Université Paris 8
Laboratoire d'Intelligence Artificielle

Recherche sélective et génération automatique de programmes

Tristan CAZENAVE
cazenave@ai.univ-paris8.fr

Habilitation à diriger des recherches

6 Janvier 2003

Résumé

Je montre comment je suis parti d'une approche de l'Intelligence Artificielle, et plus particulièrement de la programmation du jeu de Go basée sur l'apprentissage par auto-observation pour évoluer vers la génération automatique de programmes qui permettent d'accélerer le développement d'arbres de recherche. J'ai ensuite travaillé sur les choix possibles de représentation de ces programmes, ce qui m'a amené à trouver une équivalence entre les programmes de recherche engendrés automatiquement et un algorithme de recherche original, basé sur des connaissances abstraites, et plus efficace que les algorithmes existant jusqu'alors pour de nombreux jeux. Cet algorithme a été lui-même généralisé par la suite et par la même occasion rendu encore plus efficace. Il a permis de résoudre des jeux comme le Phutball 11x11 et l'Atari-Go 6x6. Afin de l'appliquer à de nombreux jeux et de comparer les algorithmes entre eux je développe une librairie d'algorithmes génériques pour l'Intelligence Artificielle. J'ai le sentiment d'avoir itéré une boucle : partant d'un système de métaprogrammation pour le transformer en système de recherche heuristique, je suis finalement revenu à la métaprogrammation par le biais de la programmation générique d'algorithmes d'IA. Je donne pour tous les thèmes que j'ai abordé les extensions futures de mes travaux. Les différents chapitres représentent ces différents thèmes : l'analyse rétrograde de règles, la métaprogrammation logique, la recherche sélective, la programmation générique pour l'IA, et enfin la programmation des jeux et surtout du jeu de Go. Je termine par un chapitre sur les travaux futurs puis je conclus.

Table des matières

Chapitre 1

Introduction

Je montre ici certaines interactions que j'ai étudiées entre recherche arborescente et utilisation de connaissances. La modélisation des comportements intelligents fait très largement appel à ces deux paradigmes.

Parcourir efficacement un arbre de possibilités pour faire un choix intelligent nécessite-t-il beaucoup de connaissances ? C'est une problématique aussi vieille que l'Intelligence Artificielle. Les chercheurs ont souvent opposé recherche arborescente et connaissances à ce propos, notamment pour la programmation des Échecs. J'ai successivement écrit un méta-système général qui transforme automatiquement un algorithme naïf de recherche arborescente en un algorithme sélectif de recherche basé sur un grand nombre de connaissances. Puis en mettant à plat ce méta-système je l'ai transformé en un système qui fait des recherches sélectives encore plus efficaces avec beaucoup moins de connaissances que le système précédent. J'en suis finalement arrivé à des programmes génériques et efficaces de recherche sélective basés sur des propriétés abstraites des problèmes à résoudre. Le domaine qui sous-tend la plupart de mes expériences est le jeu de Go. Il est particulièrement clair dans ce jeu que les capacités de discernement et de sélectivité dans les choix à envisager sont indispensables à un comportement intelligent, aussi bien pour les machines que pour les êtres humains.

La programmation du jeu de Go étant non résolue, parce que vraiment difficile, mon raisonnement initial, en 1989, fut qu'il fallait utiliser des techniques d'apprentissage pour améliorer automatiquement un programme puisqu'il était trop difficile de le faire à la main. J'ai donc successivement testé plusieurs techniques d'apprentissage sur le jeu de Go, en commençant par les réseaux de neurones, en continuant avec la programmation génétique, puis lors de mon DEA avec la génération automatique de patterns. Durant ma thèse, j'ai développé Introspect qui peut être vu comme une généralisation de l'approche à base de patterns en utilisant un langage basé sur la logique des prédicats qui permettait de représenter des situations plus complexes que les situations locales des patterns. A mesure qu'il résolvait des problèmes, mon système Introspect mémorisait les déclenchements de règles qui avaient permis la résolution, regroupait les règles déclenchées pertinentes dans une seule règle et intégrait cette nouvelle règle dans sa base de règles. C'est pourquoi j'ai appelé ce type d'apprentissage, *apprentissage par auto-observation*. Les règles apprises permettaient de sélectionner de façon sûre les coups à envisager dans une recherche tactique. Au jeu de Go, les problèmes traités étaient les problèmes de capture, de connexions, d'yeux et de vie et de mort. Introspect a aussi été appliqué à l'éjection de billes à l'abalone et à la prise de décision comptable pour la gestion d'une entreprise. Les programmes engendrés par apprentissage permettait de résoudre les problèmes exponentiellement plus rapidement qu'une recherche naïve.

J'ai par la suite réécrit Introspect en Prolog, en transformant l'apprentissage par auto-observation à partir de résolution de problèmes, en l'écriture directe de programmes sans passer par l'étape de résolution de problèmes mais en utilisant des métaprogrammes logiques pour transformer, sélectionner, ordonner et compiler en C des programmes logiques. L'avantage de cette approche par rapport à l'apprentissage par auto-observation est que le système n'a pas besoin d'exemples pour engendrer des programmes et que les programmes engendrés ont une couverture complète des cas possibles. L'inconvénient est qu'on doit écrire plus de métaprogrammes que pour l'apprentissage : on doit écrire des métaprogrammes permettant de détecter la monovaluation de certaines variables et d'autres pour détecter les programmes engendrés qui ne seront jamais exécutés. Le choix des prédicats qu'on utilise pour représenter le domaine pour lequel on veut générer des programmes est très important. Des prédicats différents qui modélisent exactement les mêmes propriétés peuvent amener à des programmes engendrés ayant des propriétés totalement différentes. Par exemple, le choix d'utiliser les prédicats *agauche*, *adroite*, *enhaut*, *enbas* sur une grille à la place du prédicat *voisine* fait augmenter exponentiellement la taille des programmes engendrés alors que les gains en temps de matchage sont faibles.

Les problèmes de représentation et leur influence sur la généralité, l'efficacité et la place mémoire prise par les bases de règles engendrées ont été aussi au coeur d'un autre système que j'ai perfectionné à cette période : un système d'analyse rétrograde pour la génération de règles sur les yeux et la vie et la mort au jeu de Go. Jusqu'alors les systèmes d'analyse rétrograde appliqués aux patterns se contentaient de représenter les patterns sans autre information. Mon apport fut de concevoir et d'appliquer un algorithme d'analyse rétrograde pour les patterns associés à des conditions sur des propriétés extérieures au pattern. Dans le cas du jeu de Go, ces propriétés sont des heuristiques admissibles sur le nombre de coups nécessaires pour capturer des chaînes de pierres. Associer des patterns à ces propriétés abstraites permet d'engendrer des règles beaucoup plus générales que les patterns simples, mais aussi de représenter les règles avec moins de place mémoire.

Suite à cette recherche sur la génération automatique de programmes et de règles et sur ses rapports avec les différentes représentations possibles de programmes, j'en suis venu à ne plus définir qu'un minimum de concepts dans la théorie du domaine qui permet de spécialiser par métaprogrammation les buts au jeu de Go, et à revérifier les définitions dynamiquement plutôt que de les déplier. Poussée jusqu'au bout, cette logique m'a amené à réécrire les programmes engendrés par Introspect comme des programmes dynamiques basés sur la recherche et sur des propriétés abstraites pour rendre cette recherche plus efficace. Ceci revient en quelque sorte à extraire les théorèmes qu'Introspect démontrait sur le jeu et à les transformer en connaissances abstraites assez générales et compactes, puis à utiliser ces propriétés abstraites dans un algorithme de recherche original non déplié. Les résultats me surprirent puisqu'avec un algorithme finalement assez simple à écrire comparé à la complexité d'Introspect, on résout dans des temps similaires plus de problèmes. De plus, comparé à un Alpha-Béta optimisé, ce nouvel algorithme résout les problèmes de capture au jeu de Go beaucoup plus rapidement. La formulation de l'algorithme est générale et le rend facile à appliquer à d'autres jeux alors que l'application d'Introspect à d'autres jeux demande de réécrire une théorie du domaine, des métarègles et un compilateur spécifique. Comme cet algorithme est un algorithme de recherche qui prouve les résultats qu'il obtient, et qu'il utilise des connaissances abstraites, je l'ai appelé *recherche abstraite de preuves* (*Abstract Proof Search*) [25]. A la conférence où je présentais cet algorithme, la présentation qui précédait la mienne concernait un algorithme qui a d´etranges similitudes avec le mien : T. Thomsen présentait la *recherche lambda* [86]. Sur le coup nous avons cru présenter le même algorithme sur le même problème (la capture de pierres au jeu de Go), mais après avoir analysé les deux algorithmes je me suis rendu compte qu'ils avaient en fait des différences qui se sont révélées importantes : la recherche abstraite de preuve développe des arbres dans lesquels le nombre maximum de coups au gain et la profondeur sont liés, alors que la recherche lambda développe des arbres pour lesquels le nombre maximum de coups au gain et la profondeur de recherche ne sont pas liés. De plus la recherche abstraite de preuve utilise des connaissances abstraites alors que la recherche lambda n'en utilise pas ou très peu. De plus les connaissances abstraites utilisées dans la recherche abstraite de preuves permettent des gains de plusieurs ordres de grandeurs [30] sur une recherche sans connaissances comme lambda search. Les comparaisons des deux algorithmes m'ont amené à en créer un troisième qui combinait les avantages des deux : *la recherche abstraite graduelle de preuve* [34, 33]. Dans la recherche abstraite graduelle de preuve, les paramètres pour contrôler la recherche ne sont plus seulement le nombre de coups maximum au gain et la profondeur comme dans la recherche lambda et la recherche abstraite de preuves, mais aussi la forme des arbres de menaces qui doivent être vérifiés à chaque noeud. La recherche abstraite graduelle de preuve permet donc de contrôler plus finement la recherche et a permis de résoudre les jeux de Phutball sur damier 11x11 et d'Atari-Go sur damier 6x6. Une autre amélioration de la recherche abstraite de preuve est *l'élargissement itératif* que j'ai aussi conçu début 2000 [24, 27]. L'élargissement itératif consiste à effectuer une recherche complète pour un ensemble de coups abstraits fixé avant de passer aux sur-ensembles de coups abstraits suivants. Sur des problèmes simples de capture de pierres au jeu de Go, l'élargissement itératif permet de gagner un facteur deux sur la recherche abstraite de preuve tout en résolvant plus de problèmes, lorsque le temps maximum de recherche est fixe. L'élargissement itératif améliore aussi la recherche arbstraite graduelle de preuves. Le dernier développement de cette famille d'algorithmes de recherche est *la recherche avec menaces généralisées* [32]. C'est un algorithme qui représente les arbres de recherches plus simplement que les précédents à l'aide de vecteurs. Il permet de modéliser à la fois la recherche abstraite de preuve, la recherche lambda et la recherche abstraite graduelle de preuves. Il est beaucoup plus rapide que tous ces algorithmes sur la résolution d'Atari-Go, et très probablement aussi sur les autres jeux concernés.

Tester la généralité de mes algorithmes en les appliquant à de nombreux jeux et problèmes m'a demandé d'écrire à chaque fois des programmes assez longs et parfois de réimplémenter plusieurs fois le même algorithme pour des jeux différents (j'ai par exemple implémenté la recherche abstraite graduelle de preuves pour Atari-Go, pour la capture au jeu de Go, pour le Phutball, pour la connexion au jeu de Go et pour la vie et la mort au jeu de Go). De même, comparer mes algorithmes à l'état de l'art demande de réimplémenter les meilleurs algorithmes. J'ai ainsi réimplémenté plusieurs fois Proof Number Search, l'Alpha-Béta optimisé, la recherche lambda, la recherche abstraite de preuves, la recherche abstraite graduelle de preuves et la recherche avec menaces généralisées. La réimplémentation des algorithmes pour différents jeux est une tâche répétitive qui peut être évitée, c'est pourquoi j'ai commencé à développer un librairie générique d'algorithmes d'Intelligence Artificielle. L'utilisation des templates de C++ m'a permis de réutiliser les même algorithmes génériques pour l'Alpha-Béta optimisé et pour la recherche avec menaces généralisées sur l'Atari-Go, le Phutball, la connexion au jeu de Go et la vie et la mort au jeu de Go. Je pense continuer dans cette voie en implémentant des versions génériques de *Proof Number Search*, de la *recherche lambda*, de la *recherche avec dépendance*, d'*IDA**, des *différences temporelles*, des *algorithmes de Monte-Carlo*, et de *l'analyse rétrograde* entre autres. Je pense aussi appliquer ces algorithmes génériques à de nombreux problèmes, comme le jeu de Hex, le jeu d'Abalone ou encore des problèmes de satisfaction de contraintes ou de plus court chemin.

Ces différents algorithmes de recherche sont bien sûr utilisés dans Golois, mon programme de jeu de Go. Mais Golois est plus que ses recherches tactiques. Il utilise aussi des algorithmes pour construire et évaluer des groupes, pour évaluer une position, ainsi qu'une recherche globale et des algorithmes pour évaluer les dépendances entre les résultats de ses recherches.

Mon parcours jusqu'ici paraît faire une sorte de boucle qui part de la métaprogrammation, pour aller vers la recherche heuristique en passant par la représentation des connaissances puis qui continue pour retrouver la métaprogrammation et plus particulièrement la programmation générique. J'ai donc choisi de présenter les divers algorithmes et systèmes auxquels je me suis intéressé dans un ordre quasi chronologique (à l'exception de mon programme de jeu de Go, qui a toujours été présent et qui a accompagné tous mes systèmes). Le deuxième chapitre est donc consacré à l'analyse rétrograde appliquée aux patterns et aux règles tactiques utilisées par Golois qui sont des patterns dont certains éléments sont associés à des conditions sur des propriétés extérieures au pattern. Le troisième chapitre traite de la métaprogrammation essentiellement appliquée au jeu de Go, mais aussi de son évolution vers un système de méta-méta-programmation et à son évolution possible vers un système de découverte d'algorithmes. Le quatrième chapitre traite des algorithmes de recherches que j'ai expérimentés à la suite de tentatives de simplification des approches basées sur la métaprogrammation et qui permettent d'accélerer très fortement la résolution d'une nombreuse famille de jeux. Il existe bien sûr des liens assez forts entre cette famille d'algorithmes de recherche et mes recherches sur la métaprogrammation, notamment en ce qui concerne les abstractions utilisées. Le cinquième chapitre traite d'une voie de recherche que je commence à explorer et qui porte sur une bibliothèque d'algorithmes génériques pour la programmation des jeux, et pour l'Intelligence Artificielle en général. C'est une voie de recherche qui me plaît tout particulièrement puisqu'elle réunit à la fois la métaprogrammation, les algorithmes de recherches heuristiques et les optimisations générales qui les accompagnent, ainsi que les recherches sur la représentation efficace des connaissances, l'abstraction, l'analyse rétrograde et l'apprentissage. Le sixième chapitre décrit l'état actuel de mon programme de jeu de Go, qui est largement basé sur les algorithmes décrits dans le chapitres précédents, et sur quelques algorithmes plus spécifiques au jeu de Go. Pour chacun de ces chapitres, je tente de donner un aperçu des recherches que j'ai mené dans le domaine concerné puis de montrer comment elles peuvent se prolonger. Je regroupe dans un septième chapitre les évolutions possibles de mes travaux.

Chapitre 2

Analyse rétrograde

2.1 Introduction

L'analyse rétrograde a été utilisée pour de nombreux problèmes. Elle permet d'engendrer des bases de positions ou de patterns contenant pour chaque position/pattern possible le résultat et/ou la distance au gain pour cette position. Une fois engendrées, les bases permettent de contrôler et de réduire effectivement les arbres de recherche. De nombreux travaux ont été effectués sur les finales d'Échecs, en commençant par van den Herik et Herschberg [89], suivis par Ken Thompson [83] et d'autres. Lewis Stiller [80] et Ken Thompson [84] ont indépendamment calculé plusieurs bases de données à six pièces. Les bases de donnés de finales d'Échecs ont non seulement permis aux ordinateurs de jouer parfaitement les finales d'Échecs, mais elles ont permis de découvrir de nouvelles connaissances échiquéenes [70] comme l'invalidation de la règle des cinquantes coups ou le résultat de la finale Roi-Fou-Fou-Roi-Cavalier pour certaines positions difficiles. Une autre application réussie de l'analyse rétrograde est le calcul de toutes les finales de Checkers à huit pièces [56] pour le programme Chinook [78]. L'analyse rétrograde a aussi été utilisée pour les problèmes de recherche de plus court chemin, par exemple pour le taquin 4x4. Des travaux initiaux de Joseph Culberson et Johnatan Schaeffer sur la génération de bases de patterns [42] ont permis de diviser par 1000 le nombre de noeuds visités, plus récemment Richard Korf a montré qu'on pouvait combiner plusieurs bases de patterns sur le taquin et obtenir ainsi de bien meilleurs résultats [53]. Richard Korf a aussi écrit un programme d'analyse rétrograde pour le Rubik's cube [52] qui lui a permis de calculer les bases de patterns pour les cubes de coins, et six des cubes de bord. En améliorant l'heuristique admissible d'IDA* avec les bases de patterns, son programme effectue des résolutions optimales en un temps raisonable alors que ce n'est pas possible avec un IDA* sans bases de patterns. La construction dynamique de bases de patterns a aussi été utilisée pour accélérer la résolution de problèmes de Sokoban [50].

J'ai commencé à utiliser l'analyse rétrograde au jeu de Go pour engendrer des patterns simples sur les yeux [8, 12, 10]. J'ai ensuite amélioré mon algorithme d'analyse rétrograde pour le rendre plus rapide, et pour engendrer des patterns associés à des conditions externes aux patterns comme les libertés [19, 26]. J'ai ensuite comparé mon approche basée sur la métaprogrammtion à celle basée sur l'analyse rétrograde de patterns [29]. Enfin j'ai décrit plusieurs méthodes pour réduire la taille des bases

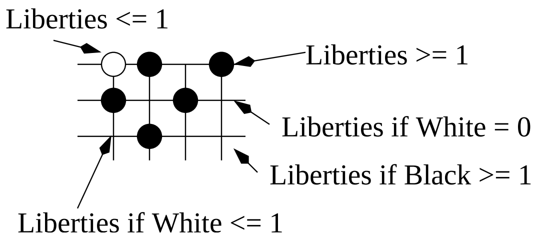

FIG. 2.1 – Une règle engendrée 4x3 sur le bord.

de patterns engendrées [35, 36].

Les autres travaux sur l'analyse rétrograde au jeu de Go sont les bases de données de D. Dyer [45] qui énumèrent des formes vivantes complétement entourées et le travail de Bruno Bouzy sur l'analyse rétrograde à partir d'une fonction d'évaluation [5].

Je pense faire évoluer mon système vers un système qui manipule une représentation plus riche et plus abstraite des problèmes, en enrichissant les conditions utilisées dans les règles.

2.2 Génération de règles au jeu de Go

Une règle est un pattern rectangulaire associée à des conditions sur les libertés externes à la pattern. Les règles sont associées à un but et à un état. Les buts pour lesquels des règles ont été engendrées sont actuellement faire un oeil et vivre. Les règles qui sont mémorisées sont les règles concluant que le but est atteint quel que soit le coup de l'adversaire et les règles concluant que le but peut être atteint en un coup.

2.2.1 Conditions sur les libertés extérieures

Seul un petit nombre de conditions sont associées aux patterns dans les règles. L'algorithme qui engendre les règles considére toujours que la situation est la plus défavorable possible pour le joueur qui atteint le but (Noir), et la plus favorable possible pour le joueur qui tente d'empêcher le but d'être atteint (Blanc). En pratique cela revient à ce que Blanc puisse toujours enlever une liberté à Noir en un seul coup, et qu'il n'ait besoin que d'un coup pour gagner autant de libertés qu'il le veut. Au contraire, Noir ne peut enlever une liberté à Blanc que si c'est la dernière liberté de la chaîne blanche.

Les chaînes noires ne peuvent être associées qu'à des conditions sur le nombre minimal de libertés pour cette raison. De même les coups noirs sur des intersections vides sont associés à un nombre minimal de libertés extérieures après le coup. Les chaînes blanches et les coups blancs sont associés à des nombres maximum de libertés.

La Figure 2.1 donne un exemple de règle engendrée. La condition *Liberties* $>=1$

signifie que la chaîne noire a au moins une liberté à l'exterieur de la pattern en plus des libertés à l'interieur de la pattern. La condition $Liberties\,if\,Black >= 1$ signifie que si Noir joue sur l'intersection vide, la chaîne qui en résultera aura au moins une liberté à l'extérieur de la pattern.

Le nombre minimum de libertés extérieures est une heuristique admissible sur le nombre minimum de coups qui doivent être joués pour pouvoir capturer la chaîne. L'utilisation d'heuristiques admissibles est très utile au jeu de Go comme je l'ai montré pour différents algorithmes [30].

2.2.2 Algorithme d'Analyse Rétrograde

Une approche naïve et inefficace de la génération de règles consiste à générer toutes les règles possibles d'une taille donnée, puis pour chaque règle à effectuer un Alpha-Béta pour lui associer un résultat. C'est l'approche par laquelle j'ai commencé [8, 10, 12]. Peu après, d'autres chercheurs se sont intéréssés aux problèmes de l'analyse rétrograde appliquée aux patterns, notament pour accélerer la résolution du taquin 4x4 [41, 42] et pour le Rubik's cube [52]. Des développements récents ont montré que les bases de patterns pouvaient être combinées plus efficacement que dans ces approches pour obtenir ainsi des gains exponentiels [53].

Après avoir lu les papiers sur l'analyse rétrograde appliquée aux fin de parties d'Échecs [84] et de Checkers [56] j'ai utilisé un algorithme beaucoup plus efficace [19] puisqu'il n'utilisait plus un Alpha-Béta sur chaque règle mais partait des règles pour lesquels le but est atteint (les règles gagnées) pour trouver un sur-ensemble des règles qui pouvaient y amener, puis sélectionner les règles qui permettaient effectivement de retrouver une règle gagnée. J'ai ensuite amélioré cet algorithme en y incluant une fonction pour déjouer les coups, ce qui n'est pas trivial quand on prend en compte les conditions extérieures. Une fois cette fonction écrite, l'algorithme consiste à effectuer alternativement deux passes. La première passe déjoue des coups noirs sur les règles gagnées pour trouver de nouvelles règles gagnantes. La deuxième passe déjoue des coups blancs sur les règles gagnantes et vérifie si la règle déjouée est gagnée pour Noir en montrant que tous les coups blancs amènent à une règle gagnée ou gagnante pour Noir. L'algorithme s'arrête quand une passe ne trouve plus de nouvelles règles gagnées. On a alors engendré toutes les règles possibles pour la taille de pattern courante. En pseudo-code, cela donne donc :

```
RuleGeneration (PatternSize) {
 FindRulesWithTwoPhysicalEyes ();
 while (NewRulesFound) {
  For all new won rules {
   Undo Black moves to find new winning rules;
   Memorize new winning rules;
  }
  For all new winning rules {
   Undo White moves to find new won rules;
   Memorize new won rules;
  }
 }
}
```

L'algorithme commence par engendrer toutes les règles possibles pour les yeux physiquement formés. Il n'y a par exemple qu'une seule règle qui contienne deux yeux physiques pour la pattern 5x2 au bord. Toutes les règles 5x2 au bord sur la vie sont trouvées à partir de cette unique règle.

2.3 Métarègles pour réduire le nombre de règles

Quelques travaux ont porté sur l'utilisation de règles pour réduire la place utilisée par les patterns ou les positions engendrées par analyse rétrograde. Par exemple, R. Gasser a travaillé sur l'utilisation des exceptions pour les bases de données de Nine Men's Morris [48] qu'il fut le premier à résoudre complétement. Ce problème est très lié au problème de l'utilité en apprentissage par explication [62]. Il arrive pour certains problèmes qu'apprendre plus de règles diminue les performances du système à cause d'une augmentation du temps de matchage des règles apprises.

Le problème majeur des bases de règles que j'ai engendré pour le jeu de Go est leur taille. De nombreuses règles utiles sur la vie ne tiennent pas dans un pattern 4x3. Le nombre de règles engendrées augmente exponentiellement avec le nombre d'intersections de la pattern associée. On est actuellement limité aux patterns 5x3 dans le coin. J'ai utilisé plusieurs méthodes pour réduire le nombre de règles engendrées tout en préservant quand cela est possible la complétude des bases. Je considère ces méthodes comme des métarègles puisque ce sont des règles sur les règles.

La première de ces métarègles est décrite dans la première sous-section, elle consiste à détruire toutes les règles qui sont un cas particulier d'une règle plus générale qui a la même taille de pattern ou une taille plus petite. La seconde métarègle détruit les règles qui peuvent être facilement retrouvées dynamiquement, elle est décrite dans la deuxième sous-section. La troisième métarègle utilise les règles engendrées sur les yeux pour réduire le nombre de règles engendrées sur la vie. Le quatrième ensemble de métarègles détecte et élimine les règles redondantes. Je parle ensuite des métarègles pour oter des conditions inutiles, des métarègles pour détecter les règles illégales et irrelevantes. Puis je m'intéresse à des critères pour déterminer l'utilité d'une règle et aux possibles restrictions sur les règles à engendrer.

2.3.1 Métarègles pour supprimer les règles subsumées

Pour chaque nouvelle règle engendrée, le système vérifie qu'elle n'est pas un cas particulier d'une autre règle. Si la règle est originale, il l'ajoute à la base de règles, et parcours la base pour éliminer les règles qui sont des cas particuliers de cette nouvelle règle. Les règles subsumées sont retirées de la base.

Le programme pour vérifier qu'une règle avec un pattern donné subsume une autre règle avec un pattern plus grand n'est pas simple. Le système doit comparer des conditions qui ne sont pas les mêmes puisqu'une partie de l'intérieur du grand pattern correspond à l'extérieur du petit pattern. Par exemple, si une condition est que Noir a au moins deux libertés extérieures en jouant sur une intersection au bord du petit pattern, et si cette intersection est voisine d'une nouvelle intersection vide dans le grand pattern. La condition équivalente dans le grand pattern est que le nombre minimal de libertés extérieures pour Noir est un. Alors que la condition dans le petit pattern est que Noir a deux libertés extérieures.

2.3.2 Suppression des règles qui peuvent être retrouvées dynamiquement

Une autre optimisation pour réduire la taille des bases de règles est de ne pas mémoriser les règles gagnantes, et de ne garder que les règles gagnées qui sont de toutes façons à seulement un coup des règles gagnées. Il est possible de pousser encore plus loin cette optimisation en ne gardant que les règles gagnées aux feuilles d'un arbre de menaces [32] fixé. On obtient ainsi une réduction exponentielle du nombre de règles au prix d'une recherche dynamique dans l'algorithme de recherche qui utilise les règles.

Pour prendre en compte cette optimisation, on modifie l'algorithme d'analyse rétrograde de façon à ce qu'il puisse fonctionner sans les bases de règles gagnantes :

```
RuleGeneration(PatternSize) {
 FindRulesWithTwoPhysicalEyes();
 while(NewRulesFound) {
  For all new won rules {
   Undo Black moves to find new winning rules;
   For these new winning rules {
    Undo White moves to find potentially new won rules;
    Select the rules really won;
    Memorize the new won rules;
   }
  }
 }
}
```

Cette optimisation permet des gains de mémoire importants, par exemple pour les règles 4x3 dans le coin sur la vie, il y a 11404 règles gagnées et 86938 règles gagnantes. En général, le nombre de règles gagnantes est de 7 à 10 fois supérieur au nombre de règles gagnées. De plus les coups gagnants et forcés associés aux règles gagnantes ne sont plus stockés non plus, ce qui gagne d'autant plus de place. La taille des bases de règles est divisée par 10 en otant les règles gagnantes, et peut être encore réduite en utilisant la réduction basée sur les menaces généralisées [32]. L'algorithme pour choisir les feuilles des menaces généralisées de façon optimale est en soit intéressant. Il n'a à être exécuté qu'une seule fois après l'analyse rétrograde.

2.3.3 Suppression de règles sur la vie à l'aide de règles sur les yeux

Des bases de règles ont été engendrées à la fois pour les yeux et pour la vie. Un groupe est vivant s'il a deux yeux, on peut donc utiliser les règles sur les yeux pour détecter la vie. Les deux yeux doivent être indépendants pour assurer la vie. Par exemple si deux yeux ont en commun une intersection vide qui menace les deux

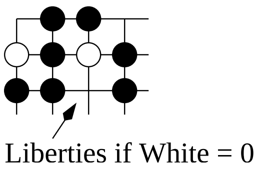

Fig. 2.2 – Deux yeux indépendants dans une règle sur la vie.

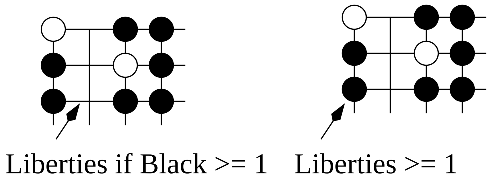

Fig. 2.3 – La première règle est redondante.

yeux à la fois, et qu'aucun coup noir ne sauve les deux yeux à la fois après le coup blanc sur l'intersection, le groupe n'est pas vivant.

Si une règle contient deux yeux indépendants, il n'est pas nécessaire de la garder puisqu'elle peut être déduite à partir de deux règles existantes sur les yeux et de métarègles sur l'indépendance des yeux. Une métarègle simple pour vérifier l'indépendance de deux yeux est de vérifier que l'intersection des deux patterns ne contient pas de pierres blanches et que toutes les intersections vides sont protégées. Cette métarègle permet d'oter de nombreuses règles sur la vie, toutefois elle est parfois fausse. Il est alors intéressant d'engendrer dans une base à part toutes les exceptions à cette règle, qui sont moins nombreuses que les règles otées. Sur les 11,404 règles 4x3 sur la vie dans le coin, 2053 contiennent deux patterns d'yeux. Il est possible que plus la pattern soit grande, plus le pourcentage de nombre de règles otées diminue, puisque le nombre de façons de combiner deux yeux augmente aussi. Il reste encore à évaluer plus précisément les gains de cette métarègle, mais aussi à généraliser l'approche pour, si possible, engendrer automatiquement les métarègles et choisir entre les différentes métarègles possibles et leurs bases d'exceptions associées.

2.3.4 Suppression des règles redondantes

Certaines conditions peuvent être déduites à partir d'autres conditions de la même règle. Et certaines conditions couvrent plus de cas que d'autres. De nombreuses règles peuvent être éliminées en utilisant ces connaissances sur les hiérarchies de conditions. Par exemple, la condition *Liberties if Black* >= 1 dans la règle gauche de la Figure 2.3 peut être déduite de la condition *Liberties* >= 1 de la règle droite.

La première règle de la Figure 2.3 s'appliquera toujours lorsque la deuxième règle s'applique. Mais dans certains cas, la première règle s'appliquera mais pas la deuxième. La première règle est plus générale que la première. La deuxième règle peut donc être détruite.

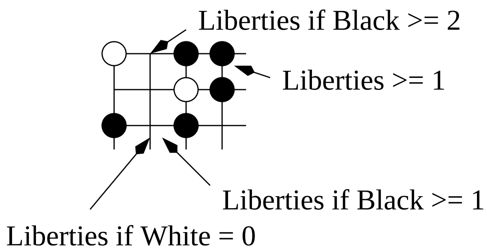

FIG. 2.4 – Une condition inutile.

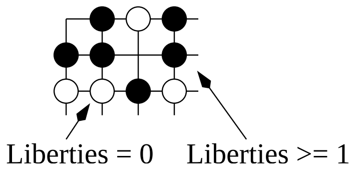

FIG. 2.5 – Une règle illégale.

2.3.5 Suppression des conditions inutiles

Enlever les conditions inutiles des règles permet d'engendrer des règles plus générales qui pourront ainsi subsumer plus de règles redondantes et les oter.

Dans la Figure 2.4, la condition $Liberties >= 1$ est inutile puisqu'on peut la déduire à partir de la condition $Liberties\ if\ Black >= 2$.

2.3.6 Suppression des règles illégales et non pertinentes

La règle de la Figure 2.5 est illégale parce que la chaîne blanche n'a pas de libertés dans la pattern et est associée à une condition $Liberties <= 0$. Or une chaîne sans liberté n'est pas légale. La règle ne peut donc jamais être matchée. Elle est donc détruite.

La règle de la Figure 2.6 est non pertinente, car la condition qu'un coup sur l'intersection n'a pas de libertés extérieures n'est pas compatible avec la condition que la chaîne voisine blanche peut avoir une liberté extérieure. Les règles non pertinentes sont éliminées avant de pouvoir être testées par l'algorithme d'analyse rétrograde. Elle ne sont donc pas engendrées par le système. Il est bon de remarquer qu'une règle est vérifiée si toutes les conditions de la règle sont indépendantes. Or dans l'exemple de la figure 2.6 les deux conditions ne sont pas indépendantes. D'une manière plus générale, deux conditions sur le nombre maximum de libertés qui portent sur les mêmes libertés ne peuvent pas être indépendantes. Les métarègles de suppression des conditions inutiles servent heureusement à éviter ce cas.

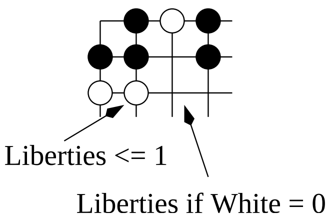

FIG. 2.6 – Une règle non pertinente.

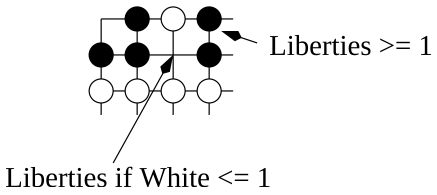

FIG. 2.7 – Une règle peu utile.

2.3.7 Suppression des règles de faible utilité

Beaucoup de règles engendrées ont des conditions sur l'absence de libertés pour les chaînes blanches. Une règle ayant plusieurs conditions de ce type a peu de chances d'être vérifiée, de plus la vie peut souvent être détectée d'une autre manière dans ces cas. La Figure 2.7 montre une règle peu utile.

Une meilleure manière d'estimer l'utilité des règles est de tenir à jour des statistiques de leur utilisation dans le résolveur de problèmes, puis de sélectionner les plus utiles après avoir testé le résolveur sur de nombreux problèmes.

2.3.8 Utilité des abstractions

Afin de montrer l'utilité des abstractions utilisées, et d'évaluer l'impact possible de nouvelles abstractions sur la généralité et l'efficacité des règles engendrées, je montre les conséquences de deux cas pour lesquels on utilise moins d'abstraction : les règles pessimistes et les règles optimistes. Pour différencier les représentations, j'appelle règles réalistes les règles associées à des listes de conditions abstraites.

Les règles pessimistes n'ont pas de conditions associées. Une règle pessimiste prend moins de place mémoire qu'une règle réaliste. Les conditions implicitement associées à la règle sont pessimistes pour Noir. Aucune des chaînes noires n'a de libertés externes, et toutes les chaînes blanches ont un nombre infini de libertés externes. De même pour les chaînes créées par des coups sur les intersections vides au bord du pattern. Ces conditions font que les règles engendrées sont correctes, Noir gagne indépendament de l'environnement externe à la pattern.

Les règles pessimistes peuvent être intéressantes parce qu'il y a moins de règles pessimistes que de règles réalistes pour une taille de pattern donnée. Toutefois, les règles pessimistes qui ont des patterns plus petites que la taille de pattern en cours couvrent moins de cas. Ce qui conduit en fait à avoir pour les tailles de patterns les

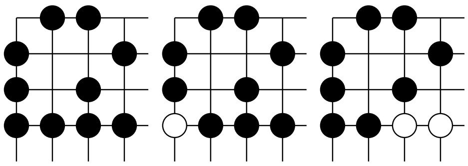

FIG. 2.8 – Plusieurs règles pessimistes pour une règle réaliste.

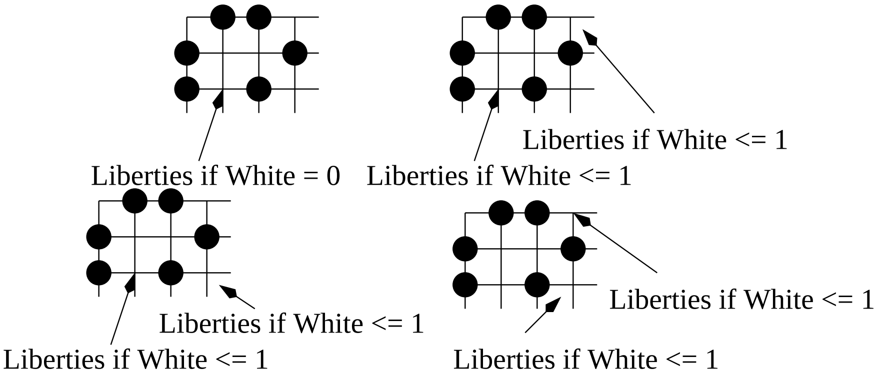

FIG. 2.9 – Plusieurs règles pour la même pattern.

plus grandes, plus de règles pessimistes que de règles réalistes car on doit couvrir des cas non couverts par les règles pessimistes plus petites. Pour une taille de pattern donnée, il y a donc moins de règles pessimistes que de règles réalistes si les bases de règles avec des patterns plus petites sont réalistes. Mais si toutes les bases sont pessimistes, il y a en fait plus de règles pessimistes que de règles réalistes. De plus à tailles de pattern égales, les bases pessimistes couvrent moins de cas que les bases réalistes. On peut voir dans la Figure 2.8 une illustration de cette observation. Elle montre trois règles pessimistes engendrées pour le pattern 4x4 dans le coin. Ces règles peuvent être comparées à la première règle de la Figure 2.9. Ce sont toutes des cas particuliers de la première règle de la Figure 2.9. De plus la première règle de la Figure 2.9 a seulement une condition externe, et sa taille de pattern n'est pas éloignée de la taille du pattern des règles pessimistes. Il y a déjà 24 règles pessimistes correspondantes. C'est encore pire lorsqu'il y a plusieurs conditions externes et une plus grande différence de taille de pattern.

Une autre famille de règles intéressantes à considérer est la famille des règles optimistes. Pour ces règles, on fait l'hypothèse que tout va se passer au mieux pour Noir (celui qui essaie de faire des yeux). On considére que toutes les chaînes noires ont un nombre infini de libertés extérieures, et que les chaînes blanches n'en ont aucune. De même pour les intersections vides voisines du bord du pattern : si Noir y joue, il a un nombre infini de libertés externes, si Blanc y joue il n'en a pas. Les règles optimistes prennent moins de mémoire que les règles réalistes. Aussi bien pour une taille de pattern donnée, et encore moins de mémoire si les bases de règles avec des patterns plus petites sont aussi des règles optimistes. Le problème est que les règles optimistes peuvent se réveler fausses, alors que les règles pessimistes et réalistes sont toujours justes. On voit dans la Figure 2.9 qu'une règle optimiste peut couvrir plusieurs règles réalistes. Toutes les règles réalistes de la Figure 2.9 correspondent à une seule règle optimiste. Pour une taille de pattern donnée, les règles optimistes peuvent être utiles, elles sont beaucoup plus rapides à calculer

TAB. 2.1 – Répartition des règles gagnées dans le coin.

	Tailles des patterns dans le coin				
Profondeur	$4x2$	$3x3$	$5x2$	$4x3$	$6x2$
0	1	10	2	71	4
2	4	18	12	411	36
4	6	29	37	1262	120
6	2	8	78	2737	359
8	1	10	54	3041	682
10	0	3	27	2601	680
12	0	0	13	1050	520
14	0	0	2	203	272
16	0	0	0	22	98
18	0	0	0	6	35
20	0	0	0	0	12
22	0	0	0	0	3
Total	14	78	225	11404	2835

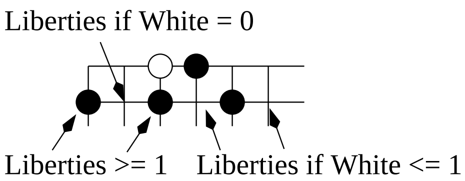

FIG. 2.10 – Une règle 6x2 gagnée de profondeur 16.

que les règles réalistes, et un pattern de règle réaliste est toujours un pattern d'une règle optimiste. On peut donc commencer l'analyse rétrograde réaliste d'une taille de pattern par une analyse rétrograde optimiste, ce qui permettra ensuite d'optimiser l'analyse rétrograde réaliste en ne s'intéressant qu'aux patterns déjà contenus dans les règles optimistes, et en ne perdant pas beaucoup de temps à chercher à prouver des patterns dont on sait qu'elles ne font partie d'aucune règle. Les règles optimistes peuvent aussi être utilisées comme règles heuristiques quand le nombre de règles réalistes est trop grand.

2.4 Résultats Expérimentaux

La table 2.1 donne le nombre de règles sur la vie dans le coin gagnées pour plusieurs tailles de patterns. Les métarègles qui enlèvent les règles moins générales, les métarègles sur la redondance, sur les conditions inutiles, sur les règles illégales et non pertinentes ont été utilisées. Certaines règles remplacent des recherches profondes pour un faible coût de matchage.

La table 2.2 donne le nombre de règles gagnées pour différentes tailles de patterns sur le bord. La table 2.3 donne le nombre de règles gagnées sur les yeux pour les patterns 3x3 et 4x3 dans le centre, ainsi que pour les patterns 3x2, 4x2 et 3x3 au bord. La table 2.4 donne les résultats pour les yeux dans le coin.

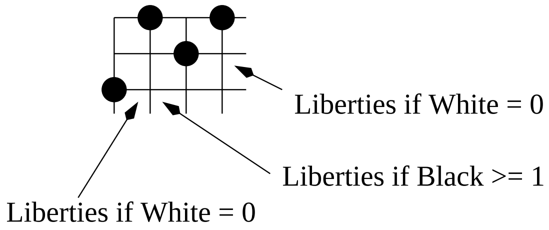

FIG. 2.11 – Une règle gagnée de profondeur 12.

TAB. 2.2 – Repartition des règles gagnées sur la vie au bord.

	Taille des patterns au bord			
Profondeur	$5x2$	$3x4$	$6x2$	$4x3$
0	1	5	1	42
2	5	22	8	156
4	11	48	35	310
6	6	55	124	243
8	5	47	122	200
10	2	16	102	75
12	1	4	77	40
14	0	2	28	8
16	0	0	14	0
18	0	0	18	0
20	0	0	2	0
Total	31	197	531	1074

TAB. 2.3 – Repartition des règles sur les yeux gagnés au centre et au bord.

	Centre		Bord		
Profondeur	$3x3$	$4x3$	$3x2$	$4x2$	$3x3$
0	9	0	1	0	9
2	34	25	3	1	37
4	76	333	4	12	120
6	52	2561	0	59	257
8	60	2831	1	38	154
10	26	4430	0	36	89
12	16	1492	0	6	36
14	6	1486	0	0	23
16	0	283	0	0	0
Total	279	13441	9	152	725

TAB. 2.4 – Repartition des règles sur les yeux gagnés dans le coin.

	Taille des patterns dans le coin			
Profondeur	$2x2$	$3x2$	$4x2$	$3x3$
0	1	1	0	7
2	2	4	11	31
4	1	13	26	111
6	0	34	93	342
8	0	7	133	387
10	0	0	93	247
12	0	0	25	74
14	0	0	0	1
Total	4	59	381	1200

2.5 Evolutions possibles

J'ai effectué des tests sur l'utilité des bases de patterns engendrées pour la résolution de problèmes de vie et de mort [19]. Les résultats expérimentaux ont montré que le nombre de problèmes résolus augmentait significativement pour un temps de recherche supplémentaire acceptable pour le programme de Go. Depuis j'ai beaucoup amélioré le résolveur de problèmes de vie et de mort, notament en améliorant les heuristiques sur les yeux qui sont détectés plus tôt, et en améliorant aussi l'algorithme de recherche à l'aide d'heuristiques de recherche standards. L'utilité des bases de règles avec des conditions sur les libertés décroît avec l'amélioration du résolveur de problème. Toutefois, si on intégre les concepts utilisés par le résolveur de problèmes comme les yeux complets [38] ou les demi yeux dans l'analyseur rétrograde comme suggéré dans la Figure 2.12 où cinq règles avec des conditions sur les libertés peuvent être remplacées par une seule règle plus petite en utilisant des conditions sur les yeux complets ; tout porte à croire que l'utilité des bases de règles augmentera de nouveau. Le nombre de règles engendrées sera plus petit, leur matchage sera probablement plus rapide, les règles couvriront plus de cas, et permettront de couper des arbres de recherche plus grands. Il reste à faire une étude sur l'évolution du nombre de problèmes résolus en fonction du temps maximum alloué à la recherche en utilisant différents niveaux d'abstraction dans les règles engendrées.

Lorsqu'on remplace des parties de l'arbre de résolution par la vérification de règles, les algorithmes de matchage des règles deviennent importants à optimiser. J'utilise un matchage incrémental des patterns, optimisé pour les bases de patterns de grandes tailles. Chaque pattern est représenté comme un entier long, et une recherche dichotomique sur les patterns triés permet de vérifier si un pattern présent sur le damier est dans la base de règles. Il est cependant clair que dans la plupart des cas, le pattern modifié incrémentalement qu'on cherche à matcher ne correspond pas à un pattern de règle engendrée. Il serait probablement intéressant de calculer une table de hachage à partir d'un sous pattern 3x3 du pattern courant, qui aurait $3^9 = 19683$ entrées qui indiquerait pour chaque base correspondant à une taille de pattern donné s'il existe au moins une règle correspondant à la pattern. Ce qui permettrait de se dispenser dans la plupart des cas de la recherche dichotomique. On pourrait aussi comparer le temps mis par la recherche dichotomique sur une grande base et le temps mis par la comparaison avec la liste des patterns ayant la même valeur de hachage. D'autres optimisations sont possibles, par exemple lorsqu'un pattern est vérifié, le système commence par vérifier les conditions sur le nombre maximum de libertés parce que ce sont les conditions qui ont le plus de chances de ne pas être vérifiées. On optimise ainsi le temps de matchage. D'autre optimisations de ce style sont probablement possibles. Comme par exemple la mise en commun dans un arbre des différents ensembles de conditions associés à un pattern, ou la décomposition des patterns en sous patterns de tailles fixes pour optimiser les vérifications.

On peut définir le graphe de dépendance de la génération des règles comme le graphe où les noeuds sont les règles engendrées, et les arcs vont des règles sources vers les règles qui ont été engendrées à partir de ces règles sources en déjouant un coup noir puis un coup blanc. Dessiner ce graphe, et connaître ses propriétés permettrait d'optimiser les interactions entre l'algorithme de recherche pour résoudre les problèmes de vie et de mort et la base de règles. On pourrait par exemple détecter les formes de menaces généralisées [32] les plus courantes dans les bases pour oter toutes les patterns qui peuvent être retrouvées dynamiquement et rapidement par des menaces généralisées prédéfinies.

Une voie de recherche séduisante mais probablement difficile est d'engendrer automatiquement les abstractions en faisant une recherche dans l'espace des concepts possibles. Pour cela, il faudrait gérer automatiquement les relations entre les concepts engendrés automatiquement et l'algorithme d'analyse rétrograde. Actuellement l'algorithme d'analyse rétrograde contient du code spécifique pour chaque condition possible. Ajouter des conditions possibles nécessite actuellement d'écrire des fonctions pour chacune des métarègles décrites précédemment. Des travaux sur la création de ces concepts à partir de bases déjà formés ont été faits principalement dans le cadre du jeu d´Echecs [92, 47], du jeu de Dames [67]. D'autres travaux rentrent dans le cadre plus général des changements de représentations pour améliorer l'apprentissage [88]. Toutefois, il n'existe pas encore de méthode clairement établie dont on connaitrait bien les propriétés, pour créer ces concepts.

Il est possible de réduire le nombre de règles engendrées en utilisant des concepts plus évolués que les libertés. On peut par exemple utiliser le concept d'oeil complet. La figure 2.12 donne un exemple où une règle associée à un pattern 3x2 dans le coin permet de subsumer cinq règles de taille 4x3 dans le coin. De plus, certaines autres règles que les cinq représentées dans la figure seront aussi subsumées et donc otées de la base de règles finale. En utilisant le concept d'oeil complet externe au pattern, on peut donc réduire le nombre de règles. Quel en est le coût ? Autrement dit, le temps de matchage des règles engendrées avec les yeux complets est il plus élevé que le temps de matchage des règles sans les yeux complets ? Je crois que les temps de matchage sont comparables, alors que la place mémoire est beaucoup plus restreinte. Mais il reste à en faire l'expérience pour confirmer cette intuition.

On peut remarquer dans la Figure 2.12 que les conditions 'CompleteEye' et 'Liberties >= 2' ne sont pas nécessairement indépendantes comme on le voit par exemple dans la première règle de la figure où une des deux libertés est aussi un oeil complet. En fait seule l'avant dernière règle de la figure est subsumée par la dernière règle (d'autres règles qui ne sont pas représentée sont aussi subsumées). Toutefois on peut imaginer un mécanisme qui permettrait de vérifier dynamiquement si les deux conditions peuvent être rendues indépendantes par une séquence de coups où l'adversaire joue en premier. C'est le cas des règles 1, 3 et 4 de la figure 2.12. Par contre dans la règle 2, il y a une interaction entre la séquence interne à la règle et la séquence externe à la règle. Résoudre ce genre de problèmes est lié à la résolution des problèmes d'interactions entre buts que je présenterais dans le chapitre sur les

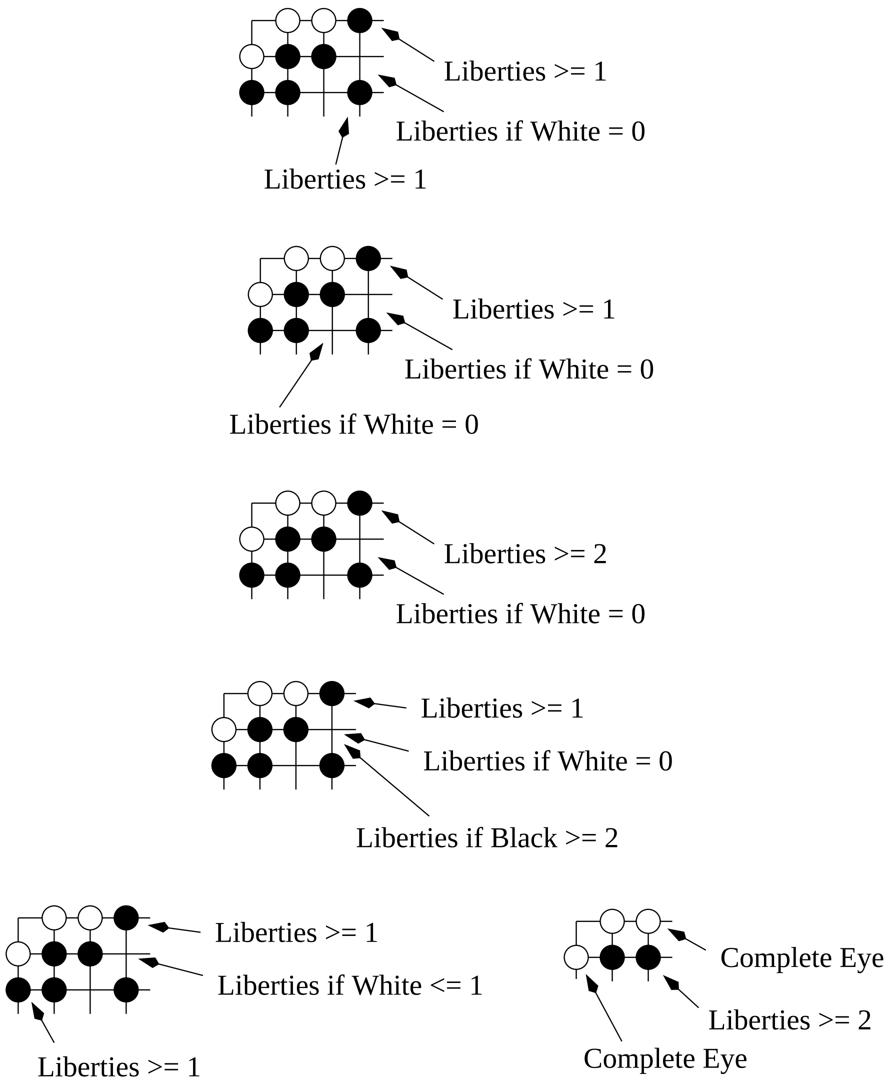

FIG. 2.12 – Le concept d'oeil complet permet de réduire le nombre de règles.

jeux.

2.6 Conclusion

J'ai écrit un système d'analyse rétrograde qui engendre des bases de règles sur les yeux et la vie et la mort au jeu de Go. La particularité de mon approche est l'utilisation de concepts abstraits sur des propriétés externes aux patterns dans les conditions associées aux patterns dans les règles. Les problèmes liés à cette approche sont la taille des bases de règles engendrées, et le temps de matchage des règles comparé au temps de qu'un système basé sur la recherche arborescente met. Une voie de recherche envisagée est l'enrichissement automatique ou non des concepts utilisés dans les conditions des règles, de façon à engendrer des règles plus abstraites qui couvriraient plus de cas avec moins de mémoire et un temps de matchage similaire, voire plus court. Une autre voie de recherche est l'amélioration de l'algorithme de matchage des règles et la comparaison de différentes optimisations du matchage. Il paraît raisonable d'espérer que de grandes bases de règles abstraites associées à un algorithme de recherche performant permettront de faire avancer l'état de l'art pour la résolution de problèmes de vie et de mort.

Chapitre 3

Métaprogrammation logique

3.1 Introduction

Ma thèse [12] a eu pour objet l'étude et l'implémentation du système Introspect qui s'auto-améliore au fur et à mesure qu'il résout des problèmes en s'écrivant lui-même des programmes. Introspect utilise une représentation basée sur des règles en logique du premier ordre pour les buts et les problèmes. Les règles qu'il apprend lui permettent des gains exponentiels sur une résolution de problème naïve.

Plusieurs systèmes ont été écrits pour améliorer les performances de résolveurs généraux de problèmes en généralisant les traces d'exécution et en les intégrant comme nouvelles règles au résolveur. Tous ces systèmes sont basés sur un principe commun qui est celui de la démonstration de théorèmes à partir d'une théorie du domaine. La première application aux jeux de cette méthode est le système d'apprentissage des Échecs de Jacques Pitrat [73] dont l'idée fut reprise plus tard par Steven Minton [60] et qui se retrouve dans de nombreux systèmes généraux par la suite [65, 54, 61, 64, 62, 66]. Alan Bundy et Frank van Harmelen [91] ont montré l'équivalence partielle entre cette forme dynamique de spécialisation de programmes et l'évaluation partielle [81, 57, 39, 72]. Le problème majeur rencontré par ces systèmes est le problème de l'utilité [62, 58] qui a découragé beaucoup de chercheurs de ce domaine. En pratique, par exemple Prodigy crée plus de règles de contrôle d'utilité négative dans le monde des blocs que de règles d'utilité positive [63], Prodigy doit donc procéder à une évaluation à posteriori des règles produites pour détruire celles qui ont une utilité négative. Contrairement aux autres systèmes à base d'apprentissage par explication, Introspect ne souffre pas vraiment du problème de l'utilité. Je pense que cela est dû aux domaines auxquels il est appliqué et à la définition de ses buts d'apprentissage. Une autre raison de son efficacité est l'utilisation de métaconnaissances propres à chacun des domaines auxquels il a été appliqué qui lui permettent de matcher rapidement les règles engendrées [11, 17]. Suite au développment de la version apprentissage par explication d'Introspect, j'ai réécrit presque entièrement le système pour en faire un système de génération automatique de règles, directement à partir de la théorie du domaine (sans apprentissage), mais en utilisant des métaconnaissances spécifiques au domaine pour ne créer que des règles vérifiables et optimisées pour leur vérification [15, 18]. J'ai ensuite étudié comment engendrer automatiquement ces métaconnaissances sur la génération efficace de règles efficaces par des méta-méta-programmes logiques [20].

Je commence donc ce chapitre par un brève description de l'apprentissage par auto-observation. J'expose ensuite quelques métaprogrammes d'optimisation de la vérification des règles engendrées. Puis, je montre comment on peut engendrer les mêmes règles sans apprentissage directement à partir de la théorie du domaine et de métaprogrammes supplémentaires. J'ouvrirai enfin des perspectives sur les évolutions possibles des métaprogrammes que j'ai écrit vers la découverte de nouveaux algorithmes.

3.2 Apprentissage par auto-observation

Les applications principales d'Introspect, mon système d'apprentissage par auto-observation ont été les problèmes de capture de pierres, de connexions et d'yeux au jeu de Go, d'éjection de billes à Abalone [12], de gestion d'entreprise [12, 14, 16], de planification multi-agents [13].

Une des caractéristiques d'Introspect est qu'il démontre les règles qu'il engendre en utilisant la théorie du domaine comme support de ses explications. Par exemple, il sait faire la différence entre une variable instanciée et une constante, ce qui n'est pas le cas d'autres systèmes d'apprentissage par explication. De plus il utilise des connaissances du domaine pour optimiser la vérification des règles engendrées.

Une autre caractéristique primordiale est que dans son application aux jeux, Introspect ne souffre pas du problème de l'utilité qu'ont rencontré tous les autres systèmes de ce type [63]. Cette particularité assez agréable et surprenante pour les autres chercheurs du domaine vient de la façon dont sont écrites les métarègles qui engendrent les règles de contrôle de la recherche. Au lieu de faire comme dans tous les autres systèmes d'apprentissage par explication, où les règles de contrôles ne sont qu'une compilation de l'algorithme de base, Introspect spécialise des métarègles de contrôle qui modifient le comportement de l'algorithme de base (en l'occurence l'Alpha-Béta). Ainsi plutôt que d'avoir un Alpha-Béta compilé sur six demi-coups, ce qui donnerait de très mauvais résultats, Introspect engendre dynamiquement un nouvel algorithme de recherche sélective beaucoup plus efficace que l'Alpha-Béta. J'ai mis environ quatre ans à comprendre exactement pourquoi mon approche marchait alors que toutes les autres approches similaires ont été abandonnées. Comme pour de nombreux algorithmes de résolution de problèmes, les heuristiques simples qui marchent bien semblent évidentes une fois qu'on les connait, mais il se passe souvent beaucoup de temps avant qu'elles ne soient correctement formulées.

La métarègle la plus importante pour comprendre cette caractéristique est la métarègle sur la génération de règles sur les coups forçés décrite dans [15]. Cette métarègle consiste à récupérer les règles qui concluent sur les coups qui atteignent le but, puis à analyser les conditions de ces règles pour trouver l'ensemble complets des coups qui peuvent invalider une des conditions de la règle. Ces coups correspondent aux branches vers la droite des menaces généralisées qui sont présentées dans le chapitre sur la recherche sélective. Les seuls coups autorisés aux branches Min de l'arbre de recherche sont les coups forcés trouvés par ces règles. La règle de contrôle sur la génération de ces règles est qu'on ne garde que celles qui ont au plus cinq coups forcés. La caractéristique qui fait marcher l'algorithme est en partie la sélectivité sur les règles sur les coups forcés en ne gardant que celles qui concluent que sur cinq coups, mais surtout leur intégration dans l'algorithme de recherche qui permet de couper de gros sous arbres en arrêtant la recherche aux noeuds Min lorsqu'aucune règle sur les coups forcés n'est vérifiée. Ainsi on obtient un algorithme de recherche correct (les coups gagnants qu'il retourne marchent toujours) mais qui coupe des sous arbres entiers qui seraient explorés par un Alpha-Béta normal alors qu'ils n'ont que très peu de chances d'aboutir à une solution.

3.3 Optimisations de la vérification

Le problème de l'utilité rencontré par les systèmes d'apprentissage par explication est en grande partie lié à l'efficacité du mécanisme de matchage des règles engendrées. J'ai programmé plusieurs mécanismes pour améliorer les matchage des règles. Certains des mécanismes ne changent pas la représentation du problème, et les règles ainsi engendrées peuvent être réutilisées pour continuer l'apprentissage. D'autres mécanismes modifient la représentation ce qui empêche par la suite un apprentissage efficace ou même empêche tout apprentissage lorsque les programmes sont compilés en C par exemple.

3.3.1 Ordonnancement des conditions

Le premier et le plus important des mécanismes d'optimisation des règles est l'ordonnancement des conditions des règles. C'est une optimisation assez proche des problèmes rencontrés en programmation par contraintes où on utilise des heuristiques pour ordonner les variables à instancier ainsi que les valeurs que prennent ces variables en fonction de la taille des domaines de variables, et du nombre de contraintes dans lesquelles les variables apparaissent. L'ordonnancement de conditions est toutefois un problème plus riche puisqu'on doit non seulement ordonner les conditions contenant des variables, mais aussi les conditions qui se transforment en tests une fois les variables instanciées, les conditions qui instancient plusieurs variables, et que les listes de valeurs possibles pour une variable ne sont pas connues à l'ordonnancement, mais varient dynamiquement en fonction des positions sur lesquelles la règle est matchée. Toutefois, on a des informations statistiques sur les domaines de valeurs d'une variable dans une condition. C'est ce qui va nous permettre d'être tout de même efficace dans l'ordonnancement des conditions. On pourrait envisager un contrôle dynamique de l'ordre d'instanciation des conditions, mais le temps mis à faire le contrôle associé à l'impossibilité de compiler ensuite les règles en C m'ont éloigné de cette solution.

```
meijin (X) :- vivant (X), nihon_ki_in (X).
meijin (X) :- nihon_ki_in (X), vivant (X).
```

Par exemple les deux règles ci dessus donnent les mêmes réponses. Pourtant il est très clair que la première règle est beaucoup moins efficace que la deuxième. Il y a beaucoup moins de personnes appartenant à la Nihon Ki In que de personnes vivantes. Un métaprogramme qui a cette information est capable d'ordonner optimalement les conditions de la règle. Introspect utilise des métarègles qui lui donnent le nombre moyen d'instanciations d'une variable, et utilise cette information pour classer les conditions en commençant par celles qui ont un nombre d'instanciations minimal. Par exemple, les tests ne comportant pas d'instanciations sont toujours placés avant les conditions qui comportent des instanciations puisqu'ils ont un facteur de branchement de un.

3.3.2 Suppression de conditions inutiles

Certaines conditions des règles engendrées sont toujours vraies. Mais les métarègles qui peuvent le détecter font appel à des connaissances spécifiques au domaine. Par exemple, le jeu de Go est joué sur une grille, les voisines des voisines des voisines d'une intersection sont toujours différentes de l'intersection originale. Or dans les règles engendrées il y a des tests pour vérifier que les intersections sont différentes. Introspect utilise des métarègles pour oter ces tests inutiles, et gagne ainsi du temps de matchage et de la place mémoire. Toutes les règles engendrées par apprentissage sont valides puisqu'elles ont été matchées au moins une fois. Dans le cas de la génération de règles par métaprogrammation, des métarègles similaires sont utilisées pour détecter les règles impossibles qui ne pourront jamais être vérifiées à cause de certaines propriétés du domaine, ou de propriétés plus générales.

3.3.3 Coupes dans la vérification des règles

Lorsque la conclusion d'une règle a été vérifiée, il est inutile de la vérifier une nouvelle fois en passant par des chemins différents. Pourtant c'est ce qui se passe si on ne dispose pas d'un mécanisme pour empêcher de déduire plusieurs fois la même conclusion. Introspect donne la priorité dans ses règles aux variables qui sont présentes en conclusion. Ce qui lui permet d'arrêter assez haut dans la règle la vérification lorsque la conclusion a déjà été déduite, et de ne backtracker que sur la dernière variable présente dans la conclusion et non pas inutilement sur toute la suite du sous arbre en dessous de cette variable. Ce mécanisme lui permet de doubler la vitesse de vérification des règles engendrées pour le jeu de Go. Ce mécanisme altère la généralité de l'apprentissage parce qu'il empêche de donner plusieurs explications à une même conclusion, et empêche donc la factorisation des conditions qui se retrouvent dans plusieurs explications. Or la factorisation permet d'engendrer des règles plus générales. Ces coupes ne sont donc effectuées que lorsque le système n'apprend pas.

3.3.4 Changements de représentation

Un mécanisme qui peut être utilisé pour accélerer la vérification de règles est de remplacer un prédicat par plusieurs prédicats plus spécialisés. Cela permet de calculer à la génération des règles certaines conditions qui doivent autrement être revérifiées à chaque fois qu'on essaie de matcher la règle. Par exemple, toujours sur la grille du jeu de Go, on peut représenter les voisines d'une intersection avec le prédicat 'voisine', mais on peut aussi spécialiser ce prédicat et le remplacer par quatre autres prédicats qui sont par exemple 'haut', 'bas', 'droite' et 'gauche'. On peut alors vérifier à la compilation la plupart des conditions qui portent sur le fait que deux intersections soient différentes, ce qui sera autant de tests évités à la vérification. On peut même aller plus loin et oter certaines règles spécialisées pour lesquelles on se rend compte qu'une condition ne peut jamais être vérifiée dans certaines configurations de spécialisation. Toutefois cette spécialisation qui permet en pratique de réduire le temps de vérification des règles de 30% augmente exponentiellement la taille mémoire prise par les règles engendrées. En effet pour chaque condition 'voisine', on crée quatre règles différentes. Or si une règle contient 5 conditions 'voisine' on crée alors $4^5 = 1024$ règles spécialisées à partir de cette seule règle. En représentant les règles spécialisées en arbre de conditions, on réduit la place mémoire et le temps de matchage des règles spécialisées. Mais pas suffisament pour que la spécialisation soit intéressante. On doit considérer pour ce genre de changement de représentation un équilibre entre le temps et la mémoire. Ici il vaut mieux garder la représentation basée sur 'voisine' parce qu'elle n'est pas beaucoup plus lente que la représentation spécialisée mais prend beaucoup moins de place.

Les réflexions sur les changements de représentation m'ont amené par la suite à transformer complétement mon système pour en faire un système basé sur la recherche [25] et des connaissances abstraites [30].

3.3.5 Compilation en C

Pour intégrer efficacement les règles engendrées dans mon programme de jeu de Go, j'ai écrit un compilateur des règles engendrées vers des programmes C qui sont liés au programme de Go. La compilation en C permet de vérifier les règles soixante fois plus vite que leur interprétation [17].

3.4 Métaprogrammation logique

On peut engendrer les mêmes règles que le système d'apprentissage sans spécialiser dynamiquement les buts sur des exemples de buts atteints [18, 15]. Pour cela, on utilise l'opération de dépliage qui consiste à remplacer un prédicat par sa définition en unifiant le prédicat avec la tête de la clause (ou la conclusion de la règle).

Lorsqu'on déplie les conditions des règles, on n'a plus l'assurance que les règles vont s'appliquer. On peut alors engendrer des règles impossibles à vérifier. Si on laisse le dépliage faire, il peut saturer la mémoire avec des règles inutiles. Pour améliorer les règles engendrées, j'ai introduit des métarègles de détection de l'impossibilité. L'impossibilité de vérifier des règles peut venir de propriétés générales comme les propriétés mathématiques, par exemple la métarègle :

```
impossible(ListAtoms):-
member(N=\=N1,ListAtoms),var(N),var(N1),N==N1.
```

qui vérifie que si une règle contient la condition 'N=\=N1' et que les métavariables N et N1 contiennent la même variable, alors la règle est impossible à vérifier. Cette métarègle est très utile et permet d'éliminer de nombreuses règles engendrées mais impossibles à vérifier.

D'autres métarègles sont spécifiques au domaine, par exemple la métarègle suivante est spécifique au jeu de Go :

```
impossible(ListAtoms):-
member(color_string(B,C),ListAtoms),C==empty.
```

Elle élimine les règles qui tentent de trouver des chaînes de pierres vides, ce qui est impossible puisque les chaînes de pierres sont noires ou blanches.

D'autres métarègles importantes sont les métarègles de monovaluation. Elle permettent d'unifier deux variables à l'intérieur d'une même règle à l'aide de connaissances du domaine. Ces unifications permettent par la suite de simplifier les règles et de détecter de nouvelles impossibilités. Un exemple de métarègle de monovaluation est :

```
monovaluation(ListAtoms):-
member(color_string(B,C),ListAtoms),
member(color_string(B1,C1),ListAtoms),
B==B1,C\==C1,C=C1.
```

Elle unifie les variables C et C1 parce qu'une chaîne de pierres ne peut avoir qu'une seule couleur. Or les variables B et B1 sont les mêmes et les variables C et C1 sont différentes alors que C et C1 ont toujours les mêmes valeurs une fois instanciées.

Pour montrer l'utilité de ces métarègles, j'ai testé le dépliage de six règles sur un coup avec une théorie du domaine des règles du jeu de Go. Le but utilisé était la capture de pierres. Sans les métarègles, le système a créé 166 391 568 règles. En utilisant des métarègles d'impossibilité et de monovaluation basiques, seules 106 règles se sont révélées valides et différentes les unes des autres.

3.5 Méta-méta programmation

Parmi les métarègles de d'impossibilité et de monovaluation, certaines sont générales et d'autres sont spécifiques. On peut écrire des méta-méta programmes qui engendrent pour un domaine donné des métarègles spécifiques d'impossibilité et de monovaluation.

3.5.1 Génération de métarègles d'impossibilité

Par exemple dans les jeux de damiers, les caractéristiques topologiques du damier sont en général invariables. Sur un damier de Go, les intersections voisines des voisines des voisines d'une intersection I ne sont jamais égales à I. Si on déplie le prédicat $n\_voisines(X, X, 3)$ avec la définition de n_voisines :

```
n_voisines (X,X,0).
n_voisines (X,W,N1) :-
N is N1 - 1, n_voisines (X,Z,N), voisine (Z,W).
```

On obtient :

```
n_voisines (X,X,3):-
voisine (X,Y), voisine (Y,Z), voisine (Z,X).
```

C'est une règle qui a été correctement générée sur une théorie du domaine correcte mais qui ne peut jamais être vérifiée dans ce domaine. On peut engendrer automatiquement des métarègles d'impossibilité pour la topologie d'un damier puisqu'on a une base de faits qui représente les relations de voisinages et la topologie, et que de surcroît la base de fait est complète et qu'aucune règle de la théorie ne peut modifier les faits représentant la topologie du damier. A partir de là, on peut trouver les ensembles de conditions qui ne seront jamais vérifiées puisqu'elles ne sont pas vérifiées sur une base de faits qui sera toujours la même. Pour trouver l'ensemble minimal des conditions insatisfiables, on peut alors oter une à une les conditions tant que l'ensemble qui reste est impossible à vérifier. On a alors des ensembles minimaux de conditions qui sont invérifiables et on peut les transformer en métarègles d'impossibilité. Pour l'exemple précédent, la métarègle suivante permet de l'éliminer :

```
impossible (ListAtoms):-
member (voisine (X, Y), ListAtoms),
var (X), var (Y), X \== Y,
member (voisine (A, Z), ListAtoms),
A == Y, var (A), var (Z), A \== Z,
member (voisine (B, C), ListAtoms),
B == Z, var (B), var (C), B \== C, C == X.
```

3.5.2 Génération de métarègles de simplification

On peut aussi engendrer des métarègles de simplification des règles par méta-méta programmes. Par exemple la règle suivante est engendrée par dépliage de $n\_voisines(X, D, 4)$:

```
n_voisines (X, D, 4) :-
voisine (X, Y),                 4
X =\= Y,                        4
voisine (Y, Z),                16
Z =\= X,                       12
Z =\= Y,                       12
voisine (Z, W),                48
W =\= X,                       48
W =\= Y,                       36
W =\= Z,                       36
voisine (W, D).               144
```

Les nombres qui suivent les conditions sont le nombre de fois que la condition est testée quand on vérifie la règle.

Au contraire des métarègles d'impossibilité qui détectent les conditions qui ne sont jamais vérifiées, il est inutile dans certains cas de vérifier des conditions car elles sont toujours vérifiées. Par exemple deux intersections voisines sont toujours différentes sur un damier de Go. Les intersections voisines de voisines de voisines sont aussi toujours différentes. On peut engendrer les métarègles de simplification de façon similaire aux métarègles d'impossibilité. En vérifiant pour les faits qui ne varient pas qu'une condition est toujours vérifiée et en trouvant l'ensemble minimal de conditions nécessaires pour que la condition soit toujours vérifiée. On peut alors transformer cette liste de conditions minimale en métarègle de simplification. On obtient alors des métarègles de simplification du style :

```
useless (X =\= Y, ListAtoms):-
member (voisine (X, Y), ListAtoms),
member ( A =\= B, ListAtoms), A == X, B == Y.
```

Ou comme la métarègle suivante qui enlève les tests sur la différence des voisines des voisines des voisines parce qu'elles sont toujours différentes :

```
useless (X =\= Z, ListAtoms):-
member (voisine (X, Y), ListAtoms),
var (X), var (Y), X \== Y,
member (voisine (Y, Z), ListAtoms),
var (Y), var (Z), Y \== Z,
member (voisine (Z, A), ListAtoms),
var (Z), var (A),Z \== Y, A == X.
```

La règle initiale fait 361 instanciations et tests, après avoir été simplifiée par les métarègles de simplification on obtient alors une règle qui ne fait que 261 instanciations :

```
n_voisines (X, D, 4) :-
voisine (X, Y),                 4
voisine (Y, Z),                16
Z =\= X,                       12
voisine (Z, W),                48
W =\= Y,                       36
voisine (W, D).               144
```

3.5.3 Génération de métarègles d'ordonnancement

Contrairement aux approches précédentes qui se basaient sur des statistiques ou des règles heuristiques pour ordonner les conditions des règles [55, 49], j'ai choisi d'engendrer des métarègles d'ordonnancement à partir d'une base de fait, sur laquelle on peut calculer des statistiques sur le nombre moyen de faits vérifiant une condition donnée. Une fois le métaprogramme d'ordonnancement ainsi créé, il ordonne les conditions de façon très satisfaisante et de façon peu coûteuse. Cette approche est la seule approche viable lorsqu'on engendre de très nombreuses règles parce qu'on ne peut pas les ordonner individuellement à partir de statistiques d'utilisation, cela prendrait trop de temps : il arrive que ne matcher qu'une seule règle non ordonnée prenne plusieurs heures. Que faire lorsqu'on en a un million à ordonner sinon utiliser des métarègles d'ordonnancement qu'on peut assez facilement engendrer une fois pour toutes et qui sont plus efficaces que la méthode d'ordonnancement heuristique qui rend le problème très difficile en pratique.

On peut alors engendrer des règles du style :

```
branching (ListAtoms, ListBindVariables, voisine (X, Y), 3.76) :-
member (voisine (X, Y), ListAtoms),
member_term (X, ListBindVariables),
non_member_term (Y, ListBindVariables).
```

```
branching (ListAtoms, ListBindVariables, element_chaine (X, Y), 94.8):-
member (element_chaine (X,Y), ListAtoms),
non_member_term (X, ListBindVariables),
non_member_term (Y, ListBindVariables).
```

3.6 Travaux futurs

3.6.1 Découverte d'algorithmes

On peut transformer beaucoup d'algorithmes simples à écrire mais inefficaces en algorithmes optimisés à l'aide d'un petit nombre d'opérations élementaires que sont le pliage, le dépliage et la définition [72]. J'ai montré qu'on peut redécouvrir un Alpha-Béta simplifié à partir d'un Minimax simplifié à l'aide de ces opérations [23]. La particularité du système que je propose par rapport aux autres systèmes d'évaluation partielle est de diriger les transformations par des mesures des temps d'exécutions des programmes transformés. Ces temps d'exécution ne sont pas suffisants pour décider des voies les plus intéressantes à poursuivre dans la transformation, ils donnent cependant une information très utile pour les programmes qui contrôle la recherche dans l'espace des programmes transformés.

3.6.2 Découverte d'heuristiques admissibles

On peut utiliser Introspect pour spécialiser les heuristiques admissibles au Taquin [21], ce qui permet de découvrir de nouvelles heuristiques qui donnent des valeurs plus élevées et permettent donc de réduire le nombre de noeuds des arbres de recherche développés par IDA* pour résoudre optimalement le Taquin.

Les heuristiques admissibles peuvent ausi être très utiles dans des domaines plus complexes et dans les jeux [30], ou elles permettent des gains énormes lorsqu'elles sont associées à certains algorithmes de recherche [32]. Plutôt que d'écrire ces heuristiques à la main, on peut les engendrer automatiquement à partir des règles du jeu. Ce n'est pas forcément intéressant lorsque cela peut être fait à la main comme pour l'heuristique admissible de la capture au jeu de Go qui prend le nombre de libertés d'une chaîne comme nombre minimum de coups à jouer avant de la capturer. En revanche cela peut être intéressant pour engendrer les fonctions qui trouvent les seuls coups possibles susceptibles d'atteindre le but en un nombre de coups de suite donné. Au Phutball par exemple [22], il serait intéressant d'avoir des fonctions de sélection fine des coups possibles qui permettraient peut être de résoudre de nouvelle tailles de Phutball. Actuellement le jeu est résolu jusqu'à la taille 11x11 [33].

Chapitre 4

Recherche sélective

4.1 Introduction

Je présente dans ce chapitre les algorithmes de recherche sélective que j'ai développé suite à mes travaux sur la génération automatique de programmes de recherche. Ces algorithmes sont basés sur les menaces. Ils ont des liens avec les autres algorithmes de recherche présentés précédemment ou conjointement dans la littérature. Le dernier développement de cette famille d'algorithmes, les menaces généralisées, généralisent et ont de meilleures performances que les approches précédentes.

Les algorithmes basés sur les menaces sont efficaces dans les jeux instables : les jeux pour lesquels jouer quelques coups de suite de la même couleur permet de gagner. Les jeux concernés sont donc les sous jeux du jeu de Go comme la capture de pierres [85, 27, 25], la connexion de chaînes, les yeux et la vie des groupes, mais aussi le Go-Moku [1], les recherches de mats aux Echecs ou au Shogi, le jeu de Hex et bien d'autre jeux. Je présente deux jeux qui ont ces propriétés et pour lesquelles j'ai effectué de nombreuses expérimentations : AtariGo et le Football des philosophes (Phutball). J'ai également testé les programmes pour les jeux de la capture de pierres, de la connexion, des yeux et de la vie au jeu de Go ainsi qu'à Hex [44].

Il est très important dans les jeux complexes qui ont un grand nombre de coups possibles à chaque position d'être sélectif et de ne considérer qu'un sous ensemble des coups possibles. Lorsqu'on envisage tous les coups, on a une explosion combinatoire. Il faut donc sélectionner les coups à envisager et éliminer des coups a priori inutiles. Toutefois, lorsqu'on choisit d'ignorer des coups, on doit éviter deux revers :

- ne pas explorer des coups amis qui vont se réveler gagnants et donc sous évaluer une position

- ne pas explorer des coups ennemis qui nous font perdre, et croire qu'une position n'est pas perdante alors qu'elle l'est.

Nous présentons dans ce chapitre des algorithmes de recherche qui permettent non seulement de faire une selection sévère des coups à envisager mais aussi de donner des résultats plus fiables que les algorithmes de recherche classiques puisque ces résultats sont prouvés. Ces algorithmes qui sont tous basés sur la recherche abstraite de preuve, améliorent à la fois le temps de réponse des programmes de jeux et leur précision.

La deuxième section explique les jeux sur lesquels j'ai comparé différents algorithmes : AtariGo et le Football des philosophes. La troisième section rappelle quelques algorithmes à base de menaces et décrit entre autres la recherche abstraite de preuves [25] ainsi que l'élargissement itératif [27]. La quatrième section décrit la recherche abstraite graduelle de preuves [34, 33] qui permet une sélection plus fine des menaces que APS. La cinquième section montre l'utilité des abstractions pour accélerer la recherche. La sixième section définit les menaces généralisées. La septième section décrit la recherche avec menaces généralisées et montre qu'elle généralise les algorithmes précédents de recherche avec menaces. La huitième section donne des résultats expérimentaux. La neuvième section suggère des évolutions possibles de ces algorithmes. Vient enfin la conclusion.

4.2 Les jeux testés

4.2.1 L'AtariGo

AtariGo est utilisé pour apprendre une version simplifiée du jeu de Go aux apprentis joueurs de Go. On y joue sur une grille carrée. Le jeu peut être joué sur n'importe quelle taille de damier. En général on utilise de petits damiers pour que les parties se terminent rapidement. Les pédagogues choisissent aussi souvent de faire jouer les parties avec deux pierres blanches et deux pierres noires croisées au centre du damier. La position est alors plus instable. J'ai résolu le jeu pour le damier de taille 6x6 [34, 33, 32].

Les règles sont similaires à celles du jeu de Go : Noir commence, Noir et Blanc jouent chacun leur tour sur les intersections du damier, les chaînes de pierres sont les pierres de la même couleur qui sont reliées par les lignes du damier. Le nombre d'intersections vides adjacentes à la chaîne est le nombre de libertés de la chaîne. Une chaîne est capturée si elle n'a plus de libertés. Par exemple toutes les chaînes de la figure 4.1 ont deux libertés. Une chaîne qui n'a plus qu'une seule liberté peut être capturée par la couleur adverse en un coup, elle est en Atari, d'où le nom du jeu. Le but du jeu est d'être le premier joueur à capturer une chaîne adverse.

4.2.2 Le Phutball

Le Football des philosophes (ou Phutball) est décrit dans Winning Ways [4], un ouvrage sur la théorie combinatoire des jeux [40] appliquée à de nombreux jeux. Il a été surnommé Phutball par J. H. Conway. Les auteurs de Winning Ways pensent que ce jeu ne peut pas être totalement analysé par la théorie combinatoire des jeux.

Le Football des philosophes a été défini par Conway pour un damier 19x15. Les buts étant les lignes de longueur 15. Il est aussi joué sur des damiers de Go 19x19. Une pierre noire représente la balle et les pierres blanches représentent les joueurs de Football. Toutes les pièces sont communes aux deux joueurs, et les deux joueurs ont les mêmes coups légaux.

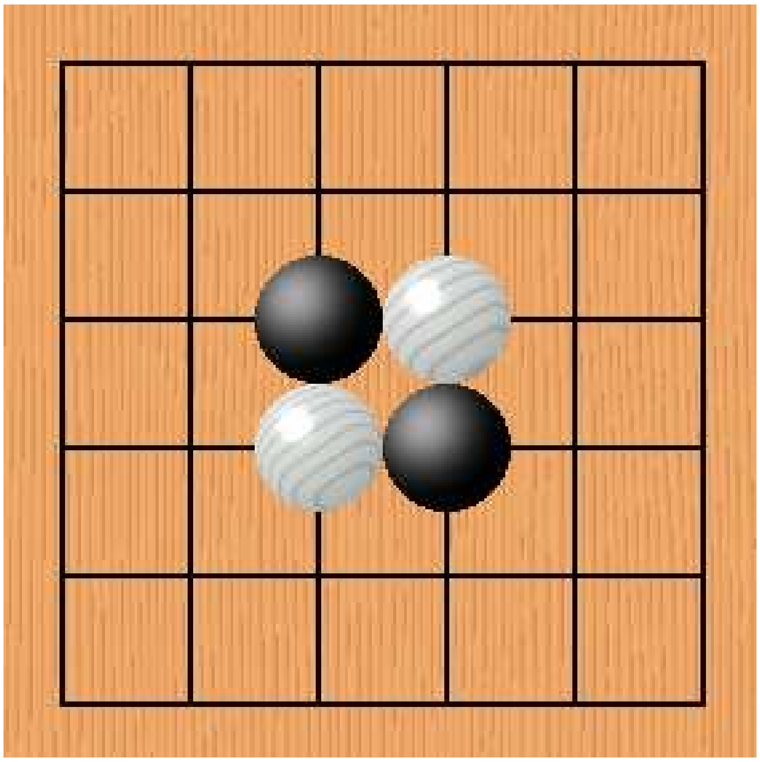

FIG. 4.1 – Le damier initial pour AtariGo 6x6.

La partie commence avec un damier vide, la balle est placée sur l'intersection centrale. Ensuite, à chaque coup, chaque joueur doit (i) soit poser une nouvelle pierre blanche sur une intersection vide (ii) soit faire sauter la balle par dessus des pierres blanches, en enlevant les pierres sautées.

Un saut peut être dans n'importe laquelle des 8 directions. On peut prendre une ligne de pierres en sautant par dessus. On peut effectuer plusieurs sauts dans le même coup. Le but du jeu est de faire parvenir la balle sur la première ligne du camp adverse, ou derrière cette première ligne.

La complexité du Phutball n'a pas encore été prouvée. Toutefois, le simple problème de déterminer si le joueur qui a la main peut gagner en un coup a été prouvé être NP-complet [43].

J'ai résolu le jeu pour la taille 11x11 à l'aide de la recherche abstraite graduelle de preuves [33].

4.3 Les algorithmes de recherche à base de menaces

Les algorithmes de recherche qui développent des arbres à l'aide de menaces donnent de bons résultats dans les jeux comme le Go-Moku [1], ou le jeu de la capture au jeu de Go [85, 27, 25]. Dans cette section je présente une vue d'ensemble des travaux effectués sur les algorithmes à base de menaces. Je commence avec l'Alpha-Béta optimisé que j'ai utilisé comme algorithme de recherche de base. Puis je présente la recherche dans l'espace des menaces (Threat Space Search) qui a été utilisé par V. Allis pour résoudre le Go-Moku. Je continue avec la recherche abstraite de preuves (ou APS) [25] qui a été testée sur la capture de pierres au jeu de Go. Puis j'expose l'élargissement itératif (IW) qui est une amélioration de APS qui est assez générale pour donner aussi de bons résultats avec d'autres algorithmes de recherche à base de menaces [32]. Viennent ensuite la recherche Lambda (Lambda Search ou LS) [85] qui traite de problèmes similaires à APS mais d'une façon légérement différente, puis la recherche abstraite graduelle de preuves (GAPS).

4.3.1 L'Alpha-Béta

Un bon historique de l'algorithme Alpha-Béta et de ses diverses optimisations a été écrit récemment par Yngvi Björnsson et Tony Marsland [59]. L'Alpha-Béta est à la base de tous les autres algorithmes de recherche décrits dans ce chapitre.

Les optimisations que j'ai utilisé sont les tables de transpositions contenant le score et le meilleur coup pour chaque entrée. L'utilisation de deux coups qui tuent après le coup de transposition s'il existe. L'heuristique de l'historique avec un poids de $2^{Profondeur}$. Une fonction d'évaluation incrémentale.

Ces optimisations sont semblables à celles utilisées par E. van der Werf [90] pour résoudre AtariGo 6x6 avec uniquement l'Alpha-Béta. Elles m'ont permis d'accélerer ma version initiale de la recherche abstraite graduelle de preuves qui résout AtariGo 6x6 [34] et d'en avoir une nouvelle version qui résout AtariGo 6x6 beaucoup plus vite ainsi que le Phutball 11x11 [33]. Ces optimisations ont aussi été utilisées pour la recherche avec menaces généralisées qui généralise les approches précédentes et qui est plus rapide [32].

4.3.2 La recherche dans l'espace des menaces

Le Go-Moku a été résolu par V. Allis en utilisant un algorithme sélectif de recherche basé sur les menaces. Dans cet algorithme, les menaces ont des noms qui correspondent à quelques patterns fixés : Four, Straight Four, Three, Five. APS, GAPS et Lambda Search sont des généralisations de la recherche dans l'espace des menaces (Threat Space Search), ils utilisent des arbres de recherche pour trouver les menaces plutôt que d'utiliser des patterns fixes.

4.3.3 La recherche abstraite de preuves

La recherche abstraite de preuve (APS ou Abstract Proof Search) est un algorithme qui démontre efficacement des théorèmes dans les jeux. Il permet des gains de temps appréciables sur l'Alpha-Béta avec toutes ses optimisations, de plus il renvoie toujours des résultats exacts, ce que ne garantit pas l'Alpha-Béta dans le cas des jeux où tous les coups possibles ne sont pas essayés à chaque noeud. La recherche abstraite de preuve permet de sélectionner les coups avec confiance.

La recherche abstraite de preuves utilise les jeux combinatoires à valeurs inconnues. La théorie des nombres de Conway [40], ainsi que la théorie combinatoire des jeux qui lui est associée [4] sont des outils très puissant d'analyse des jeux combinatoirement simples. Les jeux combinatoires ne peuvent être calculés que lorsqu'on connaît l'arbre entier du jeu à analyser. Dans les jeux complexes comme le jeu de Go, il est souvent impossible de calculer tout l'arbre de jeu, même pour des sous-jeux. En raison d'un manque de puissance de calcul, certaines des valeurs aux feuilles restent inconnues. J'ai étendu la théorie combinatoire des jeux aux arbres contenant des valeurs inconnues [9, 12]. Les jeux étendus utilisent trois lettres : g pour gagné, p pour perdu et i pour inconnu. Un jeu associé à un coup gagnant pour gauche et pour lequel on ne sait pas ce qui arrive pour droit est noté gi (abbréviation de $\{g|i\}$). Un jeu où Droit gagne s'il joue et où on ne sait pas ce qui se passe si gauche joue est noté ip (abbréviation de $\{i|p\}$).

La recherche abstraite de preuves est basée sur la vérification dynamique des jeux à valeurs inconnues. Alors qu'Introspect engendre les programmes équivalents.

Je définis maintenant de manière formelle les jeux qui sont utilisés pour la sélection des coups à essayer aux noeuds ET de l'Alpha-Béta : Soit P une position et C un coup. Soit Joue(P,C) une fonction qui renvoie la position après le coup C sur la position P. On utilise B pour Blanc et N pour Noir. On peut définir les jeux :

$gi_k(P, B)$ = B peut gagner en k coups blancs si B joue en premier sur la position P, et si les deux joueurs jouent parfaitement et chacun leur tour.

Formellement : $\exists$ un coup Blanc C tel que $\{P_1 = Joue(P, C)$ et $g_{k-1}(P_1, B)\}$

$ip_k(P, N) = gi_k(P, B)$.

$g_k(P, B)$ est vérifié si $\exists$ m $\{m \leq k, ip_m(P, N)$, $\exists$ un coup Noir C $\{P_1 = Joue(P, C)$ et $\exists$ o $\{o \leq k, gi_o(P1, B)\}\}\}$.

On peut donner une définition des jeux g qui est plus efficace à exécuter pour un programme en définissant :

$S_k(P, N)$ l'ensemble de tous les coups noirs qui empêchent Blanc de capturer en k coups ou moins dans la position P, lorsque N joue en premier et que $ip_k(P, N)$ est vérifié. Formellement : { coups Noirs C de P / { $gi_k(P, B)$, { $P_1 = Joue(P, C)$, $\exists$ o, $o \leq k$, not $(gi_o(P1, B))$ }}}

On a alors $g_k(P, B) = \exists$ m { $m \leq k$, $ip_m(P, B)$, $\exists$ C $\in S_m(P, N)$ { $P_1 = Joue(P, C)$, $\exists$ o $\{o \leq k, gi_o(P1, B)\}\}\}$.

Les définitions de jeux permettent par elles même de sélectionner de façon sûre un ensemble restreint de coups à envisager. On peut encore gagner en n'envisageant dans la construction des ensembles $S_k(P, N)$ qu'un petit nombre de coups utiles. En se basant sur la connaissance a priori de la profondeur de l'arbre associé au jeu (un jeu $ip_k(P, N)$ correspond à un arbre de profondeur 2k-1), on peut restreindre les coups utiles. Par exemple, au jeu de la prise, les seuls coups intéressants à envisager pour les jeux ip_1 sont les libertés de la chaîne à prendre et les libertés des chaînes adjacentes.

De même certains tests permettent d'éliminer rapidement les arbres qui ne donneront pas de résultats. Ainsi, toujours au jeu de la prise, le nombre minimum de coups pour prendre une chaîne est son nombre de libertés. Il est alors inutile d'essayer de prouver un jeu gi_3 pour une chaîne qui a strictement plus de trois libertés.

Sur des problème simples de jeu de la prise au jeu de Go, la recherche abstraite de preuves résout 90% de problèmes intermédiaires de capture [51] avec un temps total de recherche de 64 secondes quand on lui donne 100 000 noeuds et 10 secondes par recherche alors que l'Alpha-Béta avec les mêmes optimisations et les mêmes contraintes de temps et de noeuds ne résout que 79% des problèmes en 635 secondes. Des résultats plus complets peuvent être trouvés dans le papier correspondant [25].

La recherche abstraite de preuve a deux grands avantages sur l'Alpha-Béta classique : ses résultats sont fiables, et il sont calculés plus rapidement. Avec les mêmes contraintes de temps, elle résout plus de problèmes, et l'algorithme réagit mieux que l'Alpha-Béta à l'augmentation des capacités CPU. Cet algorithme marche pour un grand nombre de jeux.

4.3.4 L´élargissement itératif

L'élargissement itératif [24, 27] consiste à effectuer une recherche abstraite de preuve complète pour un ordre donné avant d'accroître l'ordre de la recherche. L'algorithme fonctionne bien sur le jeu de la capture de chaîne au jeu de Go. En pratique, on essaie une recherche abstraite de preuve d'ordre un, et si elle échoue a essayer une recherche d'ordre deux, et si elle échoue à essayer une recherche d'ordre trois, et ainsi de suite jusqu'à ce que le problème soit résolu ou jusqu'à ce que le temps alloué soit dépassé. Pour le jeu de la capture au jeu de Go, on obtient une accélération de deux.

D'une manière générale, la recherche sélective consiste à ne regarder qu'une partie des coups possibles. Un problème des algorithmes de type Alpha-Béta est l'ordonnancement des coups : on veut essayer en priorité les coups qui simplifient la situation et qui marchent souvent.

L'élargissement itératif consiste à ne considérer que les coups simples pour une première recherche, et s'ils ne marchent pas envisager des coups moins standards. Pour cela, on définit des ensembles de coups abstraits : S1, S2 ... Sn. L'élargissement itératif consiste à faire une recherche avec approfondissement itératif pour S1. Si elle échoue, recommencer avec S2, ainsi de suite jusqu'à Sn, ou jusqu'à ce que le temps imparti soit écoulé. Aux niveaux ET de l'arbre de preuve, il est naturel de définir les ensembles de coups abstraits à partir des définitions de jeux. Les coups S1 aux noeuds ET sont par exemple les coups associés aux jeux ip1 et ip2, alors que les coups de S2 seront les coups des jeux ip1, ip2 et ip3.

4.3.5 La recherche Lambda

La recherche Lambda (LS ou Lambda Search) [85] est un algorithme de recherche qui a des similarités étonnantes avec la recherche abstraite de preuves. Le papier a d'ailleurs été présenté à la même conférence dans la même session. J'avais diffusé mon papier six mois avant la conférence sur la mailing list computer Go. Cependant il y a des différences intéressantes entre Lambda Search et Abstract Proof Search qui m'ont permis de généraliser les deux approches et d'avoir au final un algorithme beaucoup plus efficace. La recherche Lambda peut être définie avec des arbres lambda et des coups lambda. Un arbre lambda d'ordre n est une arbre de recherche qui contient des coups lambda d'ordre n. Un coup lambda d'ordre n pour l'attaquant est un coup qui implique qu'il existe au moins un arbre lambda d'ordre strictement inférieur à n qui suit le coup. Un coup d'ordre n pour le défenseur est un coup qui implique qu'il n'y a aucun arbre gagnant d'ordre strictement inférieur à n après le coup.

La recherche abstraite de preuves impose des limites sur la profondeur et l'ordre des arbres développés à chaque noeud alors que la recherche Lambda n'impose des limites que sur l'ordre de ces arbres.

4.4 La recherche abstraite graduelle de preuves

La recherche abstraite graduelle de preuves (Gradual Abstract Proof Search ou GAPS) [34, 33] est basée sur la notion de jeu graduel. Un jeu graduel est défini comme une forme d'arbre. Les jeux graduels peuvent être plus profonds pour un plus petit ordre que les jeux utilisés dans APS. Ils peuvent aussi être plus contraints que les arbres-λ de LS. GAPS consiste à élargir itérativement la portée des jeux graduels, au lieu d´élargir les jeux uniquement à partir de leur profondeur comme dans [27]. C'est une généralisation de l´élargissement itératif.

GAPS prend à la fois de bonnes propriétés de APS et de LS. Au lieu de ne sélectionner les coups que sur la profondeur des arbres de menaces, comme dans APS, ou sur l'ordre comme dans LS, il sélectionne les coups sur une combinaison de ces deux critères. Il permet donc de mieux contrôler la recherche que APS et LS. Il peut aussi facilement utiliser les propriétés abstraites des jeux pour être très sélectif sur les coups à essayer afin de vérifier les jeux graduels.

La recherche abstraite graduelle de preuves a été testée sur un Athlon 1.7 GHz. Elle a permis de résoudre le Phutball 9x9 et 11x11. Le gain le plus court trouvé pour le damier 9x9 est donné dans la Figure 4.2. Les poses de pierres blanches sont notées 'p(Intersection)'. Les coups de balles sont notés 'b(Intersection)', Intersection est l'endroit où la balle finit son dernier saut. La solution trouvée pour le damier 9x9 est : p(F5), p(D6), p(H6), b(C7), p(D6), p(E5), p(G5), un jeu graduel de profondeur 4 est alors vérifié (par exemple p(F4), p(H2), p(B8), b(J1)), et le jeu est gagné pour Gauche. La solution est de longueur 11 et la profondeur de la recherche pour trouver cette solution est de 7. Les temps et nombre de noeuds utilisés pour chaque profondeur sont donnés dans la table 4.1, la profondeur 0 correspond au damier 9x9 vide. La solution prend 0.15 secondes.

Pour résoudre le damier 11x11, un seul coup de Gauche est essayé à la racine (poser une pierre en G7 dans la Figure 4.3). La recherche complète du damier 11x11 prend 188s. Elle essaie toutes les réponses de Droit au coup Gauche en G7. Le jeu est gagné pour Gauche. La défense la plus longue de Droit trouvée par GAPS est : p(G7), p(G5), p(J7), p(G3), p(K6), b(F2), p(G3), p(G2), p(J5), b(H2), b(F4), p(E3), p(G4), b(D2), p(E3), p(F4), p(H4), p(G3), p(K2), p(G5), p(L9). Un jeu graduel est alors vérifié (par exemple b(F8), p(G8), p(H8), b(L6)). La profondeur de la recherche est de 21, et la longueur de la solution est 25. Le début de la solution du damier 11x11 est donnée dans la Figure 4.3. La table 4.2 donne les temps et nombre de noeuds utilisés pour chercher la solution à une profondeur donnée après les coups p(G7) et p(G5).

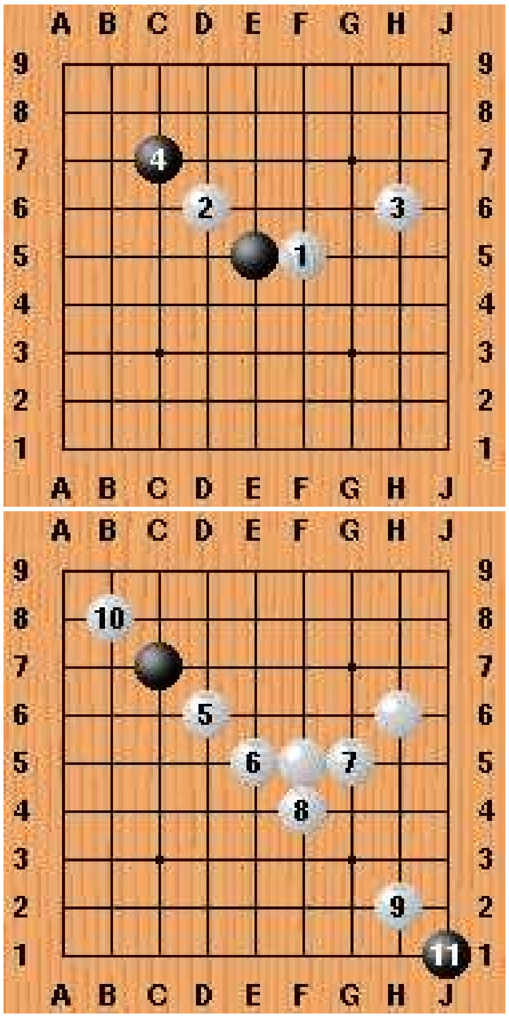

Fig. 4.2 – Une solution au Phutball 9x9.

Tab. 4.1 – Temps et nombre de noeuds utilisés pour la solution du Phutball 9x9.

Profondeur	*Temps*	*Noeuds*
1	0.00	11
2	0.01	49
3	0.01	131
4	0.01	277
5	0.03	475
6	0.03	876
7	0.06	1180

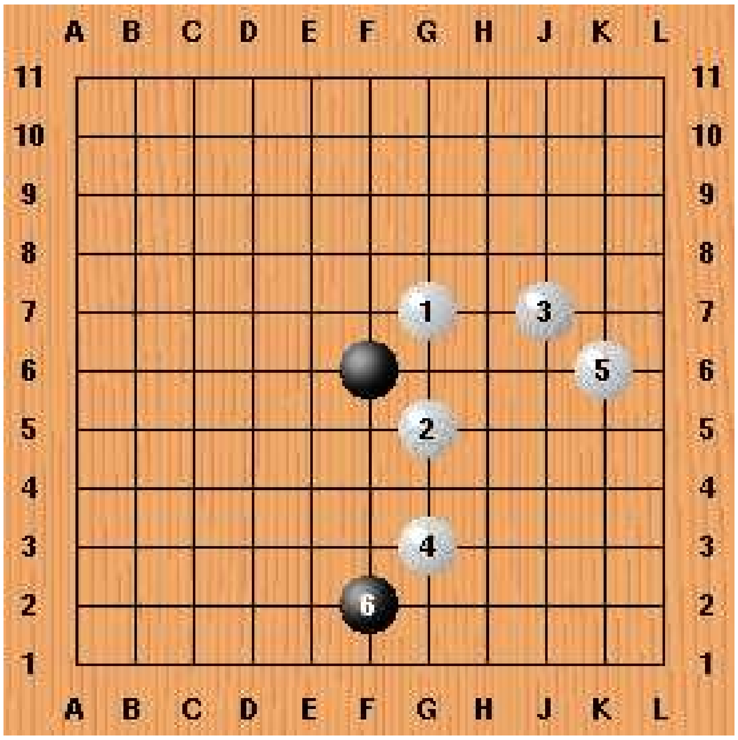

FIG. 4.3 – Le début de la solution du Phutball 11x11.

TAB. 4.2 – Recherche du Phutball 11x11 après p(G7), p(G5).

Profondeur	*Temps*	*Noeuds*	*Profondeur*	*Temps*	*Noeuds*
1	0.14	28	11	2.21	11352
2	0.14	48	12	1.13	10755
3	0.48	161	13	21.90	114150
4	0.32	324	14	1.68	15894
5	0.37	392	15	5.15	27660
6	0.37	1059	16	2.09	18104
7	0.46	1376	17	4.53	24379
8	0.48	3265	18	3.50	32832
9	1.41	7199	19	5.98	28942
10	0.79	6249			

TAB. 4.3 – Temps de recherche avec différents ordres d'abstractions.

	Temps (s)		
Profondeur	*Ordre*1	*Ordre*2	*Ordre*3
1	0.06	0.02	0.00
2	17.86	4.23	2.41
3	18.06	4.04	2.98
4	54.86	11.56	5.56
5	52.95	9.59	5.43
6	114.71	21.66	10.19
7	127.42	23.25	12.85
8	174.98	38.28	19.09
9	21.07	8.15	3.89
10	3.16	0.68	0.25
Total	585.13	121.46	62.65

4.5 Utilité des abstractions pour accélerer la recherche

L'utilisation d'heuristiques admissibles sur le nombre de coups permet d'accélerer fortement les algorithmes de recherche à base de menaces [30]. J'ai testé l'utilité de ces heuristiques sur l'Atarigo. Dans ce jeu, le nombre de libertés d'une chaîne est une heuristique admissible sur le nombre de coups qu'il est nécessaire de jouer à la suite pour la capturer. On peut donc arrêter immédiatement la recherche si l'heuristique admissible est supérieure à l'ordre de la menace à vérifier.

Les heuristiques admissibles peuvent être utiles dans de nombreux autres jeux. Par exemple, au Football des Philosophes (Phutball) [4] une heuristique admissible simple est la moitié de la longueur de la ligne la plus courte ne contenant que des intersections vides, plus un. Au jeu de la connexion au jeu de Go, une heuristique admissible sur le nombre de coups nécessaires pour connecter est la longueur du plus court chemin entre les deux chaînes à connecter. L'intérêt de l'incrémentalité est évident pour maintenir ces heuristiques. Il serait intéressant de mieux caractériser les propriétés d'un jeu ou d'un problème qui font que le maintient incrémental des heuristiques admissibles est plus ou moins bénéfique. Par exemple, au jeu de Go, mes programmes maintiennent incrémentalement les chaînes, leurs libertés, les chaines adjacentes.

De façon à estimer l'utilité des abstractions et des heuristiques admissibles pour les jeux, j'ai résolu l'AtariGo 6x6 avec GAPS en utilisant différent ordres d'abstraction [30]. L'ordre est le nombre de coups à la suite d'une même couleur qu'il est nécessaire de jouer pour arriver à une position gagnante. Les résultats sont dans la Table 4.3.

J'ai aussi appliqué GAPS au Phutball. On peut utiliser des connaissances abstraites au Phutball pour optimiser la recherche abstraite graduelle de preuves. Par exemple, un coup gagnant au Phutball ne peut être qu'un coup de balle. Donc lorsqu'on cherche à prouver qu'une position est gagnée en un seul coup, les seuls coups à essayer sont les coups de balle. Une autre conséquence est qu'un coup d'ordre deux (un coup qui permet un coup gagnant si on rejoue tout de suite après) ne peut être qu'un placement de balle. Ce ne peut pas être un coup de balle, puisque le coup gagnant est aussi un coup de balle, qui pourrait très bien être joué dans le même coup. La position serait dans ce cas gagnée en un seul coup. Une optimisation découlant de ces observations est que les seuls coups d'ordre deux que mon programme envisage sont les placements de pions sur les intersections directement atteignables en un coup de balle et les intersections voisines de ces intersections atteignables.

4.6 Les Menaces Généralisées

Je commence par définir les menaces généralisées dans une première partie, puis je traite des comparaisons entre menaces généralisées dans une deuxième partie. Je continue avec la définition d'un opérateur de composition des menaces généralisées. Cet opérateur permet de construire toutes les menaces généralisées pertinentes. Je finis par les algorithmes de vérification des menaces généralisées.

4.6.1 Définition des Menaces Généralisées

Un coup d'ordre n est un coup qui permet de gagner s'il est suivi par n-1 coups de suite du même joueur. Dans une menace, chaque coup est associé à un ordre. L'ordre d'un coup C est noté $\omega(C)$. Le dernier coup d'une menace est un coup gagnant, c'est un coup d'ordre un.

Un arbre de menaces est un arbre de recherche dans lequel les joueurs ont le droit de jouer plusieurs coups de suite. Les deux joueurs sont appelés Gauche et Droit. Une branche qui va vers la gauche représente des coups de Gauche, et une branche sur la droite des coups de Droit. Une arbre de menaces est un arbre binaire. L'algorithme MiniMax peut être représenté par un arbre de menace simple comme le premier arbre de la Figure 4.4 qui représente un arbre MiniMax de profondeur 7. Un MiniMax avec l'heuristique du coup nul, et un facteur de réduction de 2 pour cette heuristique, arrête de chercher lorsque le résultat d'un arbre de recherche de profondeur égale à la profondeur courante moins 2, et qui commence par un coup de la même couleur que le coup précédent, n'a pas un résultat meilleur que Béta pour un noeud Max, ou inférieur à Alpha pour un noeud Min. Le deuxième arbre de menaces de la Figure 4.4 correspond à un Null-Move Minimax de profondeur 5 avec un facteur de réduction de 2. La recherche avec coup nul améliore la rapidité de l'Alpha-Béta mais ne préserve pas la correction des résultats des recherches. Alors que la recherche avec menaces généralisées est beaucoup plus rapide, et préserve la correction des résultats de la recherche.

Un noeud d'ordre n dans un arbre de menaces est un noeud pour lequel le nombre de branches qui partent sur la gauche à la suite est de n. Par exemple, le noeud racine de la menace généralisés (1,0) de la Figure 4.5 est un noeud d'ordre un. La racine de la menace généralisée (6,3,2,0) est d'ordre 3.

Une menace généralisée est un ensemble d'arbres de menaces qui ont des propriétés spéciales. Elle est représentée par un vecteur d'entiers. Le premier élément du vecteur est le nombre de noeuds d'ordre un qui sont autorisés dans les arbres de menaces représentés par le vecteur. Le deuxième élément est le nombre de noeuds d'ordre 2 autorisés, et le $n^{\text{ième}}$ élément donne le nombre maximum de noeuds d'ordre

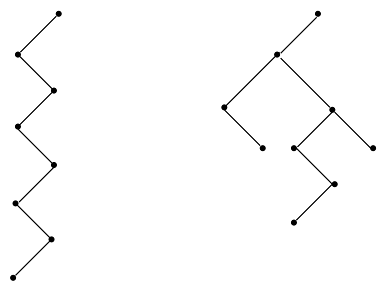

FIG. 4.4 – Arbres de menaces pour un MiniMax de profondeur 7 et un Null-Move Minimax de profondeur 5.

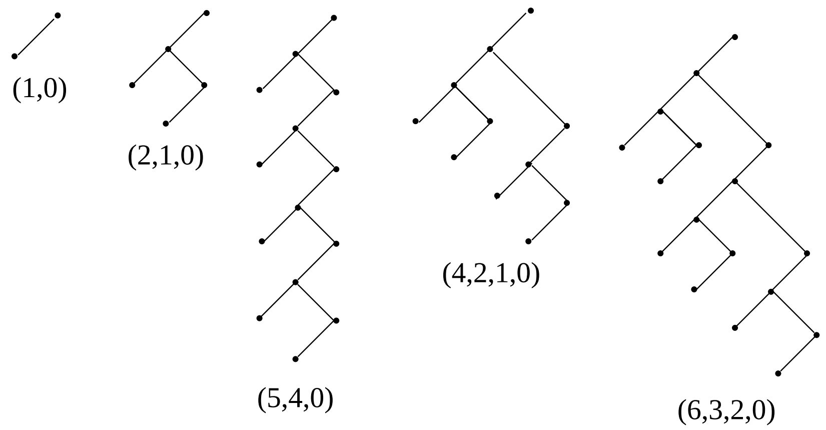

FIG. 4.5 – Des arbres de menaces correspondant à des menaces généralisées.

n autorisés lors de la vérification de la menace.

Une menace généralisée est définie par $g = (o_1, o_2, ..., o_n, 0)$ où o_i est le nombre maximum de noeuds d'ordre i qui peuvent être présents dans les arbres de menaces. Une menace généralisée se termine toujours par un zéro. Par exemple, vérifier qu'un coup gagnant existe est équivalent à vérifier la menace (1,0).

Pour qu'un arbre de menaces fasse partie d'une menace généralisée, il doit vérifier une propriété : pour chaque noeud de l'arbre duquel partent à la fois une branche gauche et une branche droite, le sous arbre gauche doit être inclus dans le sous arbre gauche qui suit la branche droite.

La Figure 4.5 donne des exemples d'arbres de menaces qui font partie de différentes menaces généralisées. Gauche essaie de gagner, et Droit tente de l'en empêcher. Les branches vers la gauche sont associées à des coups gagnants pour Gauche, et les branches vers la droite à l'ensemble complet des coups de Droit qui sont susceptibles d'empêcher Gauche de réaliser le sous-arbre gauche. Toutes les feuilles de l'arbre représentent des positions gagnantes pour Gauche. Pour que la menace soit vérifiée, Gauche doit avoir un coup gagnant à chaque noeud, et tous les coups de Droit doivent avoir été réfutés par Gauche.

Dans les arbre de menaces correspondant aux menaces généralisées, le nombre de feuilles est le nombre de menaces d'ordre un, puisque chaque feuille est une position gagnée pour Gauche. Le nombre de noeuds d'ordre 2 est le nombre de branches gauches qui sont suivies par un noeud d'ordre un pour Gauche.

Les menaces généralisées sont une généralisation des jeux graduels utilisés dans GAPS [34, 33]. Dans GAPS, la représentation des jeux graduels est moins générale. Une menace généralisée représente plusieurs arbres graduels. De plus, la programmation des menaces généralisées est plus simple la programmation des jeux graduels. Les menaces généralisées ont une définition simple et des propriétés sympathiques comme nous allons le voir dans les paragraphes qui suivent.

Chaque noeud d'ordre n+1 est suivi par au moins un noeud d'ordre n. Il est facile de voir que $\forall i : o_i < o_{i+1}$. Donc, dans un vecteur qui représente une menace, toutes les valeurs après un zéro sont aussi forcément à 0. C'est pourquoi seul le premier zéro est écrit dans la représentation vectorielle des menaces généralisées.

4.6.2 Comparaison de Menaces Généralisées

Soit g une menace généralisée, on note ω(g) l'indice du premier zéro du vecteur représentant g. Par exemple, $\omega(1{,}0) = 2$, $\omega(2{,}1{,}0) = 3$ et $\omega(6{,}3{,}2{,}0) = 4$. ω(g) est l'ordre maximum des noeuds de g plus un.

Soit $g_k = (o_{k_1}, o_{k_2}, ..., o_{k_n}, 0)$. On a $g_a \leq g_b$ si $\omega(g_a) \leq \omega(g_b)$ et $\forall i < \omega(g_b)$: $o_{a_i} \leq o_{b_i}$.

La Figure 4.6 donne des exemples de différentes menaces généralisées. Une flèche entre deux menaces signifie que la menace pointée est inférieure à la menace non pointée. Certaines menaces sont incomparables.

Quand une menace généralisée est plus grande qu'une autre, cela veut dire que tous les arbres de menaces qui peuvent être construits avec la plus petite peuvent aussi être construits avec la plus grande. Cet ordre partiel entre les menaces est particulièrement utile pour construire et pour vérifier les menaces, en particulier parce que la menace qui suit un coup de Droit est toujours plus grande que la menace Gauche que le coup Droit tente d'empêcher.

4.6.3 Composition de Menaces Généralisées

La menace de base est (1,0). Toutes les autres menaces peuvent être construites à partir de (1,0) en utilisant de l'opérateur de composition. Soit T l'opérateur qui compose deux menaces généralisées.

Soit g_l et g_r deux jeux tels que $g_l \leq g_r$. On peut définir l'opérateur T comme : $g_t = g_l$ T g_r et $\forall k \neq \omega(g_l) : o_{t_k} = o_{l_k} + o_{r_k}$ et pour $k = \omega(g_l) : o_{t_k} = o_{r_k} + 1$.

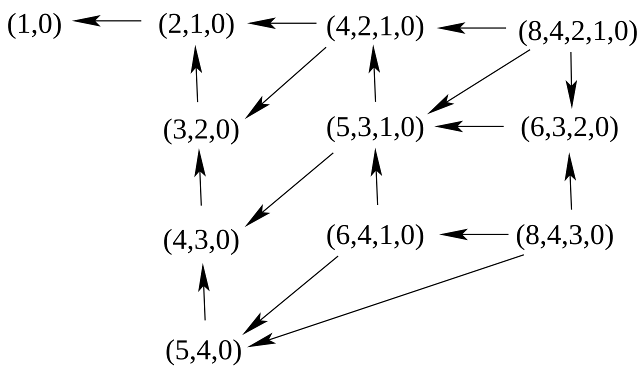

FIG. 4.6 – Ordre partiel entre les Menaces Généralisées

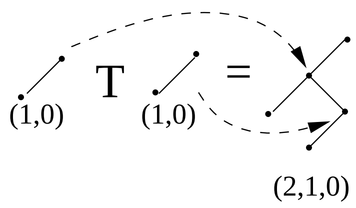

FIG. 4.7 – La composition des deux menaces les plus simples donne (2,1,0).

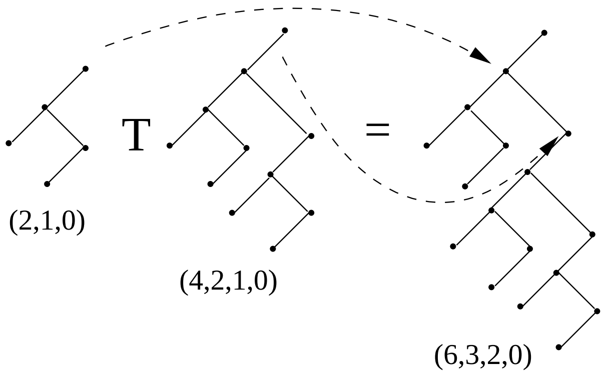

FIG. 4.8 – La composition de (2,1,0) et de (4,2,1,0) donne (6,3,2,0).

Par exemple, on a : (1,0) T (1,0) = (2,1,0) comme le montre graphiquement la Figure 4.7. Un autre exemple est donné dans la Figure 4.8 qui montre l'opération (2,1,0) T (4,2,1,0) = (6,3,2,0).

L'opérateur T assure par construction que pour tous les noeuds d'un arbre représentant une menace généralisée, si le noeud a une branche gauche et une branche droite, alors le sous arbre gauche est plus petit que le sous arbre gauche qui suit la branche droite du noeud. La branche droite représente l'ensemble des coups de Droit qui sont susceptibles d'empêcher Gauche de réaliser la menace représentée par le sous abre gauche. Comme Gauche aura plus de mal à gagner après un coup de Droit qu'avant, la branche droite doit être suivie par une menace gnénéralisée plus grande que la menace généralisée vérifiée par Gauche. Ceci de façon à pouvoir réfuter tous les coups de Droit. C'est la justification de la propriété sur les comparaisons de sous arbres gauches. L'opérateur T préserve cette propriété.

4.6.4 Vérification des Menaces Généralisées

Vérifier une menace généralisée consiste à vérifier que pour chaque branche gauche, il existe un coup gagnant pour Gauche prouvable avec un arbre inférieur ou égal au sous-arbre gauche. Mais aussi qu'il n'y a aucun coup Droit qui empêche Gauche de gagner, et ceci pour toutes les branches droites.

Il est possible de vérifier les menaces généralisées sans optimisations. Toutefois, j'utilise de nombreuses optimisations pour accélérer leur vérification. Ces optimisations sont décrites dans les paragraphes suivants.

Une optimisation pour la vérification des menaces généralisées est l'élargissement itératif sur l'ordre maximum autorisé pour les menaces, aux noeuds qui ont seulement une branche gauche. À ces noeuds, le programme commence par essayer un coup d'ordre un, s'il ne marche pas, il essaie alors un coup d'ordre deux, décrémente le nombre de coups d'ordre deux autorisés dans la menace courante, met à zéro tous les éléments du vecteur d'ordre supérieur à deux et essaie de vérifier ce vecteur. Si cette menace n'est pas vérifiée, il continue à augmenter l'ordre des menaces jusqu'à ce qu'une menace soit vérifiée ou jusqu'à atteindre l'ordre maximum.

Pour les noeuds qui contiennent à la fois une branche gauche et une branche droite, on peut utiliser une autre optimisation sympathique. On sait que la menace gauche est plus petite que la menace à gauche de la branche droite. Donc, comme on connaît la menace qui doit être vérifiée pour le noeud courant, on peut définir la menace maximum qui doit être vérifiée à gauche. La menace gauche est au plus le moitié de la menace courante. On divise donc par deux la menace courante pour avoir la menace maximum vérifiable à gauche. Diviser par deux une menace revient à diviser par deux tous les éléments de son vecteur. Par exemple, si le programme doit vérifier la menace (4,2,1,0) à un noeud qui a à la fois un sous arbre gauche et un sous arbre droit, il peut se limiter à vérifier la menace (2,1,0) pour son sous arbre gauche. Parce que le sous arbre droit est plus grand que le sous arbre gauche, et donc que le sous arbre gauche est au plus la moitié de l'arbre total (c'est pour cela que tous les entiers qui représente l'arbre total sont divisés par deux pour trouver le sous-arbre gauche maximum).

A chaque noeud de l'arbre, la menace vérifiée peut être inférieure à la menace maximale que le programme avait le droit de vérifier. Le programme mémorise toujours la menace qui a été réellement vérifiée. Pour un noeud qui contient à la fois un sous-arbre gauche et un sous-arbre droit, il calcule le sous-arbre droit maximal en soustrayant le sous-arbre gauche réellement vérifié de la menace totale. Par exemple si le programme doit vérifier (6,3,2,0), il calcule que la menace gauche maximale est (3,1,1,0). En réalité c'est la menace (1,0) qui est réellement vérifiée, la menace droite maximale est donc (6,3,2,0) - (1,0) = (5,3,2,0).

Une autre optimisation qui donne de très bons résultats et qui est décrite plus en détails dans une section précédente est l'utilisation de coups abstraits comme dans la recherche abstraite de preuves [25] de façon à limiter le nombre de coups à envisager. Par exemple, une menace d'ordre 2 ne peut jamais être vérifiée à AtariGo si toutes les chaînes de Droit ont strictement plus de 2 libertés. Dans ce cas particulier, mais aussi dans de nombreux autres cas, la recherche peut être arrêtée et renvoyer immédiatement un échec.

4.7 La recherche avec les Menaces Généralisées

La recherche avec menaces généralisées (Generalized Threats Search ou GTS) est basée sur les menaces généralisées. GTS est un généralisation des algorithmes de recherche avec menaces précédemment publiés. Elle est non seulement capable de modéliser les autres algorithmes existants basés sur les menaces. Mais elle résout les problèmes et les jeux plus rapidement que les autres algorithmes quand on l'utilise de façon appropriée. Je donne des résultats expérimentaux de comparaison des différents algorithmes pour AtariGo 6x6. GTS peut être utilisé pour résoudre des problèmes dans de nombreux jeux.

Cette section commence par expliquer comment l'Alpha-Béta est utilisé dans la recherche avec menaces généralisées. La deuxième section montre qu'on peut modéliser la recherche abstraite de preuves et l'élargissement itératif avec des menaces généralisées. La troisième section montre comment on peut modéliser la recherche λ. La dernière section donne des résultats expérimentaux.

4.7.1 Intégration à l'Alpha-Béta

Un Alpha-Béta optimisé est au coeur de GTS. Les Menaces Généralisées sont utilisées différement aux noeuds Max et aux noeuds Min. Droit est le joueur qui essaie de minimiser la fonction d'évaluation et d'empêcher Gauche de gagner. Gauche est donc le joueur Max, et Droit le joueur Min. Aux noeuds Max, les menaces généralisées sont utilisées pour trouver les coups de Gauche qui empêchent Gauche de perdre si Droit joue en premier dans une menace généralisée. Si aucune menace de ce type n'est détectée à un noeud Max, tous les coups pertinents de Gauche sont alors essayés. En revanche, aux noeuds Min, si aucune menace généralisée n'est vérifiée pour Gauche, le noeud est coupé, si une menace généralisée est vérifiée pour Gauche, tous les coups de Droit susceptibles de l'empêcher sont essayés.

Par exemple, le coup Blanc numéro 2 en E4 dans la Figure 4.9 est trouvé par un menace (4,3,0), la variation principale de la menace étant B(E4), W(D5), B(E5), W(D6), B(D6), W(C6), B(B6) qui capture la chaîne blanche). Une fois que cette menace est vérifiée, tous les coups blancs pertinents sont essayés, et après chacun de ces coups blancs, la même menace est vérifiée (dans ce cas une menace (4,3,0)). Les seuls coups blancs qui sont gardés sont les coups qui empêchent la menace d'être vérifiée.

Un exemple de sélection de coups pour Gauche est le coup numéro 5 en D2 de la Figure 4.9. C'est un coup forcé pour Gauche (qui ici est Noir). Si Gauche ne joue pas le coup en D2, Droit peut gagner avec une menace (3,2,0) dont la variation principale est W(D2), B(F3), W(F4), B(F2), W(F1) qui capture une chaîne noire.

4.7.2 Modélisation de la recherche abstraite de preuve et de l'élargissement itératif

On peut modéliser la recherche abstraite de preuves (APS) avec GTS. On a ip1 = (1,0), ip2 = (2,1,0), ip3 = (4,2,1,0), ip4 = (8,4,2,1,0), et ainsi de suite. Une recherche APS d'ordre un est une recherche GTS avec une menace (1,0). Une recherche APS d'ordre trois comme elle est décrite dans [25] est une recherche GTS avec une menace (4,2,1,0).

L'élargissement itératif [27] consiste à exécuter une recherche GTS (1,0), puis si elle échoue à effectuer une recherche GTS (2,1,0), puis si elle échoue à effectuer une recherche GTS (4,2,1,0), et ainsi de suite...

L'heuristique de l'élargissement itératif a été réutilisée avec succés pour optimiser la vérification des menaces généralisées.

4.7.3 Modélisation de la recherche λ

La recherche λ ou λ-search peut être modélisé avec les menaces généralisées. Par exemple, développer un arbre λ_1 est équivalent à vérifier si une menace $(\infty,\infty,0)$. Les différents arbres λ peuvent être modélisés comme suit : arbre $\lambda_1 = (\infty,\infty,0)$. arbre $\lambda_2 = (\infty,\infty,\infty,0)$. arbre $\lambda_3 = (\infty,\infty,\infty,\infty,0)$, arbre $\lambda_4 = (\infty,\infty,\infty,\infty,\infty,0)$.

TAB. 4.4 – Résolution de AtariGo 6x6 avec l'Alpha-Béta.

Profondeur	*Valeur*	*Coup*	*Temps*	*Noeuds*
1	1	D5	0.00	33
2	0	D5	0.00	143
3	1	D5	0.01	1234
4	0	C5	0.01	3177
5	1	C5	0.09	25662
6	0	C5	0.29	71265
7	1	C5	2.04	563k
8	0	C5	2.49	604k
9	1	C5	27.91	7442k
10	0	C5	44.04	10375k
11	1	C5	168.06	43034k
12	0	C5	303.21	69300k
13	1	C5	2094.46	518016k
14	500	C5	150.39	34178k
Total			2793.00	

TAB. 4.5 – Résolution de AtariGo 6x6 avec GAPS.

Profondeur	*Valeur*	*Coup*	*Temps*	*Noeuds*
1	1	D5	0.00	33
2	0	D5	1.84	47
3	1	D5	2.33	235
4	0	E3	4.72	325
5	1	E3	5.76	594
6	0	E3	11.52	861
7	1	E3	11.98	2557
8	1	E3	20.58	1982
9	2	E3	2.78	1838
10	500	E3	0.46	53
Total			61.97	

4.7.4 Résultats Expérimentaux

AtariGo a été résolu avec différents algorithmes sur un Celeron 600 MHz avec 256 MO de RAM sous Linux. Quatre algorithmes ont été testés.

L'Alpha-Béta résout AtariGo 6x6 à la profondeur 14 en 2793s, les résultats sont dans la Table 4.4. Toutes les optimisations décrites dans la section sur l'Alpha-Béta sont utilisées.

La recherche abstraite graduelle de preuves (GAPS) résout AtariGo 6x6 en 62s à profondeur 10 avec le jeu graduel ip4221, les résultats sont dans la Table 4.5. L'Alpha-Béta utilisé dans GAPS est le même que l'Alpha-Béta utilisé pour l'expérience de la Table 4.4. A chaque noeud de l'Alpha-Béta, tous les jeux graduels ip sont testés et l'ensemble des coups forcés est l'intersection de tous les ensembles de coups forcés renvoyés par les jeux graduels.

TAB. 4.6 – Résolution de AtariGo 6x6 avec la recherche Lambda.

Profondeur	Res	$Ordre1$	Res	$Ordre2$	Res	$Ordre3$
3	0	0.01	0	0.00	0	0.01
5	0	0.00	0	0.03	0	0.13
7	0	0.00	0	0.10	0	1.29
9	0	0.00	0	0.96	0	12.34
11	0	0.00	0	6.13	0	106.38
13	0	0.00	0	41.29	0	1521.08
15	0	0.00	0	108.10	500	1914.07
Total		0.01		156.61		3555.30

TAB. 4.7 – Résolution de AtariGo 6x6 avec les Menaces Généralisées.

Profondeur	R	$(1,0)$	R	$(2,1,0)$	R	$(5,4,0)$	R	$(4,2,1,0)$	R	$(6,3,2,0)$
1	1	0.00	1	0.00	1	0.00	1	0.00	1	0.00
2	0	0.00	0	0.00	0	0.02	0	0.01	0	0.14
3	1	0.00	1	0.00	1	0.02	1	0.12	2	0.23
4	-1	0.00	-1	0.01	0	0.05	0	0.07	0	0.30
5	0	0.00	0	0.00	1	0.02	1	0.17	1	0.56
6	-500	0.00	-500	0.00	-1	0.09	-1	0.17	0	1.00
7					0	0.11	0	0.35	1	1.56
8					-1	0.20	-1	0.69	1	4.69
9					0	0.06	0	0.51	2	1.27
10					-1	0.08	-1	0.14	500	0.08
11					0	0.04	0	0.13		
12					-1	0.08	-1	1.29		
13					0	0.04	-500	0.01		
14					-500	0.09				
Total		0.00		0.01		0.90		3.66		9.83

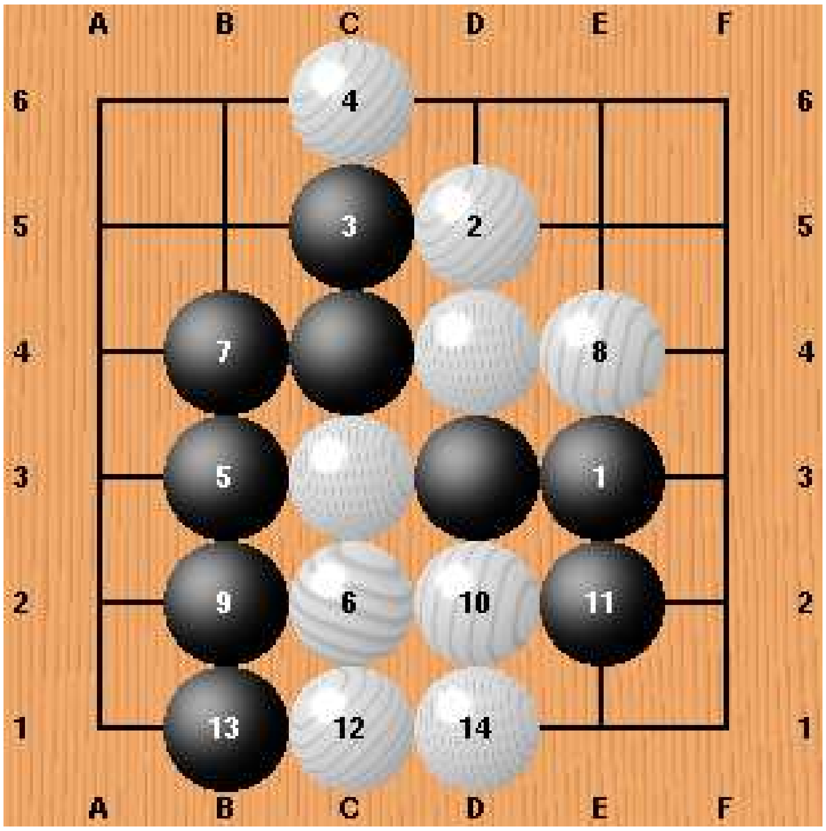

FIG. 4.9 – La solution de AtariGo 6x6 trouvée par GTS(6,3,2,0).

La recherche Lambda résout AtariGo 6x6 en 3555s à profondeur 15 et à l'ordre 3, les résultats sont dans la Table 4.6. Les optimisations de l'Alpha-Béta n'ont pas été utilisée pour LS, j'ai simplement réutilisé le code fourni par Thomas Thomsen sur sa page web.

La recherche avec menaces généralisées résout l'AtariGo 6x6 en 10s à profondeur 10 en utilisant la menace généralisée (6,3,2,0). Les résultats sont dans la Table 4.7. Les colonnes avec un R sont les résultats renvoyés par GTS. Les colonnes qui les suivent donnent les temps pris pour rechercher chaque profondeur. La solution trouvée pat GTS(6,3,2,0) est donnée dans la Figure 4.9.

4.8 Recherche avec dépendances

Un thème de recherche que je commence à explorer avec d'autres chercheurs actuellement est la recherche multi-buts. Lorsque les buts sont indépendants, cette recherche est triviale puisque on a juste à faire deux recherches indépendantes. Lorsque les buts sont très fortement dépendants, on ne peut espérer que peu de gain de performance d'une gestion élaborée des dépendances. Par contre lorsque les différents buts sont faiblement dépendants, on peut gagner énormément à bien gérer les dépendances et les indépendances entre les buts.

Les jeux pour lesquels la recherche avec dépendances a de très bonnes chances de marcher, sont les problèmes de transitivité de connexions au jeu de Go, mais aussi tous les problèmes de dépendances entre sous buts du jeu de Go comme par exemple un groupe qui est soit vivant soit sorti mais aucun des deux à la fois, ou la dépendance entre les yeux d'un groupe, ou la dépendance entre une connexion et un oeil. Au jeu de Hex, la gestion des dépendances des connexions peut aussi permettre des gains important de temps de recherche. Dans une problématique assez proche, Bernard Helmstetter développe actuellement un modèle de gestion des dépendances pour une réussite.

La recherche avec dépendance dans les jeux est naturellement basée sur la recherche avec menaces généralisées. Je la vois comme un extension de la recherche avec menaces généralisées aux problèmes avec plusieurs buts.

4.8.1 Recherche de connexions à Hex

Le meilleur programme de Hex est actuellement Hexy de V. Anshelevitch [3]. Il utilise une approche qui combine les connexions les plus simples pour en former de nouvelles plus larges. Je cherche plutôt à utiliser une approche par décomposition qui part de connexions difficiles pour les décomposer en recherches plus ou moins indépendantes sur des connexions plus faciles. Le maintien incrémental d'heuristiques admissibles come la distance entre les deux bords est important pour l'efficacité de la recherche avec dépendance [44].

4.8.2 Recherche de connexions non transitives au jeu de Go

Un problème actuellement non traité de manière générale par les programmes de Go est la non transitivité des connexions. Lorsque une chaîne A est connectée à une chaîne B, et que cette chaîne B est connectée à une autre chaîne C, la chaîne A n'est pas toujours connectée à la chaîne C. Toutefois, dans beaucoup de cas, calculer si la chaîne A et la chaîne C sont connectées est trop difficile directement, alors que calculer la conjonction des connexions de A et B, et de B et C peut être fait rapidement avec une recherche avec dépendances.

4.8.3 Résolution de réussites

Bernard Helmestetter travaille actuellement sur la gestion des dépendances de séquences de coups pour la résolution d'un jeu de carte de réussite. Il utilise une représentation abstraite du problème qui devrait permettre à son programme de n'explorer que certains noeuds du graphe de recherche au lieu d'explorer tout le graphe.

4.9 Développements possibles

GTS marche bien pour l'AtariGo, le Phutball, la connexion au Go, la connexion à Hex, la capture au jeu de Go, et la vie et la mort sur des problèmes ouverts au jeu de Go. Il a de bonnes chances de bien marcher pour d'autres jeux comme par exemple Lines of Action, la recherche de mats au Shogi et aux Échecs. Les prochains jeux qu'il pourrait résoudre sont par exemple l'AtariGo 8x8, le Phutball 13x13. La recherche avec dépendances pourrait peut-être permettre de résoudre Hex 7x7 avec la règle de swap.

4.9.1 Liens avec d'autres algorithmes

Il peut aussi être intéressant d'intégrer les nombres prouvants et déprouvants [2] dans GTS. L'utilisation de GTS pour la recherche avec une fonction d'évaluation entière reste encore à tester. GTS peut être utilisé de plusieurs façons avec une fonction d'évaluation entière. On peut par exemple fixer un seuil (la valeur d'un pion aux Echecs par exemple) et définir le gain comme les positions qui sont supérieures de ce seuil à l'évaluation de la racine. Une autre façon de l'utiliser serait d'utiliser une recherche dichotomique sur les valeurs possibles de la fonction d'évaluation comme dans la recherche avec fenêtre nulle [59] sauf qu'on peut utiliser une recherche GTS au lieu de la recherche Alpha-Béta pour déterminer si la valeur est supérieure ou inférieure à la fenêtre nulle.

4.9.2 Symétrisation de l'algorithme

Il est possible d'avoir des résultats bien encore meilleurs avec GTS si on peut symétriser l'algorithme. Dans certains jeux, il est facile de détecter le gain mais difficile de détecter que le jeu est perdu. Dans les jeux ou l'on peut facilement définir le gain et la perte, on peut utiliser les menaces généralisées aussi bien aux noeuds ET qu'aux noeuds OU. J'ai commencé à tester un GTS symétrique pour les sous-jeux du jeu de Go et les résultats sont prometteurs.

4.10 Conclusion

J'ai décrit plusieurs algorithmes de recherche sélective que j'ai programmé. Le dernier en date qui recouvre tous les autres, ainsi que les autres algorithmes du même type existant est la recherche avec menaces généralisées (GTS). GTS est non seulement plus général que les autres algorithmes, mais il est aussi plus rapide. Il permet de résoudre AtariGo 6x6 bien plus rapidement que tous les autres algorithmes, et il donne aussi de bons résultats sur d'autres problèmes. J'ai montré les extensions de GTS que j'envisage aux problèmes multi-buts et les extensions possibles de l'algorithme lui-même.

Chapitre 5

TAIL : Une librairie d'algorithmes génériques pour l'IA

TAIL est un acronyme pour Template Artificial Intelligence Library. C'est une librairie générique d'algorithmes et de structures de données en cours de développement. Son but est d'éviter de réimplémenter de nombreuses fois des algorithmes très similaires pour des applications assez semblables.

5.1 La programmation générique

La programmation générique vise à réutiliser des algorithmes et des structures de données automatiquement plutôt que de réécrire à chaque fois les mêmes algorithmes et les mêmes structures de données pour chaque nouveau problème.

Une des applications les plus connues de la programmation générique est la Standard Template Library (ou STL). C'est une librarie C++ qui contient des classes, des algorithmes et des itérateurs. Elle fournit de nombreux algorithmes et structures de données fréquement rencontrés. C'est une librairie générique basée sur les templates C++. On peut donc appliquer les algorithmes génériques à des classes et des objets que l'on définit, sans avoir à les réimplémenter.

5.2 Algorithmes génériques de recherche

J'ai pour l'instant écrit pour la TAIL deux algorithmes génériques que j'ai appliqué à plusieurs jeux. Ces deux algorithmes sont l'Alpha-Béta et la recherche avec menaces généralisées. Pour ces deux algorithmes j'ai du définir plusieurs classes elles aussi génériques. Ces classes génériques sont utilisées par exemple pour les tables de transposition, les ensembles de coups, les listes de coups ou les traces de recherche. De plus ces deux algorithmes utilisent des coups qui tuent, l'heuristique de l'historique et utilisent les mêmes interfaces avec les classes spécifiques aux jeux.

Le paramétrage des classes génériques pour les algorithmes de recherche sont les classes coup, damier et jeu. La classe jeu contient des noms de méthodes prédéfinis pour évaluer les positions, traiter les transpositions ou encore trouver des ensembles de coups à essayer aux noeuds min ou max.

Je construis actuellement des classes génériques de base pour les algorithmes génériques de recherche dont hériteront les autres algorithmes génériques de recherche. De même une classe générique pour les damiers peut être utile pour ne pas avoir à redéfinir toujours les mêmes propriétés. On peut ainsi envisager une classe générique pour les damier rectangulaires en forme de grille, et un autre classe générique de base pour les damiers hexagonaux. J'utilise déjà des classes génériques de base pour les jeux, ainsi par exemple, tous les sous jeux du jeu de Go hérite d'une classe générique 'GoGame' qui traite les kos.

L'utilisation d'algorithmes génériques de recherche permet non seulement d'éviter de reprogrammer et de réadapter les algorithmes, mais aussi d'éviter de se reposer à chaque fois des problèmes similaires de conception. Les méthodes à reprogrammer sont prédéfinies et aident à ne programmer que le strict nécessaire, en posant tout de suite les bonnes questions : qu'est ce qu'un coup ? Comment traiter les transpositions ? Comment évaluer une position ?

5.3 Travaux futurs

Je pense étendre la TAIL pour de nombreux algorithmes d'IA. Les premiers algorithmes concernés et probablement les plus faciles à rendre génériques sont A*, IDA*, IDA* avec bases de patterns. Une classe générique pour l'application d'algorithmes de Monte-Carlo semble aussi indiquées et devrait bien s'intégrer avec les classes génériques déjà définies pour les autres jeux. Je pense aussi développer des algorithmes génériques de résolution de contraintes et d'apprentissage par différences temporelles.

Plus près des algorithmes que j'ai déjà écrit, je compte rendre générique les algorithmes de recherche avec dépendances et d'analyse rétrograde que je suis en train d'étudier.

Chapitre 6

Programmation du jeu de Go

Après une présentation du jeu de Go, je reprend une petite histoire de la programmation des jeux. Je montre ensuite comment Golois résout les problèmes tactiques, puis j'esquisse sa fonction d'évaluation. Je donne ensuite des distinctions dans sa gestion de l'incrémentalité, puis je me fait l'avocat de l'utilisation de la théorie combinatoire des jeux pour le milieu de partie. Je parle ensuite du contrôle des paramètres de la recherche. Viennent ensuite les résultats du programme. Pour finir sur les travaux futurs.

6.1 Le jeu de Go

Le jeu de Go est un ancien jeu chinois. Il est très populaire en Corée, au Japon et en Chine. Les générations successives de joueurs de Go ont accumulé une expérience énorme du jeu tout au long de son histoire. Pour donner une idée de la somme des connaissances sur ce jeu, on peut citer les historiens chinois qui écrivent souvent que c'est Yao, un empereur semi-mythique du 23ème siècle avant J.-C. qui aurait inventé le jeu de Go pour instruire son fils Dan Zhu. Aujourd'hui on compte des dizaines de millions de joueurs de Go en extrême orient, et des centaines de joueurs professionnels. Le jeu de Go est un jeu à information parfaite, à somme nulle entre deux joueurs : Blanc et Noir. Un damier de Go (Goban) est une grille 19x19, qui est vide au début de la partie et que les joueurs remplissent tour à tour en posant une pierre sur une intersection. Le nombre moyen de coups possibles est de l'ordre de 250, et les parties peuvent facilement durer 200 coups. D'un point de vue programmation, les difficultés viennent à la fois du grand nombre de coups possibles, qui rend une approche combinatoire par force brute de type programme d'Echecs impossible. Mais elles viennent aussi de la complexité de l'évaluation d'une position, et de la définition ambiguë de la fin de la partie (en utilisant des règles non ambiguës, les parties dureraient plus de 350 coups). Les meilleurs programmes de Go utilisent des connaissances stratégiques et des recherches très sélectives orientées vers des sous-buts du jeu (capture de pierres, connexion, vie et mort des groupes). Les programmes de Go sont encore très faibles par rapport aux meilleurs joueurs humains. Le meilleur programme de Go a un niveau qui est approximativement de 9 kyus coréens (les kyus ont des valeurs différentes suivant les pays, et le kyu coréen est le plus fort, les kyus français sont aussi assez forts).

Les meilleurs joueurs de Go sont 9ème Dan et les débutants sont 20ème kyu. En théorie il y a donc une différence de 17 pierres entre les meilleurs joueurs et les meilleurs programmes : les programmes peuvent jouer 17 coups de suite au début de la partie et toujours se faire battre. En pratique, la différence est encore plus grande, et Many Faces of Go, un des meilleurs programmes s'est récemment fait battre à 29 pierres par Martin Mueller un 6ème Dan amateur.

Un des nombreux avantages de la programmation du jeu de Go d'un point de vue de l'Intelligence Artificielle est qu'on peut très facilement comparer deux programmes en les faisant jouer l'un contre l'autre. Et les programmeurs de Go ne s'en privent pas. De nombreux tournois de programmes sont régulièrement organisés.

6.2 Petite histoire de la programmation des jeux

L'histoire de la programmation des jeux est dominée par le jeu d'Échecs. La première est la plus connue des machine d'Échecs est le turque construit en 1769 par le baron Wolfgang von Kempelen. Malheureusement, c'était une supercherie, un maître d'Échecs était caché à l'intérieur. La première vraie machine d'Échecs fut la machine de Torres y Quevedo construite en 1890, elle est capable de jouer parfaitement les finales Roi et Tour contre Roi. Le premier programme d'Échecs fut écrit par Konrad Zuse entre 1942 et 1945 dans le langage Plankalkul qu'il avait lui-même développé [93]. Dand la même veine, Tihamer Nemes développa une machine pour jouer aux Échecs [68]. Mais ces approches basées sur des machines dédiées ne furent pas suivies par d'autres travaux. En 1947, Alan Turing spécifia le premier programme d'Échecs en insistant sur la notion de programme stocké [87]. En 1949, Claude Shannon décrit une façon de programmer les Échecs qu'il publiera l'année suivante [79], et un ordinateur fut programmé pour résoudre les mats en deux coups. Le papier de Shannon propose deux stratégies. La stratégie de type A qui consiste à essayer tous les coups jusqu'à une profondeur donnée, et la stratégie de type B qui consiste à privilégier certaines branches en utilisant de la connaissance pour couper les branches inintéressantes. Aujourd'hui encore, le débat n'est pas tranché entre les partisans des deux stratégies... La paternité de l'Alpha-Béta est controversée, Newell, Simon et Shaw le décrivent [69], mais John McCarthy soutient être la première personne à avoir formulé l'idée, Arthur Samuel affirme aussi avoir trouvé l'idée. En 1959, Arthur Samuel commence à faire des expériences sur l'apprentissage automatique pour améliorer son programme de Checkers [77].

Contrairement aux Échecs pour lesquels un parallèle a clairement été établi entre la puissance de calcul et le niveau de jeu [82], le niveau des programmes de Go n'est pas (encore) fortement corrélé à la puissance de calcul. La puissance de calcul aide, mais elle paraît moins importante que l'architecture du programme ou que les connaissances spécialisées qui y sont incluses.

Pour une histoire plus spécifique à la programmation du jeu de Go, on peut se reporter à [6].

6.3 Résolution de problèmes tactiques par Golois

Lorsqu'il résout des problèmes tactiques, Golois utilise un algorithme incrémental de calcul des chaînes et des libertés. Il utilise actuellement la recherche avec menaces généralisées pour résoudre les problèmes tactiques de capture, de connexion, et de vie et mort. Pour accélerer la recherche des problèmes de vie et de mort, il utilise des heuristiques similaires aux heuristiques statiques des programmes Handtalk et Go Intellect [38]. Il traite correctement les kos et les sekis avec le même algorithme générique pour tous les problèmes de Go. Une des connaissances qui lui permet d'accélérer le plus efficacement la recherche de vie et de mort est son algorithme de détection des groupes qui ne peuvent plus faire deux yeux. Il évite ainsi de faire des recheches dans de très grands sous arbres avec un algorithme très simple.

6.4 Evaluation d'une position

L'évaluation d'une position de Go se base sur le résultat des recherches tactiques et sur le calcul de l'influence. Golois construit des groupes en utilisant les connexions et les prises de chaînes. Il évalue ensuite la force de ses groupes et leur voisinage. Il calcule la vie des groupes qui sont incertains. Il fait passer les groupes morts à l'adversaire, et boucle sur les calculs de forces de groupes et d'inversion de couleur tant qu'il y a des groupes morts.

Le calcul de l'influence est basé sur une fonction très simple de distance. Toutes les intersections qui ont un chemin inférieur à six, ne passant que par des intersections vides, qui les relie à un groupe vivant sont influencées par ce groupe. Quand une intersection est influencée par des groupes de couleurs différentes, on ne considère son appartenance que pour le groupe le plus proche, et on ne la considère pas influencée par le groupe de couleur opposée le plus lointain.

L'évaluation d'une position est l'influence amie moins l'influence ennemie. On peut se rapporter à [6] pour une description plus poussée d'une fonction d'évaluation au Go.

6.5 Incrementalité au niveau stratégique

Lorsuq'il joue un coup au niveau stratégique, Golois ne recalcule pas tous les calculs tactiques. Il ne recalcule que ceux dont la trace contient le coup joué. Les traces de calculs tactiques sont fournies par l'algorithme générique de recherche avec menaces généralisées qui les calcule de toutes façons. Ainsi, on évite de recalculer les objets tactiques dont la trace n'est pas modifiée. Il est intéressant de mémoriser deux traces pour un calcul : la trace de succès et la trace d'échec. La trace de succès répertorie toutes les intersections qui supportent un calcul qui aboutit à prouver que des coups atteignent le but. Pour la construire, on ne garde aux noeuds OU de l'arbre que la traces du coup qui atteint le but, par contre on agrége toutes les trace des fils aux noeuds ET. Tous les coups en dehors de la trace de succès ne peuvent pas remettre en cause le résultat du calcul. La trace d'échec quant à elle agrége toutes les traces aux noeuds OU, mais ne retient que les traces du coup qui empèche aux noeuds ET. La trace d'échec permet de ne pas refaire un calcul qui a échoué si on sait qu'il va échouer à nouveau.

6.6 Décomposition en sous jeux

D'une manière générale les programmes qui jouent au Go ne semblent pas utiliser les informations assez simples à obtenir que sont les valeurs des menaces associées aux coups. La théorie combinatoire des jeux traite de façon très fine ce problèmes [4]. Toutefois son application a jusqu'ici été réservée à la fin de partie. J'ai montré que l'on peut obtenir des gains de l'ordre de 15% de points en plus à la fin d'une partie si on utilise cette information assez peu couteuse [31]. En pratique, il semble que les stratégies HotStrat et Thermostrat soient très proches de l'optimalité pour ces jeux combinatoires très simples.

6.7 Contrôle des ressources allouées aux recherches

Effectuer un contrôle sur le temps de résolution alloué aux recherches tactiques permet d'évaluer des positions beaucoup plus rapidement. Golois utilise plusieurs mécanismes de contrôle de ses recherches.

Le mécanisme le plus simple est la gestion du temps global alloué pour la partie. Il peut fonctionner en plusieurs modes. Le mode non limité par le temps, le mode semi-rapide, le mode rapide et le mode blitz. Le temps généralement alloué aux programmes de Go dans les compétitions entre programmes est de une heure. Il commence pendant les dix premières minutes par jouer dans le mode non limité, puis il passe au mode semi rapide jusqu'à 45 minutes, il passe alors en mode rapide jusqu'à atteindre 55 minutes, puis ils utilise le mode blitz pour ne pas perdre au temps.

En mode blitz, il n'utilise que 10 noeuds pour ses calculs sur la vie et la mort, et limite ses recherches sur les connexions et les prises à 50 noeuds. Il ne fait une recherche globale qu'à profondeur un avec ces recherches très rapides. En mode rapide, il utilise 100 noeuds pour la vie et la mort et se limite aux prises simples. En mode semi rapide, il envisage les prises et les connexions plus évoluées. En mode rapide, il se permet un nombre de noeuds plus élevé.

Une gestion plus évoluée du contrôle des recherches et qui permet d'évaluer les positions jusqu'à dix fois plus vite est de contrôler l'allocation de ressources allouées aux calculs tactiques en fonction de leur importance stratégique. Il est par exemple inutile de calculer une connexion difficile est très longue à calculer si les deux chaînes à connecter font déjà partie du même groupe par une suite de connexions simples. De façon similaire, il est inutile de calculer la vie d'un grand groupe qui est assez loin de faire deux yeux, si ce grand groupe a beaucoup d'espace autour de lui et un territoire assez important. Le problème est que pour évaluer l'importance stratégique d'un calcul tactique, il faut déjà avoir construit une représentation stratégique, et que cette représentation stratégique se construit à partir des calculs tactiques. La solution pour laquelle j'ai opté est de construire une première représentation stratégique basée sur des calculs tactiques simples, puis de l'utiliser pour décider d'allouer plus de temps aux calculs tactiques importants stratégiquement. Par exemple, on utilise la représentation stratégique simple pour décider de calculer la vie d'un groupe s'il est entouré est n'a pas beaucoup de territoire. On peut aussi l'utiliser pour décider de donner plus de ressources au calcul d'une connexion ou d'une prise cruciale parce qu'elle permet de sauver un groupe important qui n'est pas vivant sans possibilités de fuir. Une nouvelle représentation stratégique est alors construite qui prend en compte les résultats des calculs tactiques important stratégiquement. C'est à partir de cette représentation que Golois décide des coups importants qui doivent être essayés au niveau global. On pourrait envisager un système plus souple dans lequel il y aurait plus d'aller retour entre le niveau stratégique et le niveau tactique. On pourrai même envisager de coupler ces importances tactiques avec l'importance globale de l'évaluation de la position dans l'arbre de recherche global.

Un autre exemple du contrôle des recherches tactiques est l'opportunité de la recherche de la transitivité de la connexion. La recherche de non transitivité peut être très coûteuse si on essaie aveuglément de trouver toutes les non transitivités possibles. Elle est toutefois nécessaire pour construire les groupes sans agréger ensemble des chaînes qui peuvent être coupées. Là aussi Golois utilise des informations stratégiques et une évaluation de la complexité du calcul pour décider du calcul de la non transitivité.

Ces travaux sont très liés aux travaux sur le contrôle de la recherche [76, 74, 71] et il y a encore de nombreuses pistes à explorer sur l'interaction du contrôle de la recherche et des problématiques des jeux et de la résolution de problèmes d'une manière plus générale.

6.8 Résultats

Le monde de la programmation du Go est dominé par les diverses compétitions internationales entre programmes [37]. On peut regretter qu'actuellement, les articles scientifiques sur la programmation du Go font trop peu souvent référence à des tests sur des bases de problèmes reconnues comme pertinentes. On voit trop souvent des chercheurs prétendre que leur algorithme résout beaucoup de problèmes parce qu'il résout deux ou trois exemples qu'ils ont eux-mêmes choisi. Le développement d'une base de test qui permettrait d'évaluer les forces et les faiblesses des programmes est important et améliorerait beaucoup la qualité scientifique des programmes de Go [28]. Le protocole que j'ai commencé a utiliser pour construire cette base de test est le Go Text Protocol (GTP) développé par l'équipe de gnugo [7]. J'ai par ailleurs écrit un programme C pour connecter les programmes pouvant utiliser ce protocole aux serveurs internet de Go sur lesquels jouent des centaines de joueurs à toutes les heures. La meilleur évaluation d'un programme est probablement son niveau sur ces serveurs. Toutefois les bases de tests permettent de pointer les problèmes qui doivent être résolus par tous les programmes sans que le programmeur n'ait à recommencer la lourde tâche de reconstruire sa base de test avec des exemples similaires.

6.9 Travaux Futurs

Les aspects les plus prometteurs sur lesquels travailler pour Golois, sont le test des menaces généralisées génériques pour résoudre les semeais non complétement entourées. Cette approche a déjà été couronnée de succès pour la résolution des problèmes de vie et de mort non entourés, il y a de bonnes chances qu'elle fonctionne aussi pour les semeais.

L'étude plus approfondie du controle des recherches, et son développement vers des outils plus généraux de contrôle de la recherche semble une voie très prometteuse.

L'extension des bases de tests pour permettre une meilleure communication scientifique entre les chercheurs sur la programmation du Go, et pour permettre de mieux comprendre les avantages et les inconvénients des différents algorithmes utilisés pour résoudre les sous problèmes du jeu de Go.

Il paraît aussi très tentant de réutiliser les représentations de haut niveau et les résultats de calculs tactiques complexes de Golois comme entrées de haut niveau pour l'apprentissage avec différences temporelles. Jusqu'ici les programmes basés sur les différences temporelles marchent étonnament bien avec des entrées assez basiques [46]. On peut espérer qu'améliorer les entrées donnerait des résultats encore meilleurs.

Une autre étude que j'envisage est l'analyse de l'évolution du niveau de Golois en fonctions de divers paramètres comme le temps alloué à chaque type de recherche tactique, la complexité des menaces généralisées utilisées pour les recherches tactiques, le taux d'erreur accepté pour les recherche tactiques, la largeur et la profondeur de la recherche de quiescence globale... Ce qui permettrait de mieux comprendre les facteurs influençant le niveau des programmes, un peu comme l'article de Ken Thompson qui a marqué la programmation des Echecs en mettant l'accent sur la corrélation du niveau et de la puissance de calcul [82].

Chapitre 7

Travaux futurs

7.1 Métaprogrammation

Il serait intéressant de modifier Introspect pour qu'il soit capable de s'observer en train de fonctionner grace à un méta-interpréteur, puis qu'il tire parti de cette observation pour tenter des expériences sur le code du système observé plutôt que de fonctionner avec des métarègles de contrôle de l'apprentissage fixes. Par exemple, de modifier l'ordre des testsparties déclaratives ou de tenter des pliages et des dépliages de fonctions cruciales.

On peut envisager de faire le lien entre métaprogrammation logique et programmation générique, en permettant aux algorithmes génériques de se spécialiser automatiquement sur un problème. Par exemple en pré-compilant et en otant les branches inutiles d'un programme directement à la spécialisation. Par exemple un algorithme générique de plus court chemin appliqué à une grille devrait pouvoir détecter avec une méthode générale que les intersections à trois pas de l'intersection de départ ne peuvent en aucun cas être l'intersection de départ.

J'envisage aussi de modifier Introspect pour lui faire effectuer une recherche sélective dans l'espace des algorithmes. Il pourrait ainsi transformer des algorithmes simples mais relativement inefficaces en algorithmes qui ont de meilleures propriétés de mémoire et/ou de temps de réponse. Actuellement, il n'existe pas d'outils théoriques assez puissants pour analyser la complexité d'un algorithme non trivial. Pour contourner ce problème d'analyse des programmes transformés, je pense diriger la transformation par des tests par les temps d'exécution des programmes transformés à l'aide de bases de tests, et par des méta-opérateurs qui donneraient des patterns de suites de transformations parfois utiles, et des patterns de transformation à éviter. Les opérateurs de base applicables à un programme sont le pliage, le dépliage et les définitions de nouveaux prédicats.

7.2 Analyse rétrograde

Pour améliorer mon système d'analyse rétrograde, je pense utiliser les attributs décrits dans [38] par exemple ou des variantes comme attributs externes des règles engendrées par analyse rétrograde de façon à générer moins de règles avec une plus grande utilité.

L'analyse rétrograde de patterns avec conditions extérieures est très probablement utile dans d'autres domaines que le jeu de Go. Il reste à mieux caractériser les problèmes qui bénéficieraient de cette approche.

7.3 Recherche sélective

Une voie de recherche fructueuse concerne la quantification de l'utilité de l'incrémentalité pour différents attributs dans différents jeux. Il serait intéressant d'essayer d'en tirer des prévisions sur l'utilité de l'incrémentalité en fonction de certaines caractéristiques et si possible sur la génération automatique d'algorithmes de mise à jour incrémentale des attributs à partir de leur définition non incrémentale. Une première étape consiste à évaluer empiriquement l'utilité de l'incrémentalité pour les jeux actuellement implémentés : incrémentalité des distances a Hex, incrémentalité des distances des bords à la balle au Phutball, incrémentalité des libertés et des chaînes adjacentes au Go et à AtariGo, etc...

Une autre voie de recherche que nous explorons activement en ce moment est l'étude des algorithmes de recherche sur des combinaisons de buts.

D'un point de vue plus général, il serait bon de mesurer l'utilité des principes généraux que j'ai établis sur les algorithmes de jeux comme l'élargissement itératif pour d'autre problèmes d'IA comme par exemple les problèmes de satisfaction de contraintes ou de planification.

7.4 TAIL

Je souhaite étendre la TAIL pour de nombreux algorithmes classiques d'IA. Par exemple pour l'apprentissage par renforcement avec des réseaux de neurones et des différences temporelles, ou pour les méthodes de Monte-Carlo ou encore pour les algorithmes de satisfaction de contraintes comme le Forward Checking. Mais aussi pour généraliser les différents algorithmes que j'ai développé comme l'analyse rétrograde simple, ou l'analyse rétrograde de patterns avec conditions externes.

Il reste à quantifier le gain de temps et de lignes de programmes qu'on obtient en utilisant la TAIL pour écrire un programme pour un nouveau jeu, par rapport au temps de développement d'un algortihme sans la TAIL.

7.5 Programmation des jeux

Je pense tenter une classification des jeux en fonction des algorithmes qui permettent de les résoudre. Par exemple les algorithmes de recherche sélective que j'ai développés ne peuvent pas être appliqués tels quels au jeu de dames, car même en jouant plusieurs coups de suite, le jeu n'est pas terminé.

Il serait très utile pour la communauté des programmeurs de Go d'étendre la base de tests de problèmes de Go actuelle de Golois afin d'en faire un base de tests de référence pour les programmes de Go.

Pour les jeux, nous envisageons aussi d'utiliser et de développer les algorithmes de recherche avec dépendance créés par Bernard Helmstetter pour les appliquer à la recherche sélective avec objectifs multiples. Par exemple, pour faire des recherches sélective sur la transitivité de la connexion à Hex ou au Go.

Une autre voie de recherche très tentante et qui aurait probablement une utilité économique non négligeable est d'explorer l'utilité des algorithmes de recherches pour les jeux de stratégie temps réél ou les jeux de simulation. Actuellement l'IA de ces jeux est assez sommaire et se résume souvent à utiliser l'algorithme A* et des règles réactives pour les différents agents. On peut facilement imaginer que des algorithmes sélectifs de recherche pourraient être avantageusement utilisés pour améliorer le comportement des agents.

Dans la lignée des test effectués par K. Thompson sur l'évolution du niveau des programmes d'Echecs en fonction de la puissance de calcul [82], j'envisage de tester l'évolution des performances d'un programme de Go en fonctions des paramètre tactiques et stratégiques afin d'observer comment un programme basé sur la recherche sélective réagit à l'accroissement de la puissance de calcul. Les paramètres qu'on peut faire jouer sont par exemple les temps maximum alloués à chaque recherche tactique, leur menace maximale associée, la profondeur et la largeur de la recherche globale.

Un autre paramètre potentiellement intéressant est le taux d'erreur des recherches tactiques. Comme le montre un article récent de Thomas Wolf [75], on peut avoir un gain exponentiel en temps de calcul avec une perte linéaire en taux d'erreur, il serait aussi intéressant de mesurer le niveau d'un programme en fonction du taux d'erreur fixé. Admettre un taux d'erreur de 20% pourrait permettre de résoudre les problèmes cinq fois plus vite. Pour un temps de recherche fixé ou un nombre de noeuds fixé, on peut alors résoudre plus de problèmes, mais en commettant des erreurs. Une façon simple de mener cete expérience sans avoir des heuristiques élaborées, mais qui permettrait de mesurer l'intérêt de telles heuristiques serait d'introduire artificiellement un taux d'erreur de 20% et de donner cinq fois plus de temps au programme, on pourrait alors estimer la différence de niveau que pourrait apporter les heuristiques avant de les programmer.

7.6 Conclusion

D'une manière plus générale et sur le plus long terme, j'aimerais établir une typologie des problèmes de recherche en fonction de caractéristiques qui permettrait à un programme d'adapter automatiquement son algorithme de recherche au problème. Pour les jeux, un début de classification est par exemple de dissocier les jeux pour lesquels on peut en général gagner si on joue quelques coups de suite de la même couleur, des jeux pour lequel le gain est la plupart du temps très lointain. La recherche avec menaces généralisées marche mieux que l'Alpha-Béta sur les premiers, alors que l'Alpha-Béta marche pour l'instant mieux sur les jeux du second type.

Une autre grande question est la meilleure compréhension des liens entre mémoire et temps de calcul. J'ai montré que plusieurs méthodes permettent avec des efficacités variées d'échanger de la mémoire (des règles, des programmes ou des données engendrées) et du temps de calcul. Il reste à mieux comprendre quelles sont les interactions entre la représentation des programmes engendrés, les caractéristiques des problèmes traités et l'efficacité de leur résolution.

Chapitre 8

Conclusion

J'ai essayé de montrer quelques interactions que j'ai explorées entre recherche arborescente et connaissances. Il me reste à développer de nombreuses pistes. Notament la représentation des connaissances pour l'analyse rétrograde de règles, le contrôle de la transformation de programmes par le temps d'exécution et/ou d'autres méthodes, l'amélioration de la recherche avec menaces généralisées ainsi que son intégration à la recherche avec dépendances, ou encore le développement d'une librairie d'algorithmes génériques d'Intelligence Artificielle et l'intégration de tous ces outils dans Golois, mon programme qui joue au Go.

Bibliographie

[1] L. Victor Allis, H. Jaap van den Herik, and M. P. H. Huntjens. Go-Moku Solved by New Search Techniques. *Computational Intelligence*, 12 :7–23, 1996.

[2] L.V. Allis, M. van der Meulen, and H. J. Herik. Proof-number search. *Artificial Intelligence*, 66(1) :91–124, 1994.

[3] Vadim V. Anshelevich. A hierarchical approach to computer hex. *Artificial Intelligence*, 134(1-2) :101–120, 2002.

[4] E. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways*. Academic Press, 1982.

[5] B. Bouzy. Go patterns generated by retrograde analysis. In *Computer Olympiad Workshop*, Maastricht, 2001.

[6] B. Bouzy and T. Cazenave. Computer Go : An AI-Oriented Survey. *Artificial Intelligence*, 132(1) :39–103, October 2001.

[7] D. Bump, G. Farneback, et al. http ://www.gnu.org/software/gnugo/. web page, Free Software Fondation, 2002.

[8] T. Cazenave. Apprentissage de le résolution de problèmes de vie et de mort au jeu de go. Rapport de dea, Université Paris 6, 1993.

[9] T. Cazenave. Management of uncertainty in combinatorial game theory. Technical report, Université Paris 6, 1995.

[10] T. Cazenave. Automatic Acquisition of Tactical Go Rules. In *Game Programming Workshop in Japan '96*, pages 10–19, Kanagawa, Japan, 1996.

[11] T. Cazenave. Automatic ordering of predicates by metarules. In *5th International Workshop on Metaprogramming and Metareasoning in Logic*, Bonn, 1996.

[12] T. Cazenave. Système d'Apprentissage Par Auto-Observation. Application au jeu de Go. Phd thesis, Université Paris 6, December 1996.

[13] T. Cazenave. Automatically Improving Agents Behaviors in an Urban Simulation. In *Second International Conference of the Journal of Industrial Engineering and Applications*, San Diego, USA, 1997.

[14] T. Cazenave. Learning to Manage a Firm. In *Second International Conference of the Journal of Industrial Engineering and Applications*, San Diego, USA, 1997.

[15] T. Cazenave. Metaprogramming Forced Moves. In *ECAI 1998*, pages 645–649, Brighton, UK, 1998.

[16] T. Cazenave. Program generation for firms simulations in competitive environments. In *Strategic Valuation of Firms*, Universite Paris 2 Assas, 1998.

[17] T. Cazenave. Speedup Mechanisms For Large Learning Systems. In *IPMU 1998*, Paris, France, 1998.

[18] T. Cazenave. Synthesis of an efficient tactical theorem prover for the game of Go. *ACM Computing Surveys*, 30(3es) :18, 1998.

[19] T. Cazenave. Generation of Patterns With External Conditions for the Game of Go. In *Advance in Computer Chess 9 Conference*, Paderborn, 1999.

[20] T. Cazenave. Metaprogramming domain specific metaprograms. In *Reflection 99*, volume 1616 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 1999.

[21] T. Cazenave. Specialization of admissible path-finding heuristics. Technical report, Labo IA, Université Paris 8, 1999.

[22] T. Cazenave. Un tournoi de programmes de phutball. In *Actes du Colloque de Berder 1999*, 1999.

[23] T. Cazenave. Discovering search algorithms with program transformation. Technical report, Labo IA, Université Paris 8, December 2000.

[24] T. Cazenave. Iterative Widening. In *Proceedings of Workshop of the 2000 Computer Olympiad*, London, 2000.

[25] T. Cazenave. Abstract Proof Search. In T. Anthony Marsland and Ian Frank, editors, *Computers and Games 2000*, volume 2063 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2001.

[26] T. Cazenave. Generation of Patterns With External Conditions for the Game of Go. In H.J. van den Herik and B. Monien, editors, *Advance in Computer Games 9*, pages 275–293. Universiteit Maastricht, Maastricht, 2001.

[27] T. Cazenave. Iterative Widening. In *Proceedings of IJCAI-01, Vol. 1*, pages 523–528, Seattle, 2001.

[28] T. Cazenave. A problem library for computer Go. In Holger H. Hoos and Thomas Stuetzle, editors, *IJCAI-01 Workshop on Empirical Methods in Artificial Intelligence*, Seattle, USA, 2001.

[29] T. Cazenave. Retrograde analysis of patterns versus metaprogramming. In J. Kacprzyk, N. Baba, and L. C. Jain, editors, *Computational Intelligence in Games*, pages 57–74. Physica Verlag Rudolf Liebing KG, Vienna, Austria, 2001.

[30] T. Cazenave. Admissible moves in two-player games. In *SARA 2002*, volume 2371 of *Lecture Notes in Computer Science*, pages 52–63, Kananaskis, Alberta, Canada, 2002. Springer.

[31] T. Cazenave. Comparative evaluation of strategies based on the value of direct threats. In *Board Games in Academia VI*, Barcelona, Spain, 2002.

[32] T. Cazenave. A Generalized Threats Search Algorithm. In *Computers and Games 2002*, Lecture Notes in Computer Science, Edmonton, Alberta, Canada, 2002. Springer.

[33] T. Cazenave. Gradual abstract proof search. *ICGA Journal*, 25(1) :3–15, 2002.

[34] T. Cazenave. La recherche abstraite graduelle de preuves. In *Proceedings of RFIA-02*, pages 615–623, Angers, France, 2002.

[35] T. Cazenave. Metarules to Improve Tactical Go Knowledge. In *Joint Conference on Information Sciences*, Research Triangle Park, NC, USA, 2002.

[36] T. Cazenave. Metarules to improve tactical go knowledge. *Information Sciences*, to appear, 2002.

[37] T. Cazenave. A propos des compétitions de programmes de jeu de go. *Revue d'Intelligence Artificielle*, 16(3) :383–390, 2002.

[38] K. Chen and Z. Chen. Static analysis of life and death in the game of Go. *Information Sciences*, 121 :113–134, 1999.

[39] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings POPL'93*, pages 493–501. ACM, January 1993.

[40] J. H. Conway. *On Numbers and Games.* Academic Press, London/New-York, 1976.

[41] J. C. Culberson and J. Schaeffer. Searching with pattern databases. In *Canadian Conference on AI*, pages 402–416, 1996.

[42] J. C. Culberson and J. Schaeffer. Pattern Databases. *Computational Intelligence*, 4(14) :318–334, 1998.

[43] E. D. Demaine, M. L. Demaine, and D. Eppstein. Phutball endgames are hard. In Richard J. Nowakowski, editor, *More Games of No Chance*, MSRI Publications. Cambridge Univ. Press, 2002.

[44] T. Djearamane. Algorithmes pour Hex. Mémoire de maîtrise, Université Paris 8, Septembre 2002.

[45] D. Dyer. An eye shape library for computer Go. web page, Andromeda, http ://www.andromeda.com/people/ddyer/go/shape-library.html, 2002.

[46] M. Enzenberger. The integration of a priori knowledge into a go playing neural network. Technical report, 1996.

[47] J. Fürnkranz. Machine learning in game playing : A survey. In J. Fürnkranz and M. Kubat, editors, *Machines that Learn to Play Games.* Nova Science Publishers, 2001.

[48] R. Gasser. Endgame Database Compression for Humans and Machines. In H.J. van den Herik and L.V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3 : the third computer olympiad*, pages 180–191. Ellis Horwood, Chichester, England, 1991.

[49] T. Ishida. Optimizing rules in production system programs. In *AAAI-88*, pages 699–704, 1988.

[50] A. Junghanns and J. Schaeffer. Single-agent search in the presence of deadlock. In *AAAI-98*, pages 419–424, 1998.

[51] Y. Kano. *Graded Go Problems for Beginners, Vol 2.* The Nihon Ki-in, 1985.

[52] R. E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI-97*, pages 700–705, 1997.

[53] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2) :9–22, 2002.

[54] J. Laird, P. Rosenbloom, and A. Newell. Chunking in soar : An anatomy of a general learning mechanism. *Machine Learning*, 1(1), 1986.

[55] P. Laird. Dynamic optimization. In *ICML-92*, pages 263–272, 1992.

[56] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde-analysis problems using a network of workstations. In H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 135–162. University of Limburg, Maastricht, The Netherlands, 1994.

[57] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11 :217–242, 1991.

[58] S. Markovitch and P. D. Scott. Information filtering : Selection mechanisms in learning systems. *Machine Learning*, 10 :113–151, 1993.

[59] T. A. Marsland and Y. Björnsson. From minimax to manhattan. In H.J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 5–17. University of Maastricht and Shizuoka University, 2000.

[60] S. Minton. Constraint-based generalization - learning game-playing plans from single examples. In *AAAI-84*, pages 251–254, Los Altos, 1984. William Kaufmann.

[61] S. Minton. Learning effective search control knowledge : An explanation-based approach. Phd thesis, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[62] S. Minton. Quantitative Results Concerning the Utility of Explanarion Based Learning. *Artificial Intelligence*, 42(2-3) :363–392, 1990.

[63] S. Minton. Is there any need for domain-dependent control information : A reply. In *AAAI-96*, 1996.

[64] S. Minton, J. Carbonell, C. Knoblock, D. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning : A problem solving perspective. *Artificial Intelligence*, 40, 1989.

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

[66] T.M. Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzioni, M. Ringuette, and J.C. Schlimmer. Theo : A framework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence*, pages chapter 12, 323–355. Erlbaum, 1991.

[67] F. Napoleoni. Analyse rétrograde et classification inductive de bases de données de fins de parties pour le jeu de dames international. Rapport de dea, Université Paris 8, Saint Denis, 1999.

[68] T. Nemes. The chess-playing machine. Acta technica, Hungarian Academy of Sciences, Budapest, 1951.

[69] A. Newell, J.C. Shaw, and H.A. Simon. Chess playing programs and the problem of complexity. *IBM J. of Res. and Dev.*, 4(2) :320–335, 1958. (Also in Computers and Thought, E. Feigenbaum and J. Feldman (eds.), McGraw-Hill, 1963, 39-70).

[70] J. Nunn. Extracting information from endgame databases. *ICCA Journal*, 16(4) :191–200, 1993.

[71] T. Pannerec. Un système général avec un contrôle de la résolution à base de métaconnaissances pour des problèmes d'affectation optimale. Phd thesis, Université Paris 6, December 2002.

[72] A. Pettorossi and M. Proietti. A comparative revisitation of some program transformation techniques. In *Partial Evaluation, International Seminar*, number 1110 in LNCS, pages 355–385, Dagstuhl Castle, Germany, 1996. Springer.

[73] J. Pitrat. Realization of a program learning to find combinations at chess. In J. C. Simon, editor, *Computer Oriented Learning Processes*, number 14 in Series E : Applied Science, Noordhoff, Leyden, 1976. NATO Advanced Study Institutes Series.

[74] J. Pitrat. *Métaconnaissance - Futur de l'Intelligence Artificielle*. Hermès, Paris, 1990.

[75] M. Pratola and T. Wolf. Optimizing gotools' search heuristics using genetic algorithms. *ICGA*, 25(2), 2002. To appear.

[76] Stuart J. Russell and Eric Wefald. Principles of metareasoning. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *KR'89 : Principles of Knowledge Representation and Reasoning*, pages 400–411. Morgan Kaufmann, San Mateo, California, 1989.

[77] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. of Res. and Dev.*, pages 210–229, 1959. (Also in Computers and Thought, E. Feigenbaum and J. Feldman (eds.), McGraw-Hill, 1963, 71-105).

[78] J. Schaeffer. *One Jump Ahead : Challenging Human Supremacy at Checkers.* Springer-Verlag, New York, NY, 1997.

[79] C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41 :256–275, 1950.

[80] L. Stiller. Multilinear algebra and chess endgames. In R.J. Nowakowski, editor, *Games of No Chance*, volume 29. MSRI Publications, Cambridge, MA, 1996.

[81] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proc. 2nd Intl. Logic Programming Conf.*, Uppsala Univ., 1984.

[82] K. Thompson. Computer chess strength. In M. Clarke, editor, *Advances in Computer Chess 3*, pages 55–56. Pergamon Press, 1982.

[83] K. Thompson. Retrograde analysis of certain endgames. *ICCA Journal*, 9(3) :131–139, 1986.

[84] K. Thompson. 6-piece endgames. *ICCA Journal*, 19(4) :215–226, 1996.

[85] T. Thomsen. Lambda-search in game trees - with application to Go. *ICGA Journal*, 23(4) :203–217, 2000.

[86] T. Thomsen. Lambda-search in game trees - with application to Go. In T. Anthony Marsland and I. Frank, editors, *Computers and Games*, volume 2063 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2002.

[87] A. M. Turing. Digital computers applied to games. In B.V. Bowden, editor, *Faster Than Thought*, pages 286–297. Pitman, London, 1953.

[88] P. E. Utgoff and D. J. Stracuzzi. Many-layered learning. *Neural Computation*, to appear, 2002.

[89] H.J. van den Herik and I. S. Herschberg. The construction of an omniscient endgame database. *ICCA Journal*, 8(2) :6–87, 1985.

[90] E.C.D. van der Werf, J.W.H.M. Uiterwijk, and H.J. van den Herik. Solving ponnuki-go on small boards. In J.W.H.M. Uiterwijk, editor, *The 7th Computer Olympiad Computer-Games Workshop Proceedings*, pages 5–11, Maastricht, The Netherlands, 2002. IKAT, Department of Computer Science, Universiteit Maastricht.

[91] F. van Harmelen and A. Bundy. Explanation based generalisation = partial evaluation. *Artificial Intelligence*, 36 :401–412, 1988.

[92] J.-C. Weill. How hard is the correct coding of an easy endgame. In J. W. H. M. Uiterwijk H.J. van den Herik, I.S. Herschberg, editor, *Advances in Computer Chess VII*, pages 163–175. University of Limburg, 1994.

[93] K. Zuse. Chess programs, in the plankalkuel. Technical Report 106, Gesellschaft fuer Mathematik und Datenverarbeitung, Bonn, 1976.