THÈSE de DOCTORAT

de l'UNIVERSITÉ PARIS 6

Spécialité :

INFORMATIQUE

présentée par

Tristan CAZENAVE

Pour obtenir le grade de DOCTEUR de L'UNIVERSITÉ PARIS 6

Sujet de la thèse :

Système d'Apprentissage par Auto-Observation.
Application au jeu de Go

soutenue le Vendredi 13 Décembre 1996

devant le jury composé de :

Monsieur Paul BOURGINE	Rapporteur
Monsieur Guy GOUARDERES	Examinateur
Monsieur Jean-Yves JAFFRAY	Examinateur
Monsieur Jean MICHEL	Examinateur
Monsieur Jacques PITRAT	Directeur
Monsieur Bernard VICTORRI	Rapporteur

"Prenez les arts que sont le tir-à-l'arc, la conduite de chars, la cithare, et le Go : pour aucun de ceux-ci il n'est possible de s'arrêter d'apprendre"

Yin Xi, in "Guan Yin Zi" [Le Livre du Maître Yin],
4ème siècle avant J.-C.

Table des Matières

INTRODUCTION 7

L'intérêt d'un système d'apprentissage par auto-observation 7

L'intérêt du jeu de Go 7
Les jeux de stratégie 8
Le jeu de Go 8
L'apprentissage du jeu de Go 9

Le fonctionnement d'un système d'apprentissage par auto-observation 10
Représentation des connaissances 10
Apprentissage de nouvelles connaissances 10
Utilisation des connaissances apprises 11
Architecture générale du système d'apprentissage 11

Modes de lecture de la thèse 12

1 UN EXEMPLE DE FONCTIONNEMENT D'INTROSPECT 13

1.1 Se rendre compte de son ignorance 13
1.1.1 Résoudre un problème 13
1.1.2 Repérer les surprises 14

1.2 Créer les connaissances qui manquent en s'auto-observant 14
1.2.1 Résoudre un problème en gardant une trace 14
1.2.2 Utiliser la trace pour expliquer les faits intéressants 15
1.2.3 Généraliser 15

1.3 Utiliser les connaissances créées 15
1.3.1 Compiler les règles 15
1.3.2 Intégrer les connaissances dans un programme 16

1.4 Architecture Détaillée 16

2 LA THÉORIE COMBINATOIRE DES JEUX 18

2.1 État de l'art 18
2.1.1 Introduction à la théorie combinatoire des jeux 18
2.1.2 Théorie combinatoire des jeux et jeu de Go 20

2.2 Extension de la théorie combinatoire des jeux à la représentation de l'inconnu 22
2.2.1 Définition des jeux combinatoires comportant des valeurs inconnues 22
2.2.2 Une taxonomie des jeux combinatoires à valeurs inconnues 23

2.3 Gestion du risque 24
2.3.1 Evaluation des jeux combinatoires 24
2.3.2 Trois stratégies de gestion du risque 25
2.3.3 Utilisation de la taxonomie 26
2.3.4 Exemple d'utilisation 26

2.4 Conclusion 28

2.5 Bibliographie 28

3 REPRÉSENTATION DES CONNAISSANCES 30

3.1 Patterns géométriques 31
3.1.1 Représentation des patterns géométriques 31
3.1.2 Les concepts utilisés 31
3.1.3 Les règles 32
3.1.4 Limites des représentations à base de patterns géométriques 33

3.2 Logique des prédicats 36

3.3 Représentation des métaconnaissances 39

3.4 Une structure des connaissances dans le domaine des jeux 44
3.4.1 Trois niveaux de connaissance 44
3.4.2 Le passage entre deux niveaux 44

3.5 Les méthodes de sélection 45
3.5.1 Une définition des méthodes de sélection 45
3.5.2 Application dans le domaine des jeux 49

3.6 Conclusion 51

3.7 Bibliographie 52

4 LES EXPLICATIONS 53

4.1 Les explications tactiques 54
4.1.1 Limites des représentations existantes 54
4.1.2 Une solution : représenter le temps 57
4.1.3 La résolution de problèmes 59
4.1.4 Choix des faits à expliquer 60
4.1.5 Mécanisme de retour dans la trace 64
4.1.6 Influence de la théorie du domaine sur la généralité des explications 65

4.2 Les explications stratégiques 67
4.2.1 Les explications positives et négatives 67
4.2.2 Définition des explications à partir des méthodes de sélection 67
4.2.3 Applications dans le domaine des jeux 72

4.3 Conclusion 73

4.4 Bibliographie 74

5 L'APPRENTISSAGE ET LA GÉNÉRALISATION 76

5.1 Les systèmes qui apprennent à jouer 76
5.1.1 Les Checkers 76
5.1.2 Le Poker 78
5.1.3 Le Backgammon 78
5.1.4 Les Échecs 79
5.1.5 Les Wargames 80
5.1.6 Le Go 81

5.2 Apprentissage de patterns géométriques 84
5.2.1 Engendrer des patterns géométriques 84
5.2.2 Calculer l'état d'un jeu 85

5.2.3 Généralisation 85
5.2.4 Propriétés des règles 86
5.2.5 Oubli sélectif 87
5.2.6 Validité des calculs locaux dans un contexte global 88
5.2.7 Limites de la représentation à base de patterns géométriques 89

5.3 Apprentissage en logique des prédicats 89
5.3.1 Définitions des buts à apprendre 90
5.3.2 Résolution de problèmes et Explication 92
5.3.3 Détecter les coups forcés 92
5.3.4 Généralisation 93
5.3.5 Opérations sur les règles créées 95
5.3.6 Insertion dans la base 96
5.3.7 Bases d'exemples 96
5.3.8 Le problème de l'utilité 97

5.4 Résultats 97

5.5 Bibliographie 98

6 LA COMPILATION 101

6.1 L'ordonnancement des liste de conditions 102
6.1.1 Le mécanisme de filtrage 102
6.1.2 Ordonner les prémisses 102

6.2 Les coupes dans le graphe d'unification 102
6.2.1 Le métaprédicat absent 102

6.3 L'ordonnancement des listes de règles 106

6.4 L'Optimisation 106
6.4.1 Optimisation des calculs 106
6.4.2 Evaluation Partielle 107

6.5 La compilation en C++ 108
6.5.1 Equivalence instanciation/boucle for 108
6.5.2 Equivalence instanciation/affectation 108
6.5.3 Equivalence test/if 110
6.5.4 Equivalence absent/parcours d'arbre 111
6.5.5 Traduction d'une règle 111
6.5.6 Traduction d'une base de règles 111

6.6 Conclusion 111

6.7 Bibliographie 112

7 GOGOL : UN PROGRAMME DE GO 113

7.1 Les programmes de Go 113
7.1.1 Les programmes qui jouent une partie entière 113
7.1.2 Les programmes spécialisés sur des sous-problèmes du Go 116

7.2 Architecture de Gogol 116

7.3 Les sous-buts intéressants du Go 117

7.4 Construction des groupes **120**

7.5 Connaissances stratégiques **122**
7.5.1 La connexion au vide, une approximation du territoire 122
7.5.2 Les attaques et les défenses sur les groupes 124

7.6 Résultats **126**
7.6.1 L'échelle de niveau des programmes 126
7.6.2 Résumé des résultats des compétitions mondiales de programmes de Go 128
7.6.3 La coupe FOST 1996 129
7.6.4 Evaluation du niveau de Gogol 131

7.7 Conclusion **131**

7.7 Bibliographie **132**

8 UN PROGRAMME D'ABALONE **133**

8.1 Le jeu d'Abalone **133**

8.2 Représentation des connaissances **134**

8.3 Définition des sous-buts **135**

8.4 Règles stratégiques **135**

8.5 Conclusion **135**

9 UN PROGRAMME DE GESTION **136**

9.1 Introduction **136**

9.2 Représentation des Connaissances **136**

9.3 Résolution de Problèmes **137**

9.4 Explication **138**

9.5 Généralisation **138**

9.6 Compilation **139**

9.7 Le système d'apprentissage de la gestion **140**

9.8 Conclusion **141**

9.9 Bibliographie **141**

CONCLUSION **143**

BIBLIOGRAPHIE **145**

ANNEXES **151**

Introduction

L'intérêt d'un système d'apprentissage par auto-observation

Je m'intéresse à la prise de décision dans les domaines complexes. Les domaines complexes abordés dans cette thèse sont ceux dans lesquels un agent cherche à atteindre certains buts qui sont clairement définis. Il connaît les lois d'évolution du domaine et peut théoriquement calculer les conséquences de ses choix sur l'achèvement des buts. Cet agent est toutefois limité dans ses prévisions par ses capacités de calcul. Il est limité par le grand nombre de choix possibles à chaque étape du raisonnement et par le grand nombre d'étapes nécessaires à franchir pour prévoir les conséquences de ses choix sur l'achèvement d'un but.

Si cet agent applique une stratégie de recherche par force brute en simulant directement toutes les décisions possibles, il se retrouve face à une explosion combinatoire qui l'empêche de prévoir les conséquences de ses décisions à long terme.

Si par contre il applique une stratégie de recherche basée sur des connaissances du domaine, il peut alors sélectionner parmi toutes les décisions possibles celles qui ont une chance d'atteindre le but ou celles qui sont forcées pour atteindre le but. Il peut alors réduire énormément la largeur de l'arbre de recherche et augmenter sa profondeur. Il peut donc prévoir les conséquences de ses décisions à plus long terme qu'en effectuant une recherche par force brute.

L'objectif de mon système d'apprentissage par auto-observation, appelé Introspect, est de créer automatiquement, pour un domaine donné, les connaissances qui permettent de faire des coupes dans l'arbre de recherche. Pour cela les connaissances du domaine qu'il a au départ sont :

- des définitions simples des buts à atteindre

- une théorie du domaine qui donne les conséquences directes de ses choix

- une base de problèmes accompagnés ou non de réponses ou de solutions .

Un système qui crée automatiquement des connaissances de recherche dans un domaine est intéressant à plus d'un titre :

- Les connaissances de recherche sont très nombreuses, il est long et fastidieux de toutes les exprimer en les donnant directement au système.

- Elles sont compilées dans l'esprit de l'expert, celui-ci n'y a accès qu'au prix de grands efforts de conscientisation [Vermersch 1991].

- Les connaissances données par l'expert ne sont pas toujours celles qu'il utilise mais celles qu'il croit utiliser, ces connaissances décompilées sont souvent erronées ou incomplètes.

Un système qui crée automatiquement des connaissances de recherche à partir d'une définition simple d'un domaine complexe se libère des efforts et des erreurs liées aux pratiques classiques d'acquisition des connaissances.

L'intérêt du jeu de Go

Je montre dans cette section pourquoi le jeu de Go est intéressant à étudier d'un point de vue informatique et plus spécialement du point de vue de l'apprentissage. Je commence par expliquer l'intérêt des jeux de stratégie en général pour l'Intelligence Artificielle, je donne ensuite des raisons

pour lesquelles le jeu de Go est d'un intérêt particulier. Je finis par expliquer ce que l'apprentissage par auto-observation apporte au jeu de Go et comment le jeu de Go facilite les expérimentations sur l'apprentissage.

Les jeux de stratégie

Les jeux de stratégie sont des domaines qui permettent d'étudier et de développer des techniques d'Intelligence Artificielle telles que la recherche arborescente, l'apprentissage, la reconnaissance de formes, la résolution de problèmes ou la représentation des connaissances.

Ces jeux sont assez simples pour être complètement formalisés, mais suffisamment complexes pour fournir de bons modèles pour des problèmes réels.

Des expertises existent et sont facilement accessibles. Les connaissances concernant les jeux de stratégie sont nombreuses et accessibles sans contrainte grâce à une grande variété de joueurs et une abondante littérature.

Les jeux de stratégie sont un bon terrain d'expérimentation pour L'Intelligence Artificielle. En effet, au Go comme aux Échecs, les experts sont nombreux, et entre experts et débutants existe une grande variété de niveaux bien répertoriés. L'intérêt de la diversité et de la fiabilité du classement est primordial lorsqu'on veut valider un système ou un modèle qui s'approche d'un comportement cognitif humain.

Les progrès effectués par un programme peuvent être mesurés aisément grâce aux confrontations avec des joueurs humains dont le niveau d'expertise est connu et reconnu en raison de la compétition permanente qui règne entre eux. Un programme a aussi la possibilité de jouer contre d'autres programmes. Des programmes utilisant des méthodes ou des concepts différents peuvent ainsi être départagés.

Le jeu de Go

> "The research of Go programs is still in his infancy, but we shall see that to bring Go programs to a level comparable with current Chess programs, investigations of a totally different kind than used in computer chess are needed."
>
> John McCarthy 1990

Longtemps les Échecs ont été considérés comme le test parfait pour l'Intelligence Artificielle. Cependant, les bons programmes d'Échecs sont ceux qui envisagent le plus de positions différentes en utilisant une fonction d'évaluation simple et très rapidement calculable [Hsu 1990]. Ils n'utilisent que très peu de connaissances et n'ont pas l'approche généraliste et cognitive qui caractérise les joueurs humains. Au Go, une approche uniquement calculatoire est impossible car on ne peut pas bien approximer l'évaluation d'une position avec une fonction simple et rapidement calculable. De plus, le nombre moyen de coups possibles avoisine 250 alors qu'aux Échecs il est proche de 36. Comme le calcul Alpha-Bêta est exponentiel, la découverte d'une fonction d'évaluation même peu coûteuse ne permettrait pas avec les ordinateurs actuels de calculer plus de 8 coups à l'avance. C'est une des raisons pour lesquelles la programmation du jeu de Go est considérée aujourd'hui comme un défi pour l'informatique [Bradley 1979].

Comparaison du Go et des Échecs [Burmeister 1995] :

Caractéristique	Échecs	Go
Taille du damier	64 cases	361 intersections
Nombre de coups par partie	~40	~250
Nombre de coups légaux	~36	~250
Fin du jeu	Mat (définition claire)	Territoire (consensus)
Changements	Changements rapides	Changements incrémentaux
Evaluation des positions	Bonne corrélation avec le nombre et la valeur des pièces présentes.	Mauvaise corrélation avec le nombre de pierres.
Méthodes de programmation	Recherche arborescente avec une fonction d'évaluation.	Recherche globale impossible. Recherche orientée par un but.
Prévisions humaines	~10 coups	Les débutants peuvent lire plus de 60 coups à l'avance.
Effet d'horizon	Niveau Grand Maître	Niveau Débutant
Processus humain de regroupement	Regroupement hiérarchique [Chase & Simon 1973]	Les pierres font partie de plusieurs groupes [Reitman 1976]
Système de handicap	-	Bon système

Les programmes de Go actuels ne dépassent pas le niveau de débutant de club, et ceci bien que certains programmes aient été améliorés depuis plus de vingt ans [Wilcox & Reitman 1974 1976 1979] [Wilcox 1995] [Fotland 1993] (Cf. chapitre sur le programme de Go).

Pratiquement tous les domaines de l'informatique sont concernés par la programmation du Go. Elle est liée à la théorie des jeux, à la représentation des connaissances en passant par la recherche heuristique, la résolution de problèmes, l'apprentissage, la métaconnaissance, l'amorçage, la gestion de grandes bases de connaissances, les systèmes multi-agents, la logique floue, les réseaux de neurones, les algorithmes génétiques et l'optimisation de programmes.

L'apprentissage du jeu de Go

Les programmes de Go sont difficiles à développer pour de nombreuses raisons exposées dans le tableau ci-dessus. Utiliser des techniques déclaratives d'amorçage et d'apprentissage permet au programmeur de se libérer de beaucoup de contraintes [Pitrat 1990].

L'apprentissage est vu comme l'automatisation de la programmation du jeu de Go. Au fur et à mesure que le programmeur de Go avance dans son projet, il prend conscience des mécanismes de développement d'un programme et des mécanismes cognitifs mis en jeu par le Go. Il devient alors capable de les décrire dans un programme informatique. **Le pas suivant dans l'automatisation est de créer un programme qui écrit lui-même un programme de Go : c'est le but que j'ai poursuivi pendant ma thèse.**

Les expériences peuvent être faites sur de petits damiers, et, si elles sont concluantes, on peut agrandir le damier à volonté. On peut aussi faire varier le handicap. On a ainsi accès à un grand éventail de difficultés [Pell 1991], ce qui permet un apprentissage progressif, et facilite l'amorçage.

Les méthodes traditionnelles de recherche en arbre [Lee 1988] sont inefficaces au Go. Pourtant, tous les joueurs de Go lisent plusieurs coups à l'avance pour les situations les plus simples (par exemple les prises et les connexions), et peuvent lire jusqu'à 60 coups à l'avance pour des séquences que même les débutants connaissent (les Shichos) [Kierulf 1990]. Les programmes déclaratifs qui apprennent quels coups considérer pourraient améliorer les performances des programmes de Go.

Les mécanismes d'apprentissage que j'étudie ne sont pas spécifiques au Go et ils ont été appliqués à d'autres domaines. J'ai choisi le Go parce que c'est un jeu suffisamment complexe pour résister aux

méthodes combinatoires sans intelligence. La nécessité de comprendre les processus sous-jacents aux comportements humains en général est beaucoup plus visible lorsqu'on essaie de programmer le Go que lorsqu'on essaie de programmer le Go-moku ou les Checkers (pour ceux-ci les meilleurs programmes sont combinatoires).

Des recherches très intéressantes ont été effectuées pour les Échecs sur l'apprentissage [Pitrat 1974] mais les programmes qui apprennent n'arrivent pas au niveau des programmes combinatoires. La meilleure manière connue de programmer le Go est d'essayer de mimer au mieux le comportement humain. **Je m'attache tout particulièrement à modéliser l'apprentissage car cela me parait la méthode la plus efficace et la plus rapide pour arriver au degré de connaissance nécessaire à un programme de bon niveau**.

Le fonctionnement d'un système d'apprentissage par auto-observation

Dans les trois sections suivantes, je donne un aperçu de la manière dont Introspect apprend à jouer au Go. Pour cela il doit représenter ses connaissances de façon adéquate. Il doit ensuite apprendre de nouvelles connaissances, puis utiliser efficacement les connaissances qu'il a apprises.

Représentation des connaissances

La façon dont Introspect représente les connaissances est principalement décrite dans deux chapitres de cette thèse.

Le chapitre sur la théorie combinatoire des jeux montre de quelle façon j'ai étendu cette théorie à la représentation de valeurs inconnues. Cette extension donne plus de souplesse dans la façon de représenter l'achèvement des buts poursuivis par le système. Elle permet en premier lieu de définir de façon plus simple et plus efficace les sous-jeux du jeu de Go. Chaque sous-jeu est associé à un seul but à atteindre. Elle permet en second lieu de ne représenter que les parties aisément prouvables d'un arbre de recherche sans avoir à représenter l'ensemble des informations sur l'arbre. Cette représentation plus fine permet à mon système d'avoir des connaissances qui ne seraient pas accessibles dans le formalisme classique qui nécessite une information totale. Mon formalisme permet de représenter des informations partielles lorsque seules celles-ci sont calculables.

Le chapitre sur la représentation des connaissances donne les diverses syntaxes qui me permettent de représenter les différentes sortes de règles utilisées dans mon système. On notera que la représentation la plus utilisée et la plus importante d'Introspect est la représentation logique des règles associée à une représentation méta permettant aux programmes logiques de se manipuler. Toutefois, Gogol peut aussi utiliser des règles à base de patterns géométriques et des règles comportant des méthodes de sélection. Il sait aussi transformer ses règles logiques en programmes C++, j'en parle plus en détail dans la section sur l'utilisation des connaissances et dans le chapitre sur la compilation. Introspect peut utiliser soit des connaissances générales et interprétées donc lentes soit des connaissances très spécialisées et compilées. Le choix de l'utilisation de l'une ou de l'autre dépend du but qu'il se fixe. Quand il cherche à résoudre des problèmes efficacement, il utilise des connaissances spécialisées et compilées. Quand il cherche à résoudre des problèmes pour apprendre de nouvelles règles, il utilise alors des connaissances interprétées afin de pouvoir s'auto-observer et générales afin d'apprendre sur le moins d'exemples possibles.

Apprentissage de nouvelles connaissances

Dans Introspect, l'apprentissage est basé sur l'auto-observation. L'apprentissage comporte trois étapes principales. La première étape est la résolution d'un problème : elle consiste simplement à filtrer les règles du jeu de Go sur un exemple afin de déduire les conséquences d'un coup sur la configuration du damier, puis à filtrer les règles concluant sur l'état des différents jeux combinatoires associés aux sous-buts intéressants du jeu de Go. Pendant la deuxième étape, Introspect détecte les faits déduits qui sont intéressants à prévoir et qu'il n'avait pas prévus, puis utilise la trace de la résolution de problème pour s'expliquer à lui-même pourquoi ces faits ont été déduits. Il en résulte

une liste de faits représentant le damier avant la résolution du problème, dont la présence, quelles que soient les conditions extérieures, assure la déduction du fait qui n'avait pas été prévu. Cette deuxième étape est décrite dans le chapitre sur les explications. Elle permet de créer une règle en mettant la liste de faits en condition et le jeu régressé en conclusion. La troisième étape consiste à généraliser cette règle. Cette généralisation utilise aussi la trace de la résolution de problème. L'étape de généralisation consiste à transformer des constantes en variables. Seules les constantes qui sont des variables instanciées sont transformées en variables, les constantes qui proviennent de constantes dans les règles qui ont permis de résoudre le problème, restent des constantes. La généralisation est décrite dans le chapitre sur l'apprentissage.

Utilisation des connaissances apprises

La façon dont les connaissances sont interprétées dépend du but que Introspect poursuit. S'il cherche à résoudre des problèmes rapidement, il utilise une connaissance spécialisée et compilée. S'il cherche à apprendre, il utilise une connaissance générale et interprétée. Pour pouvoir passer d'une représentation à l'autre il utilise un compilateur en C++. Il utilise aussi un compilateur logique pour transformer les programmes logiques en d'autres programmes logiques toujours aussi généraux et toujours interprétés, mais qui s'exécutent jusqu'à cent mille fois plus vite. Ce compilateur logique utilise des informations sur la répartition moyenne des faits dans la représentation logique des problèmes pour ordonner de façon efficace les prémisses des règles. Ces deux compilateurs sont décrits dans le chapitre sur la compilation.

Une fois apprises et compilées les règles sont utilisées dans un programme qui joue au Go, appelé Gogol. Elles permettent d'accélérer le programme de deux manières. En premier lieu, elles permettent de ne sélectionner que quelques coups à envisager parmi les deux cents cinquante coups légaux envisageables à chaque branche de l'arbre de recherche. Lorsqu'on est à un noeud de l'arbre où l'on cherche à empêcher l'adversaire d'atteindre son but, ces règles sont particulièrement efficaces puisqu'elles ont prouvé que les coups qu'elles ne conseillent pas d'envisager ne marchent pas. En second lieu, elles permettent d'arrêter la recherche avant que le but ne soit complètement atteint. Elles permettent de remplacer l'activité de recherche arborescente par un programme C++ plus rapide. La façon dont ces règles sont utilisées dans mon programme de Go est décrite dans le chapitre sur la programmation du Go.

Architecture générale du système d'apprentissage

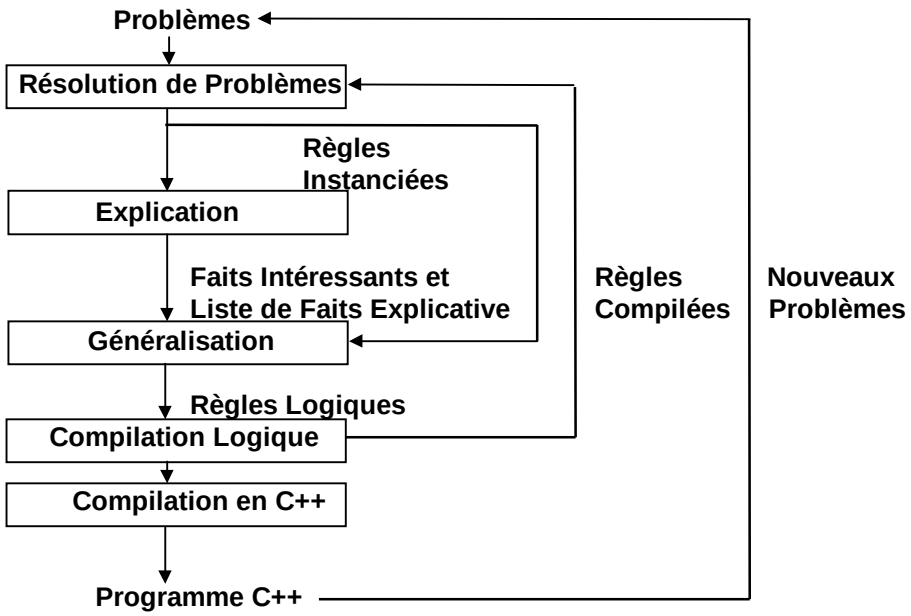


Figure Architecture générale du système d'apprentissage

Les différentes parties de Introspect dont je viens de donner un aperçu sont reliées entre elles pour former un tout cohérent représenté dans la figure Architecture générale du système d'apprentissage.

Introspect a été principalement utilisé pour écrire un programme de Go. Il a aussi été utilisé pour écrire des règles de prise de décision pour le jeu d'Abalone et pour la Gestion. Le programme de Go est beaucoup plus développé que les programmes d'Abalone et de Gestion qui ne sont que des prototypes permettant de montrer que Introspect peut être utilisé dans des domaines complexes très différents.

Modes de lecture de la thèse

Une lecture très rapide de la thèse serait de lire les chapitres : Introduction, 1 et Conclusion.

Une lecture rapide mais approfondie de la thèse serait de lire les chapitres : Introduction, 1, 2.1 et 2.2, 4.1, 5.3, 6, 7 et Conclusion. Il est indispensable de lire le chapitre 4.1 avant le chapitre 5.3.

Les chapitres qui intéresseront plus particulièrement les joueurs de Go sont : Introduction, 1, 2, 7, Conclusion et les Annexes.

Les chapitres qui intéresseront plus particulièrement les informaticiens sont : Introduction, 1, 3, 4, 5, 6 et Conclusion.

Pour avoir un idée de la généralité de l’approche utilisée dans cette thèse, la lecture des chapitres Introduction, 1, 8, 9 et Conclusion est conseillée.

Table des Matières du Chapitre 1

1 UN EXEMPLE DE FONCTIONNEMENT D'INTROSPECT 13

1.1 Se rendre compte de son ignorance 13
1.1.1 Résoudre un problème 13
1.1.2 Repérer les surprises 14

1.2 Créer les connaissances qui manquent en s'auto-observant 14
1.2.1 Résoudre un problème en gardant une trace 14
1.2.2 Utiliser la trace pour expliquer les faits intéressants 15
1.2.3 Généraliser 15

1.3 Utiliser les connaissances créées 15
1.3.1 Compiler les règles 15
1.3.2 Intégrer les connaissances dans un programme 16

1.4 Architecture Détaillée 16

1 Un exemple de fonctionnement d'Introspect

Le but de ce chapitre est de montrer sur un exemple le fonctionnement d'Introspect, de façon à en donner un vue intuitive et générale avant de donner une description plus complète de chacun de ses mécanismes. Je commence par montrer comment il est capable de se rendre compte par lui-même qu'il peut s'améliorer lorsqu'il rencontre un problème qu'il n'a pas su résoudre. Je continue en décrivant comment il s'observe en train de résoudre un problème afin d'apprendre à résoudre des problèmes similaires par la suite. Je montre enfin comment ces nouvelles connaissances de résolution de problèmes sont intégrées dans un programme.

1.1 Se rendre compte de son ignorance

Pour pouvoir progresser, un système doit d'abord se rendre compte qu'il ne fait pas certaines choses qui doivent être faites. Introspect est capable de détecter les événements surprenants. Il résout des problèmes et quand un coup joué amène une information nouvelle qu'il n'avait pas prévue, il se souvient de la position et du coup joué afin de créer une connaissance qui lui permettra de prévoir cette information par la suite.

1.1.1 Résoudre un problème

Sur le damier de la figure Exemple Damier 1, Introspect cherche à savoir si les chaînes A et B sont connectées. Pour cela, il fait les calculs donnés dans les figures Exemple Séquence 1 et 2.

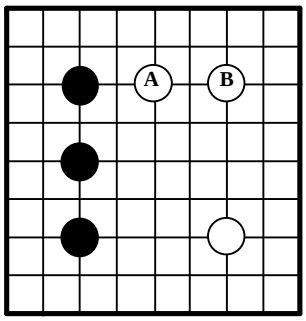


Figure Exemple Damier 1

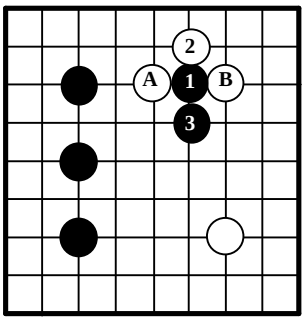


Figure Exemple Séquence 1

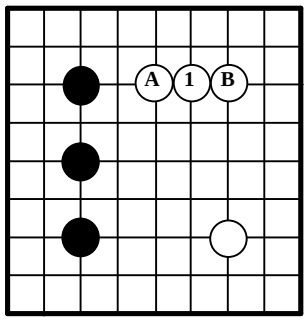


Figure Exemple Séquence 2

Le calcul de la figure Exemple Séquence 2 lui permet de conclure que si blanc joue en premier, il peut connecter ses deux pierres. Lorsqu'il fait le calcul de la figure Exemple Séquence 1, il s'arrête au coup noir numéro 3, car dans cette position, il ne connaît pas de coup qui menace de connecter les chaînes blanches A et B. Il conclut que noir est forcé de jouer en 1 pour essayer de déconnecter les chaînes blanches A et B.

1.1.2 Repérer les surprises

Introspect, suite aux calculs présentés, a joué deux coups forcés pour déconnecter les blocs A et B. Il se retrouve alors dans la position de la figure Exemple Damier 2. Dans cette position, il pense que les blocs A et B sont déconnectés car il ne connaît pas de coup qui menace de les connecter.

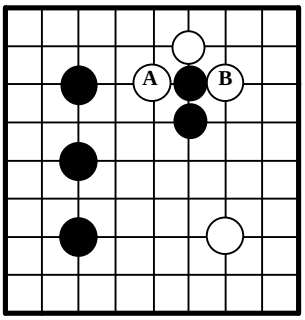


Figure Exemple Damier 2

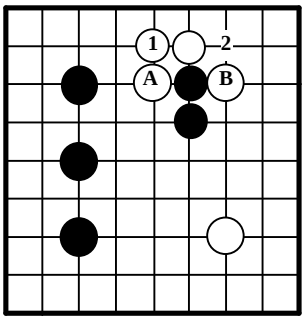


Figure Exemple Coup

Or, dans cette position, l'adversaire du programme joue le coup en 1 de la figure Exemple Coup. Introspect se rend alors compte que blanc peut connecter les blocs A et B après ce coup en jouant sur l'intersection marquée avec un 2 dans la figure Exemple Coup.

Il réalise alors que le coup blanc en 2 est une menace de connecter les chaînes A et B. Il sait aussi qu'il ne sait pas prévoir que ce coup est une menace. Il va donc se souvenir de cette position et du coup associé pour pouvoir les analyser par la suite et apprendre à prévoir ce type de menaces.

1.2 Créer les connaissances qui manquent en s'auto-observant

Le but de cette section est de montrer comment Introspect crée les connaissances qui lui manquent sur les effets des coups. Pour créer ces nouvelles connaissances, il doit d'abord comprendre pourquoi un coup permet de se rapprocher d'un but. Pour cela, il doit refaire les déductions qui l'ont amené à se rendre compte que ce coup avait des effets qu'il n'avait pas prévus. Il doit ensuite observer la façon dont il a déduit que le coup était intéressant pour créer une règle qui lui permette d'envisager ce coup. Une fois qu'il a créé une règle qui permette d'envisager ce coup sur l'exemple étudié, il généralise cette règle pour qu'elle s'applique sur le plus d'exemples possibles tout en restant toujours vraie.

1.2.1 Résoudre un problème en gardant une trace

Pour comprendre pourquoi le coup 1 de la figure Exemple Coup est une menace, Introspect associe le damier de la figure Exemple Damier au coup 1. Il transforme le damier et le coup en une base de faits qui représente la situation.

Une fois cette base créée, Introspect applique dessus les règles de transition du jeu de Go. Ce sont les règles qui déduisent les faits représentant la position après le coup à partir des faits représentant la position avant le coup et du fait représentant le coup. Une partie de ces règles est donnée dans l’annexe A.

Une fois ces règles filtrées, Introspect filtre les règles concernant l'état d'achèvement des buts avant le coup et après le coup. Ainsi dans notre exemple, il ne déduit rien sur la connexion entre A et B avant le coup, mais il déduit que A et B sont connectables après que le coup ait été joué.

Il se rend alors compte que le coup est une menace de connecter A et B, et il sait qu'il n'a pas déduit cette menace avant le coup. Il a donc déduit un fait intéressant : **le coup 1 est une menace de connecter les chaînes A et B**.

1.2.2 Utiliser la trace pour expliquer les faits intéressants

L'étape suivante est de comprendre comment il a fait pour déduire ce fait intéressant en observant la trace des déductions. Pour cela, il remonte la trace des déductions jusqu'à ce que l'explication de la menace soit uniquement composée de faits représentant le damier avant que le coup n'ait été joué.

Il obtient alors une règle qui a pour conclusion que le coup 1 est une menace de connecter A et B et pour conditions un sous-ensemble de l'ensemble des faits représentant la position avant le coup.

1.2.3 Généraliser

La règle obtenue après l'explication du fait intéressant est juste et s'applique sur l'exemple qui nous intéresse. Toutefois, il est important qu'elle puisse s'appliquer dans de nombreux autres exemples similaires. On la généralise donc pour qu'elle puisse s'appliquer dans le plus grand nombre de cas possibles.

Pour cela, on utilise la trace de la résolution de problème. Dans la nouvelle règle qui ne contient que des constantes, on distingue les constantes qui sont des instanciations de variables et les vraies constantes. Les variables instanciées sont transformées en variables. Cette transformation permet de rendre une règle beaucoup plus générale tout en assurant que ses conclusions seront toujours vraies.

1.3 Utiliser les connaissances créées

Créer de nouvelles connaissances qui permettent de mieux prévoir est nécessaire pour s'améliorer. Il est tout aussi nécessaire que ces connaissances soient utilisables efficacement. Or, les règles créées par les mécanismes précédents sont générales et justes, et permettent de mieux prévoir, mais elles ne sont pas utilisables dans un programme qui joue, à cause de leur lenteur. Il faut donc les transformer pour qu'elles soient utilisables efficacement. Pour cela, Introspect les compile.

1.3.1 Compiler les règles

La première étape de la compilation est d'ordonner les tests et les instanciations multiples dans les règles. Pour cela Introspect dispose d'informations sur la fréquence moyenne d'apparition des faits dans une base de faits. Il réordonne les prémisses d'une règle de façon à ce qu'elle soit filtrée plus rapidement. Réordonner les prémisses permet de filtrer les règles jusqu'à 100 000 fois plus rapidement.

Une fois que les prémisses sont réordonnées, il est intéressant de spécialiser les règles. La spécialisation permet de calculer au moment de la compilation des prémisses qui sont calculées à l'exécution dans les règles générales. Elle permet donc de faire certains calculs une fois pour toutes et rend donc le filtrage plus rapide.

L'étape suivante consiste à transformer les règles logiques compilées en programmes C++. En effet l'auto-observation n'est pas nécessaire dans le programme qui joue, elle ne sert que pour le programme qui apprend. On peut donc utiliser des connaissances compilées dans le programme qui

joue. La transformation des règles logiques interprétées en programmes C++ compilés permet de gagner un facteur 60 en rapidité.

1.3.2 Intégrer les connaissances dans un programme

Les programmes C++ créés sont intégrés dans le programme qui résout des problèmes. Les connaissances apprises sont de deux types : d'une part, les connaissances qui permettent de savoir qu'un but est atteint ou qu'un but est atteint si l'on joue ; ces connaissances sont utilisées pour évaluer les feuilles d'un arbre de recherche bien avant que le but ne soit atteint. Elles ont pour but de rendre les arbres de recherche moins profonds. D'autre part, il y a les connaissances sur les menaces d'atteindre un but et les connaissances sur les coups forcés pour empêcher d'atteindre un but, qui sont utilisées pour engendrer les coups à envisager pour développer l'arbre. Elles ont pour but de rendre l'arbre moins large en ne sélectionnant que quelques coups parmi les centaines de coups légaux possibles.

Ces connaissances sont utilisées pour résoudre des problèmes et permettent aussi de se rendre compte de nouveaux manques dans les connaissances du programme et de créer de nouveaux problèmes intéressants, donc de nouvelles règles intéressantes qui vont être intégrées au programme et ainsi de suite.

Le principe du programme est de partir uniquement des règles qui donnent les effets des coups et des définitions des buts intéressants pour se programmer lui-même afin de prévoir de mieux en mieux les conséquences de ses coups, et donc de créer des arbres de recherches de plus en plus courts et de moins en moins larges.

Ce processus s'arrête lorsque les connaissances créées prennent trop de place en mémoire pour qu'on puisse en rajouter ou lorsque le programme créé devient trop lent. Le niveau du programme résultant de l'apprentissage est directement dépendant de la mémoire disponible et de la vitesse de la machine utilisée.

1.4 Architecture Détaillée

La figure Exemple Architecture Détaillée donne une vue générale d'Introspect.

Les cases qui contiennent des caractères gras sont les éléments dont le système dispose au début de l'amorçage. Ces sont les parties du système que j'ai écrites.

Les règles contenues dans la base 'Règles de régression' sont les règles qui définissent les buts intéressants et les règles qui permettent de détecter les surprises dans l'évolution des jeux combinatoires à valeurs inconnues.

Les règles contenues dans la base 'Règles de transition' sont les règles qui permettent de calculer les faits représentant l'état du Goban après le coup à partir des faits représentant l'état du Goban avant le coup.

Les règles contenues dans la base 'Métarègles de compilation' sont les règles qui réordonnent les prémisses en fonction de leurs fréquences dans les bases de faits. Ce sont aussi les règles sur les symétries équivalentes d'un prédicat (par exemple le prédicat 'Voisine (?i ?i1)' est équivalent au prédicat 'Voisine (?i1 ?i)').

Les cases qui entourées par des traits plus épais forment le système qui joue au Go : Gogol.

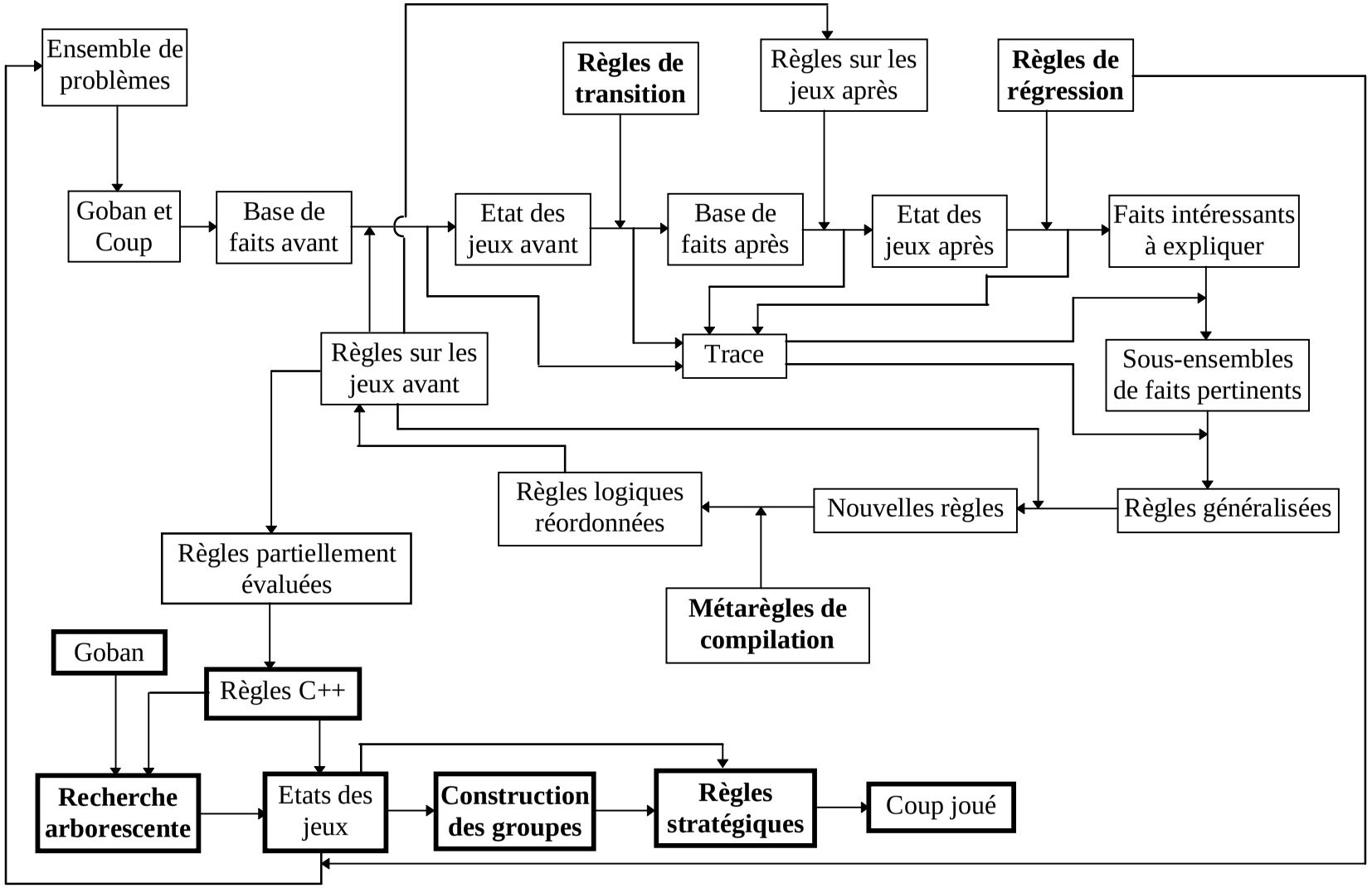


Figure Exemple Architecture Détaillée

Table des Matières du Chapitre 2

2 LA THÉORIE COMBINATOIRE DES JEUX 18

2.1 État de l'art 18
2.1.1 Introduction à la théorie combinatoire des jeux 18
2.1.2 Théorie combinatoire des jeux et jeu de Go 20

2.2 Extension de la théorie combinatoire des jeux à la représentation de l'inconnu 22
2.2.1 Définition des jeux combinatoires comportant des valeurs inconnues 22
2.2.2 Une taxonomie des jeux combinatoires à valeurs inconnues 23

2.3 Gestion du risque 24
2.3.1 Evaluation des jeux combinatoires 24
2.3.2 Trois stratégies de gestion du risque 25
2.3.3 Utilisation de la taxonomie 26
2.3.4 Exemple d'utilisation 26

2.4 Conclusion 28

2.5 Bibliographie 28

2 La Théorie Combinatoire des Jeux

Ce chapitre comprend dans une première section une description succincte de la théorie combinatoire des jeux, suivie de l'utilisation qui en a été faite jusqu'ici dans le jeu de Go. Je propose dans une deuxième section une extension de cette théorie qui permet de représenter l'inconnu. La représentation de l'inconnu donne la possibilité d'utiliser la théorie combinatoire des jeux dans des jeux complexes. Elle donne aussi le moyen de définir les jeux de façon simple et générale en les associant à un seul but. Je montre enfin dans la troisième section comment cette représentation de l'inconnu permet de gérer aisément le risque.

2.1 État de l'art

Une théorie mathématique autorise un parallèle entre jeux et nombres. Le lecteur mathématicien peut se référer à [Conway 1976], le lecteur ludique préférera plutôt [Berlekamp 1982]. Une description en français des jeux de Conway est donnée dans [Alliot 1994].

2.1.1 Introduction à la théorie combinatoire des jeux

Un jeu de Conway est un jeu à deux joueurs (Gauche et Droite) et à information complète. Le jeu se termine par la défaite d'un des deux joueurs : c'est le premier joueur qui ne peut plus jouer qui a perdu.

Définition : Un jeu de Conway x, noté x = {G|D} est formé à partir de deux ensembles G et D (éventuellement vides) de jeux de Conway. Tous les jeux de Conway sont formés de cette façon. G est l'option de jeu qui se présente au joueur Gauche. D est l'option de jeu qui se présente au joueur Droit.

Un jeu de Conway permet de définir un arbre des positions atteignables à partir d'une position initiale. Si cet arbre contient toutes les positions atteignables depuis la position initiale, il est possible

d'étiqueter chacun des noeuds de l'arbre avec deux étiquettes. La première étiquette contiendra soit Gagnant pour Droit (GD) soit Perdant pour Droit (PD). La deuxième étiquette contiendra soit Gagnant pour Gauche (GG) soit Perdant pour Gauche (PG). Pour un noeud donné, on peut donc avoir 4 cas :

	PG	**GG**
PD	Le joueur qui commence, perd, le jeu est NUL (0)	Le joueur Gauche gagne à tous les coups, le jeu est POSITIF (+)
GD	Le joueur Droit gagne à tous les coups, le jeu est NEGATIF (-)	Le joueur qui commence, gagne, le jeu est FLOU (||)

Tableau Jeux de Conway

Définition : Soit x un jeu de Conway, le jeu opposé de x, noté -x, est le jeu où, pour toute position, les options de Gauche et de Droit ont été inversées.

Définition : Soient x et y deux jeux de Conway, le jeu x+y sera le jeu dans lequel chaque joueur peut, chaque fois que c'est son tour, décider de jouer dans x ou dans y.

Dans le jeu x+y, Droit peut décider de jouer dans x et le jeu obtenu sera la somme de l'option droite de x et de y, ou décider de jouer dans y et dans ce cas, le jeu obtenu sera la somme de x et de l'option droite de y.

On notera : $x+y = \{x^D+y, x+y^G \mid x^D+y, x+y^D\}$

Et de façon générale : $x+0 = \{x^G+0 \mid x^D+0\} = \{x^G \mid x^D\} = x$

0 est un élément neutre pour +.

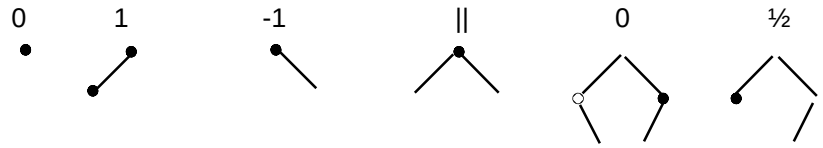


Figure Nombres de Conway

La figure Nombres de Conway donne quelques jeux de Conway représentés sous forme d'arbre. Par exemple le jeu associé au nombre ½ peut être gagné immédiatement si Gauche joue, toutefois Droit peut aussi jouer dans ce jeu et Gauche rajoute alors un coup pour le gagner.

On a aussi de façon immédiate : 1 + (-1) = 0

On montre aussi que : ½ + ½ + (-1) = 0

Dans le jeu ½ + ½ + (-1) :

- Si Gauche commence, il joue dans un des jeux ½, Droit répond dans l'autre ½ et on se retrouve dans un jeu 1 + (-1).

- Si Droit commence, il joue de préférence dans un des ½, Gauche répond dans l'autre ½ et on se retrouve aussi dans un jeu 1 + (-1).

Définition : Soient x et y deux jeux de Conway, on a $x \leq y$ si et seulement si :

- il n'existe aucun y^D tel que $y^D \leq x$ et
- il n'existe aucun x^G tel que $y \leq x^G$

Définition : Un nombre de Conway x = {G|D} est un jeu de Conway, formé de deux ensembles G et D de nombres de Conway, tels qu'il n'existe aucun élément de D qui soit ≤ à un élément de G.

2.1.2 Théorie combinatoire des jeux et jeu de Go

La théorie des jeux de Conway a été appliquée au jeu de Go par E. Berlekamp [Berlekamp 1991] et son équipe [Wolfe 1991]. Ils ont surtout étudié les situations qui se présentent dans les derniers coups d'une partie. Ce travail a permis de mettre en évidence de nouveaux concepts du Go très spécifiques de la fin de partie. Sur certaines positions, l'utilisation de cette théorie pour choisir les coups permet de gagner 1 point par rapport au jeu des joueurs professionnels [Berlekamp 1994]. L'application de cette théorie dans un programme qui joue au Go est cependant très limitée, puisqu'elle ne s'applique qu'à la toute fin de partie, que les jeux considérés doivent être indépendants et que les coups envisagés par le programme ne doivent pas modifier la vie des groupes.

Un concept important pour le jeu de Go, mis à jour dans [Berlekamp 1982] est le concept de Température. La température d'un jeu est, dans les cas simples, la moitié de la valeur absolue de la différence de points à la fin du jeu entre une position où Droite a joué et la même position dans laquelle Gauche a joué un coup dans le jeu. Cette notion de Température est liée à la notion de Sente[1] utilisé par les joueurs de Go.

Une application plus générale de cette théorie à toutes les phases du jeu a été tentée par B. Bouzy [Bouzy 1995] et indépendamment par M. Müller [Müller 1995].

M. Müller est plus proche des théories de Berlekamp que B. Bouzy. Il étudie plus particulièrement les jeux dans lesquels une répétition de la position est interdite (ce qui est le cas du Go avec la règle du Ko). Il a utilisé ces théories dans un programme de bon niveau (16ème Kyu) : Swiss Explorer.

B. Bouzy a étendu le concept de jeu combinatoire à toutes les phases de la partie. Les problèmes qu'il a rencontrés sont dus aux dépendances entre sous-jeux et à l'hétérogénéité des jeux.

Il distingue les jeux statiques et les jeux dynamiques :

- Un jeu a un état statique défini par les règles du jeu : Gagné, Perdu, Autre.

- Un jeu a un état dynamique, qui est l'état du jeu calculé quelques coups à l'avance.

Contrairement à la description de la théorie de Conway donnée dans la section précédente, les états Perdu et Gagné sont définis par rapport à l'un des joueurs. Dans le tableau suivant, on adopte toujours le point de vue du joueur Gauche (Pd=Partie Droite, Pg=Partie Gauche) :

	Pd = Perdu	Pd = Gagné	Pd = Autre
Pg = Perdu	Le jeu est toujours perdu pour Gauche : '**<**'	Le jeu est Perdu si Gauche joue et Gagné si Droite joue : '**0**'	'**?**'
Pg = Gagné	Le jeu est Gagné si Gauche joue, Perdu si Droite joue : '*****'	Le jeu est Gagné si Gauche joue et Perdu si Droite joue : '**>**'	'**?**'
Pg = Autre	'**?**'	'**?**'	'**?**'

Tableau Jeux de Bouzy

B. Bouzy a également introduit plusieurs propriétés pour un jeu :

- Une hiérarchie père-fils entre les jeux : un jeu père peut avoir plusieurs jeux fils; le jeu global est le père de tous les jeux, le jeu de l'intersection est en bas de la hiérarchie et n'a pas de fils.

[1] Un coup est Sente s'il appelle une réponse forcée.

- Un lieu statique et un lieu dynamique pour les jeux : le lieu statique est l'ensemble minimal d'intersections utilisé pour engendrer des coups dans un jeu, le lieu dynamique est la réunion de tous les lieux statiques rencontrés lors du calcul d'un jeu.

- Des jeux sont indépendants si leurs lieux ont une intersection vide.

- Des règles de recomposition : si deux jeux sont indépendants, on peut utiliser les règles de recomposition de résultats de jeux pour connaître le résultat du surjeu obtenu par disjonction ou conjonction.

Exemples :

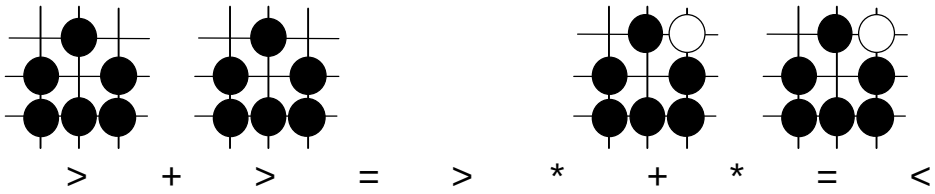


Figure Jeu Règles de Recomposition

Dans la figure Jeu Règles de Recomposition, deux jeux de l'oeil sont additionnés pour calculer un jeu de la base de vie. Au jeu de Go, un groupe est vivant s'il a deux yeux.

Exemples de calculs de jeux dynamiques pour le jeu de la chaîne :

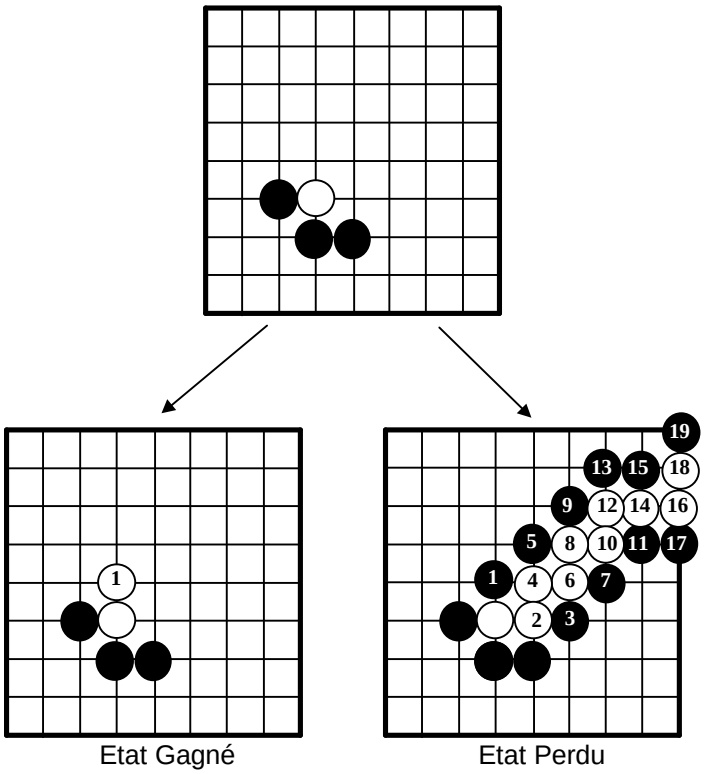


Figure Jeu *****

Le jeu de la chaîne est défini comme suit :

- Il est Perdu si la chaîne est retirée du damier (si elle n'a plus de libertés).

- Il est Gagné si la chaîne a plus de 3 libertés.

La Figure Jeu * montre le calcul du jeu de la chaîne pour une pierre blanche. Si Blanc joue en premier, le jeu de la chaîne est Gagné car blanc obtient immédiatement 4 libertés. Si Noir joue en premier, il peut ôter la pierre blanche du damier grâce à une suite d'Ataris[2]. Le jeu est donc '*****'.

2.2 Extension de la théorie combinatoire des jeux à la représentation de l'inconnu

2.2.1 Définition des jeux combinatoires comportant des valeurs inconnues

Les jeux définis par J. Conway sont limités par leur absence de représentation de l'inconnu. Il ne peuvent représenter que des situations dans des jeux simples dans lesquels toutes les situations possibles découlant de la situation actuelle peuvent être calculées et évaluées. Pour utiliser ces jeux dans des jeux plus complexes comme les sous-jeux du jeux de Go, il devient intéressant de représenter l'inconnu dans les jeux combinatoires. **La représentation de l'inconnu permet de rendre compte du fait que la recherche est impossible parce que trop longue dans certaines branches de l'arbre de jeu**.

B. Bouzy a déplacé le problème de la représentation de l'inconnu en définissant pour chaque sous-jeu du jeu de Go, à la fois un état Gagné et un état Perdu : par exemple, pour le jeu de la chaîne, l'état Perdu est atteint quand la chaîne est retirée du damier alors que l'état Gagné est défini par un nombre minimal de libertés. Or le jeu de prendre un bloc de pierres n'est pas Perdu parce qu'il a un nombre minimal de libertés. Tout ce qu'on sait, c'est que le système ne connaît pas de manière sûre de prendre le bloc. Mais, le bloc n'est pas toujours sauvé. Dans l'architecture du programme de B. Bouzy, la prise des pierres ayant plus de trois libertés est décidée par un niveau supérieur du programme. Toutefois ces niveaux supérieurs ne sont pas utilisés dans la recherche arborescente, ce qui l'empêche de voir la prise de pierres ayant plus de trois libertés si cette prise intervient dans un calcul.

J'ai pris le parti de ne pas reporter sur la définition des jeux le problème de l'inconnu. La limitation de la recherche due aux contraintes de temps, oblige à gérer l'inconnu (en pratique dans les jeux complexes, on ne peut pas connaître tout l'arbre de jeu, même dans un sous-jeu). **J'ai plutôt choisi de représenter explicitement ce que le système ne sait pas. Cette façon de faire permet aussi de définir de façon plus simple et plus claire les sous-jeux du jeu de Go**. Ainsi, pour définir l'homologue du jeu de la chaîne, il me suffit de spécifier les conditions dans lesquelles la prise d'un bloc de pierre est Gagnée et je n'ai pas besoin de définir les conditions dans lesquelles elle est Perdue. Cela permet d'avoir à la fois des jeux plus simples et plus généraux, et donne la possibilité au système de faire clairement la différence entre ce qu'il sait et ce qu'il ne sait pas.

Un jeu est associé à un but à atteindre. En théorie, il n'y a que deux états finaux possibles : Gagné (G) et Perdu (P). En pratique, il est souvent très coûteux de trouver la valeur d'un jeu [Allis 1994]. De ce fait, j'ai introduit **la valeur Inconnu (I)** qui rend compte du fait que l'on n'a pas eu le temps de calculer la valeur d'un jeu. On définit l'ensemble J = {G,I,P} qui représente l'ensemble des états finaux possibles pour un jeu.

La définition dynamique d'un jeu consiste à définir les états d'un jeu après avoir joué ou après que l'adversaire ait joué, si $x,y \in J$: {x|y} définit le jeu pour lequel le meilleur état que peut atteindre le joueur s'il joue en premier est x, alors que le meilleur état que peut atteindre l'adversaire s'il joue en premier est y. On notera $J_1 = \{ \{x|y\} / x,y \in J\}$. On peut de manière plus générale définir $J_n = \{ \{x|y\} /$

[2] Un bloc de pierres est en Atari si il ne lui reste plus qu'une seule liberté.

$x,y \in J_{n-1}\}$. Calculer un élément de J_n est deux fois plus coûteux que calculer un élément de l'ensemble J_{n-1}, on s'en tiendra donc ici aux éléments de l'ensemble J_2.

Pour simplifier la notation, j'enlèverai les accolades et les barres entre les jeux par la suite. Cette notation est équivalente à la notation de Conway pour les jeux de J, J_1 et J_2. Les jeux ont toujours la barre au milieu.

En pratique J_1 et J_2 sont suffisants pour représenter les connaissances utilisées par les joueurs, les jeux les plus connus sont les suivants :

- G indique un jeu gagné.

- {G|I} = GI indique un jeu qu'il est possible de gagner si Gauche joue en premier.

- P indique un jeu perdu.

- {P|G} = PG indique un Zugswang aux Echecs ou un Seki au Go : c'est une situation dans laquelle le premier joueur qui joue perd le jeu. Si Gauche joue en premier, Gauche perd le jeu et si Droite joue en premier, Gauche gagne le jeu.

- {I,I|I,P} = IIIP représente une menace pour Droite de faire perdre Gauche. Si Gauche joue en premier, il se retrouve dans un jeu {I|I} = II qui correspond à un état inconnu, il a donc évité la menace de Droite. Si Droite joue en premier, Gauche se retrouve avec un jeu {I|P} = IP qui correspond à un coup forcé pour Gauche. Si Droite joue deux fois de suite, Gauche perd le jeu.

- {G,I|I,I} = GIII représente une menace pour Gauche de gagner. Si Gauche joue deux fois il atteint le but. Au premier coup, il se retrouve dans un jeu {G|I} = GI, et au deuxième coup dans un jeu G.

On a J_1 = {GG, GI, GP, IG, II, IP, PG, PI, PP}. J_2 contient 3^4=81 eléments.

2.2.2 Une taxonomie des jeux combinatoires à valeurs inconnues

Une taxonomie sur les jeux permet d'optimiser le temps de déduction des jeux. Il est par exemple inutile de matcher les règles portant sur un jeu plus général qu'un jeu déjà déduit (Cf. chapitre sur la compilation). Cette taxonomie permet aussi de raisonner sur les arbres à développer en fonction des jeux déjà connus (Cf. chapitre sur le programme de Go).

Cette taxonomie est utilisée pour optimiser la déduction des jeux grâce à des relations d'héritage entre les jeux. Cette taxonomie sera utilisée dans les chapitres sur l'apprentissage et sur la compilation, elle a aussi des liens avec l'oubli (volontaire) de connaissances et le choix des noeuds à développer dans la recherche arborescente.

Pour construire une taxonomie des jeux que j'ai définis, je définis la relation "est un". La valeur I est plus générale que les valeurs G ou P et aussi plus générale que tous les jeux. Ainsi on a GG "est un" GI. De plus, on peut construire une relation d'héritage entre J_1 et J_2. On a par exemple GIPP "est un" IP. La figure suivante donne une idée de la taxonomie ainsi engendrée :

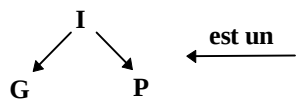


Figure Taxonomie des jeux de J

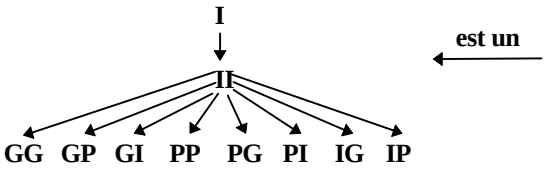


Figure Taxonomie des jeux de J_1

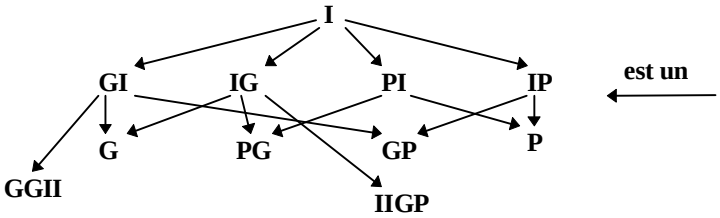


Figure Taxonomie partielle des jeux

La figure Taxonomie partielle des jeux donne une idée de ce qu'est la taxonomie totale des jeux. Les flèches représentent la relation "est un". Je n'ai pas représenté la taxonomie dans son ensemble car elle contient trop d'éléments (93) et trop de relations entre ces éléments.

Pour clarifier les idées, prenons l'exemple du jeu GG. Il a pour descendants directs dans la taxonomie les jeux : IGGI, IGGG, IGGP, GGGI, GGGG, GGGP, PGGI, PGGG et PGGP. On a donc IGGI "est un" GG.

2.3 Gestion du risque

On peut traiter différemment l'inconnu suivant que l'on est optimiste ou pessimiste. Ma représentation des jeux combinatoires est basée sur la représentation de l'inconnu. Elle se prête donc bien à des manipulations liées à la gestion de l'inconnu. La gestion du risque est liée au comportement face à l'inconnu. Le comportement le plus risqué est à celui de l'optimiste qui pense toujours que l'inconnu joue en sa faveur. Le comportement le plus prudent est celui du pessimiste ou du paranoïaque qui croient que l'inconnu joue toujours contre eux. Dans les situations normales, il vaut mieux avoir un comportement mitigé. Dans les situations désespérées, il vaut mieux avoir un comportement risqué et optimiste. Par contre, dans les situations où le gain est assuré, la meilleure stratégie à suivre est d'être prudent et pessimiste. Je vais montrer comment ces stratégies de prise de risque sont modélisées avec mon extension de la théorie combinatoire des jeux.

2.3.1 Evaluation des jeux combinatoires

J'attribue des valeurs aux différents états de J. Ainsi je définis :

Val (G) = 1, Val (P) = -1.

Je multiplie ensuite la valeur de l'état par l'importance du sous-but correspondant. La somme des états pour tous les sous-buts donne l'évaluation de la position.

A partir des valeurs des états de J, on peut donner des valeurs aux coups définis par les éléments de J_1 et J_2 :

$\forall$ x,y $\in$ J : Val (xy) = Val (x) - Val (y).

Val (xy) est la valeur du coup permettant d'atteindre l'état x dans le jeu xy. Cette valeur rend compte de ce que jouer un coup d'un jeu de J_1 permet d'atteindre l'état x mais aussi d'empêcher l'adversaire d'atteindre l'état y. Lorsqu'on utilise la fonction Val pour noter un jeu, on fait l'hypothèse que les jeux sont indépendants. Dans mon programme de Go, Val est utilisé pour le sous-jeu de la connexion (Cf. chapitre sur le programme de Go).

$\forall$ x,y,x_1, $y_1 \in$ J :

Valc (xyx_1y_1) = (Val (x) + Val (y))/2 - (Val (x_1) + Val (y_1))/2

Valc (xyx_1y_1) est la valeur du coup qui permet d'atteindre l'état xy tout en empêchant l'adversaire d'atteindre l'état x_1y_1. Cette valeur rend compte que si l'on choisit un état xy, on aura approximativement autant de chance pour la suite de la partie d'être dans l'état x que dans l'état y. Et que si l'adversaire choisit l'état x_1y_1, il aura autant de chance d'atteindre l'état x_1 que l'état y_1. Cette approximation vient du fait que lorsque une position comporte un grand nombre de sous-jeux, un coup permet de jouer à plusieurs sous-jeux à la fois et les sous-jeux sont donc dépendants les uns des autres. On arrivera en moyenne à gagner dans la moitié des sous-jeux auxquels on joue. Cette approximation est valable par exemple pour le jeu de la connexion au vide au Go (Cf. chapitre sur le programme de Go). C'est une caractéristique des jeux complexes. Lorsqu'on utilise la fonction Valc pour noter un jeu, on fait l'hypothèse que les sous-jeux sont dépendants.

Pour des sous-jeux plus simples que l'on considère comme indépendants les uns des autres, on peut définir une autre mesure des coups : Vals.

$\forall$ x,y,x_1, $y_1 \in$ J : Vals (xyx_1y_1) = min (Val (x),Val (y)) - max (Val(x_1),Val (y_1))

Vals est la valeur du coup qui permet d'atteindre l'état xy dans un jeu simple pour lequel les interactions peuvent être négligées. Si un coup ne joue qu'à un seul jeu, sa valeur est la différence entre le meilleur état atteint, si on le joue, et le meilleur état atteint si l'adversaire le joue.

2.3.2 Trois stratégies de gestion du risque

Nous avons défini les valeurs de Gagné et de Perdu mais nous n'avons pas défini de valeur pour l'état Inconnu. Cette valeur dépend de la stratégie choisie. On définit trois stratégies principales :

- la stratégie risquée, qui est une stratégie optimiste visant à détruire les positions de l'adversaire sans s'occuper des siennes. On l'utilise lorsqu'on est en train de perdre dans une partie et que l'on cherche à déstabiliser la situation, donc à maximiser l'incertitude : Val (I) = 1.

- la stratégie prudente, qui est une stratégie pessimiste visant à conserver ses positions sans s'occuper de celles de l'adversaire. On l'utilise lorsqu'on gagne largement une partie et que l'on cherche à stabiliser la situation, donc à minimiser l'incertitude : Val (I) = -1.

- la stratégie mixte, qui est une stratégie neutre visant à conserver ses positions et à affaiblir celles de l'adversaire. On l'utilise lorsqu'une partie est serrée. On donne à l'incertitude une valeur neutre : Val (I) = 0.

On calcule dans le tableau Stratégies J_1 les valeurs associées au coup ami des jeux de J_1 pour les différentes stratégies :

Jeux\Stratégies	Risquée	Mixte	Prudente
GG	0	0	0
GI	0	1	2
GP	2	2	2
IG	0	-1	-2
II	0	0	0
IP	2	1	0
PG	-2	-2	-2
PI	-2	-1	0
PP	0	0	0

Tableau Stratégies J_1

Par exemple, il n'est pas intéressant de jouer dans un jeu GI si l'on a une stratégie risquée alors que ce jeu est beaucoup plus intéressant si l'on a une stratégie prudente. La valeur associée au jeu GI est en accord avec ce principe, elle vaut 0 pour la stratégie risquée alors qu'elle vaut 2 pour la stratégie prudente.

On donne dans le tableau Stratégies J_2 les valeurs associées au coup de certains jeux de J_2 pour les différentes stratégies dans un jeu complexe utilisant Valc.

Jeux\Stratégies	Risquée	Mixte	Prudente
GGGG	0	0	0
GGGI	0	1/2	1
GGII	0	1	2
GIGI	0	0	0
GIII	0	1/2	1
IIII	0	0	0
IIIP	1	1/2	0
IPIP	0	0	0
IIPP	2	1	0
IPPP	1	1/2	0
PPPP	0	0	0

Tableau Stratégies J_2

On peut utiliser une fonction continue pour représenter le risque si on ne se cantonne pas aux valeurs {-1,0,1} pour la valeur attribuée à l'inconnu. Si l'on autorise une variation continue entre -1 et 1 pour la valeur Inconnu, on obtient une modélisation continue du risque encouru qui permet au système d'être moins radical à la frontière entre les différentes stratégies. Une fonction permettant de modéliser le risque de façon continue est donnée dans la Figure Stratégie continue.

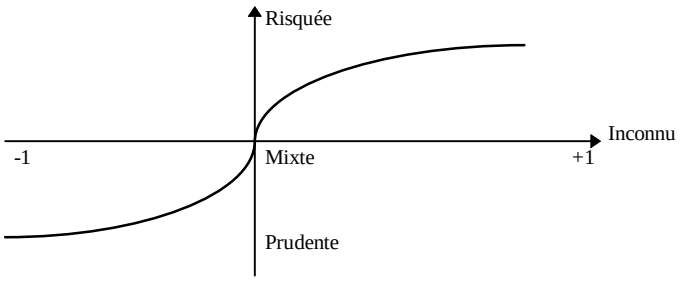


Figure Stratégie continue

2.3.3 Utilisation de la taxonomie

Chaque élément de la taxonomie est associé à une valeur qui dépend de la stratégie choisie. Cette valeur peut être utilisée pour savoir s'il est important de raffiner un jeu. Dans certain cas, il peut être intéressant de ne pas calculer plus avant un jeu que l'on sait sans incidence sur le choix du coup, par exemple lorsque l'importance de l'ensemble des buts qu'un coup atteint est petite et que la valeur du jeu ne peut plus beaucoup varier.

2.3.4 Exemple d'utilisation

Par la suite nous considérerons que Noir est la couleur amie et que Blanc est la couleur ennemie. Les règles du jeu de Go donnent des conditions suffisantes pour qu'une pierre soit retirée du damier. Cette règle du jeu donne naissance à un but intéressant à atteindre qui est le but "**Prendre**". On peut facilement définir pour ce but un état statique "Gagné" : le jeu "Prendre" est atteint si le bloc concerné par le but est retiré du damier. Il est aussi possible de donner des conditions dans lesquelles on a un état statique "Perdu" : ces conditions sont connues par les joueurs de Go comme la règle des deux yeux, le déclenchement de cette règle permet de définir l'accomplissement du but "**Vivre**".

Cependant, en pratique, les règles concernant l'état "Gagné" s'appliquent très rarement dans les mêmes situations que les règles concernant l'état "Perdu". Dans la très grande majorité des cas on a des résultats certains uniquement sur un seul côté du jeu. D'autres buts couramment utilisés par les joueurs de Go sont : le but "**Connecter**" deux blocs pour lequel l'état Gagné est atteint lorsque les deux blocs sont joints, et le but "**Occuper_Intersection**" qui est gagné lorsqu'une pierre amie posée sur l'intersection concernée ne peut pas être prise par l'adversaire.

L'exemple suivant montre une situation où l'on a déterminé les modifications qu'apportaient les coups Noir en i28, i43 et i59 aux jeux que le système a calculés :

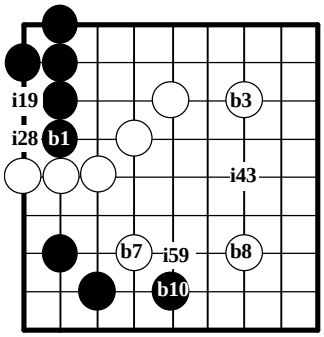


Figure Goban Jeu

L'exemple ci-dessus est tiré d'une partie de Go. On note un résultat de calcul effectué par le système avec un coup, un but, des objets concernés par le but et un jeu auquel on associera une importance. L'importance est une approximation du nombre de points que rapportera l'accomplissement du but à la fin de la partie. Les objets concernés par les buts peuvent être des intersections, des blocs ou des groupes (Cf. chapitre sur la représentation des connaissances).

Gogol obtient sur l'exemple de la Figure Goban Jeu les résultats du Tableau Déductions Jeux suivant :

Coup	But	Objets concernés	Importance	Jeu
Noir en i28	Vivre	b1	23	GI
Noir en i28	Connecter	b1 i19	1	GI
Noir en i28	Connecter	b1 i28	1	GI
Noir en i43	Déconnecter	b3 b8	14	IP
Noir en i43	Connecter	i43 i42	1	GIII
Noir en i43	Connecter	i43 i34	1	GIII
Noir en i43	Connecter	i43 i52	1	GIII
Noir en i43	Connecter	i43 i44	1	GIII
Noir en i59	Déconnecter	b7 b8	6	IP
Noir en i59	Connecter	b10 i59	1	GI
Noir en i59	Connecter	i59 i60	1	GIII
Noir en i59	Connecter	i59 i50	1	GIII

Tableau Déductions Jeux

Pour évaluer un coup on utilise la formule suivante :

$$\text{Valeur (Coup)} = \sum_i (\text{Importance}_i * \text{Valc (Jeu}_i))$$

i parcourant les indices des jeux dans lesquels le coup que l'on note joue.

Par exemple, pour la stratégie mixte :

Valeur (Noir en i43) = 14*1 + 1*½ + 1*½ + 1*½ + 1*½ = 16

On obtient de cette façon le tableau Coups Stratégies qui donne les valeurs des coups pour les différentes stratégies.

Coups\Stratégies	Risquée	Mixte	Prudente
Noir en i28	0	**25**	**50**
Noir en i43	**28**	16	4
Noir en i59	12	8	4

Tableau Coups Stratégies

Gogol évalue que Noir est très en retard, il optera donc pour une stratégie risquée, ce qui lui fera choisir le coup Noir en i43. Si la partie avait été équilibrée ou si Noir avait eu de l'avance, Gogol aurait choisi le coup Noir en i28.

2.4 Conclusion

Après avoir rappelé succinctement des éléments de théorie combinatoire de jeux, j'ai montré comment cette théorie pouvait être appliquée à des jeux complexes **en lui donnant les moyens de représenter l'inconnu**. En effet, cette représentation de l'inconnu permet à mon système non seulement de ne pas prendre en compte des pans trop importants de l'arbre de recherche, mais aussi de définir des jeux complexes de façon simple et plus générale que les représentations précédentes. De plus, cette représentation de l'inconnu donne la possibilité d'établir une taxonomie entre les jeux. Cette taxonomie est très utile pour éviter au programme de faire des calculs inutiles. Enfin, l'attribution de valeurs différentes à l'inconnu définit plusieurs stratégies de gestion du risque.

En utilisant cette formalisation, un système peut prendre en compte son évaluation de la position pour appliquer une stratégie de gestion de l'incertitude dans son choix des coups. Cette théorie diffère des essais précédents pour gérer l'incertitude dans le domaine des jeux [Berliner 1979] [Horacek 1989] [Junghanns 1995] parce qu'elle gère l'incertitude à partir des résultats d'une recherche partielle dans le graphe des états possibles alors que les autres méthodes utilisent des évaluations numériques des états pour orienter leurs recherches dans le graphe. De plus, elle permet une approche orientée vers un grand nombre de connaissances plutôt qu'une approche basée sur une recherche arborescente intensive. L'approche orientée connaissance est nécessaire dans les jeux très complexes comme le jeu de Go car le nombre d'états possibles (10^{172}), le nombre de coups possibles pour chaque position ($\approx$200) et la difficulté d'évaluer une position rendent une approche basée sur une recherche intensive impraticable.

Je pense que cette extension de la théorie combinatoire des jeux à la représentation de l'inconnu est très utile pour représenter des jeux complexes. Elle a de bonnes propriétés comme l'utilisation de la taxonomie pour réduire la recherche ou l'attribution de valeurs à l'inconnu pour gérer le risque. Elle est générale et peut être utilisée pour représenter les connaissances sur l'achèvement de buts dans la plupart des jeux.

2.5 Bibliographie

[Alliot 1994] - J. M. Alliot et T. Schiex. *Intelligence Artificielle et Informatique Théorique*. Cepadues Editions 1994.

[Allis 1994] - L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. Ph. D. Thesis, Vrije Universitat Amsterdam, Maastricht 1994.

[Berlekamp 1982] - E. Berlekamp, J.H. Conway, R. K. Guy. *Winning Ways (for your mathematical plays)*. Academic Press, New York, 1982.

[Berlekamp 1991] - E. Berlekamp. *Introductory overview of mathematical Go endgames*. Proceedings of symposia in applied mathematics - 43 - 1991.

[Berlekamp 1994] - E. Berlekamp, D. Wolfe. *Mathematical Go Endgames - Nightmares for the Professional Go Player*. Ishi Press International - San Jose, London, Tokyo - 1994.

[Berliner 1979] - H. Berliner. *The B* Tree Search Algorithm : A Best-First Proof Procedure*. Artificial Intelligence 12, pp 23-40, North-Holland 1979.

[Bouzy 1995] - B. Bouzy. *Modélisation cognitive du joueur de Go*. Thèse de l'Université Paris 6, 1995.

[Conway 1976] - J. H. Conway. *On Numbers and Games*. Academic Press 1976.

[Horacek 1989] - H. Horacek.*Reasoning with uncertainty in computer chess* Advances in Computer Chess 5, pp 43-63, Elsevier, 1989.

[Junghanns 1995] - A. Junghanns, C. Posthoff, M. Schlosser.*Search with Fuzzy Numbers* International Joint Conference on Fuzzy Systems, pp 979-986, Yokohama, 1995.

[Müller 1995] - Martin Müller. *Computer Go as a Sum of Local Games : An Application of Combinatorial Game Theory*. Thèse du Swiss Federal Institute of Technology Zürich, 1995.

[Wolfe 1991] - D. Wolfe, Mathematics of Go : Chilling Corridors, Dissertation, Université de Californie à Berkeley, Berkeley, 1991.

Table des Matières du Chapitre 3

3 REPRÉSENTATION DES CONNAISSANCES 30

3.1 Patterns géométriques 31
3.1.1 Représentation des patterns géométriques 31
3.1.2 Les concepts utilisés 31
3.1.3 Les règles 32
3.1.4 Limites des représentations à base de patterns géométriques 33

3.2 Logique des prédicats 36

3.3 Représentation des métaconnaissances 39

3.4 Une structure des connaissances dans le domaine des jeux 44
3.4.1 Trois niveaux de connaissance 44
3.4.2 Le passage entre deux niveaux 44

3.5 Les méthodes de sélection 45
3.5.1 Une définition des méthodes de sélection 45
3.5.2 Application dans le domaine des jeux 49

3.6 Conclusion 51

3.7 Bibliographie 52

3 Représentation des Connaissances

Les meilleurs programmes de Go utilisent pour la plupart une représentation des connaissances tactiques basée sur des patterns géométriques. Gogol, dans ses premières versions, utilisait une représentation des connaissances à base de patterns géométriques. Il a ensuite évolué vers une représentation des connaissances à base de logique de prédicats et de métaprédicats pour les métaconnaissances. Actuellement, les connaissances tactiques aussi bien que les connaissances stratégiques de Gogol sont écrites en logique des prédicats. Il compile ses connaissances sous forme logique en programmes C++ pour les appliquer plus rapidement. Il n'utilise plus de patterns géométriques mais des programmes C++. Les règles de transition du jeu de Go et les règles stratégiques de Gogol sont données dans les Annexes.

Je montre dans une première section comment Gogol représente les règles à base de patterns géométriques. Puis, j'explique pourquoi la représentation à base de patterns géométriques est une représentation efficace mais trop spécialisée de la connaissance. Je présente donc dans une deuxième section une représentation plus générale de la connaissance à base de logique des prédicats. Je décris dans la troisième section la façon dont mon programme manipule ses connaissances logiques à l'aide de métaconnaissances, elles aussi représentées sous une forme logique. Je donne dans la section suivante une structure des connaissances dans le domaine des jeux. Les connaissances peuvent aussi être représentées en utilisant des méthodes de sélection [Nigro 1996] que je présente dans la cinquième section et je montrerai dans le chapitre sur l'explication comment les méthodes de sélection sont utilisées pour expliquer les coups aux utilisateurs du système. Pour finir, je conclurai sur l'intérêt des différentes approches de la représentation de la connaissance.

3.1 Patterns géométriques

Nous allons montrer que les patterns géométriques sont une façon très spécialisée de représenter la connaissance du Go. L'avantage de cette représentation est son efficacité pour matcher les règles, son désavantage est son manque de généralité et de concision. Les règles basées sur des patterns géométriques ont été utilisées dans Gogol jusqu'en Octobre 1995. Elles ont été ensuite abandonnées au profit d'une représentation basée sur la logique des prédicats et des concepts plus généraux.

3.1.1 Représentation des patterns géométriques

Dans "Knowledge Representation in The Many Faces of Go" [Fotland 1993], l'utilisation des règles est définie comme suit :

"Chaque pattern est 8x8, où chaque point est spécifié comme étant Noir, Blanc, Vide, peu importe, Blanc ou Vide, Noir ou Vide, Noir ou Blanc. Chaque pattern possède un ensemble d'attributs qui doivent aussi être vérifiés. Par exemple : 'la pierre en (4,2) doit être vivante', ou 'la pierre en (2,5) doit avoir au moins 2 libertés'. Chaque pattern est accompagné d'un arbre de coups qui est utilisé pour une recherche sur tout le damier.

Un pattern est enregistré comme trois tableaux de 8 octets, un pour les pierres Noires, un pour les pierres blanches, et un pour les intersections vides... Après avoir matché un pattern, je vérifie que les attributs matchent aussi... j'utilise une fonction de hachage sur 8 bits pour réduire le nombre de patterns examinés... Le programme passe à peu près 1.5% de son temps à matcher des patterns."

La technique de hachage est la même que celle utilisée dans Goliath [Boon 1990]. Ego [Wilcox 1995] quant à lui ne tient pas à jour tous les patterns qui matchent automatiquement mais ne cherche à matcher que ceux correspondant à un but donné. Les patterns dans Ego sont matchés par ordre de priorité décroissante et seule la plus grande priorité est retournée pour une intersection. Les patterns de priorité plus petite ne sont pas matchés, à moins que la classe du pattern ne requière le matchage de tous les patterns.

Gogol utilise toutes ces techniques à la fois pour matcher les patterns géométriques. Il utilise 8 entiers codés sur 32 bits pour ses patterns qui peuvent avoir une taille allant de 1x1 à 8x8. Il utilise 64 bits pour les pierres noires, 64 bits pour les pierres blanches, 64 bits pour les intersections vides et 64 bits pour représenter les bords du damier. Il matche les patterns en fonction du but à atteindre, du type de la pierre concernée et d'une fonction de hachage qui prend des valeurs entre 0 et 256. La première version de Gogol passait la majeure partie de son temps à matcher des patterns et à vérifier les conditions associées aux patterns.

3.1.2 Les concepts utilisés

Un bon concept est un concept rapide à calculer et qui, associé à d'autres, permet une bonne discrimination entre bons et mauvais exemples.

Définition du concept chaîne :

Deux pierres A et B de la même couleur font partie d'une même chaîne si elles sont reliées dans le graphe associé au damier par des pierres de la même couleur.

A chaque chaîne on peut associer un nombre de pierres contenues dans la chaîne.

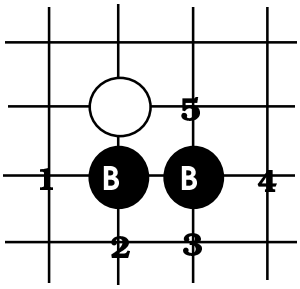
Définition du concept chaîne adjacente :

Une chaîne est adjacente d'une autre chaîne de couleur opposée si une pierre d'une chaîne est voisine d'une pierre de l'autre chaîne.

Définition du concept liberté d'une chaîne :

Les libertés d'une chaîne sont les intersections vides voisines de cette chaîne.

Exemple :



La chaîne B a 5 libertés et 2 pierres

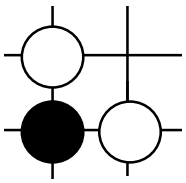
3.1.3 Les règles

Elles ont pour prémisse une configuration particulière d'une partie du Goban que j'appellerai par la suite **pattern**. A chaque pierre de ce pattern peut être associée une information additionnelle sur le nombre minimum et le nombre maximum de libertés que peut prendre la chaîne dont fait partie la pierre. Elles ont aussi pour prémisse un but à atteindre.

Elles ont en conclusion l'état du jeu correspondant au but, ainsi que si besoin est, les coups qui permettent de changer l'état du jeu.

Prémisses possibles et non exclusives:

- Un pattern :



- Un arbre de buts. Le type des chaînes est associé à chaque but de l'arbre.

  Exemple : Connecter bloc1 0 0 bloc2 1 1

  Ce qui signifie que le but que l'on cherche à atteindre est la connexion de la chaîne de la pierre de coordonnées 0 0 et de la chaîne de la pierre de coordonnées 1 1, ce qui correspond dans ce cas aux chaînes des deux pierres blanches du pattern. Les types possibles pour les chaînes sont : bloc1 (chaîne principale 1), bloc2 (chaîne principale 2), bloc_adjacent1 (chaîne adjacente à la chaîne principale 1), bloc_adjacent2 (chaîne adjacente à la chaîne principale 2), bloc_adjacent_adjacent1 (chaîne adjacente à un chaîne adjacente à la chaîne principale 1) et bloc_adjacent_adjacent2 (chaîne adjacente à une chaîne adjacente à la chaîne principale 2).

- Des propriétés sur les libertés.

  Exemple : min_libertés 0 0 3

  Ce qui signifie que la chaîne de la pierre blanche en 0 0 dans le pattern doit avoir au moins trois libertés.

Conclusions possibles :

- Un jeu.

Exemple : Jeu GI
Coup Blanc 1 0

Ce qui signifie que le jeu de connecter les deux chaînes blanches est GI. Un jeu GI est généralement associé à un coup qui permet de rendre ce jeu G. Ce qui signifie que jouer une pierre blanche sur l'intersection de coordonnées 1 0 du pattern permet de connecter les deux chaînes blanches.

3.1.4 Limites des représentations à base de patterns géométriques

La première limitation de la représentation à base de patterns géométriques est la trop grande spécialisation d'un pattern dans l'espace.

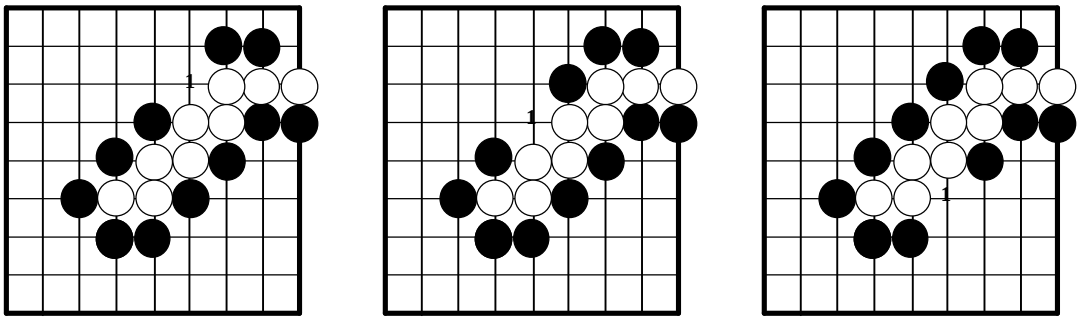
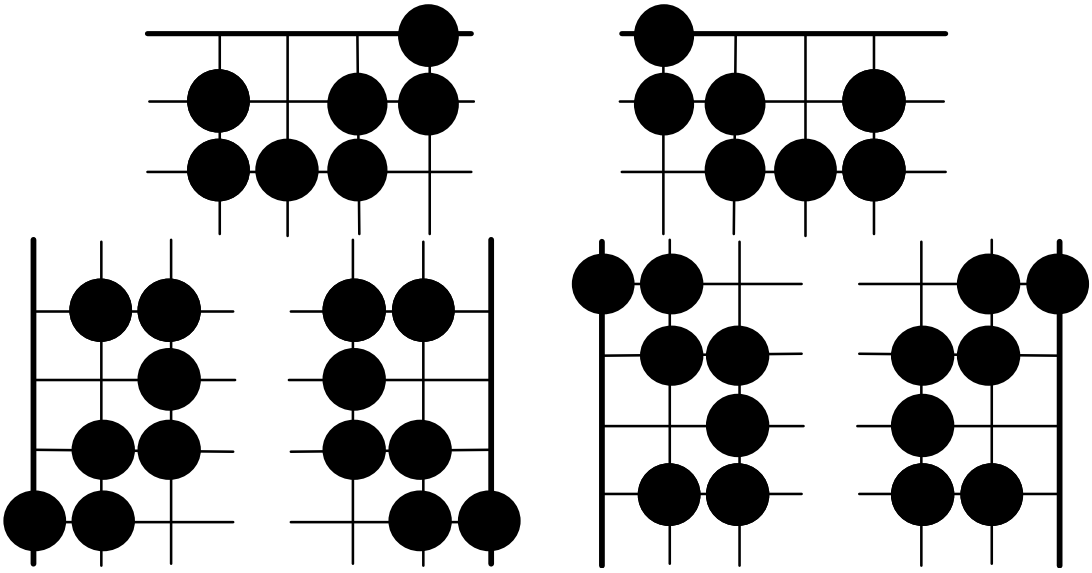


Figure Représentation Shichos

Dans les trois damiers de la Figure Représentation Shichos, un coup Noir en 1 prend les pierres blanches en shicho. Ces trois damiers sont représentés par trois patterns géométriques différents. Il serait plus intéressant d'avoir une représentation des connaissances permettant de créer une seule règle s'appliquant à ces trois damiers, concluant toujours sur le coup Noir en 1 pour prendre le bloc de pierres blanches en shicho.

La seconde limitation de la représentation à base de patterns géométriques est la trop grande spécialisation d'un pattern.



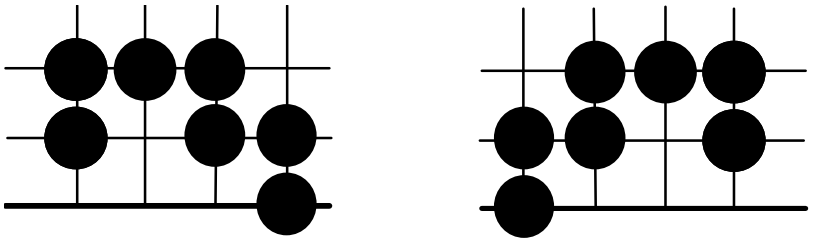


Figure Représentation Formes

Les 8 patterns de la Figure Représentation Formes correspondent en fait à une seule définition. Si on prend aussi en compte ces mêmes patterns avec les couleurs inversées, on obtient alors 16 patterns différents pour la même définition. Ces symétries de base sont incorporées dans la plupart des programmes de Go comportant des patterns géométriques.

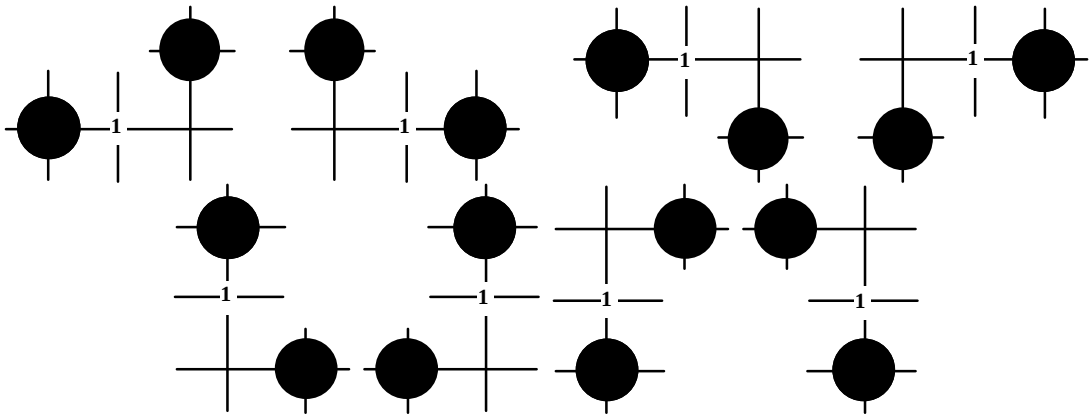
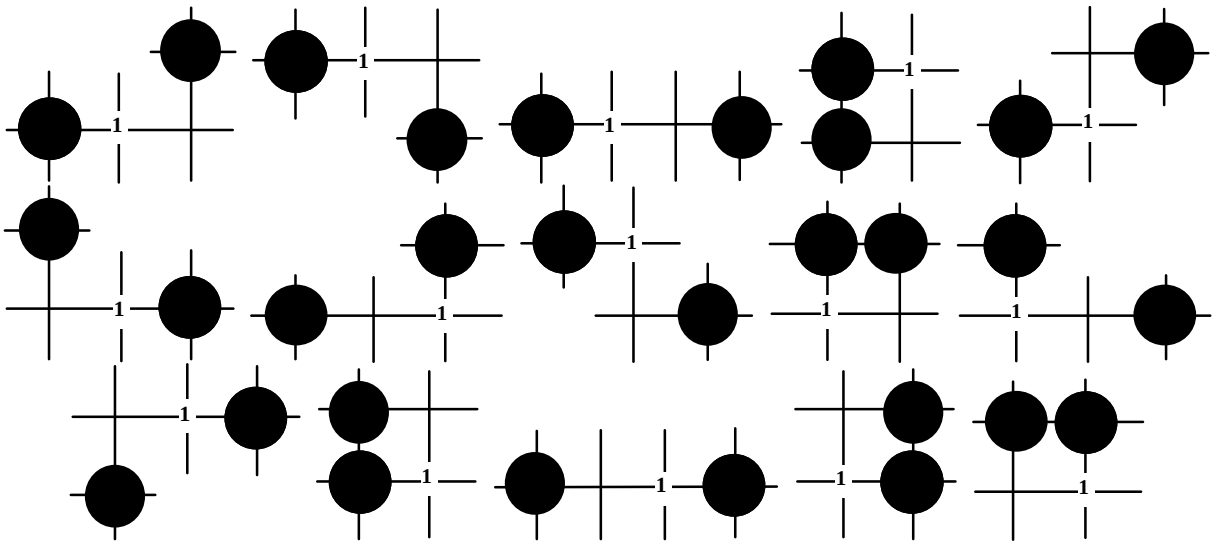


Figure Représentation Symétries classiques

La Figure Représentation Symétries classiques montre les 8 variations classiques autour d'une forme de base à laquelle on peut ajouter 8 autres formes par inversion des couleurs. Cette forme donne une menace de connecter les deux pierres noires en jouant un coup noir en 1. Toutefois il est parfois intéressant d'utiliser des définitions plus générale que celles permise par la représentation géométrique. Ainsi, si on se place au niveau du concept de voisinage, on voit que la forme de la Figure Représentation Symétries classiques peut être définie comme une pierre Noire voisine d'une intersection vide sur laquelle on peut jouer, elle-même voisine d'une intersection Vide, elle-même voisine d'une intersection Noire. En utilisant le concept Voisine pour donner une définition au lieu d'un pattern géométrique, on obtient les formes de la Figure Représentation Voisines.



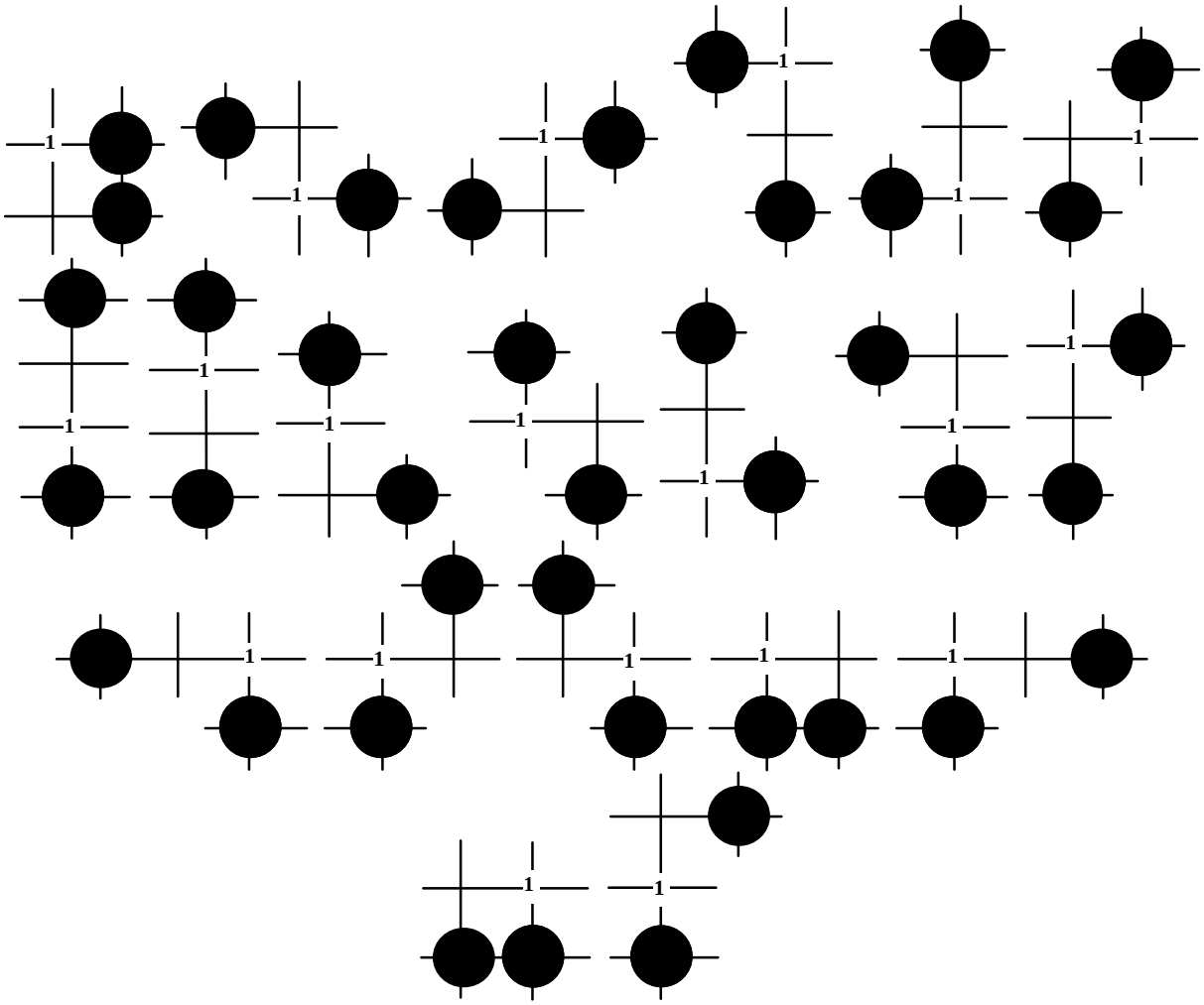


Figure Représentation Voisines

On obtient 36 formes qui correspondent à la même définition. Utiliser des concepts qui permettent de décrire les règles du jeu de façon générale permet au programme d'apprendre des règles générales. L'utilisation du concept de Voisine à la place des concepts de Au_dessus, En_dessous, A_gauche et A_droite autorise une description plus générale des règles. Ainsi une règle apprise par le programme avec des concepts généraux permet de représenter plusieurs règles composées uniquement de patterns géométriques. L'expérience m'a montré que plus les patterns sont complexes, plus la représentation avec des concepts généraux est utile et permet de créer des règles générales.

```
( premisses (
    present ( Couleur ( ?c ) )
    present ( Couleur_bloc_avant ( ?b ?c ) )
    present ( Liberte_bloc_avant ( ?i ?b ) )
    present ( Liberte_bloc_avant ( ?i ?b1 ) )
    present ( Couleur_bloc_avant ( ?b1 ?c ) )
    blocs_differents ( ?b ?b1 )
    present ( Liberte_bloc_avant ( ?i1 ?b ) )
    intersections_differentes ( ?i ?i1)
    present ( Liberte_bloc_avant ( ?i1 ?b1 ) )
  )
 conclusions (
    ajoute ( Coup_jeu_binaire_bloc_avant ( ?c Connecter ?b ?b1 G ) )
    ajoute ( Coup_jeu_binaire_bloc_avant ( ?c Connecter ?b1 ?b G ) ) ) )
```

Tableau Représentation Connexion

Le tableau Représentation Connexion donne une règle qui signifie que si un bloc a deux libertés communes avec un autre bloc de la même couleur, alors les deux blocs sont connectés. Si on utilise une représentation à base de patterns géométriques, on doit définir plusieurs patterns pour avoir l'équivalent de cette règle. La figure Représentation Connexion Géométrique donne quelques uns de ces patterns

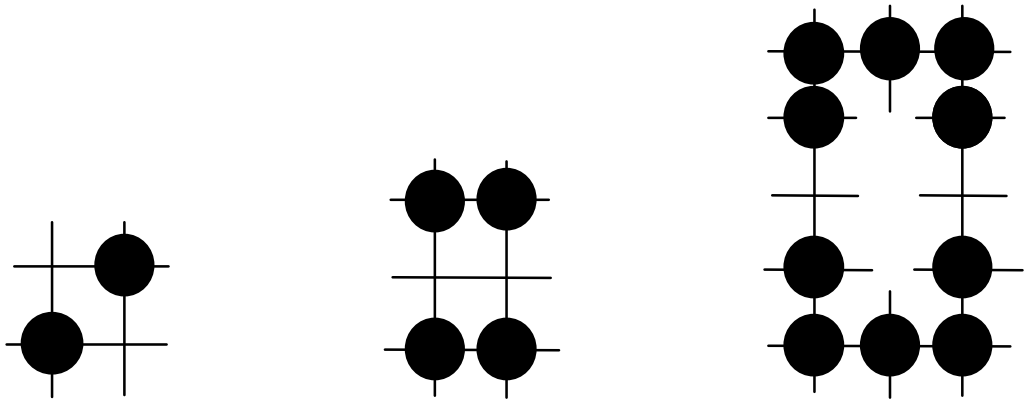


Figure Représentation Connexion Géométrique

3.2 Logique des prédicats

Pour remédier aux limitations de la représentation à base de patterns géométriques, j'ai choisi la solution qui consiste à représenter les règles en logique des prédicats. La logique des prédicats permet aux règles de contenir des variables universellement quantifiées. Le langage Prolog est un langage de programmation basé sur la logique des prédicats. L'utilisation de variables permet une représentation beaucoup plus concise de la connaissance. Dans le Tableau Représentation Voisines, on voit que les 36 patterns géométriques peuvent être représentés par une seule liste de prédicats. Par convention, les variables commencent par un '?'. Chaque variable peut avoir de multiples instanciations. Le métaprédicat présent est utilisé dans toutes les règles, il permet de tester la présence d'un ou de plusieurs faits dans la base de faits.

Présent (Couleur_intersection (?i Noir))
présent (Voisine (?i ?i1))
présent (Couleur_intersection (?i1 Blanc))
présent (Voisine (?i1 ?i2))
présent (Couleur_intersection (?i2 Vide))
présent (Voisine (?i2 ?i3))
présent (Couleur_intersection (?i3 Noir))

Tableau Représentation Voisine

Un ensemble de prédicats s'instancie sur une base de faits. Une base de faits est un ensemble de prédicats ne contenant pas de variables. Dans mon système, les informations sur la position de jeu (le Goban ou damier) sont représentées par une base de faits. Les règles s'appliquant sur cette base de faits sont représentées en utilisant des prédicats contenant des variables.

La figure Représentation Goban est représentée sous forme de base de faits dans le tableau Représentation Goban.

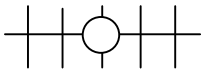


Figure Représentation Goban

Chaque intersection est représentée par une constante : i1 est l'intersection la plus à gauche sur la figure, i5 est celle la plus à droite. Le bloc Blanc est aussi représenté par une constante : b1. La couleur amie est Noir, la couleur ennemie est blanc. La couleur amie est représentée par le symbole '@', la couleur ennemie est représentée par le symbole 'O'.

Couleur (@)	Nombre_voisines (i3 2)
Couleur (O)	Couleur_intersection (i3 O)
Couleur_opposee (O @)	Bloc (i3 b1)
Couleur_opposee (@ O)	Couleur_Bloc (b1 O)
Voisine (i1 i2)	Nombre_libertés (b1 2)
Nombre_voisines (i1 1)	Nombre_pierres (b1 1)
Couleur_intersection (i1 +)	Voisine(i4 i3)
Voisine (i2 i1)	Voisine(i4 i5)
Voisine (i2 i3)	Nombre_voisines (i4 2)
Nombre_voisines (i2 2)	Couleur_intersection (i4 +)
Couleur_intersection (i2 +)	Voisine (i5 i4)
Voisine (i3 i2)	Nombre_voisines (i5 1)
Voisine (i3 i4)	Couleur_intersection (i5 +)

Tableau Représentation Goban

Le tableau Représentation Liste Types donne la liste des types utilisés.

Intersection	Prédicat
Couleur	Règle
Entier	Variable
Reel	Constante
Jeu	Liste
Bloc	Chaîne
But	Bool
Type	

Tableau Représentation Liste Types

On peut noter la présence d'un méta-type : le type Type. Des variables de type Type contiennent un type particulier.

Une variable de type Variable contient une autre variable. C'est une méta-variable.

Le type Liste est un type générique, les listes peuvent être composées d'objets de n'importe quel type. Elle peuvent aussi comporter des types hétérogènes.

Les types spécifiques au jeu de Go sont les types Intersection et Bloc. Les types Couleur, Jeu et But sont utilisés dans tous les jeux.

Bool Couleur (Couleur)
Bool Couleur_opposees (Couleur , Couleur)
Bool Couleurs_differentes (Couleur , Couleur)
Bool Voisine (Intersection , Intersection)
Bool Voisine_differente (Intersection , Intersection , Intersection)
Bool Voisine_differente_de_2 (Intersection , Intersection , Intersection , Intersection)
Bool Voisine_differente_de_3 (Intersection , Intersection , Intersection , Intersection , Intersection)
Bool Nombre_voisines (Intersection , Entier)
Bool Coup (Intersection)
Bool Tour_avant (Couleur)
Bool Nombre_intersections_prises_avant (Entier)
Bool Intersections_prises_avant (Intersection)
Bool Coup_legal_avant (Intersection , Couleur)
Bool Suicide_possible_avant (Intersection , Bloc)
Bool Couleur_intersection_modifiee (Intersection)
Bool Non_couleur_intersection_modifiee (Intersection)
Bool Couleur_intersection_avant (Intersection , Couleur)
Bool Non_couleur_intersection_avant (Intersection , Couleur)
Bool Nombre_voisines_couleur_avant (Intersection , Couleur , Entier)
Bool Nouveau_bloc (Bloc)
Bool Bloc_ote (Bloc)
Bool Non_bloc_ote (Bloc)
Bool Fusion_blocs (Bloc , Bloc)
Bool Non_fusion_blocs (Bloc , Bloc)
Bool Nombre_blocs_otes (Entier)
Bool Bloc_avant (Intersection , Bloc)
Bool Non_bloc_avant (Intersection , Bloc)
Bool Couleur_bloc_avant (Bloc , Couleur)
Bool Nombre_libertes_bloc_avant (Bloc , Entier)
Bool Nombre_libertes_communes_avant (Bloc , Bloc , Entier)
Bool Nombre_libertes_communes_au_3_avant (Bloc , Bloc , Bloc , Entier)
Bool Nombre_libertes_communes_au_4_avant (Bloc , Bloc , Bloc , Bloc , Entier)
Bool Liberte_bloc_avant (Intersection , Bloc)
Bool Non_liberte_bloc_avant (Intersection , Bloc)
Bool Nombre_pierres_bloc_avant (Bloc , Entier)
Bool Pierre_voisine_bloc_avant (Intersection , Bloc)
Bool Non_pierre_voisine_bloc_avant (Intersection , Bloc)
Bool Nombre_pierres_bloc_voisines_avant (Bloc , Bloc , Entier)
Bool Nombre_pierres_bloc_voisines_communes_avant (Bloc , Bloc , Bloc , Entier)
Bool Nombre_pierres_bloc_voisines_communes_au_3_avant (Bloc , Bloc , Bloc , Bloc , Entier)
Bool Pas_de_blocs_voisins_bloc_avant (Bloc)
Bool Blocs_voisins_avant (Bloc , Bloc)
Bool Nombre_minimum_libertes_blocs_voisins_bloc_avant (Bloc , Entier)
Bool Nombre_Blocs_voisins_avant (Intersection , Entier)
Bool Nombre_Blocs_voisins_couleur_avant (Intersection , Couleur , Entier)
Bool fait_modifie (Predicat)
Bool Oeil_apres (Intersection , Bloc)

Tableau Représentation Liste Prédicats

Le tableau Représentation Liste Prédicats donne la liste des prédicats utilisés pour jouer et apprendre à jouer au Go. Ne sont donnés dans ce tableau que les prédicats décrivant la situation avant un coup. Chaque prédicat qui se termine par le mot ‘avant’ a un correspondant qui s’obtient en remplaçant ‘avant’ par ‘après’. De plus, les prédicats concernant les jeux combinatoires sont donnés dans le tableau Représentation Prédicats Jeux.

Bool Jeu_intersection_avant (Couleur , But , Intersection , Jeu)
Bool Jeu_intersection_regresse_avant (Couleur , But , Intersection , Jeu)
Bool Coup_jeu_intersection_avant (Couleur , But , Intersection , Intersection , Jeu)
Bool Coup_jeu_intersection_regresse_avant (Couleur , But , Intersection , Intersection , Jeu)
Bool Coup_force_jeu_intersection_avant (Couleur , But , Intersection , Intersection , Jeu)
Bool Coup_force_jeu_intersection_regresse_avant (Couleur , But , Intersection , Intersection , Jeu)
Bool Jeu_bloc_avant (Couleur , But , Bloc , Jeu)
Bool Jeu_bloc_regresse_avant (Couleur , But , Bloc , Jeu)
Bool Coup_jeu_bloc_avant (Couleur , But , Bloc , Intersection , Jeu)
Bool Coup_jeu_bloc_regresse_avant (Couleur , But , Bloc , Intersection , Jeu)
Bool Coup_force_jeu_bloc_avant (Couleur , But , Bloc , Intersection , Jeu)
Bool Coup_force_jeu_bloc_regresse_avant (Couleur , But , Bloc , Intersection , Jeu)
Bool Jeu_binaire_bloc_avant (Couleur , But , Bloc , Bloc , Jeu)
Bool Jeu_binaire_bloc_regresse_avant (Couleur , But , Bloc , Bloc , Jeu)
Bool Coup_jeu_binaire_bloc_avant (Couleur , But , Bloc , Bloc , Intersection , Jeu)
Bool Coup_jeu_binaire_bloc_regresse_avant (Couleur , But , Bloc , Bloc , Intersection , Jeu)
Bool Coup_force_jeu_binaire_bloc_avant (Couleur , But , Bloc , Bloc , Intersection , Jeu)
Bool Deux_coups_forces_jeu_binaire_bloc_avant (Couleur , But , Bloc , Bloc , Intersection , Intersection , Jeu)
Bool Trois_coups_forces_jeu_binaire_bloc_avant (Couleur , But , Bloc , Bloc , Intersection , Intersection , Intersection , Jeu)
Bool Coup_force_jeu_binaire_bloc_regresse_avant (Couleur , But , Bloc , Bloc , Intersection , Jeu)
Bool Jeu_binaire_bloc_intersection_avant (Couleur , But , Bloc , Intersection , Jeu)
Bool Jeu_binaire_bloc_intersection_regresse_avant (Couleur , But , Bloc , Intersection , Jeu)
Bool Coup_jeu_binaire_bloc_intersection_avant (Couleur , But , Bloc , Intersection , Intersection , Jeu)
Bool Coup_jeu_binaire_bloc_intersection_regresse_avant (Couleur , But , Bloc , Intersection , Intersection , Jeu)
Bool Coup_force_jeu_binaire_bloc_intersection_avant (Couleur , But , Bloc , Intersection , Intersection , Jeu)
Bool Coup_force_jeu_binaire_bloc_intersection_regresse_avant (Couleur , But , Bloc , Intersection , Intersection , Jeu)
Bool Deux_coups_forces_jeu_binaire_bloc_intersection_avant (Couleur , But , Bloc , Intersection , Intersection , Intersection , Jeu)
Bool Deux_coups_forces_jeu_binaire_bloc_intersection_regresse_avant (Couleur , But , Bloc , Intersection , Intersection , Intersection , Jeu)
Bool Trois_coups_forces_jeu_binaire_bloc_intersection_regresse_avant (Couleur , But , Bloc , Intersection , Intersection , Intersection , Intersection , Jeu)
Bool Trois_coups_forces_jeu_binaire_bloc_intersection_regresse_avant (Couleur , But , Bloc , Intersection , Intersection , Intersection , Intersection , Jeu)
Bool Jeu_binaire_intersection_intersection_avant (Couleur , But , Intersection , Intersection , Jeu)
Bool Jeu_binaire_intersection_intersection_regresse_avant (Couleur , But , Intersection , Intersection , Jeu)
Bool Coup_jeu_binaire_intersection_intersection_avant (Couleur , But , Intersection , Intersection , Intersection , Jeu)
Bool Coup_jeu_binaire_intersection_intersection_regresse_avant (Couleur , But , Intersection , Intersection , Intersection , Jeu)
Bool Coup_force_jeu_binaire_intersection_intersection_avant (Couleur , But , Intersection , Intersection , Intersection , Jeu)
Bool Coup_force_jeu_binaire_intersection_intersection_regresse_avant (Couleur , But , Intersection , Intersection , Intersection , Jeu)

Tableau Représentation Prédicats Jeux

Chaque prédicat de cette liste a son correspondant comme prédicat après le coup. La sémantique des prédicats est la suivante :

Par exemple, le prédicat ‘Coup_jeu_binaire_bloc_avant (Couleur , But , Bloc , Bloc , Intersection , Jeu)’ est utilisé pour représenter qu’un coup de la couleur Couleur sur l’intersection Intersection permet de jouer au jeu Jeu associé au but But qui concerne les deux blocs Bloc.

3.3 Représentation des métaconnaissances

Pour raisonner sur les connaissances représentées sous forme de prédicats et pour les manipuler, on utilise des métaconnaissances [Pitrat 1990]. Dans mon système, les métaconnaissances sont représentées en utilisant des métaprédicats et des métavariables. Les métaprédicats sont des prédicats qui ont pour argument d’autres prédicats. Les métavariables sont des variables qui ont pour instanciation d’autres variables.

Les métaprédicats les plus utilisés sont les métaprédicats **‘présent’, ‘ajoute’, ‘absent’** et **‘enlève’**. Ils ont pour argument un prédicat, et permettent respectivement de vérifier qu’un fait est présent dans la base de fait, d’ajouter un fait, de vérifier qu’un fait est absent et d’enlever un fait. Toutes les règles utilisent ‘présent’ pour vérifier qu’un fait est présent dans la base de faits. Presque toutes les règles utilise ‘ajoute’ pour déduire de nouveaux faits. Le métaprédicat ‘absent’ est généralement utilisé dans les règles apprise par Introspect pour vérifier que la conclusion d’une règle n’a pas déjà été déduite et qu’il est utile de continuer à filtrer les prémisse suivantes de la règle. Le métaprédicat enlève est très

peu utilisé, il sert à ôter de la base de fait les faits qui sont plus généraux qu'un autre fait déjà déduit (par exemple on ôte un fait sur un jeu GI si on a déduit un fait portant sur le même but avec un jeu G).

Le métaprédicat 'présent' est utilisé dans les règles pour vérifier qu'un fait est présent dans la base de faits. Il ne faut pas le confondre avec le métaprédicat 'condition' qui vérifie qu'une prémisse est présente dans une règle.

On peut trouver des exemples d'utilisation de métavariables dans les métarègles du tableau Représentation Métarègle Simplification et du tableau Représentation Métarègle Ordonne.

Les règles sont représentées par une liste de prémisses et une liste de conclusions.

Les prémisses ne comportent pas seulement des vérifications à faire sur la base de faits. Elles peuvent aussi porter sur des appels de fonctions.

Entier addition (Entier , Entier)
Entier soustraction (Entier , Entier)
Entier multiplication (Entier , Entier)
Entier division (Entier , Entier)
Bool different (Entier , Entier)
Bool superieur (Entier , Entier)
Bool intersections_differentes (Intersection , Intersection)
Bool egal (Entier , Entier)
Bool blocs_differents (Bloc , Bloc)
Bool superieur_reel (Reel , Reel)

Tableau Représentation Liste Fonctions

Le tableau Représentation Liste Fonctions donne la liste actuelle des fonctions utilisables.

On peut noter qu'il y a autant de fonctions différentes que de types. Ceci est dû au processus de généralisation que je décris plus loin et qui a besoin du type des arguments pour pouvoir généraliser correctement. Les entiers ne sont pas généralisés de la même façon que les autres types car deux entiers peuvent avoir la même instanciation sans pour autant être issus de la même variable. Par contre, pour les types Bloc et Intersection, les généralisations sont simplifiées pour éviter une explosion combinatoire dans la généralisation. De plus, ce typage statique permet de repérer facilement certaines erreurs qui peuvent être faites lors de l'écriture des règles de la théorie du domaine.

La fonction 'égal (Entier , Entier)' peut être utilisée de deux manières. Si l'un des arguments est une variable, c'est une affectation et elle renvoie toujours Vrai. Si les deux arguments sont des constantes ou des variables déjà instanciées, c'est un test et la suite du filtrage dépend du résultat du test.

La fonction 'égal (Entier , Entier)' est la seule qui puisse instancier une variable. Les autres fonctions ne peuvent être appelées qu'avec des variables instanciées ou des constantes. Le choix de représenter le test intersections_différentes par une fonction se justifie quand on sait qu'il y a 361 intersections sur un Goban. Représenter l'information que deux intersections sont différentes avec une base de fait nécessiterait 361*361=130321 faits dans la base, ce qui prendrait beaucoup de mémoire inutilement.

Ces fonctions peuvent prendre en paramètres d'autres fonctions. Ce qui permet d'écrire les fonctions de façon concise. Par exemple, on peut écrire :

égal (?n addition (addition (soustraction (?n2 1) ?n1) ?n3))

```
(
 prémisses
   ( présent ( Couleur_intersection_avant ( ?i1 + ) )
     présent ( Voisine ( ?i2 ?i1 ) )
     présent ( Liberté_avant ( ?i2 ?b ) )
     présent ( Couleur_bloc_avant ( ?b ?c ) )
     présent ( Couleur_opposée ( ?c ?c1 ) )
     présent ( Nombre_libertés ( ?b 2 ) )
     présent ( Liberté_avant ( ?i4 ?b ) )
     intersection_différentes ( ?i4 ?i2 ) )
     présent ( Voisine ( ?i4 ?i5 ) )
     intersection_différentes ( ?i5 ?i2 ) )
     présent ( Couleur_intersection_avant ( ?i5 + ) )
   )
conclusions
 ( ajoute ( Coup_jeu_bloc_avant ( ?c1 Prendre ?b ?i3 GIII ) ) )
)
```

Tableau Représentation Règle

Le tableau Représentation Règle montre une règle qui s'applique sur la base de faits du tableau Représentation Goban. Les deux jeux possibles d'instanciation pour les variables de cette règle sont :

- ?i1=i1, ?i2=i2, ?b=b1, ?c=O, ?c1=@, ?i4=i4, ?i5=i5.

- ?i1=i5, ?i2=i4, ?b=b1, ?c=O, ?c1=@, ?i4=i2, ?i5=i1.

La semi-unification de cette règle avec la base de faits ajoutera donc les faits suivants :

- Coup_jeu_bloc_avant (@ Prendre b1 i2 GIII)

- Coup_jeu_bloc_avant (@ Prendre b1 i4 GIII)

Ces deux faits expriment que les deux coups noirs sur les libertés de la pierre blanche sont des menaces de la prendre.

Mon système utilise d'autres métaprédicats que présent, ajoute, absent et enlève. Le tableau Représentation Liste Métaprédicats donne la liste des métaprédicats actuellement utilisés. Donner une description des actions effectuées par chacun des métaprédicats serait un peu long. Aussi, je me contenterai d'en décrire quelques uns. Par la suite, je définirai les métaprédicats au fur et à mesure que le besoin s'en fera sentir.

Le métaprédicat 'Règle (Règle)' instancie dans la variable en paramètre toutes les règles qui sont contenues dans sa base de règles active. Le choix de la base de règles active est faite dans le programme.

Le métaprédicat 'ôte_doublons (Règle)' ôte tous les prédicats identiques dans les prémisses d'une règle. Il est utilisé pour ôter les tests inutiles qui sont faits par les règles créées par le système.

Bool présent (Predicat)
Bool absent (Predicat)
Bool ajoute (Predicat)
Bool enleve (Predicat)
Bool regle (Regle)
Bool ote_doublons (Regle)
Bool condition (Regle , Predicat)
Bool ordre_condition (Regle , Predicat , Entier)
Bool enleve_metapredicat (Predicat , Predicat)
Bool variable_condition (Regle , Variable)
Bool variable_absente_conclusion (Regle , Variable)
Bool liste_conditions_contenant_variable (Regle , Variable , Liste)
Bool variable_presente_dans_liste (Variable , Liste)
Bool variable_presente_dans_liste_predicats (Variable , Liste)
Bool ajoute_substitution (Liste , Variable , Variable)
Bool tailles_listes_egales (Liste , Liste)
Bool liste_plus_generale_substitution (Liste , Liste , Liste)
Bool action (Regle , Predicat)
Bool variables_differentes (Variable , Variable)
Bool meme_type (Variable , Variable)
Bool liste_conditions_presentes (Regle , Liste)
Bool ajoute_liste (Liste , Liste)
Bool conditions_plus_generales (Regle , Regle)
Bool remplace_variable (Regle , Variable , Variable)
Bool remplace_par_constante (Regle , Variable , Variable)
Bool insere_condition_avant (Regle , Predicat , Predicat)
Bool ote_condition (Regle , Predicat)
Bool ajoute_condition (Regle , Predicat)
Bool ajoute_conclusion (Regle , Predicat)
Bool ote_liste_conditions (Regle , Liste)
Bool ote_regle (Regle)
Bool fermeture_transitive_conditions_contenant_variable (Regle , Variable , Liste)
Bool taille_liste (Liste , Entier)
Bool liste_differentes (Liste , Liste)
Bool liste_variables_conclusion (Regle , Liste)
Bool liste_variables_dans_liste_predicats (Liste , Liste)
Bool liste_elements_premiere_absents_dans_deuxieme (Liste , Liste , Liste)
Bool liste_plus_generale (Liste , Liste , Liste)
Bool liste_variables_sources (Liste , Liste)
Bool liste_variables_cibles (Liste , Liste)
Bool liste_conditions_contenant_liste_variables (Regle , Liste , Liste)
Bool variable_absente_dans_liste_predicats (Variable , Liste)
Bool ote_liste (Liste , Liste)
Bool intersection_vide (Liste , Liste)
Bool constante (Variable)
Bool regle_instanciee (Regle)
Bool liste_regles_instanciees_concluant_sur (Liste , Predicat)
Bool intersection_liste_conditions_regles (Liste , Liste)
Bool liste_coups_modifiant_conditions (Liste , Liste , Liste)
Bool ajoute_conditions_regle_instanciee (Liste)
Bool ajoute_conditions_liste_regles_a_regle_instanciee (Liste)
Bool nombre_de_coups_modifiant (Regle , Predicat , Entier)
Bool ajoute_coup_modifiant (Intersection , Couleur)
Bool ajoute_raison_coup_modifiant (Predicat)
Bool oppose_apres (Predicat , Predicat)
Bool fait_present (Predicat)
Bool instanciee (Variable)
Bool non_instanciee (Variable)
Bool priorite (Regle , Predicat , Reel)
Bool affecte_priorite (Regle , Predicat , Reel)
Bool nombre_conclusions (Regle , Entier)
Bool remplace_par_couleur (Regle , Variable , Couleur)

Tableau Représentation Liste Métaprédicats

```
// Si 2 intersections sont voisines, elles sont différentes

( nom      (      Metaregle_10
           )
  premisses (
               regle ( ?r )
               condition ( ?r present ( Voisine ( ?var ?var1 ) ) )
               condition ( ?r intersections_differentes ( ?var ?var1 ) )
              )
  conclusions (
               ote_condition ( ?r intersections_differentes ( ?var ?var1 ) )
              )
)
```

Tableau Représentation Métarègle Simplification

Le tableau Représentation Métarègle Simplification donne un exemple de métarègle : c'est une métarègle de simplification des règles produites par le système. Au jeu de Go, et de façon plus générale dans la plupart des jeux, deux intersections ou deux cases voisines sont différentes. Si la prémisse 'Voisine (?i1 ?i2)' est vérifiée, la fonction 'intersections_différentes (?i1 ?i2)' l'est aussi. Il est donc coûteux et inutile de vérifier que la fonction est bien vérifiée.

La métarègle de simplification commence par instancier dans la variable ?r de type Règle, toutes les règles de la base de règles à simplifier. Puis, elle sélectionne toutes les instanciations dans ces règles de la prémisse 'présent (Voisine (?var ?var1))', les métavariables ?var et ?var1 étant dans ce cas instanciées par des variables de type Intersection. Elle vérifie ensuite que la fonction 'intersections_différentes (?var ?var1)' est bien présente dans les conditions de la règle. Pour les règles dans lesquelles cette condition n'est pas vérifiée, le filtrage s'arrête.

L'action de la règle consiste à retirer la fonction 'intersections_différentes (?var ?var1)' qui est dans ce cas un test inutile.

```
( nom          (      Metaregle_ordonne_1
               )
  premisses    (
                 regle ( ?r )
                 condition ( ?r present ( Voisine ( ?var ?var1 ) ) )
                 instanciee ( ?var )
                 non_instanciee ( ?var1 )
                 priorite ( ?r present ( Voisine ( ?var ?var1 ) ) ?reel )
                 superieur_reel ( ?reel 3.79 )
                )
  conclusions  (
                 affecte_priorite ( ?r present ( Voisine ( ?var ?var1 ) ) 3.79 )
                )
)
```

Tableau Représentation Métarègle Ordonne

Un autre exemple de métarègle est donné dans le tableau Représentation Métarègle Ordonne. C'est une métarègle qui permet d'affecter à une prémisse d'une règle le nombre moyen de noeuds qu'elle va engendrer dans l'arbre d'instanciation des variables. Dans ce cas, on regarde si la prémisse 'présent (Voisine (?var ?var1))' est présente dans une règle, et si une de ses variables a déjà été instanciée. Lorsque c'est le cas, exécuter cette prémisse va engendrer en moyenne 3.79 noeuds si elle est filtrée sur une base de faits représentant un goban 19x19.

En effet, un goban 19x19 comporte 17*17 intersections ayant 4 voisines, 4*17 intersections ayant 3 voisines et 4 intersections ayant 2 voisines. Le nombre moyen de voisines pour une intersection quelconque est donc :

$$(17*17*4 + 4*17*3 + 4*2)/361=1368/361=3.79$$

La prémisse de la métarègle 'priorite (?r présent (Voisine (?var ?var1)) ?reel)' permet d'affecter à la variable ?reel la priorité pour la prémisse de la règle 'présent (Voisine (?var ?var1))'. Avant de

déclencher les métarègles d'ordonnancement, toutes les priorités sont initialisées à la valeur maximale pour une priorité. La valeur 3.79 est une borne supérieure pour la valeur moyenne des instanciations possibles, c'est la plus basse borne supérieure que puisse donner la métarègle compte tenu des conditions qu'elle a vérifié. Toutefois, d'autres métarègles contenant plus de conditions sur les prémisses des règles à ordonner peuvent trouver des bornes supérieures plus petites. Afin de garder la déclarativité des bases de métarègles, on ajoute donc la prémisse 'superieur_reel (?reel 3.79)' qui permet de vérifier qu'une borne supérieure plus petite que 3.79 n'a pas encore été trouvée par une autre métarègle contenant plus de prémisses.

Le métaprédicat 'affecte_priorité' permet d'associer une priorité à une prémisse, cette priorité sera ensuite utilisée pour ordonner les prémisses. Ceci sera traité plus en détail dans le chapitre sur la compilation des connaissances.

Mon système utilise d'autres types de métarègles qui seront décrites dans les chapitres les concernants.

Le langage d'expression des connaissances basé sur la logique des prédicats est un langage général qui permet de représenter déclarativement des connaissances sur des domaines très différents. Mon langage a ainsi été utilisé pour écrire un système expert en chimie organique qui nomme les molécules chimiques à partir de leur formule.

Nous verrons par la suite que ce langage a aussi été utilisé pour représenter des connaissances sur plusieurs jeux différents, ainsi que des connaissances sur la gestion d'une entreprise.

3.4 Une structure des connaissances dans le domaine des jeux

3.4.1 Trois niveaux de connaissance

Les coups

La notion de coup est la base de tous les jeux. Un coup est une des actions autorisées par les règles du jeu considéré. Il représente le fait de jouer une carte au Tarot ou au Bridge, ou bien le fait de placer une pierre sur une intersection au Go. Un coup permet généralement de satisfaire un ou plusieurs buts.

Les buts

Un but est associé à une situation atteignable. Il est à court terme et on peut en général atteindre un but en quelques coups. La réalisation d'un but peut faire partie d'un plan.

Les plans

Un plan est un ensemble de buts, ordonnés ou non. Un plan est un objectif à long terme du programme qui lui permet de sélectionner les buts intéressants à atteindre.

Par exemple, dans Gogol un plan possible est "**Attaquer un groupe**". Les différents buts qui permettent d'attaquer un groupe sont : "**Réduire son territoire**, "**Oter sa base de vie**", "**Oter un oeil**", "**Réduire son Influence**", "**Prendre le groupe**".

3.4.2 Le passage entre deux niveaux

La stratégie : un lien entre les plans et les buts

La stratégie permet de choisir les buts à atteindre en fonction du plan qu'on s'est fixé. Bateleur [Nigro 1993] est un système qui joue au Tarot. Chaque fois que Bateleur joue une carte, il doit choisir, parmi les buts de son plan de jeu, celui qui s'apparente le mieux à la situation.

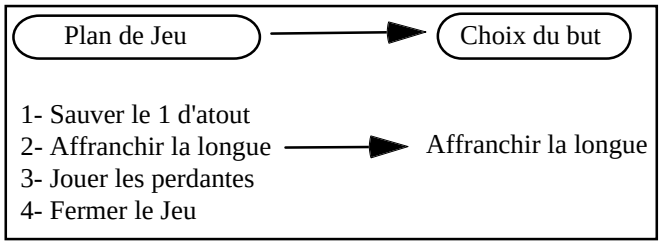


Figure Représentation Plan

Dans la figure Représentation Plan, le système possède un plan composé de quatre buts. La stratégie a conclu sur le fait qu'il fallait *Affranchir la longue* plutôt que de satisfaire les trois autres buts.

La tactique : un lien entre les buts et les coups

La tactique consiste à trouver un ou plusieurs coups à jouer qui satisfont le mieux un but fixé.

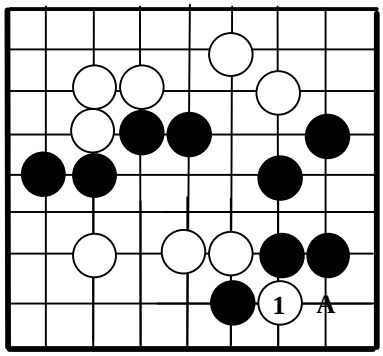


Figure Représentation But

Dans la figure Représentation But, le coup noir en A permettra à noir d'atteindre le but "Prendre le bloc 1" quelques coups plus tard.

3.5 Les méthodes de sélection

Pour modéliser le raisonnement, on peut utiliser des systèmes à base de règles, ou utiliser des algorithmes qui font des recherches arborescentes (de type Alpha-Bêta ou A*) ou encore programmer à base de contraintes. En collaboration avec J.M. Nigro, j'ai utilisé des méthodes de sélection à base de contraintes. Ces méthodes ont déjà été utilisées [Pinson 87] [Einhorn 88], mais pas dans le cadre de systèmes explicatifs. Les méthodes à base de contraintes peuvent être utilisées en combinaisons avec différents types de systèmes. Gogol mixe recherche arborescente et bases de règles pour connaître les buts qui peuvent être atteints, puis il utilise une méthode compensatoire pour choisir au niveau stratégique le coup à jouer. La méthode de sélection compensatoire a été utilisée jusqu'en Octobre 1995 pour choisir et expliquer les coups joués par Gogol.

3.5.1 Une définition des méthodes de sélection

Cette section définit cinq types de méthodes de sélection : la méthode conjonctive, la méthode disjonctive, la méthode lexicale, la méthode compensatoire ou encore une combinaison de plusieurs méthodes. Dans un souci de clarté, chaque définition est illustrée par un exemple simple, inspiré des différents mécanismes de sélection d'un élève dans une classe supérieure.

Considérons un ensemble d'individus $\alpha = \{A_1..A_n\}$, un ensemble de critères $\beta = \{B_1..B_p\}$ et un ensemble de valeurs $\varpi = \{V_{1,1}..V_{n,p}\}$. Chaque individu A_i est caractérisé par l'ensemble des critères B_j auxquels on a associé une valeur $V_{i,j}$. La formule suivante est donc vérifiée :

$$\forall A_i \in \alpha, \forall B_j \in \beta, \exists V_{i,j} / (A_i .B_j)=V_{i,j}.$$

Afin de pouvoir évaluer les élèves en fonction de leurs notes, le professeur a besoin des éléments suivants :

α = {Jean, Paul, Jacques}
β = {Mathématiques, Informatique, Français}
ϖ = {8, 13, 18, 5, 7, 3, 15, 17, 10}

	Jean	Paul	Jacques
Mathématiques	8	5	15
Informatique	13	7	17
Français	18	3	10

Ainsi, Paul a eu 7 en informatique *: (Paul . Informatique) = 7*

Considérons maintenant un ensemble de contraintes $\chi = \{C_1..C_m\}$, un ensemble d'opérateurs classiques $\theta = \{O_1..O_s\} = \{<, >, =, \geq, \leq\}$ et un ensemble de bornes $W=\{W_1..W_z\}$. Une contrainte comporte trois éléments : $C_t = (B_j\ O_g\ W_k)$. Par exemple, la contrainte (Mathématiques ≥ 10) pourrait être utilisée par le professeur pour sélectionner des élèves.

Soit la fonction de filtrage f d'un individu par une contrainte :
$f : \alpha, \chi \rightarrow \alpha$
$(A_i, C_t) \rightarrow A_i$ Si $(V_{i,j}\ O_g\ W_k)$ est vrai *(avec $A_i .B_j = V_{i,j}$ et $C_t = (B_j\ O_g\ W_k)$)*
$(A_i, C_t) \rightarrow \{\}$ Si $(V_{i,j}\ O_g\ W_k)$ est faux

Par exemple, *f(Jean, (Mathématiques ≥ 10)) = {}* car Jean n'a que 8 en Mathématiques alors que *f(Jean, (Français ≥ 18)) = Jean* car Jean a 18 en Français.

a) La méthode conjonctive : la méthode peut être modélisée de la façon suivante :
Conjonctive : $P(\chi) \rightarrow P(\alpha)$
$(C_1..C_p) \rightarrow \{A_1..A_u\}$ avec $\forall A_i \in \{A_1..A_u\}, \forall C_t \in P(\chi), f(A_i, C_t) = A_i$

Pour sélectionner les élèves qui ont la moyenne dans chacune des matières, le professeur utilisera plutôt la méthode conjonctive sur trois contraintes associées aux trois matières : *Conjonctive ((Mathématiques ≥ 10) (Informatique ≥ 10) (Français ≥ 10))*. L'application de cette méthode renvoie le singleton {Jacques}. En effet, il est le seul a avoir plus de 10 dans toutes les matières.

b) La méthode disjonctive : la méthode peut être modélisée de la façon suivante :
Disjonctive : $P(\chi) \rightarrow P(\alpha)$
$(C_1..C_p) \rightarrow \{A_1..A_u\}$ avec $\forall A_i \in \{A_1..A_u\}, \exists C_t \in P(\chi), f(A_i, C_t) = A_i$

Le professeur utilisera la méthode disjonctive , par exemple, pour sélectionner les élèves qui ont eu la moyenne dans au moins une des trois matières : *Disjonctive ((Mathématiques ≥ 10) (Informatique ≥*

10) (Français ≥ 10)). L'application de cette méthode renvoie le doubleton {Jean, Jacques}. En effet, parmi les trois élèves, seul Paul n'a pas eu une seule fois la moyenne.

c) La méthode lexicale : la méthode peut être modélisée de la façon suivante : Soit $P(\chi)$ une partition **ordonnée** de χ tel que C_{t1} est placée avant C_{t2} et C_{tj} est placée avant $C_{t(j+1)}$.
Lexicale : $P(\chi) \rightarrow P(\alpha)$

$(C_1..C_p) \rightarrow \{A_1..A_u\}$

avec $(card\{A_1..A_u\}>1)$, $\forall A_i \in \{A_1..A_u\}$, $\forall C_t \in P(\chi)$, $f(A_i, C_t) = A_i$
avec $(card\{A_1..A_u\}=1$ et $\{A_1..A_u\}=\{A_i\})$,

$\exists s \in \{1..p\}$ / **(** $\forall tj \leq s$, $C_{tj} \in P(\chi)$, $f(A_i, C_{tj}) = A_i$
et $\forall A_x \in \alpha$, $A_x \neq A_i$, $\exists tj \leq s$ / $C_{tj} \in P(\chi)$, $f(A_i, C_{tj}) = \{\}$ **)**

Imaginons que le professeur ait un comportement élitiste et préfère ne sélectionner qu'un seul élève. Il recherche un élève avec plus de 10 en mathématiques. Si plusieurs élèves ont la moyenne alors il fera une nouvelle sélection et recherchera les élèves avec plus de 10 en mathématiques et en informatique. Dans ce cas le professeur utilisera la méthode lexicale :

Lexicale ((Mathématiques ≥ 10) (Informatique ≥ 10))

Seul Jacques vérifie la première contrainte. Il n'est donc pas utile de s'intéresser à la deuxième contrainte.

Au lieu de prendre les deux matières mathématiques puis informatique, prenons les matières informatique puis français :

Lexicale ((Informatique ≥ 10)(Français ≥ 10))

dans ce cas, deux personnes (Jean et Jacques) satisfont les deux contraintes. En effet, Jean a 13 en informatique, 18 en français et Jacques 17 et 10.

d) la méthode compensatoire

La méthode compensatoire est un peu particulière car elle ne suit pas exactement le même principe que les méthodes précédentes. En effet, elle n'utilise pas un ensemble de contraintes, mais elle affecte des coefficients aux critères de sélection. Considérons qu'un ensemble de coefficients $K = \{K_1..K_p\}$ soit associé à l'ensemble des critères $\beta = \{B_1..B_p\}$ tel qu'on associe un K_i à un B_i. Deux méthodes compensatoires sont décrites ci-dessous : la méthode compensatoire avec un seuil et la méthode compensatoire élitiste.

<u>La méthode compensatoire avec un seuil</u> : la méthode peut être modélisée de la façon suivante : Soit un seuil $S \in \Re$.
CompensatoireSeuil : $(\beta \times K)^p \rightarrow P(\alpha)$

$(B_1\ K_1) .. (B_p\ K_p) \rightarrow \{A_1..A_u\}$

avec $\forall A_i \in \{A_1..A_u\}$, $\sum_{j=1}^{p} (V_{i,j} * K_j) \geq S$ $\quad (A_i . B_j = V_{i,j})$

Imaginons que le professeur donne des coefficients d'importance aux trois matières : 3 pour les mathématiques et l'informatique, 1 pour le français. Il ne prendra un élève que si la somme de toutes ses notes pondérées atteint au moins 70. Cette méthode permet à un élève qui a une mauvaise note dans une matière de pouvoir la compenser par une bonne note dans une autre matière. Dans notre exemple, le professeur utilisera la méthode suivante :

Compensatoire ((Mathématiques 3) (Informatique 3) (Français 1))

Jean (81) et Jacques (106) sont acceptés, mais Paul (39) est refusé car il a moins de 70.

Lorsque le professeur décide de sélectionner le meilleur élève tout en donnant des coefficients d'importance à chaque matière, il utilise la méthode compensatoire élitiste.

La méthode compensatoire élitiste : la méthode peut être modélisée de la façon suivante : soit un opérateur Op $\in$ {Min, Max}.
CompensatoireOp : $(\beta \times K)^p \rightarrow P(\alpha)$
$(B_1\ K_1) .. (B_p\ K_p) \rightarrow \{A_1..A_u\}$

avec $\forall A_{i1} \in \{A_1..A_u\}$, $\forall A_{i2} \in \alpha, \sum_{k=1}^{p}(V_{i1,k} * K_k) \geq \sum_{k=1}^{p}(V_{i2,k} * K_k)$ ($A_i .B_j = V_k$) Si Op=Max

En reprenant les mêmes coefficients que l'exemple précédent, Jacques (106) sera sélectionné car il possède le meilleur total et les deux autres élèves seront rejetés.

e) Une combinaison de plusieurs méthodes

Deux types de combinaisons peuvent être distingués : une imbrication de méthodes et une méta-méthode.

Une imbrication de méthodes : les méthodes peuvent s'appliquer sur des contraintes ou à nouveau sur des méthodes. Cette technique doit être employée avec beaucoup de précautions car plus le nombre de niveaux d'imbrication est important plus la complexité est forte et la compréhension difficile.

Un exemple d'imbrication d'une méthode conjonctive dans une méthode lexicale :
Lexicale ((Mathématiques ≥ 8)
(Conjonctive ((Informatique ≥ 12) (Français ≥ 12)))
(Informatique ≥ 15))

Cet exemple présente une méthode lexicale appliquée à trois contraintes. L'application de la première contrainte *(Mathématiques ≥ 8)* sélectionne Jean et Jacques. La sélection n'étant pas unique la deuxième contrainte *(Conjonctive ((Informatique ≥ 12) (Français ≥ 12)))* est traitée. Ainsi, seul Jean satisfait les deux premières contraintes. La troisième contrainte n'est pas prise en compte car une solution unique a été trouvée.

Une méta-méthode : Comme son nom l'indique, une méta-méthode est une méthode qui s'applique sur d'autres méthodes. La méta-méthode s'écrit en respectant la syntaxe suivante : *méta-méthode (méthode_1 ... méthode_M) (C_1 ... C_v)*. Lorsque la structure est développée, elle peut être assimilée à une imbrication particulière de méthodes :
méta-méthode ((méthode_1 (C_1 ... C_v))
... ...
(méthode_M (C_1 ... C_v)))

On notera que la méta-méthode s'applique sur la liste des méthodes et non pas directement sur la liste de contraintes associées aux méthodes.

Prenons un exemple : le professeur décide de sélectionner les élèves qui ont la moyenne en mathématiques ou en informatique (disjonctive). Dans le cas où plusieurs élèves sont sélectionnés, le professeur ne sélectionnera que ceux qui ont la moyenne dans les deux matières. En fait, il a utilisé la méta-méthode suivante :
Lexicale (Disjonctive Conjonctive) (Mathématiques ≥ 10) (Informatique ≥ 10)

La méta-méthode peut être développée :
Lexicale ((Disjonctive ((Mathématiques ≥ 10) (Informatique ≥ 10))) → Jean, Jacques
(Conjonctive ((Mathématiques ≥ 10) (Informatique ≥ 10))) → Jacques

La méta-méthode Lexicale indique qu'il faut tout d'abord appliquer la méthode Disjonctive sur les deux contraintes : Jean et Jacques sont sélectionnés. La solution n'est pas unique, il faut donc appliquer la seconde méthode sur les deux contraintes : Jacques sera le seul sélectionné.

Si l'imbrication de plusieurs méthodes est difficile à gérer au niveau des explications. Il est possible de le faire avec une méta-méthode : l'explication s'orientera alors sur le choix d'une méthode.

3.5.2 Application dans le domaine des jeux

a) La méthode conjonctive et la méthode disjonctive

La méthode conjonctive et la méthode disjonctive seront illustrées à travers le fonctionnement de Bateleur. Supposons que sept cartes puissent être jouées par le système (α = {R♣, C♣, 9♣, 8♣, R♥, V♥, 5♦}) et que trois critères leur soient associés (β = {Valeur de la carte, Gagner le pli, La couleur longue}). L'ensemble des valeurs ϖ peut être représenté par le tableau suivant :

	R♣	C♣	9♣	8♣	R♥	V♥	5♦
Valeur de la carte	4.5	2.5	0.5	0.5	4.5	1.5	0.5
Gagner le pli	3	1	-3	-3	3	0	-3
La couleur longue	3	3	3	3	-3	-3	-3

Le critère *Valeur de la carte* indique le nombre de points associé à une carte. Par exemple le V♥ vaut 1.5 points. Lorsqu'une carte a une évaluation de 3 pour le critère *Gagner le pli*, cela signifie que la carte a beaucoup de chance de remporter un pli. Par contre, une évaluation de -3 indique qu'elle a très peu de chance de gagner un pli. Entre ces deux valeurs extrêmes peuvent s'intercaler des valeurs intermédiaires. Par exemple le C♣ a une évaluation de 1 car il a de bonne chance de remporter un pli, mais ce n'est pas certain car la D♣ d'un adversaire peut l'en empêcher. Le troisième critère *La couleur longue* repère la couleur la plus fournie en cartes. Une carte appartenant à la couleur la plus longue aura une évaluation de 3 (sinon elle sera de -3).

Lorsqu'il faut entamer un pli qu'il veut absolument remporter tout en jouant dans sa couleur la plus longue, Bateleur utilise la méthode conjonctive sur trois contraintes :

méthode (C_1..C_p)	($g(C_1)$..$g(C_p)$)
Conjonctive ((La couleur longue > 0) (Gagner le pli > 2) (Valeur > 1))	(R♣, C♣, 9♣, 8♣) (R♣, R♥) (R♣, C♣, R♥, V♥) La sélection = (R♣)

Lorsque son partenaire remporte le pli et que Bateleur doit se défausser[3], il utilise la méthode disjonctive associée à deux contraintes :

méthode ($C_1..C_p$)	($g(C_1)..g(C_p)$)
Disjonctive ((Gagner le pli < 0) (Valeur > 3))	(9♣, 8♣, 5♦) (R♣, R♥) La sélection = (9♣, 8♣, 5♦, R♣, R♥)

b) La méthode de sélection lexicale

L'exemple d'utilisation de la méthode lexicale est basé sur un choix stratégique aux échecs. Différents buts peuvent être intéressants aux échecs suivant la stratégie choisie. La figure Représentation lexicale donne des évaluations du degré avec lequel les différents buts permettent d'atteindre les objectifs stratégiques :

Propriétés	Agressif	Gain Rapide	Long Terme	Important	Risque
Immobiliser la Reine	2	5	2	8	8
Capturer la Reine	10	10	5	10	10
Contrôler le Centre	2	0	1	8	3
Prendre le Roi	10	10	10	10	5
Protéger son Roi	0	0	10	10	0
Prendre un cavalier	6	3	3	7	3
Protéger un pion	0	0	1	1	2

Figure Représentation lexicale : évaluation des buts

La sélection stratégique d'un plan consiste alors à appliquer la méthode lexicale sur des contraintes stratégiques :

Lexicale	((Agressif > 1)	(Importance > 7)	(risque < 5))
	↑	↑	↑
	Immobiliser la Reine Capturer la Reine Contrôler le Centre Prendre le Roi Prendre un cavalier	Immobiliser la Reine Capturer la Reine Contrôler le Centre Prendre le Roi	Contrôler le Centre

Le but Contrôler le Centre est sélectionné.

c) La méthode de sélection compensatoire

[3] se défausser : un joueur se défausse lorsqu'il n'a plus aucune carte de la couleur demandée et il n'a plus d'atout. Il est alors obligé de donner une carte d'une autre couleur qui ne peut en aucun cas remporter le pli.

L'exemple qui suit est tiré du module tactique de mon système. Pour chaque coup, Gogol possède une évaluation des différents paramètres liés aux buts que le coup permet d'atteindre. Dans l'exemple qui suit, l'ensemble des buts qui sont calculés fait partie du plan Protéger son territoire.

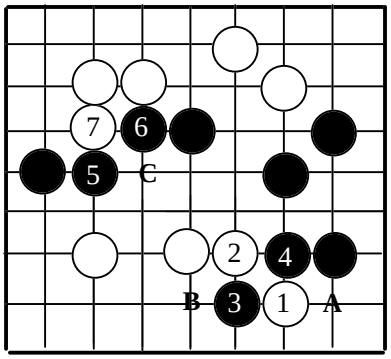


Figure Représentation Goban

Coups	A	B	C
Pierres prises	1	1	0
Pierres sauvées	1	1	2
Territoire sauvé	0	0	1
Influence sauvée	2	2	12
Territoire pris	1	2	0
Influence prise	0	3	0

Figure Représentation compensatoire

On applique alors la méthode :

Compensatoire (Pierres prises 2) (Pierres sauvées 2) (Territoire sauvé 2) (Influence sauvée 1) (Territoire pris 2) (Influence prise 1)

qui donne pour les différents coups :

Coup A : 8 Coup B : 13 Coup C : 18

Le coup choisi par la méthode compensatoire en fonction des informations de la figure Représentation compensatoire sera donc le coup C.

3.6 Conclusion

Mon système utilise des métaprédicats et des métarègles représentés sous forme logique pour manipuler et transformer ses règles. Les métarègles sont très utilisées pour la compilation (Cf. chapitre compilation).

J'ai défini les termes de coup, but, plan, stratégie et tactique qui permettent de structurer la connaissance dans la plupart des jeux. J'ai ensuite défini des méthodes de sélection à base de contraintes qui permettent de modéliser le raisonnement dans beaucoup de jeux. Gogol utilise la méthode de sélection compensatoire pour sélectionner ses coups au niveau stratégique.

Les patterns géométriques sont limités par leur manque de généralité, toutefois c'est une représentation qui permet de matcher les règles de façon efficace. Dans le chapitre sur la compilation, nous verrons comment compiler une représentation logique en représentation plus proche de la représentation géométrique, de façon à matcher plus vite plusieurs règles spécialisées plutôt qu'une seule règle générale. Ce processus de compilation est bien connu en programmation logique et s'appelle l'évaluation partielle. Il permet de faire une partie des calculs au moment de la compilation plutôt que de les refaire chaque fois que l'on exécute le programme.

Mon propos étant l'apprentissage, je cherche avant tout à représenter la connaissance de la façon la plus générale possible pour que les connaissances créées par le système soient les plus générales. Ainsi, lorsqu'une règle logique représente trente-six patterns géométriques, mon programme n'a besoin d'apprendre qu'une seule fois la règle logique alors qu'il aurait besoin de plus d'exemples et d'un apprentissage plus long pour apprendre les trente-six patterns géométriques correspondants s'il utilisait directement une représentation à base de patterns géométriques. De plus, ma représentation peut être envisagée dans un cadre autre que l'apprentissage. Elle peut aussi être très utile aux

concepteurs de programme puisqu'elle permet de donner la connaissance sous forme concise, ce qui fait gagner beaucoup de temps à l'expert : elle n'écrit qu'une règle au lieu d'en écrire trente-six.

3.7 Bibliographie

[Boon 1990] - Marc Boon. *A pattern matcher for Goliath*. Computer Go 13, pp 13-23, 1990.

[Einhorn 1988] - H.J. Einhorn, R.M. Hogarth, *Decision Making*. p 127, Cambridge University Press.

[Fotland 1993] - David Fotland. *Knowledge Representation in The Many Faces of Go*. Second Cannes/Sophia-Antipolis Go Research Day, Février 1993.

[Nigro 1993] - J.M. Nigro, *BATELEUR : un système expert modélisant le comportement de joueurs au tarot*, Second European Congress on Systems Science, Prague.

[Nigro 1996] - J.M. Nigro, T. Cazenave, *Constraint-based Explanations in Games*. IPMU96, Grenade.

[Pinson 1987] - S. Pinson, *Méta-modèles et heuristiques de jugement : le système CREDEX. Application à l'évaluation du risque crédit entreprise*.Thèse de l'Université Paris 6, 1987.

[Pitrat 1990] - Jacques Pitrat. *Métaconnaissance futur de l'intelligence artificielle.* Hermès, 1990.

[Wilcox 1995] - B. Wilcox.*The EGO program*, Message envoyé à la mailing-list Computer-Go, 1995.

Table des Matières du Chapitre 4

4 LES EXPLICATIONS 53

4.1 Les explications tactiques 54
4.1.1 Limites des représentations existantes 54
4.1.2 Une solution : représenter le temps 57
4.1.3 La résolution de problèmes 59
4.1.4 Choix des faits à expliquer 60
4.1.5 Mécanisme de retour dans la trace 64
4.1.6 Influence de la théorie du domaine sur la généralité des explications 65

4.2 Les explications stratégiques 67
4.2.1 Les explications positives et négatives 67
4.2.2 Définition des explications à partir des méthodes de sélection 67
4.2.3 Applications dans le domaine des jeux 72

4.3 Conclusion 73

4.4 Bibliographie 74

4 Les explications

Le but général des explications est de faire comprendre un mécanisme. La compréhension est une composante fondamentale de l'apprentissage, que cet apprentissage soit effectué par un système ou par une personne. Le mécanisme d'explication est un mécanisme fondamental dans mon système d'apprentissage.

Les premières recherches dans le domaine des explications ont été très liées aux systèmes à base de connaissances. Ainsi, Clancey est parti du système Mycin [Davis & al 1977] pour construire les systèmes explicatifs Guidon [Clancey 1987] puis Neomycin [Clancey 1986]. De même, Swartout a développé le système Xplain [Swartout 1983] à partir d'un système de prescription de la digitaline : "Therapy Digitalis Advisor". Ces systèmes sont basés sur l'explicitation des connaissances et notamment la décomposition de tâches en sous-tâches.

La plupart des recherches en explication portent sur des systèmes qui donnent des explications à un être humain. Mon système est capable d'engendrer des explications qui atteignent deux objectifs différents. Premièrement, **il peut expliquer les raisons principales de ses coups à son utilisateur**. Que ce soit pour le faire progresser si c'est un joueur débutant, ou que ce soit pour permettre au programmeur de détecter plus facilement les erreurs du système. Deuxièmement, **il peut s'expliquer à lui-même les raisons de ses déductions**. Les explications qu'il se donne à lui-même lui permettent de comprendre les raisons de ses succès et par la suite d'apprendre à les réitérer.

Il y a deux sortes d'explications dans mon système : les explications **tactiques** et les explications **stratégiques**. Les explications tactiques permettent de donner les faits et les règles qui ont permis de déduire un fait intéressant. Les explications tactiques portent sur des raisonnements certains. Les explications stratégiques permettent de donner les raisons pour lesquelles un coup a été choisi au niveau global, elle portent sur des raisonnements flous et approximatifs. Les explications tactiques sont plutôt destinées au système alors que les explications stratégiques sont plutôt destinées à l'utilisateur du système.

Les explications tactiques sont basées sur le mécanisme de retour dans la trace et les surprises. Les explications stratégiques sont basées sur les méthodes de sélection.

4.1 Les explications tactiques

Les explications que le système se donne à lui-même sont celles qui donnent les raisons pour lesquelles **un coup a permis de modifier l'état d'un jeu combinatoire à valeurs inconnues associé à un but**. Ces raisons sont exprimées sous forme d'une liste de faits décrivant l'état d'une partie du damier avant que le coup n'ait été joué.

Le but des explications tactiques est de trouver un sous-ensemble pertinent de l'ensemble des faits. Ce sous-ensemble est composé des faits représentant une situation avant une résolution de problème. **La propriété qui distingue ce sous-ensemble est qu'il permet la déduction d'un fait jugé intéressant à expliquer**.

Pour chaque domaine dans lequel on fait des explications, on dispose d'une théorie du domaine. C'est un ensemble de règles déclaratives qui représente les connaissances sur la façon dont le monde évolue en fonction des actions qui y sont effectuées. Dans les jeux, la théorie du domaine est composée des règles du jeu, ou plus exactement, des règles de transition entre une position et la même position après un coup. Les règles de transition de la théorie du jeu de Go sont données dans l'annexe A. Pour un programme de gestion la théorie du domaine est composée de règles permettant de calculer différents indices comme par exemple les liquidités d'une entreprise.

Pour faire de bonnes explications, on doit disposer de connaissances déclaratives. La déclarativité des connaissances comporte plusieurs contraintes. La première est la représentation explicite des connaissances sous une forme manipulable par le programme, cet aspect de la déclarativité est présent dans les programmes écrits sous forme logique, comme en Prolog ou dans un système expert. La seconde contrainte qu'impose la déclarativité est la suivante : les instructions d'un programme doivent être données sans l'ordre dans lequel les exécuter. Un programme déclaratif est donc explicite et ses instructions peuvent être exécutées dans n'importe quel ordre. L'absence d'ordre entre les instructions d'un programme est une composante très importante de la déclarativité, car elle permet de faire de bonnes explications et donc de bien apprendre.

Afin d'avoir une représentation des connaissances déclarative, j'utilise le métaprédicat 'présent', qui vérifie qu'un fait est présent dans la base de faits, et le métaprédicat 'ajoute', qui ajoute un fait dans la base de faits.

Je n'utilise pas dans la théorie du domaine les métaprédicats 'absent' et 'enlève'. L'utilisation des métaprédicats 'absent' et 'enlève' peut amener, s'ils ne sont pas utilisés avec une bonne sémantique, à avoir des théories du domaine procédurales. Je vais montrer pourquoi sur un exemple d'explication au jeu d'Echecs.

4.1.1 Limites des représentations existantes

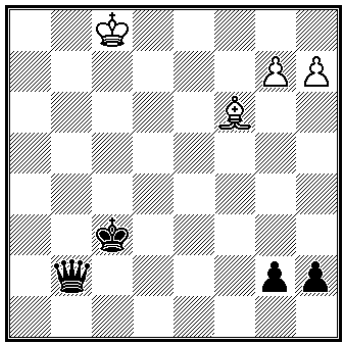


Figure Explication Echecs

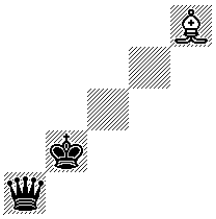


Figure Explication Fourchette

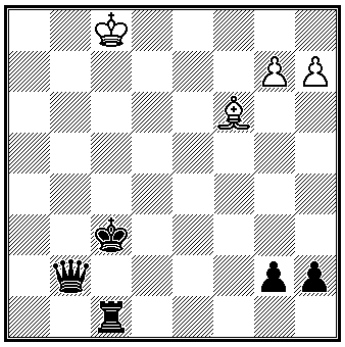


Figure Explication Contre Exemple

La figure Explication Echecs a été utilisé par [Minton 1984] pour expliquer comment son système d'apprentissage fait des explications aux Echecs. Sur cette position, son programme trouve le concept de fourchette. Il explique que la dame noire peut être prise parce que le roi noir est en échec, qu'il est forcé de se déplacer pour parer à l'échec et que la dame noire est derrière lui sur la diagonale

du fou blanc. Les faits importants de l'explication sont résumés dans la figure Explication Fourchette. Toutefois, comme il a déjà été remarqué dans [Puget 1987], la méthode d'explication utilisée par Minton, et de manière plus générale celle donnée par [Mitchell 1986] peut amener à donner des explications trop générales et donc parfois fausses. La Figure Explication Contre Exemple montre une position dans laquelle l'explication de la figure Explication Fourchette se révèle fausse car trop générale.

Les théories du domaine utilisées actuellement dans les systèmes d'apprentissage à partir d'explications ne sont pas déclaratives et amènent à des explications incomplètes. Les règles créées à partir de ces explications sont donc trop générales : dans certains cas elles donnent des conclusions fausses. Pour qu'elles donnent toujours des conclusions vraies, il faudrait leur ajouter des prémisses qui correspondent aux explications manquantes. Ce défaut des systèmes actuels est très gênant et même rédhibitoire lorsqu'on écrit un programme qui se programme lui-même en utilisant des techniques d'amorçage. En effet, un tel programme utilise les règles qu'il a apprises pour expliquer les nouvelles situations qu'il rencontre et créer de nouvelles règles. Cependant si ces nouvelles règles sont créées à partir de règles fausses, en utilisant un mécanisme qui les rend encore plus fausses, le système ne tarde pas à déduire des règles contradictoires, fausses et inutiles. Il est donc primordial pour un système autonome s'auto-améliorant de disposer d'un mécanisme d'explication juste et fiable.

```
RULE Create-four-in-a-row
    IF      input-move(<square>,<p>)
            is-empty(<square>)
            three-in-a-row(<3position>,<p>)
            extends(<3position>,<square>)
            composes(<newposition>,<square>,<3position>)
    THEN
            DELETE
                    three-in-a-row(<3position>,<p>)
                    input-move(<square>,<p>)
            ADD
                    four-in-a-row(<newposition>,<p>)
```

Tableau Explication Règle GoMoku

L'incomplétude des explications est due à l'utilisation inappropriée des métaprédicats 'absent' et 'enlève' dans les règles de la théorie du domaine. Toutefois l'utilisation de ces métaprédicats peut se justifier lorsqu'on ne cherche pas à obtenir tout de suite des connaissances générales et vraies, mais qu'on accepte de surgénéraliser pour ensuite spécialiser les règles trop générales sur des contre-exemples. J'ai choisi de ne pas créer de règles trop générales que l'on peut ensuite raffiner. Pendant tout le début de l'année 1995, Gogol a utilisé des règles trop générales pour engendrer des coups forcé dans ses arborescences. Bien qu'elles aient été raffinées, ces règles ne voyaient parfois pas certains coups forcés. Or ce coup forcé pour empêcher l'adversaire d'atteindre un but était parfois un coup qui ne marchait pas. Le résultat de la recherche arborescente était donc faux (Gogol croyait par exemple qu'un jeu était gagné alors qu'il ne l'était pas). Même si une règle ne voie pas un coup forcé que dans quelques cas, lors d'une recherche arborescente qui peut comporter plusieurs dizaines de niveaux de profondeurs, la probabilité qu'un coup forcé n'aie pas été vu est proportionnelle à ce nombre de niveaux. De plus certaines règles ne sont pas encore raffinées et ont une grande probabilité de ne pas voir un coup forcé. Le résultat était que les résultats des recherches arborescentes étaient souvent faux. Gogol jouait alors de plus en plus mal au fur et à mesure de son apprentissage. Il faut dire qu'au jeu de Go, si l'on investit dix coups pour empêcher l'adversaire d'atteindre un but, et qu'on se rend compte au bout de dix coups que ce but ne peut être empêché, cela a des conséquences catastrophiques.

J'ai donc fait le choix de n'apprendre que des règles générales mais toujours vraies.

Si on veut créer des explications fiables, il ne faut pas utiliser le métaprédicat 'enlève' en conclusion d'une règle de la théorie du domaine, car si cette règle n'est pas déclenchée sur

l'exemple d'apprentissage, elle peut être déclenchée plus tard sur une autre exemple similaire (voir Figure Explication Contre Exemple). L'explication valable sur le premier exemple se révélera fausse sur le second exemple car la règle contenant le métaprédicat 'enlève' en conclusion aura ôté un des faits nécessaires à l'explication. Par exemple dans le tableau Explication Règle GoMoku tiré de [Minton 1984], l'utilisation du 'delete' pour enlever le fait 'three-in-a-row (<3position>,<p>)' est une utilisation abusive du 'delete'. Elle force à donner un ordre entre les règles, et lorsque cette règle ne s'applique pas, mais que le fait 'three-in-a-row (<3position>,<p>)' doit être expliqué, il faudrait expliquer pourquoi la règle du tableau Explication Règle GoMoku ne s'est pas déclenchée et n'a pas enlevé le fait. Expliquer de façon rigoureuse pourquoi un fait n'a pas été déduit par une règle amène à une explosion combinatoire du nombre d'explications nécessaires, demande une théorie du domaine complète et donne des explications beaucoup trop spécifiques. Une explication faite à partir d'une théorie du domaine contenant le métaprédicat 'enlève' peut donc être trop générale. Une solution serait d'expliquer pourquoi un fait a été déduit mais aussi pourquoi toutes les règles concluant sur l'enlèvement d'un des faits de l'explication n'ont pas été déclenchées. Cependant, cela amène à de mauvaises explications.

De même, si on veut créer des explications fiables, on ne doit pas utiliser le métaprédicat 'absent' en prémisse d'une règle, car dans ce cas il faut expliquer pourquoi un fait n'est pas présent. Or, il y a une différence entre un fait absent parce qu'il n'a pas encore été déduit et un fait absent parce que non déductible. Si on utilise une représentation déclarative et donc sans ordre entre les règles, on ne peut pas savoir si un fait est absent parce qu'il n'est pas déductible ou s'il est absent parce qu'il n'a pas encore été déduit. Il faut donc utiliser des prédicats qui disent explicitement qu'un fait n'est pas déductible. Il y a deux problèmes liés à la négation par l'absence. Le premier problème est que, pour que la négation par l'absence soit fiable, on doit disposer d'une théorie du domaine complète, ce qui est très difficile à obtenir dans des domaines complexes comme le jeu de Go. Utiliser la négation par l'absence avec une théorie du domaine incomplète amène à donner des explications fausses ou incomplètes. Toutefois des recherches récentes ont montré qu'il est possible d'engendrer automatiquement une théorie du domaine complète à partir d'une définition concise et générale des règles du jeu de Go [Moneret 1996]. Le deuxième problème est que la négation par l'absence démontre l'impossibilité en calculant explicitement l'ensemble des retours arrières possibles. Elle est donc très coûteuse en terme d'explications puisque dans certains cas pour expliquer l'absence d'un fait, elle calcule l'ensemble de tous les faits possibles et montre que le fait n'est pas inclus dans cet ensemble. Elle amène donc à des explications très volumineuses et beaucoup trop spécifiques. Des tentatives pour éviter les problèmes posés par la négation par l'absence et plus généralement pour séparer les programmes logiques de leur contrôle ont été faites dans le cadre de la programmation logique comme le langage Goëdel [Hill 1994] ou l'approche déclarative de [Beckstein 1996].

Les programmes logiques, les bases de données déductives et plus généralement les théories de la non monotonie, utilisent des formes variées de la négation par défaut, notée *not F*, dont la caractéristique principale est que not F est la valeur "par défaut", c'est à dire qu'elle est tenue pour vraie si l'on n'arrive pas à prouver le contraire. La signification de "on n'arrive pas" dépend de la sémantique utilisée. Par exemple dans l'hypothèse du monde clos généralisé de Minker [Minker 1982] [Gelfond 1989] ou dans la circonscription de McCarthy [McCarthy 1980], not A signifie que A est faux dans tous les modèles minimaux. Dans la sémantique de Clark [Clark 1978], cette forme de négation est appelée négation par l'absence parce que 'not A' est dérivable quand tous les essais pour prouver A ont échoués. Par exemple la formule accusé (x) $\wedge$ $\neg$coupable (x) $\supset$ acquitté (x) signifie qu'une personne accusée d'un crime doit être acquittée si elle est prouvée non coupable. Alors que la formule accusé (x) $\wedge$ not coupable (x) $\supset$ acquitté (x) signifie qu'une personne doit être acquittée à moins que sa culpabilité n'aie été prouvée.

La négation par l'absence est utilisée en programmation logique pour représenter l'inconnu parce qu'elle est efficace à implémenter lorsqu'on utilise le chaînage arrière. J'ai choisi pour ma part de représenter explicitement l'inconnu et de faire les déductions en chaînage avant.

Ces critiques sur l'utilisation abusive des métaprédicats 'enlève' et 'absent' ne sont pas des critiques sur le bien-fondé de ces métaprédicats qui peuvent être utiles. Mais une critique sur la sémantique associée à ces métaprédicats. Il est possible d'utiliser ces métaprédicats dans une théorie du domaine et malgré tout que cette théorie du domaine soit déclarative. Cependant il faut être très

prudent lors de leur utilisation. **Le métaprédicat ‘enlève’ ne doit être utilisé que pour enlever des faits devenus inutiles et le métaprédicat ‘absent’ ne doit être utilisé que pour ne pas redéduire des faits déjà déduits**.

4.1.2 Une solution : représenter le temps

La non utilisation des prédicats ‘absent’ et ‘enlève’ pose toutefois le problème de représenter les changements qui interviennent dans le monde après qu’une action ait été effectuée. Certains faits ne sont plus vrais après un coup. La solution que j'ai retenue est de représenter explicitement l’évolution des faits dans le temps. Ainsi pour chacun des faits susceptibles de changer, j'ai un prédicat qui donne sa valeur avant le coup et un autre prédicat qui donne sa valeur après le coup. Par exemple au jeu de Go, je fais la différence entre la couleur d’une intersection avant le coup et sa couleur après le coup.

Le métaprédicat ‘absent’ est en fait utilisé mais uniquement afin de ne déclencher que les règles qui ajoutent un nouveau fait. Sa sémantique n’est pas celle de la négation mais celle de l’absence.

Les connaissances sur la configuration topologique du damier ne peuvent pas varier au cours d’une partie, les faits correspondants ne sont représentés que par un seul prédicat. Le tableau Explication Règles du Domaine donne des exemples de règles de la théorie du domaine pour le jeu de Go. Les expressions précédées d’un point d’interrogation représentent les variables. La couleur amie est notée @, la couleur ennemie est notée O, une intersection vide est notée +. La règle Règle_valuation_jeu_1 est une règle qui donne une valeur au jeu de prendre le bloc ennemi ?b. Ce jeu est gagné si le joueur ami joue sur l’intersection ?i et que les conditions de la règle sont vérifiées. La règle Règle_coup_légal_5 est une règle qui donne des conditions suffisantes pour qu’un coup ami sur l’intersection ?i soit légal.

Règle_valuation_jeu_1 : premisses (présent (Couleur_intersection_après (?b O)) présent (Nombre_libertés_après (?b 1)) présent (Liberté_bloc_après (?b ?i)) présent (Coup_légal_après (@ ?i))) conclusions (ajoute (Coup_après (@ Prendre ?b ?i GI)))	Règle_coup_légal_5 : premisses (présent (Couleur_intersection_avant (?i +)) présent (Voisine (?i ?i1)) présent (Couleur_intersection_avant (?i1 +))) conclusions (ajoute (Coup_légal_avant (@ ?i)))
Règle_intersection_5 : premisses (présent (Coup (@ ?i)) présent (Coup_légal_avant (@ ?i)) présent (Voisine (?i ?i1)) présent (Couleur_intersection_avant (?i1 +))) conclusions (ajoute (Couleur_intersection_après (?i @)))	Règle_bloc_1 : premisses (présent (Nombre_libertés_avant (?b ?n)) supérieur (?n 1) présent (Bloc_avant (?i ?b))) conclusions (ajoute (Bloc_après (?i ?b)))

Tableau Explication Règles du Domaine

Les règles Règle_intersection_5 et Règle_bloc_1 sont des règles de transition entre l’état de la position avant le coup et l’état de la position après le coup. Ces règles de transition entre l'état du Goban avant un coup et l'état du Goban après un coup sont très importantes. Ce sont elles qui permettent au système de comprendre pourquoi un coup atteint un but. La présence des prédicats donnant l'état avant le coup et l'état après le coup permet la coexistence en mémoire des deux états du Goban. Ainsi, il n'est pas nécessaire de détruire les faits concernant l'état du Goban avant le coup, et on peut avoir un ensemble de règles de transition déclaratif. La règle Règle_intersection_5 signifie qu'un coup légal sur une intersection rend cette intersection de la couleur du coup si cette intersection a au moins une voisine vide. La couleur d'une intersection avant le coup est donnée par le prédicat 'Couleur_intersection_avant (Intersection Couleur)' alors que la couleur d'une intersection après le coup est donnée par le prédicat 'Couleur_intersection_après (Intersection Couleur)'. La règle Règle_bloc_1 exprime la pérennité d'un bloc ayant strictement plus d'une liberté. Le prédicat

'Bloc_avant (Intersection Bloc)' signifie qu'une intersection appartient à un bloc. Si une intersection appartient à un bloc ayant strictement plus de 1 liberté avant le coup, elle appartiendra toujours à ce bloc après un coup. Un coup ne peut pas boucher plus d'une liberté d'un bloc. Là aussi, les prédicats 'Bloc_avant' et 'Bloc_après' coexistent en mémoire et permmettent d'avoir un théorie du domaine déclarative et donc des explications complètes.

Les règles de la théorie du domaine du jeu de Go sont de l'ordre de 300. Elles sont données dans l'annexe A. On peut toutefois les déduire à partir d'une définition concise des règles du jeu de Go [Moneret 1996].

Pour des raisons de place, je ne donne dans le tableau Explication Base Avant qu'un extrait de la base de faits correspondant au damier 4x4 de la figure Explication Goban Avant[4]. Cette base contient plus de 300 faits, c'est pourquoi je n'en donne qu'une partie afin de donner une idée de la façon dont est représenté un Goban.

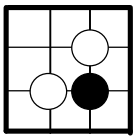


Figure Explication Goban Avant

La couleur amie est Noir, la couleur ennemie est Blanc. La couleur amie est représentée par le signe '@', la couleur ennemie est représentée par le signe 'O' et la couleur vide est représentée par le signe '+'. Chaque intersection a un numéro. L'intersection i1 est l'intersection en haut du damier à gauche, l'intersection i2 est l'intersection qui est à sa droite, et ainsi de suite pour les autres intersections. Les blocs sont numérotés selon un principe similaire. Le bloc b1 est le bloc de pierres le plus en haut à gauche. Le bloc b2 est le premier bloc que l'on trouve après b1 en parcourant les intersections dans l'ordre croissant, et ainsi de suite pour les autres blocs.

Le fait que '@' est une couleur est représenté par le fait 'Couleur (@)'. Les deux couleurs sont '@' et 'O', '+' n'est pas une couleur. Pour chaque intersection, un fait donne le nombre de voisines de l'intersection, par exemple le fait 'Nombre_voisines (i1 2)' indique que l'intersection i1 a deux voisines. Le fait 'Nombre_voisines_couleur_avant (i1 + 2)' indique que l'intersection i1 a deux voisines qui sont des intersections vides. Le fait 'Voisine (i1 i2)' représente l'information que l'intersection i1 est voisine de l'intersection i2. De manière générale le prédicat a un nom qui rappelle sa signification.

[4] Habituellement le jeu de Go se joue sur des damiers 9x9, 13x13 ou 19x19.

Couleur (@)	Pierre_voisine_bloc_avant (i7 b3)
Couleur (O)	Pierre_voisine_bloc_avant (i10 b3)
Couleur_opposees (O @)	Pierre_voisine_bloc_avant (i11 b1)
Couleurs_differentes (O @)	Pierre_voisine_bloc_avant (i11 b2)
Couleur_opposees (@ O)	Couleur_intersection_avant (i1 +)
Couleurs_differentes (@ O)	Couleur_intersection_avant (i2 +)
Couleurs_differentes (@ +)	Couleur_intersection_avant (i3 +)
Couleurs_differentes (+ @)	Couleur_intersection_avant (i4 +)
Couleurs_differentes (O +)	Couleur_intersection_avant (i5 +)
Couleurs_differentes (+ O)	Couleur_intersection_avant (i6 +)
Nombre_intersections_prises_avant (0)	Couleur_intersection_avant (i7 O)
Nombre_voisines (i1 2)	Couleur_intersection_avant (i8 +)
Nombre_voisines_couleur_avant (i1 + 2)	Couleur_intersection_avant (i9 +)
Nombre_voisines_couleur_avant (i1 @ 0)	Couleur_intersection_avant (i10 O)
Nombre_voisines_couleur_avant (i1 O 0)	Couleur_intersection_avant (i11 @)
Voisine (i1 i2)	Couleur_intersection_avant (i12 +)
Voisine_differente (i1 i2 i5)	Couleur_intersection_avant (i13 +)
Voisine (i1 i5)	Couleur_intersection_avant (i14 +)
Voisine_differente (i1 i5 i2)	Couleur_intersection_avant (i15 +)
Nombre_Blocs_voisins_avant (i1 0)	Couleur_intersection_avant (i16 +)
Nombre_Blocs_voisins_couleur_avant (i1 @ 0)	Tour_avant (O)
...	Coup (i6)

Tableau Explication Base Avant

4.1.3 La résolution de problèmes

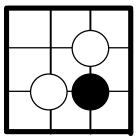


Figure Explication Hane Avant

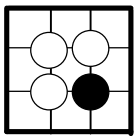


Figure Explication Hane Après

La résolution de problème est l'étape déductive qui consiste à déclencher les règles jusqu'à ce qu'on ne puisse plus déduire de nouveaux faits. Les faits déduits lors d'une résolution sont ceux qui représentent une position après qu'un coup ait été joué. Le tableau Explication Base Après donne les faits déduits sur l'exemple, à partir du tableau Explication Base Avant, en utilisant les règles du jeu du type des règles données dans le Tableau Explication Règles du Domaine. La totalité des règles de transition est donné dans l'annexe A. Encore une fois, le nombre de faits déduits sur cet exemple dépasse 300, je ne donne donc qu'un extrait des faits déduits.

Coup_legal_avant (i3 @)	Non_bloc_ote (b2)
Coup_legal_avant (i8 @)	Non_bloc_ote (b3)
Coup_legal_avant (i9 @)	Nombre_blocs_otes (0)
Coup_legal_avant (i14 @)	Fusion_blocs (b1 b2)
Coup_legal_avant (i1 @)	Fusion_blocs (b2 b1)
Coup_legal_avant (i2 @)	Non_fusion_blocs (b1 b3)
Coup_legal_avant (i4 @)	Non_fusion_blocs (b2 b3)
Coup_legal_avant (i5 @)	Non_fusion_blocs (b3 b1)
Coup_legal_avant (i12 @)	Non_fusion_blocs (b3 b2)
Coup_legal_avant (i13 @)	Couleur_intersection_apres (i1 +)
Coup_legal_avant (i15 @)	Couleur_intersection_apres (i2 +)
Coup_legal_avant (i16 @)	Couleur_intersection_apres (i3 +)
Coup_legal_avant (i6 @)	...
Non_bloc_ote (b1)	Coup_legal_apres (i16 O)

Tableau Explication Base Après

Dans l'exemple, le système a déduit que les coups amis sur les intersections i1, i2, i3, i4, i5, i6, i8, i9, i12, i13, i14, i15 et i16 sont des coups légaux. Il a aussi déduit que les blocs b1, b2 et b3 n'étaient pas ôtés après le coup blanc, que les blocs b1 et b2 fusionnaient alors que les blocs b1 et b3 et les blocs b2 et b3 ne fusionnaient pas, que les intersections i1, i2 et i3 étaient vides après le coup, ainsi que beaucoup d'autres faits de ce style que je ne donne pas ici pour des raisons de place.

4.1.4 Choix des faits à expliquer

Une fois les déductions sur l'état de la position après le coup faites, il est temps de choisir les faits qui sont intéressants à expliquer. Un ensemble spécial de règles de la théorie du domaine est spécialisé dans la sélection des faits intéressants à expliquer. Cet ensemble est l'ensemble des règles de régression. La régression d'un fait concernant l'état d'un jeu après le coup consiste à donner l'état du jeu avant le coup. Par exemple si un jeu est dans l'état G après un coup et que le système n'a pas déduit avant le coup qu'il était dans l'état G ou dans l'état GI, alors il est intéressant de régresser le jeu. S'il est dans l'état G après le coup et que le coup est de la couleur associée au but, il est dans l'état GI avant le coup. Certains faits sont intéressants à régresser, d'autre pas. **Les faits intéressants à régresser sont les faits nouveaux que le système n'avait pas prévu avant le coup, les faits inintéressants sont ceux concernant l'évolution normale prévue par le système des jeux combinatoires à valeurs inconnues**.

```
( nom          (   Regle_regression_jeu_binaire_1 )
  premisses
    ( present ( Tour_avant ( ?c ) )
      present ( Coup ( ?i ) )
      present ( Jeu_binaire_bloc_apres ( ?c Connecter ?b ?b1 G ) )
      absent ( Jeu_binaire_bloc_avant ( ?c Connecter ?b ?b1 G ) )
      absent ( Coup_jeu_binaire_bloc_avant ( ?c Connecter ?b ?b1 ?i GI ) )
      absent ( Coup_jeu_binaire_bloc_regresse_avant ( ?c Connecter ?b ?b1 ?i GI ) )
    )
conclusions
   ( ajoute ( Coup_jeu_binaire_bloc_regresse_avant ( ?c Connecter ?b ?b1 ?i GI ) ) )
```

Tableau Explication Règle Régression Coup Gagnant

Dans l'exemple dont les faits après le coup sont dans le tableau Explication Base Après, la règle de régression du tableau Explication Règle Régression Coup Gagnant s'applique et permet de déduire le fait 'Coup_jeu_binaire_bloc_régréssé_avant (O Connecter b1 b2 i6 GI)'. Ce fait est intéressant car avant le coup ennemi en i6, le programme n'avait pas déduit que les blocs b1 et b2 étaient connectables, ce qui est traduit dans la règle de régression par la prémisse 'absent (Coup_jeu_binaire_bloc_avant (?c Connecter ?b ?b1 ?i GI))'. Le programme n'avait pas non plus déduit que les deux blocs étaient connectés, ce qui est traduit dans la règle de régression par la prémisse 'absent (Jeu_binaire_bloc_avant (?c Connecter ?b ?b1 G))'. Cette règle permet donc de choisir d'expliquer les faits qui sont nouveaux ou surprenants, à savoir les faits dont la déduction n'avait pas été prévue avant le coup.

```
( nom          (   Regle_regression_jeu_binaire_2 )
  premisses    (   present ( Tour_avant ( ?c ) )
                   present ( Couleur ( ?c ) )
                   present ( Couleur_opposees ( ?c ?c1 ) )
                   present ( Coup ( ?i ) )
                   present ( Coup_force_jeu_binaire_bloc_avant ( ?c Deconnecter ?b ?b1 ?i IP ) )
                   present ( Coup_jeu_binaire_bloc_apres ( ?c1 Connecter ?b ?b1 ?i1 GI ) )
                   blocs_differents ( ?b ?b1 )
                   absent ( Jeu_binaire_bloc_regresse_avant ( ?c1 Connecter ?b ?b1 G ) )
                   absent ( Jeu_binaire_bloc_regresse_avant ( ?c1 Connecter ?b1 ?b G ) )
                   absent ( Jeu_binaire_bloc_avant ( ?c1 Connecter ?b ?b1 G ) )
               )
  conclusions (  ajoute ( Jeu_binaire_bloc_regresse_avant ( ?c1 Connecter ?b ?b1 G ) ) )
)
```

Tableau Explication Règle Régression Jeu Gagné

Les règles de régression ne se limitent pas à découvrir les faits intéressants sur les buts GI. Elles permettent aussi de sélectionner les faits concernant les jeux Gagné (G). Par exemple, le tableau

Explication Règle Régression Jeu Gagné donne la règle qui permet de découvrir que deux blocs étaient connectés avant le coup. Pour découvrir cela, la règle commence par vérifier que le coup joué était un coup forcé pour déconnecter les deux blocs avec la prémisse ‘present (Coup_force_jeu_binaire_bloc_avant (?c Deconnecter ?b ?b1 ?i IP))’. Puis elle vérifie que les deux blocs sont toujours connectables bien que le coup forcé ait été joué, avec la prémisse ‘present (Coup_jeu_binaire_bloc_apres (?c1 Connecter ?b ?b1 ?i1 GI))’. Les deux blocs ne peuvent donc pas être déconnectés puisqu’il n'y avait qu’un coup forcé pour les déconnecter et qu’après ce coup forcé, il peuvent toujours être connectés. Les prémisses suivantes sont destinées à ne pas déduire un jeu inintéressant. La prémisse ‘blocs_differents (?b ?b1)’ vérifie que si les deux blocs sont différents, il n’est pas intéressant de déduire un jeu sur la connexion d’un bloc à lui-même. Les prémisses ‘absent (Jeu_binaire_bloc_regresse_avant (?c1 Connecter ?b ?b1 G))’ et sa symétrique vérifient que le fait intéressant ou son symétrique n’ont pas déjà été déduits. La prémisse ‘absent (Jeu_binaire_bloc_avant(?c1 Connecter ?b ?b1 G))’ vérifie que le système ne savait pas, avant la résolution de problème, que les deux blocs étaient connectés. La sélection de ce fait pour l’explication amènera à l’explication et à l’apprentissage d’une règle que le système n’a pas encore. Elle lui permettra donc de s’améliorer automatiquement.

```
( nom           ( Regle_regression_jeu_5 )
 premisses      ( present ( Tour_avant ( ?c ) )
                  present ( Couleur ( ?c ) )
                  present ( Couleur_opposees ( ?c ?c1 ) )
                  present ( Coup ( ?i ) )
                  present ( Coup_jeu_bloc_avant ( ?c1 Prendre ?b ?i1 GI ) )
                  absent ( Coup_jeu_bloc_regresse_avant ( ?c Vivre ?b ?i IP ) )
                  absent ( Coup_jeu_bloc_avant ( ?c Vivre ?b ?i IP ) )
                  absent ( Jeu_bloc_avant ( ?c1 Prendre ?b G ) )
                  absent ( Coup_jeu_bloc_apres ( ?c1 Prendre ?b ?i2 GI ) )
                  absent ( Jeu_bloc_apres ( ?c1 Prendre ?b G ) )
                  regle_instanciee ( ?r )
                  action ( ?r ajoute ( Coup_jeu_bloc_avant ( ?c1 Prendre ?b ?i1 GI ) ) )
                  condition ( ?r ?p )
                  enleve_metapredicat ( ?p ?p1 )
                  present ( fait_modifie ( ?p1 ) )
                )
 conclusions    (  ajoute ( Coup_jeu_bloc_regresse_avant ( ?c Vivre ?b ?i IP ) ) )
)
```

Tableau Explication Règle Régression Coup Forcé

Le tableau Explication Règle Régression Coup Forcé donne une règle qui permet de créer des règles qui trouvent les coups forcés pour faire Vivre un bloc de pierres. Pour cela, la règle vérifie qu’avant le coup, l’ennemi pouvait prendre le bloc (jeu GI) et que le bloc n’était pas pris (jeu G). Elle vérifie ensuite qu’après le coup, il n’y a pas de règles concluant sur la prise du bloc. L’explication consiste à trouver une prémisse de la règle concluant sur le jeu GI qui est vraie avant le coup et qui a été modifiée après le coup. Pour cela, le système dispose des faits qui ont été modifiés par le coup. ces faits ont été repérés par des règles et mis dans la base de fait à l'interieur du métaprédicat fait_modifié. Pour chaque condition de la règle instanciées observée, on prend une de ses conditions dans la variable ?p de type Prédicat. Cette condition est de la forme 'métaprédicat (prédicat ())'. La prémisse enleve_metaprédicat (?p ?p1) enlève les métaprédicat de ?p et met le résultat dans ?p1. Le prédicat contenu dans la variable ?p1 est donc de la forme 'prédicat ()'. Pour vérifier que ce prédicat a été modifié par le coup, la règle vérifie ensuite que le fait 'fait_modifie (prédicat ())' est présent dans la base de faits.

```
(       nom             (       Regle_regression_jeu_6 )
        premisses       (       present ( Tour_avant ( ?c ) )
                                present ( Couleur ( ?c ) )
                                present ( Couleur_opposees ( ?c ?c1 ) )
                                present ( Coup ( ?i ) )
                                present ( Liberte_bloc_avant ( ?i ?b ) )
                                present ( Coup_jeu_bloc_apres ( ?c1 Prendre ?b ?i1 GI ) )
                                absent ( Coup_jeu_bloc_regresse_avant ( ?c Vivre ?b ?i PI ) )
                                absent ( Coup_jeu_bloc_avant ( ?c Vivre ?b ?i PI ) )
                                absent ( Jeu_bloc_avant ( ?c1 Prendre ?b G ) )
                                absent ( Coup_jeu_bloc_avant ( ?c1 Prendre ?b ?i2 GI ) )
                                absent ( Nouveau_bloc ( ?b ) )
                        )
        conclusions     (       ajoute ( Coup_jeu_bloc_regresse_avant ( ?c Vivre ?b ?i PI ) )
                        )
)
```

Tableau Explication Règle Régression Coup Perdant

Le tableau Explication Règle Régression Coup Perdant permet de créer des règles qui donnent des coups qui font perdre des blocs. Avant le coup il n'y a aucune règle concluant sur la prise du bloc, mais après le coup ami, l'ennemi peut prendre le bloc. Cette règle permet de détecter les coups qui ont une composante néfaste. Toutefois un coup qui fait mourir un bloc ami peut aussi tuer un groupe ennemi plus gros que le bloc ami, et donc finalement être bénéfique.

Les coups perdants sont aussi utilisés pour régresser les jeux Gagné (G). Si tous les coups forcés pour empêcher d'atteindre un but sont perdants, alors le jeu associé au but est Gagné (G).

```
( nom           (   Regle_regression_jeu_binaire_intersection_1 )
  premisses
    ( present ( Tour_avant ( ?c ) )
      present ( Coup ( ?i1 ) )
      present ( Bloc_après ( ?i1 ?b ) )
      present ( Jeu_binaire_intersection_intersection_apres ( ?c Connecter ?b ?i GI ) )
      absent ( Coup_jeu_binaire_ intersection_intersection _avant ( ?c Connecter ?i1 ?i ?i1 GIII ) )
      absent ( Coup_jeu_binaire_bloc_intersection_avant ( ?c Connecter ?b ?i ?i1 GI ) )
      absent ( Coup_jeu_binaire_ intersection_intersection _regresse_avant ( ?c Connecter ?i1 ?i ?i1 GIII ) )
    )
conclusions
   ( ajoute (Coup_jeu_binaire_ intersection_intersection _regresse_avant ( ?c Connecter ?i1 ?i ?i1 GIII ) ) )
```

Tableau Explication Règle Régression Menace

Le tableau Explication Règle Régression Menace donne une règle qui permet de détecter les conclusions sur les menaces qui n'avaient pas été prévues avant le coup. Le système se rend compte qu'un jeu est GI après le coup et qu'il était donc GIII avant le coup. Les prémisses commençant par le métaprédicat 'absent' vérifient que le fait régressé est nouveau et intéressant (le jeu n'était pas GI avant).

Les règles concernant les menaces sont très utilisées. Les menaces de connecter deux intersections vides sont utilisées pour faire une approximation du territoire créé et enlevé par un coup (voir le chapitre sur le programme de Go). Les menaces concernant les autres buts sont utilisées pour permettre de développer les noeuds OU de l'arbre ET/OU de recherche lorsque le programme résout un problème en phase de jeu. Pour chaque noeud OU, le programme essaie toutes les menaces d'atteindre le but recherché.

```
( nom ( Regle_regression_coups_forces_1 )
premisses  (
present ( Tour_avant ( ?c ) )
present ( Couleur ( ?c ) )
present ( Couleur_opposees ( ?c ?c1 ) )
present ( Coup ( ?i ) )
present ( Coup_jeu_binaire_bloc_intersection_avant ( ?c1 Faire_oeil ?b ?i1 ?i2 GI ) )
absent ( Jeu_binaire_bloc_intersection_avant ( ?c1 Faire_oeil ?b ?i1 G ) )
absent ( Coup_jeu_binaire_bloc_intersection_apres ( ?c1 Faire_oeil ?b ?i1 ?i3 GI ) )
absent ( Jeu_binaire_bloc_intersection_apres ( ?c1 Faire_oeil ?b ?i1 G ) )
absent ( Coup_force_jeu_binaire_bloc_intersection_regresse_avant ( ?c Empecher_oeil ?b ?i1 ?i IP ) )
absent ( Coup_force_jeu_binaire_bloc_intersection_avant ( ?c Empecher_oeil ?b ?i1 ?i IP ) )
liste_regles_instanciees_concluant_sur
        ( ?liste ajoute ( Coup_jeu_binaire_bloc_intersection_avant ( ?c1 Faire_oeil ?b ?i1 ?i2 GI ) ) )
intersection_liste_conditions_regles ( ?liste1 ?liste )
liste_coups_modifiant_conditions ( ?liste2 ?liste3 ?liste1 )
taille_liste ( ?liste2 1 )
)
conclusions (
ajoute ( Coup_force_jeu_binaire_bloc_intersection_regresse_avant ( ?c Empecher_oeil ?b ?i1 ?i IP ) ) ) )
```

Tableau Explication Règle Régression Tous les Coups Forcés

Le tableau Explication Règle Régression Tous les Coups Forcés donne un exemple qui permet de trouver tous les coups forcés pour empêcher un oeil à partir des règles sur les yeux qui ont été instanciées sur l'exemple. Pour cela, la règle commence par vérifier qu'il était possible pour l'ennemi de donner un oeil au bloc ?b sur l'intersection ?i1 en jouant sur l'intersection ?i2, cette information est représentée par la prémisse 'present (Coup_jeu_binaire_bloc_intersection_avant (?c1 Faire_oeil ?b ?i1 ?i2 GI))'. La règle vérifie ensuite que l'oeil n'était pas gagné avant le coup avec la prémisse 'absent (Jeu_binaire_bloc_intersection_avant (?c1 Faire_oeil ?b ?i1 G))'. Elle vérifie alors, qu'après le coup ami, il n'y a plus de possibilité connue pour l'ennemi de faire un oeil, avec les prémisses 'absent (Coup_jeu_binaire_bloc_intersection_apres (?c1 Faire_oeil ?b ?i1 ?i3 GI))' et 'absent (Jeu_binaire_bloc_intersection_apres (?c1 Faire_oeil ?b ?i1 G))'. Si aucune règle n'a conclut sur le fait que ce coup est forcé pour empêcher l'oeil (les prémisses 'absent (Coup_force_jeu_binaire_bloc_intersection_regresse_avant (?c Empecher_oeil ?b ?i1 ?i IP))' et 'absent (Coup_force_jeu_binaire_bloc_intersection_avant (?c Empecher_oeil ?b ?i1 ?i IP))'), la règle récupère la liste de toutes les règles concluant sur le fait que l'oeil était possible avant le coup avec la prémisse 'liste_regles_instanciees_concluant_sur (?liste ajoute (Coup_jeu_binaire_bloc_intersection_avant (?c1 Faire_oeil ?b ?i1 ?i2 GI)))'. La liste des règles instanciées concluant sur l'oeil avant se trouve alors dans la variable ?liste. L'étape suivante est de faire l'intersection de toutes les prémisses de ces règles concluant sur le fait qui a été invalidé par le coup en utilisant l'instruction 'intersection_liste_conditions_regles (?liste1 ?liste)'. L'intersection des prémisses se trouve alors dans la variable ?liste1.

```
// On ne peut pas changer la topologie : Voisine

( nom ( Regle_coups_modifiant_1 )
premisses (
regle ( ?r )
condition ( ?r present ( Voisine ( ?var ?var1 ) ) )
 )
conclusions ( nombre_de_coups_modifiant ( ?r present ( Voisine ( ?var ?var1 ) ) 0 ) ) )
```

Tableau Explication Règle coup modifiant

On appelle ensuite une base de règles spécialisée qui trouve tous les coups susceptibles de modifier les prémisse de ?liste1 en utilisant l'instruction 'liste_coups_modifiant_conditions (?liste2 ?liste3 ?liste1)'. Le tableau Explication Règle coup modifiant donne un exemple d'une règle de cette base. Puis, on vérifie que le nombre de coups forcés est de 1. On peut alors déduire qu'il n'y a qu'un coup forcé possible pour Empêcher l'oeil du bloc ?b sur l'intersection ?i1, c'est le coup de couleur ?c sur l'intersection ?i.

Le système dispose de règles qui lui permettent de trouver tous les coups forcés sur une position à partir des règles qu'il connaît déjà et qui ont été instanciées sur la position. Le système dispose de plusieurs règles de ce type pour chaque but.

La capacité à trouver tous les coups forcés pour empêcher l'achèvement d'un but est une capacité très importante de mon système d'apprentissage. Elle permet en effet de régresser les coups qui tentent d'empêcher d'atteindre un but. Les règles qui donnent tous les coups forcés sont essentiellement utilisées pour régresser les jeux Gagné (G). Si tous les coups forcés pour empêcher l'achèvement d'un but sont Perdant (PI), alors le jeu est Gagné (G).

Ces règles s'enchaînent pour prévoir autant de coups à l'avance que l'on veut (si on dispose d'assez de mémoire pour stocker les règles et d'assez de temps pour les matcher). Les règles sur les jeux G permettent de déduire les règles sur les jeux GI, les règles sur les jeux GI permettent de déduire les règles sur les jeux IP et PI, et les règles sur les jeux IP et PI permettent de déduire des règles sur les jeux G. Les règles déduites par l'utilisation des autres règles permettent de prévoir un coup de plus à l'avance.

De plus, les règles qui donnent l'ensemble des coups forcés sont utilisées pour développer les noeuds ET de l'arbre ET/OU de recherche lorsque le programme résout un problème en phase de jeu. Pour chaque noeud ET, le programme essaie l'ensemble minimal des coups forcés qui peuvent empêcher d'atteindre le but recherché par l'adversaire.

4.1.5 Mécanisme de retour dans la trace

Le module explicatif repère les faits concluant sur un jeu après le coup. Dans notre cas il repère le fait 'Coup_jeu_binaire_bloc_regresse_avant (@ Connecter b2 b3 i11 GI)'. Son but est alors de créer une règle qui explique pourquoi ce fait est présent uniquement en fonction de faits présents avant le coup joué et du fait représentant le coup joué lui-même. Pour cela, le module explicatif remonte la trace des règles déclenchées en remplaçant au fur et à mesure les faits décrivant un état après le coup joué par des faits décrivant l'état avant le coup joué. L'explication du fait 'Coup_jeu_binaire_bloc_regresse_avant (@ Connecter b2 b3 i11 GI)' est indiquée dans le tableau Explication Sous Ensemble de Faits.

Le tableau Explication Règles Déclenchées donne la trace du mécanisme d'explication qui fait un retour arrière dans la trace de la résolution de problème. Pour chaque fait nouveau ajouté à l'explication, il regarde si ce fait a lui-même été déduit par une règle de la théorie du domaine déclenchée avant la règle utilisant le fait en prémisse. Si c'est le cas, il ôte le fait et le remplace par les prémisses de la règle ayant permis de le déduire. Ainsi, par exemple, la prémisse 'present (Non_bloc_ote (b2))' qui a été ajoutée au cours de l'explication est ôtée de l'explication et remplacée par les prémisses de la règle Regle_bloc_ote_6 (les règles de la théorie du domaine ont un numéro, non pas pour que le système les ordonnent mais simplement pour que je puisse les retrouver facilement, la règle numéro 6 peut très bien être déclenchée avant la règle numéro 5). Après ce retour dans la trace, on obtient les faits du tableau Explication Sous Ensemble de Faits qui expliquent en fonction de prédicats avant le coup pourquoi le coup permet de connecter les deux blocs.

```
( nom        ( Regle_regression_jeu_binaire_1 )
 premisses ( present ( Tour_avant ( @ )  )
                present ( Couleur ( @ )  )
                present ( Coup ( i11 )  )
                present ( Jeu_binaire_bloc_apres ( @ Connecter b2 b3 G )  )
                blocs_differents ( b2 b3 ) )
 conclusions ( ajoute ( Coup_jeu_binaire_bloc_regresse_avant ( @ Connecter b2 b3 i11 GI ) ) ) )

( nom        ( Regle_jeu_apres_6 )
 premisses ( present ( Couleur ( @ )  )
                present ( Fusion_blocs ( b2 b3 )  )
                present ( Couleur_bloc_avant ( b2 @ )  ) )
  conclusions   ( ajoute ( Jeu_binaire_bloc_apres ( @ Connecter b2 b3 G ) ) ) )

( nom        ( Regle_fusion_blocs_1 )
 premisses     ( present ( Tour_avant ( @ )  )
                   present ( Coup ( i11 )  )
                   present ( Liberte_bloc_avant ( i11 b2 )  )
                   present ( Couleur_bloc_avant ( b2 @ )  )
                   present ( Non_bloc_ote ( b2 )  )
                   present ( Liberte_bloc_avant ( i11 b3 )  )
                   blocs_differents ( b2 b3 )
                   present ( Couleur_bloc_avant ( b3 @ )  ) )
  conclusions   ( ajoute ( Fusion_blocs ( b2 b3 ) ) ) )

( nom        ( Regle_bloc_ote_6 )
 premisses     ( present ( Nombre_libertes_bloc_avant ( b2 3 )  )
                   superieur ( 3 1 )
                 )
  conclusions   ( ajoute ( Non_bloc_ote ( b2 ) ) ) )
```

Tableau Explication Règles Déclenchées

```
present ( Tour_avant ( @ )  )
present ( Couleur ( @ )  )
present ( Coup ( i11 )  )
blocs_differents ( b2 b3 )
present ( Couleur_bloc_avant ( b2 @ )  )
present ( Liberte_bloc_avant ( i11 b2 )  )
present ( Liberte_bloc_avant ( i11 b3 )  )
present ( Couleur_bloc_avant ( b3 @ )  )
present ( Nombre_libertes_bloc_avant ( b2 3 )  )
superieur ( 3 1 cte )
ajoute ( Coup_jeu_binaire_bloc_regresse_avant ( @ Connecter b2 b3 i11 GI )  )
```

Tableau Explication Sous Ensemble de Faits

4.1.6 Influence de la théorie du domaine sur la généralité des explications

Les prédicats utilisés pour représenter la théorie du domaine ont une influence sur la généralité des explications. Je vais illustrer ceci par deux exemples.

Exemple 1

On peut représenter le fait qu'une intersection ?i a une voisine vide de deux façons différentes :

- Façon plus générale :

Voisine (?i ?i1)
Couleur_intersection_avant (?i1 +)

- Façon moins générale :

Au_dessus (?i ?i1) Couleur_intersection_avant (?i1 +)	A_droite (?i ?i1) Couleur_intersection_avant (?i1 +)
En_dessous (?i ?i1) Couleur_intersection_avant (?i1 +)	A_Gauche (?i ?i1) Couleur_intersection_avant (?i1 +)

La deuxième façon a l'avantage de créer des explications plus rapides à vérifier et correspond à une représentation à base de patterns géométriques. Toutefois la première représentation permet de recouvrir beaucoup plus de cas avec une seule explication.

Exemple 2

On peut représenter le fait qu'un bloc ?b a un nombre minimum de libertés de deux façons différentes :

- Façon plus générale :

Min_libertés_bloc (?b 2)

- Façon moins générale :

Nombre_libertés_bloc (?b ?n)
supérieur (?n 2)

Il est beaucoup moins coûteux et beaucoup plus général d'expliquer le fait 'Min_libertés_bloc (?b 2)' que d'expliquer les deux faits 'Nombre_libertés_bloc (?b ?n)' et 'supérieur (?n 2)'. En effet, expliquer le fait 'Min_libertés_bloc (?b 2)' revient à trouver deux libertés différentes pour le bloc ?b alors qu'expliquer le fait 'Nombre_libertés_bloc (?b ?n)' demande de trouver et de compter toutes les libertés du bloc ?b.

En résumé, la généralité des explications dépend de la généralité de la théorie du domaine. Dans [Moneret 1996], R. Moneret donne une méthode qui permet d'engendrer automatiquement les règles qui donnent le damier après le coup en fonction du damier avant le coup. Il crée ces règles à partir d'une définition générale et concise des règles du jeu de Go. Cette recherche est une continuation logique de la recherche entreprise avec Introspect. En dehors du travail de programmation d'Introspect qui est assez important mais indépendant du domaine, le plus gros du travail dans un domaine d'application d'Introspect est de créer la base des règles de transition. Un programme qui engendre automatiquement de telles bases à partir d'une définition de très haut niveau des règles du domaine permet donc de réduire le temps nécessaire à l'adaptation d'Introspect à de nouveaux domaines. La problématique essentielle qui intervient, lorsqu'on spécialise automatiquement une définition générale d'un domaine pour créer des règles de transition pour Introspect, est de créer des règles de transition qui contiennent des prédicats ayant un bon niveau de généralité. Introspect sait se placer a deux niveaux différents de représentation de la connaissance, suivant qu'il veut filtrer efficacement les règles, ou qu'il veut apprendre des règles générales. Il serait très intéressant d'automatiser les choix dans les représentations des connaissances pour qu'un système puisse de lui même créer ces deux niveaux. Ce travail a déjà été abordé dans [Cheng 1995], mais c'est un domaine qui mérite de plus amples recherches. Actuellement dans Introspect, c'est moi qui ait défini ces deux niveaux.

4.2 Les explications stratégiques

J.M. Nigro et moi-même, en comparant les modules explicatifs de mon système avec le sien [Nigro 1995, 1996], avons construit un modèle général de sélection des coups et des buts dans les jeux. Ce modèle est composé de méthodes de sélection. A chaque méthode de sélection est associé une méthode d'explication. Le but de ce chapitre est de présenter un ensemble de méthodes de sélection et d'explication qui peuvent être utilisées dans tous les jeux. Nous les avons appliquées au jeu de Tarot[5], au jeu d'Echecs et au jeu de Go.

Nos méthodes de sélection et d'explication sont basées sur une structure des connaissances du domaine des jeux décrite dans le chapitre sur la représentation des connaissances.

Les explications que mon système donne à son utilisateur portent sur les raisons pour lesquelles un coup a été choisi. Elles donnent les buts principaux que le coup permet d'atteindre. Mon système n'utilise que la méthode de sélection compensatoire pour ses choix stratégiques et les explications qu'il donne sont basées sur cette méthode. La méthode de sélection compensatoire fait partie d'un arsenal de méthodes de sélection que je n'utilise pas toutes mais que je présente aussi ici car l'ensemble de ces méthodes forme un tout cohérent.

4.2.1 Les explications positives et négatives

Notre but est de modéliser les différentes méthodes de sélection et de montrer leurs avantages au niveau des explications. Certains systèmes d'ailleurs effectuent déjà des explications à partir des méthodes de sélection. En effet, le système Bateleur, qui est capable de simuler un joueur de Tarot (un jeu de cartes), utilise très souvent la méthode lexicale. Il permet au système Genecom [Nigro 1995] d'engendrer des explications en fonction des méthodes de sélection utilisées par Bateleur.

Une méthode de sélection peut être utilisée par un système à base de règles. Par exemple, Bateleur a des règles du type "Si *but = Jouer les cartes perdantes* et *le joueur doit fournir à Coeur* alors *employer la méthode lexicale* ($C_1..C_p$)". Dans ce cas les techniques d'explications propres aux méthodes de sélection peuvent compléter les techniques déjà existantes.

La méthode de sélection peut également être employée indépendamment des systèmes à base de règles. Dans ce cas, les techniques d'explication propres aux méthodes de sélection doivent être appliquées. Par exemple, Gogol [Nigro & Cazenave 1996] utilise la méthode compensatoire élitiste et engendre des explications.

Les explications positives expliquent pourquoi un choix a été fait alors que les explications négatives expliquent pourquoi un choix n'a pas été fait. Les explications négatives ont été abordées dans [Safar 1987] pour la logique des propositions et dans [Jimenez Dominguez 1990] pour la logique des prédicats associée à une représentation objet.

Certaines personnes se sont intéressées à la caractérisation des faits et des règles [Wallis et Shortliffe 1984]. Il est, en effet, très utile de connaître la complexité et l'importance d'une règle ou d'un fait. Il est inutile d'expliquer un fait trop complexe ou trop évident. Cette technique pourrait très bien compléter notre approche.

4.2.2 Définition des explications à partir des méthodes de sélection

[5] Le jeu du Tarot est un jeu de cartes, le but du jeu est de gagner un certain nombre de points (chaque carte ayant une valeur).

La section reprend les différentes méthodes de sélection. Pour chaque méthode, un cadre d'une explication positive et négative est proposé. Tous les cas présentés sont illustrés par un exemple relatif à la sélection d'un élève par un professeur. Dans cette section, le cas de l'imbrication de plusieurs méthodes n'est pas abordé car le problème est véritablement trop complexe : il est difficile de trouver un cadre général englobant tous les cas particuliers dérivés de cette imbrication de méthodes.

Soit $\{C_1..C_p\}$ l'ensemble des contraintes associées à une méthode (conjonctive, disjonctive ou lexicale). Rappelons que $\{C_1..C_p\}$ est **ordonné** pour la méthode lexicale.

Considérons la fonction de filtrage g : $\chi \rightarrow P(\alpha)$

$Ct \rightarrow \{A_1..A_v\}$ tel que $\forall A_i \in \{A_1..A_v\}$, $f(A_i, C_t) = A_i$

Reprenons l'exemple du professeur qui cherche à sélectionner des élèves (voir le chapitre sur la représentation des connaissances) :

	Jean	Paul	Jacques
Mathématiques	8	5	15
Informatique	13	7	17
Français	18	3	10

L'application de la fonction g*(Informatique ≥ 10)* renvoie l'ensemble *{Jean, Jacques}* car les deux élèves vérifient la contrainte.

Soit $\mathbf{A}_E$ l'individu sur lequel les explications seront orientées.
Deux cas de figures sont possibles :

+) $A_E \in \{A_1..A_u\}$ il faut alors donner une information de type explication positive
-) $A_E \notin \{A_1..A_u\}$ il faut alors donner une information de type explication négative

a) La méthode conjonctive :

Les exemples d'explication positive et négative se baseront sur le tableau suivant :

méthode $(C_1..C_p)$	$(g(C_1)..g(C_p))$
Conjonctive ((Mathématiques ≥ 10) (Informatique ≥ 10) (Français ≥ 10))	(Jacques) (Jean, Jacques) (Jean, Jacques) La sélection = (Jacques)

+) $A_E \in \{A_1..A_u\}$ a été choisi par la méthode car $\forall C_t \in \{C_1..C_p\}$, $A_E \in g(C_t)$

Dans l'exemple décrit ci-dessus A_E est remplacé par Jacques. L'explication positive répond à la question "*pourquoi Jacques a-t-il été sélectionné ?*". La réponse sera la suivante : "*Jacques a été sélectionné parce que ses notes sont supérieures à 10 en mathématiques, informatique et en français.*"

-) $A_E \notin \{A_1..A_u\}$ n'a pas été choisi par la méthode car $\exists C_t \in \{C_1..C_p\}$ / $A_E \notin g(C_t)$

Supposons que A_E soit remplacé par Jean. L'explication négative répond à la question "*pourquoi Jean n'a-t-il pas été sélectionné ?*". La réponse sera la suivante : "*Jean n'a pas été sélectionné car sa note de mathématiques n'est pas supérieure à 10 : sa note est de 8.*"

b) La méthode disjonctive :

Les exemples d'explication se baseront sur le tableau suivant :

méthode ($C_1..C_p$)	($g(C_1)..g(C_p)$)
Disjonctive ((Mathématiques ≥ 10) (Informatique ≥ 10) (Français ≥ 10))	(Jacques) (Jean, Jacques) (Jean, Jacques) La sélection = (Jean Jacques)

+) $A_E \in \{A_1..A_u\}$ a été choisi par la méthode car $\exists\ C_t \in \{C_1..C_p\}$ / $A_E \in g(C_t)$

Dans l'exemple décrit ci-dessus A_E est remplacé par Jacques ou Jean. L'explication positive répond à la question "*pourquoi Jean a-t-il été sélectionné ?*". La réponse sera la suivante : "*Jean a été sélectionné parce qu'il a une note supérieure à 10 en informatique et une note supérieure à 10 en français.*"

-) $A_E \notin \{A_1..A_u\}$ n'a pas été choisi par la méthode car $\forall\ C_t \in \{C_1..C_p\}$, $A_E \notin g(C_t)$

Supposons que A_E soit remplacé par Paul. L'explication négative répond à la question "*pourquoi Paul n'a-t-il pas été sélectionné ?*". La réponse sera la suivante : "*Paul n'a pas été sélectionné car il a moins de 10 (une note de 5) en mathématiques et moins de 10 (une note de 7) en informatique et moins de 10 (une note de 3) en français.*"

c) La méthode lexicale :

Etant donné la définition de la méthode lexicale, deux cas de figures sont possibles : soit toutes les contraintes sont traitées, soit la méthode n'a pas vérifié toutes les contraintes car elle a trouvé une solution unique au cours de sa sélection.

- <u>toutes les contraintes sont traitées</u>

les exemples d'explication se baseront sur le tableau suivant :

méthode ($C_1..C_p$)	($g(C_1)..g(C_p)$)
Lexicale ((Informatique ≥ 7) (Français ≥ 10) (Mathématiques ≥ 10))	(Jean, Paul, Jacques) (Jean, Jacques) (Jacques) La sélection = (Jacques)

+) *card{$A_1..A_u$}>1*, $A_E \in \{A_1..A_u\}$ a été choisi par la méthode car $\forall\ C_t \in \{C_1..C_p\}$, $A_E \in g(C_t)$
L'explication positive est identique à celle produite par la méthode conjonctive

-) $A_E \notin \{A_1..A_u\}$ n'a pas été choisi par la méthode car $\exists\ C_t \in \{C_1..C_p\}$ / $A_E \notin g(C_t)$

Supposons que A_E soit remplacé par Paul. L'explication négative répond à la question "*pourquoi Paul n'a-t-il pas été sélectionné ?*". La réponse sera la suivante : "*Paul n'a pas été sélectionné surtout*

parce qu' il a moins de 10 (une note de 7) en français et en plus il a moins de 10 (une note de 5) en mathématiques. Ce qui n'est pas le cas de Jacques."

- <u>toutes les contraintes n'ont pas été traitées</u>

les exemples d'explication se baseront sur le tableau suivant :

méthode (C_1..C_p)	($g_{(C_1)}$..$g_{(C_p)}$)
Lexicale ((Mathématiques ≥ 10) (Informatique ≥ 10) (Français ≥ 10))	(Jacques) pas traitée pas traitée La sélection = (Jacques)

+) *card{A_1..A_u}=card{A_E}=1*, A_E a été choisi par la méthode
car $\exists$ s $\in$ {1..p} / **(** $\forall$ tj $\leq$ s, $C_{tj} \in P(\chi)$, $f(A_O, C_{tj}) = A_E$ et
$\forall A_x \in \alpha$, $A_x \neq A_i$, $\exists$ tj $\leq$ s / $C_{tj} \in P(\chi)$, $f(A_x, C_{tj})$ = {} **)**

Dans l'exemple décrit ci-dessus A_E est remplacé par Jacques. L'explication positive répond à la question "*pourquoi Jacques a-t-il été sélectionné ?*". La réponse sera la suivante :

"*Jacques a été sélectionné parce qu'il a une note en mathématiques supérieure à 10 et qu'il est le seul dans ce cas.*"

-) *card{A_1..A_u}=card{A_x}=1, $A_E \neq A_x$*, A_E n'a pas été choisi par la méthode
car $\exists$ s $\in$ {1..p} / **(** $\forall$ tj $\leq$ s, $C_{tj} \in P(\chi)$, $f(A_x, C_{tj}) = A_x$ et
$\exists$ tj $\leq$ s / $C_{tj} \in P(\chi)$, $f(A_E, C_{tj})$ = {} **)**

Supposons que A_E soit remplacé par Jean. L'explication négative répond à la question "*pourquoi Jean n'a-t-il pas été sélectionné ?*". La réponse sera la suivante : "*Jean n'a pas été sélectionné car Jacques est le seul qui ai une note de mathématiques supérieure à 10 : sa note est de 15 alors que Jean n'a que 8.*"

d) la méthode compensatoire

- <u>La méthode compensatoire avec seuil</u> :

On a sélectionné les élèves parce que leurs notes pondérées était au-dessus d'un seuil. L'explication consiste à trouver les matières qui ont le plus influé sur la note.

CompensatoireOp ((Mathématiques 3)(Informatique 3)(Français 1)) Seuil = 70

Individu	$\sum (V_{i,j} * K_j)$	Sélection (S = 70)
Jean	81	oui
Paul	39	non
Jacques	106	oui

+) $A_E \in \{A_1..A_u\}$ a été choisi par la méthode car $\forall A_i \in \{A_1..A_u\}, \sum_{j=1}^{p} (V_{i,j} * K_j) \geq S$

Ce type de formule n'est pas très intéressant pour engendrer des explications. En effet, dire que "*Jean a été sélectionné parce que la somme de ses notes pondérées est supérieure à 70*" n'est pas très parlant. Pour la méthode compensatoire, il est plus judicieux de présenter le critère qui a été le plus influent sur la décision.

	Mathématiques * 3	Informatique * 3	Français * 1	$\sum (V_{i,j} * K_j)$
Jean	8*3=24	13*3=39	18*1=18	81

Dans le cas où A_E est remplacé par Jean, l'explication positive répond à la question "*pourquoi Jacques a-t-il été sélectionné ?*". La réponse sera axée sur le critère *Informatique* car c'est celui qui apporte le plus (39) au cumul des notes pondérées de Jean : "*Jean a été sélectionné en grande partie grâce à sa note en informatique (13).*"

-) $A_E \notin \{A_1..A_u\}$ n'a pas été choisi par la méthode car $\sum_{j=1}^{p} (V_{O,j} * K_j) < S$

Comme pour l'explication positive, la formule décrite ci-dessus n'est pas très intéressante pour une explication. Il est préférable de se focaliser sur le critère de A_E qui a été le plus pénalisant. Afin de simplifier la recherche de ce critère, nous supposerons que l'ensemble des critères évolue dans la même fourchette de valeurs (dans notre exemple scolaire, les notes vont de 0 à 20).

Soit m la valeur moyenne de chacun de critères, le critère B_s que l'individu A_E doit améliorer est celui dont la valeur a le plus gros écart négatif à la moyenne (l'écart est pondéré par le coefficient du critère) : $B_s \in \{B_1..B_p\} / \forall B_j \in \beta, [(V_{O,j} - m) * K_j] > [(V_{O,s} - m) * K_j]$.

	Mathématiques * 3	Informatique * 3	Français * 1	$\sum (V_{i,j} * K_j)$
Paul	5*3=15	7*3=21	3*1=3	39
écart pondéré	(5-10)*3 = **-15**	(7-10)*3 = -9	(3-10)*1 = 7	

Prenons le cas où A_E= Paul. La moyenne des notes m est égale à 10. Le critère Mathématiques est celui qui a le plus gros écart négatif. Ainsi, à la question "*pourquoi Paul n'a-t-il pas été sélectionné?*" le système répond "*Paul n'a pas été sélectionné car sa note de mathématiques l'a fortement pénalisée (il n'a eu que 5)*".

- <u>La méthode compensatoire élitiste</u> :

On a sélectionné les élèves qui ont eu les meilleurs notes pondérées. L'explication consiste à trouver les matières qui ont fait la différence avec les autres élèves.

+) $A_E \in \{A_1..A_u\}$ a été choisi par la méthode car $\forall A_{i2} \in \alpha, \sum_{k=1}^{p}(V_{O,k} * K_k) \geq \sum_{k=1}^{p} (V_{i2,k} * K_k)$

L'explication est identique à celle produite par la méthode compensatoire avec un seuil.

-) $A_E \notin \{A_1..A_u\}$ n'a pas été choisi par la méthode car
$$\forall A_{i1} \in \{A_1..A_u\}, \sum_{k=1}^{p}(V_{O,k} * K_k) < \sum_{k=1}^{p} (V_{i1,k} * K_k)$$

Comme pour les explications basées sur la méthode compensatoire avec seuil, la définition (décrite ci-dessus) n'est pas très parlante. Aussi, il est plus judicieux de trouver le critère qui a lésé l'individu A_E et qui a avantagé l'individu sélectionné A_c. Pour trouver ce critère B_s, il faut tout d'abord ordonner (par ordre décroissant) les critères suivant la valeur de $V_{i,j}*K_j$ pour les individus A_c et A_O. Il faut ensuite parcourir la liste des critères et rechercher le premier critère pertinent B_s qui vérifie la formule $V_{O,s}*K_s < V_{c,s}*K_s$.

	Critères ordonnés par ordre décroissant			
	Informatique * 3	Mathématiques * 3	Français * 1	$\sum (V_{i,j} * K_j)$
Paul	7*3=21	5*3=15	3*1=3	39

Jacques	17*3=51	15*3=45	10*1=10	106

Dans le cas où A_E= Paul et A_c= Jacques, le critère pertinent sera l'Informatique car c'est le premier critère classé et la note pondérée de Jacques (51) est supérieure à celle de Paul (21). Ainsi, à la question "*pourquoi Paul n'a-t-il pas été sélectionné?*" le système répond "*Paul n'a pas été sélectionné car il n'a pas été assez performant en informatique (7) à l'inverse de Jacques (17)*".

e) Une méta-méthode

méta-méthode	individus
lexicale ((disjonctive ((mathématiques ≥ 10) (informatique ≥ 10))) (conjonctive ((mathématiques ≥ 10) (informatique ≥ 10))))	(Jean, Jacques) (Jacques)

Le principe est le même que celui utilisé par une méthode sur des contraintes sauf qu'il s'agit là d'appliquer une méta-méthode sur des méthodes.

+) Supposons que A_E soit remplacé par Jacques. L'explication positive répond à la question "*pourquoi Jacques a-t-il été sélectionné ?*". La réponse sera la suivante : "*Jacques a été sélectionné parce qu'il a plus de 10 en mathématiques **et** en informatique.* ". Le terme ***et*** est employé car Jacques vérifie (C_1 ***ou*** C_2) et (C_1 ***et*** C_2) qui peut être simplifié en (C_1 ***et*** C_2).

-) Supposons que A_E soit remplacé par Jean. L'explication négative répond à la question "*pourquoi Jean n'a-t-il pas été sélectionné ?*". La réponse sera la suivante : "*Jean n'a pas été sélectionné parce qu'il n'a pas au moins 10 en mathématiques **et** en informatique (il a 8 en mathématiques) à l'inverse de Jacques*".

4.2.3 Applications dans le domaine des jeux

Les explications dans le domaine des jeux sont intéressantes pour les joueurs qui utilisent les systèmes. Elles leur permettent de comprendre leurs erreurs et ainsi de progresser plus rapidement. Elles sont également intéressantes pour les créateurs de systèmes de jeux. Elles leur permettent de comprendre les erreurs du système et de savoir comment y remédier.

a) Explication avec les méthodes conjonctive et disjonctive

Reprenons les exemples où Bateleur utilise les méthodes conjonctive et disjonctive. Les deux types d'explications (positive et négative) se greffent très bien sur l'application de ces deux méthodes au jeu du Tarot.

méthode (C_1..C_p)	($g(C_1)$..$g(C_p)$)
Conjonctive ((La couleur longue > 0) (Gagner le pli > 2) (Valeur > 1))	(R♣, C♣, 9♣, 8♣) (R♣, R♥) (R♣, C♣, R♥, V♥) La sélection = (R♣)

+) A la question "*pourquoi le R♣ a-t-il été joué?*", le système répond "*le R♣ a été joué car il fait partie de la couleur longue, il a beaucoup de chance de gagner le pli et c'est une carte de valeur (le R♣ vaut 4.5 points)*".

-) A la question "*pourquoi le 8♣ n'a-t-il pas été joué?*", le système répond "*le 8♣ n'a pas été joué car il a peu de chance de remporter le pli et c'est une carte sans valeur (le 8♣ vaut 0.5 point)*".

méthode (C_1..C_p)	($g(C_1)$..$g(C_p)$)
Disjonctive ((Gagner le pli < 0) (Valeur > 3))	(9♣, 8♣, 5♦) (R♣, R♥) La sélection = (9♣, 8♣, 5♦, R♣, R♥)

+) A la question "*pourquoi le R♥ a-t-il été joué?*", le système répond "*le R♥ a été joué car il vaut beaucoup de points (le R♥ vaut 4.5 points)*".

-) A la question "*pourquoi C♣ n'a-t-il pas été joué?*", le système répond "*le C♣ n'a pas été joué car il peut encore gagner un pli (Gagner le pli = 1) et il ne vaut pas au moins 3 points (à l'inverse d'un Roi ou d'une Dame)*".

b) Explications avec la méthode de sélection lexicale

+) En se référant à la figure Représentation lexicale, on obtient l'explication de la sélection du but :
"*Le but Contrôler le Centre a été sélectionné parce qu'il a une agressivité supérieure à 1, qu'il a une importance supérieure à 7 et un risque inférieur à 5 et qu'il est le seul dans ce cas.*"

-) Si on veut comparer les buts Capturer la Reine et Contrôler le Centre on obtient :
"*Le but Capturer la Reine n'a pas été sélectionné car il a un risque supérieur ou égal à 5 : son risque est de 8. Ce qui n'est pas le cas du but Contrôler le Centre : son risque est de 3.*"

c) Explications avec la méthode de sélection compensatoire

+) En se référant à la figure Représentation compensatoire, on obtient l'explication de la sélection du coup :
"*Le coup C a été sélectionné en grande partie grâce à son Influence Sauvée (12).*"

-) Si on veut comparer les coups B et C on obtient :
"*Le coup B n'a pas été sélectionné car le coup C est celui qui a une Influence Sauvée la plus grande : le coup B a une Influence prise de 3 mais ce n'est pas assez par rapport au coup C qui a une Influence Sauvée de 12.*"

4.3 Conclusion

Mon système comporte deux types très différents d'explications : les explications que le système se fait à lui-même et les explications que le système donne à l'utilisateur. Ces deux types d'explications sont différents non seulement par leur objectif mais aussi par les connaissances sur lesquelles elles sont basées. Les explications que le système se donne à lui-même sont basées sur une résolution de problème déductive en logique des prédicats alors que les explications données à l'utilisateur utilisent des méthodes de sélection à base de contraintes.

Pour ce qui est des explications que le système se donne à lui-même, j'ai introduit une représentation des connaissances basée sur la représentation explicite du temps, ce qui permet une représentation déclarative des règles du jeu. Les systèmes précédents d'explication dans le domaine des jeux utilisaient des règles du jeu procédurales ce qui les amenaient à se donner des explications fausses et donc à créer des règles fausses. Mon approche est plus déclarative, elle est nécessaire pour créer un système qui fait un amorçage. Un tel système utilise les connaissances qu'il a créées pour créer de nouvelles connaissances. Or, s'il créait des connaissances fausses à un moment donné, l'utilisation de ces connaissances fausses amènerait par la suite à créer de plus en plus de connaissances fausses. Un tel système arriverait à des conclusions contradictoires et finirait certainement par prendre souvent de mauvaises décisions. C'est pourquoi mon système ne crée que des connaissances vraies. Il choisit ce qu'il va expliquer en se basant sur la surprise que lui a apporté

ses déductions. Un fait est intéressant à expliquer et donc à apprendre si le système n'est pas arrivé à le prévoir avant la résolution d'un problème. Enfin, j'ai montré comment les prédicats utilisés pour représenter les règles du jeu ont une influence sur la généralité des explications.

Pour ce qui est des explications que le système donne à l'utilisateur, nous avons [Nigro 1996] défini des notions communes à beaucoup de jeux : les plans, les buts, les coups, la stratégie et la tactique. Nous avons répertorié différentes méthodes de sélection à partir de contraintes. Les méthodes de sélection sont faciles à implémenter et à utiliser. Elles sont générales et permettent de s'abstraire du système sur lequel portent les explications. Chacune des méthodes de sélection employée a été associée à une méthode d'explication. Les explications pouvant aussi bien porter sur les raisons du choix (explications positives) que sur les raisons du non-choix (explications négatives). Pour mieux illustrer mon propos, j'ai présenté de nombreux exemples de méthodes de sélections et de méthodes d'explications. Ils ont été illustrés principalement par deux systèmes opérationnels : Bateleur [Nifro 1995] pour le jeu de Tarot et Gogol pour le jeu de Go.

4.4 Bibliographie

[Beckstein 1996] C. Bekstein, R. Stolle, G. Tobermann, *Meta-Programming for Generalized Horn Clause Logic*. Proceedings of the 4th International Workshop on Metareasoning and Metaprogramming in Logic, Bonn, 1996.

[Cazenave 1995] T. Cazenave, *Learning and Problem Solving in Gogol : A Go Playing Program*. Rapport n° 95-10 du LAFORIA.

[Cazenave 1996a] T. Cazenave, *Learning to Forecast by Explaining the Consequences of Actions*. First International Workshop on Machine Learning, Forecasting and Optimization, Madrid, 1996.

[Cheng 1995] J. Cheng, *Management of Speedup Mechanisms in Learning Architecture.*. Ph.D. Thesis of Carnegie Mellon University, Janvier 1995.

[Clancey 1986] W. Clancey, *From GUIDON to NEOMYCIN and HERACLES in twenty short lessons*. AI Magazine, 7(3).

[Clancey 1987] W. Clancey, *Knowledge-Based Tutoring - The GUIDON program*. The MIT Press, Cambridge.

[Clark 1978] K. L. Clark, *Negation as failure*. In H. Gallaire and J. Minker, editors, Logic and Data Bases, pp. 293-322, Plenum Press, New York, 1978.

[Davis & al 1977] R. Davis, B. Buchanan, E. Shortliffe, *Production rules as a representation for a knowledge- based consultation program*, Artificial Intelligence 8(1).

[Gelfond 1989] M. Gelfond, H. Przymusinski, T. C. Przymusinski, *On the relationship between circumscription and negation as failure*, Journal of Artificial Intelligence, 38(1):75-94, February 1989.

[Hill 1994] P. M. Hill, J. W. Lloyd, *The Gödel Programming Language*, M.I.T. press, 1994.

[Jimenez Dominguez 1990] C. Jimenez Dominguez, *Sur l'explication dans les systèmes à base de règles : le système prose*. Thèse de l'Université Paris 6.

[McCarthy 1980] J. Mc Carthy, *Circumscription - a form of non-monotonic reasoning*. Journal of Artificial Intelligence, 13:27-39, 1980.

[Minker 1982] - J Minker. *On indefinite data bases and the closed world assumption*. In Proc. 6th Conference on Automated Deduction, pp. 292-308, Springer-Verlag, 1982.

[Minton 1984] - S. Minton. *Constraint-Based Generalization - Learning Game-Playing Plans from Single Examples*. Proceedings of the Fourth National Conference on Artificial Intelligence, 251-254. Los Altos, William Kaufmann, 1984.

[Minton 1988] - S. Minton. *Learning Search Control Knowledge - An Explanation Based Approach*. Kluwer Academic, Boston, 1988.

[Mitchell 1986] - T. M. Mitchell, R. M. Keller, S. T. Kedar-Kabelli. *Explanation-based Generalization : A unifying view*. Machine Learning 1 (1), 1986.

[Moneret 1996] - R. Moneret. *Mise à jour incrémentale des concepts du jeu de Go*. Rapport de stage du D.E.A. I.A.R.F.A., 1996.

[Nigro 1995] J.M. Nigro, *La conception et la réalisation d'un générateur automatique de commentaires : le système GénéCom*. *Application au jeu du Tarot*. Thèse de l'Université Paris 6.

[Nigro 1996] J.M. Nigro, T. Cazenave, *Constraint-based Explanations in Games*. IPMU96, Grenade.

[Ohlsson 1992] S. Ohlsson, *Constraint-Based Student Modelling*. Journal of Artificial Intelligence in Education (1992) 3 (4), 429-447.

[Pell 1993] - Barney Pell. *Logic Programming for General Game-Playing*. Proceedings of the workshop on Knowledge Compilation and Speedup Learning, at Machine Learning Conference, Amherst, Mass., 1993.

[Pitrat 1976] - J. Pitrat. *Realization of a Program Learning to Find Combinations at Chess*. Computer Oriented Learning Processes, Simon J. Ed., Noordhoff, 1976.

[Pitrat 1990] - J. Pitrat. *Métaconnaissances*. Hermès, 1990.

[Puget 1987] - J. F. Puget. *Goal Regression with Opponent*. Progress in Machine Learning, Sigma Press, Wilmslow, 1987.

[Reiter 1978] - R. Reiter. *On closed-world data bases*. Journal of Artificial Intelligence, 13:81-132, 1980.

[Reiter 1980] - R. Reiter. *A logic for default theory*. In H. Gallaire and J. Minker, editors, Logic and Data Bases, pp. 55-76, Plenum Press, New York, 1978.

[Safar 1987] B. Safar, *Le problème des explications négatives dans les Systèmes Experts : Le système POURQUOI-PAS?.* Thèse de l'Université Paris-Sud, 1987.

[Swartout 1983] W. R. Swartout, *XPLAIN: a System for Creating and Explaining Expert Consulting Programs*. Artificial Intelligence 21(3).

[Wallis & Shortliffe 1984], *Explanatory Power for Medical Expert Systems: Studies in the representation of Causal Relationship for Clinical Consultation*. Tech. Report 82-923. Stanford University.

Table des Matières du Chapitre 5

5 L'APPRENTISSAGE ET LA GÉNÉRALISATION 76

5.1 Les systèmes qui apprennent à jouer 76
5.1.1 Les Checkers 76
5.1.2 Le Poker 78
5.1.3 Le Backgammon 78
5.1.4 Les Échecs 79
5.1.5 Les Wargames 80
5.1.6 Le Go 81

5.2 Apprentissage de patterns géométriques 84
5.2.1 Engendrer des patterns géométriques 84
5.2.2 Calculer l'état d'un jeu 85
5.2.3 Généralisation 85
5.2.4 Propriétés des règles 86
5.2.5 Oubli sélectif 87
5.2.6 Validité des calculs locaux dans un contexte global 88
5.2.7 Limites de la représentation à base de patterns géométriques 89

5.3 Apprentissage en logique des prédicats 89
5.3.1 Définitions des buts à apprendre 90
5.3.2 Résolution de problèmes et Explication 92
5.3.3 Détecter les coups forcés 92
5.3.4 Généralisation 93
5.3.5 Opérations sur les règles créées 95
5.3.6 Insertion dans la base 96
5.3.7 Bases d'exemples 96
5.3.8 Le problème de l'utilité 97

5.4 Résultats 97

5.5 Bibliographie 98

5 L'Apprentissage et la généralisation

Dans ce chapitre, je fais un état de l'art des systèmes qui apprennent à jouer, et plus particulièrement des systèmes qui apprennent à jouer au Go. La deuxième section est consacrée à l'apprentissage de règles sous forme de patterns géométriques. Ce type d'apprentissage a été utilisé au début par Gogol. Il utilise maintenant un apprentissage basé sur la logique des prédicats qui lui permet d'apprendre des règles plus générales sur moins d'exemples. Cet apprentissage basé sur la logique des prédicats est décrit dans la troisième section de ce chapitre. Je termine par les résultats obtenus par mon système d'apprentissage.

5.1 Les systèmes qui apprennent à jouer

5.1.1 Les Checkers

Arthur Samuel a essayé de nombreuses techniques d'apprentissage sur le jeu des checkers (les dames anglaises, jouées sur un damier 8x8).

L'apprentissage par coeur

Après avoir calculé un alpha-bêta sur une position, Samuel enregistrait la position avec le résultat du calcul. Ces positions pouvaient plus tard être utilisées pour évaluer statiquement les feuilles d'un autre calcul alpha-bêta, permettant ainsi d'augmenter la profondeur d'analyse. Cette technique pouvait être employée aux checkers grâce à la taille relativement petite de l'espace de recherche. Après avoir appris près de 53000 positions, le programme jouait mieux mais n'avait toujours pas atteint un très bon niveau comparé aux normes humaines [Samuel 1959].

L'avantage de cette technique était de permettre de descendre plus profondément dans la recherche arborescente des coups. Mais la seule capacité de prévoir plus de coups à l'avance n'était pas suffisante pour avoir un bon niveau aux checkers. Cela indiquait qu'il était nécessaire d'améliorer l'évaluation statique plutôt que la simple profondeur de recherche.

L'apprentissage de combinaisons linéaires

Pour améliorer l'évaluation statique des feuilles de l'arbre, Samuel utilisa des combinaisons linéaires de caractéristiques. Chaque caractéristique était un attribut du damier qu'on pensait être utilisé par les bons joueurs de checkers ; par exemple, la mobilité des pièces, le contrôle d'une rangée ou le contrôle du centre du damier. Samuel avait programmé 38 caractéristiques ; seulement 16 étaient utilisables en même temps. Chacune était associée à un poids, et seules celles correspondant aux 16 plus forts poids étaient utilisées. Une méthode pour apprendre les poids était une descente de gradient. Cette méthode est la suivante : le vecteur de poids est figé et on fait jouer un programme avec un vecteur figé contre un programme avec un vecteur qui se modifie. Le vecteur se modifie en fonction de la différence entre son évaluation directe des coups et l'évaluation retournée par une recherche plus profonde. Lorsque le nouveau vecteur gagne la plupart des parties contre l'ancien vecteur, il est figé et remplace l'ancien.

Toutes les caractéristiques programmées par Samuel étaient simples, et son programme n'autorisait qu'une combinaison linéaire de ces caractéristiques. Il ne pouvait pas comprendre les non-linéarités du domaine. Le niveau se stabilisait au bout d'une trentaine de parties, et le résultat était un programme qui connaissait les rudiments mais n'était pas encore un expert.

L'apprentissage de tableaux de signatures

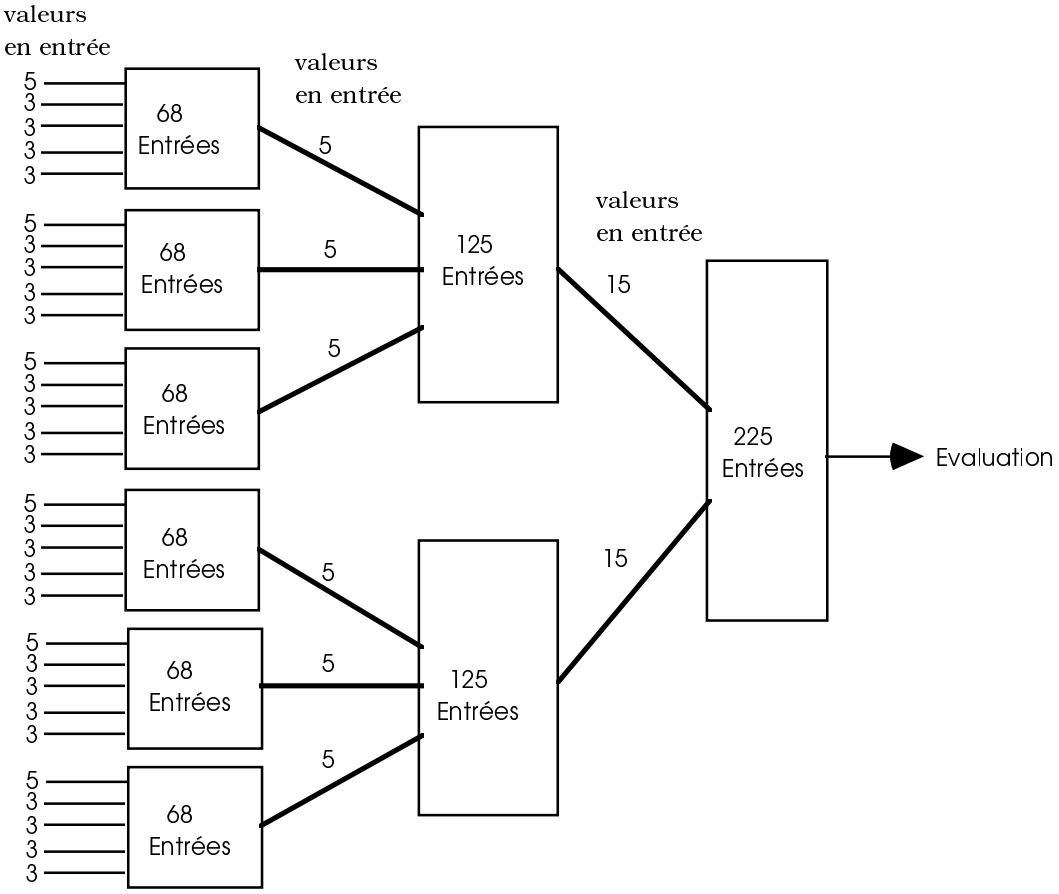
Un tableau de signatures est un tableau qui contient un élément pour n'importe quelle combinaison de ses entrées. Un tel tableau recouvre entièrement l'espace des entrées possibles ; il est ainsi capable de découvrir des non-linéarités entre les caractéristiques d'entrée. Ce tableau est donc un grand progrès sur la combinaison linéaire, qui ne marche bien que si les caractéristiques sont indépendantes les unes des autres, ce qui n'est souvent pas le cas. Le problème de ces tableaux de signatures est que leur taille croît exponentiellement avec le nombre de caractéristiques en entrée.

Samuel a donc réduit le nombre de caractéristiques à 24, et a aussi réduit leur domaine : une caractéristique n'avait désormais plus de valeur que dans {-1,0,1} qui indiquait si la caractéristique était bonne, neutre ou mauvaise pour un joueur. Même avec cette simplification, un tableau de signatures contenant toutes ces entrées contenaitt 3^{24} soit $3x10^{11}$ entrées. Ce qui est beaucoup plus grand que les 250 000 positions qu'avait Samuel.

Pour pallier à ce problème, Samuel utilisa plusieurs techniques. Il divisa manuellement les caractéristiques en 6 sous-groupes, chacun comportant 3 entrées comprises dans {-1,0,1} et une entrée comprise dans {-2,-1,0,1,2}. La symétrie dans les entrées était utilisée pour réduire à 68 le nombre d'entrées pour chaque sous-groupe. Les sorties étaient utilisées par un second niveau dont les sorties étaient elles-mêmes envoyées sur un troisième niveau comportant 225 entrées.

En général, l'apprentissage de tableaux de signatures est difficile ; leur avantage, qui réside dans leur capacité à représenter les non-linéarités du domaine, entraîne aussi un inconvénient : les méthodes de descente de gradients s'appliquent mal. Il existe tout de même des techniques simples pour trouver des optima locaux. Samuel utilisa la technique suivante : pour chaque élément du tableau de

signatures, un compte séparé était tenu pour les accords et les désaccords avec les parties de maîtres. La sortie du tableau de signatures était calculée de façon à ce qu'elle soit un coefficient de corrélation qui indiquait la propension de cet élément à être présent dans les coups tirés de livres sur les checkers. En obligeant la sortie de chaque tableau de signatures à être en corrélation avec la sortie du tableau de signatures du troisième niveau, Samuel a contraint le problème. Une autre optimisation de Samuel fut d'utiliser sept hiérarchies différentes de tableaux de signatures pour chaque phase d'une partie : en effet, la fin de partie aux checkers est très différente du début de partie.



L'utilisation de tableaux de signatures s'est montrée bien plus efficace que les précédentes méthodes. Après s'être entraîné sur 173 989 coups tirés de livres, le programme était capable de prédire, sans recherche arborescente, un coup de maître dans 38% des cas sur un ensemble de 895 coups qui ne faisaient pas partie de l'ensemble d'apprentissage [Samuel 1967].

5.1.2 Le Poker

Le système d'apprentissage du poker de Waterman crée des règles qui indiquent l'action à exécuter. Pour éviter les conflits, les règles sont ordonnées et la première règle déclenchable est exécutée [Waterman 1970]. Ce système pose problème lorsqu'on désire ajouter une nouvelle règle : elle risque de s'appliquer à des situations déjà rencontrées à la place de la règle créée pour ces situations et les performances peuvent donc se dégrader.

5.1.3 Le Backgammon

La particularité du Backgammon

D'après Tesauro [Tesauro 1989], le choix d'un jeu à étudier doit dépendre des aptitudes particulières nécessaires pour pouvoir bien y jouer. Il fait la distinction entre deux aptitudes fondamentalement

différentes. La première est l'aptitude de pouvoir "prévoir" ce qui va se passer, soit par une recherche arborescente soit par un raisonnement. La seconde, est la capacité de "juger" précisément une position à partir des patterns et des caractéristiques de la position, sans calculer explicitement le déroulement futur de la partie. Le Backgammon est un jeu insolite parce que les jugements positionnels y prédominent, par rapport à d'autres jeux comme les Échecs qui nécessitent souvent de prévoir beaucoup de coups à l'avance. Cette particularité du Backgammon est dûe à l'intervention du hasard.

Les réseaux de neurones

Le programme de backgammon de Tesauro joue en assignant des valeurs aux coups. Le premier coup légal qui a la plus grande valeur est choisi. Un réseau de neurones a été utilisé pour apprendre ces évaluations à partir d'une base de données contenant des coups associés à leur évaluation par un expert. Il a donc appris une fonction non-linéaire en utilisant un réseau de neurones multi-couches, avec apprentissage par rétropropagation. Le réseau apprend à minimiser la somme des carrés des différences d'évaluation entre les coups du réseau et ceux de l'expert en suivant une méthode de gradient. Tesauro a introduit de nouveaux exemples dans sa base d'exemples pour corriger certaines erreurs de comportement qu'il avait observées après le premier apprentissage.

Les caractéristiques

Utiliser un réseau de neurones pour apprendre entraîne des problèmes de choix d'une topologie pour le réseau adaptée au problème, et des problèmes de choix des entrées et d'une représentation de ces entrées. Tesauro ajouta des caractéristiques spécifiques au backgammon en plus de la position du damier. Il trouva que ces informations supplémentaires étaient très importantes pour que le réseau puisse apprendre. Il a utilisé 8 caractéristiques : (1) le nombre de points des dés, (2) le degré de contact, (3) le nombre de points occupés dans le camp ami, (4) le nombre de points occupés dans le camp adverse, (5) le nombre de jetons dans le camp adverse, (6) la présence d'une forme "première", (7) l'exposition d'une case (la probabilité qu'une case puisse être atteinte), (8) la force d'un blocage (la probabilité qu'un ennemi prisonnier d'un blocage puisse s'échapper).

5.1.4 Les Échecs

Appech

APPECH est un programme qui joue aux Échecs. Il a été écrit par Jacques Pitrat en 1976 [Pitrat 1976]. Il est basé sur la notion de conseil. L'ajout de nouvelles connaissances ne perturbe pas l'apprentissage, l'augmentation du nombre de conseils suggère d'examiner un plus grand nombre de coups. Ce programme commence par trouver pour quelles raisons un coup joué au cours d'une partie est bon. Trois méthodes lui permettent de comprendre un coup; dans la première méthode, quand un événement intéressant (par exemple une capture) s'est produit, le programme regarde si le coup joué R est légal un coup plus tôt. Si c'est le cas, il le joue à la place du coup Q réellement joué avant R et développe ensuite une arborescence contenant des coups que ses connaissances du moment lui suggèrent d'envisager. Si l'arborescence où R est joué à la place de Q donne par un minimax un bilan défavorable et si celle commençant par Q a un bilan favorable, il estime que Q avait pour rôle de préparer une combinaison; il cherche alors les liens entre les coups de ces arborescences. Une partie du système est ensuite chargée de généraliser une situation. Une deuxième méthode permet de détecter les menaces en simulant un coup vide pour l'adversaire. Une troisième méthode donne les moyens de comprendre pourquoi certaines menaces ne se sont pas concrétisées dans une partie.

ID3

Quinlan a utilisé ID3 pour classifier correctement une grande base de données contenant des positions de fins de parties aux Échecs [Quinlan 1984]. ID3 ne permet d'induire que des arbres de décisions utilisant les attributs fournis au système.

Cependant, l'apprentissage est linéaire en fonction :

- du nombre d'exemples,
- du nombre d'attributs,
- de la profondeur de l'arbre de décision à induire.

Le but de cet apprentissage est de trouver des attributs peu coûteux en temps de calcul qui permettent de classifier correctement une position. La classification de la position prendra alors moins de temps que le calcul du minimax sur cette position. L'arbre de décision induit sur les positions de mat en trois coups est 5 fois plus rapide qu'une recherche arborescente spécialisée et 80 fois plus rapide qu'un minimax.

Le problème réside en fait dans la découverte des bons attributs. Quinlan donne dans la fin de son article des moyens de généraliser les positions pour en faire des patterns correspondant à des attributs utiles, mais il ne donne pas de méthode explicite et générale pour créer automatiquement ces attributs.

Morph

Morph est un système développé par Robert Levinson [Levinson 91, 92, 93] qui apprend à jouer aux Échecs en intégrant plusieurs méthodes d'apprentissage. Il a comme principe de ne pas utiliser de recherche en profondeur, il essaie tous les coups, évalue la position et choisit le coup qui amène à la meilleure position. Pour apprendre, il utilise des pattern-weights, c'est à dire des patterns associés à des poids. L'apprentissage consiste à créer des patterns puis à ajuster leurs poids. Le programme n'a pas un bon niveau. Une grande partie de ses connaissances viennent du langage de représentation des patterns.

L'Apprentissage à Partir d'Explications

Des techniques d'apprentissage à base d'explications (Explanation Based Learning) pour apprendre à jouer aux Échecs ont été utilisées de façons différentes par [Minton 1984], [Puget 1987], [Tadepalli 1989] et [Pell 1993]. S. Minton utilise des explications basées sur le déclenchement de règles décrivant les règles du jeu, mais comme l'a remarqué plus tard J. F. Puget, ces règles amènent à apprendre des règles trop générales et donc fausses. La solution que propose J. F. Puget à ce problème est de créer des règles associées à des arbres représentant seulement une partie de la situation. C'est une solution qui est proche de la notion de conseil utilisée dans Appech. Une autre tentative faite pour pallier ce problème a été proposée par P. Tadepalli : elle consiste à apprendre des règles trop générales et à les raffiner par la suite. B. Pell dans le cadre de son projet Metagame a aussi utilisé une forme d'apprentissage à partir d'explications.

5.1.5 Les Wargames

Eurisko [Lenat 1983] est un programme qui apprend des concepts et des heuristiques d'un certain domaine et dans le métadomaine de découverte de nouveaux éléments utiles pour la découverte. Il a été appliqué en particulier à un wargame de bataille navale. Il a été deux fois de suite champion du monde grâce aux failles qu'il avait trouvées dans les règles très complexes de ce wargame. Eurisko utilise beaucoup de règles qui portent sur les connaissances et qui sont donc des métarègles. Un exemple de métarègle est le suivant :

> **Si** les résultats obtenus en exécutant f ne sont que parfois utiles
> **Alors** envisager de créer de nouvelles spécialisations de f en spécialisant un de ses attributs.

Le mode de fonctionnement d'Eurisko est interactif. Il crée des hypothèses sur son domaine d'application à partir des expériences qu'il fait. Il propose ensuite ces heuristiques à son programmeur qui sélectionne celles qui lui paraissent bonnes. Il est donc très difficile de reproduire les expériences de D. Lenat sur Eurisko puisqu'il fait lui-même partie de la boucle du programme.

5.1.6 Le Go

L'apprentissage symbolique par exemples et contre-exemples

Dans le cadre de sa thèse de doctorat, P. Pompidor [Pompidor 1992] a développé un système d'apprentissage symbolique au jeu de Go. Ce système recherche des séquences de coups dont les caractéristiques géographiques évoluent de manière relativement similaire.

En utilisant des concepts géométriques (colinéarité, orthogonalité, proximité, etc.), des valeurs de tolérance de similitude et des notions temporelles, le système parvient à généraliser certaines séquences conceptuellement proches et à en extraire les caractéristiques invariantes.

Son système a été évalué dans la thèse au niveau de 12ème Kyu, il n'a toutefois à ma connaissance participé à aucune compétition contre des humains ou contre d'autres programmes.

La méthode des différences temporelles

Trois programmes d'apprentissage du jeu de Go sont basés sur des principes similaires : le programme de N. Schraudolph [Schraudolph 1994], Gobble le programme de Bernd Brügmann [Brügmann 1993] et NeurogoII le programme de Markus Enzenberger [Enzenberger 1996]. Ils utilisent des réseaux de neurones qui ont en entrée la position sur le damier et certaines caractéristiques de cette position, et en sortie une évaluation de la position. La méthode d'apprentissage appelée TD(0) ne nécessite aucune connaissance préalable du jeu de Go pour le programme ou le programmeur, en dehors des caractéristiques d'entrée. Elle consiste à faire jouer un programme contre un autre. Une fois la partie terminée, on effectue une rétropropagation positive sur toutes les positions de la partie avec pour couleur amie celle du programme qui a gagné, et une rétropropagation négative avec pour couleur amie celle du programme qui a perdu.

L'approche de Markus Enzenberger est originale car elle adapte la structure du réseau à la position sur le goban. Chaque bloc est représenté par plusieurs neurones, NeurogoII utilise un neurone par caractéristique du bloc (nombre de libertés, nombre de pierres etc...). C'est cette architecture très particulière qui adapte la topologie du réseau à la position qui fait le succés de cette approche.

Les performances en apprentissage de ces programmes sont très étroitement liées à la représentation du damier. Une grande partie des connaissances données au programme est dans cette représentation. De plus, ces programmes sont basés sur un apprentissage statistique et ne comprennent pas les raisons de leurs succés ou de leurs échecs. Ils n'apprennent pas à prévoir mais à juger une position, ce qui les limite dans les domaines comme le jeu de Go pour lesquels la prévision est aussi importante que le jugement positionnel. Pour apprendre à bien jouer contre un autre programme, il doivent jouer contre lui plusieurs milliers de parties. Ils ont toutefois une grande qualité : ils ne demandent pas beaucoup de travail à leurs programmeurs (Neurogo ne comporte que 5000 lignes de C).

Golem et les réseaux de neurones

Golem est un programme qui apprend à jouer au Go en utilisant une méthode basée sur les réseaux de neurones. Il a été développé à Carnegie Mellon par H. Enderton [Enderton 1992] et il utilise certaines caractéristiques intéressantes en entrée de son réseau de neurones.

La première action de Golem est une recherche tactique sur le statut des blocs ayant 3 libertés ou moins, recherche qui lui permet de voir si ces blocs sont capturés ou non. Pour envisager ses coups, il utilise les règles suivantes :

1. Si le défenseur a trois libertés et qu'il peut jouer, il est sauvé.

2. L'attaquant peut essayer de remplir une liberté du bloc attaqué.

3. Si le défenseur a seulement deux libertés,

l'attaquant peut jouer un coup adjacent aux deux libertés.

4. Si le défenseur a deux libertés et qu'un bloc attaquant est en atari,
l'attaquant peut essayer de sauver son bloc.

5. Si le défenseur est en atari,
il peut seulement essayer les coups qui augmentent ses libertés.

6. Si le défenseur a deux libertés,
il peut faire atari un attaquant, jouer à coté d'une liberté, ou passer.

7. Au-dessus d'une certaine profondeur, l'attaquant a seulement le droit
de remplir les libertés et ne peut plus appliquer les règles 3 et 4.

Golem utilise un réseau de neurones pour élaguer cette recherche quand les règles conseillent trop de coups à envisager.

Chaque intersection vide tombe dans une des classes suivantes :

- Une liberté Noire protégée (Si Blanc y joue, il est capturé).

- Une liberté Blanche protégée.

- Un coup sûr pour Noir. Une pierre sur cette intersection serait clairement connectée à un groupe apparemment vivant (par une connexion directe ou diagonale).

- Un coup sûr pour Blanc.

- Une intersection neutre est un coup sûr pour les deux joueurs.

En entrée de son réseau de neurones, Golem utilise les caractéristiques suivantes :

- La couleur de chacune des 8 intersections entourant le coup envisagé. Les valeurs possibles de ces intersections sont : pierre amie, pierre ennemie, vide ou extérieure au damier. Il y a un ordre des intersections : la quatrième entrée correspond à l'intersection 'en dessous' du coup (en direction du bord le plus proche).

- Le nombre de libertés d'une pierre amie, si elle était posée sur cette intersection.

- Le nombre de libertés d'une pierre ennemie, si elle était posée sur cette intersection.

- Le nombre de libertés de chacune des quatre intersections voisines, si celles-ci sont occupées par des pierres.

Pour chaque position de parties de professionnels, le but de Golem est de classer le coup du professionnel au-dessus d'un autre coup tiré au hasard. Après 2000 essais, il classe correctement le coup de la base de test dans 87% des cas. Toutefois cette méthode de mesure n'est pas bonne. Un programme de Go doit être évalué en fonction de son niveau de jeu par rapport aux humains ou par rapport aux autres programmes de Go. Dans une partie de Go, si dans un séquence un programme joue dix coups de professionnel et un mauvais coup, celui ci annule souvent tous les bons coups précédents. Un programme de Go est mieux évalué par sa capacité à ne pas faire de grosses erreurs que par le bons coups qu'il peut parfois jouer.

Un autre réseau de neurones a été utilisé pour ne garder que les coups intéressants, il utilise les caractéristiques suivantes :

- La valeur du premier réseau

- L'état tactique des intersections dans une région 3x3 autour du coup (intersection protégée ou pierre prise).

- Le fait de savoir si le coup sauve ou prend des pierres.

- La couleur de la pierre la plus proche, sa distance et sa force, pour chacune des 8 directions.

- La distance au bord le plus proche.

- La distance au second bord le plus proche.

- Le fait de savoir s'il y a un Ko.

Ce réseau entrainé de la même façon que le précédent préfère le coup du professionnel à un coup tiré au hasard dans 90% des cas. D'autres caractéristiques telles que la couleur du plus grand ensemble de pierres voisin, la taille des groupes voisins, et la densité moyenne des pierres sur le damier ont été essayées sans améliorer les performances.

Metagame

Les expériences de Barney Pell sur le jeu de Go [Pell 1991] font partie d'un projet appelé Metagame dont le but est l'apprentissage des jeux, uniquement à partir de leurs règles. Barney Pell a appliqué ses théories au jeu de Go en développant un programme qui apprend en jouant contre ses adversaires. Son programme privilégie le long terme sur le court terme, c'est-à-dire que celui-ci au cours d'une partie, ne cherche pas à jouer le coup qui selon lui a le plus de chances de gagner, mais le coup qui a le plus de chances de lui fournir des informations utiles pour les parties futures. Il propose des tournois de programmes pouvant apprendre afin de départager ceux qui apprennent le mieux.

Les caractéristiques qu'il utilise (et qui sont des caractéristiques de coups et non pas de positions) sont les suivantes :

Capture? :
- Renvoie Vrai si le coup capture un groupe ennemi, sinon renvoie Faux.

Edge? :
- Vrai si le coup est joué sur le bord du goban.

Friend-Strength :
- Le nombre de libertés du groupe ami le plus faible adjacent à l'intersection où le coup a été joué, évalué avant le coup.

Valeurs possibles : 0 (pas de groupe ami), 1, 2, 3 ou Plus.

Ennemy-Strength :
- Le nombre de libertés du groupe ennemi le plus faible adjacent à l'intersection où le coup a été joué, évalué avant le coup.

Valeurs possibles : 0, 1, 2, 3 ou Plus.

Group-Strength :
- Le nombre de libertés du groupe que le coup crée, rejoint ou relie ensemble.

Valeurs possibles : 1, 2, 3 ou Plus.

Merge-Count :
- Le nombre de groupes que le coup relie ensemble.

Valeurs possibles : 0 (crée un nouveau groupe),1 (extension), 2, 3, 4.

Le programme apprend tout seul des règles pour noter les coups :

Si Capture et Group-Strength > 1, Alors augmente le score de 200.
(Les captures sont bonnes)

Si Edge, Alors diminue le score de 30.
(Jouer au bord est mauvais)

Si Group_Strength=3 et Ennemy-Strength=3, Alors augmente le score de 40.
(Opposer les forces est bon)

Si Group_Strength=1 et not Capture, Alors diminue le score de 100.
(un damezumari sans capture est mauvais)

Le programme de Barney Pell joue un peu mieux qu'un joueur aléatoire sur des gobans allant de 3x3 à 9x9.

5.2 Apprentissage de patterns géométriques

Tous les meilleurs programmes de Go actuels utilisent des patterns géométriques pour envisager des coups et évaluer des sous-jeux. Il parait donc intéressant d'engendrer automatiquement ces patterns géométriques. Cette acquisition automatique de patterns géométriques a été la première forme d'apprentissage déductif utilisée par Gogol [Cazenave 1994,1996d]. Cette section décrit la méthode qui permet d'engendrer automatiquement ces patterns.

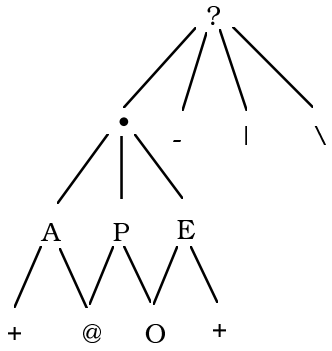
5.2.1 Engendrer des patterns géométriques

Un programme engendre tous les patterns envisageables dans un espace restreint. Cet espace restreint est défini par un modèle de pattern qui permet d'engendrer des patterns.

Signification des symboles :

@	: Pierre amie
O	: Pierre ennemie
+	: Intersection vide
-	: Extérieur du goban
|	: Extérieur du goban
\	: Extérieur du goban
A	: @ ou +
E	: O ou +
P	: @ ou O
.	: @ ou O ou +
?	: @ ou O ou + ou - ou | ou \

On peut construire une hiérarchie des symboles, du plus général au moins général :



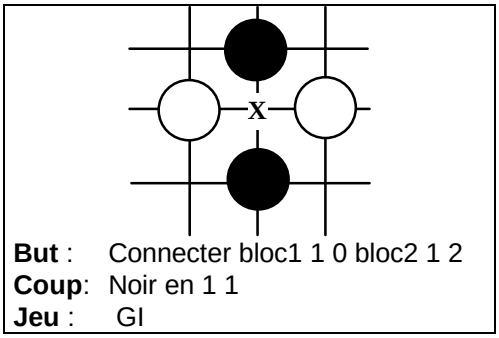
Exemple de modèle de génération de patterns :

```
. @ .
. + .
. @ .
```

Dans cet exemple, les '.' sont remplacés successivement par des @, des O puis des +. On engendre ainsi 3^6=729 patterns.

5.2.2 Calculer l'état d'un jeu

Sur chacun de ces patterns, le programme calcule l'état du jeu pour un but choisi parmi ceux exposés plus haut et retient la règle ainsi déduite si elle est intéressante, c'est-à-dire si le jeu est GI ou G et éventuellement les coups associés.



Exemple de règle apprise par le système

Dans cet exemple, si noir joue en X il connecte ses deux pierres (la pierre en 1 0 et la pierre en 1 2) alors que si c'est blanc qui joue en X les deux pierres noires ne seront plus connectables.

5.2.3 Généralisation

Gogol compare chaque règle calculée avec les règles ayant les mêmes conclusions. Si deux règles ne diffèrent que par une seule prémisse, il crée une nouvelle règle contenant la généralisée des deux prémisses et il supprime les deux anciennes règles. Il peut aussi effectuer la généralisation quand trois règles ont une seule prémisse qui prend les trois valeurs possibles.

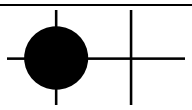
Exemple :

Règle 1		Règle 2		Règle 3		Règle généralisée
+ @ .		+ @ .		+ @ .		+ @ .
+ + +	+	+ + +	+	+ + +	→	+ + +
@ @ +		@ @ @		@ @ O		@ @ .

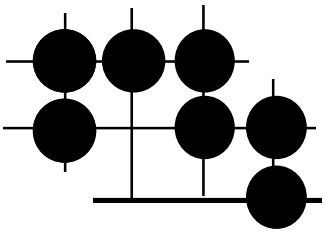
5.2.4 Propriétés des règles

La certitude

Une règle certaine est une règle qui atteint toujours le but.



But :
Prendre bloc1 0 0
Libertés :
max_libertés 0 0 1
min_pierres 0 0 1
Coup : Blanc en 1 0
Certitude : 1



But : Vivre bloc1 0 0
Coup : Noir en 1 2
Certitude : 1

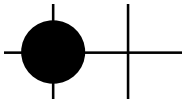
Tableau Apprentissage Règles Certaines

Par exemple la première règle du tableau Apprentissage Règles Certaines atteint toujours le but qui lui est associé si elle joue le coup conseillé et que ce coup est un coup légal. Jouer sur la dernière liberté d'un bloc qui n'a qu'une seule liberté permet toujours de prendre ce bloc si jouer sur cette liberté est un coup légal. Les conditions sur les libertés sont 'max_libertés 0 0 1' qui signifie que le bloc de la pierre noire en 0 0 doit avoir au plus une liberté, et 'min_pierres 0 0 1' qui signifie que le bloc de la pierre noire en 0 0 doit avoir strictement plus d'une pierre (ce qui garantit qu'un coup blanc en 1 0 est légal : ce n'est pas un Ko).

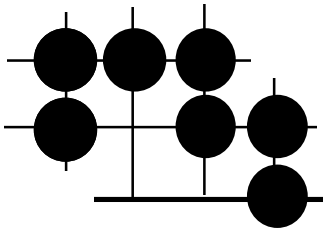
La deuxième règle du tableau Apprentissage Règles Certaines conseille un coup pour vivre qui fait deux yeux. Jouer ce coup quand le pattern de la règle est vérifié permettra donc toujours d'atteindre le but Vivre, la certitude d'atteindre le but en jouant ce coup est donc de 1.

La nécessité

Une règle nécessaire est une règle dont on ne connait pas le résultat ; à savoir, va-t-elle ou non permettre d'atteindre le but. On sait seulement que si on ne l'applique pas, on est sûr de ne pas atteindre le but. Quelle que soit la position dans laquelle elle s'applique, on est sûr qu'il faut l'appliquer elle et pas une autre si on veut atteindre le but.



But :
Vivre bloc1 0 0
Libertés :
max_libertés 0 0 1
min_libertés_ blocs_adjacents 0 0 2
Coup : Noir en 1 0
Nécessité : 1



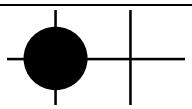
But : Prendre bloc1 0 0
Coup : Noir en 1 2
Nécessité : 1

Tableau Apprentissage Règles Nécessaires

On peut remarquer que les règles nécessaires sont liées aux règles certaines. Une règle nécessaire est une règle qui empêche une règle certaine de s'appliquer. On voit cependant que les prémisses des règles nécessaires peuvent être plus complètes que celles des règles certaines comme dans la première règle du tableau Apprentissage Règles Nécessaires par rapport à la première règle du tableau Apprentissage Règles Certaines. En effet une règle nécessaire doit prendre en compte le fait qu'elle est la seule alternative possible, dans le cas ou plusieurs coups empêchent d'atteindre le but inverse, il est plus difficile d'arbitrer entre ces coups. La nécessité est une métaconnaissance très importante qui permet d'être certain que l'on n'a qu'un seul coup à essayer dans une situation. L'arbre de recherche ne comporte alors plus qu'un seul noeud dans les cas où une règle nécessaire s'applique.

Les statistiques

Les autres règles ont des statistiques estimant la probabilité de gagner si on joue le coup. Cette mesure est appelée ‘Statistique de gain'. Elles ont aussi des statistiques estimant la probabilité de perdre si on ne joue pas le coup. Cette mesure est appelée 'Statistique d’urgence'.



But :	Sauver bloc1 0 0
Libertés :	max_libertés 0 0 3
	min_libertés_coup_ami 1 0 5
Coup :	Noir en 1 0
Statistiques de gain :	70 %
Statistiques d’urgence :	60 %
Nombre de tests :	10

Tableau Apprentissage Statistiques

Les règles contiennent aussi le nombre d’exemples sur lesquels les statistiques ont été effectuées.

5.2.5 Oubli sélectif

Une règle peut être plus ou moins générale. La généralité d'une règle peut être mesurée par sa fréquence d'application. Un moyen coûteux de calculer la généralité d'une règle est d'avoir un ensemble de parties et de calculer le nombre de fois où la règle s'applique sur cet ensemble de partie. Un moyen moins coûteux mais un peu plus approximatif est de créer une fonction qui donnera une généralité en fonction des prémisses de la règle.

Au début d'une partie de Go, le goban est vide. On rajoute une pierre de chaque couleur à chaque coup. Si on néglige les prisonniers au cours de la partie et si on compte qu'une partie sur un goban 19x19 dure à peu près 250 coups, on peut évaluer la probabilité d'avoir une intersection vide à :

(361+360+...+111)/(250*361)=(125*(111+361))/(250*361)=0,65.

La probabilité d'avoir une intersection noire est égale à la probabilité d'avoir une intersection blanche et vaut alors 0,17.

Une intersection vide est approximativement 4 fois plus fréquente qu'une intersection noire et qu'une intersection blanche. L'indice de spécialisation d'un pattern sera incrémenté de 1 pour chaque intersection vide et de 4 pour chaque intersection noire ou blanche.

Un pattern au bord apparaît en moyenne un peu moins souvent qu’un pattern de même taille au centre. On fera une approximation en incrémentant de 2 l'indice de spécialisation lorsque le pattern est au bord.

On augmentera de 5 la spécificité pour chaque prémisse sur les libertés ou sur le type des blocs d'éléments du pattern.

Le tableau Apprentissage calculs de spécialisation donne deux exemples de calculs de spécialisation.

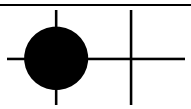
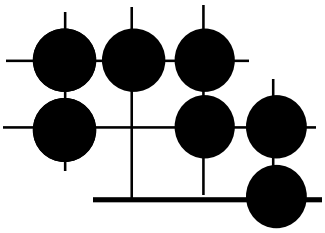
 **But** : Sauver bloc1 0 0 **Libertés** : max_libertés 0 0 1 min_libertés_min_blocs_adjacents 0 0 2 **Coup** : Noir en 1 0 **Spécialisation** : 20	 **But** : Prendre bloc1 0 0 **Coup** : Noir en 1 2 **Spécialisation** : 35

Tableau Apprentissage calculs de spécialisation

Il est possible de filtrer les règles qu'on veut obtenir ; on peut par exemple ne retenir que les règles ayant une spécialisation inférieure à 30. Le calcul de spécialisation permet de ne garder que les règles les plus générales en oubliant les règles trop spécifiques qui ralentissent le programme sans lui donner un avantage important.

5.2.6 Validité des calculs locaux dans un contexte global

Pour connaître la validité d'un jeu calculé sur une sous-partie du damier dans le contexte global du damier tout entier, on doit classer les buts en deux catégories :

- Les buts atteignables, c'est-à-dire ceux dont on connaît un état du goban pour lequel ils sont atteints et qui sont :

  - Prendre une pierre : le but est atteint lorsque la pierre est remplacée par une intersection vide.

  - faire Vivre une pierre : le but est atteint lorsque la pierre fait partie d'un bloc qui a deux yeux.

  - Connecter deux pierres : le but est atteint lorsque les deux pierres font partie du même bloc.

  - Faire un oeil : le but est atteint lorsque l'un des trois patterns d'oeil matche au bon endroit.

- Les buts inatteignables, c'est-à-dire ceux dont on ne peut pas définir facilement un état du goban pour lequel ils sont atteints. Ce sont souvent les buts inverses des buts atteignables, comme par exemple :

  - Déconnecter deux pierres : le but est atteint lorsque l'adversaire n'a plus de coups permettant d'atteindre le but Connecter.

  - Empêcher un oeil.

Pour les buts atteignables, on peut utiliser les métarègles suivantes :

Si un jeu est localement prouvé G
Alors il est aussi globalement (sur tout le goban) prouvé G

Si un jeu est localement prouvé GI
Alors il est aussi globalement prouvé GI ou G

Pour les buts inatteignables, on peut utiliser les métarègles suivantes :

Si un jeu est localement prouvé P
Alors il est aussi globalement prouvé P

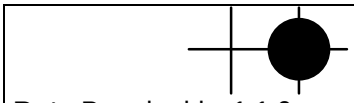
Si un jeu est localement prouvé IP
Alors il est aussi globalement prouvé IP ou P

Ces métarègles découlent de la taxonomie des jeux présentée dans le chapitre sur la théorie combinatoire des jeux.

5.2.7 Limites de la représentation à base de patterns géométriques

Au jeu de Go, il existe des situations pour lesquelles un calcul limité dans une sous-partie du damier ne permet pas de connaître la solution d'un problème. Dans le cas du shicho, qui est un calcul que même les débutants savent effectuer, on doit poser mentalement des pierres sur toute une diagonale du damier. Le calcul est nécessaire pour savoir si on peut prendre une pierre ou non, car il existe un très grand nombre de shichos différents et essayer de les répertorier serait vain. Mais les coups à essayer lors de la construction mentale de l'arbre de calcul sont très simples et très faciles à repérer et la plupart du temps, l'arbre se réduit à un tronc.

Le système ne peut pas apprendre des patterns ayant une trop grande taille, il ne peut donc pas apprendre les patterns correspondant aux shichos, ni ceux correspondant à beaucoup de calculs qui s'effectuent sur une grande partie du damier.



But : Prendre bloc1 1 0
Libertés :
Libertés_max 1 0 1
Libertes_blocs_adjacents_min 1 0 1
Jeu : G

Tableau Apprentissage Exemple règle non apprise

De même, cette façon d'apprendre ne permet pas de prouver des règles contenant des conditions extérieures au pattern. Le tableau Apprentissage Exemple règle non apprise donne un exemple de règle où la condition 'Libertés_max 1 0 1' qui vérifie que la pierre noire sur l'intersection de coordonnées 1 0 aura au plus 1 liberté après un coup Noir et la condition 'Libertes_blocs_adjacents_min 1 0 1' qui vérifie que tous les blocs adjacents à la pierre noire ont plus d'une liberté, ne peuvent pas être trouvées par le mécanisme d'apprentissage par patterns géométriques. Pourtant cette règle est très simple et très utile pour un programme de Go.

L'apprentissage en logique des prédicats n'a pas les limites de l'apprentissage à base de patterns géométriques. Il permet à la fois d'apprendre plus de règles dans le domaine du Go et d'apprendre des règles beaucoup plus générales. En outre il peut facilement s'appliquer à d'autres domaines.

5.3 Apprentissage en logique des prédicats

Dans les domaines pour lesquels il existe une théorie, une méthode d'apprentissage déductif a été développée : l'apprentissage basé sur les explications[6] [Mitchell 1986] [Dejong 1986] [Minton 1990]. Ce type d'apprentissage est particulièrement utile dans le domaine des jeux. Plusieurs projets utilisant l'apprentissage basé sur les explications pour apprendre des combinaisons tactiques dans des jeux ont été décrits [Minton 1984] [Puget 1987] [Pell 1993], le travail de J. Pitrat sur les Échecs [Pitrat 1976] en ayant été un précurseur. Toutefois, la représentation des connaissances utilisée dans

[6] Explanation Based Learning (EBL) ou Speed-up Learning.

ces programmes les amène à apprendre des connaissances trop générales. Cela les oblige par la suite, soit à vérifier partiellement dans chaque cas si les connaissances apprises s'appliquent effectivement, soit à utiliser ces connaissances comme des heuristiques. Contrairement à ces programmes, mon approche du problème est de représenter explicitement l'incertitude et de bien différencier ce qui est certain de ce qui ne l'est pas. Je cherche à obtenir que les prévisions données par les règles apprises par le système n'aient pas à être vérifiées par la suite.

Mon approche diffère également de projets comme Prodigy [Minton 1988], où l'apprentissage basé sur les explications est utilisé pour créer des règles de contrôle de la recherche, ou comme Soar [Laird 1986] dans lequel le 'chunking' est utilisé pour créer des macro opérateurs en utilisant une représentation des connaissances limitée [Tambe 1994].

Mon but est de remplacer automatiquement les parties coûteuses de la recherche arborescente par le filtrage efficace d'une base de règles apprises.

5.3.1 Définitions des buts à apprendre

Dans cette section, j'expose les principales règles qui détectent qu'un but a été atteint. Ces règles sont déclenchées à la fin de la résolution de problème. Elles permettent au système de connaître les buts qui ont été atteints après le coup. Les conclusions qu'elles donnent sont ensuite utilisées par les règles de régression (Cf. chapitre sur les explications) afin qu'elles décident si les jeux déduits sont intéressants à régresser. Cet ensemble de règles est l'ensemble initial des règles concernant l'achèvement des buts données à Gogol. Toutes les autres règles concernant l'achèvement des buts sont découvertes par lui-même à partir de cet ensemble de règles.

```
( nom ( Regle_jeu_apres_5 )
  premisses (     present ( Couleur ( ?c ) )
                  present ( Bloc_ote ( ?b ) )
                  present ( Couleur_bloc_avant ( ?b ?c ) )
                  present ( Couleur_opposees ( ?c ?c1 ) ) )
                  absent ( Jeu_bloc_apres ( ?c1 Prendre ?b G ) ) )
  conclusions (   ajoute ( Jeu_bloc_apres ( ?c1 Prendre ?b G ) ) ) )
```

Tableau Apprentissage Définition But Prendre

Le tableau Apprentissage Définition But Prendre donne la règle qui permet de déduire qu'un bloc a été pris après le coup (la prémisse 'present (Bloc_ote (?b))') alors qu'il était présent sur le damier avant le coup (la prémisse 'present (Couleur_bloc_avant (?b ?c))'). Le jeu de Prendre le bloc ?b est donc gagné après le coup puisque le bloc a été retiré du damier. La règle vérifie que le jeu combinatoire G n'avait pas été déjà déduit avec la prémisse 'absent (Jeu_bloc_apres (?c1 Prendre ?b G))'. Elle peut alors déduire comme un fait nouveau que le jeu de Prendre le bloc ?b est gagné.

```
( nom ( Regle_jeu_apres_6 )
   premisses (    present ( Couleur ( ?c ) )
                  present ( Fusion_blocs ( ?b ?b1 ) )
                  present ( Couleur_bloc_avant ( ?b ?c ) )
                  absent ( Jeu_binaire_bloc_apres ( ?c Connecter ?b ?b1 G ) )
                  absent ( Jeu_binaire_bloc_apres ( ?c Connecter ?b1 ?b G ) ) )
   conclusions ( ajoute ( Jeu_binaire_bloc_apres ( ?c Connecter ?b ?b1 G ) ) ) )
```

Tableau Apprentissage Définition But Connecter deux blocs

Le tableau Apprentissage Définition But Connecter deux blocs donne la règle qui permet de déduire que deux blocs qui étaient disjoints avant le coup sont maintenant réunis (la prémisse 'present (Fusion_blocs (?b ?b1))'). Le jeu de connecter deux blocs est un jeu symétrique, si le bloc ?b est

connecté au bloc ?b1, alors le bloc ?b1 est aussi connecté au bloc ?b. La règle vérifie que le jeu de connecter les dexu blocs n'a pas encore été déduit avec la prémisse ' absent (Jeu_binaire_bloc_apres (?c Connecter ?b ?b1 G))' et sa symétrique ' absent (Jeu_binaire_bloc_apres (?c Connecter ?b1 ?b G))'. Il peut alors déduire comme un fait nouveau que les deux blocs sont connectés avec la prémisse en conclusion 'ajoute (Jeu_binaire_bloc_apres (?c Connecter ?b ?b1 G)))'.

```
( nom ( Regle_jeu_apres_7 )
  premisses (  present ( Couleur ( ?c ) )
               present ( Coup ( ?i ) )
               present ( Nouveau_bloc ( ?b ) )
               present ( Jeu_binaire_bloc_apres ( ?c Connecter ?b ?b1 G ) )
               absent ( Jeu_binaire_bloc_intersection_apres ( ?c Connecter ?b1 ?i G ) )
               absent ( Jeu_binaire_bloc_intersection_apres ( ?c Connecter ?b1 ?i GI ) )
               absent ( Bloc_avant ( ?i ?b1 ) )
               absent ( Nouveau_bloc ( ?b1 ) ) )
  conclusions ( ajoute ( Jeu_binaire_bloc_intersection_apres ( ?c Connecter ?b1 ?i G ) ) ) )
```

Tableau Apprentissage Définition But Connecter au Vide

Le tableau Apprentissage Définition But Connecter au Vide donne la règle qui permet de déduire que le bloc dans lequel se trouve la pierre qui vient d'être jouée (la prémisse 'present (Nouveau_bloc(?b))') est connecté à un autre bloc (la prémisse 'present (Jeu_binaire_bloc_apres (?c Connecter ?b ?b1 G))') . L'intersection vide sur laquelle le coup a été joué est donc connectée au bloc ?b1. La règle peut donc alors déduire que le bloc ?b1 est connecté à l'intersection ?i avec la prémisse en conclusion 'ajoute (Jeu_binaire_bloc_intersection_apres (?c Connecter ?b1 ?i G))'.

```
( nom ( Regle_jeu_apres_10 )
  premisses (  present ( Couleur ( ?c ) )
               present ( Nombre_voisines ( ?i 2 ) )
               present ( Liberte_bloc_apres ( ?i ?b ) )
               present ( Couleur_intersection_apres ( ?i + ) )
               present ( Voisine ( ?i ?i1 ) )
               present ( Couleur_intersection_apres ( ?i1 ?c ) )
               present ( Voisine ( ?i ?i2 ) )
               intersections_differentes ( ?i1 ?i2 )
               present ( Couleur_intersection_apres ( ?i2 ?c ) )
               present ( Voisine ( ?i1 ?i3 ) )
               present ( Voisine ( ?i2 ?i3 ) )
               present ( Couleur_intersection_apres ( ?i3 ?c ) )
               absent ( Oeil_apres ( ?i ?b ) ) )
  conclusions ( ajoute ( Jeu_binaire_bloc_intersection_apres ( ?c Faire_oeil ?b ?i G ) )
               ajoute ( Oeil_apres ( ?i ?b ) ) ) )
```

Tableau Apprentissage Définition But Oeil Coin

Le tableau Apprentissage Définition But Connecter Oeil Coin donne la règle qui permet de déduire que le bloc ?b a un oeil sur l'intersection ?i. Pour cela, elle vérifie que l'intersection ?i est dans le coin avec la prémisse ' present (Nombre_voisines (?i 2))', puis que les deux voisines de ?i : ?i1 et ?i2 sont de la couleur ?c, et enfin que la voisine commune à ?i1 et ?i2 : ?i3 est aussi de la couleur ?c. On a donc défini un oeil dans le coin comme celui de la figure Apprentissage Oeil Coin. On peut remarquer que cette règle utilise la symétrie naturelle de la représentation à base de logique des prédicats pour définir les quatres yeux possibles dans le coin en une seule règle.

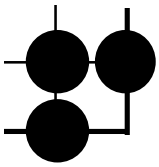


Figure Apprentissage Oeil Coin

Le système peut alors déduire que le but Faire_oeil est gagné sur l'intersection ?i pour le bloc ?b dont ?i est une liberté. Il le déduit avec la prémisse 'ajoute (Jeu_binaire_bloc_intersection_apres (?c Faire_oeil ?b ?i G)'.

Le système utilise des règles similaires pour définir l'oeil sur le bord et l'oeil au centre.

5.3.2 Résolution de problèmes et Explication

Les premières étapes de l'apprentissage sont la résolution d'un problème, la sélection de faits intéressants à expliquer, puis l'explication de ces faits. Ces étapes ont été décrites dans le chapitre sur les explications.

5.3.3 Détecter les coups forcés

Des heuristiques admissibles pour les buts du système de Go

En résolution de problèmes, une recherche est un tri-uplet {S,c,G}, où S est un ensemble d'états décrivant une situation du monde; c : S X S $\rightarrow$ R est une fonction de coût positive qui représente le coût d'application d'une action pour passer d'un état à un autre; et G $\in$ S un ensemble d'états désirables. Un problème instancié est un problème associé à un état initial. La fonction h* (s) est définie comme le coût minimal pour passer de l'état s à un but. Une fonction d'évaluation heuristique h (s) est une estimation de h* (s). Une heuristique admissible est une heuristique qui ne surestime jamais h* (s) : h (s) $\leq$ h* (s). Une bonne heuristique admissible est une heuristique qui a les plus grandes valeurs possibles.

Au Go, si on définit h* (s) comme le nombre minimal de coups nécessaires pour atteindre le but, une heuristique admissible pour le but **Prendre** est le nombre de libertés du bloc à prendre.

Cette utilisation d'heuristiques admissibles basée sur une abstraction des actions possibles a déjà été utilisée avec succès dans [Prieditis 1995].

Je définis la distance de Go-Manhattan comme étant la distance de Manhattan classique entre deux pierres, hormis le fait qu'un passage sur une pierre amie ne compte pas alors que le passage sur une intersection vide ou sur une pierre ennemie compte pour 1. Une heuristique admissible pour le but **Connecter** est la distance de Go-Manhattan minimale entre les deux blocs à connecter. Une amélioration de cette heuristique est de donner une valeur infinie aux pierres ennemies appartenant à un bloc statiquement vivant. On peut encore améliorer cette heuristique en ajoutant le cardinal de l'ensemble des libertés des blocs adverses traversés. Cette heuristique admissible pourrait être utilisée pour implémenter les concepts définis dans [Victorri 1992].

Il n'existe pas d'heuristique admissible pour les buts inverses des buts que nous venons de voir mais une heuristique admissible pour un but devient une heuristique "majorante" pour le but inverse.

Les jeux ont une hiérarchie, 'P' est plus petit que 'GI' qui est plus petit que 'G'. Cette hiérarchie signifie qu'un jeu calculé sur une petite surface est 'au moins' celui calculé. Ainsi, un jeu prouvé 'G' est effectivement 'G', un jeu prouvé 'GI' est soit 'GI' soit 'G', et un jeu calculé 'I' peut être 'P', 'GI' ou 'G'. En ce sens, les règles prouvées sont des heuristiques admissibles pour la détermination des jeux.

Utilisation d'heuristiques admissibles pour détecter les coups forcés

Dans le Tableau Apprentissage Règle Coup Forcé, on voit une règle qui permet de détecter un coup forcé pour Vivre. Cette règle utilise comme prémisses :

' present (Nombre_minimum_libertes_blocs_voisins_bloc_avant (?b ?n))' et
' superieur (?n 1)'

ce qui revient à dire que tous les blocs adjacents au bloc ?b ont plus d'une liberté. Quelque soit le bloc adjacent, une heuristique admissible décrite dans la section précédente nous informe qu'il faut donc au moins 2 coups pour le prendre. Or, le bloc ?b a une seule liberté. Le joueur à qui appartient le bloc ?b n'a donc pas le temps de prendre un bloc adjacent pour augmenter le nombre de libertés de ce bloc ?b. La seule option qui reste est de jouer sur la dernière liberté pour ajouter de nouvelles libertés. L'utilisation de cette heuristique admissible permet donc dans ce cas de détecter qu'il y a un et un seul coup forcé pour que le bloc ?b ne soit pas pris. Le but inverse de Prendre étant Vivre, le programme rend compte du coup forcé en concluant que le jeu Vivre pour le bloc ?b est IP et qu'il faut jouer sur ?i qui est la liberté du bloc ?b.

```
( nom ( Regle_jeu_avant_1 )
  premisses (   present ( Couleur ( ?c ) )
                present ( Couleur_intersection_avant ( ?i ?c ) )
                present ( Nombre_libertes_bloc_avant ( ?b 1 ) )
                present ( Nombre_minimum_libertes_blocs_voisins_bloc_avant ( ?b ?n ) )
                superieur ( ?n 1 )
                present ( Liberte_bloc_avant ( ?i ?b ) )
                present ( Coup_legal_avant ( ?i ?c ) ) )
                absent ( Coup_force_jeu_bloc_avant ( ?c Vivre ?b ?i IP ) ) )
  conclusions ( ajoute ( Coup_force_jeu_bloc_avant ( ?c Vivre ?b ?i IP ) ) ) )
```

Tableau Apprentissage Règle Coup Forcé

5.3.4 Généralisation

Après l'étape d'explication, le système dispose d'une règle qui ne contient que des constantes. Ces constantes correspondent à l'exemple sur lequel la règle s'est appliquée. **L'étape de généralisation consiste à transformer la règle qui s'applique spécifiquement sur un exemple en une règle plus générale qui pourra s'appliquer sur beaucoup plus d'exemples**. Pour cela, le système remplace certaines constantes par des variables. Il ne faut remplacer une constante par une variable qu'à bon escient pour éviter d'être trop général et de créer des règles fausses. L'option que j'ai retenue est de ne généraliser que les constantes qui sont des instanciations de variables, et non pas les vraies constantes qui sont aussi des constantes dans les règles. Cette façon de généraliser est propre à mon système. Je ne connais aucune autre description de ce mécanisme qui est pourtant vital pour créer des règles fiables.

5.3.4.1 Différence entre les constantes et les variables instanciées

Lors du retour dans la trace du module d'explication, celui-ci a retenu une information capitale pour le module de généralisation, à savoir la provenance des constantes contenues dans les faits composant l'explication. La généralisation procédera de façon différente pour les constantes provenant de constantes et les constantes provenant d'instanciations de variables contenues dans les règles de la théorie du domaine.

Dans le cas où un fait a été déduit deux fois, une fois avec une variable instanciée et une fois avec une constante, la priorité va à la constante. Dans la règle finale, l'objet correspondant à la constante sera partout une constante. Ceci afin de ne pas créer des règles trop générales.

5.3.4.2 Le mécanisme de généralisation dépend du type des constantes

Le type entier et le type réel ne sont pas généralisés de la même façon que les autres types. Deux constantes entières ou réelles peuvent être égales sans pour autant que les variables dont elles sont issues soient identiques. Ceci est aussi vrai pour certains autres types (intersections, blocs), mais le problème avec la théorie du jeu de Go vient de ce qu'un très grand nombre de règles utilisent les intersections et les blocs. Utiliser une variable différente à chaque règle pour chaque intersection ou pour chaque bloc amène à créer trop de généralisations et rend le processus de généralisation beaucoup trop coûteux. On se contente donc pour les types différents des types entier et réel de généraliser de la même façon deux instanciations identiques même si elles proviennent de variables différentes. En pratique cela n'a pas une grande incidence sur la qualité de la généralisation. Ce n'est pas le cas pour le type entier et le type réel qui sont beaucoup plus sensibles, et généraliser de la même façon deux entiers parce qu'ils sont égaux amène à une mauvaise généralisation.

En résumé, les variables entières et réelles, qui ont la même instanciation mais qui sont à l'origine différentes, ont des généralisations différentes. Alors que pour les autres types, si deux variables instanciées ont la même instanciation, elles ont aussi la même généralisation, ceci pour des raisons pratiques liées à la théorie du jeu de Go.

5.3.4.3 Exemple

Reprenons l'explication du chapitre explication dans la section Mécanisme de retour dans la trace. Cette explication avait permis de trouver un sous-ensemble de la base de faits expliquant la déduction du fait :

'ajoute (Coup_jeu_binaire_bloc_regresse_avant (@ Connecter b2 b3 i11 GI))'

Cette explication est rappelée dans le Tableau Apprentissage Explication.

present (Tour_avant (@))
present (Couleur (@))
present (Coup (i11))
blocs_differents (b2 b3)
present (Couleur_bloc_avant (b2 @))
present (Liberte_bloc_avant (i11 b2))
present (Liberte_bloc_avant (i11 b3))
present (Couleur_bloc_avant (b3 @))
present (Nombre_libertes_bloc_avant (b2 3))
superieur (3 1 cte)
ajoute (Coup_jeu_binaire_bloc_regresse_avant (@ Connecter b2 b3 i11 GI))

Tableau Apprentissage Explication

La généralisation de l'explication en utilisant la trace de la résolution de problème permet de créer la règle du Tableau Apprentissage Explication Généralisée. On peut remarquer que les prémisses :

present (Nombre_libertes_bloc_avant (b2 3))
superieur (3 1)

sont généralisée de la façon suivante :

present (Nombre_libertes_bloc_avant (?b ?n))
superieur (?n 1)

En effet, dans le fait Nombre_libertes_bloc_avant (b2 3), la valeur 3 est une variable instanciée qui vient d'une règle de la théorie du domaine : la règle Regle_bloc_ote_6 qui est instanciée dans le tableau Explication Règles Déclenchées. alors que dans cette même règle, la valeur 1 est une

constante. Le 1 vient d'une constante alors que le 3 vient d'une variable instanciée. Le 1 ne sera donc pas généralisé alors que le 3 sera transformé en variable. De plus la variable contenue dans la prémisse 'present (Nombre_libertes_bloc_avant (?b ?n))' et dans la prémisse 'superieur (?n 1)' est la même dans la même règle source, on généralisera donc les deux instanciations avec la même variable.

Ce ne serait pas le cas si les deux '3' venaient de variables différentes dans des règles différentes. Contrairement aux autres types, les types Entier et Réel peuvent avoir des généralisations différentes avec une même instanciation.

```
( premisses (   present (Tour_avant ( ?c )  )
                present ( Couleur ( ?c )  )
                present (Coup( ?i )  )
                blocs_differents ( ?b ?b1 )
                present ( Couleur_bloc_avant ( ?b ?c )  )
                present ( Liberte_bloc_avant ( ?i ?b )  )
                present ( Liberte_bloc_avant ( ?i ?b1 )  )
                present ( Couleur_bloc_avant ( ?b1 ?c )  )
                present ( Nombre_libertes_bloc_avant ( ?b ?n )  )
                superieur ( ?n 1 )
                blocs_differents ( ?b1 ?b )
            )
 conclusions   (
        ajoute ( Coup_jeu_binaire_bloc_regresse_avant ( ?c Connecter ?b ?b1 ?i GI ) ) ) )
```

Tableau Apprentissage Explication Généralisée

5.3.5 Opérations sur les règles créées

Les opérations effectuées sur les règles créées sont la simplification et l'ajout de symétries. Une autre manipulation opérée sur les règles créées par généralisation d'explications de résolution de problèmes est la compilation logique (qui est différente de la compilation des règles logiques en C++). La compilation logique est très importante si l'on veut pouvoir utiliser efficacement les règles apprises par mon système. Une description détaillée de la compilation est donnée dans le chapitre sur la compilation.

5.3.5.1 Simplifications

Il y a deux sortes de simplifications à l'intérieur des règles. La première simplification consiste à compiler les expressions, elle est décrite dans le chapitre sur la compilation. La deuxième simplification utilise des connaissances spécifiques au jeu de Go. Elle permet d'ôter des prémisses inutiles. Par exemple, la règle du tableau Apprentissage Métarègle Simplification permet d'enlever le test sur les intersections différentes si ces intersections sont aussi voisines. En effet deux intersections voisines sont toujours différentes sur un damier de Go. Les métavariables ?var et ?var1 s'instancient sur des variables de type Intersection.

```
(       nom             (       Metaregle_Simplification_10 )
        premisses       (
                          regle ( ?r )
                          condition ( ?r present ( Voisine ( ?var ?var1 ) ) )
                          condition ( ?r intersections_différentes ( ?var ?var1 ) )
                         )
conclusions             ( ote_condition ( ?r intersections_différentes ( ?var ?var1 ) ) )
)
```

Tableau Apprentissage Métarègle Simplification

5.3.5.2 Symétries

L'étape suivante de transformation des règles est aussi une étape dépendante du domaine. Elle consiste à symétriser les prémisses qui peuvent l'être. Cette symétrisation est nécessaire pour que le programme soit capable d'unifier efficacement deux règles. Sinon, si deux règles différent d'une seule prémisse, et que les prémisses qui diffèrent sont symétriques, les deux règles sont tenues pour différentes. Comme il est préférable que ces deux règles soient tenues pour égales, le programme symétrise toutes les prémisses qui peuvent l'être à l'intérieur des règles.

```
(       nom             (       Metaregle_symétrie_14 )
        premisses       (
                         regle ( ?r )
                         condition ( ?r present ( Blocs_voisins_avant ( ?var ?var1 ) ) )
                        )
        conclusions  ( ajoute_condition ( ?r present ( Blocs_voisins_avant ( ?var1 ?var ) ) ) )
)
```

Tableau Apprentissage Métarègle Symétrie

La métarègle de symétrisation dut tableau Métarègle Symétrie instancie avec le métaprédicat 'regle (?r)' toutes les règles de la base de règles dans la variable ?r de type Règle. Pour chacune de ces règles, le système vérifie avec le métaprédicat 'condition (?r present (Blocs_voisins_avant (?var ?var1))' que la prémisse 'present (Blocs_voisins_avant (?var ?var1)' est présente dans la règle. Les métavariables ?var et ?var1 sont des variables de type Variable et peuvent s'instancier dans n'importe quelle autre variable, dans le cas de cette règle, elles s'instancieront dans des variables de type Bloc. Pour toutes les règles contenant la prémisse 'present (Blocs_voisins_avant (?var ?var1)', le système ajoute en condition de la règle la prémisse 'present (Blocs_voisins_avant (?var1 ?var)"

5.3.5.3 Ordonnancement

Afin de matcher les règles beaucoup plus rapidement, des métarègles d'ordonnancement sont utilisées. Elles sont décrites en détail dans le chapitre sur la compilation.

5.3.6 Insertion dans la base

Après avoir créé les nouvelles règles, le système les insère dans la base de règles. Pour cela, il commence par ôter de la base toutes les règles moins générales que la nouvelle règle. Il vérifie ensuite que la nouvelle règle n’est pas moins générale qu’une règle déjà existante. Si c'est le cas, il insère la nouvelle règle dans la base.

5.3.7 Bases d’exemples

5.3.7.1 Bases créées à la main

Pour donner des problèmes résolus à mon système, j'ai utilisé le livre Graded Go Problems For Beginners Volume 1. Une autre source importante de problèmes dont je me suis servi est celle des parties jouées contre d’autres programmes de Go ou contre moi. J'analyse les positions où mon système a fait des erreurs, puis je donne le problème correspondant à l'erreur au programme qui se charge de se modifier lui-même pour ne plus commettre la même erreur la fois suivante.

5.3.7.2 Bases créées automatiquement

Créer des bases de problèmes est une tâche qui demande beaucoup de temps. Aussi, mon système a la possibilité de créer lui-même ses problèmes en détectant lui-même ses erreurs. Pour cela, il utilise les règles données dans le chapitre sur les explications qui permettent de détecter les coups surprenants. Pour détecter les coups surprenants, mon programme peut utiliser sa version compilée s'il ne cherche pas à apprendre les coups mais simplement à les détecter. Cette activité est donc très

peu coûteuse et lui permet de créer automatiquement ses bases de problèmes. Une version de mon système est prête à tourner sur le World Wide Web de façon à ce qu'il puisse faire beaucoup de parties et qu'il puisse ainsi se créer automatiquement une grande base de connaissances sans pour autant détériorer son temps de réponse.

5.3.8 Le problème de l'utilité

Mon approche qui consiste à transformer une partie de l'activité de recherche arborescente en activité de filtrage d'une base de règles est différente des approches de SOAR ou de PRODIGY. SOAR [Laird 1986] [Tambe 1994] sait créer des chunks de connaissances, mais ces chunks représentent des macro opérateurs alors que mon système apprend aussi à réduire le nombre de coups à envisager à chaque noeud de l'arbre de recherche. PRODIGY [Minton 1988, 1989] crée des règles de contrôle de la recherche mais les règles qu'il crée sont heuristiques alors que les règles que je crée sont toujours vraies.

Mon système, qui filtre une base de règles apprises, marche mieux qu'une recherche arborescente pour 3 raisons principales :

1 - Une recherche arborescente recalcule très souvent les mêmes choses. Mon programme ne vérifie qu'une seule fois les conditions, même si celles-ci correspondent à plusieurs vérifications dans la recherche arborescente.

2 - L'ordre de vérification des conditions est très contraint dans la recherche arborescente par l'ordre dans lequel les coups sont joués. Par contre l'ordre dans lequel sont vérifiées les conditions est libre dans mon programme. Il peut donc mieux optimiser cet ordre pour faire le moins de travail possible et donc gagner du temps.

3 - La recherche arborescente passe beaucoup de temps à tester des coups qui ne marchent pas. Mon programme ne sélectionne que les conditions qui font marcher un coup.

Il est toutefois vrai que certaines règles créées par mon système se révèlent de peu d'utilité car très peu souvent vérifiées ou utilisées. Une composante qu'il est nécessaire d'ajouter à mon système pour qu'il ne soit pas engorgé par un trop grand nombre de connaissances inutiles est la composante d'oubli. Ce module n'a pas encore été implémenté pour le système d'apprentissage basé sur la logique des prédicats. Les solutions à ce problème décrites dans [Minton 1988] et [Markovitch 1993] semblent permettre l'oubli des connaissances jugées inutiles. Elles sont basées sur des statistiques concernant l'utilité des règles, collectées sur une base de problème. L'utilité est calculée à partir de la fréquence d'application d'une règle et de son temps moyen de matchage [Minton 1990].

Actuellement, la solution que j'ai retenue est de limiter l'apprentissage aux problèmes que j'ai sélectionné. Pour rendre le système complètement autonome, il faudrait lui ajouter un module de calcul de l'utilité des règles apprises.

5.4 Résultats

L'apprentissage de règles en utilisant la logique des prédicats est plus efficace que l'apprentissage de patterns géométriques car les règles apprises sont beaucoup plus générales.

Afin de tester l'efficacité de l'apprentissage, j'ai sélectionné 20 problèmes du livre pour débutants Graded Go Problems for Beginners Volume 1. Pour tous ces problèmes j'ai ajouté des séquences qui correspondaient aux solutions cherchées. En rejouant ces séquences sur les cinq premiers problèmes, mon programme a découvert et appris 100% des solutions à ces problèmes. Il a aussi, par la même occasion, résolu des problèmes qui n'étaient pas posés. Les règles apprises ont été ensuite testées sur les 15 problèmes suivants. Le système a donné la bonne réponse des problèmes dans 80% des cas. Il a aussi donné des réponses qui étaient justes mais ne faisaient pas partie du test. Les 20% restants correspondent à des problèmes pour lesquels il n'a pas appris de règles répondant directement au problème. Toutefois, dans ces cas là, il n'a pas donné de solution. **Une**

propriété très importante de mon système d'apprentissage est que les règles apprises ne donnent jamais de solutions fausses à un problème. On peut considérer ces règles comme des théorèmes du jeu de Go que mon système a démontrés.

Une autre caractéristique importante de mon système d'apprentissage utilisant la logique des prédicats est qu'il peut être très facilement utilisé dans d'autres jeux ou dans d'autres domaines que les jeux.

5.5 Bibliographie

[Cazenave 1994] T. Cazenave, *Système Apprenant à Jouer au Go*. 2ème Rencontres Nationales des Jeunes Chercheurs en Intelligence Artificielle, Marseille, 1994.

[Cazenave 1996a] T. Cazenave, *Learning to Forecast by Explaining the Consequences of Actions*. First International Workshop on Machine Learning, Forecasting and Optimization, Madrid, 1996.

[Cazenave 1996d] T. Cazenave, *Automatic Acquisition of Tactical Go Rules*. Third International Game Programming Workshop, Tokyo, 1996.

[Dejong 1986] - G. Dejong, R. Mooney. *Explanation Based Learning : an alternative view.* Machine Learning 2, 1986.

[Enderton 1991] - H. D. Enderton. *The Golem Go Program*. Carnegie Mellon University, CMU-CS-92-101, 1991. Rapport récuperable sur ftp.pasteur.fr.

[Enzenberger 1996] - Markus Enzenberger. *The Integration of A Priori Knowledge into a Go Playing Neural Network.* Third Game Programming Workshop in Japan, Hakone, Septembre 1996.

[Laird 1986] - J. Laird, P. Rosenbloom, A. Newell. *Chunking in SOAR : An Anatomy of a General Learning Mechanism*. Machine Learning 1 (1), 1986.

[Lee 1988] - Kai-Fu Lee and Sanjoy Mahajan. *A pattern classification approach to evaluation function learning*. Artificial Intelligence, 36:1-25, 1988.

[Lenat 1983] - D. Lenat. *EURISKO: A program that learns new heuristics and domain concepts*. Artificial Intelligence, 21:61-98, 1983.

[Levinson 1991] - Jeffrey Gould, Robert Levinson. *Method Integration for Experience-Based Learning.* University of California Santa-Cruz, Rapport UCSC-CRL-91-27, 1991.

[Levinson 1992] - Jeffrey Gould, Robert Levinson. *Experience-Based Adaptive Search.* University of California Santa-Cruz, Rapport UCSC-CRL-92-10, 1992.

[Levinson 1993] - Robert Levinson. *Exploiting the Physics of State-Space Search.* University of California Santa-Cruz, Rapport UCSC-CRL-93-38, 1993.

[Markovitch 1993] - S. Markovitch, P. D. Scott. *Information Filtering: Selection Mechanisms in Learning Systems*. Machine Learning, 10, pp. 113-151, 1993.

[Minton 1984] - S. Minton. *Constraint-Based Generalization - Learning Game-Playing Plans from Single Examples*. Proceedings of the Fourth National Conference on Artificial Intelligence, 251-254. Los Altos, William Kaufmann, 1984.

[Minton 1988] - S. Minton. *Learning Search Control Knowledge - An Explanation Based Approach*. Kluwer Academic, Boston, 1988.

[Minton 1989] - S. Minton, J. Carbonell, C. Knoblock, D. Kuokka, O. Etzioni, Y. Gil. *Explanation-Based Learning : A Problem Solving Perspective*. Artificial Intelligence 40, 1989.

[Minton 1990] - S. Minton. *Quantitative Results Concerning the Utility of Explanation-Based Learning*. Artificial Intelligence 42, 1990.

[Mitchell 1986] - T. M. Mitchell, R. M. Keller, S. T. Kedar-Kabelli. *Explanation-based Generalization : A unifying view*. Machine Learning 1 (1), 1986.

[Pell 1991] - Barney Pell. *Exploratory Learning in the Game of GO*. In D.N.L. Levy and D.F. Beal, editors, Heuristic Programming in Artificial Intelligence 2 - The Second Computer Olympiad. Ellis Horwood, 1991.

[Pell 1993] - Barney Pell. *Logic Programming for General Game-Playing*. Proceedings of the workshop on Knowledge Compilation and Speedup Learning, at Machine Learning Conference, Amherst, Mass., 1993.

[Pitrat 1976] - J. Pitrat. *Realization of a Program Learning to Find Combinations at Chess*. Computer Oriented Learning Processes, Simon J. Ed., Noordhoff, 1976.

[Pitrat 1990] - J. Pitrat. *Métaconnaissances*. Hermès, 1990.

[Pompidor 1992] - P. Pompidor. *Apprentissage Symbolique par Exemples et Contre-Exemples Géométrisables en Prise de Décisions*. Thèse de Doctorat de l'Université des Sciences et Techniques du Languedoc - Montpellier II, 1992.

[Prieditis 1995] - A. Prieditis. *Quantitatively relating abstractness to the accuracy of admissible heuristics*. Journal of Artificial Intelligence, 74, pp165-175, 1995.

[Puget 1987] - J. F. Puget. *Goal Regression with Opponent*. Progress in Machine Learning, Sigma Press, Wilmslow, 1987.

[Quinlan 1984] - J. Ross Quinlan. *Learning efficient classification procedures and their application to chess end games.* Machine Learning, pp 463-482, 1984.

[Samuel 1959] - A. Samuel, *Some studies in machine learning using the game of checkers*, IBM Journal of Research and Development, 3, pp 210-229, 1959.

[Samuel 1967] - A. Samuel, *Some studies in machine learning using the game of checkers - recent progress*, IBM Journal of Research and Development, 11, pp 601-617, 1967.

[Schraudolph 1994] - N. Schraudolph, *Temporal Difference Learning of Position Evaluation in the Game of Go*, Neural Information Pocessing Systems 6, Morgan Kaufmann, 1994. Accessible par ftp bsdserver.ucsf.edu.

[Simon 1984] - H. Simon, *Why should machine learn?*, Machine Learning : an Artificial Intelligence approach, Springer Verlag 1984

[Stoutamire 1991] - D. Stoutamire, *Machine learning, Game Play, and Go*. MS thesis, Case Western Reserve University, 1991.

[Tadepalli 1989] - P. Tadepalli. *Lazy Explanation-Based Learning: A Solution to the Intractable Theory Problem*. IJCAI 1989, pp. 694-700, 1989.

[Tambe 1994] - M. Tambe, P. S. Rosenbloom. *Investigating production system representations for non combinatorial match*. Artificial Intelligence 68, pp 155-199, 1994.

[Tesauro 1989] - G. Tesauro, T. Sejnowski, *A parallel network that learn to play backgammon*, Artificial Intelligence, 39, pp 357-390, 1989.

[Victorri 1993] - B. Victorri, *Eléments d'une Théorie Géométrique du Jeu de Go*, Second Cannes/Sophia-Antipolis Go Research Day, 1993.

[van Harmelen 1988] - F. van Harmelen, A. Bundy. *Explanation based generalisation = partial evaluation*. Artificial Intelligence 36, pp 401-412, 1988.

[Waterman 1970] - D. Waterman, *Generalization learning techniques for automating the learning of heuristics*, Artificial Intelligence 1 pp 121-170, 1970.

Table des Matières du Chapitre 6

6 LA COMPILATION 101

6.1 L'ordonnancement des liste de conditions 102
6.1.1 Le mécanisme de filtrage 102
6.1.2 Ordonner les prémisses 102

6.2 Les coupes dans le graphe d'unification 102
6.2.1 Le métaprédicat absent 102

6.3 L'ordonnancement des listes de règles 106

6.4 L'Optimisation 106
6.4.1 Optimisation des calculs 106
6.4.2 Evaluation Partielle 107

6.5 La compilation en C++ 108
6.5.1 Equivalence instanciation/boucle for 108
6.5.2 Equivalence instanciation/affectation 108
6.5.3 Equivalence test/if 110
6.5.4 Equivalence absent/parcours d'arbre 111
6.5.5 Traduction d'une règle 111
6.5.6 Traduction d'une base de règles 111

6.6 Conclusion 111

6.7 Bibliographie 112

6 La Compilation

Un aspect très important des systèmes d'apprentissage déductif est l'efficacité de l'utilisation des règles apprises. Ce problème a été soulevé par J. Pitrat avec son programme Appech [Pitrat 1976] [Pitrat 1990], il a été aussi rencontré plus tard par S. Minton et a été appelé le problème de l'utilité [Minton 1988]. Une bonne formalisation de ce problème et des solutions possibles ont été données dans [Markovitch 1993]. Mon système est différent des systèmes de Minton et de Pitrat qui apprenaient à jouer aux Echecs. Ces systèmes apprennent des méta-règles de contrôle de la recherche, c'est-à-dire des règles qui donnent des conseils pour choisir entre les branches à chaque étape d'une résolution de problème. Le coût de filtrage des métarègles peut alors être plus grand que le temps gagné par les coupes dans l'arborescence résultantes de leurs bons choix. Mon système n'apprend pas de métarègles mais des règles sur l'état des jeux. Son apprentissage consiste à remplacer le développement d'arborescences par le filtrage d'une base de règles. Il est très important que les connaissances apprises par le système soient utilisées de manière efficace. Le système doit donc savoir s'auto-modifier pour utiliser ses connaissances à bon escient.

Des recherches pour améliorer le filtrage portent sur l'incrémentalité du filtrage [Forgy 1982] [Doorenbos 1995]. D'autres recherches portent sur l'auto-observation d'un moteur d'inférence afin d'éviter les actions inutiles [Parchemal 1988]. Une autre option est la transformation de règles en logique du premier ordre vers un langage impératif [Porcheron 1990]. Quelques heuristiques pour ordonner les prémisses d'une règle sont données dans [Ishida 1988], elles consistent à faire les choix les plus contraignants en premier. Ces heuristiques rappellent celles utilisées dans [Laurière 1976] pour ordonner les essais dans la résolution de tâches combinatoires. Une autre approche de ce problème semble être abordée dans le cadre de la métaprogrammation logique [Barklund 1994] qui étudie les programmes logiques qui transforment d'autres programmes logiques afin de les rendre plus efficaces.

Mon approche du problème de l'utilité est d'utiliser plusieurs étapes dans la transformation d'un programme. Pour cela, j'ai créé un compilateur de bases de règles. Le but de la compilation est de transformer un programme en un autre programme qui donne les mêmes résultats plus rapidement. Mon compilateur prend en entrée une liste de règles représentées en utilisant un formalisme de logique du premier ordre (Cf Chapitre Représentation des Connaissances). Il donne en sortie un programme C++.

Mon système utilise quatre types de compilation. Les listes de prédicats forment les conditions des règles qu'il utilise : le premier type de compilation consiste à réordonner ces listes de façon que le filtrage se fasse plus efficacement. Le deuxième type de compilation se fait par l'insertion de coupes dans les graphes d'unification des conditions des règles. Le troisième type de compilation réordonne les règles de façon a ne pas redéduire des faits déjà connus. Enfin, le quatrième type de compilation transforme des bases de règles exprimées en logique des prédicats vers un programme C++.

6.1 L'ordonnancement des liste de conditions

6.1.1 Le mécanisme de filtrage

Mon système a la possibilité de s'observer lorsqu'il unifie des règles [Cazenave 1996b]. Il compte le nombre de fois qu'un prédicat a été unifié. Cette information est très utile pour détecter d'où viennent les inefficacités des règles. Le tableau Compilation Espionnage donne les informations récoltées lors de l'unification d'une règle avec une base de faits correspondant à un Goban 9x9. Chaque prémisse et chaque conclusion sont suivies du nombre de fois qu'elles ont été unifiées. Le système compte aussi le nombre de noeuds de l'arbre d'unification et le nombre de nouveaux faits déduits par une règle.

6.1.2 Ordonner les prémisses

Le tableau Compilation Règle sans Ordre montre un exemple de règle apprise par le système avant que les prémisses soient ordonnées. Le nombre de noeuds nécessaires à son unification avec la base de faits rend cette unification impossible dans des temps raisonnables. L'ordre d'unification des prémisses d'une règle a une très forte influence sur son temps d'unification. Le tableau Compilation Règle sans Absent donne le coût d'unification d'une règle une fois les prémisses ordonnées par le système. On passe de 808 206 733 à 111 289 noeuds, ce qui représente pour cette règle une unification 7262 fois plus rapide. A temps de réponse du système constant, celui-ci peut donc unifier 7000 fois plus de règles, ce qui lui donne une différence de niveau de jeu assez conséquente.

6.2 Les coupes dans le graphe d'unification

6.2.1 Le métaprédicat absent

Un autre mécanisme utilisé pour optimiser le filtrage est l'insertion de métaprédicats vérifiant qu'un fait n'a pas déjà été déduit après chaque nouvelle instanciation multiple. Ceci afin de ne pas déduire deux fois la même chose avec des conditions différentes. Cette optimisation n'est valide que lorsqu'on cherche à optimiser le filtrage en utilisation finale des règles. Elle n'est plus intéressante lorsqu'on veut effectuer des explications puisqu'elle privilégie une explication sur d'autres explications possibles. Privilégier une explication sur une autre amène à de mauvaises performances en généralisation. Les coupes dans l'arbre de filtrage ne sont donc effectuées que lorsque le programme utilise ses connaissances sans avoir à fournir d'explications sur ses déductions.

L'observation attentive des unifications effectuées par le programme m'a montré que celui-ci passait beaucoup de temps à déduire à nouveau des conclusions qu'il avait déjà déduites au cours de l'unification d'une règle. Ceci m'a amené à créer un mécanisme qui lui permet d'optimiser lui-même ses règles afin qu'elles ne déduisent qu'une seule fois le même fait. Pour cela, mon programme doit insérer dans ses règles des prémisses qui coupent automatiquement les chemins redondants dans le graphe d'unification. Il insère ainsi automatiquement des prémisses vérifiant l'absence de la

conclusion après chaque instanciation multiple de variable. Il vérifie aussi, avant d'insérer ces prémisses que toutes les variables présentes en conclusion de la règle sont instanciées. Si au moins une variable en conclusion n'est pas encore instanciée, il ne peut pas insérer la prémisse de coupe car il ne sait pas encore quel fait aurait déjà pu être déduit.

Ceci m'a amené à faire faire au système le choix de donner la priorité à l'instanciation des variables contenues dans les conclusions, de façon à ce que le maximum de coupes soit faites dans le graphe d'instanciation. Plus les variables contenues dans la conclusion sont instanciées tôt dans la règle, plus on peut vérifier que l'unification de la règle est utile, c'est à dire qu'elle n'amène pas à une conclusion déjà faite.

```
Nombre_de_noeuds 55315
Nombre_de_faits_déduits 208

( premisses     (
                present ( Couleur ( ?c ) ) 1
                present ( Couleur_opposees ( ?c1 ?c ) ) 2
                present ( Couleur ( ?c1 ) ) 2
                present ( Nombre_voisines_couleur_avant ( ?i + 4 ) ) 2
                present ( Nombre_voisines ( ?i 4 ) ) 76
                present ( Nombre_Blocs_voisins_avant ( ?i 0 ) ) 76
                present ( Couleur_intersection_avant ( ?i + ) ) 76
                present ( Voisine ( ?i ?i4 ) ) 76
                present ( Couleur_intersection_avant ( ?i4 + ) ) 304
                present ( Voisine ( ?i ?i6 ) ) 304
                present ( Couleur_intersection_avant ( ?i6 + ) ) 1216
                intersections_differentes ( ?i4 ?i6 ) 1216
                present ( Voisine ( ?i ?i2 ) ) 912
                present ( Couleur_intersection_avant ( ?i2 + ) ) 3648
                intersections_differentes ( ?i6 ?i2 ) 3648
                intersections_differentes ( ?i4 ?i2 ) 2736
                present ( Voisine ( ?i2 ?i1 ) ) 1824
                present ( Couleur_intersection_avant ( ?i1 + ) ) 7044
                present ( Nombre_voisines ( ?i1 4 ) ) 6912
                present ( Nombre_voisines_couleur_avant ( ?i1 ?c1 0 ) ) 5724
                present ( Voisine ( ?i4 ?i1 ) ) 5562
                intersections_differentes ( ?i1 ?i ) 2676
                present ( Nombre_Blocs_voisins_couleur_avant ( ?i1 ?c 0 ) ) 852
                present ( Voisine ( ?i ?i7 ) ) 832
                absent ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i ?i1 ?i GIII ) ) 3328
                present ( Couleur_intersection_avant ( ?i7 + ) ) 328
                intersections_differentes ( ?i4 ?i7 ) 328
                intersections_differentes ( ?i6 ?i7 ) 222
                intersections_differentes ( ?i2 ?i7 ) 114
                present ( Voisine ( ?i1 ?i8 ) ) 108
                absent ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i ?i1 ?i GIII ) ) 432
                present ( Couleur_intersection_avant ( ?i8 + ) ) 170
                intersections_differentes ( ?i4 ?i8 ) 170
                intersections_differentes ( ?i2 ?i8 ) 166
                present ( Couleur_intersection_avant ( ?i8 ?c2 ) ) 112
                present ( Voisine ( ?i1 ?i3 ) ) 112
                absent ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i ?i1 ?i GIII ) ) 448
                present ( Couleur_intersection_avant ( ?i3 + ) ) 448
                intersections_differentes ( ?i4 ?i3 ) 448
                intersections_differentes ( ?i8 ?i3 ) 336
                intersections_differentes ( ?i2 ?i3 ) 224
                present ( Voisine ( ?i4 ?i5 ) ) 112
                absent ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i ?i1 ?i GIII ) ) 448
                present ( Couleur_intersection_avant ( ?i5 + ) ) 192
                intersections_differentes ( ?i ?i5 ) 174
                intersections_differentes ( ?i1 ?i5 ) 114
            )
  conclusions   ( ajoute ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i ?i1 ?i GIII ) ) 104
                  ajoute ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i1 ?i ?i1 GIII ) ) 104 )
)
```

Tableau Compilation Espionnage

```
Nombre_de_noeuds 808206733
```

```
Nombre_de_faits_déduits 104
( premisses    (
            present ( Voisine ( ?i1 ?i3 ) ) 1
            present ( Couleur_intersection_avant ( ?i + ) ) 288
            present ( Voisine ( ?i ?i4 ) )  21600
            present ( Couleur_intersection_avant ( ?i8 + ) ) 76800
            present ( Couleur ( ?c ) ) 5836800
            present ( Voisine ( ?i4 ?i5 ) ) 11673600
            present ( Couleur_opposees ( ?c1 ?c ) ) 41558016
            present ( Couleur ( ?c1 ) ) 41558016
            present ( Nombre_voisines_couleur_avant ( ?i + 4 ) ) 41558016
            present ( Couleur_intersection_avant ( ?i2 + ) ) 41558016
            present ( Voisine ( ?i1 ?i8 ) ) 20009416
            present ( Nombre_voisines_couleur_avant ( ?i1 ?c1 0 ) ) 988120
            present ( Nombre_Blocs_voisins_avant ( ?i 0 ) ) 963722
            present ( Couleur_intersection_avant ( ?i4 + ) ) 963722
            present ( Couleur_intersection_avant ( ?i6 + ) ) 904232
            intersections_differentes ( ?i4 ?i6 ) 68721706
            present ( Voisine ( ?i ?i2 ) ) 65328042
            present ( Nombre_voisines ( ?i1 4 ) ) 3266402
            intersections_differentes ( ?i6 ?i2 ) 1572712
            intersections_differentes ( ?i4 ?i2 ) 1553296
            present ( Couleur_intersection_avant ( ?i7 + ) ) 1534120
            present ( Voisine ( ?i2 ?i1 ) ) 116593074
            intersections_differentes ( ?i4 ?i8 ) 115153652
            present ( Couleur_intersection_avant ( ?i1 + ) ) 113732002
            present ( Voisine ( ?i4 ?i1 ) ) 106711508
            present ( Voisine ( ?i ?i6 ) ) 5269704
            present ( Nombre_voisines ( ?i 4 ) ) 260232
            intersections_differentes ( ?i1 ?i ) 260232
            present ( Nombre_Blocs_voisins_couleur_avant ( ?i1 ?c 0 ) ) 257020
            present ( Voisine ( ?i ?i7 ) ) 222116
            intersections_differentes ( ?i4 ?i7 ) 10968
            present ( Couleur_intersection_avant ( ?i3 + ) ) 10834
            intersections_differentes ( ?i4 ?i3 ) 10164
            intersections_differentes ( ?i8 ?i3 ) 10040
            intersections_differentes ( ?i2 ?i3 ) 9916
            present ( Couleur_intersection_avant ( ?i5 + ) ) 9792
            intersections_differentes ( ?i ?i5 ) 9188
            intersections_differentes ( ?i1 ?i5 ) 9074
            present ( Couleur_intersection_avant ( ?i8 ?c2 ) ) 8962
            intersections_differentes ( ?i6 ?i7 ) 8962
            intersections_differentes ( ?i2 ?i7 ) 6812
            intersections_differentes ( ?i2 ?i8 ) 4912  )
  conclusions   ( ajoute ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i ?i1 ?i GIII ) ) 3248
                  ajoute ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i1 ?i ?i1 GIII ) ) 3248 ) )
```

Table Compilation Règle sans Ordre

```
( nom            (      Metaregle_ordonne_11 )
  premisses     (
                  regle ( ?r )
                  condition ( ?r present ( Voisine ( ?var ?var1 ) ) )
                  instanciee ( ?var )
                  instanciee ( ?var1 )
                  condition ( ?r present ( Voisine ( ?var2 ?var1 ) ) )
                  instanciee ( ?var2 )
                  variables_differentes ( ?var ?var2 )
                  priorite ( ?r present ( Voisine ( ?var2 ?var1 ) ) ?reel )
                  superieur_reel ( ?reel 0.04 )
                 )
   conclusions  ( affecte_priorite ( ?r present ( Voisine ( ?var2 ?var1 ) ) 0.04 )  ) )
```

Tableau Compilation Métarègle Ordonne

Mon système sait donc modifier ses propres règles pour qu'elles ne déduisent pas plusieurs fois le même fait. Il fait cette optimisation de façon efficace en donnant une priorité plus grande aux prémisses contenant des variables présentes dans la conclusion.

L'insertion des métaprédicats 'absent' permet de doubler la vitesse d'unification. L'unification coûte 111 289 noeuds dans le Tableau Compilation Règle sans Absent lorsque les coupes dans le graphe d'unification ne sont pas faites. L'unification ne coûte plus que 55 315 noeuds dans le tableau Compilation Espionnage lorsque le système s'est rajouté des métaprédicats qui par la suite lui font faire les coupes.

```
Nombre_de_noeuds 111289
Nombre_de_faits_déduits 104

( premisses    (
            present ( Couleur ( ?c ) ) 1
            present ( Couleur_opposees ( ?c1 ?c ) ) 2
            present ( Couleur ( ?c1 ) ) 2
            present ( Nombre_voisines_couleur_avant ( ?i + 4 ) ) 2
            present ( Nombre_voisines ( ?i 4 ) ) 76
            present ( Nombre_Blocs_voisins_avant ( ?i 0 ) ) 76
            present ( Couleur_intersection_avant ( ?i + ) ) 76
            present ( Voisine ( ?i ?i4 ) ) 76
            present ( Couleur_intersection_avant ( ?i4 + ) ) 304
            present ( Voisine ( ?i ?i6 ) ) 304
            present ( Couleur_intersection_avant ( ?i6 + ) ) 1216
            intersections_differentes ( ?i4 ?i6 ) 1216
            present ( Voisine ( ?i ?i2 ) ) 912
            present ( Couleur_intersection_avant ( ?i2 + ) ) 3648
            intersections_differentes ( ?i6 ?i2 ) 3648
            intersections_differentes ( ?i4 ?i2 ) 2736
            present ( Voisine ( ?i2 ?i1 ) ) 1824
            present ( Couleur_intersection_avant ( ?i1 + ) ) 7044
            present ( Nombre_voisines ( ?i1 4 ) ) 6912
            present ( Nombre_voisines_couleur_avant ( ?i1 ?c1 0 ) ) 5724
            present ( Voisine ( ?i4 ?i1 ) ) 5562
            intersections_differentes ( ?i1 ?i ) 2676
            present ( Nombre_Blocs_voisins_couleur_avant ( ?i1 ?c 0 ) ) 852
            present ( Voisine ( ?i ?i7 ) ) 832
            present ( Couleur_intersection_avant ( ?i7 + ) ) 3328
            intersections_differentes ( ?i4 ?i7 ) 3328
            intersections_differentes ( ?i6 ?i7 ) 2496
            intersections_differentes ( ?i2 ?i7 ) 1664
            present ( Voisine ( ?i1 ?i8 ) ) 832
            present ( Couleur_intersection_avant ( ?i8 + ) ) 3328
            intersections_differentes ( ?i4 ?i8 ) 3328
            intersections_differentes ( ?i2 ?i8 ) 2496
            present ( Couleur_intersection_avant ( ?i8 ?c2 ) ) 1664
            present ( Voisine ( ?i1 ?i3 ) ) 1664
            present ( Couleur_intersection_avant ( ?i3 + ) ) 6656
            intersections_differentes ( ?i4 ?i3 ) 6656
            intersections_differentes ( ?i8 ?i3 ) 4992
            intersections_differentes ( ?i2 ?i3 ) 3328
            present ( Voisine ( ?i4 ?i5 ) ) 1664
            present ( Couleur_intersection_avant ( ?i5 + ) ) 6656
            intersections_differentes ( ?i ?i5 ) 6576
            intersections_differentes ( ?i1 ?i5 ) 4912
           )
  conclusions   (
            ajoute ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i ?i1 ?i GIII ) ) 3248
            ajoute ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i1 ?i ?i1 GIII ) ) 3248
           )
)
```

Tableau Compilation Règle sans Absent

Le métaprédicat absent apparaît quatre fois dans la règle du tableau Compilation Espionnage parce qu'il permet à chaque fois de faire des coupes dans le graphe d'unification. La stratégie de filtrage est de filtrer en profondeur d'abord, c'est-à-dire que le système parcours tout les graphe d'unification qui

est en-dessous de l'instanciation d'une nouvelle variable. Si toutes les variables de la conclusion sont connues et qu'une instanciation de variable amène à déduire la conclusion de la règle en descendant dans le graphe d'unification, il est inutile de voir si d'autres instanciations de variables amèneront à la même conclusion. On peut donc couper le graphe et ne pas considérer les autres instanciations possibles pour la variable. Dans la règle du tableau Compilation Espionnage, une fois que toutes les variables en conclusion sont connues il reste quatre prémisses qui amènent à des instanciations multiples de variables (les prémisses concernant les Voisines). Pour chacune des instanciations de ces prémisses, il est inutile de vérifier que plusieurs instanciations donne la même conclusion (lorsqu'on utilise pas la règle pour l'apprentissage). L'insertion des prémisses vérifiant l'absence après les prémisses amenant à des instanciations multiples permet donc de ne pas continuer l'unification inutilement.

6.3 L'ordonnancement des listes de règles

La taxonomie des jeux est utilisée pour ordonner les listes de règles. Rappelons que la taxonomie des jeux donne des relations "est-un" entre les jeux. Ainsi, un jeu GG est un jeu GI. Si une règle a déjà déduit un jeu spécifique, il est inutile de filtrer les règles qui concluent sur un jeu plus haut dans la taxonomie que le jeu déjà déduit. Par exemple, il est inutile de matcher les règles qui concluent sur un jeu GI si une règle concluant sur un jeu GG a déjà été déduite. Ordonner l'ordre de filtrage des règles a donc une influence sur le coût final du filtrage. Même si les règles sont finalement déclaratives, il est bon de les ordonner pour aller plus vite. Dans ce cas là, l'ordre fixé des règles ne nuit pas à la déclarativité des règles parce que la taxonomie des jeux est immuable.

Il est aussi nécessaire d'introduire des prémisses du type 'absent (Coup_jeu_binaire_bloc_avant (?c Connecter ?b ?b1 ?i G))' dès que ?c, ?b, ?b1 et ?i sont définis dans une règle qui conclue sur 'ajoute (Coup_jeu_binaire_bloc_avant (?c Connecter ?b ?b1 ?i GI))'.

6.4 L'Optimisation

6.4.1 Optimisation des calculs

L'optimisation des calculs est une technique courante des compilateurs [Tremblay 1985]. Elle consiste à effectuer lors de la compilation les calculs comportant des valeurs connues à ce moment là. On remplace alors une fonction par son résultat. Par exemple on transforme la liste suivante de fonctions, cette liste est tirée d'une règle découverte par mon système :

```
egal ( ?n9 addition ( 1 1 )  )
egal ( 1 soustraction ( ?n9 1 )  )
egal ( ?n10 addition ( 1 1 )  )
egal ( ?n11 addition ( 3 ?n10 )  )
egal ( ?n12 soustraction ( ?n11 1 )  )
superieur ( ?n12 1 )
egal ( ?n13 addition ( ?n12 1 )  )
superieur ( ?n13 1 )
```

en une autre liste plus courte à vérifier :

```
egal ( ?n9 2  )
egal ( ?n10 2  )
egal ( ?n11 5  )
egal ( ?n12 4  )
egal ( ?n13 5  )
```

L'étape suivante est de remplacer toutes les apparitions des variables ?n9, ?n10, ?n11, ?n12 et ?n13 dans les prémisses et les conclusions de la règle par les constantes correspondantes. On peut alors supprimer des conditions de la règle toutes les conditions de la liste ci-dessus.

6.4.2 Evaluation Partielle

L'évaluation partielle d'une règle permet de spécialiser certaines de ses prémisses afin de pouvoir effectuer certains calculs une fois pour toute lors de la compilation plutôt que de les refaire à chaque fois que la règle est exécutée. La principale activité du mécanisme d'évaluation partielle dans mon programme de Go est de transformer le prédicat général Voisine (?i ?i1) en prédicats spécialisés : Au_dessus (?i ?i1), En_dessous (?i ?i1), A_gauche (?i ?i1) et A_droite (?i ?i1). Cette représentation spécialisée se rapproche des patterns géométriques. Elle permet de calculer beaucoup de prédicats à la compilation. Ainsi les prédicats concernant les tests intersections_differentes (?i ?i1) sont calculés directement et ôtés de la règle s'il sont vrais. S'ils sont faux, c'est alors que la règle n'est pas filtrable et celle-ci est ôtée. De plus, les coupes dans le graphe d'unification ne sont plus nécessaires après la définition d'une nouvelle intersection par les prédicats spécialisés, puisqu'une seule intersection correspond à la définition. Là aussi, le filtrage gagne en rapidité. Le tableau Compilation Evaluation Partielle donne un exemple de règle évaluée partiellement. C'est la transformée de la règle du tableau Compilation Espionnage.

```
( premisses    (
            present ( Couleur ( ?c ) )
            present ( Couleur_opposees ( ?c1 ?c ) )
            present ( Couleur ( ?c1 ) )
            present ( Nombre_voisines_couleur_avant ( ?i + 4 ) )
            present ( Nombre_voisines ( ?i 4 ) )
            present ( Nombre_Blocs_voisins_avant ( ?i 0 ) )
            present ( Couleur_intersection_avant ( ?i + ) )
            present ( Au_dessus ( ?i ?i4 ) )
            present ( Couleur_intersection_avant ( ?i4 + ) )
            present ( A_droite ( ?i ?i6 ) )
            present ( Couleur_intersection_avant ( ?i6 + ) )
            present ( A_gauche ( ?i ?i2 ) )
            present ( Couleur_intersection_avant ( ?i2 + ) )
            present ( Au_dessus ( ?i2 ?i1 ) )
            present ( Couleur_intersection_avant ( ?i1 + ) )
            present ( Nombre_voisines ( ?i1 4 ) )
            present ( Nombre_voisines_couleur_avant ( ?i1 ?c1 0 ) )
            present ( A_gauche ( ?i4 ?i1 ) )
            present ( Nombre_Blocs_voisins_couleur_avant ( ?i1 ?c 0 ) )
            present ( En_dessous ( ?i ?i7 ) )
            present ( Couleur_intersection_avant ( ?i7 + ) )
            present ( A_gauche ( ?i1 ?i8 ) )
            present ( Couleur_intersection_avant ( ?i8 + ) )
            present ( Couleur_intersection_avant ( ?i8 ?c2 ) )
            present ( Au_dessus ( ?i1 ?i3 ) )
            present ( Couleur_intersection_avant ( ?i3 + ) )
            present ( Au_dessus ( ?i4 ?i5 ) )
            present ( Couleur_intersection_avant ( ?i5 + ) )
          )
 conclusions   ( ajoute ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i ?i1 ?i GIII ) )
                 ajoute ( Coup_jeu_binaire_intersection_intersection_avant ( ?c Connecter ?i1 ?i ?i1 GIII ) ) )
)
```

Tableau Compilation Evaluation Partielle

On remarquera que les prédicats Voisines sont tous successivement remplacés par les prédicats En_dessous, A_gauche, A_droite, Au_dessus quelque soit la règle. Cette modification des règles est générale et s'applique à toutes les règles trouvées par le système. Le mécanisme d'évaluation partielle augmente beaucoup le nombre de règles et donc la place prise en mémoire par le programme, mais il augmente aussi beaucoup sa rapidité en remplaçant beaucoup de tests faits lors de l'utilisation des règles par des tests fait lors de la compilation des règles.

6.5 La compilation en C++

6.5.1 Equivalence instanciation/boucle for

L'instanciation d'une variable dans le prédicat d'une règle en logique des prédicats est équivalent à une boucle en langage impératif.

On gagne beaucoup de temps à transformer une instanciation par une boucle C. Les accès se font directement dans un tableau au lieu de demander à l'interpréteur de règles de faire un parcours de liste. De plus, on gagne aussi de la place en mémoire, car avoir une valeur dans un tableau prend beaucoup moins de place en mémoire que d'avoir un prédicat dans une base de faits.

Par exemple, la prémisse 'present (Liberte_bloc_avant (?i ?b))' dans lequel ?b est déjà connu est remplacée par :

```
for (_i=0; _i<Nb_libertes_bloc [b]; _i++) {
i=Ieme_liberte_du_bloc [b] [_i];
```

6.5.2 Equivalence instanciation/affectation

Lorsqu'il n'y a qu'une seule variable à instancier, la traduction ne se fait plus vers une boucle mais vers une affectation.

Par exemple, la prémisse 'present (Nombre_libertes_bloc_avant (?b1 ?n17))' dans lequel ?b1 est déjà connu est remplacé par n17=Nb_libertes_bloc [b1];

(premisses (	present (Nombre_pierres_bloc_voisines_avant (?b3 ?b2 ?n6))
present (Couleur (@))	present (Nombre_libertes_communes_avant (?b4 ?b3 ?n12))
present (Couleurs_differentes (@ +))	present (Nombre_pierres_bloc_voisines_communes_avant (?b2 ?b3 ?b4 ?n9))
present (Couleurs_differentes (+ @))	present (Voisine (?i ?i1))
present (Couleur_opposees (@ ?c1))	present (Couleur_intersection_avant (?i1 ?c1))
present (Couleur (?c1))	present (Pierre_voisine_bloc_avant (?i1 ?b3))
present (Couleurs_differentes (@ ?c1))	present (Nombre_voisines (?i1 4))
present (Couleurs_differentes (?c1 @))	present (Bloc_avant (?i1 ?b2))
present (Nombre_libertes_bloc_avant (?b2 1))	present (Voisine (?i1 ?i2))
present (Couleur_bloc_avant (?b2 ?c1))	present (Bloc_avant (?i2 ?b3))
present (Nombre_pierres_bloc_voisines_avant (?b ?b2 0))	intersections_differentes (?i2 ?i)
present (Couleur_bloc_avant (?b @))	present (Voisine (?i1 ?i3))
blocs_differents (?b2 ?b)	present (Bloc_avant (?i3 ?b4))
present (Nombre_libertes_bloc_avant (?b ?n20))	intersections_differentes (?i ?i3)
superieur (?n20 1)	intersections_differentes (?i2 ?i3)
present (Liberte_bloc_avant (?i ?b))	present (Voisine (?i1 ?i4))
present (Nombre_Blocs_voisins_couleur_avant (?i @ 3))	intersections_differentes (?i ?i4)
present (Nombre_voisines_couleur_avant (?i + 0))	intersections_differentes (?i3 ?i4)
present (Liberte_bloc_avant (?i ?b2))	intersections_differentes (?i2 ?i4)
present (Nombre_Blocs_voisins_avant (?i 4))	present (Bloc_avant (?i4 ?b1))
present (Liberte_bloc_avant (?i ?b4))	present (Non_liberte_bloc_avant (?i ?b1))
present (Couleur_bloc_avant (?b4 @))	present (Couleur_bloc_avant (?b1 @))
blocs_differents (?b ?b4)	blocs_differents (?b ?b1)
blocs_differents (?b2 ?b4)	present (Nombre_libertes_bloc_avant (?b1 ?n17))
present (Nombre_libertes_communes_avant (?b ?b4 ?n11))	superieur (?n17 1)
present (Nombre_pierres_bloc_voisines_avant (?b4 ?b2 ?n8))	present (Nombre_pierres_bloc_voisines_avant (?b1 ?b2 ?n15))
present (Nombre_libertes_bloc_avant (?b4 ?n18))	egal (?n1 addition (?n17 ?n15))
superieur (?n18 1)	superieur (?n1 1)
present (Liberte_bloc_avant (?i ?b3))	egal (?n soustraction (soustraction (soustraction (soustraction (soustraction (addition (addition (addition (addition (addition (?n20 ?n19) ?n18) ?n6) ?n8) ?n13) ?n9) ?n10) ?n11) ?n12) 1))
present (Couleur_bloc_avant (?b3 @))	
blocs_differents (?b ?b3)	
blocs_differents (?b2 ?b3)	
blocs_differents (?b4 ?b3)	
present (Nombre_libertes_communes_au_3_avant (?b ?b3 ?b4 ?n13))	

present (Nombre_libertes_communes_avant (?b ?b3 ?n10)) present (Nombre_libertes_bloc_avant (?b3 ?n19)) superieur (?n19 1)	superieur (?n 1)) conclusions (ajoute (Coup_jeu_binaire_bloc_avant (@ Connecter ?b ?b1 ?i GI)) ajoute (Coup_jeu_binaire_bloc_avant (@ Connecter ?b1 ?b ?i GI))))

Tableau Compilation Règle Connecter

if (Couleur [1]) { c1=Couleur_opposee (1); if (Couleur [c1]) { if (1 != c1) { for (_b2=0; _b2<Nb_blocs_ayant_nb_libertes [1]; _b2++) { b2=Ieme_bloc_ayant_nb_libertes [1] [_b2]; if (Couleur_bloc [b2] == c1) { for (b=0; b<nb_blocs; b++) if (Nombre_pierres_bloc_voisines [b] [b2] == 0) { if (Couleur_bloc [b] == 1) { if (b2 != b) { n20=Nb_libertes_bloc [b]; if (n20 > 1) { for (_i=0; _i<Nb_libertes_bloc [b]; _i++) { i=Ieme_liberte_du_bloc [b] [_i]; if (Nombre_blocs_voisins_couleur [i] [1] == 3) { if (Nombre_voisines_couleur [i] [2] == 0) { if (Liberte_bloc [i] [b2]) { if (Nombre_blocs_voisins [i] == 4) { for (_b4=0; _b4<Nombre_blocs_voisins [i]; _b4++) { b4=Ieme_bloc_voisin [i] [_b4]; if (Couleur_bloc [b4] == 1) { if (b != b4) { if (b2 != b4) { n11=Nb_libertes_communes [b] [b4]; n8=Nombre_pierres_bloc_voisines [b4] [b2]; n18=Nb_libertes_bloc [b4]; if (n18 > 1) { for (_b3=0; _b3<Nombre_blocs_voisins [i]; _b3++) { b3=Ieme_bloc_voisin [i] [_b3]; if (Couleur_bloc [b3] == 1) { if (b != b3) { if (b2 != b3) { if (b4 != b3) { n13=Nombre_libertes_communes_aux_3 (b,b3,b4); n10=Nb_libertes_communes [b] [b3]; n19=Nb_libertes_bloc [b3]; if (n19 > 1) { n6=Nombre_pierres_bloc_voisines [b3] [b2]; n12=Nb_libertes_communes [b4] [b3];	n9=Nombre_pierres_bloc_voisines_communes (b2,b3,b4); for (_i1=0; _i1<Nb_voisines [i]; _i1++) { i1=Ieme_intersection_voisine [i] [_i1]; if (Couleur_intersection [i1] == c1) { if (Pierre_voisine_bloc [i1] [b3]) { if (Nb_voisines [i1] == 4) { if (Bloc [i1] [b2]) { for (_i2=0; _i2<Nb_voisines [i1]; _i2++) { i2=Ieme_intersection_voisine [i1] [_i2]; if (Bloc [i2] [b3]) { if (i2 != i) { for (_i3=0; _i3<Nb_voisines [i1]; _i3++) { i3=Ieme_intersection_voisine [i1] [_i3]; if (Bloc [i3] [b4]) { if (i != i3) { if (i2 != i3) { for (_i4=0; _i4<Nb_voisines [i1]; _i4++) { i4=Ieme_intersection_voisine [i1] [_i4]; if (i != i4) { if (i3 != i4) { if (i2 != i4) { b1 = Numero_bloc [i4]; if (!Liberte_bloc [i] [b1]) { if (Couleur_bloc [b1] == 1) { if (b != b1) { n17=Nb_libertes_bloc [b1]; if (n17 > 1) { n15=Nombre_pierres_bloc_voisines [b1] [b2]; n1 = (n17 + n15); if (n1 > 1) { n = ((((((((((n20 + n19) + n18) + n6) + n8) + n13) - n9) - n10) - n11) - n12) - 1); if (n > 1) { je=(Jeu *)memoire_jeu->nouveau (); je->affecte_binaire_bloc (1,2,b,b1,Abscisse (i),Ordonnee (i),5); liste_jeux->append (je); je->detruit (); je=(Jeu *)memoire_jeu->nouveau (); je->affecte_binaire_bloc (1,2,b1,b,Abscisse (i),Ordonnee (i),5); liste_jeux->append (je); je->detruit (); }

Tableau Compilation Programme Connecter

```
(       premisses       (
                         present ( Couleur ( ?c ) )
                         present ( Couleur_opposees ( ?c1 ?c ) )
                         present ( Couleur ( ?c1 ) )
                         present ( Nombre_libertes_bloc_avant ( ?b 1 ) )
                         present ( Couleur_bloc_avant ( ?b ?c1 ) )
                         present ( Liberte_bloc_avant ( ?i ?b ) )
                         present ( Voisine ( ?i ?i1 ) )
                         absent ( Coup_jeu_bloc_avant ( ?c Prendre ?b ?i GI ) )
                         present ( Couleur_intersection_avant ( ?i1 + ) )
                        )
        conclusions     (
                         ajoute ( Coup_jeu_bloc_avant ( ?c Prendre ?b ?i GI ) )
                        )
)
```

Tableau Compilation Règle Prendre

```
#include "gestion_memoire.h"
#include "jeu.h"

extern Gestion_memoire *memoire_jeu;
extern Jeu *jeu;

#include "liste_variables.c"

void Goban::deductions_Prendre_GI_0 (Liste *liste_jeux)
{
Jeu *je;
for (c=0; c<2; c++) {
c1=Couleur_opposee (c);
if (Couleur [c1]) {
for (_b=0; _b<Nb_blocs_ayant_nb_libertes [1]; _b++) {
b=Ieme_bloc_ayant_nb_libertes [1] [_b];
if (Couleur_bloc [b] == c1) {
for (_i=0; _i<Nb_libertes_bloc [b]; _i++) {
i=Ieme_liberte_du_bloc [b] [_i];
for (_i1=0; _i1<Nb_voisines [i]; _i1++) {
i1=Ieme_intersection_voisine [i] [_i1];
if (!jeu->existe_dans_liste_jeu_bloc (liste_jeux,c,0,b,Abscisse (i),Ordonnee (i),5)) {
if (Couleur_intersection [i1] == 2) {
        je=(Jeu *)memoire_jeu->nouveau ();
        je->affecte_bloc (c,0,b,Abscisse (i),Ordonnee (i),5);
        liste_jeux->append (je);
        je->detruit ();
} } } } } } } }
}
```

Tableau Compilation Programme Prendre

6.5.3 Equivalence test/if

L'exécution d'un test du type 'if (Liberte [i] [b])' est immédiate alors que l'exécution d'un test du type 'présent (Liberte_bloc_avant (?i ?b))' demande à l'interpréteur de tester une présence dans la base de faits. Il doit retrouver les valeurs instanciées de ?i et ?b, puis vérifier que le fait est présent dans la base de faits.

6.5.4 Equivalence absent/parcours d'arbre

L'exécution d'un test du type :

'absent (Coup_jeu_binaire_intersection_intersection_avant (?c Connecter ?i ?i1 ?i GIII))'

correspond à une descente dans l'arbre représentant les différents jeux. Le prédicat est traduit par un appel à une méthode de la classe Jeu du programme de Go :

if (!jeu->existe_dans_arbre_jeu_binaire_ intersection_intersection (arbre_jeux, c, CONNECTER, i, i1, Abscisse (i), Ordonnee (i), GIII)) {

6.5.5 Traduction d'une règle

Pour fixer les idées, je donne un exemple de traduction d'une règle logique en fonction C++. Cette fonction C++ résultante est ensuite intégrée dans une classe de mon programme de Go qui y fera appel lorsqu'il cherchera à savoir si deux blocs sont connectables.

La règle source se trouve dans le tableau Compilation Règle Connecter, le programme C++ résultant est donné dans le tableau Compilation Programme Connecter.

6.5.6 Traduction d'une base de règles

Dans ses premières versions Gogol écrivait une seule fonction C++ par base de règle. Toutefois lorsque les bases se sont mises à grandir, il est devenu impossible au compilateur C++ de compiler les fonctions écrites par Gogol qui dépassaient 10 000 lignes. Gogol a donc dû apprendre à découper les fonctions en sous-fonctions. La version actuelle crée une fonction pour 10 règles, et crée une fonction qui appelle toutes les sous-fonctions.

La règle du tableau Compilation Règle Prendre est traduite par Gogol vers le programme C++ du tableau Compilation Programme Prendre.

6.6 Conclusion

La compilation est nécessaire à un programme d'apprentissage qui crée des connaissances générales et déclaratives. Mon système utilise quatre types de compilations :

- La compilation par réordonnancement des listes de prémisses est très efficace et permet de rendre plus rapide aussi bien la partie logique et interprétée du programme que sa partie C++ compilée.

- La compilation, par coupe dans les graphes d'unification des conditions des règles, accélère aussi le programme, mais il est préférable de ne pas l'utiliser dans le programme d'apprentissage car les coupes dans le graphe d'unification permettent de ne pas faire deux fois la même déduction. Or il est important pour le module d'apprentissage de connaître le plus de façons possibles de faire une déduction pour créer les règles les plus générales possibles.

- La compilation des bases de règles réordonne les règles de façon à ne pas redéduire des faits plus généraux que des faits déjà connus ; elle est utile pour la partie interprétée aussi bien que pour la partie compilée.

- Et enfin, la compilation des bases de règles exprimées en logique du premier ordre vers un programme C++ permet d'utiliser les règles apprises de façon efficace en les transformant en programmes C++ . Elle multiplie par soixante la vitesse d'exécution de ces règles, mais en contrepartie le système ne peut plus s'observer directement, mais il peut toujours interpréter les règles quand il a besoin de s'observer.

6.7 Bibliographie

[Alliot 1994] - J. M. Alliot et T. Schiex. *Intelligence Artificielle et Informatique Théorique*. Cepadues Editions 1994.

[Barklund 1994] - J. Barklund. *Metaprogramming in Logic*. UPMAIL Technical Report N° 80, 1994.

[Cazenave 1996b] T. Cazenave, *Automatic Ordering of Predicates by Metarules*. Proceedings of the 4th International Workshop on Metareasonning and Metaprogramming in Logic, Bonn, 1996.

[Doorenbos 1995] - R. B. Doorenbos, *Production Matching for Large Learning Systems*, Thèse de l'Université Carnegie Mellon, 1995.

[Forgy 1982] - C.L. Forgy, *RETE : A Fast Algorithm for the Many Pattern / Many Object Pattern Matching Problem*, Artificial Intelligence vol. 19, pp 17-37, 1982.

[Ishida 1988] - T. Ishida, *Optimizing Rules in Production System Programs*, AAAI 1988, pp 699-704, 1988.

[Laurière 1976] - J.L. Laurière, *Un langage et un programme pour énoncer et résoudre les problèmes combinatoires*, Thèse d'état Paris 6, 1976.

[Markovitch 1993] - S. Markovitch, P. D. Scott, *Information Filtering: Selection Mechanisms in Learning Systems*, Machine Learning 10, pp. 113-151, 1993.

[Minton 1988] - S. Minton, *Learning Search Control Knowledge - An Explanation Based Approach*, Kluwer Academics, Boston, 1988.

[Parchemal 1988] - Y. Parchemal. *SEPIAR : un système à base de connaissances qui apprend à utiliser efficacement une expertise.* Thèse de l'Université Paris 6, 1988.

[Pitrat 1976] - J. Pitrat. *Realization of a program learning to find combination in chess*. Computer orientated learning processes. NATO Advanced Study Institute.

[Pitrat 1990] - J. Pitrat. *Métaconnaissance futur de l'intelligence artificielle.* Hermès, 1990.

[Porcheron 1990] - M. Porcheron. *Utilisation de méta-connaissances pour la compilation des règles de production.* Thèse de l'Université Paris 6, 1990.

[Tremblay 1985] - J.P. Tremblay, P. G. Sorenson. *The theory and practice of compiler writing.* McGraw-Hill International Editions, 1985.

Table des Matières du Chapitre 7

7 GOGOL : UN PROGRAMME DE GO 113

7.1 Les programmes de Go 113
7.1.1 Les programmes qui jouent une partie entière 113
7.1.2 Les programmes spécialisés sur des sous-problèmes du Go 116

7.2 Architecture de Gogol 116

7.3 Les sous-buts intéressants du Go 117

7.4 Construction des groupes 120

7.5 Connaissances stratégiques 122
7.5.1 La connexion au vide, une approximation du territoire 122
7.5.2 Les attaques et les défenses sur les groupes 124

7.6 Résultats 126
7.6.1 L'échelle de niveau des programmes 126
7.6.2 Résumé des résultats des compétitions mondiales de programmes de Go 128
7.6.3 La coupe FOST 1996 129
7.6.4 Evaluation du niveau de Gogol 131

7.7 Conclusion 131

7.7 Bibliographie 132

7 Gogol : un programme de Go

Dans ce chapitre, je décris dans une première section les programmes de Go les plus importants. Je donne ensuite dans une deuxième section les sous-buts du jeu de Go qui sont utilisés dans mon programme Gogol et que celui-ci apprend à calculer. La troisième section est consacrée à un problème clé de la programmation du Go : la construction des groupes de pierres. La quatrième section explique comment les connaissances stratégiques sont utilisées dans Gogol. Une cinquième partie expose les résultats obtenus par mon programme contre les autres programmes de Go. Dans une dernière partie je conclus sur mon programme de Go et donne une idée du travail qui me reste à faire.

7.1 Les programmes de Go

7.1.1 Les programmes qui jouent une partie entière

> "I think we have weak programs because computers are still too slow, and that no one has figured out how to make a go program that learns from experience. There is just too much go knowledge required to play well, and no one has time or patience enough to enter it all by hand.
>
> I've always found that improvements in program strength did not come from any sudden change due to a new algorithm. Instead there are very small, incremental changes that accumulate with each new piece of go knowledge added. Unfortunately, there is no simple, magic computer go algorithm that will make a strong program. The most important thing is attention to small details, completeness of knowledge in each area,

seamless integration of knowledge, having good rough approximations when the exact knowledge is missing.

The go program is only as strong as its weakest portion, so for example, if have a Dan level life/death reader for surrounded areas, but you only have a 20 Kyu unerstanding of running and connecting, you will have a 20 Kyu program.

Programs are weak because there are so many areas that have to be done well."

-David Fotland
Message envoyé à la mailing list computer Go le 11 Juillet 1996.

Historiquement, on a vu les programmes basés sur des algorithmes de reconnaissance floue [Zobrist 1970] [Wilcox 1974] surpassés par les programmes utilisant un pattern matching intensif [Boon 1985] [Fotland 1993] [Wilcox 1995]. Quelques programmes commencent à utiliser la théorie des jeux [Conway 1976] [Müller 1995][Bouzy 1995] qui a donné de très bons résultats sur la fin de partie [Wolfe 1992]. La plupart des programmes sont écrits en langage C ou C++, sauf Handtalk qui est écrit en Assembleur.

Handtalk

Ce programme a été champion du monde en 1993. Il a gagné la coupe FOST en 1995 et 1996. Son auteur, Mr Chen Zhixing, est chinois originaire de Guangzhou où il est professeur de chimie à la retraite. C'est le meilleur joueur amateur de sa région, ce qui en Chine révèle un très bon niveau. De plus, il se fait aider par une jeune professionnelle chinoise pour le développement de son programme. Son programme a un niveau de 12ème Kyu. Il a été écrit entièrement en assembleur. Son auteur ne désire pas donner de description de son programme. Il réfléchit à la programmation du Go depuis plus de 20 ans. Suite à l'argent gagné avec son programme, Mr Chen envisagerait de créer un institut de la programmation du Go en Chine.

Go Intellect

C'est le programme champion du monde en 1994. Ce programme a été développé par Ken Chen, américain, 6ème Dan amateur, à partir de la plateforme de développement de jeu de damiers "Smart Game Board" [Kierulf 1990] plateforme qui a aussi servi au développement de Swiss Explorer [Müller 1990]. C'est un programme qui sait bien attaquer et défendre les groupes [Chen 1992]. Il a un niveau de 12ème Kyu. Go Intellect existe depuis 1984.

Go4++

Son auteur Michaël Reiss est anglais et l'un des rares programmeurs de Go qui n'ait pas un bon niveau au jeu de Go. Go4++ a terminé deuxième aux coupes FOST 1995 et 1996. C'est le seul programme qui a battu Handtalk en 1996 ; il faut dire que son auteur l'avait spécialement optimisé afin de battre Handtalk. Michaël Reiss a fait jouer son programme 700 fois contre Handtalk entre la coupe FOST 1995 et la coupe FOST 1996. Go4++ a un niveau de 12ème Kyu. Il existe depuis 1983 et Michaël Reiss estime avoir investi 6 hommes/années dans celui-ci. Il n'existe pas de papier décrivant ce dernier.

Goliath

Champion du monde en 1989, 1990, 1991 et développé par Mark Boon [Boon 1990], hollandais, 5ème Dan amateur. Goliath avait un niveau de 12ème Kyu en 1991. Il n'a pas participé aux championnats du monde ni à l'échelle des niveaux des programmes depuis 1991. Mark Boon encadre une équipe de deux personnes à temps plein et d'une personne à mi-temps depuis 3 ans grâce à la

générosité de la société Nintendo. Son but est d'avoir un programme 1er Dan. Goliath existe depuis 1987.

Many Faces of Go

Second au championnat du monde 1994, ce programme a été développé par David Fotland [Fotland 1993], américain, 2ème Dan amateur. Il a un niveau de 12ème Kyu et existe depuis 1981. Son auteur travaille dessus depuis 15 ans à raison de 500 heures par an.

Star of Poland

Quatrième au championnat du monde 1994, ce programme est développé par Janus Krascek, polonais, 5ème Dan amateur, professeur en Informatique. Star of Poland a un niveau de 14ème Kyu et existe depuis 1987.

GoAhead

Développé par Peter Woitke, allemand, GoAhead a un niveau de 14ème Kyu. Son auteur y travaille depuis 7 ans à temps partiel. Ce programme est composé de 15 000 lignes de GFA-Basic.

Ego

Développé par Bruce Wilcox, américain, 5ème Dan amateur, c'est un programme fort qui a vu le jour en 6 mois. Son auteur avait développé auparavant Interim [Wilcox 1978, Wilcox 1979], Nemesis, Riscigo [Wilcox 1995]. Ce programme est basé sur un pattern matching intensif et sur un très grand nombre de buts très spécialisés. Son niveau est de 14ème Kyu. Ego existe depuis 1994, mais son auteur programme le Go depuis 1974.

Swiss Explorer

Développé par Martin Müller [Müller 1995], autrichien, 5ème Dan amateur, c'est un programme qui peut décomposer le damier en somme de sous-jeux mathématiques. Il a un niveau de 16ème Kyu et existe depuis 1984 comme Go Intellect.

Gogol

C'est le programme que je développe, il est écrit en C++ et utilise la théorie des jeux et diverses techniques pour apprendre à résoudre des problèmes tactiques. Il a un niveau de 16ème Kyu et existe depuis 1993.

Indigo

Développé par Bruno Bouzy [Bouzy 1995], français, 3ème Dan amateur, c'est un programme écrit en C++, basé sur la théorie des jeux et la morphologie mathématique. Il a été développé dans le cadre d'une thèse sur la modélisation cognitive du joueur de Go. Il a un niveau de 18ème Kyu et existe depuis 1991.

7.1.2 Les programmes spécialisés sur des sous-problèmes du Go

Problèmes de vie et mort

Sur des problèmes très localisés et complètement encerclés de base de vie, Gotools résout [Wolf 1991] et engendre [Wolf 1994] des problèmes de niveau 5ème Dan amateur. Son programme a même trouvé une erreur de classification d'un problème dans un dictionnaire de problèmes de vie et de mort pour professionnels [Wolf 1995]. Cependant, si pouvoir lire les problèmes encerclés de vie et de mort est nécessaire pour bien jouer au Go, ce n'est qu'une toute petite partie des capacités nécessaires aux bons joueurs de Go. A l'aide de son programme, Thomas Wolf a créé une base de 40 000 problèmes de vie et de mort.

La fin de partie

Sur les fins de partie, le programme de David Wolfe [Wolfe 1991] joue mieux que les meilleurs joueurs professionnels. Mais la théorie qu'il a développée ne s'applique qu'aux 5 ou 10 derniers coups de la partie, dans des cas très particuliers où les jeux sont indépendants et où tous les groupes sont vivants.

Le début de partie

Le système de Patrick Ricaud [Ricaud 1995] est spécialisé sur le début de partie. Il crée une abstraction du damier au début de partie qui lui permet de représenter de façon générale les connaissances sur le début de partie. Cette approche est plus intéressante que les approches utilisées dans la plupart de programmes de Go qui consistent à créer de grandes bases de données et qui est assimilable à de l'apprentissage par coeur sans compréhension des principes du Go. Alors que la compréhension du sens des coups est indispensable pour bien jouer au Go.

7.2 Architecture de Gogol

La figure Go Architecture Gogol donne une vue générale du fonctionnement de Gogol. Il reçoit un Goban en entrée et calcule un coup joué en sortie.

Il commence par calculer les tableaux C++ associés aux prédicats du jeu de Go. Une fois ces tableaux calculés, il peut déclencher les règles C++ programmées par Introspect pour déduire directement des jeux combinatoires sur le Goban. Les jeux correspondants à des menaces (jeux GIII) et les jeux correspondants à des coups forcés (jeux IP) donnent lieu à des recherches arborescentes. Ces recherches ont pour but de voir si les jeux GIII sont aussi des jeux GI ou G, et si les jeux IP sont aussi des jeux P. Pour développer l’arbre ET/OU de recherche, Gogol utilise les coups associés aux jeux GIII (menaces) pour développer les branches des noeuds OU, et les coups associés aux jeux IP (coups forcés) pour développer les branches des noeuds ET. Pour chaque noeud développé, on a un Goban correspondant pour lequel Gogol recalcule les tableaux C++ homologues des prédicats du jeu de Go et déclenche les règles C++ sur les jeux combinatoires.

Les résultats des recherches arborescentes et du déclenchement des règles C++ donne l’état des jeux connus sur la position. En utilisant les jeux G sur le but Connecter deux blocs, Gogol construit les groupes de pierres. En utilisant les autres jeux calculés, il calcule les propriétés de ces groupes de pierres.

Les coups qui modifient les propriétés des groupes de pierres sont déduits des états des jeux combinatoires. Il sont utilisés par les règles stratégiques pour noter les coups. La notation des coups dépend de la valeur territoriale du coup et des attaques et des défenses qu’il permet sur les groupes. Les règles stratégiques qui permettent de noter les coups sont données dans l’annexe G. Chaque coup a des effets sur chaque groupe, on note donc la valeur du coup pour chaque groupe.

Pour sélectionner le coup joué, on additionne dans une matrice de la taille du Goban les valeurs des coups pour chaque groupe et la valeur territoriale du coup. On sélectionne ensuite le coup qui a la plus grande valeur positive. Si toutes les valeurs sont négatives ou nulles, le programme passe.

Des traces de Gogol sur différents Gobans sont données dans l'annexe H.

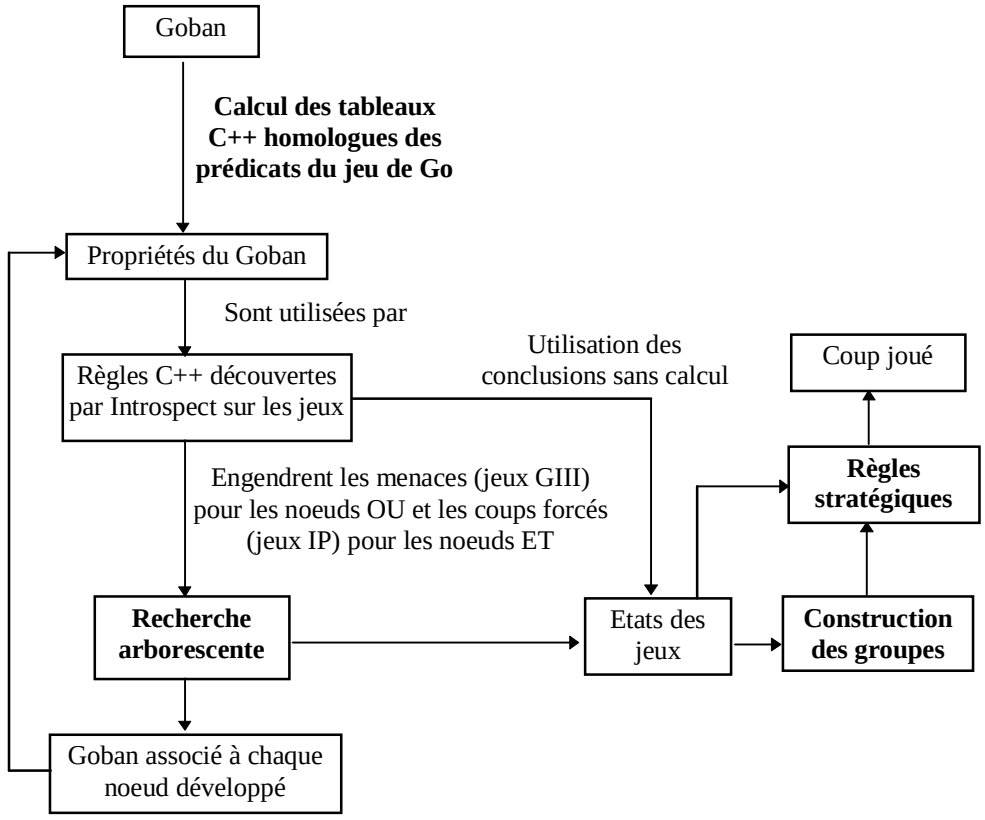


Figure Go Architecture Gogol

7.3 Les sous-buts intéressants du Go

Dans cette section, je donne de façon informelle les buts que mon système apprend à atteindre. La finalité de cette section est de montrer sur des exemples concrets ce que signifient ces buts. Pour une définition formelle, on peut se reporter au chapitre sur l'apprentissage. Les règles C++ écrites par Introspect et utilisées par Gogol portent sur les jeux associés à ces buts, une partie de ces règles est donnée dans l'annexe C.

Le but **Connecter** deux chaînes de pierres :

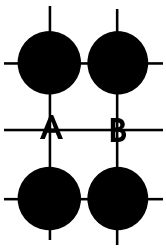


Figure Go Connecter Chaîne Chaîne

Deux pierres sont connectées si elles font partie de la même chaîne. Dans l'exemple ci-dessus le jeu de la connexion est G car si Blanc joue en A, Noir joue en B et vice et versa. Cette figure est connue par les joueurs de Go comme le noeud de bambou. Un proverbe dit que le noeud de bambou est incassable.

Le jeu de la connexion de deux chaînes de pierres est utilisé pour construire les groupes. Il est aussi utilisé pour connecter les groupes amis quand cela les renforce et pour déconnecter les groupes ennemis si cela les affaiblit. C'est le jeu le plus important pour Gogol avec le jeu décrit par la suite de la connexion de deux intersections. Le jeu de la connexion de deux blocs sert à attaquer et à défendre les groupes. Le jeu de la connexion de deux intersections sert à faire et à enlever du territoire.

Le but **Connecter** une chaîne et une intersection :

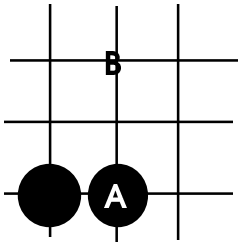


Figure Go Connecter Chaîne Intersection

Une chaîne est connectée à une intersection vide si le fait de jouer une autre pierre de même couleur sur cette intersection vide connecte la pierre à la chaîne. La connexion au vide est utilisée pour estimer les territoires propres à chaque groupe, pour étendre les groupes amis et pour attaquer les groupes ennemis.

Le but **Connecter** deux intersections :

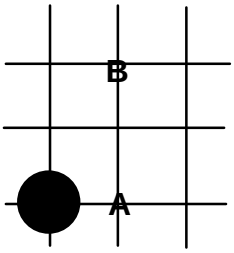


Figure Go Connecter Intersections

Dans l'exemple de la figure Go Connecter Intersections, si noir joue en A, il menace de connecter sa nouvelle pierre sur l'intersection A avec l'intersection B. Le but Connecter deux intersections est exclusivement utilisé pour les jeux qui menacent de connecter deux intersections. Les menaces de connecter deux intersections interviennent dans toute la partie, mais elles sont particulièrement importantes au début de la partie. Pour chaque intersection, Gogol calcule le nombre d'intersections qui lui seront connectables si on pose un pierre amie dessus. Il calcule aussi ce nombre pour une pierre amie posée. Ces nombres sont utilisés pour calculer la valeur territoriale des coups. L'annexe H donne un exemple de Goban vide sur lequel ce jeu est utilisé pour choisir le premier coup.

Le but **Prendre** une chaîne de pierres : une chaîne de pierres est retiré du jeu si elle n'a plus de libertés, les libertés d'une chaîne de pierres étant les intersections vides voisines de cette chaîne.

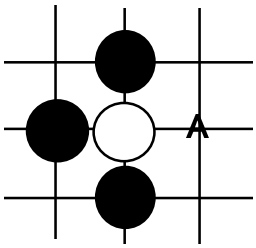


Figure Go Prendre

Dans l'exemple ci-dessus, si Noir joue en A, il fait prisonnier la pierre blanche qui est alors retirée du damier. Un coup qui prend une chaîne ennemie ou qui empêche l'ennemi de prendre une chaîne amie est augmenté de deux fois le nombre de pierres de la chaîne.

Le but **Faire_un_oeil**, c'est-à-dire atteindre une des positions ci-dessous sur une partie du damier ou Goban :

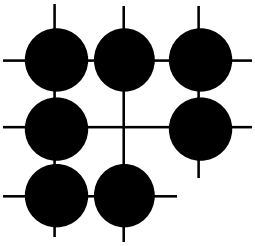


Figure Go Oeil

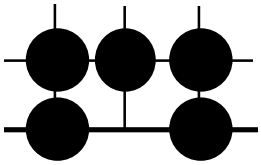


Figure Go Oeil Bord

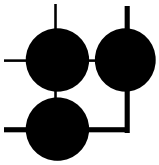


Figure Go Oeil Coin

La possibilité d'atteindre ou non ce but donne naissance au jeu de l'oeil. Les jeux concernant les yeux sont importants pour évaluer la santé d'un groupe, pour attaquer les groupes adverses en leur enlevant leurs yeux et pour protéger ses groupes en leur ajoutant des yeux.

Le but faire **Vivre** une chaîne de pierres est réalisé quand une chaîne a deux yeux.

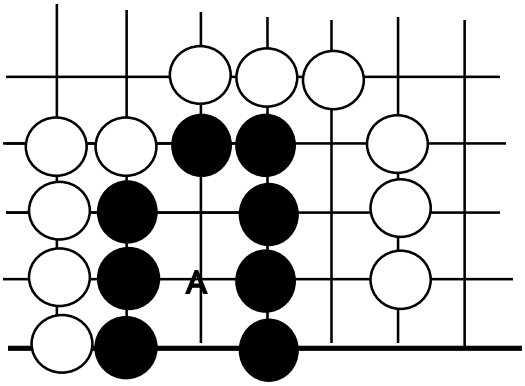


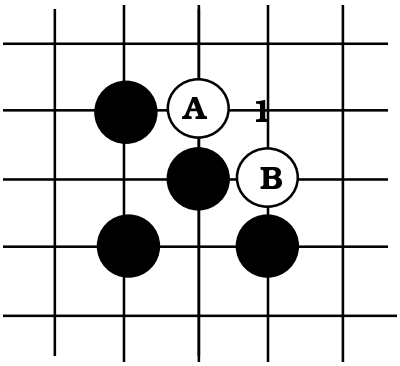
Figure Go Vivre

Dans l'exemple ci-dessus, si Noir joue en A, il fait vivre les deux chaînes noires, le jeu associé au but Vivre est GI. Gogol n'a appris que très peu de règles associées au but Vivre.

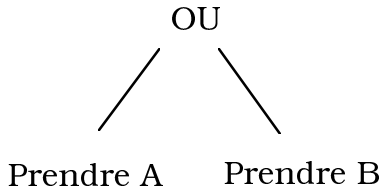
Le but Vivre est le but inverse du but Prendre. Il est aussi utilisé pour éviter de jouer des coups qui font mourir ses propres blocs, si une intersection à un jeu Vivre PI, jouer sur cette intersection fera perdre la pierre amie posée dessus et tous les blocs amis qui ont cette intersection pour liberté.

Les arbre de buts

Pour pouvoir gérer la dépendance de certains buts entre eux, il est nécessaire de créer des arbres de buts.



Ainsi dans cette position, le coup en 1 peut prendre soit A soit B. L'arbre de buts correspondant est le suivant :

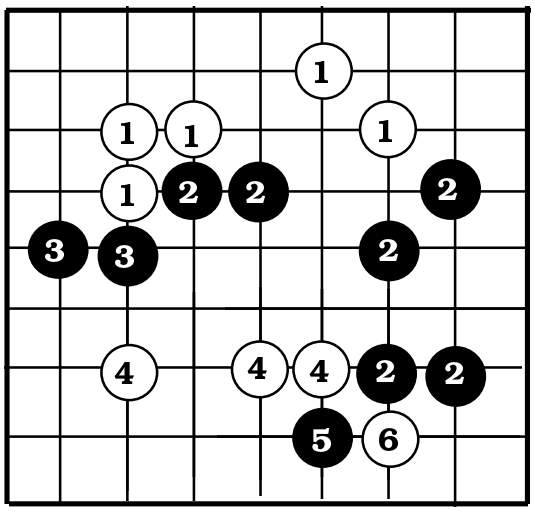


Gogol ne sait pas encore utiliser les arbres de buts. Savoir utiliser ces arbres est une condition nécessaire pour qu'il puisse atteindre un bon niveau aux normes humaines.

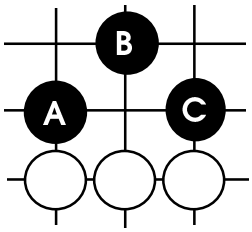
7.4 Construction des groupes

Un groupe de pierres est défini comme des pierres de la même couleur qui peuvent être connectées les unes avec les autres. Chaque groupe a un numéro et chaque pierre appartient à un seul groupe dans le modèle actuel.

Exemple de construction des groupes :

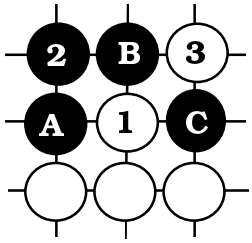


Dans la plupart des cas, il y a transitivité de la connexion, mais dans certains cas, comme celui de la figure ci-dessous, la connexion n'est pas transitive.



Non transitivité

A est connecté à B, B est connecté à C, mais A n'est pas connecté à C.



La figure ci-dessus montre la séquence qui permet à Blanc de déconnecter A et C.

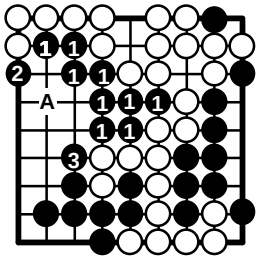


Figure Go 1

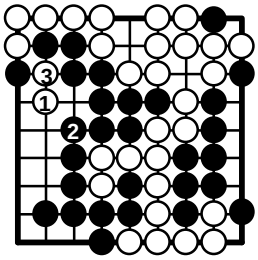


Figure Go 2

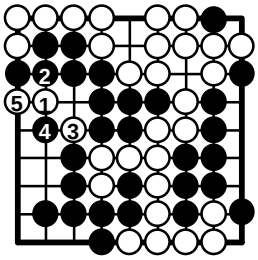


Figure Go 3

Dans l'exemple ci-dessus, l'approximation devient problématique. En effet, le bloc 2 de la figure Go 1 peut être pris si Blanc joue en A. Cependant Blanc ne peut pas empêcher Noir de Connecter les blocs 1 et 2, ni de Connecter les blocs 2 et 3. Or le groupe formé des blocs 1, 2 et 3 est vivant. Noir voit qu'il ne peut pas connecter les blocs 2 et 3 mais son heuristique de transitivité lui fait croire qu'il peut le faire. Les figures Go 2 et Go 3 donnent les séquences qui permettent à blanc de déconnecter les blocs noirs.

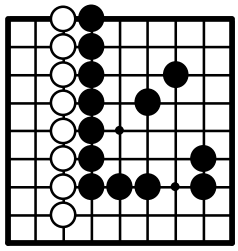
Ces exemples où l'heuristique de transitivité est mise en défaut sont toutefois assez rares (approximativement 1% des cas). Cependant les erreurs liées à cette heuristique peuvent faire perdre des parties (annexe F).

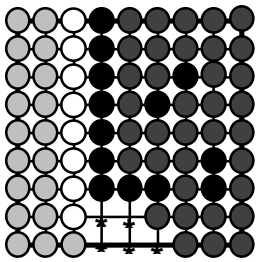
7.5 Connaissances stratégiques

La stratégie consiste à noter les coups en fonction des buts qu'ils atteignent. On peut distinguer deux préoccupations principales dans le module stratégique de Gogol. La première concerne les territoires, à savoir leur construction, leur destruction et leur préservation, ceux-ci étant approximés par la connexion au vide. La deuxième préoccupation du module stratégique concerne les groupes et plus particulièrement leur attaque et leur défense. Pour cela, le module stratégique de Gogol détermine la santé des groupes à l'aide d'heuristiques et cherche à prévoir les conséquences de ses coups sur celle-ci.

7.5.1 La connexion au vide, une approximation du territoire

Exemple de damier sur lequel on note la connexion au vide :





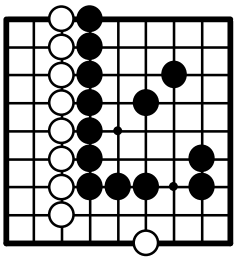
Les connexions au vide des groupe noirs vivants sont notées par des pierres gris foncé. Les connexions au vide des groupe blancs vivants sont notées par des pierres gris clair. Si une intersection vide est à la fois connectée à un groupe blanc vivant et à un groupe noir vivant, elle est notée '*'.

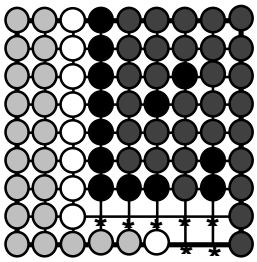
Territoire calculé sur le damier précédent (le territoire est estimé avec une méthode proche des règles chinoises, on compte comme points aussi bien le territoire que les pierres vivantes) :

Territoire Noir : 49
Territoire Blanc : 27

L'estimation du score est alors de 49-27=22 points d'avance pour noir.

Si Blanc joue la glissade du singe :



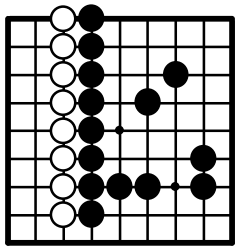


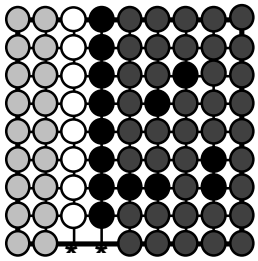
On a alors l'estimation territoriale suivante :

Territoire Noir : 44
Territoire Blanc : 30

Noir ne gagne plus que de 14 points.

Si Noir Bloque :

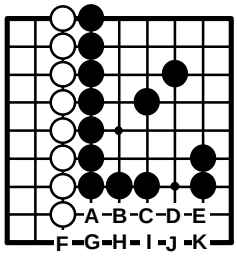




Territoire Noir : 53
Territoire Blanc : 26

Noir gagne alors de 27 points.

Différences de territoire attribuées aux différents coups :



On envisage tous les coups qui sont connectés à un groupe non mort. Pour chacun de ces coups, on fait la liste des intersections qu'ils déconnectent d'un groupe adverse non mort. On fait aussi la liste des intersections faisant auparavant partie du territoire ennemi qu'il menace de connecter au groupe.

Coup ○	A	B	G	H	I
● est déconnecté de	A,G	B,G,H	G,H	G,H,I	G,H,I,J
○ est connectable à	C,J	D,J	C,D,J	C,D,E,J,K	C,D,E,J,K

On fait de même pour l'adversaire.

Coup ●	A	B	G	H	I
○ est déconnecté de	A,B,H,I	B,H,I	B,G,H,I	B,H,I	B,H
● se connecte à	F	-	F	F	-

On peut alors calculer les menaces que chaque coup blanc empêche de réaliser, par exemple un coup blanc en A empêche noir de menacer de se connecter à F.

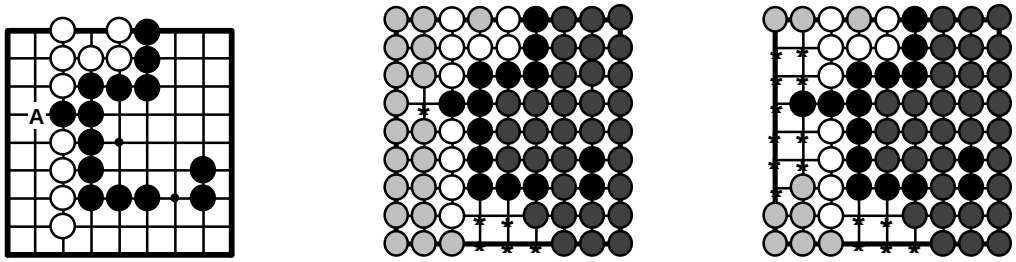
Coup ○	A	B	G	H	I
empêche la menace de connexion de ● à	C,D,E,J,K	C,D,E,J,K	C,D,E,J,K	C,D,E,J,K	C,D,E,J,K

On effectue alors, cette fois-ci pour noir, la somme du nombre d'intersections nouvelles qu'il menace de connecter, du nombre d'intersections ennemies qu'il déconnecte et du nombre d'intersections amies dont il empêche l'ennemi de menacer de s'y connecter.

Coup ●	A	B	G	H	I
Valeur territoriale	10	8	10	9	7

L'algorithme trouve que les deux meilleurs coups sont A et G. Dans la réalité, A vaut 1 point de plus que G. Ces deux coups sont toutefois les deux meilleurs coups.

Ces deux exemples n'ont cependant pas permis de montrer l'utilité de calculer le nombre d'intersections amies dont un coup empêche l'ennemi de menacer de s'y connecter. Ce calcul est pourtant très utile. Pour montrer son utilité, prenons un autre exemple.



Le coup Blanc en A ne rapporte pas de nouvelles intersections connectables, mais il déconnecte Noir de A. Si Noir joue en A, 10 nouvelles intersections lui sont connectables. Le coup blanc en A empêche donc noir de menacer de se connecter à 10 nouvelles intersections. Si on ne calculait pas le nombre de menaces de connecter qu'un coup empêche, le coup blanc en A serait évalué à 2 points, alors qu'il en vaut beaucoup plus.

7.5.2 Les attaques et les défenses sur les groupes

Les connaissances stratégiques dans les jeux portent sur des buts à long terme. La combinatoire trop grande des jeux comme le jeu de Go ou le jeu d'Echecs fait qu'il est impossible de prévoir à long terme les conséquences des coups joués. Une solution à ce problème est de gradualiser l'achèvement des buts à long terme afin de pouvoir prévoir si un coup se rapproche d'un but ou s'en éloigne.

Ceci est particulièrement vrai dans le cas de la stratégie au jeu de Go. Le but ultime du joueur de Go est de faire vivre le plus de pierres possibles sur le damier. Toutefois, dans la phase du milieu de partie, la plupart des groupes de pierres ont un statut incertain et leur évolution est imprévisible de façon certaine. Il est alors très utile d'avoir une évaluation floue de leur statut et de l'évolution que peuvent amener les différents coups agissant sur leurs attributs. Une modélisation floue de l'évolution d'une partie de Go a déjà été proposée dans [Bouzy 1996] et de façon différente dans [Cazenave 1996c].

Définition : Un groupe de pierres est un ensemble de pierres de la même couleur transitivement connectées.

Exemple :

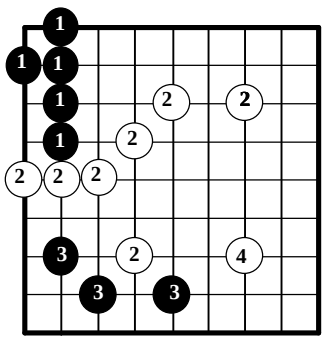


Figure Go 4

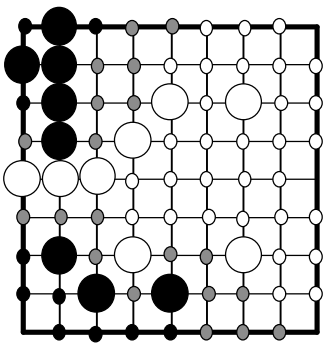


Figure Go 5

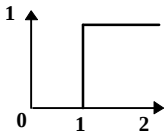
Les pierres d'un même groupe sont marquées avec le même numéro dans la figure Go 4.

On peut calculer plusieurs attributs pour un groupe. Certains des attributs que je vais décrire sont tirés de [Bouzy 1995], alors que d'autres sont propres à mon système. Le tableau Go 1 donne la liste des attributs utilisés.

Nombre de bases de vie Gagnées
Nombre de bases de vie Instables
Nombre d'yeux Gagnés
Nombre d'yeux Instables
Nombre d'intersections amies connectables
Nombre d'intersections connectables
Nombre de pierres
Nombre de connexions à des amis voisins vivants

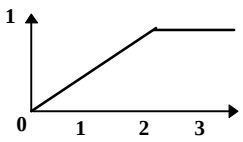
Tableau Go 1

Chacun de ces attributs contribue au but final qui est de faire vivre un groupe. Ces contributions sont plus ou moins graduelles et sont représentées dans les figures Go 6 à 11. L'ordonnée représente chaque fois, le degré de vie du groupe compris entre 0 et 1.



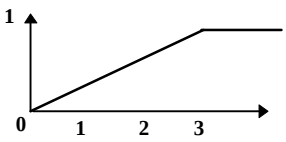
Nombre de bases de vie Gagnées

Figure Go 6



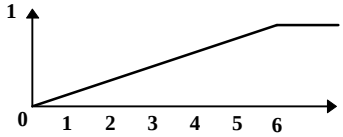
Nombre de bases de vie Instables

Figure Go 7



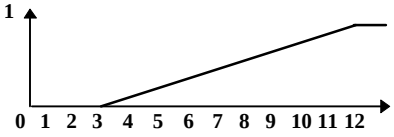
Nombre d'yeux Gagnés

Figure Go 8



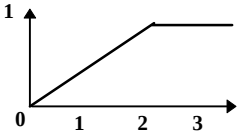
Nombre d'yeux Instables

Figure Go 9



Nombre d'intersections amies connectables

Figure go 10



Connexions à des voisins amis vivants

Figure Go 11

Dans la figure Go 5, les intersections connectables à un groupe blanc sont marquées d'un point blanc. Les intersections connectables à un groupe noir sont marquées d'un point noir. Les intersections connectables à la fois à un groupe noir et à un groupe blanc sont marquées d'un point gris. Le tableau Go 2 donne l'évaluation des différents attributs pour les 4 groupes de la figure Go 4.

Attributs\Groupes	1	2	3	4
Nombre de bases de vie Gagnées	0	0	0	0
Nombre de bases de vie Instables	1	0	0	0
Nombre d'yeux Gagnés	1	0	0	0
Nombre d'yeux Instables	1	0	0	0
Nombre d'intersections amies connectables	3	26	7	11
Nombre d'intersections connectables	11	40	19	18
Nombre de pierres	5	7	3	1
Nombre de connexions à des amis voisins vivants	0	0	0	2

Tableau Go 2

Le choix qui a été fait pour agréger ces attributs est celui de ne jamais surestimer le degré de vie d'un groupe. Le degré de vie d'un groupe est donc le maximum de tous les degrés de vie correspondant à chaque attribut. Le tableau Go 3 donne les degrés de vie correspondant à chaque attribut pour chaque groupe et le degré de vie finalement déduit pour le groupe.

Attributs\Groupes	1	2	3	4
Nombre de bases de vie Gagnées	0	0	0	0
Nombre de bases de vie Instables	0.5	0	0	0
Nombre d'yeux Gagnés	0.33	0	0	0
Nombre d'yeux Instables	0.16	0	0	0
Nombre d'intersections amies connectables	0	1	0.44	0.89
Nombre de connexions à des amis voisins vivants	0	0	0	1
Degré de vie du groupe	0.5	1	0.44	1
Importance du groupe	24	80	32	31

Tableau Go 3

L'importance d'un groupe est évaluée de façon approximative par la formule suivante :

$Importance_i$ = 2*Nombre de pierres$_i$ + Nombre d'intersections connectables$_i$ + Nombre d'intersections amies connectables$_i$

Les importances des différents groupes de l'exemple sont données dans le tableau Go 3. L'importance d'un groupe reflète la différence de points due au fait que le groupe sera vivant ou mort à la fin de la partie.

Une fois les importances et les degrés de vie des groupes calculés, on peut écrire une fonction d'évaluation d'une position de Go :

$$Evaluation = \sum_i (Degré_i * Importance_i) - \sum_j (Degré_j * Importance_j)$$

avec $i \in$ Groupes amis et $j \in$ Groupes ennemis.

Dans l'exemple de la figure Go 4, en supposant que Noir est la couleur amie, l'évaluation de la position est la suivante :

Evaluation = 0.5*23 + 0.44*32 - 1.0*80 - 1.0*31 = 11.5 + 14.1 - 80 - 31 = -85.4

Ce qui veut dire que Noir va probablement perdre la partie d'approximativement 43 points. Cette méthode de comptage a été testée sur de nombreuses positions de Go et elle donne une bonne approximation de l'évaluation d'une position.

Les règles stratégiques qui mettent en application cette formalisation des attaques et défenses sur les groupes sont données dans les Annexes sous deux formes. La représentation basée sur la logique des prédicats que j'ai utilisée pour les écrire, et la représentation sous forme de programme C++ que Introspect a écrit à partir des règles logiques que je lui ai donné.

7.6 Résultats

7.6.1 L'échelle de niveau des programmes

L'échelle de niveau Computer Go est une compétition informelle entre programmes de Go [Pettersen 1994]. Son but est de stimuler la recherche sur la programmation du Go et de permettre de mesurer l'état de l'art des différents programmes.

Les programmes exposés ci-après participent actuellement à l'échelle de niveaux :

Programme	Auteur	Adresse e-mail
Ego	Bruce Wilcox	wilcox@slip.net
Explorer	Martin Mueller	mueller@icsi.berkeley.edu
Go4++	Michaël Reiss	udah219@bay.cc.kcl.ac.uk
Gobble	Bernd Bruegmann	bruegman@mppmu.mpg.de
GoAhead	Peter Woitke	woitke@physik.tu-berlin.de
Gogol	Tristan Cazenave	Tristan.Cazenave@laforia.ibp.fr
Golife I	Henrik Rydberg	rydberg@fy.chalmers.se
gottaGo	Eric Pettersen	pett@cgl.ucsf.edu
Indigo	Bruno Bouzy	Bruno.Bouzy@laforia.ibp.fr
Many Faces of Go	David Fotland	fotland@hpihoc.cup.hp.com
Neurogo II	Markus Enzenberger	Markus.Enzenberger@Physik.uni-muenchen.de
Perception	Bill Shubert	wms@hevanet.com
Poka	Howard Landmann	landman@hal.com

Le 2 Octobre 1996, l'échelle de niveaux était la suivante :

Le handicap donné au suivant correspond au nombre de coups d'avance donnés au début de la partie par le programme le plus haut placé à celui qui le suit. Pour le dernier programme, ce chiffre n'a pas de sens.

Échelle 9X9 :

Programme	Handicap donné au suivant
Go4++	0
Many Faces of Go	0
Ego	1
Neurogo II	0
Gobble	1
Gogol	0
gottaGo	0
Explorer	1
Indigo	0
Perception	0
Poka	0
Golife I	N/A

Échelle 19x19 :

Programme	Handicap donné au suivant
Many Faces of Go	2
GoAhead	0
Ego	0
Explorer	0
Gogol	0
Indigo	0
Poka	0
gottaGo	N/A

7.6.2 Résumé des résultats des compétitions mondiales de programmes de Go

Le championnat Ing a lieu tous les ans au mois de Novembre. Il est doté d'un prix de 40 000 Francs français pour le programme qui atteint la première place. Le meilleur programme joue un match contre un joueur humain avec un handicap pour le joueur humain. S'il gagne le match le handicap décroît. Les prix pour les matchs contre les humains varient de 30 000 Francs français à 8 000 000 Francs français.

Les Olympiades des ordinateurs sont des compétitions mondiales entre programmes de jeux. Le jeu de Go est une des disciplines olympiques.

La coupe FOST a lieu tous les ans au mois de Septembre au Japon. Le premier prix est de 100 000 Francs français.

David Fotland tient à jour un classement officieux des programmes dans lequel il inclut tous les programmes qui ont participé plus d'une fois ou qui ont fini dans les 5 premières places. Il ordonne les programmes par leur force relative à long terme, les résultats les plus récents ayant un poids plus fort.

87 - Ing, Taipei, Taiwan
88 - Ing, Taipei, Taiwan
89 - Ing, Taipei, Taiwan
89o - 1ère Olympiade des ordinateurs, Londres, Angleterre
90 - Ing, Pékin, Chine
91 - Ing, Singapour
92 - Ing, Tokyo, Japon
93 - Ing, Chendu, Chine
94 - Ing, Taipei, Taiwan
95F - 1ère coupe FOST, Tokyo, Japon
95 - Ing, Seoul, Corée
96F - 2ème coupe FOST, Tokyo, Japon

Auteur	Programme	87	88	89o	89	90	91	92	93	94	95F	95	96F	R
ZhiXing Chen	Handtalk						6	2	1	3	1	1	1	1.8
Michael Reiss	Go4++	10									2	2	2	3.0
Ken Chen	Go Intellect		5	4	3	2	2	1	3	1	4	3	4	3.1
David Fotland	Many Faces of Go	4	8		7		10	6	4	2	3	4	3	3.9
Janusz Kraszek	Star Of Poland	9	4	3	6	3	5	5	2	4				4.9
Mark Boon	Goliath	7	3	2	1	1	1	3						5.4
Kao Kuo Yuan	Stone		7	5	10		7		6	5		5		5.7
Oishi Yasuo	Goro										7		5	6.3
Japanese team	GOG							4						6.5
Dong-Yue Liu	Dragon	2	2	8	12	4	3							6.6
Shimada Kou & Takuo Tabuchi	Takuchan										6		9	7.0
Alfred Knoepfle	Modgo			6			8		5	6				7.0
Martin Mueller & Anders Kierulf	Explorer			1	4	7	13		8			6	8	7.2
Noriaki Sanechica	Igo		10		5	6	4							7.6
Chung Ho Lee	Sason						9		10	7		7		7.9
Yan Shi-Jin	Jimmy										9		6	8.0
Toshikazu Sato	TY '96									9	5		14	8.9
Bruce Wilcox	Nemesis	5	11		2	5	11	7						9.6
Lee Chong-Cheol	Big Stone											8	17	10.5
T. Yoshikawa	Dai Honinbo					9	14	8	13		11			11.4
Hori Tsuneo	Utoro										10		18	11.8
Saito Koichi	Igo Meijin										14		19	15.7
K. Hayashi	Codan		1											
Loh-Tsi Wang	Friday	1												
Kaihu Chen	Peanut	3												
Allan Scarf	Microgo 2	6	13	7										

R est la moyenne pondérée des résultats, le poids de chaque année valant 2/3 du poids de l'année suivante. Les années pendant lesquelles un programme ne participe pas, ses résultats sont obtenus par interpolation. Un programme qui n'a pas participé récemment est pénalisé d'une place par an depuis sa dernière participation.

7.6.3 La coupe FOST 1996

La coupe FOST 1996 s'est tenu les 13 et 14 Septembre 1996 au premier étage de l'association japonaise de Go : la Nihon Ki-in, à Tokyo, Japon. Ce fut le plus grand championnat de Go sur ordinateur jamais organisé. 27 programmes étaient inscrits, et 19 ont participé. Il y eut 9 rondes.

Le classement final fut :

Rang	Nom	Programme	Pays	Score	SDOS
1	Chen Zhixing	Handtalk	Chine	8	
2	Michaël Reiss	Go4++	Angleterre	7	33
3	David Fotland	Many Faces of Go	USA	7	31
4	Ken Chen	Go Intellect	USA	7	29
5	Yasuo Oishi	Goro	Japon	6	
6	Yan Shi-Jim	Jimmy	Taiwan	5	24
7	Takahisa Yoshida	Mutsuki	Japon	5	23
8	Martin Mueller	Explorer	Autriche	5	19
9	Takuo Tabuchi	Takuchan	Japon	5	13
10	Masahiro Tanaka	Biwako	Japon	5	11
11	Tetsuya Wakamatsu	Tai Go	Japon	5	10
12	Tristan Cazenave	Gogol	France	4	17
13	Shinichi Sei	Katsunari	Japon	4	16
14	Toshikazu Sato	Tokyo 96	Japon	4	6
15	Yoshitaka Kohiyama	Gooter	Japon	3	8
16	Hiroshi Yamashita	Aya	Japon	3	7
17	Lee Chong-Cheol	Big Stone	Corée	2	
18	Tsuneo Hori	Utoro	Japon	1	
19	Koichi Saito	Igo Meijin	Japon	0	

La grille du tournoi étant la suivante :

	Programme	1	2	3	4	5	6	7	8	9	Score
1	Handtalk	Bye	+13	+16	+17	+3	-2	+4	+5	+7	8
2	Go4++	+15	Bye	+12	+10	+7	+1	-3	-4	+5	7
3	Many Faces of Go	+19	+5	Bye	+7	-1	+11	+2	+15	-4	7
4	Go Intellect	-16	+12	+19	Bye	+17	+6	-1	+2	+3	7
5	Jimmy	+18	-3	+10	+12	+8	+7	-11	-1	-2	5
6	Gogol	-12	+16	+9	+19	-5	-4	-7	+11	-10	4
7	Explorer	+11	+10	+13	-3	-2	-5	+6	+19	-1	5
8	Big Stone	-10	-17	+14	-9	-19	-16	+13	-18	-15	2
9	Tokyo 96	-17	-11	-6	+8	Bye	-10	-16	+13	+19	4
10	Takuchan	+8	-7	-5	-2	+14	+9	+19	-12	+6	5
11	Goro	-7	+9	+17	+16	+15	-3	+5	-6	+12	6
12	Mutsuki	+6	-4	-2	-5	+16	+18	+15	+10	-11	5
13	Utoro	+14	-1	-7	-15	-18	-19	-8	-9	-17	1
14	Igo Meijin	-13	-15	-8	-18	-10	---	---	---	---	0
15	Biwako	-2	+14	+18	+13	-11	+17	-12	-3	+8	5
16	Katsunari	+4	-6	-1	-11	-12	+8	+9	+17	-18	4
17	Aya	+9	+8	-11	-1	-4	-15	-18	-16	+13	3
18	Tai Go	-5	-9	-15	+14	+13	-12	+17	+8	+16	5
19	Gooter	-3	+18	-4	-6	+8	+13	-10	-7	-9	3

Les 'Bye' correspondent à une victoire sans jouer d'un programme. Les programmes qui ont ainsi bénéficié d'une victoire en plus sont ceux qui avaient déjà participé l'année précédente à la coupe FOST.

Je dois dire que Gogol a perdu deux parties presque terminées, qu'il avait gagné sur le terrain, à cause d'un bug dans la gestion du Ko dû à un changement de dernière minute dans l'algorithme de recherche arborescente. Ce sont l'antépénultième et la dernière partie : les parties contre Explorer et contre Takuchan.

7.6.4 Evaluation du niveau de Gogol

Les quatre premiers programmes de la coupe FOST 1996 ont un niveau similaire et notablement plus élevé que les programmes qui les suivent. Gogol se situe dans l'ensemble des programmes qui sont derrière ces quatre programmes forts. Si l'on prend le classement à long terme de David Fotland, il n'y apparaît pas encore parce qu'il n'a encore pris part qu'à une seule compétition mondiale. Cependant, on peut estimer son niveau d'après les matchs qu'il a effectué contre les programmes qui y sont classés. Le meilleur qu'il ait battu est Goro qui se trouve à la 8ème place. Le plus mauvais programme qui ait battu Gogol est Jimmy qui se trouve à la 16ème place. Je pense que Gogol se trouve actuellement entre la 8éme et la 17ème place mondiale.

7.7 Conclusion

Gogol a appris plusieurs milliers de règles tactiques portant sur des sous-buts intéressants du jeu de Go. Il a appris à calculer efficacement l'achèvement de ces sous-buts et représente le résultat de ses calculs sous forme de jeux combinatoires à valeurs inconnues.

Il a appris correctement tous les problèmes d'une base de 100 problèmes tactiques.

Il comporte actuellement 50 000 lignes de C++ qui représentent trente classes d'objets. Au cours de ma thèse j'ai écrit plus de 150 000 lignes de C++, mais certaines parties du code ne sont plus utilisées actuellement.

Gogol a écrit lui-même 120 000 lignes de C++, toutefois tout ce qu'il a écrit n'est pas intégré dans le programme qui joue au Go.

Chaque coup est joué en 10 secondes sur un Pentium 133 Mhz.

Le programme qui a été présenté à la coupe FOST 1996 est entièrement basé sur l'apprentissage en logique des prédicats. Tout ce que contient ce programme a été écrit après Octobre 1995. Il n'utilise pas les règles apprises avec le mécanisme d'apprentissage des patterns géométriques. Il a créé lui même les connaissances qui lui permettent de restreindre le nombre de coups à envisager dans la recherche arborescente. Pour cela, il a utilisé comme connaissances du Go les règles de transition données dans l'annexe A, les définitions des buts données dans le chapitre sur les explications et les métarègles de compilation décrites dans le chapitre sur la compilation. Les raisons pour lesquelles je pense que l'apprentissage en logique des prédicats est bien meilleur que l'apprentissage direct de patterns géométriques sont expliqués dans le chapitre sur la représentation des connaissances, en résumé, la représentation par la logique des prédicats est plus générale, permet de représenter plus de chose et peut être transformée automatiquement en une représentation spécialisée et rapidement matchable similaire aux patterns géométriques.

Les connaissances du Go données au programme en dehors de celles qui lui ont permis d'apprendre à couper dans ses arbres de recherche arborescente sont les connaissances stratégiques décrites dans ce chapitre et données dans les annexes.

Il reste encore beaucoup de choses à améliorer dans Gogol comme par exemple ne plus utiliser l'heuristique de transitivité dans la construction des groupes, apprendre à résoudre des problèmes plus compliqués que ceux qu'il résout actuellement, améliorer les règles stratégiques, utiliser efficacement les arbres de buts pour représenter les dépendances entre les sous-jeux, améliorer encore la compilation en partageant les prémisses des règles dans les programmes C++, implémenter un module efficace de mesure de l'utilité des règles apprises, apprendre à calculer les problèmes de vie et de mort, utiliser le mécanisme d'apprentissage pour apprendre des règles portant sur d'autres buts ou apprendre des connaissances stratégiques.

7.7 Bibliographie

[Boon 1990] - Marc Boon. *A pattern matcher for Goliath*. Computer Go 13, pp. 13-23, 1990.

[Bouzy 1995] - Bruno Bouzy. *Modélisation cognitive du joueur de Go*. Thèse de l'université Paris 6, 1995.

[Bouzy 1996] - Bruno Bouzy. *There are no winning moves, except the last*. IPMU96, Grenade.

[Brügmann 1993] - Bernd Brügmann. *Monte Carlo Go*. rapport récupérable par ftp ftp.bsdserver.ucsf.edu, 1993.

[Cazenave 1996c] T. Cazenave, *Self Fuzzy Learning*. Proceedings of the International Workshop on Logic Programming and Soft Computing, Bonn, 1996.

[Chen 1992] - Ken Chen. *Attack and Defense, The dominating facet of computer Go.* Heuristic Programming in Artificial Intelligence 3, Ellis Horwood, 1992.

[Enzenberger 1996] - Markus Enzenberger. *The Integration of A Priori Knowledge into a Go Playing Neural Network.* Third Game Programming Workshop in Japan, Hakone, Septembre 1996.

[Fotland 1993] - David Fotland. *Knowledge Representation in The Many Faces of Go*. Second Cannes/Sophia-Antipolis Go Research Day, Février 1993.

[Kierulf 1990a] - Anders Kierulf, Ken Chen, and Jurg Nievergelt. *Smart Game Board and Go explorer : A case study in software and knowledge engineering.* Communications of the ACM, 33(2), Février 1990.

[Kierulf 1990b] - Anders Kierulf. *Smart Game Board : A Workbench for Game-Playing Programs, with Go and Othello as Case Studies.* Thèse de l'ETH Zürich, 1990.

[Müller 1990] - Martin Müller. *The smart game board as a tool for game programmers*. Heuristic programming in Artificial Intelligence 2, Levy/Beal éditeurs, Londres, 1990.

[Müller 1995] - Martin Müller. *Computer Go as a Sum of Local Games : An Application of Combinatorial Game Theory*. Thèse du Swiss Federal Institute of Technology Zürich, 1995.

[Pettersen 1994] - Eric Pettersen. *The Computer Go Ladder*. http://cgl.ucsf.edu/go/ladder.html, 1994.

[Ricaud 1995] - P. Ricaud. *GOBELIN : Une Approche Pragmatique de l'Abstraction Appliquée à la Modélisation de la Stratégie Elémentaire du Jeu de Go*. Thèse de l'Université Paris 6, Décembre 1995.

[Wilcox 1995] - B. Wilcox.*The EGO program*, Message envoyé à la mailing-list Computer-Go, 1995.

[Wolf 1991] - T. Wolf, *Investigating Tsumego Problems with Risiko*, Heuristic programming in Artificial Intelligence 2, Levy/Beal éditeurs, Londres, 1991.

[Wolf 1994] - T. Wolf, *The program GoTools and its computer-generated tsume-go database*, First Game Programming Workshop in Japan, Hakone, Octobre 1994.

[Wolf 1996] - T. Wolf, *About problems in generalizing a tsumego program to open positions* , Third Game Programming Workshop in Japan, Hakone, Septembre 1996.

[Wolfe 1991] - D. Wolfe, Mathematics of Go : Chilling Corridors, Dissertation, Université de Californie à Berkeley, Berkeley, 1991.

Table des Matières du Chapitre 8

8 UN PROGRAMME D'ABALONE 133

8.1 Le jeu d'Abalone 133

8.2 Représentation des connaissances 134

8.3 Définition des sous-buts 135

8.4 Règles stratégiques 135

8.5 Conclusion 135

8 Un programme d'Abalone

Le jeu de Go est un jeu qui a une histoire quadri-millénaire. Les connaissances utilisées actuellement sont donc le fruit de l'expérience de millions d'hommes qui ont joué chacun beaucoup de parties de Go. Le niveau des meilleurs joueurs au Go est comparativement bien meilleur que le niveau de joueurs pratiquant des jeux plus récents pour lesquels on dispose de peu de littérature et pour lesquels les meilleurs joueurs ont été obligés de créer eux-mêmes leurs connaissances. Le jeu d'Abalone est un jeu récent et les gens qui y jouent ne disposent pas de beaucoup de références. Ils ne sont donc pas aidés dans leur apprentissage du jeu comme le sont les joueurs de Go. Etudier les performances de mon système d'apprentissage sur le jeu d'Abalone permet donc de bien montrer qu'il est capable de découvrir des connaissances intéressantes uniquement à partir des règles du jeu.

8.1 Le jeu d'Abalone

L'Abalone se joue sur un damier hexagonal. La figure Abalone 1 donne un aperçu du plateau de jeu vide. Les numéros correspondent aux numéros des cavités. J'ai choisi de numéroter les cavités de 1 à 61 en partant de la plus en haut à gauche et en allant vers la droite puis vers le bas.

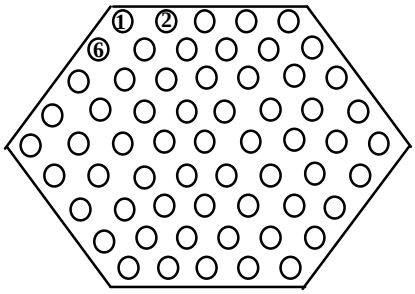


Figure Abalone 1

Au début de la partie, les boules sont disposées comme indiqué dans la figure Abalone 2. Les ronds avec un croix au milieu représentent les cavités vides. Les ronds noirs représentent les boules noires et les ronds blancs le boules blanches.

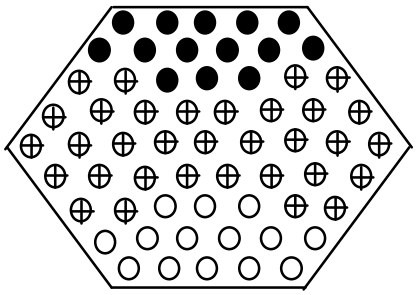


Figure Abalone 2

Les coups permis sont représentés dans la figure Abalone 3. Ces diagrammes indique les changement permis par un coup, le coup consiste à pousser de une à trois de ses boules et de une à deux boules de l'adversaire dans l'une des six directions possibles. Dans un diagramme, un coup permet de remplacer la configuration à gauche de la flèche par la configuration à droite de la flèche. Pour résumer, on a le droit de pousser ses boules sur des cavités vides, on a le droit de pousser les boules de l'adversaire si elles sont en nombre inférieur, on ne peut pas bouger plus de trois de ses boules, et on peut déplacer deux ou trois boules en ligne dans une direction donnée si les cavités de destination sont vides. Ces diagrammes sont valables dans les six directions. Le but de la partie est de pousser les boules de l'adversaire hors du plateau de jeu. Le premier joueur qui a poussé 6 boules de son adversaire hors du plateau de jeu a gagné.

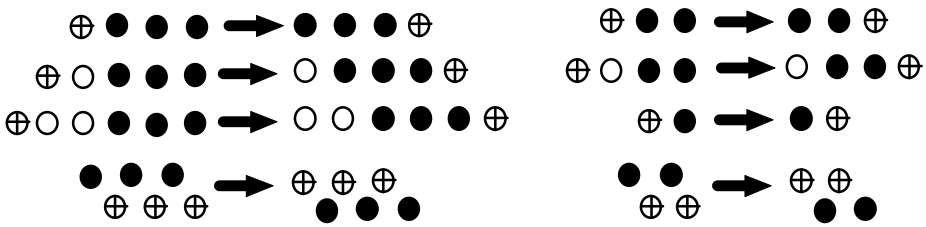


Figure Abalone 3

8.2 Représentation des connaissances

Les connaissances sont représentées avec des prédicats. Les types utilisés pour les variables et les constantes sont les types Cavité, Couleur, Direction, Jeu, Entier et But. Un extrait d'une base de fait représentant un damier d'Abalone est donné dans le tableau Abalone base de faits. Les cavités à l'extérieur du damier sont appelées cav101 à cav130. Les couleurs sont @ (pour amie), O (pour ennemie), + (pour vide) et - (pour extérieur). Il y a six directions d1 à d6.

Couleur_cavité (cav1 @)	Couleur_cavité (cav61 O)	Voisine (cav1 d4 cav7)
Couleur_cavité (cav2 @)	Couleur_cavité (cav100 -)	Voisine (cav1 d5 cav6)
Couleur_cavité (cav3 @)	Couleur_cavité (cav101 -)	Voisine (cav1 d6 cav130)
Couleur_cavité (cav4 @)	...	...
Couleur_cavité (cav5 @)	Couleur_cavité (cav129 -)	Voisine (cav61 d1 cav55)
Couleur_cavité (cav6 @)	Couleur_cavité (cav130 -)	Voisine (cav61 d2 cav56)
Couleur_cavité (cav7 @)		Voisine (cav61 d3 cav115)
Couleur_cavité (cav8 @)	Voisine (cav1 d1 cav100)	Voisine (cav61 d4 cav116)
...	Voisine (cav1 d2 cav101)	Voisine (cav61 d5 cav117)
Couleur_cavité (cav60 O)	Voisine (cav1 d3 cav2)	Voisine (cav61 d6 cav60)

Tableau Abalone base de faits

Les règles donnant les coups légaux et donnant les transitions du damier sont exprimées en utilisant des règles avec variables et métaprédicats décrites dans le chapitre sur la représentation des connaissances. Le tableau Abalone Règles montre une règle sur les coups légaux et une règle sur les transitions de couleurs entre deux situations.

(nom (Regle_coup_legal_1) premisses (present (Couleur (?c)) present (Couleur_opposée (?c ?c1)) present (Couleur_cavité_avant (?cav ?c)) present (Voisine (?cav ?dir ?cav1)) present (Couleur_cavité_avant (?cav1 ?c1)) present (Voisine (?cav1 ?dir ?cav2)) present (Couleur_cavité (?cav2 +))) conclusions (ajoute (Coup_legal (?c ?cav ?dir))))	(nom (Regle_couleur_2) premisses (present (Couleur (?c)) present (Coup (?c ?cav ?dir)) present (Coup_legal (?c ?cav ?dir)) present (Voisine (?cav ?dir ?cav1))) conclusions (ajoute (Couleur_cavité_après (?cav1 ?c))))

Tableau Abalone Règles

8.3 Définition des sous-buts

J'ai trouvé deux sous-buts intéressants au jeu d'Abalone. Le premier sous-but est évident, il consiste à expulser une bille de l'adversaire hors du damier. Le second sous-but est le contrôle d'une case. Il est atteint si une case est de la couleur amie.

(nom (Regle_définition_1) premisses (present (Couleur (?c)) present (Couleur_opposée (?c ?c1)) present (Couleur_cavité_avant (?cav -)) present (Couleur_cavité_après (?cav ?c)) present (Coup (?c1 ?cav1 ?dir)) present (Voisine (?cav2 ?dir ?cav))) conclusions (ajoute (But (?c1 Expulser ?cav2 G))))	(nom (Regle_définition_2) premisses (present (Couleur (?c)) present (Couleur_cavité_après (?cav ?c))) conclusions (ajoute (But (?c Controler ?cav G)))))

Tableau Abalone définitions sous-buts

8.4 Règles stratégiques

La stratégie dans le jeu d'Abalone suit deux principes : expulser les billes de l'adversaire et contrôler le centre. Chaque case est associée à une valeur. La case centrale a une valeur de 5, ses voisines ont une valeur de 4 et ainsi de suite, les cases du bord ont une valeur de 1. Pour évaluer une position, on ajoute toutes les valeurs des cases amies, on retranche toutes les valeurs des cases ennemies, on ajoute 100 par pierre ennemie prise et on retranche 100 par pierre amie prise.

8.5 Conclusion

Mon système a été capable d'apprendre à atteindre des sous-buts intéressants dans le jeu d'Abalone. Pour cela il utilise les mêmes mécanismes que pour le programme de Go. Il représente aussi l'achèvement des sous-buts en utilisant les jeux combinatoires à valeurs inconnues.

L'application de mon système au jeu d'Abalone sert essentiellement à montrer que c'est un système général qui peut s'appliquer aisément à d'autres jeux que le jeu de Go.

Table des Matières du Chapitre 9

9 UN PROGRAMME DE GESTION 136

9.1 Introduction 136

9.2 Représentation des Connaissances 136

9.3 Résolution de Problèmes 137

9.4 Explication 138

9.5 Généralisation 138

9.6 Compilation 139

9.7 Le système d'apprentissage de la gestion 140

9.8 Conclusion 141

9.9 Bibliographie 141

9 Un programme de Gestion

9.1 Introduction

Pour appliquer mon système à l'apprentissage de la gestion [Cazenave 1996e], j'ai utilisé la modélisation d'une entreprise faite par [Alia 1992] dans le cadre d'une thèse sur un modèle didactique d'enseignement de la gestion. Cette thèse s'inscrit dans le cadre de recherches sur l'Enseignement Assisté par Ordinateur en gestion [Gouardères 1986]. La modélisation de C. Alia décompose une entreprise en quatre niveaux hiérarchiques. Chacun de ces niveaux est associé à un but à atteindre. Mon système apprend à agir de façon à satisfaire ces quatre buts : l'équilibre entre les entrées et les sorties au niveau Physique, la fixation des prix au niveau Valorisé, le maintien d'une trésorerie positive au niveau Monétaire, l'obtention d'une bonne rentabilité financière au niveau Financier.

Je décris tout d'abord la façon dont sont représentées les connaissances, puis, je montre sur un exemple du niveau Monétaire comment fonctionnent la résolution de problèmes, les explications, la généralisation puis la compilation. Le niveau Monétaire est celui qui est le plus complexe à apprendre. J'expose ensuite l'ensemble du système d'apprentissage de la gestion d'une entreprise. Enfin, je conclus sur cette application de mon système.

9.2 Représentation des Connaissances

La théorie de l'entreprise est représentée en utilisant le même langage que celui employé pour la théorie du jeu de Go. Seuls les prédicats changent. Le tableau Gestion 1 donne deux exemples de règles du premier ordre concernant la calcul de la trésorerie au niveau Monétaire.

La première règle du tableau Gestion 1 (la règle Règle_trésorerie_1) calcule les encaissements au temps ?t2. Pour cela elle multiplie la quantité de produits vendue au temps ?t par le prix de vente au temps ?t ; elle ajoute au temps ?t le délai d'encaissement ?t1 des ventes pour connaître l'encaissement au temps ?t2 = ?t + ?t1.

La deuxième règle du tableau Gestion 1 (la règle Règle_trésorerie_2) calcule la trésorerie au temps ?t1. Pour cela elle ajoute à la trésorerie du temps ?t = ?t1 - 1 l'encaissement et elle ôte le décaissement et la quantité de travail valorisée.

Règle_trésorerie _1:	Règle_trésorerie _2:
prémisses (présent (Quantité_Vendue (?t ?n1)) présent (Prix_de_Vente (?t ?n2)) présent (Delai_Encaissement (?t1)) egal (?t2 addition (?t ?t1)) egal (?n3 multiplication (?n1 ?n2))) conclusions (ajoute (Encaissement (?t2 ?n3)))	premisses (présent (Trésorerie (?t ?n)) présent (Encaissement (?t ?n1)) présent (Quantité_Travail (?t ?n3)) présent (Décaissement (?t ?n4)) egal (?t1 addition (?t 1)) egal (?n5 soustraction (soustraction (addition (?n ?n1) ?n3) ?n4))) conclusions (ajoute (Trésorerie (?t1 ?n5)))

Tableau Gestion 1

Le tableau Gestion 2 donne un exemple de mémoire de travail du système. Les règles de la théorie du domaine s'appliquent sur la mémoire de travail qui ne contient que des prédicats et des constantes alors que les règles contiennent en plus des métaprédicats et des variables.

Cette mémoire de travail donne la trésorerie au début de l'activité. Elle donne aussi les opérations d'achats et de ventes qui sont effectuées lors des cinq premiers jours de l'activité de l'entreprise, les prix d'achat et les prix de vente, la quantité de travail nécessaire pour transformer les matières premières en produits manufacturés. Elle donne enfin les délais d'encaissement des ventes et les délais de décaissement des achats.

Trésorerie (0 5000)	Prix_de_Vente (5 160)	Prix_Produit (4 70)
Quantité_Vendue (0 10)	Quantité_produits_achetés (0 10)	Prix_Produit (5 70)
Quantité_Vendue (1 10)	Quantité_produits_achetés (1 10)	Quantité_Travail (0 700)
Quantité_Vendue (2 10)	Quantité_produits_achetés (2 10)	Quantité_Travail (1 700)
Quantité_Vendue (3 10)	Quantité_produits_achetés (3 10)	Quantité_Travail (2 700)
Quantité_Vendue (4 15)	Quantité_produits_achetés (4 15)	Quantité_Travail (3 700)
Quantité_Vendue (5 15)	Quantité_produits_achetés (5 15)	Quantité_Travail (4 1050)
Prix_de_Vente (0 170)	Prix_Produit (0 70)	Quantité_Travail (5 1050)
Prix_de_Vente (1 170)	Prix_Produit (1 70)	Delai_Decaissement (3)
Prix_de_Vente (2 170)	Prix_Produit (2 70)	Delai_Encaissement (2)
Prix_de_Vente (3 170)	Prix_Produit (3 70)	Début_activité (0)
Prix_de_Vente (4 160)		

Tableau Gestion 2

9.3 Résolution de Problèmes

La résolution de problèmes est une étape déductive qui consiste à matcher les règles de la théorie du domaine jusqu'à ce qu'aucun fait nouveau ne puisse être déduit. Le tableau Gestion 3 donne les faits déduits à partir de la mémoire de travail du tableau Gestion 2 et des règles de la théorie du domaine de la gestion.

Les faits déduits correspondent à la simulation du fonctionnement de l'entreprise en tenant compte des informations données par la base de faits initiale. Les règles de la théorie du domaine de la gestion concernant la trésorerie, calculent les encaissements et les décaissements pour tous les jours pour lesquels on a des informations, en tenant compte des délais de décaissement qui sont imposés par les fournisseurs de l'entreprise et des délais d'encaissement que l'entreprise impose à ses clients. Une fois les encaissements et les décaissements calculés, et connaissant les quantités de travail journalières, on peut calculer l'évolution de la trésorerie. On peut ainsi connaître les jours où la trésorerie a été positive et les jours où elle a été négative.

Encaissement (0 0)	Décaissement (3 700)	Trésorerie (4 4900)
Encaissement (1 0)	Décaissement (4 700)	Trésorerie_positive (4)
Encaissement (2 1700)	Décaissement (5 700)	Trésorerie (5 4850)
Encaissement (3 1700)	Décaissement (6 700)	Trésorerie_positive (5)
Encaissement (4 1700)	Décaissement (7 1050)	Trésorerie (6 4800)
Encaissement (5 1700)	Décaissement (8 1050)	Trésorerie_positive (6)
Encaissement (6 2400)	Trésorerie (1 4300)	Trésorerie (7 6500)
Encaissement (7 2400)	Trésorerie_positive (1)	Trésorerie_positive (7)
Décaissement (0 0)	Trésorerie (2 3600)	Trésorerie (8 7850)
Décaissement (1 0)	Trésorerie_positive (2)	Trésorerie_positive (8)
Décaissement (2 0)	Trésorerie (3 4600)	Trésorerie (9 6800)
	Trésorerie_positive (3)	Trésorerie_positive (9)

Tableau Gestion 3

9.4 Explication

Le module d'explication trouve les faits qui sont à l'origine de la déduction d'un fait intéressant. Son but est de créer une règle expliquant pourquoi ce fait est déductible, en utilisant uniquement des faits qui représentent la situation avant le temps de ce fait. Pour cela, le module d'explication retourne dans la trace des règles exécutées durant la résolution du problème, en remplaçant les faits qui représentent la situation au temps ?t par des faits qui représentent la situation avant le temps ?t. Dans notre exemple, le module d'explication sélectionne le fait 'Trésorerie_positive (4)' et trouve l'explication du tableau Gestion 4.

L'intérêt des faits est lié aux buts que le système cherche à atteindre. Au niveau Monétaire, le système cherche à avoir une trésorerie positive. Les faits qui représentent que la trésorerie est positive ou négative sont intéressants à expliquer. C'est en effet l'explication de ces faits qui va permettre de créer des règles qui vont prévoir si le but va être atteint ou non.

Trésorerie (0 5000)	egal (2 addition (1 1))	superieur (6 4)
Quantité_Vendue (0 10)	egal (3 addition (3 1))	Delai_Decaissement (3)
Quantité_Vendue (1 10)	egal (4 addition (3 1))	Delai_Encaissement (2)
Prix_de_Vente (0 170)	egal (2 addition (0 2))	Quantité_produits_achetés (0 10)
Prix_de_Vente (1 170)	egal (3 addition (1 2))	egal (1700 multiplication (170 10))
Prix_de_Vente (2 170)	egal (4 addition (2 2))	egal (1700 multiplication (170 10))
Prix_Produit (0 70)	egal (5 addition (3 2))	egal (700 multiplication (70 10))
Quantité_Travail (0 700)	egal (3 addition (0 3))	egal (4300 soustraction (addition (5000 0) 0 700))
Quantité_Travail (1 700)	egal (4 addition (1 3))	egal (3600 soustraction (addition (4300 0) 0 700))
Quantité_Travail (2 700)	egal (5 addition (2 3))	egal (4600 soustraction (addition (3600 1700) 0 700))
Quantité_Travail (3 700)	superieur (5 4)	egal (4900 soustraction (addition (4600 1700) 700 700))
egal (1 addition (0 1))	egal (6 addition (3 3))	superieur (4900 0)
		Début_activité (0)

Tableau Gestion 4

Si toutes les conditions de l'explication sont réunies, on a une trésorerie positive au quatrième jour.

9.5 Généralisation

L'étape de généralisation consiste à transformer une règle qui s'applique spécifiquement sur l'exemple, et qui ne contient que des constantes, en une règle plus générale qui s'appliquera sur beaucoup plus d'exemples parce qu'elle contient des variables. Une constante n'est remplacée par une variable que dans certains cas pour éviter de créer des règles trop générales et donc fausses. Le système ne généralise que les constantes qui sont des instanciations de variables et pas les 'vraies' constantes qui sont aussi des constantes dans les règles de la théorie du domaine. Les conditions de la nouvelle règle plus générale sont données dans le tableau gestion 5.

Le mécanisme de généralisation qui tient compte de l'origine des constantes pour décider de leur généralisation est propre à mon système. Je ne connais pas d'autre système qui utilise cette information pour généraliser les explications qu'il donne. Ce mécanisme est très général et permet

d'utiliser avec rigueur mon système d'apprentissage dans des domaines très variés. Le domaine de la gestion est un bon exemple de domaine dans lequel la distinction entre constantes et variables instanciées est très importante. Par exemple les 1 et le 0 des prédicats du tableau Gestion 5 ne doivent en aucun cas être remplacés par des variables, sinon la règle donnerait des conclusions absurdes. Par contre la variable ?n doit absolument être une variable, sinon la règle n'aurait pas d'intérêt car elle ne serait jamais appliquée.

Trésorerie (?t ?n)
Quantité_Vendue (?t ?q)
Quantité_Vendue (?t1 ?q1)
Prix_de_Vente (?t ?sp)
Prix_de_Vente (?t1 ?sp1)
Prix_Produit (?t ?p)
Quantité_Travail (?t ?qw)
Quantité_Travail (?t1 ?qw1)
Quantité_Travail (?t2 ?qw2)
Quantité_Travail (?t3 ?qw3)
egal (?t1 addition (?t 1))
egal (?t2 addition (?t1 1))
egal (?t3 addition (?t2 1))
egal (?t4 addition (?t3 1))
egal (?t2 addition (?t ?de))
egal (?t3 addition (?t1 ?de))
egal (?t4 addition (?t2 ?de))
egal (?t5 addition (?t3 ?de))
egal (?t3 addition (?t ?dd))
egal (?t4 addition (?t1 ?dd))
egal (?t5 addition (?t2 ?dd))
superieur (?t5 ?t4)
egal (?t6 addition (?t3 ?dd))
superieur (?t6 ?t4)
Delai_Decaissement (?dd)
Delai_Encaissement (?de)
Quantité_produits_achetés (?t ?qb)
egal (?si multiplication (?sp ?q))
egal (?si1 multiplication (?sp1 ?q1))
egal (?bo multiplication (?p ?qb))
egal (?n1 soustraction (addition (?n 0) 0 ?qw))
egal (?n2 soustraction (addition (?n1 0) 0 ?qw1))
egal (?n3 soustraction (addition (?n2 ?si) 0 ?qw2))
egal (?n4 soustraction (addition (?n3 ?si1) ?bo ?qw3))
superieur (?n4 0)
Début_activité (?t)

Tableau Gestion 5

La conclusion de la nouvelle règle est ‘ajoute (Trésorerie_positive (?t4))’.

9.6 Compilation

Après leur généralisation, les règles sont vraies et générales, mais elles ne sont pas efficaces et les matcher prend beaucoup de temps. Pour pouvoir les matcher efficacement, je les compile en simplifiant les expressions et en réordonnant les prédicats [Cazenave 1996b]. La compilation transforme la règle du tableau Gestion 5 en celle du tableau Gestion 6.

Delai_Encaissement (2) Delai_Decaissement (3) Début_activité (?t) Trésorerie (?t ?n) Quantité_Vendue (?t ?q) Quantité_Vendue (addition (?t 1) ?q1) Prix_de_Vente (?t ?sp) Prix_de_Vente (addition (?t 1) ?sp1) Quantité_produits_achetés (?t ?qb) Prix_Produit (?t ?p) Quantité_Travail (?t ?qw) Quantité_Travail (addition (?t 1) ?qw1) Quantité_Travail (addition (?t 2) ?qw2) Quantité_Travail (addition (?t 3) ?qw3) superieur (soustraction (addition (?n multiplication (?sp ?q) multiplication (?sp1 ?q1)) ?qw ?qw1 ?qw2 multiplication (?p ?qb) ?qw3) 0) equal (?t4 addition (?t 4))

Tableau Gestion 6

La conclusion de la règle est toujours ‘ajoute (Trésorerie_positive (?t4))’.

9.7 Le système d'apprentissage de la gestion

La modélisation d'une entreprise est tirée de l'analyse effectuée dans [Alia 1992]. Ce modèle comporte quatre niveaux hiérarchiques représentés dans la figure Gestion 1. Chaque niveau est associé à une but. Mon système apprend à atteindre ces buts pour chaque niveau.

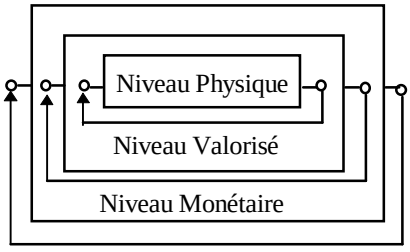


Figure Gestion 1

Au niveau physique, le système apprend à acheter et à produire en fonction des ventes attendues. Le tableau Gestion 7 donne des règles de la théorie du domaine du niveau Physique. Le système apprend à fixer les valeurs des variables ?n et ?n1 dans les prédicats ‘Quantité_Produits_achetés (?t ?n)’ et ‘Quantité_Travail (?t ?n1)’.

Regle_Produits_Manufacturés_1 : premisses (présent (Stock_MP_après_Vente (?t ?n)) présent (Stock_Produits (?t ?n1)) présent (Quantité_Produits_achetés (?t ?n2)) egal (?n3 addition (?n1 ?n2)) présent (Quantité_Travail (?t ?n4)) superieur(?n3 ?n4) egal (?n5 addition (?n ?n3 ?n4)) egal (?t1 addition (?t 1))) conclusions (ajoute (Stock_MP_avant_Vente (?t1 ?n5)))	Règle_Vente_1: premisses (présent (Stock_MP_avant_Vente (?t ?n)) présent (Quantité_Vendue (?t ?n1)) supérieur (?n ?n1) egal (?n2 soustraction (?n ?n1))) conclusions (ajoute (Stock_MP_après_Vente (?t ?n2)))

Tableau Gestion 7

Au niveau Valorisé, il apprend à calculer le prix auquel les produits doivent être vendus. Le système apprend à fixer la valeur de la variable ?p dans le prédicat ‘Prix_de_Vente (?t ?p)’.

Au niveau Monétaire, il apprend à avoir une trésorerie positive. Ce niveau a déjà été décrit dans les sections précédentes.

Au niveau Financier, il apprend à avoir une bonne rentabilité. Ce niveau n'a pas encore été testé.

9.8 Conclusion

Avant l'apprentissage, le système devait, pour fixer le délai de paiement, essayer un ensemble de valeurs possibles et simuler pour chaque valeur son influence sur la trésorerie en déclenchant les règles de la théorie du domaine plusieurs fois pour connaître l'évolution de la trésorerie. Après l'apprentissage, le système peut déduire une valeur convenable pour le délai de paiement plus vite car au lieu d'utiliser plusieurs fois la théorie du domaine, il matche les règles apprises qui lui donnent directement le résultats de ses choix sur la trésorerie. L'apprentissage permet donc au système de trouver la solution de la fixation des délais de paiement plus rapidement.

J'ai décrit une méthode pour apprendre des règles de prise de décision dans la gestion d'une entreprise, méthode basée sur une représentation des connaissances à base de logique du premier ordre étendue à l'usage des nombres entier et réels. Cette méthode permet à mon système de trouver les solutions à ses problèmes plus rapidement qu'en faisant une simulation des conséquences des différents choix possibles. Cette méthode est générale et a été appliquée à d'autres domaines avec succès. Il existe plusieurs pistes pour améliorer ce système d'apprentissage de la gestion:

- Etendre le langage de représentation pour pouvoir représenter la connaissance de manière plus générale afin d'apprendre des règles plus générales.

- Lui faire trouver par lui-même l'ordre hiérarchique des buts en lui donnant seulement la théorie de l'entreprise et le but le plus élevé qui est d'avoir une rentabilité financière.

- Le tester sur des problèmes de gestion d'une entreprise réelle.

- Matcher les règles plus rapidement en permettant aux règles apprise de partager des ensembles de prémisses.

9.9 Bibliographie

[Alia 1992] C. Alia, *Conception et réalisation d'un modèle didactique d'enseignement de la gestion en milieu professionnel*. Thèse de l'Université Montpellier, 1992.

[Cazenave 1996b] T. Cazenave, *Automatic Ordering of Predicates by Metarules.* 4th International Workshop on Metaprogramming and Metareasonning in Logic, Bonn, 1996.

[Cazenave 1996e] T. Cazenave, *Learning to Manage a Firm*. International Conference on Industrial Engineering Applications and Practice, Houston, 1996.

[Dejong 1986] G. Dejong, R. Mooney, *Explanation Based Learning : an alternative view.* Machine Learning 2, 1986.

[Gouardères 1986] G. Gouardères, *Représentation des connaissances dans le dialogue homme-machine en E.A.O..* Thèse d'état, EUPS Toulouse, Juin 1986.

[Minton 1988] S. Minton, *Learning Search Control Knowledge - An Explanation Based Approach*. Kluwer Academic, Boston, 1988.

[Mitchell 1986] T. M. Mitchell, R. M. Keller, S. T. Kedar-Kabelli, *Explanation-based Generalization : A unifying view*. Machine Learning 1 (1), 1986.

[Pitrat 1990] J. Pitrat, *Métaconnaissances*. Hermès, 1990.

Conclusion

> "Mais quand nous sommes confrontés à dix facteurs différents qui agissent tous les uns sur les autres et se combinent pour engendrer au total un nombre astronomique de variantes, la raison abdique et seule l'intuition s'avère à la hauteur."
>
> Yehudi Menuhin, Le Voyage inachevé (1979)

Mon système d'apprentissage utilise une représentation des connaissances à base de logique des prédicats. Il représente ses connaissances de façon différente suivant qu'il veut apprendre de nouvelles connaissances ou qu'il veut utiliser les connaissances qu'il a apprises. Dans la phase d'apprentissage il utilise une représentation générale qui lui permet d'apprendre des règles s'appliquant dans beaucoup de situations en utilisant peu d'exemples. Toutefois cette représentation générale ne permet pas de filtrer les règles apprises rapidement. Mon système transforme donc les règles apprises générales en règles spécifiques pour pouvoir les filtrer plus rapidement lorsqu'il les utilise dans un autre but que l'apprentissage. Pour effectuer cette transformation, il utilise une mécanisme d'évaluation partielle de certaines prémisses des règles.

De plus, afin de pouvoir s'auto-observer, mon système utilise pour résoudre les problèmes en phase d'apprentissage, une représentation qu'il peut manipuler, en d'autres termes, il interprète ses règles et mémorise leurs déclenchements. Ceci n'est plus utile en phase d'utilisation, c'est pourquoi il compile ses règles en programmes C++ pour pouvoir les utiliser de façon efficace.

J'ai montré que jusqu'ici, les systèmes d'apprentissage à partir d'explications appliqués aux jeux utilisaient certains métaprédicats avec une mauvaise sémantique, ce qui a pour conséquence que l'ordre dans lequel les règles de leur théorie du domaine sont déclenchées est important. Ils utilisent donc une représentation des règles non déclarative comportant des connaissances de contrôle non explicites : l'ordre de déclenchement des règles. La non explicitation de ces connaissances de contrôle les amènent à ne pas inclure dans les règles apprises des prémisses qui devraient pourtant y être présents pour que la règle soit vraie. Ceci les amène donc à créer des règles qui donnent parfois des conclusions fausses.

Un système d'apprentissage qui s'amorce utilise les connaissances qu'il apprend pour apprendre de nouvelles connaissances. S'il apprend des connaissances fausses et les utilise pour apprendre d'autres connaissances, la fausseté des connaissances s'accumule et le système en arrive très vite à créer des connaissances complètement fausses qui détériorent énormément ses performances. Mon système étant un système qui utilise les connaissances qu'il a apprises pour en apprendre de nouvelles, il est primordial qu'il ne crée que des connaissances vraies.

Mon système en outre utilise une représentation des connaissances déclarative. Cela lui permet, lors de la phase d'explication, de trouver tous les prémisses impliqués dans la déduction d'un fait intéressant. Cette complétude de ses explications due à sa théorie du domaine déclarative lui permet d'apprendre des règles dont les conclusions sont toujours vraies.

J'ai montré comment la théorie combinatoire des jeux pouvait être appliquée à des jeux complexes en incluant une représentation de l'inconnu. Cette représentation de l'inconnu permet au système non seulement de ne pas prendre en compte des pans trop importants de l'arbre de recherche, mais lui permet aussi de définir des jeux complexes de façon simple et plus générale que les représentations

précédentes. De plus cette représentation de l'inconnu donne les moyens d'établir une taxonomie entre les jeux. Cette taxonomie est très utile pour éviter au programme de faire des calculs inutiles. Enfin, l'attribution de valeurs différentes à l'inconnu permet de définir plusieurs stratégies de gestion du risque.

L'extension de la théorie combinatoire des jeux à la représentation de l'inconnu agit en synergie avec la contrainte de n'apprendre que des connaissances vraies. La possibilité que donne la théorie des jeux étendue de ne représenter qu'une partie de l'information sur un jeu, permet de créer des règles qui ne concluent que sur la partie d'un jeu qui peut être représentée de façon concise.

Mon système ne possède au départ qu'une définition simple et concise des buts qu'il doit atteindre et un ensemble de règles décrivant les conséquences directes d'une action. A partir des exemples qu'il rencontre, il se spécialise automatiquement en un autre programme qui permet de prévoir efficacement à long terme les conséquences de ses actions sur l'achèvement des buts définis.

Les règles qu'il apprend ont deux propos. Le premier propos est qu'elles permettent de ne sélectionner que les menaces d'atteindre un but pour les noeuds OU de l'arbre ET/OU de recherche. Elles permettent aussi de ne sélectionner qu'un ensemble de coups forcés aux noeuds ET de l'arbre ET/OU. Cette sélection des coups est très efficace puisqu'elle permet de n'envisager que très peu de coups à chaque noeud de l'arbre alors qu'il y a deux cents cinquante coups légaux pour chaque noeud. Le deuxième propos de l'apprentissage est de remplacer le parcours de certaines branches de l'arbre de recherche par le filtrage d'une base de règles. Les règles apprises par mon système lui permettent de s'arrêter plus haut dans l'arbre car elles lui donne la possibilité de prévoir l'achèvement d'un but plusieurs coups à l'avance.

La combinaison de ces diverses méthodes m'a permis d'écrire en une année un programme de Go qui a sa place dans les compétitions mondiales de programmes de Go.

Ma méthode d'apprentissage est générale et peut être appliquée à d'autres domaines que celui du jeu de Go. J'ai donné des exemples d'applications pour le jeu d'Abalone et pour la prévision en Gestion. Dans ces domaines aussi, mon système remplace la recherche combinatoire par le filtrage d'une base de règles apprises.

Je pense que les techniques décrites dans cette thèse n'en sont qu'à leurs débuts et qu'elles montreront toute leur efficacité dans les années à venir, notamment dans les domaines très complexes qui nécessitent beaucoup de connaissances comme le jeu de Go. Introspect n'en est qu'au commencement de son amorçage de l'apprentissage des connaissances du Go. Contrairement aux autres programmeurs de Go qui accroissent eux-mêmes les connaissances de leur programme, j'accrois les connaissances qui permettent à mon programme d'apprendre des connaissances. Introspect crée automatiquement les connaissances qui lui manquent à partir d’exemples de ses erreurs et de ses oublis, alors que les autres programmeurs de Go doivent eux-mêmes analyser les erreurs et les oublis de leurs programmes pour modifier leurs bases de connaissances. De plus la déclarativité des connaissances utilisées dans Introspect autorise l’ajout de nouvelles connaissances sans provoquer des effets de bords qui pourraient dégrader les performances du système.

Bibliographie

[Alia 1992] C. Alia, *Conception et réalisation d'un modèle didactique d'enseignement de la gestion en milieu professionnel*. Thèse de l'Université Montpellier, 1992.

[Alliot 1994] - J. M. Alliot et T. Schiex. *Intelligence Artificielle et Informatique Théorique*. Cepadues Editions 1994.

[Allis et al. 1991] - L. V. Allis, H. J. Van Den Herik, H. J. Herschberg. *Which games will survive*. in Heuristic programming in Artificial Intelligence, Ellis Horwood 1991.

[Barklund 1994] - J. Barklund. *Metaprogramming in Logic*. UPMAIL Technical Report N° 80, 1994.

[Beckstein 1996] C. Bekstein, R. Stolle, G. Tobermann, *Meta-Programming for Generalized Horn Clause Logic*. Proceedings of the 4th International Workshop on Metareasoning and Metaprogramming in Logic, Bonn, 1996.

[Berlekamp 1982] - E. Berlekamp, J.H. Conway, R.K. Guy. *Winning Ways*. Academic Press, Londres 1982.

[Berlekamp 1991] - E. Berlekamp. *Introductory overview of mathematical Go endgames*. Proceedings of symposia in applied mathematics - 43 - 1991.

[Berlekamp 1994] - E. Berlekamp, D. Wolfe. *Mathematical Go Endgames - Nightmares for the Professional Go Player*. Ishi Press International - San Jose, London, Tokyo - 1994.

[Berliner 1979] - H. Berliner. *The B* Tree Search Algorithm : A Best-First Proof Procedure*. Artificial Intelligence 12, pp 23-40, North-Holland 1979.

[Bradley 1979] - Bradley, M.B. *The Game of Go - The Ultimate Programming Challenge ?* Creative Computing 5, 3 (Mars 1979), 89-99.

[Boon 1990] - Marc Boon. *A pattern matcher for Goliath*. Computer Go 13, pp 13-23, 1990.

[Bouzy 1994] - Bruno Bouzy. *Modélisation des groupes au jeu de Go*. Rapport du Laforia, 1994.

[Bouzy 1995] - Bruno Bouzy. *Modélisation cognitive du joueur de Go*. Thèse de l'université Paris 6, 1995.

[Bouzy 1996] - Bruno Bouzy. *There are no winning moves, except the last*. IPMU96, Grenade.

[Brügmann 1993] - Bernd Brügmann. *Monte Carlo Go*. rapport récupérable par ftp ftp.bsdserver.ucsf.edu, 1993.

[Burmeister 1995] - Jay Burmeister, Janet Wiles. *Chess vs Go : a comparison of their computational features*. posté sur la mailing list computer-go, Université de Queensland, Australie, 1995.

[Cazenave 1994] T. Cazenave, *Système Apprenant à Jouer au Go*. 2ème Rencontres Nationales des Jeunes Chercheurs en Intelligence Artificielle, Marseille, 1994.

[Cazenave 1995] T. Cazenave, *Learning and Problem Solving in Gogol : A Go Playing Program*. Rapport n° 95-10 du LAFORIA.

[Cazenave 1996a] T. Cazenave, *Learning to Forecast by Explaining the Consequences of Actions*. First International Workshop on Machine Learning, Forecasting and Optimization, Madrid, 1996.

[Cazenave 1996b] T. Cazenave, *Automatic Ordering of Predicates by Metarules*. Proceedings of the 4th International Workshop on Metareasonning and Metaprogramming in Logic, Bonn, 1996.

[Cazenave 1996c] T. Cazenave, *Self Fuzzy Learning*. Proceedings of the International Workshop on Logic Programming and Soft Computing, Bonn, 1996.

[Cazenave 1996d] T. Cazenave, *Automatic Acquisition of Tactical Go Rules*. Third International Game Programming Workshop, Tokyo, 1996.

[Cazenave 1996e] T. Cazenave, *Learning to Manage a Firm*. International Conference on Industrial Engineering Applications and Practice, Houston, 1996.

[Chase 1973] - W. G. Chase, H. A. Simon,*Perception in chess,* Cognitive Psychology 4, pp 55-81, 1973.

[Chen 1992] - Ken Chen. *Attack and Defense, The dominating facet of computer Go.* Heuristic Programming in Artificial Intelligence 3, Ellis Horwood, 1992.

[Cheng 1995] J. Cheng, *Management of Speedup Mechanisms in Learning Architecture.*. Ph.D. Thesis of Carnegie Mellon University, Janvier 1995.

[Clancey 1986] W. Clancey, *From GUIDON to NEOMYCIN and HERACLES in twenty short lessons*. AI Magazine, 7(3).

[Clancey 1987] W. Clancey, *Knowledge-Based Tutoring - The GUIDON program*. The MIT Press, Cambridge.

[Clark 1978] K. L. Clark, *Negation as failure*. In H. Gallaire and J. Minker, editors, Logic and Data Bases, pp. 293-322, Plenum Press, New York, 1978.

[Conway 1976] - J. Conway,*On Numbers and Games,* Academic Press, Londres/New-York, 1976.

[Davis & al 1977] R. Davis, B. Buchanan, E. Shortliffe, *Production rules as a representation for a knowledge- based consultation program*, Artificial Intelligence 8(1).

[Dejong 1986] - G. Dejong, R. Mooney. Explanation Based Learning : an alternative view. Machine Learning 2, 1986.

[Doorenbos 1995] - R. B. Doorenbos, *Production Matching for Large Learning Systems*, Thèse de l'Université Carnegie Mellon, 1995.

[Einhorn 1988] - H.J. Einhorn, R.M. Hogarth, *Decision Making*. p 127, Cambridge University Press.

[Enderton 1991] - H. D. Enderton. *The Golem Go Program*. Carnegie Mellon University, CMU-CS-92-101, 1991. Rapport récuperable par ftp ftp.bsdserver.ucsf.edu.

[Enzenberger 1995] - Markus Enzenberger. *Neural networks and Go*. Message envoyé à la Computer Go mailing list, 30 Janvier 1995.

[Enzenberger 1996] - Markus Enzenberger. *The Integration of A Priori Knowledge into a Go Playing Neural Network.* Third Game Programming Workshop in Japan, Hakone, Septembre 1996.

[Forgy 1982] - C.L. Forgy, *RETE : A Fast Algorithm for the Many Pattern / Many Object Pattern Matching Problem*, Artificial Intelligence vol. 19, pp 17-37, 1982.

[Fotland 1993] - David Fotland. *Knowledge Representation in The Many Faces of Go*. Second Cannes/Sophia-Antipolis Go Research Day, Février 1993.

[Frege 1891] - Gottlob Frege. *Fonction et Concept.* Ecrits logiques et philosophiques, éditions du Seuil, 1994.

[Gelfond 1989] M. Gelfond, H. Przymusinski, T. C. Przymusinski, *On the relationship between circumscription and negation as failure*, Journal of Artificial Intelligence, 38(1):75-94, February 1989.

[Hardy 1940] - G. H. Hardy. *Hardy l'apologie d'un mathématicien*. Editions Belin.

[Hill 1994] P. M. Hill, J. W. Lloyd, *The Gödel Programming Language*, M.I.T. press, 1994.

[Horacek 1989] - H. Horacek.*Reasoning with uncertainty in computer chess* Advances in Computer Chess 5, pp 43-63, Elsevier, 1989.

[Hsu 1990] - Feng-Hsiung Hsu, Thomas Anantharaman, Murray Campbell, Andreas Nowatzyk. *Un ordinateur parmi les grands maîtres d'échecs*. Pour la Science n° 156, Octobre 1990.

[Ishida 1988] - T. Ishida, *Optimizing Rules in Production System Programs*, AAAI 1988, pp 699-704, 1988.

[Jimenez Dominguez 1990] C. Jimenez Dominguez, *Sur l'explication dans les systèmes à base de règles : le système prose*. Thèse de l'Université Paris 6.

[Junghanns 1995] - A. Junghanns, C. Posthoff, M. Schlosser.*Search with Fuzzy Numbers* International Joint Conference on Fuzzy Systems, pp 979-986, Yokohama, 1995.

[Kierulf 1990a] - Anders Kierulf, Ken Chen, and Jurg Nievergelt. *Smart Game Board and Go explorer: A case study in software and knowledge engineering.* Communications of the ACM, 33(2), Février 1990.

[Kierulf 1990b] - Anders Kierulf. *Smart Game Board : A Workbench for Game-Playing Programs, with Go and Othello as Case Studies.* Thèse de l'ETH Zürich, 1990.

[Laird 1986] - J. Laird, P. Rosenbloom, A. Newell. *Chunking in SOAR : An Anatomy of a General Learning Mechanism*. Machine Learning 1 (1), 1986.

[Laurière 1976] - J.L. Laurière, *Un langage et un programme pour énoncer et résoudre les problèmes combinatoires*, Thèse d'état Paris 6, 1976.

[Lee 1988] - Kai-Fu Lee and Sanjoy Mahajan. *A pattern classification approach to evaluation function learning*. Artificial Intelligence, 36:1-25, 1988.

[Lenat 1983] - D. Lenat. *EURISKO: A program that learns new heuristics and domain concepts*. Artificial Intelligence, 21:61-98, 1983.

[Levinson 1991] - Jeffrey Gould, Robert Levinson. *Method Integration for Experience-Based Learning.* University of California Santa-Cruz, Rapport UCSC-CRL-91-27, 1991.

[Levinson 1992] - Jeffrey Gould, Robert Levinson. *Experience-Based Adaptive Search.* University of California Santa-Cruz, Rapport UCSC-CRL-92-10, 1992.

[Levinson 1993] - Robert Levinson. *Exploiting the Physics of State-Space Search.* University of California Santa-Cruz, Rapport UCSC-CRL-93-38, 1993.

[McCarthy 1980] J. Mc Carthy, *Circumscription - a form of non-monotonic reasoning*. Journal of Artificial Intelligence, 13:27-39, 1980.

[Markovitch 1993] - S. Markovitch, P. D. Scott. *Information Filtering: Selection Mechanisms in Learning Systems*. Machine Learning, 10, pp. 113-151, 1993.

[Minker 1982] - J Minker. *On indefinite data bases and the closed world assumption*. In Proc. 6th Conference on Automated Deduction, pp. 292-308, Springer-Verlag, 1982.

[Minton 1984] - S. Minton. *Constraint-Based Generalization - Learning Game-Playing Plans from Single Examples*. Proceedings of the Fourth National Conference on Artificial Intelligence, 251-254. Los Altos, William Kaufmann, 1984.

[Minton 1988] - S. Minton. *Learning Search Control Knowledge - An Explanation Based Approach*. Kluwer Academic, Boston, 1988.

[Minton 1989] - S. Minton, J. Carbonell, C. Knoblock, D. Kuokka, O. Etzioni, Y. Gil. *Explanation-Based Learning : A Problem Solving Perspective*. Artificial Intelligence 40, 1989.

[Minton 1990] - S. Minton. *Quantitative Results Concerning the Utility of Explanation-Based Learning*. Artificial Intelligence 42, 1990.

[Mitchell 1986] - T. M. Mitchell, R. M. Keller, S. T. Kedar-Kabelli. *Explanation-based Generalization : A unifying view*. Machine Learning 1 (1), 1986.

[Moneret 1996] - R. Moneret. *Mise à jour incrémentale des concepts du jeu de Go*. Rapport de stage du D.E.A. I.A.R.F.A., 1996.

[Müller 1990] - Martin Müller. *The smart game board as a tool for game programmers*. Heuristic programming in Artificial Intelligence 2, Levy/Beal éditeurs, Londres, 1990.

[Müller 1995] - Martin Müller. *Computer Go as a Sum of Local Games : An Application of Combinatorial Game Theory*. Thèse du Swiss Federal Institute of Technology Zürich, 1995.

[Nigro 1993] - J.M. Nigro, *BATELEUR : un système expert modélisant le comportement de joueurs au tarot*, Second European Congress on Systems Science, Prague.

[Nigro 1995] J.M. Nigro, *La conception et la réalisation d'un générateur automatique de commentaires: le système GénéCom*. *Application au jeu du Tarot*. Thèse de l'Université Paris 6.

[Nigro 1996] J.M. Nigro, T. Cazenave, *Constraint-based Explanations in Games*. IPMU96, Grenade.

[Ohlsson 1992] S. Ohlsson, *Constraint-Based Student Modelling*. Jl. of Artificial Intelligence in Education (1992) 3 (4), 429-447.

[Parchemal 1988] - Y. Parchemal. *SEPIAR : un système à base de connaissances qui apprend à utiliser efficacement une expertise.* Thèse de l'Université Paris 6, 1988.

[Pell 1991] - Barney Pell. *Exploratory Learning in the Game of GO*. In D.N.L. Levy and D.F. Beal, editors, Heuristic Programming in Artificial Intelligence 2 - The Second Computer Olympiad. Ellis Horwood, 1991.

[Pell 1993] - Barney Pell. *Logic Programming for General Game-Playing*. Proceedings of the workshop on Knowledge Compilation and Speedup Learning, at Machine Learning Conference, Amherst, Mass., 1993.

[Pettersen 1994] - Eric Pettersen. *The Computer Go Ladder*. http://cgl.ucsf.edu/go/ladder.html.

[Pinson 1987] - S. Pinson, *Méta-modèles et heuristiques de jugement : le système CREDEX. Application à l'évaluation du risque crédit entreprise*.Thèse de l'Université Paris 6, 1987.

[Pitrat 1976] - J. Pitrat. *Realization of a Program Learning to Find Combinations at Chess*. Computer Oriented Learning Processes, Simon J. Ed., Noordhoff, 1976.

[Pitrat 1990] - Jacques Pitrat. *Métaconnaissance futur de l'intelligence artificielle.* Hermès, 1990.

[Pompidor 1992] - P. Pompidor. *Apprentissage Symbolique par Exemples et Contre-Exemples Géométrisables en Prise de Décisions*. Thèse de Doctorat de l'Université des Sciences et Techniques du Languedoc - Montpellier II, 1992.

[Porcheron 1990] - M. Porcheron. *Utilisation de méta-connaissances pour la compilation des règles de production.* Thèse de l'Université Paris 6, 1990.

[Prieditis 1995] - A. Prieditis. *Quantitatively relating abstractness to the accuracy of admissible heuristics*. Journal of Artificial Intelligence, 74, pp165-175, 1995.

[Puget 1987] - J. F. Puget. *Goal Regression with Opponent*. Progress in Machine Learning, Sigma Press, Wilmslow, 1987.

[Quinlan 1984] - J. Ross Quinlan. *Learning efficient classification procedures and their application to chess end games.* Machine Learning, pp 463-482, 1984.

[Ricaud 1995] - P. Ricaud. *GOBELIN : Une Approche Pragmatique de l'Abstraction Appliquée à la Modélisation de la Stratégie Elémentaire du Jeu de Go*. Thèse de l'Université Paris 6, Décembre 1995.

[Reiter 1978] - R. Reiter. *On closed-world data bases*. Journal of Artificial Intelligence, 13:81-132, 1980.

[Reiter 1980] - R. Reiter. *A logic for default theory*. In H. Gallaire and J. Minker, editors, Logic and Data Bases, pp. 55-76, Plenum Press, New York, 1978.

[Reitman 1976] - J. S. Reitman,*Skilled perception in Go : Deducing memory structures from inter-response times,* Cognitive Psychology 8, pp 336-356, 1976.

[Safar 1987] B. Safar, *Le problème des explications négatives dans les Systèmes Experts : Le système POURQUOI-PAS?.* Thèse de l'Université Paris-Sud, 1987.

[Samuel 1959] - A. Samuel, *Some studies in machine learning using the game of checkers*, IBM Journal of Research and Development, 3, pp 210-229, 1959.

[Samuel 1967] - A. Samuel, *Some studies in machine learning using the game of checkers - recent progress*, IBM Journal of Research and Development, 11, pp 601-617, 1967.

[Schraudolph 1994] - N. Schraudolph, *Temporal Difference Learning of Position Evaluation in the Game of Go*, Neural Information Pocessing Systems 6, Morgan Kaufmann, 1994. Accessible par ftp bsdserver.ucsf.edu.

[Simon 1984] - H. Simon, *Why should machine learn?*, Machine Learning : an Artificial Intelligence approach, Springer Verlag 1984

[Stoutamire 1991] - D. Stoutamire, *Machine learning, Game Play, and Go*. MS thesis, Case Western Reserve University, 1991.

[Swartout 1983] W. R. Swartout, *XPLAIN: a System for Creating and Explaining Expert Consulting Programs*. Artificial Intelligence 21(3).

[Tadepalli 1989] - P. Tadepalli. *Lazy Explanation-Based Learning: A Solution to the Intractable Theory Problem*. IJCAI 1989, pp. 694-700, 1989.

[Tambe 1994] - M. Tambe, P. S. Rosenbloom. *Investigating production system representations for non combinatorial match*. Artificial Intelligence 68, pp 155-199, 1994.

[Tesauro 1989] - G. Tesauro, T. Sejnowski, *A parallel network that learn to play backgammon*, Artificial Intelligence, 39, pp 357-390, 1989.

[Tremblay 1985] - J.P. Tremblay, P. G. Sorenson. *The theory and practice of compiler writing.* McGraw-Hill International Editions, 1985.

[Uhry 1991] - J.-P. Uhry, *Complexité des algorithmes*, Les théories de la complexité, pp 86-92, 1991.

[Victorri 1993] - B. Victorri, *Eléments d'une Théorie Géométrique du Jeu de Go*, Second Cannes/Sophia-Antipolis Go Research Day, 1993.

[van Harmelen 1988] - F. van Harmelen, A. Bundy. *Explanation based generalisation = partial evaluation*. Artificial Intelligence 36, pp 401-412, 1988.

[Vermersch 1991] - P. Vermersch. *Les connaissances non conscientes de l'homme au travail*. Le journal des psychologues 84, 1991.

[Wallis & Shortliffe 1984], *Explanatory Power for Medical Expert Systems: Studies in the representation of Causal Relationship for Clinical Consultation*. Tech. Report 82-923. Stanford University.

[Waterman 1970] - D. Waterman, *Generalization learning techniques for automating the learning of heuristics*, Artificial Intelligence 1 pp 121-170, 1970.

[Wilcox 1974] - B. Wilcox, W. Reitman, *Perceptions and representation of spatial relations in a program for playing Go*, Proceedings of the ACM national conference, San Diego, pp 123-127, 1974.

[Wilcox 1976] - B. Wilcox, W. Reitman,J. Reitman,R. Nado,J. Kerwin, *Goals and plans in a program for playing Go*, Proceedings of the ACM national conference, Minneapolis, pp 37-41, 1976.

[Wilcox 1978] - B. Wilcox, W. Reitman, *Pattern recognition and pattern directed inference in a program for playing Go*, in Pattern directed inference systems, édité par D. A. Waterman et F. Hayes-Roth, Academic Press, New York, pp 503-523, 1978.

[Wilcox 1979] - B. Wilcox, W. Reitman, *The structure and performance of the Interim.2 Go program*, Proceedings of the 6th IJCAI, Tokyo, pp 711-719 1979.

[Wilcox 1995] - B. Wilcox.*The EGO program*, Message envoyé à la mailing-list Computer-Go, 1995.

[Wolf 1991] - T. Wolf, *Investigating Tsumego Problems with Risiko*, Heuristic programming in Artificial Intelligence 2, Levy/Beal éditeurs, Londres, 1991.

[Wolf 1994] - T. Wolf, *The program GoTools and its computer-generated tsume-go database*, First Game Programming Workshop in Japan, Hakone, Octobre 1994.

[Wolf 1996] - T. Wolf, *About problems in generalizing a tsumego program to open positions* , Third Game Programming Workshop in Japan, Hakone, Septembre 1996.

[Wolfe 1991] - D. Wolfe, Mathematics of Go : Chilling Corridors, Dissertation, Université de Californie à Berkeley, Berkeley, 1991.

Annexes