Introduction de connaissances expertes en Bandit-Based Monte-Carlo Planning avec application au Computer-Go

Louis Chatriot, Sylvain Gelly, Jean-Baptiste Hoock, Julien Pérez, Arpad Rimmel, Olivier Teytaud

Introduction de connaissances expertes en Bandit-Based Monte-Carlo Planning avec application au Computer-Go

Louis Chatriot, Sylvain Gelly, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, Olivier Teytaud

TAO (Inria), LRI, UMR 8623 (CNRS - Univ. Paris-Sud),
bat 490 Univ. Paris-Sud 91405 Orsay, France, teytaud@lri.fr

Résumé : Nous présentons notre travail sur l'introduction de l'expertise go dans l'algorithme de BBMCP appliqué au jeu de go. Notre contribution inclue (i) les livres d'ouvertures (ii) le biais dans l'arbre des possibles (iii) l'amélioration de la partie Monte-Carlo et la mise en évidence des facteurs critiques de succès tels que le risque de perte de diversité dans le jeu aléatoire, l'utilité des contre exemples pour la conception de Monte-Carlo ou pour l'introduction du biais. Le programme résultant de cette étude a récemment obtenu une victoire en condition de temps standard face à un joueur professionnel de Go en 9x9.

1 Introduction

Les diverses définitions des termes du jeu de go utilisées dans cet article peuvent être trouvées sur le site internet `http://senseis.xmp.net`

Le Bandit Based Monte Carlo Planning, BBMCP Kocsis & Szepesvari (2006), est un outil récent pour les tâches de planification difficiles. D'impressionnants résultats ont déjà été publiés Coulom (2006); Gelly & Silver (2007). Dans ce papier, nous présentons l'introduction de connaissance d'experts pour un problème spécifique de prise de décision, le jeu de Go, un jeu séquentiel à somme nulle. Après avoir présenté les principes de l'algorithme BBMCP, nous présenterons notre travail sur les trois grandes parties de l'algorithme : (1) les livres d'ouverture (2) la partie bandit (les simulations dans l'arbre) (3) l'estimation de Monte-Carlo. Toutes les modifications sont développées dans MoGo qui gagna en 2007 la première partie en Blitz en 9x9 contre un joueur professionnel et en 2008 la première partie en temps standard (30 minutes par coté) contre un joueur professionnel.

L'algorithme BBMCP consiste à construire un arbre dans lequel chaque noeud est une situation de l'environnement considéré et les branches sont des actions pouvant être prises par l'agent. Le point important du BBMCP est que l'arbre est fortement déséquilibré : un biais est appliqué en faveur des parties importantes de l'arbre. Ainsi, l'intérêt est mis aux zones de l'arbre ayant une plus forte espérance de gain. Plusieurs algorithmes ont été proposés : UCTKocsis & Szepesvari (2006) (Upper Confidence Trees, i.e. UCB applied to Trees), qui porte son attention sur les noeuds de l'arbre ayant le plus grand nombre de simulations gagnantes *plus* un terme favorisant les noeuds peu explorés, AMAF Gelly & Silver (2007) (All Moves As First), qui fait un compromis entre le score à la UCB1 et des informations heuristiques obtenues par permutations de simulations, et BAST Coquelin & Munos (2007) (Bandit Algorithm for Search in Trees), qui se focalise quant à lui sur le nombre total de noeuds dans l'arbre. D'autres algorithmes proches sont également disponibles dans Chaslot *et al.* (2007).

Dans le contexte du jeu de Go, les noeuds de l'arbre incorporent la configuration du tableau de jeu qu'ils représentent (aussi appelée coloration du plateau), et des statistiques telles que le nombre de victoires obtenues et le nombre total de parties jouées traversant ce noeud. Comme expliqué dans l'Algorithme 1, BBMCP utilise ces statistiques afin d'étendre itérativement l'arbre des possibles dans la région où la plus forte espérance de gain aura été calculée. Après chaque simulation à partir de la racine de l'arbre, les statistiques de gain et de perte sont mises à jour dans tous les noeuds ayant été parcourus et un nouveau noeud est créé dans l'arbre ; ce nouveau noeud correspond au premier noeud visité lors de la partie simulation Monte-Carlo (i.e. la première situation rencontrée dans la simulation et qui n'était pas encore dans l'arbre).

Ainsi, afin de concevoir des simulations à partir de la racine de l'arbre (la position courante) jusqu'à la position finale (victoire ou défaite), plusieurs décisions doivent être prises : (i) dans l'arbre, jusqu'à

parvenir à une feuille (ii) hors de l'arbre, jusqu'à ce que la partie soit terminée. Les buts de ces deux choix de transition sont différents. Dans l'arbre, la transition doit assurer un bon compromis entre exploration et exploitation. Hors de l'arbre, les simulations, habituellement nommées simulation Monte-Carlo, doivent permettre d'obtenir une bonne estimation de la probabilité de victoire.

L'Algorithme 1 résume le planning par méthode Monte-Carlo ; une version plus détaillée est donnée en Algorithme 4.

Algorithm 1 Pseudo-code d'un algorithme BBMCP appliqué à un jeu à somme nulle (comme le Go ou les échecs). T est un arbre de positions, avec chaque noeud contenant des statistiques (nombre de parties gagnées et perdues dans les simulations passant par ce noeud). En ce qui concerne la décision à la toute fin du pseudo-code, choisir le coup le plus simulé est la solution la plus sure ; d'autres solutions comme le choix du meilleur "ratio nombre de parties gagnées sur nombre de parties total" ne sont pas robustes car il peut y avoir très peu de simulations. Ici la récompense pour chaque partie est binaire (gagnée ou perdue) mais des distributions arbitraires peuvent être utilisées.

Initialiser T à un noeud unique, représentant l'état courant.
while Temps restant > 0 **do**
  Simuler une partie jusqu'à une feuille (une position) L de T (à l'aide de l'algorithme de bandit).
  Choisir un fils (un successeur) L' de L.
  Simuler une partie depuis la position L' jusqu'à la fin de la partie.
  Développer l'arbre : ajouter L' en tant que fils de L dans T.
  Mettre à jour les statistiques dans tout l'arbre. Dans UCT, chaque noeud connaît combien de simulations gagnées (depuis ce noeud) ont été jouées ainsi que le nombre de simulations totales. Pour d'autres formes d'exploration d'arbre, il peut y avoir besoin de plus d'informations (information heuristique dans le cas d'AMAF, nombre total de noeuds dans le cas de BAST Coquelin & Munos (2007)).
end while
Retourner le coup qui a été simulé le plus souvent à partir de la racine.

La fonction utilisée pour prendre les décisions hors de l'arbre est définie dans l'algorithme 2. Un atari se produit lorsqu'une chaîne (un groupe de pierres) peut être capturé en un coup. Des connaissances de Go ont été ajoutées dans cette partie, sous la forme de patterns 3x3 conçus pour jouer des parties plus réalistes.

Algorithm 2 Algorithme permettant de choisir un coup dans une simulation MC dans le cas du jeu de Go.

if le dernier coup est un atari **then**
  Sauver les pierres en atari.
else
  if il y a un emplacement vide parmi les 8 emplacements autour du dernier coup qui correspond à un pattern **then**
    Jouer au hasard sur l'un de ces emplacements (sortir).
  else
    if il y a un coup qui capture des pierres **then**
      Capturer les pierres (sortir).
    else
      if il reste des coups légaux **then**
        Jouer au hasard un coup légal (sortir).
      else
        Renvoyer "passe".
      end if
    end if
  end if
end if

La fonction utilisée pour prendre les décisions dans l'arbre est présentée dans l'algorithme 3. Cette fonction est l'élément principal de notre programme : elle décide de la direction dans laquelle l'arbre sera étendu. Il existe plusieurs formules différentes ; toutes sont basées sur l'idée de compromis entre exploitation (simuler dans le sens des coups à succès) et l'exploration (simuler les coups qui ne l'ont pas encore été ou peu).

Algorithm 3 Algorithme permettant de choisir un coup dans l'arbre, pour un jeu à somme nulle avec une récompense binaire (l'extension pour le cas de distributions arbitraires est directe). $Sims(s,d)$ est le nombre de simulations commençant à s avec comme premier coup d. Le nombre total de simulations pour une situation s est $Sims(s) = \sum_d Sims(s,d)$. Nous présentons ici différentes formules pour calculer le score (voir Lai & Robbins (1985); Auer *et al.* (2002); Gelly & Silver (2007) pour UCB1, UCB-Tuned et AMAF respectivement) ; d'autres variantes très importantes (pour le cas de domaines infinis, d'un très grand nombre de bras, de suppositions spécifiques) peuvent être trouvées dans Banks & Sundaram (1992); Agrawal (1995); Dani & Hayes (2006); Coquelin & Munos (2007); Chaslot *et al.* (2007).

Fonction $decision = Bandit(situation\ s\ in\ the\ tree)$.
for d dans un ensemble de décisions possibles **do**
Soit $\hat{p}(d) = Wins(s,d)/Sims(s,d)$.
switch (formule de bandit) :
- **UCB1** : calculer $score(d) = \hat{p}(d) + \sqrt{2\log(Sims(s))/Sims(s,d)}$.
- **UCB-Tuned.1** : calculer $score(d) = \hat{p}(d) + \sqrt{\hat{V}\log(Sims(s))/Sims(s,d)}$ avec $\hat{V} = \max(0.001, \hat{p}(d)(1-\hat{p}(d)))$.
- **UCB-Tuned.2** : calculer $score(d) = \hat{p}(d) + \sqrt{\hat{V}\log(Sims(s))/Sims(s,d)} + \log(Sims(s))/Sims(s,d)$ avec $\hat{V} = \max(0.001, \hat{p}(d)(1-\hat{p}(d)))$.
- **exploration guidée par AMAF** : Calculer $score(d) = \alpha(d)\hat{p}(d) + (1-\alpha(d))\hat{\hat{p}}(d)$ avec :
  - $\hat{\hat{p}}(d)$ la proportion de simulations gagnées parmis les simulations AMAF utilisant la décision d dans la situation s ;
  - $\alpha(d)$ un coefficient dépendant de $Sims(s,d)$ (voir Gelly & Silver (2007)).

end switch
end for
Retourner $\arg\max_d\ score(d)$.

L'algorithme AMAF a été dérivé de Gelly & Silver (2007). Les simulations d'AMAF sont créées par permutation de coup dans la simulation réelle. Un progrès important, le "Progressive Widening", de l'algorithme 3 consiste à considérer uniquement les $K(n)$ meilleurs coups (selon une heuristique choisie) à la n-ième simulation d'un noeud donné. Avec $K(n)$ une application non-décroissante de $\mathbb{N}$ dans $\mathbb{N}$.

Algorithm 4 Algorithme de planification Monte-Carlo plus détaillé.

Initialiser T à un noeud unique, représentant l'état courant. T est un arbre de positions.
while Temps restant > 0 **do**
Simuler une partie jusqu'à une feuille (une position) L de T (à l'aide de l'algorithme de bandit jusqu'à ce qu'on arrive à une feuille, voir algorithme 3).
Choisir un fils (un successeur) L' de L, éventuellement à l'aide de connaissances a priori.
Simuler une partie (voir algorithme 2) depuis la position L' jusqu'à la fin de la partie.
Développer l'arbre : ajouter L' en tant que fils de L dans T.
Mettre à jour les statistiques dans tout l'arbre.
end while
Retourner le coup qui a été simulé le plus souvent à partir de la racine.

Malheureusement, plusieurs problèmes surviennent : premièrement, la partie Monte Carlo n'est pas encore complètement comprise. En effet, de nombreuses discussions autour de ce sujet ont été publié sur la mailing list computer-go, mais personne n'est encore parvenu à proposer un critère pour déterminer sans expérimentations coûteuses de l'algorithme BBCMP, ce qu'est un bon simulateur de Monte Carlo. Ainsi, utiliser un bon joueur de go, tel qu'un programme comme GnuGo, dans ce but, réduit fortement les performances du programme, même à nombre de simulations fixées. Une explication intuitive de ce phénomène pourrait être qu'un simulateur de Monte Carlo fortement biaisé mène à une estimation incorrecte de l'espérance de victoire pour les noeuds. Deuxièmement, la partie Bandit est encore trouble. Des implémentations utilisent encore la formule UCB Auer *et al.* (2002) mais avec une petite constante d'exploration, ainsi les coups avec le meilleur résultats empiriques sont utilisés, indépendamment du nombre de simulations, jusqu'à ce que la probabilité de victoire deviennent plus petite que la probabilité de victoire des noeuds

Sans livre d'ouverture	livre standard	livre cosmique
55.5 % $\pm$ 1%	53.5 % $\pm$ 1%	56.6% $\pm$ 1.3%

TAB. 1 – Résultats expérimentaux pour les livres d'ouverture. L'adversaire est gnugo3.6, niveau par défaut. MoGo joue en tant que noir.

non parcourus. Nous allons présenter à présent (i) les améliorations pour la partie arbre de BBMCP, voir l'algorithme 3 et (ii) les améliorations de la partie Monte-Carlo, voir l'algorithme 2.

2 Livres d'ouvertures

Les livres d'ouvertures constituent la méthode la plus simple d'introduire de la connaissance experte au sein de l'algorithme BBMCP appliqué au jeu de go. En particulier, il est connu que BBMCP fait de mauvais choix en début de partie. Ceci semble simple à améliorer a priori. Nous avons donc développé (i) un livre d'ouvertures standard basé sur des parties de professionnels (ii) un ensemble de parties spécifiques proches du style de jeu de MoGo, privilégiant un contrôle rapide du centre du plateau, appelé dans la littérature "style cosmique". Les résultats expérimentaux sont présentés dans le Tableau 1, avec 5000 simulations par coup.

L'amélioration est faible, voire négative dans le cas des livres d'ouvertures classiques.

Nous avons également généré des livres d'ouvertures automatiquement en faisant jouer MoGo contre lui-même. Ceux-ci se sont révélés efficaces (approximativement 60% contre la version initiale) mais uniquement contre une version de MoGo avec un niveau modéré. En effet, ces mouvements obtenus par des parties de MoGo avec des longs temps de simulation contre un MoGo aux décisions bruitées afin de diversifier les situations d'ouvertures, ne sont pas adaptés à la version parallèle de MoGo. En effet, nous aurions besoin de milliers d'heures de calcul pour générer, par ce type de parties, un livre d'ouverture suffisamment performant pour la version parallèle de MoGo, sans quoi la taille du livre d'ouverture resterait trop restreinte.

3 Amélioration des simulations dans l'arbre

Dans la partie arbre, le but n'est pas de trouver de bons coups mais de trouver les coups qui valent la peine d'être explorés. Dans beaucoup de programmes, cela est fait par des algorithmes de type UCT, i.e. en choisissant le coup tel que la quantité suivante soit maximale :

$$\underbrace{w(move)/n(move)}_{Exploitation} + C \underbrace{\sqrt{\log(\sum_{m \in M} n(m))/n(move)}}_{Exploration} \quad (1)$$

où :

- $n(move)$ est le nombre de fois où le coup a été exploré dans cette situation ;
- $w(move)$ est le nombre de fois où le coup a été essayé et a conduit à une victoire ;
- M est l'ensemble des coups possibles dans cette situation.

Une amélioration classique consiste à remplacer Eq. 1 par certaines versions améliorées, typiquement UCB-Tuned Auer *et al.* (2002), c'est une solution générale qui n'est pas spécifique aux jeux et qui peut être appliquée à la planification. Dans le cas des jeux, des formules comme les valeurs AMAF dans l'algorithme 3 peuvent être utilisées pour biaiser le terme d'exploration vers les coups qui paraissent intéressants selon les statistiques "AMAF" Bruegmann (1993). Nous voulons introduire ici un biais par dessus ce biais en ajoutant des simulations AMAF virtuelles.

La suppression de l'arbre des coups vraissemblablement mauvais a été expérimenté ; cependant, de tels suppressions sont dangereuses au Go, car les exceptions aux règles sont fréquentes. Nous nous contenterons donc d'introduire un biais en faveur des coups "intéressants" en ajoutant des simulations virtuelles : au lieu de créer de nouveaux noeuds dans l'arbre sans aucune simulation, nous introduisons artificiellement des gains pour des coups qui sont supposés être bons ou, plus précisément, pour des coups qui sont supposés valoir la peine d'être explorés. Inversement, nous introduirons des pertes pour les coups supposés faibles. D'autres solutions pour inclure des connaissances expertes existent :

– Progressive widening Coulom (2007) ou progressive unpruning Chaslot *et al.* (2007) : le coup joué est choisi *parmi les n coups* ayant le score le plus élevé (eq. 1), n dépendant de connaissances expertes ;
– Progressive bias Chaslot *et al.* (2007) : la formule 1 est modifiée par un terme $+heuristic\ value(move)/n(move)$;
– First play urgency Wang & Gelly (2007) : le score représenté par la formule 1 est utilisé pour les coups m tels que $n(m) > 0$, et une valeur à priori dépendant des connaissances expertes est utilisée pour les autres coups ;
– MoGo incorpore un mélange de "Progressive bias" et de "Progressive widening" par l'ajout d'un bonus diminuant avec le nombre de simulations pour des coups proches au dernier coup joué. La distance est conçue à travers l'expertise Go : c'est une distance topologique qui suppose la distance 1 entre toutes les pierres d'une chaîne, quoiqu'elle peut être la taille de cette chaîne.

Divers éléments sur des connaissances expertes déjà publiés (e.g. Coulom (2007); Bouzy & Chaslot (2005)) incluent :

– les formes : dans Coulom (2007); Bouzy & Chaslot (2005), les formes sont générées automatiquement à partir d'un ensemble de données. Nous présentons ci-dessous les connaissances expertes introduites sur les formes ;
– les coups de capture (en particulier, le coup voisin à une nouvelle chaîne en atari), chaîneion (notamment le shisho), l'évitement de se mettre en atari, l'atari (en particulier quand il y a un ko), la distance au bord (distance optimale $= 3$ en 19x19 Go), une petite distance au coup précédemment joué, une petite distance à l'avant-dernier coup, et aussi la caractéristique reflétant le fait que la probabilité (selon les simulations de Monte-Carlo) d'un emplacement à être de sa propre couleur à la fin de la partie est $\simeq 1/3$.

Les outils suivants sont utilisés dans nos implémentations en 19x19, et améliorent les résultats de l'heuristique AMAF :

– La ligne de territoire (i.e. ligne numéro 3) : 1.333 gains dans les simulations AMAF sont ajoutés ;
– La ligne de la mort (i.e. première ligne) : 1.333 pertes dans les simulations AMAF sont ajoutés ;
– Peep-connect (ie. connecter deux chaînes quand l'adversaire menace de couper) : 1 gain dans les simulations AMAF est ajouté ;
– Hane (un coup qui "contourne" une ou plusieurs pierres de l'adversaire) : 1 gain dans les simulations AMAF est ajouté ;
– Connect : 1 gain dans les simulations AMAF est ajouté ;
– Wall : 0.666 gains dans les simulations AMAF sont ajoutés ;
– Bad Kogeima(même motif que le mouvement du cavalier aux échecs) : 0.5 pertes dans les simulations AMAF sont ajoutés ;
– Empty triangle (trois pierres de même couleur formant un triangle sans pierre adverse dans l'angle formé) : 1 perte dans les simulations AMAF est ajoutée.

Ces formes sont illustrées sur la Figure 1. Avec un réglage naïf des paramètres à la main, elles fournissent 63.9$\pm$0.5 % de victoires contre la version sans ces améliorations. Nous sommes optimistes sur le fait que le réglage des paramètres améliorera considérablement les résultats. D'ailleurs, dans les premiers développements de MoGo, certains bonus de "coupe" sont déjà inclus (i.e., les avantages de jouer à des emplacements qui vérifie des motifs de "coupe", i.e. des motifs pour lesquels un emplacement empêche l'adversaire de connecter deux groupes).

4 Amélioration des simulations Monte-Carlo (MC)

On ne sait pas encore comment créer un bon simulateur Monte-Carlo pour BBMCP dans le cas du Go. De nombreuses personnes ont essayé d'améliorer le simulateur en augmentant sa force en tant que joueur indépendant, mais il a clairement été prouvé dans Gelly & Silver (2007) que ce n'est pas un bon critère : un simulateur MC_1 qui joue beaucoup mieux qu'un simulateur MC_2 peux conduire à de très mauvais résultats une fois utilisé avec BBMCP et ceci même en comparant à nombre de simulations identique. Certains simulateurs ont été appris à partir de base de données Coulom (2007), mais les résultats sont fortement améliorés par un réglage manuel des paramètres. Les raisons expliquant qu'un simulateur MC va être efficace étant encore inconnues, il est nécessaire d'expérimenter toute modification de manière intensive afin de la valider.

De nombreuses formes sont définies dans Bouzy (2005); Wang & Gelly (2007); Ralaivola *et al.* (2005).

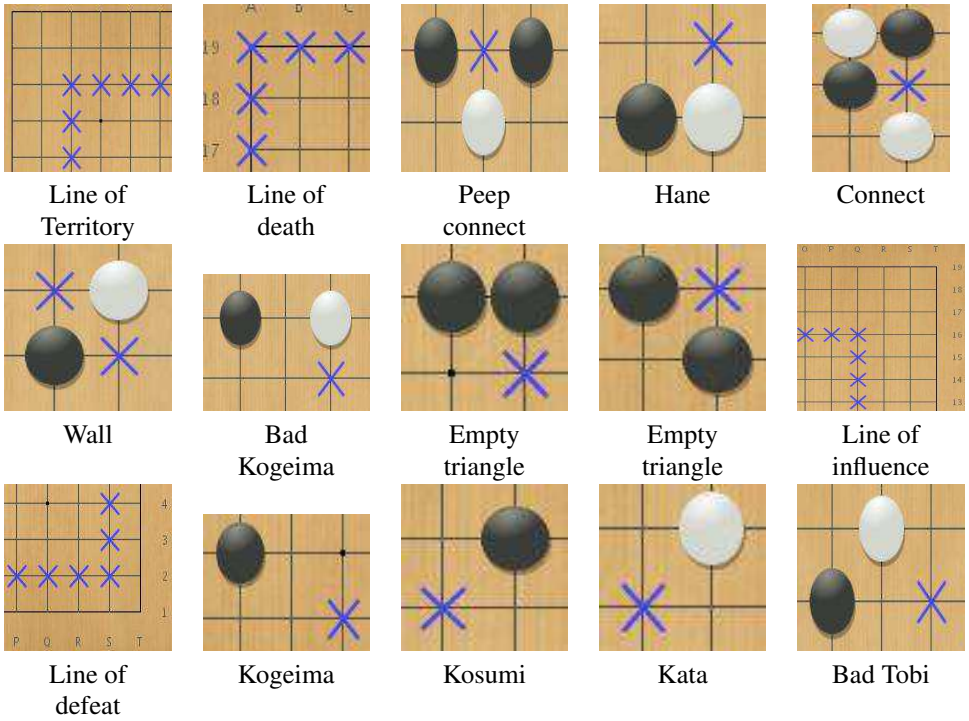


FIG. 1 – Nous présentons ici les motifs exacts pour lesquels s'appliquent les bonus /malus. Dans tous les cas, les motifs sont présentés avec trait aux Noirs : les formes s'appliquent pour un coup Noir à une des croix. Les motifs avec couleurs inversées s'appliquent bien sûr aux Blancs. Dans le cas du Bad Kogeima, il ne doit pas y avoir de pierres noires autour de la croix pour que le bonus s'applique. Dans le cas du Kosumi et du Kogeima, il ne doit pas y avoir de pierres blanches entre la pierre noire et la croix. Le motif "seul sur la première ligne" s'applique seulement si les 5 emplacements autour du coup considéré sont vides. Dans le cas du bad tobi, il ne doit pas y avoir de pierre noire parmi les 8 emplacements autour de la croix. Dans le cas du Hane, la pierre blanche ne doit avoir aucune pierre amie parmi les 4 emplacements voisins.

Wang & Gelly (2007) utilise des patterns et des connaissances expertes comme expliqué dans l'algorithme 2. Nous présentons ci-dessous nouvelles améliorations.

4.1 fillboard : on remplit les espaces vides du plateau en premier

Le principe de cette modification est de jouer en priorité aux emplacements du plateau où il n'y a pas encore de pierre autour. L'idée est de jouer de manière plus uniforme dans ces zones. Essayer chaque emplacement du plateau serait trop coûteux en temps, c'est pourquoi quoi nous utilisons la procédure suivante. Une position du plateau est choisie au hasard ; si les 8 emplacements contiguës sont vides alors le coup est joué, sinon on essaye les $N-1$ positions suivantes du plateau. Cette modification introduit plus de diversités dans les simulations. Ceci est dû au fait que le simulateur MC de MoGo utilise beaucoup de patterns. Or quand un pattern correspond, le coup est joué. Les simulations n'ont donc qu'un petit nombre de manières de commencer, en particulier en début de partie où il n'y a que peu de pierres sur le plateau. Cette modification intervenant avant la détection de patterns, elle permet d'obtenir des simulations plus diversifiées (voir figure 2 (gauche)). L'algorithme détaillé est présenté dans l'algorithme 5.

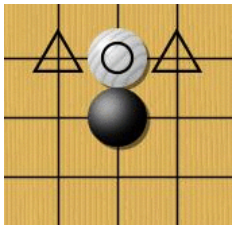
Les expériences sont présentées dans la figure 2 (droite).

4.2 Le problème du "Nakade"

. Une faiblesse connue de MoGo, ainsi que de beaucoup de programme BBMCP, est que les "nakade" ne sont pas gérés correctement. Nous utiliserons le terme *nakade* pour désigner une situation dans laquelle un groupe encerclé possède un unique espace interne enfermé à partir duquel le joueur ne pourra pas créer deux yeux si l'adversaire répond correctement. Le groupe est donc mort mais le simulateur MC utilisé dans

Algorithm 5 Algorithme permettant de choisir un coup dans les simulations MC, comprenant l'amélioration "fillboard". Nous avons également expérimenté avec une contrainte de 4 emplacements libres au lieu de 8 mais les résultats furent décevants.

if le dernier coup est un atari **then**
  Sauver les pierres en atari.
else
  partie "Fillboard".
  for $i \in \{1, 2, 3, 4, \ldots, N\}$ **do**
    Tirer une position x au hasard sur le plateau.
    Si x est un emplacement vide et que les 8 emplacements autour sont également vide, jouer x (sortir).
  end for
  Fin de la partie "fillboard".
  if il y a un emplacement vide parmi les 8 emplacements autour du dernier coup qui correspond à un pattern **then**
    Jouer au hasard sur l'un de ces emplacements (sortir).
  else
    if il y a un coup qui capture des pierres **then**
      Capturer les pierres (sortir).
    else
      if il reste des coups légaux **then**
        Jouer au hasard un coup légal (sortir).
      else
        Renvoyer "passe".
      end if
    end if
  end if
end if



9x9 board		19x19 board	
Nb de simulations par coup ou temps/coup	Taux de succès	Nb de simulations par coup ou temps/coup	Taux de succès
10 000	52.9 % ± 0.5%	10000	49.3 ± 1.2 %
5s/coup, 8 coeurs	54.3 % ± 1.2 %	5s/coup, 8 coeurs	77.0 % ± 3.3 %
100 000	55.2 % ± 1.4 %	100 000	73.7 % ± 2.9%
200 000	55.0 % ± 1.1 %	200 000	78.4 % ± 2.9 %

FIG. 2 – Gauche : perte de diversité quand on n'utilise pas l'option "fillboard" : la pierre blanche est la dernière pierre jouée, une simulation MC pour le joueur noir commencera obligatoirement par l'une des cases marquée d'un triangle. Droite : Résultats associés à la modification "fillboard". Comme la modification induit un surcoût computationnel, les résultats sont meilleurs pour un nombre de simulations par coup fixé ; cependant, l'amélioration est clairement significative. Le surcoût computationnel est réduit lorsque l'on utilise plusieurs coeurs : en effet, il y a moins de concurrence pour l'accès mémoire lorsque les simulations sont plus longues. C'est pourquoi la différence en temps entre des simulations simples et complexes se réduit avec le nombre de coeurs. Cet élément montre l'intérêt accru des simulations complexes dans le cadre de la parallélisation.

la version de MoGo de référence (Algorithme 2) va estimer que le groupe vit avec une forte probabilité. En effet, le simulateur ne répondra pas les coups considérés comme évident pour un joueur et peut donc conduire à la survie du groupe. L'arbre va donc sous estimer le danger correspondant à cette situation et éventuellement se développer dans cette direction. La partie MC doit donc être modifiée pour que la probabilité de gagner prenne en compte ces situations de *nakade* .

Il est intéressant de noter que MoGo étant principalement développé en jouant contre lui-même, cette faiblesse n'est apparue qu'une fois trouvée par un humain (voir le mail de D.Fotland "UCT and solving life

and death" sur la mailing list computer-go).

Il serait théoriquement possible d'encoder dans les simulations MC un vaste ensemble de comportements relatifs au *nakade* , mais cette approche a deux faiblesses : (i) cela serait coûteux en temps et les simulations doivent être rapides (ii) changer le comportement du simulateur MC de manière trop abrupte conduit généralement à des résultats décevants. C'est pourquoi la modification a été créée : si on trouve un ensemble d'exactement 3 intersections libres entourées de pierres adverses, on joue au milieu (le point vital). Cela permet de fortement diminuer la probabilité de survie de la plupart des situations de *nakade* . Le nouvel algorithme est présenté dans l'algorithme 6.

Algorithm 6 Nouveau simulateur MC, permettant d'éviter le problème de *nakade* .

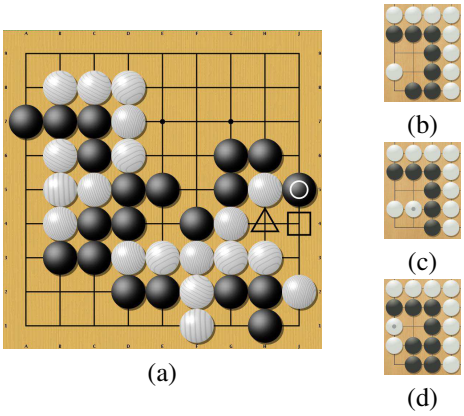
if le dernier coup est un atari **then**
  Sauver les pierres en atari.
else
  Début de la modification *nakade*
  for x parmis l'un des 4 emplacements libres autour du dernier coup joué **do**
    if x fait parti d'un trou de 3 emplacements contigus entourés de pierres adverses **then**
      Jouer au centre du trou (sortir).
    end if
  end for
  Fin de la modification *nakade*
  partie "Fillboard".
  for $i \in \{1, 2, 3, 4, \ldots, N\}$ **do**
    Tirer une position x au hasard sur le plateau.
    Si x est un emplacement vide et que les 8 emplacements autour sont également vide, jouer x (sortir).
  end for
  Fin de la partie "fillboard".
  if il y a un emplacement vide parmi les 8 emplacements autour du dernier coup qui correspond à un pattern **then**
    Jouer au hasard sur l'un de ces emplacements (sortir).
  else
    if il y a un coup qui capture des pierres **then**
      Capturer les pierres (sortir).
    else
      if il reste des coups légaux **then**
        Jouer au hasard un coup légal (sortir).
      else
        Renvoyer "passe".
      end if
    end if
  end if
end if

Cette approche est validée par deux expériences différentes : (i) des positions connues où la version de MoGo de référence ne choisit pas le bon coup (figure 3) (ii) des parties nouveau MoGo contre MoGo de référence (table 2).

4.3 Améliorations cumulées

Les deux améliorations de la partie Monte-Carlo sont indépendantes. Le tableau suivant montre le pourcentage de victoire de chaque amélioration et des deux cumulées contre la version de référence avec 10 000 simulations par coup.

Amélioration	Taux de succès
fillboard	52.9 % $\pm$ 0.5%
nakade	52.8 % $\pm$ 0.5 %
les deux	57.2 % $\pm$ 1.0 %



(a) (b) (c) (d)

FIG. 3 – figure (a) (une partie réelle jouée et perdue par MoGo), MoGo (blanc) sans la modification pour les *nakade* joue H4 ; noir répond J4 et le groupe F1 est mort (MoGo perd). Après la modification, MoGo joue J4 qui est le bon coup car il permet de vivre après un ko. Les exemples (b), (c) et (d) sont des situations similaires dans lesquelles MoGo ne réalise pas que son groupe est mort. Dans chacun des cas, la modification permet de résoudre le problème.

Nb de simulations par coup	Taux de succès	Nb de simulations par coup	Taux de succès
plateau 9x9		plateau 19x19	
10000	52.8 % $\pm$ 0.5%		
100000	55.6 % $\pm$ 0.6 %	100 000	53.2 % $\pm$ 1.1%
300000	56.2 % $\pm$ 0.9 %		
5s/coup, 8 coeurs	55.8 % $\pm$ 1.4 %		
15s/coup, 8 coeurs	60.5 % $\pm$ 1.9 %		
45s/coup, 8 coeurs	66.2 % $\pm$ 1.4 %		

TAB. 2 – Validation expérimentale de la modification *nakade* : MoGo modifié contre MoGo de référence. Il apparaît que plus le nombre de simulations est important (ce qui est directement lié au niveau), plus l'impact de la modification est important.

5 Conclusion

Nos conclusions sont les suivantes :

- Les ouvertures sont un problème difficile : utiliser des ouvertures de joueurs professionnels est une mauvaise idée. Le niveau et le style des premiers coups doivent être consistants avec ceux des coups suivants. Générer ces ouvertures en faisant jouer MoGo contre lui-même est une approche stable et efficiente. Cependant, pour la version parallèle de MoGo, nous aurions besoin d'un cluster aussi gros que celui que nous utilisons dans les parties réelles, et cela pour un grand nombre de positions. Malheureusement, nous n'avons pas accès à un si grand nombre d'heures-CPU.
- Introduire de l'expertise Go en réduisant la largeur de l'arbre est dangereux, et donne des résultats négatifs, notamment lorsque le nombre de simulations est élevé. Une autre approche qui consiste à augmenter le nombre initial de simulations gagnées ou perdues est fiable et améliore grandement le programme.
- Nous n'avons pas d'idée de critères formels permettant de juger la partie Monte-Carlo de l'algorithme. Nous utilisons des modifications qui améliorent sa performance, mais un il est intéressant de noter que la variance ne devrait pas être trop réduite. En augmentant le nombre possible de coups, la modification "fillboard" permet un nombre de futurs possibles plus large. Les patterns permettent d'éviter les simulations absurdes, mais cela peut aussi réduire fortement la variance du résultat.
- Se concentrer sur des contre-exemples est efficace. La solution au problème du nakade a été trouvée en observant le comportement de la partie Monte-Carlo dans des situations de nakade. De ce point de vue, des ensembles de tests tels ceux proposés par M. Yamato (`http://computer-go.org/pipermail/computer-go/2008-April/014773.html`) sont un moyen pratique d'amélio-

rer le programme.
– Dans le futur, il sera important de (i) réduire l'agressivité de BBMCP en changeant la partie Monte-Carlo, de la même manière que le "style cosmic" avait été réduit par la modification "fillboard", (ii) améliorer MoGo dans les situations de ko, probablement en rajoutant un biais dans l'arbre, (iii) introduire plus de diversité dans les simulations, par exemple en introduisant différentes formes de Monte Carlo, ou en faisant dépendre le coup joué dans la simulation de coups antérieurs au dernier coup joué et (iv) développer des bases de données d'ouvertures meilleures et plus exhaustives.

Remerciements

Nous remercions Rémi Munos, Yizao Wang, Rémi Coulom, Guillaume Chaslot, Tristan Cazenave, Jean-Yves Audibert, David Silver, Martin Mueller,KGS, Cgos, ainsi que toute la mailing list computer-go pour toutes les discussions intéressantes que nous avons pu avoir. Un grand merci à la fédération française de Go et à Recitsproque pour avoir organisé un match officiel contre un humain de haut niveau ; un grand merci également à tous les joueurs de la fédération française de Go qui ont accepté de jouer et de commenter des parties test contre MoGo, et à Catalin Taranu, joueur professionnel 5e dan, pour ses commentaires après le challenge IAGO. Nous remercions Antoine Cornuejols pour ses commentaires très utiles sur la première version de cet article.

Références

AGRAWAL R. (1995). The continuum-armed bandit problem. *SIAM J. Control Optim.*, **33**(6), 1926–1951.

AUER P., CESA-BIANCHI N. & FISCHER P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, **47**(2/3), 235–256.

BANKS J. S. & SUNDARAM R. K. (1992). Denumerable-armed bandits. *Econometrica*, **60**(5), 1071–96. available at http ://ideas.repec.org/a/ecm/emetrp/v60y1992i5p1071-96.html.

BOUZY B. (2005). Associating domain-dependent knowledge and monte carlo approaches within a go program. In K. CHEN, Ed., *Information Sciences, Heuristic Search and Computer Game Playing IV*, volume 175, p. 247–257.

BOUZY B. & CHASLOT G. (2005). Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go. *In G. Kendall and Simon Lucas, editors, IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK*, p. 176–181.

BRUEGMANN B. (1993). Monte carlo go. *Unpublished*.

CHASLOT G., WINANDS M., UITERWIJK J., VAN DEN HERIK H. & BOUZY B. (2007). Progressive strategies for monte-carlo tree search. In P. WANG & OTHERS, Eds., *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, p. 655–661 : World Scientific Publishing Co. Pte. Ltd.

COQUELIN P.-A. & MUNOS R. (2007). Bandit algorithms for tree search. In *Proceedings of UAI'07*.

COULOM R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*.

COULOM R. (2007). Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands*.

DANI V. & HAYES T. P. (2006). Robbing the bandit : less regret in online geometric optimization against an adaptive adversary. In *SODA '06 : Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, p. 937–943, New York, NY, USA : ACM Press.

GELLY S. & SILVER D. (2007). Combining online and offline knowledge in uct. In *ICML '07 : Proceedings of the 24th international conference on Machine learning*, p. 273–280, New York, NY, USA : ACM Press.

KOCSIS L. & SZEPESVARI C. (2006). Bandit-based monte-carlo planning. In *ECML'06*, p. 282–293.

LAI T. & ROBBINS H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, **6**, 4–22.

RALAIVOLA L., WU L. & BALDI P. (2005). Svm and pattern-enriched common fate graphs for the game of go. In *Proceedings of ESANN 2005*, p. 485–490.

WANG Y. & GELLY S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, p. 175–182.