K. L. CLARK. Negation as failure. Dans Jack MINKER, éditeur, *Logic and Data Bases*, volume 1, pages 293–322. Plenum Press, 1978.

[CLUNE 2007] J. CLUNE. « Heuristic Evaluation Functions for General Game Playing ». Dans *AAAI'07*, pages 1134–1139, 2007.

[COHEN *et al.* 2006] D. A. COHEN, P. JEAVONS, C. JEFFERSON, K. E. PETRIE, et B. M. SMITH. « Symmetry Definitions for Constraint Satisfaction Problems ». *Constraints*, 11(2-3) :115–137, 2006.

[COLLETTE *et al.* 2006] S. COLLETTE, J.-F. RASKIN, et F. SERVAIS. « On the Symbolic Computation of the Hardest Configurations of the RUSH HOUR Game ». Dans *CG'06*, pages 220–233, 2006.

[CONDON 1992] A. CONDON. « The Complexity of Stochastic Games ». *Information and Computation*, 96 :203–224, 1992.

[COOK 1971] S. A. COOK. « The Complexity of Theorem-proving Procedures ». Dans *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, 1971.

[COQUELIN & MUNOS 2007] P.-A. COQUELIN et r. MUNOS. « Bandit Algorithms for Tree Search ». Dans *UAI'07*, pages 67–74, 2007.

[COWLING *et al.* 2012] Peter I. COWLING, Edward J. POWLEY, et Daniel WHITEHOUSE. « Information Set Monte Carlo Tree Search ». *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2) :120–143, 2012.

[COX *et al.* 2009] E. COX, E. SCHKUFZA, R. MADSEN, et M. GENESERETH. « Factoring General Games using Propositional Automata ». Dans *IJCAI'09 Workshop on General Game Playing (GIGA'09)*, pages 13–20, 2009.

[CRAWFORD *et al.* 1996] J. M. CRAWFORD, M. L. GINSBERG, E. M. LUKS, et A. ROY. « Symmetry-Breaking Predicates for Search Problems ». Dans *KR'96*, pages 148–159, 1996.

[CREIGNOU *et al.* 2001] N. CREIGNOU, S. KHANNA, et M. SUDAN. *Complexity Classifications of Boolean Constraint Satisfaction Problems*. Society for Industrial and Applied Mathematics, 2001.

[DANTSIN *et al.* 2001] E. DANTSIN, T. EITER, G. GOTTLOB, et A. VORONKOV. « Complexity and expressive power of logic programming ». *ACM Computing Surveys*, 33(3) :374–425, 2001.

[DARGA *et al.* 2008] P. T. DARGA, K. A. SAKALLAH, et I. L. MARKOV. « Faster Symmetry Discovery Using Sparsity of Symmetries ». Dans *DAC'08*, pages 149–154. ACM, 2008.

[DAVIS *et al.* 1962] M. DAVIS, G. LOGEMANN, et D. LOVELAND. « A Machine Program for Theorem-proving ». *Communications of the ACM*, 5(7) :394–397, 1962.

[DE BONDT 2012] M. de BONDT. « Solving Mahjong Solitaire boards with peeking ». *Computing Research Repository*, abs/1203.6559, 2012.

[DEBRUYNE & BESSIÈRE 1997] R. DEBRUYNE et C. BESSIÈRE. « Some Practicable Filtering Techniques for the Constraint Satisfaction Problem ». Dans *IJCAI'97*, pages 412–417, 1997.

[DECHTER & FROST 2002] Rina DECHTER et Daniel FROST. « Backjump-based backtracking for constraint satisfaction problems ». *Artificial Intelligence*, 136(2) :147 – 188, 2002.

[DECHTER & MEIRI 1989] R. DECHTER et I. MEIRI. « Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems ». Dans *IJCAI'89*, pages 271–277. Morgan Kaufmann Publishers Inc., 1989.

[DECHTER & MEIRI 1994] Rina DECHTER et Itay MEIRI. « Experimental Evaluation of Preprocessing Algorithms for Constraint Satisfaction Problems ». *Artificial Intelligence*, 68(2) :211–241, 1994.

[DECHTER & PEARL 1989] R. DECHTER et J. PEARL. « Tree Clustering for Constraint Networks ». *Artificial Intelligence*, 38(3) :353–366, 1989.

[DECHTER 1990a] R. DECHTER. « Enhancement Schemes for Constraint Processing : Backjumping, Learning, and Cutset Decomposition ». *Artificial Intelligence*, 41(3) :273–312, 1990.

[DECHTER 1990b] R. DECHTER. « On the Expressiveness of Networks with Hidden Variables ». Dans *AAAI'90*, pages 556–562, 1990.

[DECHTER 2003] R. DECHTER. *Constraint Processing*. Morgan Kaufmann Publishers Inc., 2003.

[DUPACOVÁ *et al.* 2000] J. DUPACOVÁ, N. GRÖWE-KUSKA, et W. RÖMISCH. « Scenario reduction in stochastic programming : An approach using probability metrics ». Dans *Stochastic Programming E-Print Series*. Institut für Mathematik, 2000.

[EDELKAMP *et al.* 2012] S. EDELKAMP, T. FEDERHOLZNER, et P. KISSMANN. « Searching with Partial Belief States in General Games with Incomplete Information ». Dans *KI'12*, pages 25–36, 2012.

[ENZENBERGER & MÜLLER 2009] M. ENZENBERGER et M. MÜLLER. « A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm ». Dans *ACG'09*, pages 14–20, 2009.

[EPSTEIN 1994] S. L. EPSTEIN. « Identifying the Right Reasons : Learning to Filter Decision Makers ». Dans *AAAI'94*, pages 68–71, 1994.

[FAHLE *et al.* 2001] T. FAHLE, S. SCHAMBERGER, et M. SELLMANN. « Symmetry Breaking ». Dans *CP'01*, pages 93–107, 2001.

[FARGIER & LANG 1993] H. FARGIER et J. LANG. « Uncertainty in Constraint Satisfaction Problems : a Probalistic Approach ». Dans *ECSQARU'93*, pages 97–104, 1993.

[FARGIER *et al.* 1993] H. FARGIER, J. LANG, et T. SCHIES. « Selecting preferred solutions in fuzzy constraint satisfaction problems. ». Dans *EUFIT'93*, pages 1128–1134, 1993.

[FERNANDEZ & SALMERON 2008] A. FERNANDEZ et A. SALMERON. « BayesChess : A computer chess program based on Bayesian networks ». *Pattern Recognition Letters*, 2008.

[FINNSSON & BJÖRNSSON 2008] Hilmar FINNSSON et Yngvi BJÖRNSSON. « Simulation-based approach to General Game Playing ». Dans *AAAI'08*, pages 259–264. AAAI Press, 2008.

[FINNSSON & BJÖRNSSON 2011] H. FINNSSON et Y. BJÖRNSSON. « CadiaPlayer : Search-Control Techniques ». *KI - Künstliche Intelligenz*, 25(1) :9–16, 2011.

[FOCACCI & MILANO 2001] F. FOCACCI et M. MILANO. « Global Cut Framework for Removing Symmetries ». Dans *CP'01*, pages 77–92, 2001.

[FRAYN 2005] C. FRAYN. « An Evolutionary Approach to Strategies for the Game of Monopoly. ». Dans *CIG'05*. IEEE, 2005.

[FREUDER 1978] E. C. FREUDER. « Synthesizing Constraint Expressions ». *Communications of the ACM*, 21(11) :958–966, 1978.

[FREUDER 1991] Eugene C. FREUDER. « Eliminating Interchangeable Values in Constraint Satisfaction Problems ». Dans *AAAI'91*, pages 227–233, 1991.

[FROST & DECHTER 1994] D. FROST et R. DECHTER. « Dead-End Driven Learning ». Dans *AAAI'94*, pages 301–306, 1994.

[FROST & DECHTER 1995] D. FROST et R. DECHTER. « Look-ahead Value Ordering for Constraint Satisfaction Problems ». Dans *IJCAI'95*, pages 572–578. Morgan Kaufmann Publishers Inc., 1995.

[GALATTI *et al.* 2005] J. GALATTI, S. Cha nad M. GARGANO, et C. TAPPERT. « Applying Artificial Intelligence Techniques to Problems of Incomplete Information : Optimizing Bidding in the Game of Bridge ». Dans *Proceedings of Student/Faculty Research Day*, 2005.

[GEISSER *et al.* 2014] F. GEISSER, T. KELLER, et R. MATTMÜLLER. « Past, Present, and Future : An Optimal Online Algorithm for Single-Player GDL-II Games ». Dans *ECAI'14*, pages 357–362, 2014.

[GELLY & SILVER 2011] S. GELLY et D. SILVER. « Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go ». *Artificial Intelligence*, 175(11) :1856–1875, 2011.

[GELLY *et al.* 2006] S. GELLY, Y. WANG, R. MUNOS, et O. TEYTAUD. « Modification of UCT with Patterns in Monte-Carlo Go ». Research Report RR-6062, INRIA, 2006.

[GELLY *et al.* 2008] S. GELLY, J.-B. HOOCK, A. RIMMEL, O. TEYTAUD, et Y. KALEMKARIAN. « The parallelization of monte-carlo planning - parallelization of mc-planning ». Dans *ICINCO-ICSO*, pages 244–249. INSTICC Press, 2008.

[GENESERETH & BJÖRNSSON 2013] M. GENESERETH et Y. BJÖRNSSON. « The International General Game Playing Competition ». *AI Magazine*, 34(2) :107–111, 2013.

[GENESERETH & LOVE 2005] M. GENESERETH et N. LOVE. « General game playing : Overview of the AAAI competition ». *AI Magazine*, 26 :62–72, 2005.

[GENESERETH & THIELSCHER 2014] M. R. GENESERETH et M. THIELSCHER. *General Game Playing*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2014.

[GENT & SMITH 1999] I. P. GENT et B. SMITH. « Symmetry Breaking During Search in Constraint Programming ». Dans *ECAI'2000*, pages 599–603, 1999.

[GENT *et al.* 2003] Ian P. GENT, Warwick HARVEY, Tom KELSEY, et Steve LINTON. « Generic SBDD Using Computational Group Theory. ». Dans Francesca ROSSI, éditeur, *CP'03*, volume 2833 de *Lecture Notes in Computer Science*, pages 333–347. Springer, 2003.

[GHERRITY 1993] M. GHERRITY. « *A Game-learning Machine* ». PhD thesis, University of California at San Diego, 1993.

[GILL 1974] J. T. GILL. « Computational Complexity of Probabilistic Turing Machines ». Dans *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC'74, pages 91–95. ACM, 1974.

[GILPIN & SANDHOLM 2007] A. GILPIN et T. SANDHOLM. « Better automated abstraction techniques for imperfect information games, with application to Texas Hold'em poker ». Dans *AAMAS'07*, pages 1–8. ACM, 2007.

[GINSBERG & MCALLESTER 1994] Matthew L. GINSBERG et David A. MCALLESTER. « GSAT and Dynamic Backtracking ». *Artificial Intelligence*, 1 :25–46, 1994.

[GINSBERG 1993] M. L. GINSBERG. « Dynamic Backtracking ». *Computing Research Repository*, cs.AI/9308101, 1993.

[GINSBERG 2011] M. L. GINSBERG. « GIB : Imperfect Information in a Computationally Challenging Game ». *CoRR*, abs/1106.0669, 2011.

[GOLOMB & BAUMERT 1965] S. W. GOLOMB et L. D. BAUMERT. « Backtrack Programming ». *Journal of the ACM*, 12(4) :516–524, 1965.

[GRAF *et al.* 2011] T. GRAF, . LORENZ, M. PLATZNER, et L. SCHAEFERS. « Parallel Monte-Carlo Tree Search for HPC Systems ». Dans *Euro-Par'11*, volume 6853 de *LNCS*, pages 365–376. Springer, Heidelberg, 2011.

[GRIM *et al.* 2005] J. GRIM, P. SOMOL, et P. PUDIL. « Probabilistic neural network playing and learning Tic–Tac–Toe ». *Pattern Recognition Letters*, 26(12) :1866 – 1873, 2005. Artificial Neural Networks in Pattern Recognition.

[HARALICK & ELLIOTT 1979] R. M. HARALICK et G. L. ELLIOTT. « Increasing Tree Search Efficiency for Constraint Satisfaction Problems ». Dans *IJCAI'79*, pages 356–364. Morgan Kaufmann Publishers Inc., 1979.

[HNICH *et al.* 2009] B. HNICH, R. ROSSI, S. A. TARIM, et S. PRESTWICH. « *Synthesizing Filtering Algorithms for Global Chance-Constraints* », pages 439–453. Springer Berlin Heidelberg, 2009.

[HULUBEI & O'SULLIVAN 2005] T. HULUBEI et B. O'SULLIVAN. « Search Heuristics and Heavy-Tailed Behaviour ». Dans Peter BEEK, éditeur, *CP'05*, volume 3709 de *Lecture Notes in Computer Science*, pages 328–342. Springer-Verlag, 2005.

[HURWITZ & MARWALA 2007] E. HURWITZ et T. MARWALA. « Learning to bluff ». Dans *SMC'07*, pages 1188–1193, 2007.

[HWANG & MITCHELL 2005] J. HWANG et D. G. MITCHELL. « 2-Way vs. d-Way Branching for CSP ». Dans *CP'05*, pages 343–357, 2005.

[JEFFERSON *et al.* 2006] C. JEFFERSON, A. MIGUEL, I. MIGUEL, et S. A. TARIM. « Modelling and solving English Peg Solitaire ». *Computers & Operations Research*, 33(10) :2935 – 2959, 2006. Part Special Issue : Constraint Programming.

[JÉGOU 1993] P. JÉGOU. « Decomposition of Domains Based on the Micro-Structure of Finite Constraint-Satisfaction Problems ». Dans *AAAI'93*, pages 731–736, 1993.

[JOHANSON 2007] M. JOHANSON. « Robust strategies and counter-strategies : building a champion level computer poker player ». Master's thesis, University of Alberta, 2007.

[JUNGHANNS & SCHAEFFER 2001] A. JUNGHANNS et J. SCHAEFFER. « Sokoban : Enhancing general single-agent search methods using domain knowledge ». *Artificial Intelligence*, 129(1–2) :219 – 251, 2001.

[JUSSIEN *et al.* 2002] N. JUSSIEN, R. A. KASTLER, O. LHOMME, et I. SA. « Unifying Search Algorithms for CSP ». Rapport technique, École des Mines de Nantes, 2002.

[KAISER 2005] D. M. KAISER. « The Structure of Games ». Dans *Proceedings of the 43rd Annual Southeast Regional Conference*, ACM-SE 43, pages 61–62. ACM, 2005.

[KAISER 2007] D. M. KAISER. « Automatic feature extraction for autonomous general game playing agents ». Dans *AAMAS'07*, pages 1–7. ACM, 2007.

[KATO & TAKEUCHI 2010] H. KATO et I. TAKEUCHI. « Parallel Monte-Carlo Tree Search with Simulation Servers ». *2013 Conference on Technologies and Applications of Artificial Intelligence*, 0 :491–498, 2010.

[KIRCI *et al.* 2011] M. KIRCI, N. R. STURTEVANT, et J. SCHAEFFER. « A GGP Feature Learning Algorithm. ». *KI - Künstliche Intelligenz*, 25(1) :35–42, 2011.

[KISSMANN & EDELKAMP 2010] P. KISSMANN et S. EDELKAMP. « Instantiating General Games Using Prolog or Dependency Graphs ». Dans *KI'10*, pages 255–262. Springer-Verlag, 2010.

[KOCSIS & SZEPESVÁRI 2006] L. KOCSIS et C. SZEPESVÁRI. « Bandit Based Monte-carlo Planning ». Dans *ECML'06*, pages 282–293. Springer-Verlag, 2006.

[KORF 1991] R. E. KORF. « Multi-player alpha-beta pruning ». *Artificial Intelligence*, 48(1) :99 – 111, 1991.

[KORF 1997] R. E. KORF. « Finding Optimal Solutions to Rubik's Cube Using Pattern Databases ». Dans *AAAI'97*, pages 700–705, 1997.

[KORICHE *et al.* 2014] F. KORICHE, S. LAGRUE, E. PIETTE, et S. TABARY. « Traduction de jeux à information incertaine en réseaux de contraintes stochastiques ». Dans *JFPC'14*, pages 59–68, 2014.

[KORICHE *et al.* 2015a] F. KORICHE, S. LAGRUE, E. PIETTE, et S. TABARY. « Compiling Strategic Games with Complete Information into Stochastic CSPs ». Dans *AAAAI'15 workshop : Planning, Search, and Optimization*, 2015.

[KORICHE *et al.* 2015b] F. KORICHE, S. LAGRUE, É. PIETTE, et S. TABARY. « Résolution de SCSP avec borne de confiance pour les jeux de stratégie ». Dans *JFPC'15*, pages 160–169, 2015.

[KORICHE *et al.* 2016a] F. KORICHE, S. LAGRUE, E. PIETTE, et S. TABARY. « Programmation par contraintes stochastiques pour le General Game Playing avec informations incomplètes ». Dans *JFPC'16*, pages 7–16, 2016.

[KORICHE *et al.* 2016b] F. KORICHE, S. LAGRUE, E. PIETTE, et S. TABARY. « Stochastic Constraint Programming for General Game Playing with Imperfect Information ». Dans *IJCAI'16 WorkShop on General Game Playing (GIGA'16)*, 2016.

[KORICHE *et al.* 2016c] F. KORICHE, S. LAGRUE, É. PIETTE, et Sé. TABARY. « General game playing with stochastic CSP ». *Constraints*, 21(1) :95–114, 2016.

[KRZYSZTOF 2003] A. KRZYSZTOF. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[KUHLMANN & STONE 2007] G. KUHLMANN et P. STONE. « Graph-Based Domain Mapping for Transfer Learning in General Games ». Dans *ECML'07*, pages 188–200, 2007.

[KUHLMANN *et al.* 2006] G. KUHLMANN, K. DRESNER, et P. STONE. « Automatic Heuristic Construction in a Complete General Game Player ». Dans *AAAI'06*, pages 1457–62, 2006.

[L. *et al.* 1998] Michael L. L., Judy G., et Martin M.. « The Computational Complexity of Probabilistic Planning ». *Journal of Artificial Intelligence Research*, 9 :1–36, 1998.

[LANCTOT *et al.* 2009] Marc LANCTOT, K. WAUGH, M. ZINKEVICH, et M. BOWLING. Monte Carlo Sampling for Regret Minimization in Extensive Games. Dans Y. BENGIO, D. SCHUURMANS, J. D. LAFFERTY, C. K. I. WILLIAMS, et A. CULOTTA, éditeurs, *Advances in Neural Information Processing Systems 22*, pages 1078–1086. Curran Associates, Inc., 2009.

[LANCTOT *et al.* 2013] M. LANCTOT, A. SAFFIDINE, J. VENESS, C. ARCHIBALD, et M. H. M. WINANDS. « Monte Carlo *-Minimax Search ». *CoRR*, 2013.

[LARROSA & SCHIEX 2004] J. LARROSA et T. SCHIEX. « Solving weighted {CSP} by maintaining arc consistency ». *Artificial Intelligence*, 159(1–2) :1 – 26, 2004.

[LARROSA *et al.* 1999] J. LARROSA, P. MESEGUER, et T. SCHIEX. « Maintaining reversible {DAC} for Max-CSP ». *Artificial Intelligence*, 107(1) :149 – 163, 1999.

[LECOUTRE & CARDON 2005] C. LECOUTRE et S. CARDON. « A greedy approach to establish singleton arc consistency ». Dans *IJCAI'05*, pages 199–204, 2005.

[LECOUTRE & HEMERY 2007] C. LECOUTRE et F. HEMERY. « A Study of Residual Supports in Arc Consistency ». Dans *IJCAI'07*, pages 125–130, 2007.

[LECOUTRE & SZYMANEK 2006] C. LECOUTRE et R. SZYMANEK. « Generalized Arc Consistency for Positive Table Constraints ». Dans *CP'06*, pages 284–298, 2006.

[LECOUTRE *et al.* 2003] C. LECOUTRE, F. BOUSSEMART, et F. HEMERY. « Exploiting Multidirectionality in Coarse-Grained Arc Consistency Algorithms ». Dans *CP'03*, pages 480–494, 2003.

[LECOUTRE *et al.* 2004] C. LECOUTRE, F. BOUSSEMART, et F. HEMERY. « Backjump-Based Techniques versus Conflict-Directed Heuristics ». Dans *ICTAI'04*, pages 549–557, 2004.

[LECOUTRE *et al.* 2007] C. LECOUTRE, L. SAIS, S. TABARY, et V. VIDAL. « Nogood Recording from Restarts. ». Dans Manuela M. VELOSO, éditeur, *IJCAI'07*, pages 131–136, 2007.

[LECOUTRE *et al.* 2009] C. LECOUTRE, L. SAÏS, S. TABARY, et V. VIDAL. « Reasoning from last conflict(s) in constraint programming ». *Artificial Intelligence*, 173(18) :1592 – 1614, 2009.

[LECOUTRE 2009] C. LECOUTRE. *Constraint networks : techniques and algorithms*. ISTE/John Wiley, 2009.

[LECOUTRE 2011] C. LECOUTRE. « STR2 : optimized simple tabular reduction for table constraints ». *Constraints*, 16(4) :341–371, 2011.

[LEFLER & MALLETT 2002] M. LEFLER et J. MALLETT. « Zillions of games. Commercial website ». `http://www.zillions-of-games.com/index.html`, 2002.

[LEVINSON 1994] R. LEVINSON. « MORPH II : A UNIVERSAL AGENT - PROGRESS REPORT AND PROPOSAL ». Rapport technique, University of California at Santa Cruz, 1994.

[LIFSCHITZ & YANG 2011] V. LIFSCHITZ et F. YANG. Eliminating function symbols from a nonmonotonic causal theory. Dans *Knowing, Reasoning, and Acting : Essays in Honour of Hector J. Levesque*. College Publicaions, 2011.

[LISY *et al.* 2013] V. LISY, V. KOVARIK, M. LANCTOT, et B. BOSANSKY. Convergence of Monte Carlo Tree Search in Simultaneous Move Games. Dans C. J. C. BURGES, L. BOTTOU, M. WELLING, Z. GHAHRAMANI, et K. Q. WEINBERGER, éditeurs, *NIPS'13*, pages 2112–2120. Curran Associates, Inc., 2013.

[LITTMAN *et al.* 2001] M. L. LITTMAN, S. M. MAJERCIK, et T. PITASSI. « Stochastic Boolean Satisfiability ». *Journal of Automated Reasoning*, 27(3) :251–296, 2001.

[LONG *et al.* 2010] J. R. LONG, N. R. STURTEVANT, M. BURO, et T. FURTAK. « Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search ». Dans *AAAI'10*, 2010.

[LOVE *et al.* 2006] N. LOVE, M. GENESERETH, et T. HINRICHS. « General Game Playing : Game Description Language Specification

». Rapport technique LG-2006-01, Stanford University, 2006.

[MACKWORTH 1977] A. K. MACKWORTH. « Consistency in Networks of Relations ». *Artificial Intelligence*, 8(1) :99–118, 1977.

[MCKAY & PIPERNO 2014] B. D. MCKAY et A. PIPERNO. « Practical graph isomorphism, {II} ». *Journal of Symbolic Computation*, 60(0) :94 – 112, 2014.

[MEHAT & CAZENAVE 2008] J. MEHAT et T. CAZENAVE. « An Account of a Participation to the 2007 General Game Playing Competition ». `http://www.ai.univ-paris8.fr/~jm/ggp/ggp2008-2.pdf`, 2008. unpublished.

[MÉHAT & CAZENAVE 2010] J. MÉHAT et T. CAZENAVE. « Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing ». *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4) :271–277, 2010.

[MESEGUER & TORRAS 2001] P. MESEGUER et C. TORRAS. « Exploiting symmetries within constraint satisfaction search ». *Artificial Intelligence*, 129(1-2) :133–163, 2001.

[MINTON *et al.* 1992] s. MINTON, M. D. JOHNSTON, A. B. PHILIPS, et P. LAIRD. « Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems ». *Artificial Intelligence*, 58(1) :161 – 205, 1992.

[MOHR & HENDERSON 1986] R. MOHR et T. C. HENDERSON. « Arc and Path Consistency Revisited. ». *Artificial Intelligence*, 28(2) :225–233, 1986.

[MOTAL 2011] T. MOTAL. « General Game Playing in imperfect Information Games ». Master's thesis, Czech Technical University, Prague, 2011.

[MYERSON 1991] R. B. MYERSON. *Game theory : analysis of conflict*. Harvard university press, 1997, 1991.

[MÉHAT & CAZENAVE 2011a] J. MÉHAT et T. CAZENAVE. « A Parallel General Game Player. ». *KI - Künstliche Intelligenz*, 25(1) :43–47, 2011.

[MÉHAT & CAZENAVE 2011b] Jean MÉHAT et Tristan CAZENAVE. « Tree Parallelization of Ary on a Cluster ». Dans *IJCAI'11 Workshop on General Game Playing (GIGA'11)*, pages 39–43, 2011.

[NADEL 1990] B. A. NADEL. « Some applications of the constraint satisfaction problem ». Dans *Workshop on Constraint Directed Reasoning Working Notes of AAAI'90*, 1990.

[NASH 1951] J.F. NASH. « Non-cooperative Games ». *Annals of Mathematics*, 54(2) :286–295, 1951.

[NGUYEN & LALLOUET 2014] T. NGUYEN et A. LALLOUET. « A Complete Solver for Constraint Games ». Dans *CP'14*, pages 58–74, 2014.

[PAPADIMITRIOU 1994] C. H. PAPADIMITRIOU. *Computational complexity*. Addison-Wesley, 1994.

[PARKER *et al.* 2010] A. PARKER, D. NAU, et V.S. SUBRAHMANIAN. « *Paranoia versus overconfidence in imperfect-information games.* », pages 63–87. London : College Publications, 2010.

[PAWLEWICZ 2011] J. PAWLEWICZ. « *Nearly Optimal Computer Play in Multiplayer Yahtzee* », pages 250–262. Springer Berlin Heidelberg, 2011.

[PELL 1994] B. PELL. « A Strategic Metagame Player for General Chess-Like Games ». Dans *Computational Intelligence*, volume 12, pages 1378–1385, 1994.

[PEREZ & MARWALA 2008] M. PEREZ et T. MARWALA. « Stochastic Optimization Approaches for Solving Sudoku ». *Computing Research Repository*, abs/0805.0697, 2008.

[PITRAT 1968] J. PITRAT. « Realization of a general game-playing program ». Dans *IFIP Congress (2)*, pages 1570–1574, 1968.

[PROSSER 1993] P. PROSSER. « Hybrid Algorithms for the Constraint Satisfaction Problem ». *Computational Intelligence*, pages 268–299, 1993.

[PUGET 1993] J.-F. PUGET. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. Dans J. KOMOROWSKI et Z. W. RAS, éditeurs, *ISMIS'93*, pages 350–361. Springer, Berlin, Heidelberg, 1993.

[PUGET 1998] J.-F. PUGET. « A Fast Algorithm for the Bound Consistency of alldiff Constraints. ». Dans *AAAI'98*, pages 359–366, 1998.

[RAJARATNAM & THIELSCHER 2015] D. RAJARATNAM et M. THIELSCHER. « Execution Monitoring as Meta-Games for General Game-Playing Robots ». Dans *IJCAI'15*, pages 3178–3185, 2015.

[REFALO 2004] P. REFALO. « Impact-Based Search Strategies for Constraint Programming ». Dans *CP'04*, pages 557–571, 2004.

[RÉGIN 1994] J. C. RÉGIN. « A Filtering Algorithm for Constraints of Difference in CSPs ». Dans *AAAI'94*, volume 94, pages 362–367, 1994.

[ROMEIN & BAL 2003] J. W. ROMEIN et H. E. BAL. « Solving Awari with Parallel Retrograde Analysis ». *Computer*, 36(10) :26–33, 2003.

[ROMEIN 2001] J. W. ROMEIN. « *Multigame - An environment for Distributed Game-Tree Search* ». PhD thesis, Vrije Universiteit Amsterdam, 2001.

[ROSSI *et al.* 2006] F. ROSSI, P. v. BEEK, et T. WALSH. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.

[ROY & PACHET 1998] P. ROY et F. PACHET. « Using Symmetry of Global Constraints to Speed up the Resolution of Constraint Sa-

tisfaction Problems ». Dans *in Workshop on Non Binary Constraints, ECAI'98*, pages 27–33, 1998.

[SABIN & FREUDER 1994] D. SABIN et E. C. FREUDER. « Contradicting Conventional Wisdom in Constraint Satisfaction ». Dans *ECAI'94*, pages 125–129, 1994.

[SAFFIDINE 2014] A. SAFFIDINE. « The Game Description Language Is Turing Complete ». *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4) :320–324, 2014.

[SCHAEFFER *et al.* 2005] J. SCHAEFFER, Y. BJÖRNSSON, N. BURCH, A. KISHIMOTO, M. MÜLLER, R. LAKE, P. LU, et S. SUTPHEN. « Solving Checkers. ». Dans Leslie Pack KAELBLING et Alessandro SAFFIOTTI, éditeurs, *IJCAI'05*, pages 292–297. Professional Book Center, 2005.

[SCHAEFFER 2008] J. SCHAEFFER. *One Jump Ahead : Computer Perfection at Checkers*. Springer Publishing Company, Incorporated, 2nd édition, 2008.

[SCHAUL 2013] T. SCHAUL. « A video game description language for model-based or interactive learning ». Dans *CIG'13*, pages 1–8, 2013.

[SCHIEX 1992] T. SCHIEX. « Possibilistic Constraint Satisfaction Problems or "How to Handle Soft Constraints ?" ». Dans Didier DUBOIS et Michael P. WELLMAN, éditeurs, *UAI'92*, pages 268–275. Morgan Kaufmann, 1992.

[SCHIFFEL & THIELSCHER 2007a] S. SCHIFFEL et M. THIELSCHER. « Automatic construction of a heuristic search function for general game playing ». Dans *IJCAI'07*, 2007.

[SCHIFFEL & THIELSCHER 2007b] S. SCHIFFEL et M. THIELSCHER. « Fluxplayer : A successful general game player ». Dans *AAAI'07*, pages 1191–1196. AAAI Press, 2007.

[SCHIFFEL & THIELSCHER 2010] S. SCHIFFEL et M. THIELSCHER. « *A Multiagent Semantics for the Game Description Language* », pages 44–55. Springer Berlin Heidelberg, 2010.

[SCHIFFEL & THIELSCHER 2011] S. SCHIFFEL et M. THIELSCHER. « Reasoning About General Games Described in GDL-II. ». Dans *AAAI'11*, pages 846–851. AAAI Press, 2011.

[SCHIFFEL 2010] S. SCHIFFEL. « Symmetry Detection in General Game Playing ». Dans *AAAI'10*. AAAI Press, 2010.

[SCHKUFZA *et al.* 2008] E. SCHKUFZA, N. LOVE, et M. GENESERETH. « *Propositional Automata and Cell Automata : Representational Frameworks for Discrete Dynamic Systems* », pages 56–66. Springer Berlin Heidelberg, 2008.

[SCHNEIDER & GARCIA ROSA 2002] M. O. SCHNEIDER et J. L. GARCIA ROSA. « Neural Connect 4 - a connectionist approach to the game ». Dans *Neural Networks'02*, pages 236–241, 2002.

[SCHOFIELD & THIELSCHER 2015] M. J. SCHOFIELD et M. THIELSCHER. « Lifting Model Sampling for General Game Playing to Incomplete-Information Models ». Dans *AAAI'15*, pages 3585–3591, 2015.

[SCHOFIELD *et al.* 2012] M. J. SCHOFIELD, T. J. CEREXHE, et M. THIELSCHER. « HyperPlay : A Solution to General Game Playing with Imperfect Information ». Dans *AAAI'12*, 2012.

[SCHREIBER 2014] S. SCHREIBER. « GGP.org - Tiltyard Gaming Server ». `http://tiltyard.ggp.org/`, 2014. CS227b class at Stanford University.

[SCHÄFER 2008] J. SCHÄFER. « *The UCT Algorithm Applied to Games with Imperfect Information* ». PhD thesis, Otto-von-Guericke-Universitat, 2008.

[SHAFIEI *et al.* 2009] M. SHAFIEI, N. STURTEVANT, et J. SCHAEFFER. « Comparing UCT versus CFR in Simultaneous Games ». Dans *IJCAI Workshop on General Game Playing (GIGA'09)*, 2009.

[SHAPIRO *et al.* 2014] A. SHAPIRO, D. DENTCHEVA, et A. RUSZCZYNSKI. *Lectures on Stochastic Programming : Modeling and Theory, Second Edition*. SIAM, 2014.

[SHEPPARD 2002] B. SHEPPARD. « World-championship-caliber Scrabble ». *Artificial Intelligence*, 134(1–2) :241 – 275, 2002.

[SHOHAM & LEYTON-BROWN 2009] Y. SHOHAM et K. LEYTON-BROWN. *Multiagent Systems : Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

[SILVER *et al.* 2016] D. SILVER, A. HUANG, C. J. MADDISON, A. GUEZ, L. SIFRE, D. v. d. DRIESSCHE, J. SCHRITTWIESER, I. ANTONOGLOU, V. PANNEERSHELVAM, M. LANCTOT, S. DIELEMAN, D. GREWE, J. NHAM, N. KALCHBRENNER, I. SUTSKEVER, T. LILLICRAP, M. LEACH, K. KAVUKCUOGLU, T. GRAEPEL, et D. HASSABIS. « Mastering the game of Go with deep neural networks and tree search ». *Nature*, 529 :484–503, 2016.

[SMITH 1996] B. M. SMITH. « Succeed-first or Fail-first : A Case Study in Variable and Value Ordering », 1996.

[SOEJIMA *et al.* 2010] Y. SOEJIMA, A. KISHIMOTO, et O. WATANABE. « Evaluating Root Parallelization in Go ». *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4) :278–287, 2010.

[STOCKMEYER 1976] L. J. STOCKMEYER. « The Polynomial-Time Hierarchy ». *Theoretical Computer Science*, 3(1) :1–22, 1976.

[STURTEVANT 2002] N. STURTEVANT. « A Comparison of Algorithms for Multi-Player Games ». Dans *CG'02*, pages 108–122, 2002.

[STURTEVANT 2008] N. R. STURTEVANT. « An Analysis of UCT in Multi-Player Games ». *International Computer Games Association Journal*, 31(4) :195–208, 2008.

[SUTTON 1988] R. S. SUTTON. « Learning to Predict by the Methods of Temporal Differences ». *Machine Learning*, 3(1) :9–44, 1988.

[TARIM *et al.* 2006] S. A. TARIM, S. MANANDHAR, et T. WALSH. « Stochastic Constraint Programming : A Scenario-Based Approach ». *Constraints*, 11(1) :53–80, 2006.

[THIELSCHER 2010] M. THIELSCHER. « A General Game Description Language for Incomplete Information Games ». Dans *AAAI'10*, 2010.

[THIELSCHER 2011a] M. THIELSCHER. « The General Game Playing Description Language is Universal ». Dans *IJCAI'11*, pages 1107–1112. AAAI Press, 2011.

[THIELSCHER 2011b] Michael THIELSCHER. « GDL-II ». *KI - Künstliche Intelligenz*, 25(1) :63–66, 2011.

[THIELSCHER 2011c] Michael THIELSCHER. « General Game Playing in AI Research and Education ». Dans *KI'11*, volume 7006 de *LNAI*, pages 26–37. Springer, 2011.

[THIELSCHER 2015] M. THIELSCHER. « Simulation of Action Theories and an Application to General Game-Playing Robots ». Dans *Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation*, pages 33–46, 2015.

[THIELSCHER 2016] M. THIELSCHER. « GDL-III : A Proposal to Extend the Game Description Language to General Epistemic Games ». Dans *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 1630–1631. IOS Press, 2016.

[TURING *et al.* 1991] A. M. TURING, J.Y. GIRARD, J. BASCH, et P. BLANCHARD. *La machine de Turing*. Points. Edition du Seuil, collection source du savoir, 1991.

[TURING 1936] A. M. TURING. « On Computable Numbers, with an Application to the Entscheidungsproblem ». *Proceedings of the London Mathematical Society*, 2(42) :230–265, 1936.

[ULLMANN 2007] J. R. ULLMANN. « Partition Search for Non-binary Constraint Satisfaction ». *Information Sciences*, 177(18) :3639–3678, 2007.

[VAN DONGEN 2002] Marc R. C. van DONGEN. « AC-3d an Efficient Arc-Consistency Algorithm with a Low Space-Complexity. ». Dans Pascal Van HENTENRYCK, éditeur, *CP'02*, volume 2470 de *Lecture Notes in Computer Science*, pages 755–760. Springer, 2002.

[VAN ECK & VAN WEZEL 2008] N. J. van ECK et M. C. van WEZEL. « Application of reinforcement learning to the game of Othello. ». *Computers & Operations Research*, 35(6) :1999–2017, 2008.

[VAN HENTENRYCK *et al.* 1999] P. van HENTENRYCK, L. MICHEL, L. PERRON, et J.-C. RÉGIN. « Constraint Programming in OPL ». Dans *PPDP'99*, pages 98–116, 1999.

[VERFAILLIE & JUSSIEN 2005] G. VERFAILLIE et N. JUSSIEN. « Constraint Solving in Uncertain and Dynamic Environments : A Survey ». *Constraints*, 10(3) :253–281, 2005.

[VIGODA 1999] E. VIGODA. « Sampling from Gibbs Distributions ». Rapport technique, University of California at Berkeley, 1999.

[VITTAUT & MÉHAT 2014] J.-N. VITTAUT et J. MÉHAT. « Fast Instantiation of GGP Game Descriptions Using Prolog with Tabling ». Dans *ECAI'14*, pages 1121–1122, 2014.

[VON NEUMANN & MORGENSTERN 1944] J. VON NEUMANN et O. MORGENSTERN. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[VON NEUMANN 1928] J. von NEUMANN. « Zur Theorie der Gesellschaftsspiele ». *Mathematische Annalen*, 100(1) :295–320, 1928.

[WALSH 2009] T. WALSH. « Stochastic Constraint Programming ». *Computing Research Repository*, abs/0903.1152, 2009.

[YOSHIZOE *et al.* 2011] K. YOSHIZOE, A. KISHIMOTO, T. KANEKON, H. YOSHIMOTO, et Y. ISHIKAWA. « Scalable Distributed Monte-Carlo Tree Search ». Dans *SOCS'11*, 2011.

[ZHAO *et al.* 2009] D. ZHAO, D. SCHIFFEL, et M. THIELSCHER. « Decomposition of Multi-player Games ». Dans *AI'09*, pages 475–484, 2009.

[ZINKEVICH *et al.* 2008] M. ZINKEVICH, M. JOHANSON, M. BOWLING, et C. PICCIONE. « Regret Minimization in Games with Incomplete Information ». Dans *NIPS'08*, pages 905–912, 2008. A longer version is available as a University of Alberta Technical Report, TR07-14.

[ZOBRIST 1990] A. ZOBRIST. « A new hashing method with application for game playing ». *International Computer Games Association Journal*, 13(2) :69–73, 1990.

Résumé

Développer un programme capable de jouer à n'importe quel jeu de stratégie, souvent désigné par le *General Game Playing* (GGP) constitue un des Graal de l'intelligence artificielle. Les compétitions GGP, où chaque jeu est représenté par un ensemble de règles logiques au travers du *Game Description Language* (GDL), ont conduit la recherche à confronter de nombreuses approches incluant les méthodes de type Monte Carlo, la construction automatique de fonctions d'évaluation, ou la programmation logique et ASP. De par cette thèse, nous proposons une nouvelle approche dirigée par les contraintes stochastiques.

Dans un premier temps, nous nous concentrons sur l'élaboration d'une traduction de GDL en réseaux de contraintes stochastiques (SCSP) dans le but de fournir une représentation dense des jeux de stratégies et permettre la modélisation de stratégies.

Par la suite, nous exploitons un fragment de SCSP au travers d'un algorithme dénommé MAC−UCB combinant l'algorithme MAC (*Maintaining Arc Consistency*) utilisé pour résoudre chaque niveau du SCSP tour après tour, et à l'aide de UCB (*Upper Confidence Bound*) afin d'estimer l'utilité de chaque stratégie obtenue par le dernier niveau de chaque séquence. L'efficacité de cette nouvelle technique sur les autres approches GGP est confirmée par WoodStock, implémentant MAC−UCB, le leader actuel du tournoi continu de GGP.

Finalement, dans une dernière partie, nous proposons une approche alternative à la détection de symétries dans les jeux stochastiques, inspirée de la programmation par contraintes. Nous montrons expérimentalement que cette approche couplée à MAC−UCB, surpasse les meilleures approches du domaine et a permis à WoodStock de devenir champion GGP 2016.

Mots-clés: General Game Playing, CSP stochastique, Upper Confidence Bounds

Abstract

The ability for a computer program to effectively play any strategic game, often referred to General Game Playing (GGP), is a key challenge in AI. The GGP competitions, where any game is represented according to a set of logical rules in the Game Description Language (GDL), have led researches to compare various approaches, including Monte Carlo methods, automatic constructions of evaluation functions, logic programming, and answer set programming through some general game players. In this thesis, we offer a new approach driven by stochastic constraints.

We first focus on a translation process from GDL to stochastic constraint networks (SCSP) in order to provide compact representations of strategic games and to model strategies.

In a second part, we exploit a fragment of SCSP through an algorithm called MAC−UCB by coupling the MAC (Maintaining Arc Consistency) algorithm, used to solve each stage of the SCSP in turn, together with the UCB (Upper Confidence Bound) policy for approximating the values of those strategies obtained by the last stage in the sequence. The efficiency of this technical on the others GGP approaches is confirmed by WoodStock, implementing MAC−UCB, the actual leader on the GGP Continuous Tournament.

Finally, in the last part, we propose an alternative approach to symmetry detection in stochastic games, inspired from constraint programming techniques. We demonstrate experimentally that MAC−UCB, coupled with our constraint-based symmetry detection approach, significantly outperforms the best approaches and made WoodStock the GGP champion 2016.

Keywords: General Game Playing, Stochastic CSP, Upper Confidence Bounds