L'intelligence artificielle pour le jeu de Go

Pour le TER de Mr pompidor :
Guillaume Tisserant
Guillaume Maurin
Matthieu Laclau

Pour le projet d'algorithme de Mr Koriche :
Guillaume Tisserant
Guillaume Maurin
Jérome Dorado
Basile Fabre

Pour le projet de C de Mme Laurent :
Rémi Le Brouster

2010

Table des matières

Chapitre 1

Introduction

Sommaire

"S'il y a une vie intelligente sur Mars, ses habitants doivent avoir découvert le jeu de go"

E. Lasker, champion du monde d'échecs

1.1 Remerciements

Devant l'ampleur de la tâche à laquelle nous nous sommes attelés, toute aide et soutien, tout particulèrement de la part des personnes qui ont cru en nous, ont été une bouffée d'air frais sans laquelle le projet n'aurait pas pu aller aussi loin. Nous tenons donc à remercier chaleureusement :

- Pierre Pompidor, enseignant encadrant du TER pour avoir accépté de nous suivre sur ce projet et pour son aide précieuse pendant son déroulement.
- Frederic Koriche, enseignant d'algorithmique de l'IA qui a accepté de nous laisser créer un module pour notre programme pour le projet de sa matière, et pour ses cours qui nous ont aidés à la conception du présent projet.
- Anne Laurent, enseignante à Polytech', pour avoir laissé Rémi Le Brouster intégrer notre groupe comme membre du projet de TER.
- L'ensemble des enseignants en informatique de l'Université Montpellier 2 dont la majorité des cours a été utile pour la réalisation de notre projet.
- Sikusiku, développeur japonais de Mugo, l'interface graphique que nous avons réutilisé, pour avoir créé seul un programme libre et puissant, et pour nous avoir cités sur son blog.
- Solène Planas pour son soutien moral et la réalisation graphique des schémas.
- Lucas Bottelin et Florian Vial pour leurs relectures du rapport et leur rôle de béta-testeur.
- Guillaume Douron pour son aide dans le remplissage du dictionnaire de josekis et son rôle de béta-testeur.

1.2 Présentation du rapport

1.2.1 Un mot d'introduction

Nous allons vous présenter l'aboutissement de notre travail sur l'Intelligence Artificielle au jeu de Go. Ce projet a été réalisé dans le cadre de trois Unités d'Enseignement différentes.
Il s'agit tout d'abord du sujet de TER de Master 1 pour Matthieu Laclau, Guillaume Maurin et Guillaume Tisserant. Une partie algorithmique de l'Intelligence Artificielle est venue se greffer au projet, avec Bazile Fabre, Jérome Dorado, Guillaume Maurin, Guillaume Tisserant pour le projet demandé dans l'UE du même nom.
Enfin, un étudiant de Polytech', Rémi Le Brouster nous a rejoint sur le TER dans le cadre d'un de ses projets.

Bien qu'un sujet alliant l'informatique et le jeu de Go puisse paraître rebutant pour les novices dans l'une ou l'autre des disciplines, nous avons voulu faire un rapport clair et compréhensible, fournissant des explications sur les bases du jeu de Go en parallèle de notre travail dans le domaine de l'Intelligence Artificielle.
Pour faciliter la lisibilité, le rapport possède un glossaire dans lequel tout le vocabulaire technique se référant au Go ou à l'informatique sera expliqué. Nous espérons qu'il vous sera agréable de le parcourir.

1.2.2 Le projet, concrètement, c'est quoi ?

Ce projet a été initié par Guillaume Tisserant, joueur de Go depuis 8 ans. Dans le monde des joueurs de Go, les ordinateurs ont toujours été considérés comme très mauvais face aux humains. De plus, les seuls algorithmes qui obtiennent des résultats à peu près corrects sont des monstres de calculs "bruts" testant toutes les fins de parties sur des petits plateaux, sans aucune analyse du jeu, ni l'ombre de ce que l'on pourrait imaginer être de l'"intelligence". Vouloir réaliser une intelligence artificielle de Go basée sur des principes cognitifs est un véritable défi, mais aussi un rêve pour beaucoup de gens qui s'intéressent au sujet. C'est pourquoi ce projet a une valeur particulière à nos yeux, et même si le temps imparti ne nous aura pas permis de révolutionner le domaine, nous sommes fiers de vous présenter notre travail.

Le résultat concret de ce projet est un programme capable de jouer correctement des débuts de parties sur un plateau en 19*19, et de gérer toute une partie en 9*9. Mais surtout, nous avons créé les outils pour améliorer son niveau de jeu, et ce potentiellement sans aucune limite.

Le projet ne sera pas abandonné après ce TER, le programme continuera à être amélioré et jouera de mieux en mieux dans l'avenir.

1.2.3 Le projet en chiffres

- Environ 6000 lignes de code originales.
- Plus de 500 mises à jour de notre SVN, disponible à l'adresse http ://code.google.com/p/machiabot/
- Des centaines d'heures de travail, dont une bonne partie de corrections de bugs.
- Seulement 4 mois de temps imparti


1.3 Le groupe et ses objectifs

1.3.1 Objectifs

Nous étions tous attirés par l'Intelligence Artificielle et nous voulions un projet capable d'en aborder les différents aspects. Le jeu de Go, par ses difficultés stratégiques, tactiques et combinatoires, mais aussi par toutes les subtilités qu'il offre, est un très bon terrain d'entraînement pour tester le fonctionnement des différentes composantes de l'Intelligence Artificielle.

Notre objectif concret était de faire une intelligence artificielle de Go la plus humaine possible. En effet, la plupart des programmes qui jouent actuellement au Go ont des styles caractéristiques qui permettent des les distinguer facilement des êtres humains. Cela dénote un certain échec par rapport au but premier d'une intelligence artificielle qui est d'imiter l'Intelligence Humaine. De plus, nous voulions un programme modulable et évolutif.

Notre deuxième objectif était plus théorique : nous voulions comprendre la façon dont les intelligences artificielles actuelles jouent au Go, pourquoi elles continuent de perdre face aux humains, et comment elles progressent. C'est pourquoi, à chaque fois que l'on parlera dans ce rapport des algorithmes que nous avons implémentés, nous les comparerons aux idées développées dans les programmes codés par nos prédécesseurs.

1.3.2 Orientation du travail

Notre projet est constitué de plusieurs modules fonctionnant ensemble qui nous ont permis de voir plusieurs facettes de l'Intelligence Artificielle, et de mettre à contribution nos connaissances informatiques sur un projet de grande envergure. Le projet de TER, dirigé par M Pompidor, nous a permis de développer un outil d'ingénierie cognitive, intégré dans un programme capable de jouer au Go. Le projet d'algorithme de l'Intelligence Artificielle dirigé par M Koriche nous a permis de construire un algorithme UCT, permettant d'intégrer de la combinatoire à notre programme. le projet de C dirigé par Mme Laurent nous a permis d'agrémenter le système expert de fonctions avancées. Ce projet aura été pour nous à la croisée des disciplines car nous nous sommes servis des connaissances de multiples matières pour en développer tous ses modules. Il contient en effet de la programmation orientée objet (avec de la généricité paramétrique), des algorithmes sur les graphes (recherche dans un arbre pour le dictionnaire), de la représentation des connaissances (système expert), du réseau (cloud computing, jeu sur serveur distant), du système (thread et mutex), de la programmation par aspect (gestionnaire de traces du système expert), et englobe la plupart des concepts que nous avons vu dans les Unités d'Enseignement d'Intelligence Artificielle.

1.3.3 Répartition des tâches

Le projet de faire une intelligence artificielle de Go étant extrêmement vaste, nous avons réparti les nombreuses tâches entre les différents membres selon leurs compétences et leur implication dans le projet :

- Guillaume Tisserant, dans le cadre de son TER et de son projet d'algorithme de l'Intelligence Artificielle, a co-dirigé le projet, travaillé sur la création des outils de base, du système de dictionnaires, et du système expert pour l'intelligence artificielle. Il a conçu le système de communication avec l'interface graphique.


Il a supervisé la création de l'algorithme UCT, et de son extension RAVE. Il a dirigé la rédaction du rapport.

- Guillaume Maurin, dans le cadre de son TER et de son projet d'algorithme de l'Intelligence Artificielle, a co-dirigé le projet, travaillé sur la création des outils de base et du système de dictionnaires pour l'intelligence artificielle. Il a supervisé la création du système expert, et a conçu le système multi-threading de l'algorithme UTC. Il a contribué à la rédaction du rapport.
- Matthieu Laclau, dans le cadre de son TER, a travaillé sur une solution pour comprendre les décisions de l'intelligence artificielle, et pouvoir les améliorer.
- Jérome Dorado dans le cadre de son projet d'algorithmique de l'Intelligence Artificielle, a travaillé sur la mise en place du cloud computing pour UTC.
- Basile Fabre, dans le cadre de son projet d'algorithmique de l'Intelligence Artificielle, a travaillé sur la composante RAVE d'UTC,et sur la mise en place du cloud computing pour UTC.
- Remi Le brouster, pour son projet sur le langage C à Polytech' a participé à la conception d'un système expert, et a codé de nombreuses fonctions qui lui étaient nécessaires pour accepter des règles complexes.

Chapitre 2

Le jeu de Go et l'Intelligence Artificielle

Sommaire

"La position d'un pion évolue au fur et à mesure qu'on déplace les autres. Leur relation, de plus en plus complexe, se transforme et ne correspond jamais tout à fait à ce qui fut médité. Le go se moque du calcul, fait affront à l'imagination. Imprévisible comme l'alchimie des nuages, chaque nouvelle formation est une trahison. Jamais de repos, toujours sur le qui-vive, toujours plus vite, vers ce qu'on a de plus habile, de plus libre, mais aussi de plus froid, précis, assassin. Le go est le jeu du mensonge. On encercle l'ennemi de chimères pour cette seule vérité qu'est la mort."

Shan Sa

2.1 Qu'est ce que le jeu de Go

2.1.1 Introduction

Le jeu de go est né en Chine il y a 4000 ans. Il se joue au Japon depuis 1200 ans, mais il ne s'est répandu que récemment en occident. Les règles, les plus simples de tous les jeux de stratégie, s'apprennent en quelques minutes. Mais ceux qui en exploreront les subtilités pourront constater que sous son apparente simplicité, le jeu de Go est d'une richesse inépuisable.

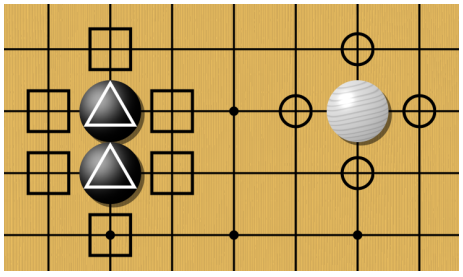
2.1.2 Les règles

Résumé

Chaque joueur possède des "pierres", l'un a les pierres noires, l'autre les pierres blanches. Chacun son tour, les joueurs posent une de leur pierre sur le plateau. Si de cette façon, ils entourent une ou plusieurs pierres adverses, elles sont considérées comme prisonnières, et sont retirées du plateau. Le but du jeu est d'occuper le plus d'espace possible.

Version détaillée

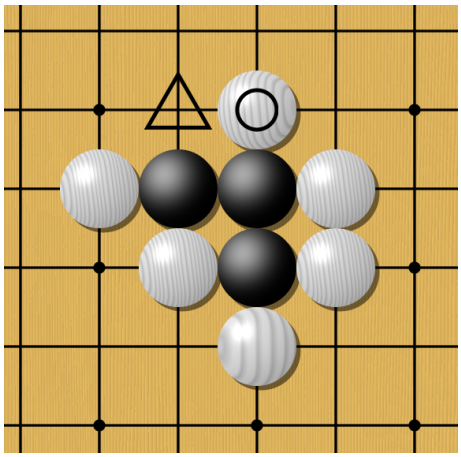
Deux joueurs posent à tour de rôle une pierre de leur couleur sur une intersection inoccupée d'une grille. Des intersections sont voisines si elles sont reliées par une ligne de la grille et sans autre intersection entre elles. Les libertés d'une pierre sont les intersections inoccupées voisines de l'intersection sur laquelle est cette pierre. Des pierres de même couleur posées sur des intersections voisines partagent leurs libertés en commun.



Les libertés de la pierre blanches sont marquées par les ronds. Les deux pierres noires sont voisines et partagent donc leurs libertés, marquées par des carrés.

Capture

Si un joueur supprime la dernière liberté de pierres adverses, il les capture en les retirant de la grille. De plus, il ne doit pas supprimer la dernière liberté de ses propres pierres, sauf s'il capture au moins une pierres adverse.



Si Blanc joue sur l'intersection marqué par le triangle, il capturera les pierres noires.

Répétition, ou ko

Un joueur ne doit pas, en posant une pierre, ramener la grille dans un état identique à l'un de ceux qu'il lui a lui-même déjà donné.

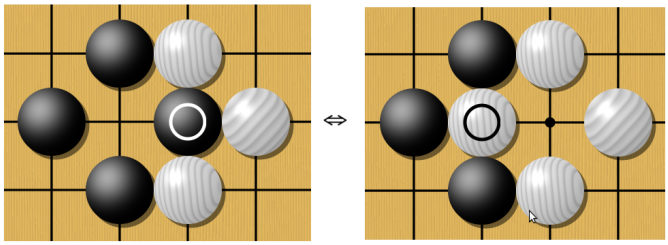


Figure 1 Figure 2

Partons de la figure 1, Blanc peut prendre la pierre noir marquée d'un rond. Cela nous mène à la figure 2. Si la règle précédente n'était pas donnée, Noir pourrait reprendre la pierre blanche marqué d'un rond, et cela nous ramenerait à la figure 1.
Noir n'a donc pas le droit de prendre la pierre immédiatement, et devra jouer son prochain coup sur une autre intersection.

Détermination du gagnant

Des intersections inoccupées sont entourées par un joueur s'il possède une pierre sur chacune des intersections leur étant voisines.

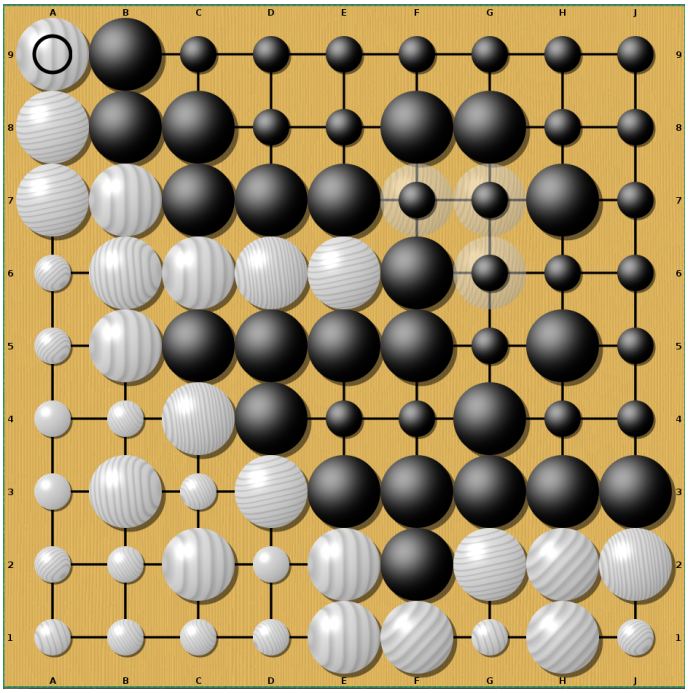
La partie s'arrête lorsque les deux joueurs passent consécutivement. Ils peuvent alors retirer de la grille toutes les pierres adverses qu'ils sont certains de pouvoir capturer. Puis ils comptent les points qui sont le nombre d'intersections qu'ils sont certains de pouvoir occuper ou entourer avec leurs pierres.

En cas de désaccord des joueurs sur le statut de certaines intersections, la partie reprend dans l'état où elle s'était arrêtée jusqu'à ce que les deux joueurs passent de nouveau consécutivement, sachant qu'ensuite, aucune des pierres encore sur la grille ne pourra être retirée. Les points des joueurs seront alors le nombre d'intersections qu'ils occupent ou entourent avec leurs pierres.

Si des intersections inoccupées ne sont entourées par aucun des joueurs, elles sont neutres, et ne donnent de points à personne.

Celui qui a eu le désavantage de ne pas commencer peut recevoir des points de compensation. Mais parfois, on donne volontairement un handicap à l'un des joueurs sous forme de points ou en laissant l'autre jouer plusieurs coups de suite en début de partie.

Le gagnant est celui qui a le plus de points.



Cette figure montre une partie terminée. Noir a fait deux prisonniers pendant la partie. Blanc également. De plus, les pierres blanches F7, G7, G6 sont considérées comme mortes, car elles ne pourront échapper à une capture certaine.
Le score final est donc de 23 (nombre d'intersections dans le camp de Noir) + 5 (nombre de prisonniers faits par Noir) = 28 à 15 (nombre d'intersections dans le camp de Blanc) + 2 (nombre de prisonniers faits par blancs) + 6.5 (komi, points d'avance laissés à blanc pour compenser le désavantage de jouer en deuxième) = 23.5
Noir gagne donc de 4.5 points.

2.1.3 Histoire du Go

Le jeu de Go est probablement né au Tibet il y a près de 4000 ans. Il se développera rapidement en Chine. Des le II° siècle de notre ère, des traités sont écrits sur le jeu, permettant l'affinement au cours des générations de la façon de jouer des grands joueurs. Il arrivera au V° siècle de notre ère en Corée, puis au Japon, ou il connaîtra de grandes avancées. En 1603, le Go, soutenu par les militaires et le Shogunat, entre dans sa période classique et connaît un développement ininterrompu pendant plus de deux siècles et demi. Grace à la protection du shogun, le Go acquiert un statut officiel et devient une institution gouvernementale. Les joueurs professionnels, mécènés par la noblesse, ont donc pu étudier de manière approfondie toutes les subtilités du jeu. Le Go n'arrivera en europe qu'au XVIIe siècle.

Au cours du XXe siècle, le Go se démocratise, et le nombre de joueur de tous les pays augmente d'année en année. De nouvelles ouvertures continuent à être découvertes et de nouveaux styles voient le jour. Le Go semble avoir de beaux jours devant lui.

2.2 Qu'est ce que l'Intelligence Artificielle

2.2.1 Introduction

La notion d'intelligence est une notion que les êtres humains n'ont jamais parfaitement réussi à définir. La question s'est posée à de nombreuses époques mais n'a pas été complètement résolue. D'abord parce que l'homme fait trop référence à lui-même pour définir l'intelligence, ensuite parce que son ego le pousse à ne reconnaître que sa propre intelligence.

2.2.2 Définition

La définition du mot "intelligence" donnée dans le dictionnaire est : "Fonction mentale d'organisation du réel en pensées". Cette définition laisse une grande marge d'interprétation. En effet, il est difficile de définir si un ordinateur arrive à émuler artificiellement une organisation du monde en pensée, sachant que les ordinateurs actuels n'ont pas de vision du monde. La notion d'Intelligence Artificielle consiste donc à donner à l'ordinateur une vision d'un monde virtuel, à la faire raisonner sur les informations qu'il peut en tirer et éventuellement d'en tirer des conclusions qui lui permettent d'agir sur cet environnement. Dans le domaine de la programmation agent, le monde virtuel est appelé "environnement". Si l'on considère que le but théorique d'une intelligence artificielle est de se rapprocher d'une intelligence humaine, il faut que l'environnement ait un maximum de points communs avec celui des êtres humains et que les actions que le programme effectue sur cet environnement soient les plus proches possible de celles des humains.

2.2.3 Les différents types d'intelligences artificielles

Les programmes dits d'intelligences artificielles ont été divisés en trois catégories :

L'Intelligence Artificielle virtuelle

C'est le plus bas niveau d'Intelligence Artificielle, les programmes catégorisés ainsi sont souvent construit à partir d'algorithmes probabilistes. Le but n'est pas ici de copier l'être humain dans sa manière de penser, mais d'arriver grâce à diverses méthodes algorithmiques et/ou mathématiques à simuler son comportement face à un problème donné. Les algorithmes comme A*, un algorithme de parcours en profondeur avec une heuristique, ou comme UCT (Upper Confidence bounds applied to Trees), un algorithme purement probabiliste d'optimisation, en sont de bon exemples.

L'Intelligence Artificielle faible

Ce sont les intelligences qui sont actuellement les plus proches des êtres humains. Elles sont construites dans le but de raisonner de la manière la plus proches possible des êtres humains dans un domaine très précis donné. Les concepts du système expert et les algorithmes ayant des notions d'apprentissage en sont de bons exemples.

L'Intelligence Artificielle forte

Le but d'une intelligence artificielle forte est de ressembler en tous points à l'intelligence humaine. Une intelligence artificielle forte est censée avoir conscience d'elle-même, et doit être capable d'analyser ses propres raisonnements. Selon les défini-


tions, elle devrait avoir un même niveau de conscience du monde qui l'entoure que les humains, et être doté de sentiments. Personne ne peut dire à l'heure actuelle si l'être humain sera un jour capable de construire une intelligence artificielle de ce niveau.

2.2.4 Le jeu de Go comme environnement

La question du choix de l'environnement pour une intelligence artificielle est cruciale car il va définir la précision du jugement que l'on va pouvoir porter sur l'intelligence du programme. Le jeu de Go est un environnement qui appartient au monde des humains. D'abord parce que le Go est un jeu inventé par les humains pour jouer entre eux ; ce qui montre bien que le programme émule une réflexion sur un sujet humain. Ensuite parce que le jeu de Go peut être vu comme une simplification du monde réel. Il possède des dimensions spatiales et temporelles, son état évolue au cours du temps. Mais il possède aussi des différences. La première est qu'il est discret (on connaît son nombre d'intersections, et donc son nombre d'états possibles), alors que notre monde nous apparait comme continu. La deuxième est que la plus petite partie de notre monde peut prendre un nombre d'états infiniment supérieur au nombre d'états que peut avoir un plateau de jeu (nombre toutefois supérieur au nombre estimé d'atomes dans l'univers). Mais ce qui est le plus intéressant, c'est que l'être humain réfléchit face à un goban comme il peut réfléchir dans sa vie : il est capable de hiérarchiser les problèmes auxquels il fait face, de les synthétiser rapidement, en partie inconsciemment, et d'isoler les parties de l'environnement qui l'intéressent pour se concentrer dessus, en mettant de coté les parties qu'il considère comme moins importantes dans la situation présente. C'est pour cela que l'on peut dire que le jeu de Go est un bon outil pour tester les idées d'algorithmes dits d'Intelligence Artificielle.

2.3 Les difficultés des intelligences artificielles de Go

2.3.1 Historique

La création d'une intelligence artificielle de Go est un des grand défi de la création d'intelligences artificielles modernes. Depuis la défaite du meilleur joueur du monde d'échecs face à une machine (en 1997, Kasparov perd contre Deep Blue dans un tournoi en 6 manches), l'envie d'arriver à faire un programme capable de vaincre les meilleurs joueurs du monde au jeu de Go devient à la mode. Mais à cette époque, les intelligences artificielles de Go sont extrêmement mauvaises, et il est déjà dur de créer une intelligence artificielle capable de battre des amateurs moyens. Depuis ce temps, grâce au progrès des machines (en 13 ans, les machines grands publics comme les supercalculateurs sont de puissance largement supérieure), mais aussi des algorithmes de programmation, les programmes de Go ont beaucoup évolués, et certains arrivent à battre des joueurs professionnels avec un certain nombre de pierres d'avance. En Juillet 2008, pour la première fois, un ordinateur sur lequel tourne le programme Mogo bat un professionnel avec 9 pierres d'avance pour l'ordinateur. Plus tard, en Février 2009, MoGo gagne une partie avec 7 pierres d'avance contre un des meilleurs joueurs de notre époque, Zhou Junxun. En aout 2009, Zhou Junxun prendra sa revanche, et vaincra les programmes Many Faces of Go, MoGo, et Zen dans des parties à 7 pierres de handicap.

2.3.2 Vision des joueurs de Go

Le Go est considéré par ses grands joueurs comme un jeu qui reflète la personnalité de celui qui y joue. Cela explique pourquoi dans le monde professionnel du Go, peu de gens croient en l'arrivée d'une intelligence artificielle capable un jour de les battre. De plus, ces joueurs sont censés faire évoluer le jeu, et refusent de croire que ce sera finalement des ordinateurs qui arriveront à nous faire découvrir des stratégies toujours plus proches des coups ultimes. Toutefois, même si on reste loin d'une intelligence artificielle capable de rivaliser avec eux, les progrès récents dans ce domaine risquent de les forcer à finir par remettre ce jugement en question.

2.3.3 Le problème de l'espace des possibles

Au Go, l'espace des possible est extrêmement grand.Tout d'abord, la taille du plateau est énorme. Le plateau possède 361 intersections. Cela en fait 6 fois plus que sur un plateau d'échecs. Ensuite, un coup consiste à poser une pierre sur n'importe laquelle des intersections libres. Soit au premier coup 361 possibilités contre 20 aux échecs. Une exploration des différentes évolutions possibles du plateau au Go sera donc beaucoup moins efficace qu'aux échecs, même en utilisant des algorithmes d'exploration partielle. Cela pose un premier gros problème aux programmeurs qui doivent réfléchir à de nouveaux types d'algorithmes.

2.3.4 Le problème de la mise en algorithme

Il est assez facile de faire une fonction d'estimation du score pour la plupart des jeux de stratégie. Sur tous ceux de la famille des échecs, cela dépend des pièces sur le plateau et de leurs emplacements. Au Go, il n'y a à ce jour aucun algorithme capable d'estimer une position de manière correcte avant la fin de partie. Cela est du au fait que l'estimation est basée à la fois sur un très grand nombre de paramètres, mais aussi sur le "ressenti" du joueur. Aucun joueur ne peut chiffrer de manière absolue une position de début de partie. Ils peuvent "préférer" la position de Noir ou celle de Blanc, mais sans donner de formule absolue permettant d'expliquer leur choix. De la même façon, lorsqu'un joueur dira qu'un coup est meilleur qu'un autre, ce sera toujours basé sur des conditions trop floues pour être implémentées de manière algorithmique. Il est donc très difficile de faire un algorithme fonctionnant "comme un humain".

2.4 L'apport du Go aux intelligences artificielles

Le jeu de Go est intéressant d'un point de vue informatique parce qu'aucun algorithme connu n'est capable à ce jour de rivaliser avec un joueur professionnel sur une partie à égalité. Il stimule donc les programmeurs à trouver de nouveaux algorithmes pour atteindre cet objectif. Grâce à ces règles, son système de pierres d'avance, dites pierres de Handicap pour le joueur le plus faible, et la fin de partie qui se finie par une différence de points entre les deux joueurs, il est facile de mesurer les progrès entre deux intelligences artificielles. De plus, les programmes tentant de fonctionner en prenant en compte les connaissances de l'Humain sur le jeu n'étant pas les plus efficaces, on peut penser que les progrès en algorithmes créés pour ce défi seront certainement utiles pour d'autre domaines de l'informatique. Actuellement, les intelligences artificielles permettent de montrer l'efficacité d'algorithmes,


et comment ceux-ci peuvent être combinés. Ce sujet de recherche devrait donc rester encore longtemps ouvert et intéressant.

2.5 Les principales idées autour des intelligences artificielles de Go

2.5.1 Introduction

Le jeu de Go demande de la stratégie, de la tactique, et ce que les joueurs appellent de l'instinct. Au fur et à mesure des défaites de leurs programmes, les programmeurs d'intelligences artificielles de Go ont tentés de mettre en place des stratégies très différentes les unes des autres pour tenter de trouver une façon d'aborder le jeu de manière efficace. Elles ont toutes des points faibles et des points forts, selon le type d'adversaire qu'elles rencontrent, et la façon dont la partie évolue.

2.5.2 Les stratégies des programmeurs

Système expert

L'idée la plus naturelle lorsque l'on commence à développer une intelligence artificielle pour un jeu est d'essayer de lui faire mimer le comportement humain. Le but d'un système expert est d'essayer de transmettre à la machine sous forme d'algorithmes les connaissances humaines. Les programmes utilisant cette technique ont beaucoup de mal car il faut sans cesse rajouter des connaissances pour se rapprocher de la connaissance humaine, qui au Go représente 4000 ans d'histoire et devient très vaste et très complexe . Ce genre de programmes requiert beaucoup plus de temps de création que ceux réalisés avec les autres stratégies, et leurs sont à l'heure actuelle inférieurs. De plus, face à des joueurs qui jouent loin de la théorie, ils vont vite se retrouver perdu. Mais ils ont aussi des avantages : une fois que l'on repère leurs erreurs, il est relativement facile de les corriger, et son style de jeu est beaucoup plus humain que celui des autre programmes. Ainsi, pour un joueur débutant qui voudrait s'entrainer avec un programme, il est beaucoup plus intéressant d'utiliser ce type de programme.

Reconnaissance de formes

Il y a plusieurs types d'algorithmes de reconnaissance de formes. Le plus utilisé est l'utilisation d'une base de données de parties professionnelles (plusieurs milliers de parties de très forts joueurs, au format SGF), et l'Intelligence Artificielle tente de retrouver dans la base des formes qui ressemblent à celles qu'elle croise en partie, en recoupant des positions locales dans les parties de la base. Ces algorithmes ont pour inconvénient de perdre de manière régulière contre des joueurs de faible niveau, car les formes qu'ils posent n'existent pas dans le dictionnaire. Toutefois, ce type d'algorithmes est assez efficace lorsqu'il est utilisé en combinaison avec d'autres algorithmes.

Algorithmes d'apprentissage : les réseaux neuronaux

Certains programmeurs ont tentés d'implémenter des réseaux neuronaux pour évaluer les positions, puis de faire tourner un Min-Max pour trouver le meilleur coup en se servant de l'algorithme d'évaluation à base de neurones. Les neurones sont basés sur des connaissances simples sur la partie : le nombre de libertés de chaque

pierre, les pierres voisines de chaque intersection, le nombre de pierres de chaque couleur... Ils ont souvent des jeux plus intéressants que les systèmes experts, car leur apprentissage en fait des programmes qui évoluent plus vite, car ils ne nécéssitent pas un expert pour compléter régulièrement la base de connaissances. Leur plus gros problème est qu'il est très difficile de corriger leurs travers lorsque l'on les voit faire des erreurs en partie. La seule solution consiste alors à les faire repasser par une phase d'apprentissage en compétant la base de données de positions, ce qui peut à nouveau entraîner d'autre erreurs de la part du programme. Leur apprentissage est donc en partie négativement compensé par la difficulté à les corriger par le programmeur.

UCT

UCT est un algorithme qui a pour but de générer un arbre partiel des parties possibles, en n'ouvrant que les branches qui lui semblent être les plus intéressantes. Il va donc créer au fur et à mesure des millions de parties sur le principe du min/max (chaque coup d'une partie va être un de ceux considéré comme le meilleur pour le joueur à qui c'est de jouer), et choisir au final le coup qui lui donnera le plus de victoire sur les parties qu'il engendre. C'est un algorithme qui exploite au maximum les capacités de l'ordinateur, et qui ne nécessite que des connaissances minimes sur le jeu pour le coder. UCT semble être un des algorithmes les plus prometteurs, malgré quelque défauts : A partir d'un certain point des parties théoriques créées, les données sur la qualité d'un coup deviennent faible, et la partie théorique se fini de manière assez aléatoire. Du coup, il se retrouve mauvais sur des positions ou il faut lire de longues séquences de coups. De plus, vu l'infime nombre de partie jouée par rapport à la taille de l'arbre des possibles, et le fait que beaucoup de choix d'exploration de branches soient en partie aléatoire, le coup trouvé est probabiliste et peut donc être plus ou moins bon selon les exécutions.

Et les autres...

Beaucoup de programmes jouant au go sont des programmes commerciaux, et leurs algorithmes ne sont pas rendus publiques. Le prix de vente des logiciels et les sommes mises en jeux dans les concours entre programmes poussent les programmeurs à ne pas dévoiler leurs idées et encore moins leurs sources. Toute une partie de la recherche sur ce sujet reste donc caché.

Chapitre 3

Les fondations

Sommaire

" Le Go utilise la plupart des matériaux et des concepts élémentaires, la ligne et le cercle, le bois et la pierre, le noir et le blanc, les combinant avec des règles simples pour générer des stratégies et des tactiques subtiles qui stimulent l'imagination. "

Iwamoto Kaoru

3.1 Modélisation du jeu de Go

Le système de règles a besoin d'informations pour fonctionner en respectant les règles du jeu. Ainsi, le goban permet de savoir à tout moment quels sont les groupes de pierres morts, les libertés, les coups interdits, les limites du goban, le nombre de prisonniers de chaque joueur. Ces informations essentielles sont calculées par nos outils à chaque coup joué.

3.2 Les outils de l'intelligence artificielle

L'outil fondamental nécessaire pour implémenter une intelligence artificielle de go est bien évidemment le jeu en lui-même, représenté par le plateau (goban) sur lequel on pose des pierres. Mais pour aller au delà, nous avons créé une classe goban puissante, non seulement capable de modéliser une partie en cours et de poser des pierres, mais également capable de faire un grand nombre de calculs utiles pour l'IA qui n'aura qu'à interroger l'objet de la classe pour obtenir des informations pré-calculées. Nous allons détailler les fonctionnalités de la classe Goban, ainsi que des "sous-classes" qu'elle utilise.

3.2.1 Les attributs d'un goban

Un goban est constitué d'intersections, réparties sur une matrice "carrée" d'une taille généralement de 9x9 ou 19x19. Les intersections sont des objets à part entière, qui connaissent leur position sur le plateau et un florilège d'autres informations. La taille du goban peut être définie dans le constructeur de l'objet, ce qui permettra de créer dynamiquement des gobans de différentes tailles, nécéssaires aux calculs des fonctions de l'intelligence artificielle, ou pour jouer sur différents plateaux. Le goban dispose également d'un constructeur par copie.

L'objet de la classe Goban calcule la liste des groupes de pierres blanches et des groupes de pierres noires qui le composent. Ces listes (ainsi que la majorité des informations que possède le goban) sont mises a jour a chaque coup joué. Un groupe est un ensemble relié de pierres d'une même couleur. Les groupes sont eux-mêmes des objets qui possèdent des fonctions complexes, comme la capacité de "fusionner" entre eux ou avec d'autres objets, et connaissent leur nombre de libertés (ils peuvent donc être détruits lorsqu'ils meurent).

Le goban a également un historique de tous les coups joués depuis le début de la partie, ce qui permet de recomposer des séquences locales dans l'ordre dans lequel elles ont été jouées, même si des coups ont été joués ailleurs pendant la séquence.

Enfin, le goban tient à jour quatre listes de coups dans les coins, sait ou a été joué le dernier ko, est capable de calculer les voisins d'une pierre donnée et met à jour les libertés de tous les groupes lorsqu'un groupe meurt.

Cet objet, que nous pensions au départ simplement comme un support pour poser des pierres, s'est enrichi de nombreuses fonctions qui en ont fait un outil complet pour récolter des informations sur l'état d'une partie.

3.2.2 Les intersections

Une intersection est un emplacement sur le plateau qui possède quatre états : vide, interdit, blanc ou noir. En plus de cela, il nous a paru intéressant que chaque intersection puisse être interrogée individuellement pour fournir des informations très locales. Ainsi, une intersection non-vide connaît la séquence des pierres qui

l'entourent sur une surface de 9x9, dans l'ordre dans lesquelles elles ont été jouées. Cette information est très importante afin de retrouver les séquences contenues dans le dictionnaire des josekis, par exemple.

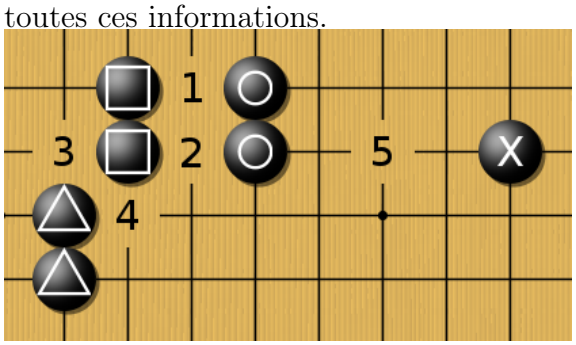
De plus, chaque intersection peut être interrogée sur ses voisins les plus proches à gauche et à droite, ainsi que leur distance. Cela permet d'évaluer le potentiel de "fuite" d'un coup précis. Il n'est en effet pas recommandé de jouer un coup isolé entouré de deux pierres ennemies.

3.2.3 Les groupes

La classe Groupe fut délicate à implémenter, car nous voulions plusieurs types de groupes qui soient "compatibles" entre eux, c'est à dire qui soient basés sur un même modèle et puissent fusionner quel que soit leur type. Nous avons implémenté une classe "abstraite" TGroupe qui possède notamment une fonction "fusionner" qui utilise la généricité paramétrique pour prendre en paramètre un objet d'un type d'une sous-classe. Ces sous-classes sont les trois types de groupes possibles : le Groupe (pierres reliées), le SuperGroupe (pierres proches, non reliées mais non coupables) et l'HyperGroupe (ensemble de SuperGroupes étant "presque" reliés et faits pour travailler ensemble.).

Ainsi, n'importe lequel de ces objets peut appeler la fonction "fusionner" sur un des autres types de groupes.

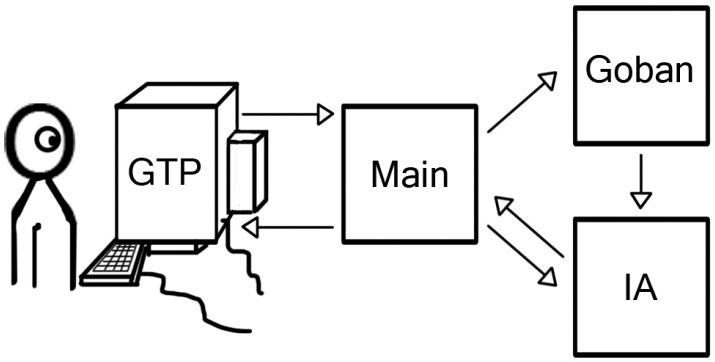
Nous avons ainsi une modélisation assez complète des relations entre les pierres et les groupes, ainsi qu'un outil qui se charge automatiquement de mettre a jour toutes ces informations.



Chaque symbole désigne un groupe.
Le groupe marqué par des ronds appartient au même superGroupe que le groupe marqué par des carrés, car si blanc joue en 1 ou en 2 pour tenter de séparer les pierres, noir jouera sur l'autre des deux intersections.
Le groupe marqué par des triangles appartient au même superGroupe que le groupe marqué par des carrés, car si blanc joue en 3 ou en 4 pour tenter de séparer les pierres, noir jouera sur l'autre des deux intersecions.
Par transitivité, le groupe marqué par des ronds appartient au même superGroupe que le groupe marqué par des triangles.
Le groupe marqué par une croix, composé d'une seule et unique pierre, n'appartient pas au même superGroupe que les trois autre groupes, car si blanc joue en 5, les pierres ne seront pas sûres de pouvoir se relier.
Toutefois, on peut dire que son superGroupe appartient au meme hyperGroupe que les trois autre pierres, car les pierres travaillent ensemble, et que blanc devra prendre un risque si il veut couper les superGroupes.
Une pierre blanche en 5 serait en danger.

3.3 Interface graphique GTP

Jouer contre notre programme en mode console n'était pas satisfaisant, ni pour le faire évoluer, ni pour pouvoir prendre plaisir à jouer contre lui. Nous devions donc pouvoir jouer contre lui à travers une interface graphique. De là, il y avait deux solutions : Créer notre propre interface graphique, ou en utiliser une existante. Avant de choisir l'interface, (les interfaces ressemblent beaucoup les unes aux autres) nous avons choisi le protocole. GTP, ou Go Text Protocol, est un protocole permettant de transmettre des coups d'un programme qui joue au Go vers un programme qui les interprète, comme une interface graphique, une autre intelligence artificielle, ou un serveur de jeu en ligne. Ainsi, notre programme peut facilement être utilisé à travers des interfaces différentes.



Chapitre 4

Le système expert.

Sommaire

"Le Monde est un Jeu de Go dont les règles ont été inutilement compliquées."
poète chinois

4.1 Qu'est ce qu'un système expert

4.1.1 Définition d'un système expert

Un système expert est un outil ayant pour objectif d'imiter le raisonnement d'un humain dans un domaine précis.

De manière plus technique, il permet d'obtenir des conclusions à partir de faits et de règles. Les faits sont des prédicats prenant des paramètres et considérés comme vrais une fois qu'ils sont intégrés à une base de faits. Les règles se construisent comme une formule logique, elles partent de faits et aboutissent à une conclusion, d'après le modèle :

Fait1 et Fait2 $\rightarrow$ Conclusion

La conclusion est souvent unique, si on veut déduire plusieurs choses des faits 1 et 2 il faudra créer une nouvelle règle. Le système expert se charge de parcourir les règles à partir d'une base de faits qui sont considérés comme vrais, en général jusqu'à ce qu'il ne trouve plus de nouvelles règles à appliquer. Une conclusion peut induire de nouveaux faits, qui pourront eux même déclencher de nouvelles règles. Le système s'arrête lorsque plus aucun nouveau fait ne peut être déduit.

Cycle

Il peut se créer ce que l'on appelle un "Cycle" s'il existe des variables conditionnelles. Les variables conditionnelles sont des variables qui n'existaient pas dans l'hypothèse, par exemple

Humain(x) $\rightarrow \exists$ H (Humain(H) et père (x,H)).

A partir de cette règle on peut générer un nombre infini de "pères".

Il est également possible de créer un cycle lorsque l'on autorise le retrait des faits de la base ou la négation, et que des faits soient retirés et rajoutés alternativement. par exemple :

$\neg$a $\rightarrow$ a

a $\rightarrow \neg$a

Ces deux règles ajoutent et retirent a à chaque passage.

Notre implémentation n'utilise pas de variables conditionnelles mais autorise la négation de faits.

Les limites des systemes experts

Lorsque le système devient trés complexes, il peut contenir un grand nombre de règles. Avec plusieurs milliers de règles le système atteint des temps de calculs inacceptables, puisque la complexité d'un système expert est exponentielle avec le nombre de règles. Cela peut obliger à segmenter les règles en différents modules afin de n'appliquer qu'un type de règle pour une situation donnée, mais cela contredit le principe de base du système expert qui est de pouvoir ajouter des règles de façon incrémentale sans se soucier du contexte dans lequel elles vont être testées. De plus, ce procédé est délicat dans sa mise en oeuvre et cela ne garantit plus d'arriver au résultat demandé.

4.1.2 Le système expert dans le jeu de Go

Appliqué au jeu de go, le système Expert est théoriquement fait pour donner une stratégie à l'intelligence artificielle, de jeu qui sera ensuite épaulée par de la

combinatoire pour les combats. Les régles donnent des directions relativement générales de coups à jouer, mais certaines règles se rapprochent un peu de la tactique et utilisent un début de combinatoire. Par exemple, le prédicat "shisho" utilise un prémice l'algorithme combinatoire pour mesurer l'importance d'un coup précis.

Les prédicats qui déclenchent des règles statueront sur des questions telles que : Ce coin est il vide ? Est ce qu'il existe un bord vide ? A quel stade de la partie en sommes-nous ? Un groupe est-il en danger ?

Les conclusions sont principalement des fonctions qui mettent à jour des poids dans un tableau, comme par exemple "jouer joseki" qui modifie les poids des cases données par un dictionnaire des joseki, ou plus simplement "jouer bord" qui incrémente les poids des cases prés d'un bord donné. De plus, les conclusions peuvent être des faits utiles pour déclencher de nouvelles règles. Par exemple, un fait peut être "on est dans le yosé (fin de partie)".

4.2 Fonctionnement de notre système expert

4.2.1 Générateur de règles

La première étape consiste à transformer des règles écrites par un expert en code informatique. Pour cela nous avons implémenté un mini programme qui lit un fichier de règles sous forme de texte et le transforme automatiquement en code compilable.

Les règles sont composées d'une hypothèse et d'une conclusion. L'hypothèse est formée d'un ensemble de prédicats. Leur syntaxe est la suivante :

```
Fait1 ^ Fait2 ^ ... ^ Faitn -> Conclusion
```

Certaines conclusions donnent des informations sur la valeur de coups, par exemple :

```
Coin(x) -> jouer(J_joseki(x),15)
```

Cette ligne est à lire de la manière suivante : "Pour tout coin x, jouer un joseki dans ce coin x avec un poids de 15".

Nous avons également des règles qui génèrent des faits, par exemple :

```
Groupe(x) ^ nbLibertes(x,3) ->  ajouterFait(Danger(x))
```

qui signifie "Pour tout groupe x qui dispose de 3 libertés ou moins, créer le fait « Danger »pour ce groupe. "

Le fait est stocké dans une base de faits, et il pourra être testé de la manière suivante :

```
Groupe(x) ^ testerFait(Danger(x)) -> jouer(J_liberté(x),10)
```

traduit par "Pour tout groupe signalé en danger, augmenter son nombre de libertés avec un poids de 10".

Il faut voir les prédicats de l'hypothèse comme des filtres qui éliminent les éléments qui ne vérifient pas une proprité. Quand on écrit "Coin(x)" le programme va créer une structure de données "x" contenant tous les coins du plateau. Le prédicat sera toujours vrai, car il existe toujours un coin dans le plateau. Si j'écris à la suite "CoinVide(x)", les coins qui ne sont pas vides vont être retirés de x, et le prédicat sera vrai s'il existe au moins un coin vide, dans ce cas on pourra appliquer la conclusion sur ce coin vide. Ecrire "Coin(x)" est obligatoire pour initialiser et faire des "filtrages" par la suite. Si j'écris simplement CoinVide(x) -> conclusion, la conclusion ne sera jamais exécutée car x restera toujours vide. On obtient une règle :

```
Coin(x) ^ CoinVide(x) -> jouer(J_coin(x),15)
```

qui incite très fortement à jouer sur un coin qui est vide.

On peut également avoir plusieurs structures de données contenant deux types d'éléments différents, pour les utiliser dans un prédicat commun, par exemple :

```
Coin(x) ^ Bord(y) ^ CoinVide(x) ^ BordVide(y) ^ appartientabord(x,y)
 -> jouer(J_joseki(x),5)
```

Cette règle indique qu'il faut jouer un joseki dans un coin vide appartenant à un bord vide.

On pourrait cependant vouloir préciser que le coup appartenant au joseki doit être joué sur le bord vide, dans ce cas la conclusion deviendrait :

```
jouer(J_joseki(x,y),5)
```

Ces règles sont chacune automatiquement transformées en une fonction qui sera compilée avec le reste du programme.

Le cas du "non"

Le "non" a été traité pour permettre d'inverser l'effet d'un prédicat en hypothèse. Ainsi, on peut écrire :

```
Coin(x) ^ non(CoinVide(x))
```

apres le deuxième prédicat, x va contenir les coins non vides, et le prédicat renverra vrai s'il existe au moins un coin non vide.

Le fonctionnement reste donc identique, le "non" filtre les valeurs de x pour ne laisser que celles qui ne respectent pas le prédicat.

Fonctions et prédicats disponibles

La liste des prédicats et fonctions disponibles pour l'écriture des règles est disponible en annexe. Cette liste n'est pas exhaustive et comprend ce qui a été implémenté au moment de l'écriture de ce rapport.

Les conclusions peuvent être de deux types, soit utilisant la fonction "jouer" pour déclencher une fonction qui modifie des valeurs de coups, soit manipulant des faits avec "ajouterFait" et "retirerFait".

4.2.2 La conclusion "jouer"

La fonction jouer est générique, elle prend en paramètres une fonction et ses arguments, ainsi qu'un poids. La fonction passée en paramètre est chargée de renvoyer une table de poids, qui seront eux même pondérés par le poids donné en paramètre de jouer.

Les fonctions qui modifient des poids, et donc qui donnent un "ordre de jeu" à l'IA, sont identifiables par leur nom commençant par "J_" Par exemple, la fonction J_joseki prend en paramètre une liste de coins et appelle le dictionnaire des josekis sur ces coins. Les plus simples servent à donner de l'importance à une partie du plateau, comme J_bord qui incite à jouer (ou à ne pas jouer si l'on met un poids négatif) sur un ou plusieurs bords. De même que pour les prédicats, ces fonctions doivent être écrites par le programmeur selon la demande de l'expert.

4.2.3 Les conclusions "ajouterFait" et "retirerFait"

ajouterFait ajoute un fait et ses arguments à une base de faits (si le fait n'a pas déjà été ajouté). Les arguments sont de même type que précédemment et doivent être définis par un prédicat, Coin ou Bord par exemple. Cette base est indépendante du reste, et contrairement à ce que l'on voit dans un système expert standard, les générer n'est pas l'objectif recherché. Ces faits servent à définir une connaissance qui pourra être utilisée ultérieurement par une règle grâce au prédicat testerFait, qui compare le fait donné et ses arguments à la base de faits.

retirerFait retire un fait dans la base, cependant l'expert doit faire attention à ne pas créer de "boucles" dans la système, avec un fait qui s'ajoute et se retire infiniement. S'il existe une règle qui ajoute un fait s'il n'y est pas et une autre qui le retire s'il y est, cela crée un risque de boucle infinie.

Ces faits sont utiles notamment pour définir des règles valables uniquement à un moment donné de la partie, par exemple le chuban (milieu de partie) :

```
nbcoupsjoues(12) ^ Bord(x) ^ non(placeSurBord(5)) -> ajouterFait(Chuban())
```

Se traduit par : Si le nombre de coups jouées est supérieur à 12 et qu'il n'existe plus de bords avec 5 intersections vides côtes à côtes alors la partie en est au chuban.

Ensuite, si on veut une règle valable uniquement en milieu de partie, on écrira :

```
Intersection(x) ^ PierreAlliee(x) ^ Bord(y) ^ appartientAbord(x,y)
 ^ testerFait(Chuban()) ->jouer(J_Extention(x),3)
```

qui, en milieu de partie, fait jouer autour d'une pierre alliée qui est sur un bord.

4.2.4 L'exécution des règles

Les fonctions générées automatiquement sont simplement exécutées les unes à la suite des autres. Tous les prédicats de type "Coin(x)" sont des fonctions implémentées qui manipulent une structure de données et utilisent toutes les informations de la classe Goban. Le goban fournissant une grande quantité d'informations, il est possible de créer une infinité de prédicats, selon l'envie de l'expert. Cependant, le codage de ces fonctions est le rôle du programmeur.

4.2.5 Les limites de notre implémentation

Ordonnancement des faits

Le principal défaut de notre modélisation de règles est que l'ordre dans lequel apparaissent les prédicats dans une hypothèse influe sur le résultat. En effet, ceux-ci agissent comme des "filtres" et les prédicats les plus à droite utilisent ce qu'il reste des arguments. Par exemple,

```
Coin(x) ^ CoinVide(x) ^ Bord(y) ^ appartientAbord(x,y) ^ bordleplusvide(y)
```

renvoie dans y le "bord le plus vide" parmi les bords qui ont on coin vide.

```
Coin(x) ^ CoinVide(x) ^ Bord(y) ^ bordleplusvide(y) ^ appartientAbord(x,y)
```

indique si le "bord le plus vide" a un coin vide ou non.

En échangeant de place les deux derniers prédicats, le test ne fait plus du tout la même chose, alors que ce n'est pas censé être le cas dans un système expert. Cependant, pour notre utilisation, nous n'avons pas rencontré de cas ou cela posait un grave problème.

Cycles

Notre système autorise le retrait de faits, ce qui crée un risque potentiel de création de cycle. Cependant, en temps normal, les faits sont ajoutés de manière ordonnée. Par exemple, lorsque l'on arrive en milieu de partie, le fait "chuban" est généré, le fait "fuseki" est retiré et à priori rien ne pourra le regénérer.
Pour assurer l'absence de cycle dans le cas ou la base grandirait, il faudrait que les fonctions "ajouterFait" gère elle-même le retrait des faits obselètes et que la fonciton "retirerFait" soit cachée à l'expert.

Par exemple, lorsque l'on appellerait ajouterFait(chuban) le fait fuseki serait automatiquement retiré. De plus, si ajouterFait(fuseki) était appelé, fuseki ne serait pas ajouté à la base si la base contient déja chuban.

Chapitre 5

La reconnaissance des formes

Sommaire

" Une partie de Go se construit toujours avec des formes. Celui qui fait les meilleurs formes, donc les plus efficaces, prend l'avantage. le jeu de go, est, finalement, d'une grande simplicité : Il suffit de jouer les bonnes formes."

Fan Hui

5.1 L'importance des formes au Go

Au Go, la maîtrise des formes est l'élément central de toute partie. Gagner une partie peut se faire en se contentant de faire des bonnes formes, tout en empêchant son adversaire d'en faire. Mais la qualité d'une forme dépend d'énormément de paramètres. Certains sont plutôt objectifs, comme la distance des pierres par rapport aux bords et aux coins, l'arrivée de shishos, le nombre de menaces de ko de chaque joueur... D'autres sont beaucoup plus subjectifs, comme la solidité des groupes qui l' entoure, l'écart de points estimé, le partage influence/territoire de chaque joueur... Les subtilités pour choisir une forme sont extrêmement complexes pour un humain, et encore plus difficiles à entrer dans un programme de manière conventionnelle.

Nous avons essayé de gérer un maximum de paramètres de choix des formes de différentes manières avec notre programme, et même si certains restent mal gérés, nous avons fait en sorte que le programme soit évolutif à ce niveau, et que les points manquants soient plus ou moins compensés par les autre modules du programme. Nous doutons cependant qu'un programme puisse un jour choisir les formes avec autant d'élégance qu'un humain.

5.2 Les outils utilisés

5.2.1 Le système de stockage

La reconnaissance de formes par notre programme est basée sur une série de dictionnaires de forme. Pour l'évolutivité du programme, il nous semblait important de choisir un format standardisé pour stocker les dictionnaires de formes. Après de rapides recherches nous avons vu qu'un format standard permettant de stocker des parties de Go, ainsi que des variations de parties existait : le format SGF (Smart Game Format). A la base destiné à être utilisé pour stocker les parties de tous types de jeux, il n'est finalement principalement utilisé que pour le jeu de Go. Il enregistre les parties et leurs variations sous forme d'arbres, et permet d'ajouter des symboles et commentaires à chaque position de la partie. Il existe plusieurs éditeurs, dont certains open source. Ce format semblait nous convenir parfaitement et nous avons décidé de stocker les formes comme des positions dans une partie. Nous avons ensuite utilisé des fonctions présentes dans le programme open source Mugo pour parser les fichiers et les transformer en données compréhensibles par notre programme. Puis nous avons défini des spécificités pour plusieurs symboles de façon à pouvoir donner un maximum d'informations sur les positions. Nous avons donc un système de stockage de formes qui permet une modification simple des dictionnaires, et une utilisation rapide par le programme.

5.2.2 La construction de l'arbre

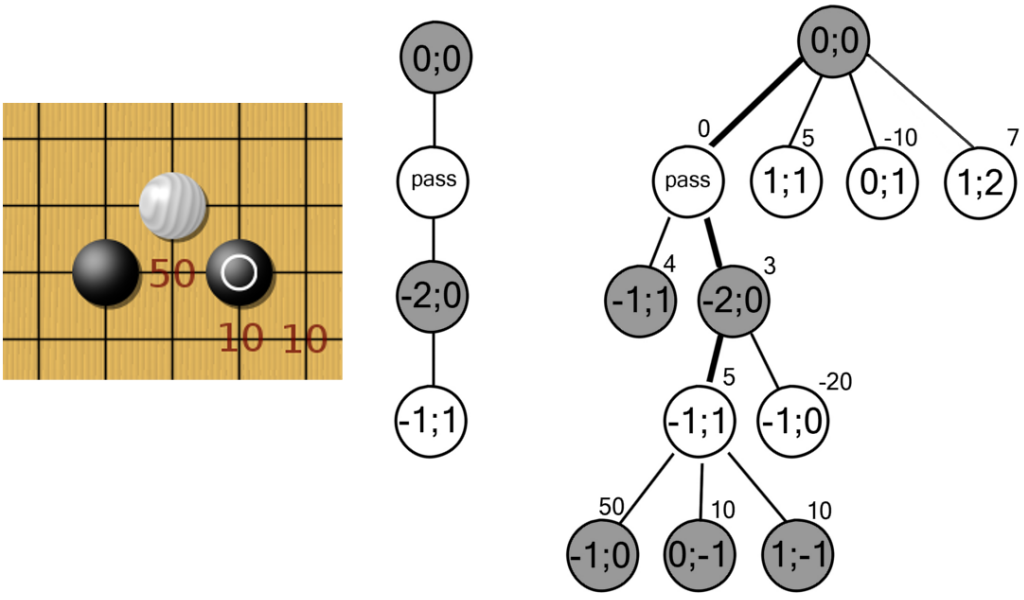
La construction d'un arbre C++ depuis un fichier SGF a été un des grands problèmes de cette partie du projet. Nous avons décidé de récuperer un code déjà fait et de l'adapter, puisqu'il existe de nombreuses implémentations de lecteurs de fichier SGF. Après plusieurs tests d'adaptation des principaux programmes connus sur le sujet, dont celui créé par l'inventeur du format sur lesquels nous avons passé beaucoup de temps, nous nous sommes rendu compte qu'ils étaient très peu modulaires et difficiles à adapter. Nous avons récupéré le code de Mugo, moins connu, mais plus simple à comprendre et à utiliser.

5.3 Algorithmes de reconnaissance des formes

Il existe plusieurs types d'algorithmes de reconnaissance de formes. Nous allons vous présenter ici les différents algorithmes existants, ceux que nous avons implémentés, et ceux des autre programmes.

5.3.1 La reconnaissance d'une liste de coups depuis le dictionnaire en arbre

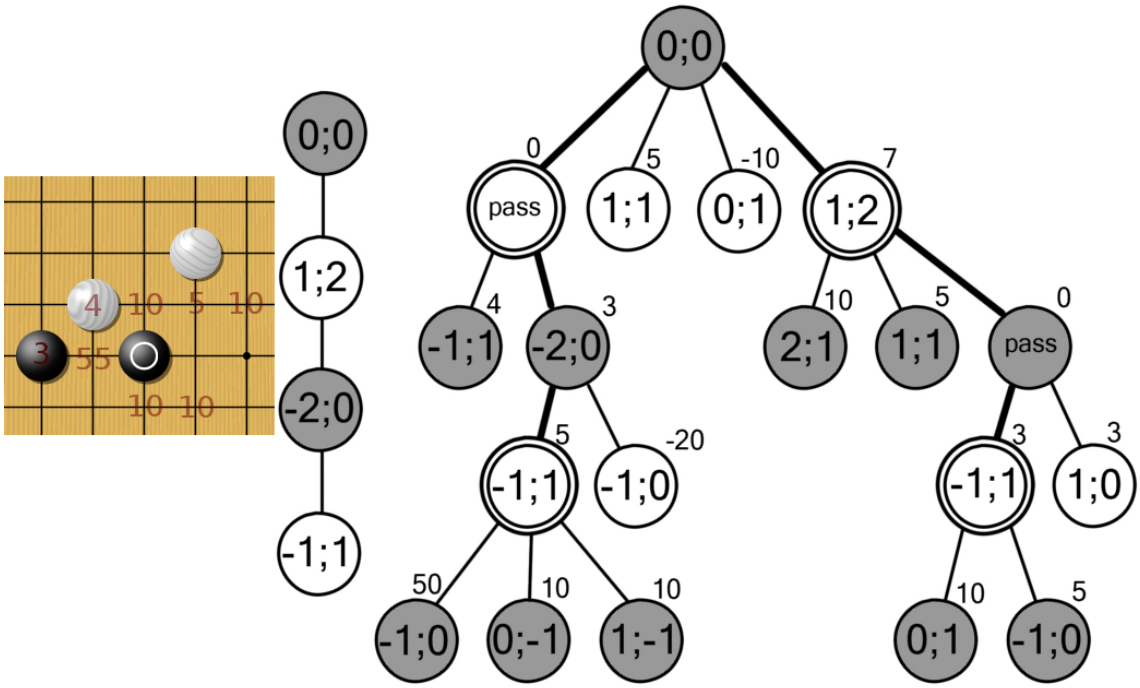
Dans notre goban, chaque intersection contenant une pierre connaît les coordonnées relatives des pierres voisines, et l'ordre dans lequel elles ont été jouées. La première idée naturelle qui nous est venu a été de comparer cette liste avec l'arbre des listes de coups du dictionnaire de formes, et de voir si une des branches de l'arbre correspondait à la série de coups donnée. Les tests étaient fait dans toutes les orientations possibles, de façon à tenir compte des symétries. De plus, un sytème de découpe de zones a été mis en place pour que la pierre puisse trouver des formes correspondant juste à une partie géométrique du goban. Par exemple, elle pourrait ne tenir compte que des pierres au nord-ouest d'elle-même. Cet algorithme nous donnait de bons résultats au début, mais posait un problème quand la forme n'était pas construite dans l'ordre connu par le dictionnaire. Bien que cela soit rare au Go, car les formes sont constuites dans un ordre logique, la gravité d'une erreur dans la lecture d'une forme aurait pu pousser le programme à jouer un très mauvais coup, il nous fallait donc trouver une solution capable de reconnaitre les formes jouées dans le désordre.



La forme de la pierre noire marquée d'un cercle contient les deux autres pierres dans sa liste de voisins. Comme son premier voisin a été la deuxième pierre noire, blanc à un coup vide représenté par le coup "pass" entre les deux pierres noires.
La séquence est retrouvée par le dictionnaire qui marque les cases qui complètent la forme par les poids qui correspondent.

5.3.2 La reconnaissance d'une forme depuis le dictionnaire en arbre

Cette deuxième idée est une amélioration de la première, et utilise les mêmes outils : le dictionnaire et la liste de pierres voisines d'une intersection. Elle permet de reconnaître des formes même si celles-ci ne sont pas entrées dans le bon ordre, et peut reconnaître plusieurs formes pour une même intersection. L'idée est de regarder le coup proposé par chaque branche, et de voir si il est dans la liste de l'intersection sélectionnée. Une fois cela fait, on considère tous les noeuds par lesquels la recherche est passée et où la dernière pierre posée est de la couleur demandée (blanche si on lit une forme appartenant à la couleur du programme, noire sinon) sont considérées comme des formes reconnues.



La forme de la pierre noire marquée d'un cercle contient les autres pierres dans sa liste de voisins.
Toutes les pierres filles par lesquelles l'algorithme de recherche est passé, et qui sont entourées, renvoient le poids de leurs noeuds fils.
Ainsi l'intersection entre les deux pierres noires (-1 ;0) vaut 55 car un noeud l'incrémente de 50 et l'autre de 5.
L'arbre complet passe dans cette situation par un nombre de noeuds très superieur et renvoie un nombre de poids beaucoup plus important.
Les poids posés sur des pierres seront par la suite annulés par le sytème qui gère les dictionnaires.

5.3.3 Amélioration : pierres "importantes"

Dans certaines situations, il arrive que des pierres soient considérées comme non-importantes, et qu'elles ne doivent pas être prises en compte lorsque l'on fait une recherche dans une zone. Pour cela, le dictionnaire bénéficie de deux outils : le "terrain important", et les intersections "vides ou noires" et "vides ou blanches". Lorsque l'on arrive au bout d'une branche (plus aucune sous branche ne correspond à un des coups de la forme donnée), on regarde les pierres voisines restantes. Si elles sont hors de la zone d'importance dans la position courante de la branche, ou qu'elles arrivent sur une case marquée comme devant être de leurs couleur ou vide,

la position reste validée comme une forme dont il faut tenir compte.

5.3.4 Amélioration : zones spéciales

Certaines formes ne sont valables que dans certaines zones (les coins ou les bords), et pas dans les autres. Après une réflexion sur la façon d'indiquer cette information dans un dictionnaire, nous avons décidé de faire un dictionnaire par type de formes. Nous avons donc le dictionnaire de joseki pour les coins, le dictionnaire "formesBord", pour les formes sur les bords, et le dictionnaire "formes" pour les formes standard, valables n'importe où sur le goban. En plus de ces trois dictionnaires, nous avons un dictionnaire de fusekis, qui nous permet de faire jouer à notre programme les ouvertures théoriques.

5.3.5 Le dictionnaire par base de données de parties

Bien que par choix, nous n'ayons pas implémenté ce système, nous tenions à présenter cette solution qui est très différente de ce que nous avons fait. Une idée développée par beaucoup d'autres programmeurs a été de faire une base de données géante contenant des milliers de parties professionnelles, puis d'extraire des positions locales au milieu des parties ressemblant à des positions locales de la partie jouée par le programme. Ce système a pour avantage de ne pas avoir à remplir manuellement des dictionnaire, et possède donc un dictionnaire beaucoup plus étoffé que le système que nous avons choisi. Mais ce système a deux défauts : Le premier est qu'il est impossible lorsque le programme fait un mauvais coup de "corriger" la base de données. Le deuxième est que lorsque le programme se retrouve face à un joueur faible, sa base de donnée qui ne contient que des "bonnes formes" ne trouve aucune forme et ne pourra pas "punir" les erreurs de son adversaire.

5.4 Intégration à notre programme

Après chaque coup adverse, le programme regarde quel poids renvoie le dictionnaire de forme pour chaque pierre posée sur le goban. Pour les pierres qui ne sont pas de sa couleur, il ajoute un coup vide (représenté par "pass") à la fin de la liste de coups. Ainsi, il connaît à la fois ses coups de formes defensifs grace aux formes renvoyées par ses pierres, mais aussi ses coups de formes offensifs grace aux formes renvoyées par les pierres adverses. Le dictionnaire de formes renvoie pour une forme donnée tous les coups de formes suivants possibles, et le poids qui leur est associé. Après avoir fait pour chaque case la somme des poids renvoyés par les formes, il connaît le meilleur coup de forme. Mais les dictionnaires de formes servent aussi pour d'autre parties de l'intelligence artificielle, avec un dictionnaire de josekis, et un dictionnaire de fusekis.

5.5 L'interface graphique

L'interface graphique pour la modification des formes devait être pratique et surtout adaptée à nos besoins spécifiques. Bien que les formes soient stockées dans un format standard qui possède plusieurs éditeurs, il nous fallait un éditeur personnalisé pour avoir un accès rapide et intuitif aux symboles que nous utilisions pour indiquer des données supplémentaires (limite d'une forme, ko potentiel, shisho, case vide ou noire, case vide ou blanche).

Nous avons donc décidé de modifier un éditeur existant pour y intégrer les fonctionnalités supplémentaires dont nous avions besoin. Après avoir regardé les fonctionnalités et le code source des principaux programmes existants, nous avons découvert un programme méconnu, mais open source, écrit en C++, utilisant la dernière version de QT, et étant en développement actif. Etant la seule à répondre à tous ces critères, nous avons décidé de la choisir, malgré son côté "underground" (un seul développeur, très peu référencé par google, et relativement nouveau, donc inconnu dans la communauté des joueurs de Go). Puis nous avons rajouté divers boutons pour les symboles dans l'interface, et une fonction qui délimite une zone si on lui donne tous les coins de la zone à sélectionner. Au final, nous avons une interface facile à utiliser et très évolutive.

Chapitre 6

UCT

Sommaire

"UCT tend vers le meilleur coup"
Olivier Teytaud

6.1 L'algorithme UCT

6.1.1 Présentation d'UCT

Monte Carlo

L'algorithme de Monte Carlo au Go pourrait se résumer de cette façon : on joue "au hasard" un grand nombre de parties, en commencant autant de partie par chacune des intersections jouables du Goban, et on comptabilise la proportion de partie gagné. Puis on joue sur l'intersection ayant eu le meilleur ratio de victoires. Quelques programmes ont tenté sans trop de succès d'utiliser Monte-Carlo pour le jeu de Go. Cela n'a pas du tout marché : il suffisait qu'un coup soit très mauvais si l'adversaire répondait bien, mais très bon dans les autre cas, pour que Monte-Carlo le considère comme probablement bon, car la réponse choisi par l'adversaire étant aléatoire, elle sera peu testé avec le bon coup adverse. D'une manière générale, finir les parties de manière complètement aléatoire ne donnait pas une indication fiable sur la qualité du coup.

L'algorithme UCT est extrêmement puissant, car il est polyvalent, et nécessite extrêmement peu de connaissances a priori sur le domaine sur lequel on va l'utiliser. Par exemple, il peut être codé pour le jeu de Go juste en ayant lu les règles. Son but est d'explorer partiellement les branches de l'"arbre des possibles", en essayant de se concentrer sur les plus intéressantes. Après avoir exploré une partie "suffisante" de l'arbre, il choisit la branche qui semble donner statistiquement les meilleurs résultats.

Bandit ou Comment choisir les coups à explorer

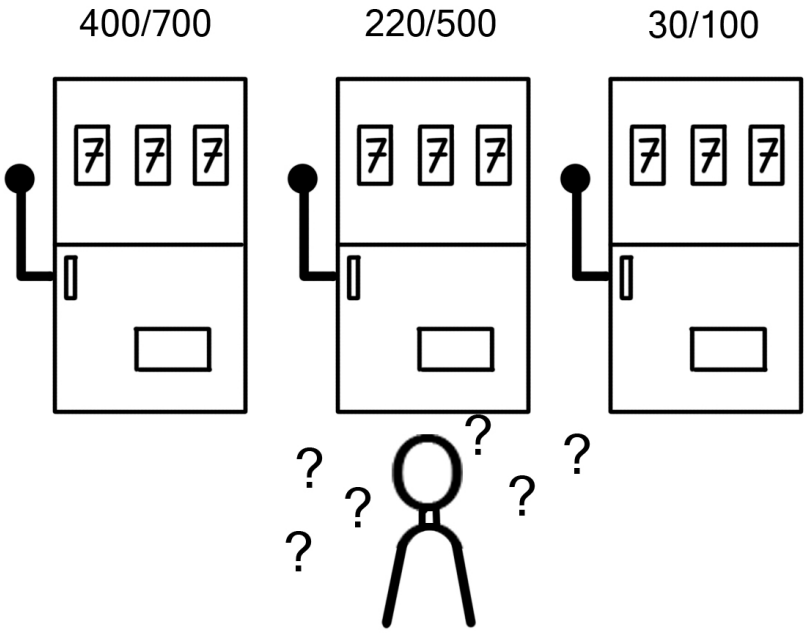
UTC est basé sur l'algorithme de Bandit. C'est un algorithme utilisé dans des domaines très variés, demandant une fonction d'estimation des risques, ou de maximisation du profit. Un exemple d'utilisation, dont l'algorithme tire son nom, est celui du jeu optimal face à une série de machines à sous. Un joueur se trouve face à n machines à sous, il possède x jetons (un nombre de jetons très supérieur au nombre de machines à sous), et il veut optimiser son gain. On part du principe que les machines à sous ont toute une probabilité différente de gagner qui n'évolue pas au cours du temps. Le but est donc de trouver cette machine pour y jouer un maximum de jetons.

L'idée de base est, comme pour Monte Carlo, de tester avec un certain nombre de jetons équivalent toutes les machines, et de jouer sur celle ayant offert le plus de victoire pendant la période de test. Bandit propose quelque chose de plus fin : L'idée est de jouer en priorité sur les machines ayant le plus de chance d'être la machine offrant le meilleur gain, pour affiner leurs statistiques, et petit à petit trouver la machine ultime. L'algorithme indique donc qu'il faut commencer par tester toutes les machines, puis se concentrer petit à petit sur celles qui nous ont offert jusque-là les ratio les plus intéressants, tout en jouant de temps en temps sur les machines sur lesquels on a très peu joué, pour ne pas risquer de sous-estimer une machine à cause d'une statistique trop floue dûe au manque de tests. Pour avoir le risque le plus faible dans le pire des cas, il faut prendre la machine m ayant le meilleur résultat à la formule :

$$\text{Moyenne empirique pour m} + \frac{\sqrt{log(\text{nombre de jetons total déjà joués})}}{\text{nombre d'essais avec m}}$$

Ainsi, sans avoir aucune information a priori lorsque l'on commence à jouer les jetons, on joue statistiquement de manière de plus en plus optimale au fur et à mesure des jetons joués.

L'algorithme fonctionne de la même manière dans la plupart de ses applications : Le but est toujours de faire des tirages aléatoires d'évènements après avoir modifié une ou plusieurs variables initiales, et de trouver quelles valeurs donner aux variables initiales pour maximiser le gain statistique, en orientant les tirages aléatoires pour avoir le résultat le plus précis possible sur les variables a priori les plus intéressantes.



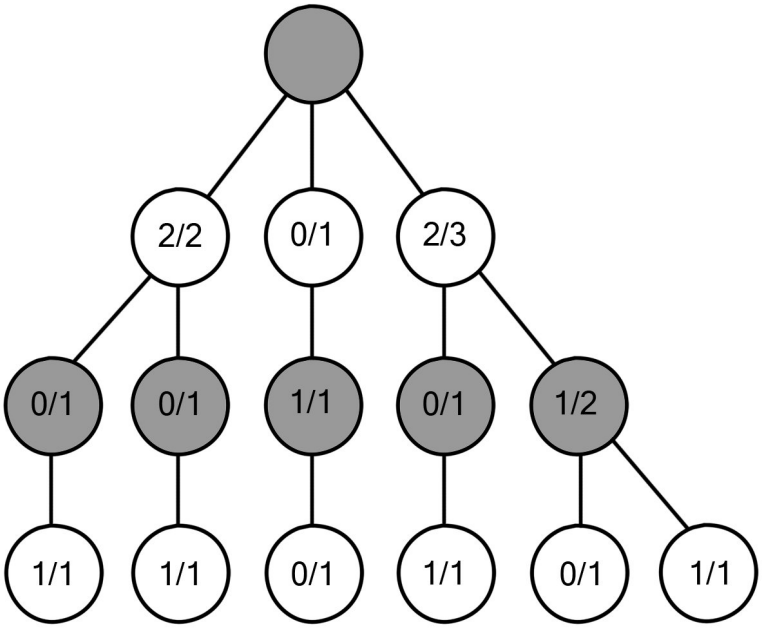
UCT ou Bandit dans un arbre

Monte-Carlo peut trouver des coups correct au niveau stratégique, mais est incapable de jouer correctement au niveau tactique, des qu'il faut lire une série de coups. Le fait de choisir au hazard les coup pour finir les parties empechent de regarder des séquences de combat cohérentes. UCT est beaucoup plus puissant car il utilise Bandit à chaque étage de l'arbre. Les coups de l'adversaire ne sont plus aléatoire, car lui même explore en priorité les branches qu'il a peu explorées ou celles qui lui donnent le plus de victoires possibles. Ainsi, les parties jouées sont plus réaliste, et les statistiques plus justes. De plus, les coups qui ont un meilleur ratio pendant que tourne l'algorithme sont plus éxplorés, et le programme a donc plus d'informations sur eux, et une probabilité plus juste des chances de victoire.

Exemple de fonctionnement

L'arbre des possibles est l'arbre de toutes les parties possibles depuis une position donnée. Chaque noeud représente un coup, dans une position donnée (sauf le noeud racine). Chaque noeud connaît son coup, son nombre de passages et son nombre de victoires. Chaque descente vers un prochain noeud est choisi en fonction de la fonction donné par Bandit. Une fois descendu sur un noeud, on met le goban à jour. Une fois arrivé en bas, on regarde le score, on remonte la branche en incrémentant le nombre de victoire des coups de la couleur gagnante, et le nombre de passages de toutes les pierres. Puis on réinitialise le goban à sa position initiale. A la suite d'un

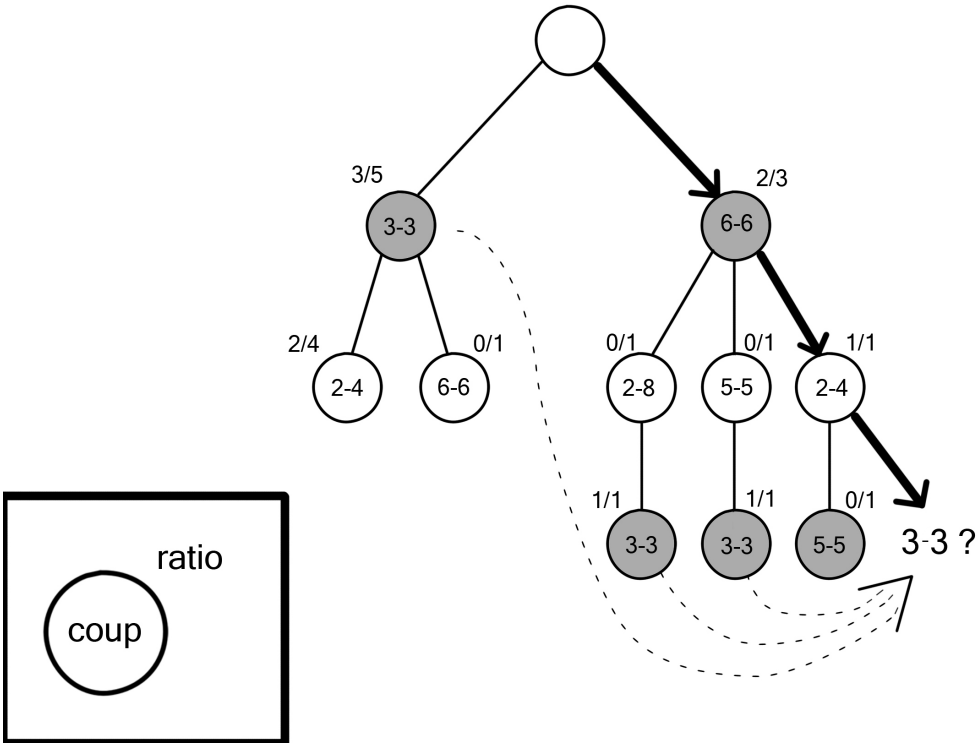
grand nombre de passages, on choisit le noeud ayant le meilleur ratio dans l'arbre, et on joue le coup qui lui correspond.



6.2 Variantes et améliorations

6.2.1 RAVE

RAVE est une grande amélioration pour UCT, surtout dans les cas ou l'on arrive sur des noeuds encore non ouverts, ce qui représente la majorité des descentes que l'on fait. Il part du principe que si un coup est très bon dans les autre branches, il l'est aussi dans celle que l'on est en train d'explorer. Il permet d'explorer en priorité le coup jouable ayant le meilleur ratio toute branche confondu, lorsque l'on n'a aucun a priori sur les noeuds. Il peut aussi être utilisé comme un complément à l'éstimation donné par bandit : Le score d'un noeud n peut être donné comme (a x Bandit ?(n)) + ((1 - a) x Rave(n)))



6.2.2 Le partage des calculs

UCT est extrêmement gourmand en ressource, que ce soit au niveau calcul (processeur), qu'au niveau mémoire (RAM). Il est donc important de disposer d'une grande puissance de calcul alliée à une grande capacité mémoire pour obtenir des résultats efficaces. Mais UCT est relativement facile à adapter à du calcul partagé. Plusieurs stratégies sont envisageables :

- Différentes tâches peuvent explorer différentes branches.
- Chaque tâche peut construire son arbre, puis les arbres peuvent être fusionnés à différents moments. La fusion peut se faire sur la totalité de l'arbre, ou juste sur les noeuds considérés comme étant les plus intéressants.
- Chaque tâche peut construire son arbre de manière purement indépendante, puis partager à la fin les statistiques de chaque noeud.

6.3 Notre implémentation

6.3.1 Implémentation d'UCT

UCT en lui même a été codé de manière assez proche de la théorie. Mais quelques choix d'implémentations ont dû être faits.

- Le format des noeuds : Les noeuds ont étés codés pour être les plus légers possibles. En effet, nous arrivons très vite à plusieurs dizaines de millions de noeuds, et ce nombre croît de manière quasi-proportionnelle au nombre de descentes. Les noeuds saturent donc très vite la mémoire vive, et il nous les fallait les plus légers possible. Un noeud est donc constitué : d'un pointeur sur son premier fils, d'un pointeur vers son prochain frère, et d'un pointeur vers son père. (On retrouve tous les fils d'un père en cherchant récursivement tous

les frères du premier fils). Il possède en plus de cela le nombre de passages, le nombres de victoires (codé sur des short), et le coup qui lui correspond (codé par trois char représentant la couleur, l'abscisse et l'ordonnée de la pierre joué à ce noeud.) Il possède aussi le "score" du noeud (représenté par un short), qui prend une valeur arbitraire si le noeud n'est pas une fin de partie, ou le score si la partie est finie au moment du noeud. Cela fait au total 21 octets pour une architecture 32 bits classique. Pour descendre en-dessous, il faudrait sacrifier certaines données ce qui obligerait le programme à faire des calculs complexe pour les retrouver, et ralentirait l'execution.

- La construction des noeuds : Lorsque UCT doit choisir un nouveau noeud, il doit regarder itérativement les noeuds existants, et choisir celui qui a le meilleur ratio. Une première implémentation vérifiait tous les coups possibles à chaque fois qu'il arrivait à un noeud, puis si plusieurs d'entre eux n'étaient pas explorés, en choisissait un au hasard. Cette implémentation était très gourmande en ressource, car l'ensemble des coups possible devait être recalculé à chaque passage. L'implémentation finale construit tous les noeuds fils possibles d'un noeud exploré pour la première fois. De cette façon, à chaque nouveau passage sur ce noeud, les noeuds fils possibles sont déjà construits. Cette implémentation est beaucoup plus gourmande en mémoire, car une énorme partie des noeuds créés ne sera jamais explorée, mais elle est beaucoup moins consommatrice en calculs et s'est avérée beaucoup plus rapide.
- La formule de Bandit : La formule de Bandit donne théoriquement une priorité infinie à un noeud jamais exploré, (car elle contient le ratio nombre de passages / nombre de passage par le noeud). Cela permet d'être sûr que tous les noeuds sont explorés au moins une fois. Mais cela posait problème dans les étages bas de l'arbre : En effet, il y a très peu de chances que tous les noeuds fils soient explorés, et il est dommage d'explorer au hasard d'autre noeuds lorsque l'on en possède un avec un bon ratio, et que nous aurons peu de chance de pousser loin nos statistiques sur les noeuds enfants. Nous avons donc mis au point une formule tenant compte des étages. Plus on descend dans l'arbre, moins la différence ratio - 0.5 a besoin d'être grande pour suffir à ne pas choisir un autre noeud. Cela sera probablement moins efficace sur de grosses machines capables d'explorer une partie conséquente de l'arbre des possibles, mais nous a semblé plus efficace sur nos machines personnelles. Le gain en efficacité nous a semblé diminué une fois RAVE implémenté.

6.3.2 RAVE

Rave a été codé sous forme d'une liste chaînée triée mémorisant le ratio de chaque coup sur le goban, quelle que soit sa position dans l'arbre. RAVE a été utilisé de plusieurs façons durant les tests. A la base pensé uniquement pour choisir la branche à explorer lorsque l'on se retrouve sur un noeud exploré pour la première fois, il s'est avéré tout aussi efficace pour influencer le choix proposé par Bandit. Au final, il est même utilisé pour le choix final du meilleur coup dans la position donnée.

6.3.3 Le multi-threading

Le programme a un paramètre méta défini "Nombre de threads" qui peut être modifié selon la machine sur laquelle on le fait tourner. Chaque thread fait ses propres descentes dans une des branches du même arbre. Cette branche est protégée par un mutex pour ne pas être modifiée par les autres threads, qui devront en

choisir une autre. De plus, les threads possèdent des pointeurs vers les deux vecteurs RAVE originaux, et des copies de ceux-ci. Lorsqu'un thread a fini une descente, il regarde si le mutex protégeant le vecteur RAVE original est libre. Si c'est le cas, il donne les informations de ses précédentes descentes contenues dans le vecteur copie au vecteur principal, puis regénère son vecteur copie à partir du vecteur principal. Sinon, il repart dans une descente, et mémorise les informations de la descente pour pouvoir mettre le vecteur principal à jour.

6.3.4 Le cloud computing

Le programme peut utiliser un mode réseau, pour partager les calculs à effectuer entre plusieurs machines. Mais contrairement au multi-threading, où chaque thread agit sur les même variables que les autres, les machines n'ont pas de mémoire commune et doivent se passer des données. Faire passer tout l'arbre d'un ordinateur aux autres de manière régulière étant hors de question pour ne pas ralentir le programme, l'arbre pesant plusieurs Giga, il nous fallait trouver une meilleure solution de partage. RAVE a été le premier outil à être partagé en réseau : En effet, il est léger, donc rapide à transférer et offre une information d'une importance cruciale pour le programme. Toutes les 100 descentes, le programme se met en écoute sur le réseau en mode serveur, récupère les listes chaînées données par les ordinateurs en mode client, les fusionne avec sa liste, en faisant la moyenne de toutes les listes pour chaque maillon, puis renvoie la liste reconstituée aux clients. Ainsi, tous les ordinateurs ont une nouvelle liste, moyenne des listes des autres ordinateurs, pour faire leurs prochaines descentes. En plus de cela, la version réseau permet de transferer le premier étage de l'arbre. Pour l'instant, sa moyenne n'est faite qu'à la fin des descentes, au moment du choix du coup, pour garder des ratios cohérents entre père et fils pendant la descente. Mais il est tout à fait possible, à condition de changer légèrement l'agorithme de choix des maillons, de faire des moyennes du premier étage de manière régulière, comme on fait les moyennes de la liste chaînée RAVE.

Chapitre 7

Outils d'amélioration

Sommaire

"Le jeu joue de lui meme, le joueur ne le controle pas."
Audouard, Pierre

7.1 Amélioration par le programmeur.

Nous avions prévu d'intégrer un module permettant de retracer le cheminement du programme jusqu'au coup choisi. Un début d'implémentation utilisant la programmation par aspect indique quelles fonctions du système de règles ont été utilisées. La programmation par aspect permet d'écrire un code indépendant du code principal, qui réagit à son exécution sans altérer son fonctionnement. Dans la suite, le module aurrait aussi du pouvoir indiquer à chaque coup quels formes avaient été repéré par le dictionnaire. Le fichier généré devait pouvoir être lu dans une version modifié de l'interface graphique "Mugo" pour afficher les formes et les règles qui ont permis d'arriver au résultat. Cela aurrait permis à un expert de voir plus facilement les erreurs de sa base de règles et de ses dictionnaires de forme. Cette partie pourra cependant être implémentée par la suite, dans le cadre de la réutilisabilité de l'outil de création d'Intelligence Artificielle.

7.2 Auto-Amélioration ou apprentissage.

Dans nos objectifs initiaux, nous voulions donner une capacité d'apprentissage au programme. Bien que nous n'ayons pas eu le temps de l'implémenter, le programme est conçu pour pouvoir accepter une couche d'apprentissage.

7.2.1 Les dictionnaires

Mugo, le programme qui nous a servi à transformer les fichiers SGF en arbre, possède aussi un algorithme pour enregistrer ses arbres au format SGF. Nous pensions donc faire en sorte de donner une capacité d'apprentissage au dictionnaire.

Création de nouvelles formes

Pendant une partie, le dictionnaire peut être amené à rencontrer des formes qu'il ne connaît pas. Si l'on part du principe que son adversaire est meilleur que lui, le programme aurait intérêt à mémoriser les formes inconnues qu'il rencontre pour les réutiliser lors des parties suivantes. Cela pose néanmoins plusieurs problèmes. Pour ne pas alourdir le dictionnaire de cas uniques, il aurait fallu qu'il soit capable de décider à partir de quand la forme n'a plus d'intérêt et quelles sont les pierres pertinentes qui la composent.

Evolutivité des poids

Le poids d'un coup dans une forme est une valeur assez subjective, et il aurait été intéressant que le programme soit capable de le modifier. Pour cela, il aurait fallu qu'il puisse estimer à chaque nouvelle position le score de la partie, et modifier à posteriori les poids. Le meilleur outil d'estimation dont nous disposions était UCT. Vu les défauts qu'il possède, nous n'avons pas trouvé sage de lui donner le pouvoir de modifier le dictionnaire.

7.2.2 Le système expert

Un système expert n'est pas a priori fait pour avoir une couche d'apprentissage. Toutefois, une couche d'apprentissage peut proposer des faits ou des variations de poids qui peuvent être validés ou non par l'expert. Faire générer des fait au programme est quelque chose de très complexe, mais lui faire modifier ses poids nous

semblait réalisable. Malheureusement, cela nous ramène au problème de l'évaluation d'une position, ce que nous ne pouvions pas faire de manière assez satisfaisante. Mais le système de règles, écrit dans un fichier texte, peut permettre ce genre d'évolutions.

7.3 Apprentissage controlé

Dans le but de garder une capacité de controle forte sur la façon de jouer de l'Intelligence artificielle, la meilleure stratégie aurrait été de permetre à un expert de controler l'apprentissage de l'intelligence artificielle. Il aurrait donc fallu créer un outil capable de donner à l'expert la capacité d'analyser ce que le programme cherche à apprendre. L'idée générale serait de garder le système proposé dans le chapitre de l'amélioration par le programmeur, mais de rajouter un sous-module qui permetrait à l'expert de voir ce que l'intelligence artificielle a pensé avoir compris durant la partie. L'expert pourrait ensuite choisir de valider ou non les nouvelles connaissances qu'elle propose.

Chapitre 8

Au coeur de l'Intelligence Artificielle

Sommaire

"Une voie directe menant au cœur du secret de l'univers, dans les échanges de l'inspiration et de l'expiration, du ciel et de la terre, du Yin et du Yang. Une philosophie dont le jeu et l'ambition étaient d'approcher de la perfection, de l'être pur, de la réalité pleinement accomplie."

Hermann Hesse

8.1 Comment sont agencées les différentes solutions.

8.1.1 Pour le goban en 19 x 19

Les idées

Le goban de taille 19 par 19 est le goban de taille standard, c'est celui sur lequel se font toutes les parties officielles, et c'est aussi celui sur lequel les intelligences artificielles ont le plus de mal. C'est pourtant celui que nous avons choisi pour faire nos premiers tests, et celui sur lequel nous nous sommes le plus penchés pendant la première partie de notre projet. Dans un premier temps, notre programme ne jouait dessus qu'en utilisant le dictionnaire des formes, avec un dictionnaire des josekis, et une initialisation de la valeur des cases clés du début de partie. De cette manière, il pouvait jouer une douzaine de coups correct avant de se retrouver face à des problèmes qu'il ne savait pas gérer. En venant épauler les dictionnaires, le système expert a permis une grande avancée. D'abord parce que les valeurs des points clés du goban étaient réinitalisés d'un coup sur l'autre, en tenant compte de l'évolution de la situation. Ensuite, parce que le programme, après avoir joué les formes qu'il connaissait, ne se retrouvait pas complètement démuni.

L'implémentation

D'abord, le dictionnaire des forme est appelé sur toutes les pierres, pour obtenir un premier tableau de poids. Puis ensuite, le système expert envoi un deuxième tableau de poids, en utilisant d'autre dictionnaires telle que celui des josekis. Une fusion des deux tableaux de poids est faite, et le coup joué est le meilleur coup du tableau.

8.1.2 Pour le goban en 9 x 9

Les idées

Le goban en 9x9 est celui sur lequel nous avons réussi à faire tourner UCT/RAVE en ayant des résultats corrects. C'est donc celui sur lequel le programme joue le mieux, mais aussi celui sur lequel les connexions entre les algorithmes sont les plus complexes. Nous avons fait jouer plusieurs parties au programme avec seulement UCT/RAVE. Il en est ressorti des coups naïfs, où le programme jouait des coups qui donnaient l'impression qu'il s'attendait à une grosse erreur de son adversaire. Cela est en partie dû au manque de puissance des ordinateurs sur lesquels ils ont été téstés. En tentant un coup naïf nécessitant obligatoirement une bonne réponse de l'adversaire, l'ordinateur pouvait faire beaucoup de victoires sur les descentes, jusqu'à ce que ce coup remonte dans les statistiques de l'adversaire, ce qui peut prendre un certain temps, et donner un bon ratio au coup naïf du programme. Cela peut être renforcé par Rave si le coup de défense de l'adversaire est inutile voir mauvais dans les autre branches. Avec un ordinateur plus puissant, UCT pourrait faire beaucoup plus de tests, donc les descentes faites avant que le ratio de la bonne défense augmente représenteraient une plus petite partie du nombre de descentes final, beaucoup plus seraient faites avec la bonne réponse adverse, et au final, le coup naïf verrait son ratio redescendre. Parallèlement à cela, UCT ratait beaucoup de coups simples, visibles au premier coup d'oeil par un joueur moyen, car répondant au fait de faire une belle forme. Il suffisait que ces coups fassent quelques mauvaises descentes (ce qui n'est pas statistiquement aberrant) pour être délaissés. Guider UCT avec le dictionnaire des formes nous a donc semblé important. Après une

deuxième série de tests avec ce duo, sous différentes implémentations, nous avons vu certaines erreurs corrigibles grace à quelques concepts simples. Nous avons donc rajouté au dessus du duo le système expert, et avons écrit une série de règles et les fonctions qui vont avec spécialement pour le 9x9 (les règles du 19x19 n'étant pas du tout adaptées au 9x9).

L'implémentation

L'idée la plus simple est de faire une moyenne des tableaux en faisant tourner les algorithmes séparément,puis de faire la moyenne des poids renvoyés par chacun pour choisir un coup, comme pour le 19x19. Mais il nous semblait que si nous pouvions guider UCT pour qu'il fasse en sorte de faire ses descentes en priorité vers les points remarqués comme important, il renverrait des résultats plus intéréssants. Nous commençons donc par modifier les poids du Rave grâce aux poids donnés par le dictionnaire des formes, et le système expert. Ensuite, comme les poids de Rave sont modifiés, UCT explore en priorité les coups donnés comme intéressant par le premier duo. Comme il fait peu de descentes sur les autre coups, il peut arriver que certains aient à la fin un ratio surestimé, car exploré un nombre trop faible de fois. Pour palier à ce manque, Les poids renvoyés par UCT sont légèrement modifiés par les poids donnés par le dictionnaire de forme et le système de règles.

8.2 Quelles sont les pistes à explorer.

8.2.1 Le système expert

Un résolveur de Tsumegos

Toute la partie combat pourrait être gérée directement par UCT, mais celui ci ne peut que difficilement s'intégrer aux règles, et se trouve au final plus adapté aux problèmes stratégiques que tactiques. Une fonction capable d'indiquer le potentiel de vie d'un groupe et les solutions pour menacer cette vie donnerait au système expert, et au programme dans son intégralité, un potentiel beaucoup plus grand. En effet, pour l'instant, le programme a un niveau correct en stratégie, mais la partie tactique nous aurait probablement demandé un semestre de plus pour être réalisé. Notre programme reste toutefois très modulable, et l'insertion d'un algorithme de ce type, que ce soit directement dans le code, ou grâce à une série de règles écrite pour le système expert est tout à fait envisageable.

Un algorithme MinMax

Le système expert se base pour choisir son coup sur l'état actuel du goban sans aucune prévision de ses états futurs. L'idée serait de séléctionner les meilleurs coups proposés par le système expert. Puis, de regarder pour chaqun d'eux quelles seront les valeurs des meilleurs coups pour son adversaire. Ensuite pour les meilleurs d'entre eux, regarder les meilleurs réponses de l'intelligence artificielle et ainsi de suite jusqu'a une profondeur déterminée. On construit ainsi un arbre des possibilités d'évolution à partir de l'état courant du goban. Cet arbre est limité par une profondeur et une largeur données.

On récupère la branche de l'arbre ayant les meilleurs coups pour le programme en considérant que le joueur a choisi ses meilleurs coups de la même façon. On maximise le gain en considérant que l'adversaire maximise le sien en utilisant le


même algorithme. Ainsi, on peut choisir le coup en fonction des évolutions possibles de la partie et non uniquement par rapport à son état courant.

8.2.2 Les dictionnaires

Une reconnaissances des formes en fonction de l'influence

Alors que nous avions prévu de le faire, le manque de temps nous a empêché de définir pour chaque forme si elle était bonne en fonction de qui maîtrise l'influence de chaque côté de la forme. Il nous aurait pour cela fallu définir une fonction d'influence sur le goban, pour décider pour chaque pierre qui avait l'influence de chaque côté. Ce sont des fonctions compliquées qui demandent de faire des choix subjectifs et qui nécessitent énormément d'heures d'implémentation. Toutefois, cela aurait grandement amélioré le niveau de notre programme, surtout sur les grands plateaus.

Une reconnaissance de formes "floues"

Le dictionnaire de formes ne reconnait que les formes qui sont strictement dans le dictionnaire. Si une forme proche est présente, il ne la considère pas. Nous avons volontairement écarté ce type d'évolution, car les résultats peuvent être hasardeux, et difficile à corriger et à améliorer. Nous préférions garder au maximum le contrôle sur le fonctionnement du dictionnaire, quitte à le laisser un peu rigide.

8.2.3 UCT/RAVE

Nous avons implémenté l'algorithme UCT/RAVE avec la plupart des options possibles, et plusieurs modifications pour l'adapter à la puissance des machines sur lesquels on le faisait tourner et à l'intégration dans le reste du code. Toutefois, nous avons eu plusieurs idées d'amélioration qui impliquent des changements importants dans l'algorithme de base, que nous n'avons pas eu le temps d'implémenter, mais qui seraient des pistes qui nous semblent intéressantes à explorer.

L'importance d'une descente

UCT considère que toutes les descentes sont équivalentes. Il lui arrive donc de choisir au final des noeuds qui ont eu un bon ratio, grace à des victoires au début quand UCT ne connaît que peu de noeuds suivant et que RAVE est construit à partir d'un échantillon de données trop petit. L'idée la plus simple pour remédier au problème est de choisir des noeuds qui ont un certain nombre de descentes. Mais cela ne règle pas vraiment le problème : les descentes aléatoires du début ont autant de poids que les dernières qui sont pourtant choisies avec une quantité d'information largement supérieur. Le fait de faire augmenter linéairement l'importance d'une descente en fonction du temps ne suffit pas non plus à rendre compte de ce paramètre, car certains noeuds réexplorés tardivement, car ayant eu de mauvaises descentes aléatoires, auront toujours peu de noeuds descendants, et leur exploration tardive ne traduira pas une augmentation de l'information. La meilleure façon de traduire l'importance d'une descente est donc surement de la définir en fonction du nombre de noeuds déjà visités présents sur le parcours.

La vitesse du choix du noeud

Pour l'algorithme UCT/RAVE, le choix du noeud suivant, lorsque les noeuds fils existent déjà est le calcul prenant le plus de ressources processeur, d'abord parce

qu'il demande à comparer les ratios de tous les noeuds fils, mais aussi parce qu'il est fait plusieurs fois par descentes. Une accélération de ce choix, quitte à le rendre moins précis serait quelque chose de très intéressant, car cela permettrait sur des machines assez riches en ram de faire plus de descentes sans augmenter le temps de calcul. Une des idées les plus simples serait d'avoir une liste de maillons ordonnée. Les maillons seraient construits en copiant la liste donnée par RAVE au moment de la découverte du noeud père (en vérifiant que ce sont des coups jouables). Puis on choisit le premier noeud. S'il est gagnant, il reste en première place. S'il est perdant, il prend une place dans la liste de noeud correspondant à son ratio. Par exemple, si son ratio est de 3/4, il se placera après le premier quart des noeuds fils. Cette méthode possède toutefois plusieurs inconvénients. Le premier est une perte dans la précision du calcul, puisqu'il n'y a plus de comparaisons précises de ratios. La deuxième est que le vecteur du vecteur donné par RAVE est fixe. Ainsi, les noeuds choisis en premiers ont dans cette méthode un gros inconvénient, car le vecteur RAVE qui sert à fabriquer leurs fils a peu d'information et sera peu précis. Il faudrait éventuellement, en fonction du temps écoulé entre la construction et un passage, retrier la liste de fils en fonction du vecteur RAVE mis à jour. Malgré ces défauts, il y a probablement une piste intéressante derrière cette idée. Elle n'a toutefois pas été implémentée car la RAM était déjà le facteur limitant sur les ordinateurs qui nous ont servis à faire les tests, et la rapidité du temps de calcul n'a donc pas été notre priorité.

Un RAVE plus précis

RAVE considère que les intersections intéressantes à certains endroits le sont sûrement aussi à d'autres. Mais dans plusieurs cas, certains coups ne sont bons uniquement que face un certain coup adverse. On pourrait donc considérer le fait de mémoriser les bons coups par rapport à un noeud père concernant tous ses fils. Si cela se faisait sur tous les noeuds, cela serait long à mettre à jour au moment de la victoire ou défaite d'une descente, et surtout, prendrait énormément de RAM. Mais le faire sur les premiers étages pourrait être intéressant. Une autre variante consisterait à mémoriser les bons coups à partir du moment où un coup adverse précis est joué. Cela ne ferait que rajouter 2xn vecteurs, où n est le nombre d'intersections du plateau. Le temps perdu causé par la mise à jour de tous les vecteurs, et du calcul du meilleur coup suivant en fonction des pierres adverses serait probablement compensé par un gain énorme en qualité des descentes, car les pierres posées par l'adversaire sont un indicateur extrêmement pertinent pour choisir un coup. L'algorithme s'éloignerait beaucoup de UCT mais le tester pourrait être extrêmement intéressant, d'autant que si l'idée marche, elle serait sûrement aussi très efficace sur d'autre problèmes que sur le jeu de Go.

8.2.4 Le programme dans son intégralité

Une couche d'apprentissage

Beaucoup de programmes d'intelligence artificielle ont la capacité d'apprendre. Notre programme est fait pour être capable de recevoir une couche d'apprentissage. Cette partie des évolutions possibles est détaillé dans le chapitre précédent qui lui est dédié.

Une meilleure intéraction entre les algorithmes

Les algorithmes ont encore des interactions entre eux assez faibles. Or il pourrait y avoir une très forte synergie entre eux. Par exemple, le système expert pourrait choisir le nombre de descentes d'UCT et son importance dans la décision finale en fonction de certains paramètres propres. (Est-il en danger, y'a-t-il des groupes faibles, en est-on au Yosé ...). Ensuite, il pourrait lui interdire certaines cases (que ce soit pour RAVE ou pour le premier étage de l'arbre UCT), ou le guider vers les cases qu'il considère comme les meilleures (ce qui est déjà fait indirectement dans notre implémentation). Dans l'autre sens, UCT pourrait se servir du système expert (avec une série de règles légères) pour se guider dans les premiers étages des descentes, et explorer en priorité les cases données gagnantes par le système dans chaque position donnée.

Une version éducative.

En plus des améliorations à ajouter à ses divers composants, le programme manque de certaines fonctionnalités que nous aurions aimé lui donner. Une des plus importantes est sans doute la correction des coups de ses adversaires : il faudrait que le système de règles puisse prévoir les coups possibles de son adversaire, et lui indiquer pourquoi ils sont bons, et lui indiquer pourquoi certains autres sont mauvais. Cela nécessiterait de rajouter une phrase à chaque règle, de faire calculer au système les coups adverses possibles, et de lui donner la possibilité de comprendre à partir de quelles règles il a fabriqué ses coups (un forward chaining par exemple). Il est peut-être aussi important pour un programme d'être éducatif que d'être fort. Nous voyons aux échecs que beaucoup de programmes ont atteint un très haut niveau, mais ils n'intéressent que peu de monde car les humains sont frustrés de perdre contre eux, et voient rarement l'intérêt de perdre de manière systématique contre un programme.

Chapitre 9

Conclusion

Sommaire

"En cas de danger, abandonnez quelque chose."
Youyi Chen

9.1 Les erreurs que nous aurions dû éviter.

Durant le projet, nous avons commis plusieurs erreurs plus ou moins graves. Malgré cela, le projet s'est dans son ensemble bien passé. Mais nous voulions tout de même revenir sur les les problèmes rencontrés durant le projet, étape indispensable pour progresser.

9.1.1 Une confiance trop grande au début

Lorsque l'on a commencé le projet, très motivés face aux défis qui nous attendaient, nous voulions faire une intelligence artificielle capable de jouer des parties complètes sur toute les tailles de plateau, et d'indiquer ses erreurs à son adversaire. Suite aux nombreux problèmes que nous avons rencontrés par la suite, mais aussi à cause de notre manque de lucidité face à l'ampleur de la tâche, nous avons été contraints au cours du semestre de revoir nos objectifs à la baisse.

9.1.2 Un départ trop rapide

Au moment de commencer le projet, nous avions une idée assez clair de ce que nous voulions : un programme capable de jouer "comme un humain". Nous avons donc réfléchi aux algorithmes qui nous donneraient le plus de chance de se rapprocher de l'objectif, et nous avons choisi de travailler sur la reconnaissance de formes et la création d'un système expert. Dès la première semaine de travail, tout en commençant à lire les thèses faites sur le sujet, nous avons commencé à coder notre goban, et deux semaines plus tard, nous attaquions déjà le dictionnaire, prenant conscience que le semestre serait court pour remplir l'objectif fixé. Finalement, au fur et à mesure que nous faisions le point sur notre avancement, nous nous rendions compte que nous aurions dû passer plus de temps sur ce que nous attendions de chaque morceau du programme, et sur les évolutions que nous pourrions avoir envie de lui apporter. Nous ne regrettons pas d'avoir commencé le code assez tôt, car le temps nous a vraiment manqué sur ce projet, mais nous nous rendons compte qu'une réflexion plus poussée, et une plus grande documentation sur l'existant, au début de chaque partie, aurait permis une meilleure évolutivité de certaines fonctionnalités.

9.1.3 Des erreurs dans le choix des outils

Le projet nous a demandé de nombreux choix dans les formats de données à utiliser, et parfois d'algorithmes déjà développés par la communauté open-source. La plupart de ces choix ont été heureux, et nous ont permis d'avancer assez vite dans notre travail, mais l'un d'eux s'est avéré mauvais. Lorsque nous avons voulu trouver un algorithme de lecture de fichiers SGF les transformant en arbre, nous avons comparé une grande partie des logiciels connus du monde de l'open source traitant du sujet, nous avions choisi de transformer un petit programme écrit en C par l'inventeur du format en objet C++. Comme le programme était de très bas niveau, assez volumineux, et qu'il utilisait des pointeurs vers des fonctions, nous avons eu besoin de faire de nombreuses adaptations qui nous ont pris plusieurs semaines avant d'arriver à un résultat fonctionnel. Quelques jours plus tard, nous avons voulu instancier plusieurs objets de la classe créée, et nous avons à nouveau rencontré des problèmes. Après quelques jours à tenter de les corriger, nous avons abandonné ce programme, et nous avons récupéré un morceau du code de Mugo, écrit en C++ avec QT4. Malgré le fait que ce programme soit quasiment inconnu,

ce choix s'est avéré bien meilleur que le premier, et après le travail d'adaptation, ne nous a causé aucun problème.

9.1.4 Une mauvaise coordination de groupe

Au milieu du projet, nous avons perdu une membre du groupe. Cela nous a affecté dans notre fonctionnement, mais aussi dans la taille des objectifs visés. Par la suite, l'agrandissement du groupe par l'arrivée des membres liés au projet par d'autre projets nous a fortement remotivés, et nous avons pu repartir sur des objectifs différents, mais clairement non-amoindris.

9.2 Ce que nous a apporté le travail sur ce projet.

9.2.1 Le travail de groupe

Se retrouver face à un projet de cet ampleur nous a poussé à changer notre façon de travailler. Nous avons tout de suite commencé à utiliser un serveur SVN et un wiki. Malgré cela, il nous a assez vite fallu en plus discuter régulièrement entre membres du groupe, pour comprendre ce que chacun attendait des autres, et faire en sorte que l'intégration du travail personnel à celui du groupe de passe le mieux possible. Au final, nous avons réussi à répartir le travail entre les sous-groupes, qui ont évolué au fur et à mesure du projet, et à assembler parfaitement les différents rouages du programme.

9.2.2 Intelligence artificielle

Ce projet, fortement orienté Intelligence Artificielle nous a donné notre premier contact technique avec la matière. Nous avons pu implémenter divers types d'algorithmes dits d'Intelligence Artificielle. Nous avons compris les difficultés que l'on pouvait rencontrer face à ces algorithmes, au niveau implémentation, optimisation, et prévision des résultats. Notre programme arrive finalement à utiliser plusieurs facettes de l'Intelligence Artificielle de manière coordonnée selon le problème.

9.2.3 Attaquer un problème déjà exploré

Cela fait longtemps que le problème de la création d'un programme capable de rivaliser avec les humains au jeu de Go alimente les espoirs et les recherches en Intelligence Artificielle. La recherche est partie de loin, mais a assez vite progressé au fur et à mesure des années. Nous avons donc dû réfléchir à ce qui avait déjà été fait dont nous allions pouvoir nous servir et ce que nous pourrions apporter dans le domaine. Nous nous sommes rendus compte que bien que le sujet ait été exploré de plein de manières différentes, il manquait beaucoup de choses dans le monde de l'open-source au niveau des intelligences artificielles. Les algorithmes qui étaient utilisés étaient vieux et obsolètes, et la façon dont ils avaient été programmés indiquait qu'ils seraient difficilement évolutifs. Nous avons réussi à faire un programme open-source facilement évolutif, en utilisant des techniques de programmation modernes. Le projet nous a montré comment nous pouvions nous inséres dans un domaine complexe.

9.2.4 Travailler sur un domaine de recherche actif

Les algorithmes autour du Go donnent lieu à de la recherche active. Tout au long de notre TER, nous nous sommes penchés sur ce qui se faisait sur le sujet, et sur les différentes façons dont le problème avait été abordé au cours de l'histoire de l'Intelligence Artificielle, et ce qui se faisait maintenant. Nous avons donc grandement fait évoluer nos connaissances théoriques sur l'Intelligence Artificielle actuelle et ses applications dans un domaine concret.

9.3 L'apport de notre programme au monde des intelligences artificielles.

Nous avons implémenté au cours de notre projet plusieurs fonctions dites d'Intelligence Artificielle. Elles ont été faites pour être les plus modulables et évolutives possibles.

9.3.1 Dans le jeu de Go

Notre programme d'Intelligence Artificielle est une première dans le monde des programmes jouant au Go open-source. En effet, il est écrit dans un langage objet (contrairement à GNUGO par exemple, programme open-source joueur de Go le plus connu), et implémente des algorithmes modernes comme UCT (algorithme jamais implémenté dans une intelligence artificielle de go auparavant). De plus, le système comprend un outil d'ingénierie cognitive, composé d'un lecteur de dictionnaires et d'un système expert, tous deux très simple à utiliser, même pour un non-informaticien, ce qui permet à tout joueur de Go d'améliorer facilement le projet de manière conséquente. Même si notre programme est pour l'instant clairement un niveau en-dessous des programmes les plus connus, nous avons donc beaucoup d'espoir pour le futur de notre programme comme concurrent sérieux aux programmes Open-Source qui nous ont précédés.

9.3.2 Dans les autres domaines

Outre le jeu de Go, sur lequel ils ont été testés, chacun des outils que nous avons développés, et dans certains cas, notre implémentation de leur interactivité, pourra être adapté à un autre jeu, voir à un problème totalement différent. Tous les outils ont été faits pour être les plus polymorphes possible, et leur utilité assez généraliste en font des outils réutilisables dans de multiples domaines.

Tout le programme pourrait être réutilisé pour le Backgammon. Le format des dictionnaires lui est adapté (le format SGF permet aussi d'enregistrer entre autre les parties de Backgammon), le système expert pourrait aussi directement être réutilisé, comme UCT/RAVE. La plus grande tâche d'adaptation serait d'écrire la classe tablier, qui remplacerait la classe goban.

Avec un peu plus de travail, l'architecture pourrait en partie s'adapter à Eternity. Le système expert pourrait permettre de rentrer des règles simples pour la gestion des bords ou des endroits vides plus ou moins entourés, et le dictionnaire des formes, avec quelques adaptations, permettrait de reconnaître certains types de situations, même si cela semble compromis, vu qu'il est difficile de construire un dictionnaire des bons coups pour Eternity. Toutefois, une couche d'apprentissage pourrait permettre de trouver les formes donnant les meilleurs résultats. Quand à notre implémentation

d'UCT/RAVE, contenant multi-threading et cloud Computing, elle est directement indiquée pour ce genre de travail, et peut rester guidée par les deux outils précédents.

Chapitre 10

Documentation

Sommaire

"Le go reflète l'âme."
Shan Sa

10.1 Bibliographie

10.1.1 sur le jeu de Go

Livres

- Fan Hui, L'âme du Go : Livre sur les formes dans le jeu de Go.
- Lee Chang-ho : Korean Style of Badduk : livre sur le Fuseki mini-Chinois.

Sites internet

- Sensei's Library : Site généraliste sur le jeu de Go : http ://senseis.xmp.net/
- Kogo Joseki Dictionary : Dictionnaire de josekis : http ://waterfire.us/joseki.htm
- Site internet de Denis Feldman : Site très complet sur les règles et l'histoire du Go : http ://denisfeldmann.fr/

10.1.2 Travaux sur les intelligences artificielles de Go

- `Travaux de Olivier Teytaud : http://www.lri.fr/~teytaud/`
- `Thèse de Sylvain Gelly : http://www.lri.fr/~gelly/`
- `Thèse de Erik van der Werf : http://erikvanderwerf.tengen.nl/`
- `Travaux de Tristan Cazenave : http://www.lamsade.dauphine.fr/~cazenave/`
- `Travaux de Olivier Teytaud : http://www.lri.fr/~teytaud/`
- `Articles de Markus Enzenberger :`
  `http://webdocs.cs.ualberta.ca/~emarkus/neurogo/`

10.2 Lexique

10.2.1 Vocabulaire sur le jeu de Go

Vocabulaire de "phases" dans une partie :

Fuseki : début de partie, ou ouverture
Chuban : milieu de partie
Yose : fin de partie
Joseki(séquence théorique) : séquence de début classique dans un coin ou sur un bord, amenant à une bonne situation pour les deux joueurs

Vocabulaire de base

Goban : Plateau de jeu.
Intersection : Le Go se joue sur les intersections entre les lignes du goban.
Pierre : Nom donné aux "pions" que posent les joueurs, chacun à leur tour sur le goban.
Groupe : Un groupe est un regroupement de pierres directement reliées entre elles. Un groupe est dit "mort" lorsqu'il n'a plus aucune chance de faire deux yeux. Il n'est pas enlevé du plateau tant qu'il lui reste des libertés, mais sera considéré comme mort à la fin de la partie. Un groupe qui a deux yeux est dit "vivant".
Oeil : Un oeil est une intersection entourée par quatre pierres appartenant à un même groupe.
Atari : une pierre ou un groupe de pierres est en Atari lorsqu'il n'a plus qu'une liberté, et est donc en position d'être capturé au coup suivant.
Liberté : Une liberté d'un groupe est une intersection libre à côté de ce groupe. Un groupe qui n'a plus aucune liberté est mort et retiré du plateau.

Tsume-go : étude de vie et de mort des groupes

Te-nuki : coup ignorant la dernière menace de l'adversaire pour jouer ailleurs

Sente : un coup " Sente " permet de prendre ou de conserver l'initiative

Gote : un coup " Gote " fait perdre l'initiative

Shisho : Un shisho ("Chemin à une liberté") est une situation ou pendant toute une séquence de coups, un groupe va tenter de s'enfuir, mais ne pourra jamais avoir plus de deux libertés. Un shisho peut finir par une connexion avec une pierre du groupe qui tente de se sauver, ou par la mort du groupe.

Ko : Un ko ("Eternité") est un point du goban où une pierre est en prise, mais où sa prise créerait une répétition de position, ce qui pourrait amener une partie sans fin, la position pouvant revenir éternellement. La pierre ne peut donc être prise immédiatement.

Menace de Ko : Coup senté sur le plateau, pour tenter de forçer l'adversaire à répondre, pour pouvoir ensuite reprendre une pierre de Ko, ou récpuérer des points si l'adversaire n'a pas répondu.

Influence : Les pierres qui font de l'influence sont les pierres donnant de la force à un joueur sur une grande zone du goban. Les zones d'influence sont plus vastes mais moins solides que les zones de territoires. Les zones d'influence peuvent servir à être transformées en territoire ou à pousser l'adversaire à les attaquer pour lancer des combats favorables.

Territoire : Des pierres qui font du territoire sont des pierres qui défendent une partie du goban de manière solide de façon à garantir des points en fin de partie. Ces pierres sont en générales plus proches les unes des autres, et plus proches du bord que les pierres faisant de l'influence.

Tengen : Intersection centrale du goban.

Vocabulaire de base pour les coins

Hoshi (étoile) : point 4-4 dans un coin

San-san : point 3-3 dans un coin

Ko-moku : point 3-4 dans un coin

10.2.2 vocabulaire informatique

Termes informatiques généraux

Arbre : Un arbre est un graphe non-orienté connexe et sans cycle.

Graphe : un graphe est un ensemble de points, dont certaines paires sont reliées par des liens. les points sont appelés des noeuds et les liens des arrêtes.

Cloud computing : Partage des ressources informatiques entre plusieurs ordinateurs. Voir chapitre 6-3-4.

Thread : Un thread est un sous-processus ou processus léger, capable d'executer un certain nombre de calculs en parallèle d'une tâche principale. Sur les machines multi-Coeurs, multi-threader un programme (le diviser en plusieurs threads) peut améliorer ses performances. Voir chapitre 6-3-3

Mutex : Outil informatique permettant de proteger une ressource pour éviter qu'elle soit modifiée par plusieurs threads en même temps. Voir chapitre 6-3-3

Open-Source (logiciel) : Logiciel dont le code-source est en accès libre et est modifiable et redistribuable.

Acronymes

GTP Go Text Protocol : Protocole permettant à un programme joueur de Go de communiquer avec une interface. Voir Chapitre 3-3.

SGF Smart Game Format : Format permetant de stocker des parties de beaucoup de jeux, ainsi que leurs variations. Principalement utilisé pour le jeu de Go. Voir chapitre 5-2

SVN Subersion : Système de gestion des versions. Permet à plusieurs personnes travaillant sur un même projet de partager facilement du code et d'autres données, que le serveur fusionne.

UCT Upper Confidence bounds applied to Trees : Algorithme probabiliste basé sur Monte Carlo. Voir Chapitre 6.

Programmes

Many faces of Go : Une des plus anciennes intelligence artificielle de Go. Programme commercial.

Mugo : Editeur de fichiers SGF pouvant servir d'interface graphique grace au protocole GTP. Programme Open-Source.

Mogo : Intelligence artificielle de Go française, créée à l'institut universitaire Paris-Sud 11, basée sur UCT/Rave.

Zen : Intelligence artificielle de Go récente. Programme commercial.

Gnugo : Une des premières intelligences artificelles de Go, créée par l'équipe de Gnu.

10.3 Annexes

10.3.1 Diagrammes

Diagramme de classe Complet

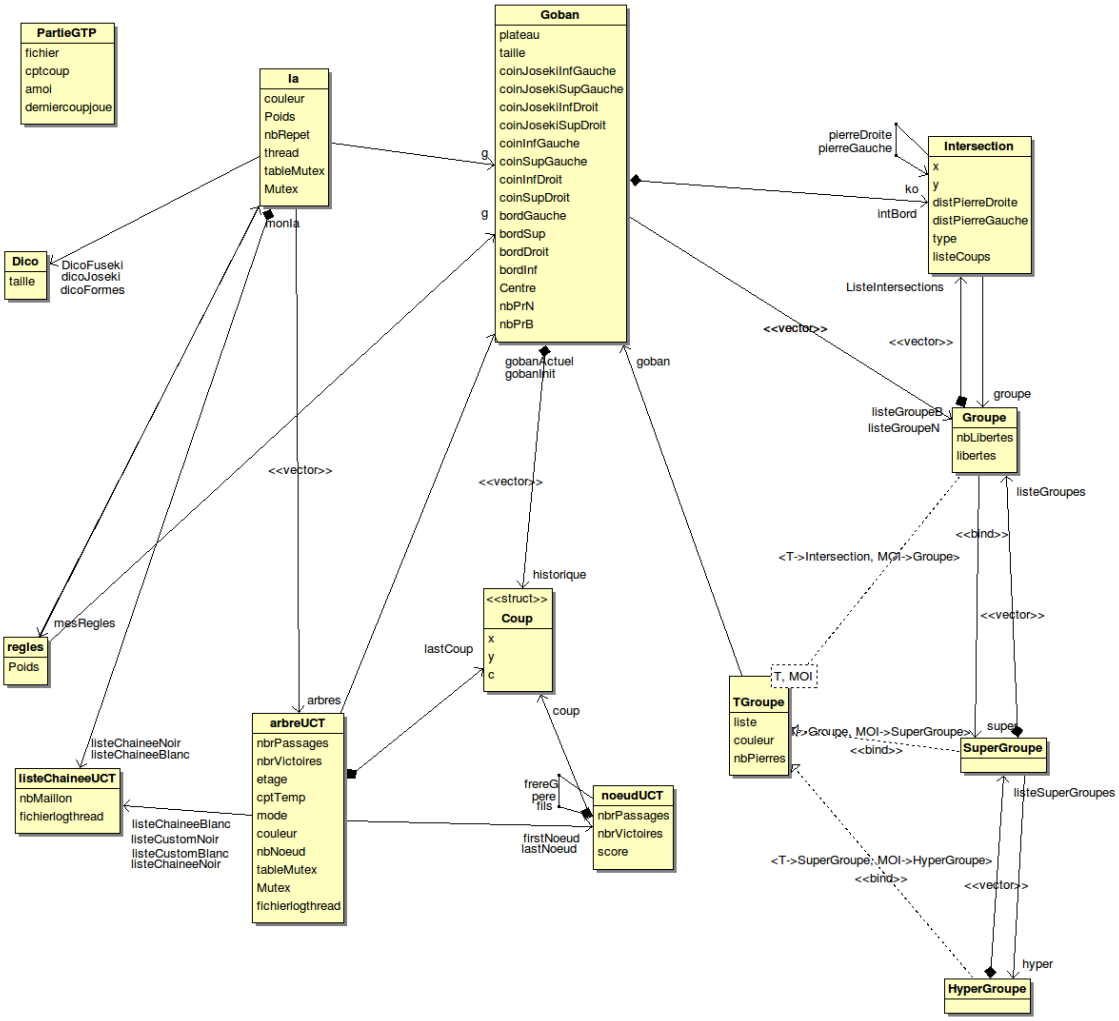


Diagramme de classe des fondations du programme

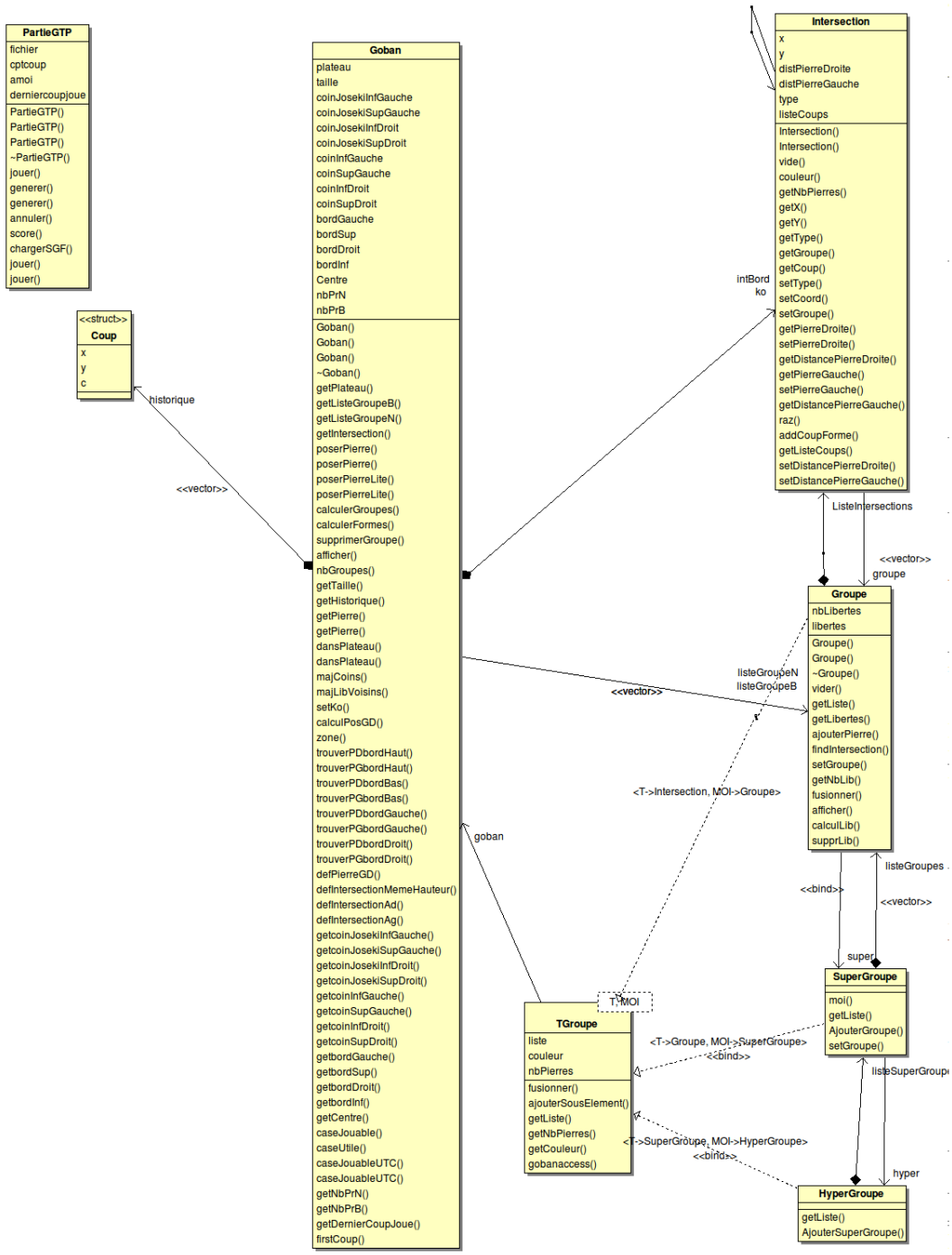
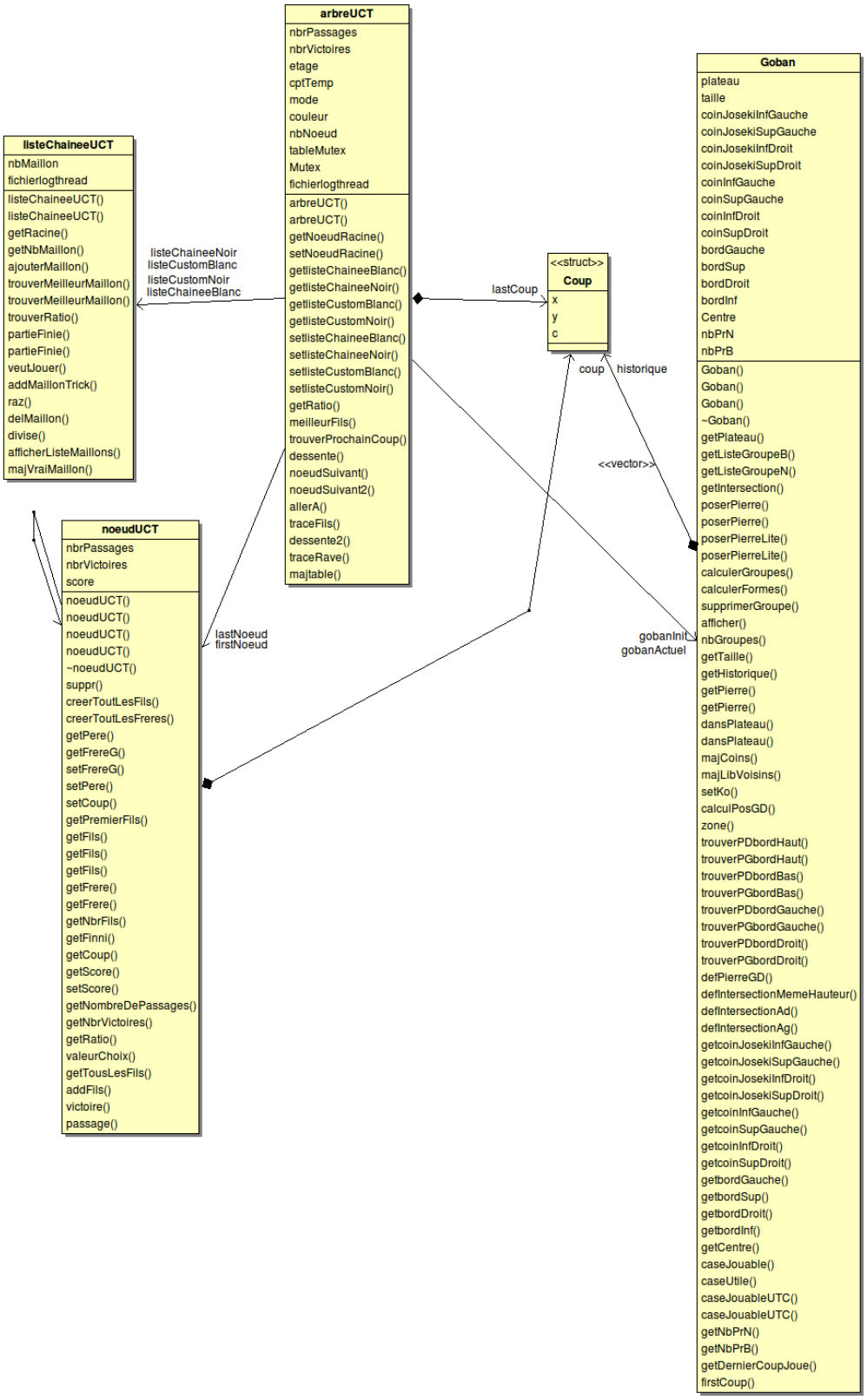


Diagramme de classe d'UCT



10.3.2 Prédicats et fonctions implémentés pour le système de règles

Prédicats

- Coin(Coins) : réinitialise une liste de coins
- Bord(Bords) : réinitialise une liste de bords
- groupe(Groupes) : réinitialise une liste avec tous les groupes
- ligne(lignes) : réinitialise une liste des lignes
- intersection(Intersections) : réinitialise une liste d'intersections
- BordLePlusVide(Bords) : retourne dans Bords le ou les bords où il y a le plus grand espace entre deux pierres
- CoinVide(Coins) : Retourne dans Coins tous les coins vides
- appartientabord(Coins, Bords) : Retourne dans Coins tous les coins qui appartiennent aux bords donnés.
- joseki(Coins) : Retourne dans Coins tous les coins dans lesquels le dictionnaire des josekis a trouvé quelque chose
- BordVide(Bords) : Retourne dans Bords tous les bords vides
- GroupeN (Groupes) : initialise la liste avec les groupes noirs si elle est vide, retire les groupes blancs sinon
- GroupeB (Groupes) : initialise la liste avec les groupes blancs si elle est vide, retire les groupes noirs sinon
- lignes(lignes, numLigne) : supprime de lignes toutes les lignes différentes de numLigne, qui est un entier, 0 désignant le bord extrême, 1 la première ligne etc...
- libertes(Groupes, libertes) : supprime les groupes ayant au moins libertes (un entier) libertés de la liste, retourne faux si la liste finale est vide
- nbcoupsjoues(nb) : indique si au moins nb coups ont été joués
- shisho(Groupes) : retourne vrai si le shisho fonctionne
- Tengen(Intersections) : supprime les intersections ne correspondant pas au tengen, retourne faux si le tengen n'est pas dans la liste
- videAutour(Intersections, nb) : supprime les intersections ayant des intersections non-vides aux nb intersections alentours.
- nbPierres(Groupe, nb) : supprime les groupes n'étant pas composés d'au moins nb pierres.

Conclusions

- J_intersection(Intersections) : Augmente la valeur des intersections données
- J_joseki(Coins) : Augmente la valeur des coups joseki dans les coins donnés
- J_liberte(Groupe) : Augmente la valeur des coups ajoutant des libertés au groupe
- J_coin(Coins) : Augmente la valeur des coups dans les Coins donnés
- J_Bord(Bords) : Augmente la valeur des coups dans les Bords donnés
- J_Ligne(Lignes) : Augmente la valeur des intersections situées sur les lignes indiquées (les lignes correspondent à des carrés : ex : jouer en ligne 0 augmetera la valeur des intersections situées sur un extrême bord)
- J_PlusGrandEspace(Bords) : Augmente sur les bords le poids des cases qui ont le plus de cases libres autour d'elles.

10.3.3 Un exemple de fichier de règles pour le 9x9

```
groupe(x) ^ GroupeN(x) ^ libertes(x,4) ->jouer(J_liberte(x),2)
groupe(x) ^ GroupeN(x) ^ libertes(x,3) ->jouer(J_liberte(x),5)
groupe(x) ^ GroupeN(x) ^ libertes(x,4) ^ testerFait(modeDefensif())
-> ajouterFait(modeCombat())
groupe(x) ^ GroupeN(x) ^ libertes(x,3) ->ajouterFait(modeCombat())
groupe(x) ^ GroupeN(x) ^ libertes(x,2) ^ non(shisho(x)) ->jouer(J_liberte(x),10)
groupe(x) ^ GroupeN(x) ^ libertes(x,2) ^ shisho(x) ->jouer(J_liberte(x),-50)
groupe(x) ^ GroupeB(x) ^ libertes(x,4) ->jouer(J_liberte(x),4)
groupe(x) ^ GroupeB(x) ^ libertes(x,3) ->jouer(J_liberte(x),12)
groupe(x) ^ GroupeB(x) ^ libertes(x,2) ->jouer(J_liberte(x),18)
groupe(x) ^ GroupeB(x) ^ libertes(x,2) ^ nbPierres(x,2) ->jouer(J_liberte(x),35)
groupe(x) ^ GroupeB(x) ^ libertes(x,2) ^ nbPierres(x,3) ->jouer(J_liberte(x),45)
groupe(x) ^ GroupeB(x) ^ libertes(x,4) ^ testerFait(modeOffensif())
-> ajouterFait(modeCombat())
groupe(x) ^ GroupeB(x) ^ libertes(x,3) ->ajouterFait(modeCombat())

testerFait(modeCombat()) ->retirerFait(modeCalme())

GroupeB(x) ^ shisho(x) -> jouer(J_shisho(x),50)

testerFait(modeCalme()) ^ intersection(x) ^ videAutour(x,1)
 ->jouer(J_intersection(x),1)
testerFait(modeCalme()) ^ intersection(x) ^ videAutour(x,2)
 ->jouer(J_intersection(x),2)
testerFait(modeCalme()) ^ intersection(x) ^ videAutour(x,3)
 ->jouer(J_intersection(x),5)

nbcoupsjoues(8) -> ajouterFait(Chuban())
nbcoupsjoues(20) -> ajouterFait(Yose())

testerFait(Chuban())  ->retirerFait(PreFuseki())
testerFait(Chuban())  ->retirerFait(Fuseki())
testerFait(Yose()) ->retirerFait(Chuban())

ligne(x) ^ lignes(x,0) ^ non(testerFait(Yose())) -> jouer(J_Ligne(x),-40)
ligne(x) ^ lignes(x,1) ^ non(testerFait(Yose())) ^ non(testerFait(Chuban()))
 -> jouer(J_Ligne(x),-10)
```