Extraction bayésienne et intégration de patterns représentés suivant les K plus proches voisins pour le go 19x19

Bruno Bouzy*, Guillaume Chaslot*

*Université Paris 5, C.R.I.P.5
45, rue des Saints-Pères 75270 Paris Cedex 06 France
bouzy@math-info.univ-paris5.fr,
http ://www.math-info.univ-paris5.fr/∼bouzy
**Ecole Centrale de Lille
Cité Scientifique - BP 48, 59651 Villeneuve d'Ascq Cedex
chaslot.guillaume@ec-lille.fr

Résumé. Cet article décrit la génération automatique et l'utilisation d'une base de patterns pour le go 19x19. La représentation utilisée est celle des K plus proches voisins. Les patterns sont engendrés en parcourant des parties de professionnels. Les probabilités d'appariement et de jeu des patterns sont également estimées à ce moment là. La base créée est intégrée dans un programme existant, Indigo. Soit elle est utilisée comme un livre d'ouvertures en début de partie, soit comme une extension des bases pré-existantes du générateur de coups du programme. En terme de niveau de jeu, le gain résultant est estimé à 15 points en moyenne.

1 Introduction

Le facteur de branchement et la longueur d'une partie interdisant la recherche arborescente globale au go et l'évaluation de positions non terminales étant difficile [14], la programmation du jeu de go est une tâche difficile pour l'informatique [15, 13]. Cependant, la programmation du go est un terrain d'expériences approprié pour l'IA [8]. INDIGO [7], programme de go développé dans l'esprit de valider des méthodes d'IA, est composé d'un module Monte Carlo (MC) et d'un module basé sur des connaissances. Le module MC a été décrit récemment [9, 4], et le module basé sur les connaissances a été décrit dans des travaux antérieurs à 2003 [8, 5, 6]. La figure 1 donne un aperçu du processus de choix du coup à jouer dans INDIGO. Le module basé sur les connaissances fournit *ns* coups au module MC qui, en vue de sélectionner le meilleur coup, joue un grand nombre de parties aléatoires jusqu'au bout et commençant par l'un de ces coups et calcule des moyennes. Le module basé sur les connaissances est donc un pré-processeur du module MC.

L'amélioration du module basé sur les connaissances est l'objet de cet article. Ce module comprend plusieurs bases de "patterns" construits à la main. Les bases de connaissances construites à la main ont plusieurs désavantages : elles contiennent des erreurs, elles ont des lacunes et elles ne peuvent pas être mises à jour facilement. Par ailleurs, les multiples bases de connaissances dans INDIGO ne partagent pas le même format : la première (FORME_M) contient des caractéristiques dépendantes du domaine

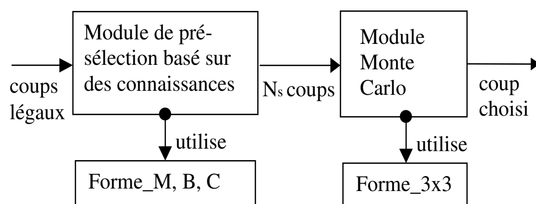

FIG. 1 – L'architecture de INDIGO comprend un module de pré-sélection et un module Monte Carlo et des bases hétérogènes de patterns, FORME_B, FORME_C, FORME_M et FORME_3X3.

utilisées par la fonction d'évaluation conceptuelle, la seconde (FORME_3X3) comprend des formes 3x3 optimisées pour faire des simulations rapides, et la dernière (FORME_B et FORME_C) est dédiée au début de partie et contient des patterns larges. La figure 1 montre comment sont utilisées les différentes bases de patterns dans INDIGO. Etant donné le succès de l'approche Monte Carlo dans INDIGO, nous avons voulu à nouveau utiliser des statistiques dans le module basé sur les connaissances. C'était donc le bon moment pour tester la création automatique d'une nouvelle base de patterns et d'observer ses effets positifs dans l'architecture de INDIGO. La création automatique de patterns évite les erreurs et les manques dans la base. La création automatique est effectuée en parcourant des parties de professionnels pour créer des patterns et estimer leur probabilité de "match" et leur probabilité d'être joué sachant qu'ils ont "matchés". En d'autres termes, l'approche adoptée est une approche bayésienne. Pour éviter une limitation liée à la taille des patterns, particulièrement au début de la partie, nous avons choisi la représentation des K plus proches voisins dans laquelle les faits saillants sont les intersections occupées et les bords. Pour cette raison, la base de patterns construites s'appelle FORME_K.

Le but de cet article est la description de l'extraction bayésienne de patterns représentés suivant les K plus proches voisins à partir de parties de joueurs professionnels et l'utilisation de ces patterns dans le programme INDIGO jouant des parties de go. Le but de l'article n'est pas de décrire le programme INDIGO dans on ensemble, ni son module Monte Carlo (déjà décrit dans [9, 4]). La partie 2 rappelle les travaux antérieurs liés à ce travail. La partie 3 définit la représentation basée sur les K plus proches voisins utilisée dans ce travail. Puis, la partie 4 décrit la création de patterns et leurs propriétés probabilistes. La partie 5 présente les expériences effectuées en vue d'intégrer ce travail dans INDIGO, et elle évalue les progrès réalisés. Avant la conclusion, les perspectives sont mises en lumière par la partie 6.

2 Travaux existants

Malgré son importance dans les programmes de go, la littérature sur l'acquisition de patterns et la génération locale de coups n'est pas très développée. [2] de Mark Boon

fut le premier article décrivant un algorithme d'appariement au go en détails : celui de Goliath, meilleur programme en 1990. Mais cet article ne décrivait pas l'acquisition des patterns. Plus récemment, Erik van der Werf travaille suivant une approche basée sur un réseau de neurones pour engendrer des coups locaux [17], prédire la vie et la mort des groupes (concepts importants au jeu de go) [19], ou affecter un score aux positions terminales [18]. Comme notre système parcourt des parties de professionnels pour engendrer des patterns conseillant des coups locaux, on peut dire que notre travail est dans la lignée de celui de Erik van der Werf. Cependant, il est moins sophistiqué car il utilise l'approche des K plus proches voisins au lieu d'un réseau de neurones. De plus, il ne prédit pas la vie et la mort et ne donne pas de scores aux positions terminales. Tristan Cazenave a travaillé sur l'acquisition automatique de patterns tactiques pour les "yeux" et la connexion [10], incluant aussi des conditions externes [11]. Notre travail est similaire à celui de Cazenave par son approche automatique mais elle est assez différente par ailleurs : les patterns sont créés avec une approche bayésienne dans le but d'être utilisé par le niveau global du programme alors que les patterns de Cazenave sont créés dans un but particulier (capturer ou connecter des chaînes de pierres, etc.), et suivant une méthode d'apprentissage basée sur les explications. Enfin [3] décrit la génération de patterns 4x4 par analyse rétrograde. Bien que concernant la génération automatique, ce travail était appliqué à des patterns limités en taille et il utilisait l'analyse rétrograde au lieu de l'approche bayésienne.

3 La representation des K plus proches voisins

La représentation des K plus proches voisins est classique en reconnaissance des formes [1]. Cette partie définit la representation des K plus proches voisins utilisée dans ce travail.

3.1 Patterns K plus proches voisins

Le dessin ci-dessous montre un exemple de pattern K plus proches voisins.

```
 +@+
+++++
++*+O
+++++
 ++@
```

Un pattern conseille un coup *en son centre* indiqué par un '*'. '+' désigne une intersection vide. 'O' désigne une pierre blanche et '@' une pierre noire. '+' est un fait *non important* dans cette representation. Inversement, une pierre noire, une pierre blanche ou un bord ('—') sont des faits *importants*. Un pattern contient un nombre de faits importants appelé K. Dans l'exemple précédent, K=3.

Le centre du pattern étant donné, nous supposons que les intersections voisines sont ordonnées suivant une distance. Nous supposons aussi que cet ordre pré-défini évite les ex-aequo entre les intersections situées à distance égale du centre du pattern. Avec de

telles suppositions, l'algorithme d'appariement reste simple et peut être programmé pour être efficace.

L'avantage de cette représentation est l'absence de limitation de taille des patterns. Au go, beaucoup de coups sont joués en fonction du voisinage des pierres et du voisinage du bord. Pour simplifier le travail, nous avons contraint les patterns de conseiller un coup en son centre, et pas ailleurs.

La représentation des K plus proches voisins ne contient pas explicitement de symbole '#' désignant une intersection inconnue (donc potentiellement égale à '@', 'O', '+', '—'), approche habituelle dans les programmes de go [2]. Cependant, dans d'autres représentations gérant les '#', les patterns sont grossièrement centrés autour du coup conseillé, et les '#' sont souvent situés "loin" du centre du pattern, alors que les points importants sont situés près du centre. Donc la représentation des K plus proches voisins contient implicitement des '#'. Ne pas gérer ces points explicitement simplifie l'algorithme d'appariement. Parce que remplacer un pattern contenant un '#' par 4 patterns contenant une des 4 valeurs possibles ('@', 'O', '+', '—') est toujours possible, cette représentation ne perd pas de généralité pourvu que la mémoire disponible soit suffisante.

L'algorithme d'appariement doit gérer les symétries, rotations et inversions noir-blanc d'une façon ou d'une autre. Sur les 16 patterns équivalents à un pattern donné, une première approche consiste à ne stocker en mémoire que le pattern donné et l'autre approche consiste à stocker les 16 patterns. Dans une première version de notre travail, nous avons utilisé la seconde approche qui simplifie la programmation de l'algorithme d'appariement.

3.2 La création des patterns

Pour un ensemble donné de parties, la création des patterns est brutale. Elle correspond au pseudo-code suivant :

```
Forme_k::creerPatterns() {
  Pour k = 1 a Kmax
    Pour chaque partie
      Pour chaque coup i de la partie
        creerPattern(k, i);
}
```

Si le pattern n'existe pas encore, la fonction $creerPattern(k, i)$ crée le pattern centré en i avec k faits voisins importants suivant l'ordre pré-défini entre les intersections. Les patterns sont stockés dans un arbre dont les noeuds ont 4 fils : le noeud "si vide", le noeud "si noir", le noeud "si blanc" et le noeud "si bord". Grâce à un tel arbre, l'algorithme d'appariement est rapide, même avec un très grand nombre de patterns.

4 Génération bayésienne

Cette partie décrit l'aspect bayésien du travail effectué, classique dans les tâches de classification [1]. D'abord, nous définissons et nommons les probabilités intéressantes.

Puis, nous montrons comment notre générateur de patterns estime ces probabilités. Enfin, nous présentons la façon d'éliminer les "mauvais" patterns.

4.1 Définitions

P désigne une probablilité. i désigne soit une intersection soit un coup joué sur cette intersection. p désigne un pattern. $P(p)$ est la probabilité que le pattern p s'apparie sur une intersection arbitraire sur l'ensemble des parties de joueurs professionnels. $P(i)$ est la probabilité que le coup soit joué sur i. $P(i, p)$ est la probabilité que le coup soit joué sur i et que le pattern p s'apparie sur i. $P(i|p)$ est la probabilité que le coup soit joué sur i sachant que le pattern p s'apparie sur i. Finalement, $P(p|i)$ est la probabilité que le pattern p s'apparie sur i sachant que le coup est joué sur i. $P(i)$ et $P(p)$ sont des probabilités à priori. $P(i|p)$ et $P(p|i)$ sont des probabilités à postériori.

Pendant la phase de jeu, l'idée de la méthode est d'effectuer l'appariement sur toutes les intersections i du damier, et d'utiliser $P(i|p)$ comme une estimation de l'urgence à jouer un coup sur i. Pendant la phase d'apprentissage, l'approche est basée sur la fréquence des événements, la probabilité qu'un événement arrive est approximée par le nombre d'occurences de l'événement divisé par le nombre de tests effectués. Dans la suite, nous dirons qu'un pattern est *fréquent* si $P(p)$ est élevée, *bon* si $P(i|p)$ est élevée, et *utile* si $P(p|i)$ est élevée. Donc, nous avons défini une classe FORME_K dont les propriétés bayésiennes sont spécifiées ci-dessous en C++. Le terme "static" est un mot-clé C++ désignant une propriété de classe.

```
class Forme_k {
  static int n_test;
  static int n_joue;
  int n_match;                 // p.n_match
  int n_joue_sachant_match;    // p.n_joue
  static float p_joue;         // P(i)
  float p_match;               // P(p)
  float p_joue_sachant_match;  // P(i|p)
  float p_match_sachant_joue;  // P(p|i)
  ...
};
```

Les formules pour calculer les probabilités sont les suivantes :

```
  P(i)   = n_joue/n_test;
  P(p)   = p.n_match/n_test;
  P(i|p) = p.n_joue/p.n_match;
  P(p|i) = p.n_joue/n_joue;
```

Précisons que nous n'avons pas besoin d'utiliser la formule de Bayes mais seulement des probabilités à postériori. Cependant, avec nos définitions, la formule de Bayes est valide :

$(p.n_match/n_test).(p.n_joue/p.n_match) = (n_joue/n_test).(p.n_joue/n_joue)$

4.2 Estimation des probabilités

Pour un ensemble donné de parties et pour un ensemble donné de patterns, le processus bayésien correspond au pseudo-code suivant :

```
Forme_k::calculerProbabilites() {
  n_joue = n_test = 0;
  Pour chaque pattern p,
    p.n_match = p.n_joue = 0;
  Pour chaque partie {
    Pour chaque coup de la partie {
      n_joue++;
      Pour chaque intersection i du damier,
        test(i);
    }
  }
  Pour chaque pattern p,
    p.p_joue = p.n_joue/p.n_match;
}

Forme_k::test(i) {
  n_test++;
  patternMatching();
  pour chaque pattern p matchant sur i {
    p.n_match++;
    si le coup est joue sur i alors
      p.n_joue++;
  }
}
```

Un test sur une intersection i sur une position donnée d'une partie donnée répond aux deux questions : le coup a-t-il été joué sur i, et quels patterns se sont appariés sur i ? Sur des damiers 19x19, 200 à 300 tests sont effectués par position et une partie contient grossièrement 200 coups, donc 50,000 tests sont effectués par partie. Avec les 2,000 parties de professionnels que nous avons aujourd'hui, cela fait environ 100,000,000 tests.

4.3 Eliminer les mauvais patterns

L'idée est d'éliminer les patterns qui ne sont pas bons ($P(i|p)$ faible) ou bien les patterns dont la probabilité est estimée avec un niveau de confiance trop faible. Premièrement, parce que les patterns avec une probabilité faible sont moins intéressants le processus de génération ne garde que les patterns tels que $P(i|p) > 0.01$. Deuxièmement, le niveau de confiance de P (étant $P(i|p)$) est calculé en phase d'apprentissage. Les bases des statistiques [12] donnent $\sigma = \sqrt{P(1-P)}$. Pour la plupart des patterns on a $P << 1$, donc $\sigma = \sqrt{P}$. La quantité adéquate pour estimer le niveau de confiance est $s(i|p) = \sigma/\sqrt{p.n_match} = \sqrt{p.n_joue}/p.n_match$. Par ailleurs, on peut éliminer

les patterns tels que $P(i|p) < threshold \times s(i|p)$. Cependant, en pratique,nous avons décidé d'appliquer cette règle seulement lorsque notre ensemble de parties sera plus grand. Avec ce choix, notre système a extrait des bases paramétrées par K le nombre maximal de voisins pertinents. La table 1 donne le nombre de patterns engendrés pour certaines valeurs de K.

K	6	9	15
patterns	8,000	27,000	85,000

TAB. 1 – Nombre de patterns engendrés pour $K = 6, 9, 15$.

5 Expériences

Il y a deux façons d'utiliser FORME_K dans INDIGO : jeu par coeur comme un dictionnaire d'ouvertures sans vérification Monte Carlo (partie 5.1), et utilisation dans le pré-processeur pour vérification Monte Carlo (partie 5.2). Pour chaque façon, nous avons fait des expériences pour évaluer l'effet de FORME_K. Une expérience consiste en une série de 100 parties entre le programme à évaluer, KATIA, et le programme de référence, INDIGO2004 qui a participé aux olympiades d'ordinateurs 2004. Chaque programme joue 50 parties avec Noir. Le résultat d'une expérience est un ensemble de scores présentés dans une table supposant que KATIA est le joueur max. Une résultat positif dans une case de la table indique donc un succès. Etant donné que l'écart-type des scores de parties jouées sur damier 19x19 par nos programmes est d'environ 75 points, 100 parties permettent d'abaisser σ à 7.5 points (seulement) et d'avoir un intervalle de confiance à 95% avec un rayon égal à 2σ, soit 15 points. Nous avons utilisé des ordinateurs à 2.4 GHz. INDIGO et KATIA utilisaient donc tous les deux FORME_B, FORME_C, FORME_M et FORME_3X3. Et évidemment, KATIA utilisait FORME_K.

5.1 Utiliser Forme_K comme un livre d'ouvertures

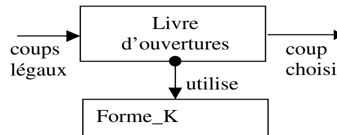

FIG. 2 – Au début de la partie, KATIA se réduit à un livre d'ouvertures utilisant la base FORME_K.

Pour observer rapidement l'effet de l'utilisation de FORME_K, il est décidé que, lors des *debut* premiers coups de la partie, KATIA joue directement le meilleur coup conseillé par FORME_K. Ainsi, au début de la partie, KATIA utilise FORME_K comme un livre

d'ouvertures sans vérification MC ultérieure (cf. figure 2). Le meilleur coup conseillé par FORME_K est le coup conseillé par le pattern s'appariant avec la position et ayant la probabilité $P(i|p)$ maximale. Pour que deux exécutions différentes de KATIA donnent des débuts de parties différents sans perte de niveau de jeu, une légère randomisation a été introduite dans le livre d'ouvertures. La figure 3 donne les 40 premiers coups d'un début de partie jouée par KATIA contre elle-même de cette manière. Les forts joueurs de go reconnaissent que ce début de partie est excellent. Ce qui reflète la force de l'approche bayésienne associée à une représentation basée sur les K plus proches voisins. C'est le point fort de cet article.

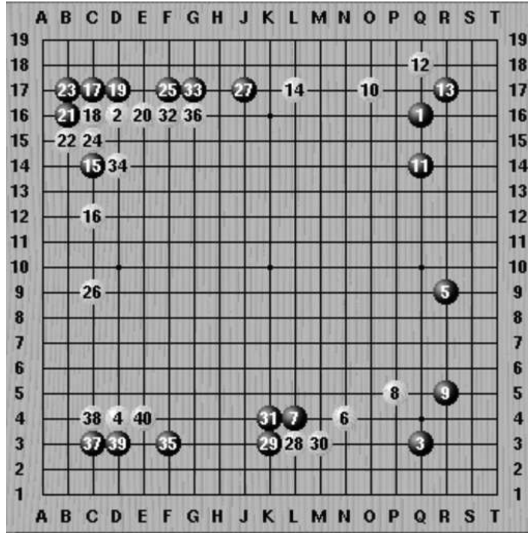

FIG. 3 – Les 40 premiers coups d'une partie jouée par KATIA contre elle-même

Cette évaluation qualitative étant faite, il est important d'évaluer FORME_K quantitativement en termes de scores de fin de parties. La table 2 montre les résultats entre KATIA(K, DEBUT) et INDIGO. Pendant les *debut* premiers coups, le coup joué par KATIA est celui conseillé par FORME_K. Après les *debut* premiers coups de la partie, KATIA utilise le même processus de choix que INDIGO (cf. figure 1).

Comme on s'y attendait, le résultat augmente avec K, mais jusqu'à $K = 15$ les résultats sont négatifs. La valeur convenable de *debut* est à déterminer. $begin <= 40$ donne de bons réusltats. En revanche, pour $begin >= 50$, le niveau de jeu diminue lorsque *begin* augmente. Cela s'explique par l'absence d'utilisation des concepts importants du domaine dans la représentation utilisée. Parce que KATIA(BEGIN=40) joue instantanément pendant les 40 premiers coups, elle économise environ 30% du temps de réflexion sur une partie complète. Donc, arrivé à e stade, l'intégration montre déjà un effet positif en termes de niveau de jeu et de temps de réflexion.

	6	9	15
10	-1	+1	+1
20	-6	+2	+5
30	-7	-2	+2
40	-29	-4	+4
50	-45	-26	-11

TAB. 2 – Résultat moyen of KATIA(K, DEBUT) contre INDIGO pour k = 6, 9, 15 et debut = 10, 20, 30, 40, 50.

5.2 Intégrer Forme_K dans le pré-processeur MC

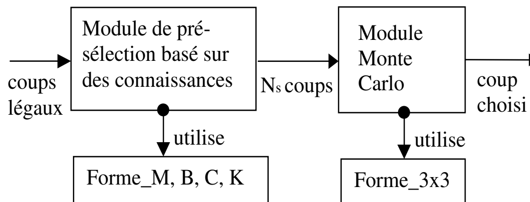

FIG. 4 – Après le début de partie, KATIA utilise une architecture identique à celle de INDIGO, avec la base FORME_K en supplément.

Le but de cette partie est de valider l'intégration de FORME_K dans le pré-processeur basé sur des connaissances (cf figure 4). Il faut bien noter que FORME_K est utilisé par le pré-processeur MC mais pas par le module MC lui-même. Le module MC n'utilise que la base FORME_3X3. Nous appelons KATIA(NK) la version de KATIA qui sélectionne nk coups avec FORME_K et $ns - nk$ coups avec le pré-processeur existant. En 2004, INDIGO et KATIA utilisent $ns = 7$. Parce que FORME_K ne contient pas de représentation élaborée relative aux concepts importants du jeu de go, nous avons fait varier nk entre 0 et 4 pour garder au moins 3 coups engendrés par des connaissances incluant ces concepts importants. La table 3 montre ces résultats.

	0	10	20	30	40	50
0		+1.0	+5.4	+0.6	+3.5	-11.1
1	-1.3	+4.9	+1.8	+3.6	+2.1	-2.9
2	+9.6	+15.8	+10.0	+6.1	+5.8	-13.0
3	+3.9	+8.6	-0.7	-1.5	-6.7	-20.1
4	+5.1	-2.5	+6.7	-4.2	+1.0	-16.9

TAB. 3 – Résultat moyen de KATIA(DEBUT, NK) contre INDIGO pour debut = 0, 10, 20, 30, 40, 50 et pour nk = 0, 1, 2, 3, 4.

Plusieurs de ces résultats sont positifs. KATIA(DEBUT=10,NK=2) est 15 points meilleure que INDIGO en moyenne. Il est intéressant de commenter la ligne de résultats correspondant à KATIA(NK=2). Premièrement, KATIA(DEBUT=0,NK=2) montre le résultat de l'intégration de FORME_K avec vérification MC sans livre d'ouvertures. Il faut noter l'amélioration de 10 points causée par l'insertion de 2 coups FORME_K au sein des 7 coups sélectionnés. Ce fait traduit le manque connu de patterns dans les bases construites manuellement et la présence de patterns importants dans la base construite automatiquement. Deuxièmement, le résultat de KATIA(DEBUT=10,NK=2) est aussi surprenant. Il montre que les 5 premiers coups joués en réflexe donne une amélioration de 5 points en moyenne. Troisièmement, le résultat de KATIA(DEBUT=20,NK=2) traduit un bon compromis : le niveau de jeu obtenu et le même que celui pour $debut = 0$ et le temps de réflexion est économisé de 20% sur le temps total sur une partie. Finalement, KATIA(DEBUT=30 OU 40,NK=2) peuvent être considérés comme des compromis raisonnables entre le temps de réflexion économisé et le niveau de jeu. En revanche KATIA(DEBUT=50,NK=2) n'est pas raisonnable, la perte de niveau de jeu étant trop importante. Enfin, il est possible de commenter la table colonne par colonne. Les meilleurs résultats sont obtenus pour $nk = 2$. FORME_K n'incluant pas de concepts importants du jeu de go, il est normal que nk ne soit pas très élevé en regard de $ns - nk$. En résumé, en copiant la version appropriée de KATIA dans INDIGO, peut-être KATIA(DEBUT=20,NK=2), nous pouvons conclure que FORME_K peut être intégrée avec succès dans INDIGO, et nous attendons la prochaine compétition d'ordinateurs pour mesurer les éventuels progrès contre des programmes conçus différemment.

6 Perspectives

Nous avons prévu de ré-engendrer FORME_K avec un nombre de parties supérieur à 2,000. Par exemple, le CDROM GoGod contient 30,000 parties de professionnels et a la bonne taille pour évaluer le gain en niveau de jeu en fonction du nombre de parties utilisées pour la génération. Cela permettra d'affiner les estimations des probabilités et par conséquent d'affiner les urgences de coups en phase de jeu. Une amélioration devrait être observée. Prendre en compte les symétries, rotations et inversions noir-blanc est aussi un travail à faire pour mieux estimer les probabilités. Nous souhaitons également étendre la représentation pour que les coups puissent être conseillés non seulement au centre du pattern mais aussi sur des intersections voisines du centre.

A moyen terme, nous avons deux perspectives vraiment intéressantes. D'abord, intégrer FORME_K avec la fonction d'évaluation conceptuelle pour éventuellement remplacer FORME_M. Ensuite, intégrer un sous-ensemble de FORME_K dans le moteur de parties aléatoires pour éventuellement remplacer FORME_3X3 dans le module MC lui-même. La première intégration pose un problème de génie logiciel, FORME_M étant spécifiquement utilisée par la fonction d'évaluation conceptuelle de INDIGO. La seconde intégration pose un problème de performance d'abord. Les parties aléatoires devant être très rapides, l'appariement effectué à chaque coup d'une partie aléatoire doit être limité à un voisinage très restreint du coup précédent. Un problème d'ajustement automatique des urgences des patterns se pose ensuite. L'apprentissage par renforcement [16] est la solution envisagée.

7 Conclusion

Nous avons présenté une méthode pour extraire automatiquement des patterns avec des parties de professionnels. Cette méthode utilise des estimations de probabilités et ne présuppose pas de connaissances dépendantes du domaine. A ce titre, c'est une bonne continuation d'un programme de go basé sur Monte Carlo. La représentation utilisée est celle des K plus proches voisins. La génération bayésienne sur cette représentation a produit une base de patterns donnant des débuts de parties excellents. Ce travail démontre par l'expérience que cette représentation est très adaptée au go. C'est le point fort de cet article. Sa faiblesse réside dans l'absence prévue de compréhension de concepts importants du jeu de go tels que la vie et de la mort des groupes. Donc cette approche ne devait pas être utilisée telle quelle et devait être combinée avec des techniques existantes, ce qui a été fait.

Nous avons intégré la base de patterns dans le programme INDIGO. Le résultat est positif. Ajouter la base de patterns dans le pré-processeur du module MC permet à INDIGO d'augmenter son niveau de jeu de 15 points en moyenne sur des damiers 19x19, ce qui est significatif au go. Qui plus est, au début de la partie, la qualité des coups produits permet au programme de jeu de jouer ces coups en réflexe sans vérification MC. Par conséquent, 20% du temps de réflexion est économisé et laisse de la place pour d'autres améliorations.

Références

[1] C. Bishop. *Neural networks and pattern recognition.* Oxford University Press, 1995.

[2] M. Boon. A pattern matcher for Goliath. *Computer Go*, 13 :13–23, 1990.

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

[4] B. Bouzy. Associating knowledge and Monte Carlo approaches within a go program. In *7th Joint Conference on Information Sciences*, pages 505–508, Raleigh, 2003.

[5] B. Bouzy. Mathematical morphology applied to computer go. *International Journal of Pattern Recognition and Artificial Intelligence*, 17(2) :257–268, March 2003.

[6] B. Bouzy. The move decision process of Indigo. *International Computer Game Association Journal*, 26(1) :14–27, March 2003.

[7] B. Bouzy. Indigo home page. www.math-info.univ-paris5.fr/~bouzy/INDIGO.html, 2004.

[8] B. Bouzy and T. Cazenave. Computer go : an AI oriented survey. *Artificial Intelligence*, 132 :39–103, 2001.

[9] B. Bouzy and B. Helmstetter. Monte Carlo go developments. In Ernst A. Heinz H. Jaap van den Herik, Hiroyuki Iida, editor, *10th Advances in Computer Games*, pages 159–174, Graz, 2003. Kluwer Academic Publishers.

[10] T. Cazenave. Automatic acquisition of tactical go rules. In *3rd Game Programming Workshop in Japan*, pages 10–19, Hakone, 1996.

[11] T. Cazenave. Generation of patterns with external conditions for the game of go. In B. Monien H.J. van den Herik, editor, *Advances in Computer Games*, volume 9, University of Limburg, Maastricht, 2001.

[12] CISIA CERESTA, editor. *Aide-mémoire statistique.* 1999.

[13] M. Müller. Computer go. *Artificial Intelligence*, 134 :145–179, 2002.

[14] M. Müller. Position evaluation in computer go. *ICGA Journal*, 25(4) :219–228, December 2002.

[15] J. Schaeffer and J. van den Herik. Games, Computers, and Artificial Intelligence. *Artificial Intelligence*, 134 :1–7, 2002.

[16] R. Sutton and A. Barto. *Reinforcement Learning : an introduction.* MIT Press, 1998.

[17] E. van der Werf, J. Uiterwijk, E. Postma, and J. van den Herik. Local move prediction in Go. In Yngvi Björnsson J. Schaeffer, M. Müller, editor, *Computers and Games*, volume 2883 of *Lecture Notes in Computer Science*, pages 393–412. Springer, 2002.

[18] E. van der Werf, J. Uiterwijk, and J. van den Herik. Learning to score final positions in the game of go. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *Advances in Computer Games, Many Games, Many Challenges*, volume 10, pages 143–158. Kluwer Academic Publishers, 2003.

[19] E. van der Werf, M. Winands, J. van den Herik, and J. Uiterwijk. Learning to predict life and death from go game records. In *7th Joint Conference on Information Sciences*, pages 501–504, Raleigh, 2003.

Summary

This paper describes the generation and utilisation of a pattern database for 19x19 go with the K-nearest-neighbor representation. Patterns are generated by browsing recorded games of professional players. Meanwhile, their matching and playing probabilities are estimated. The database created is then integrated into an existing go program, Indigo, either as an opening book or as an enrichment of other pre-existing databases used by Indigo move generator. The improvement brought about by the use of this pattern database is estimated at 15 points on average, which is significant in go standards.