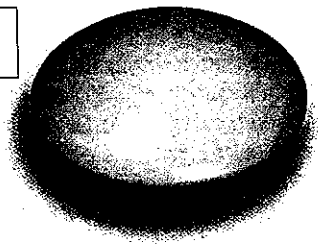
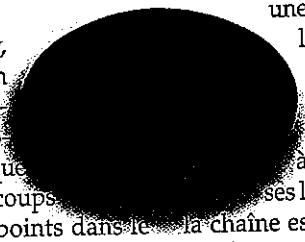
L'ordinateur, champior

Le jeu de go est d'une richesse telle que les programmes informatiques ne peuvent encore rivaliser avec les joueurs professionnels. Mais l'introduction de techniques probabilistes est en train de changer la donne. L'ordinateur champion de go est peut-être pour bientôt.

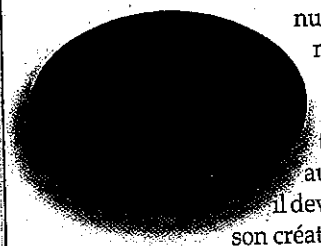
Sylvain Gelly • Rémi Munos

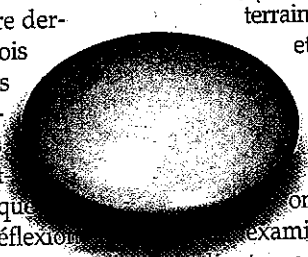


Le monde est un jeu de go, dont les règles ont été inutilement compliquées.
(Proverbe chinois)

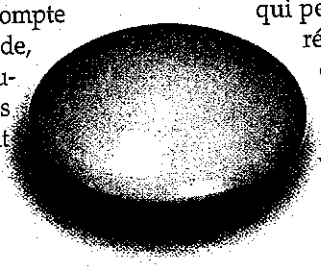


Le 11 mai 1997, Garry Kasparov, alors indiscutable numéro un mondial aux échecs, abandonnait la partie qui l'opposait au programme informatique *Deeper Blue* après à peine 19 coups. Il s'inclinait ainsi 3,5 à 2,5 points dans le match qui l'opposait au logiciel développé par la Société IBM. Cette défaite résonna comme celle de l'homme face à l'ordinateur dans ce qui symbolisait le dernier bastion de l'intelligence humaine. G. Kasparov eut beau prendre sa revanche peu après, les programmes d'échecs ont continué à progresser depuis. En décembre dernier, *Deep Fritz* a écrasé le numéro trois mondial Vladimir Kramnik 4 à 2 sans perdre une seule partie. La domination des programmes informatiques aux échecs est totale. L'ordinateur est-il devenu définitivement plus intelligent que son créateur? Pas encore. Il reste un jeu de réflexion où les joueurs humains dominent encore largement les meilleurs programmes informatiques: le jeu de go.





Certaines légendes chinoises font remonter l'origine du go à l'empereur Yao, il y a plus de 4000 ans, d'autres l'attribuent à deux dragons nommés Hei-Tzu et Bai-Tzu. La plus ancienne référence écrite au jeu de go date cependant de 548 avant notre ère. Aujourd'hui, on compte plus de 40 millions de joueurs à travers le monde, principalement en Asie, où ce jeu est très populaire: les compétitions les plus prestigieuses proposent des prix équivalents à ceux dotant le tournoi de tennis de Roland Garros!



Le go se joue sur une grille de 19 × 19 intersections nommée *goban*. Deux joueurs, les noirs et les blancs, posent chacun à leur tour un jeton, ou « pierre », sur une intersection libre. Deux pierres de même couleur situées sur des intersections adjacentes sont dites connectées, et les pierres connectées de proche en proche forment une « chaîne ». Les intersections libres adjacentes à une chaîne sont ses « libertés ». Lorsque toutes ses libertés sont occupées par des pierres adverses, la chaîne est capturée et enlevée du plateau. Le but du jeu est de gagner des territoires, c'est-à-dire des intersections libres délimitées par des pierres de même couleur. À la fin de la partie, le vainqueur est celui qui possède le plus de territoires *(voir l'encadré page 30)*.

Aujourd'hui, le go a pris la place des échecs comme terrain d'affrontement entre l'intelligence de l'homme et celle de la machine. Pour l'heure, l'avantage est à l'homme : les meilleurs programmes de go atteignent le niveau d'un amateur moyen. Cette situation pourrait toutefois ne pas durer. Les progrès des programmes informatiques ont été rapides ces dernières années. Après avoir examiné les méthodes classiques utilisées par les ordinateurs pour jouer au go et leurs limitations, nous verrons que l'approche proposée par notre équipe et par d'autres, fondée sur les notions de hasard et de probabilité, a permis ces progrès.

Le go, les échecs, les dames ou encore le morpion sont des jeux finis, c'est-à-dire qu'il existe une stratégie optimale qui permet de jouer au mieux en fonction des répliques jouées par l'adversaire. Il suffit donc en théorie d'explorer tous les coups possibles jusqu'à la fin de la partie et ensuite de suivre la séquence de coups qui mène à une partie victorieuse – si une telle séquence existe, ce

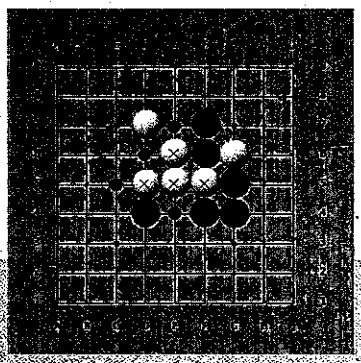
de go ?

qui n'est prouvé ni pour les échecs ni pour le go. Cependant, la richesse combinatoire du jeu de go est gigantesque : il existe 361 possibilités pour poser la première pierre, 360 pour la deuxième, etc. Le nombre total de positions possibles au go est ainsi de l'ordre de 10^{170} ! Par comparaison, il est de l'ordre de 10^{6} pour le morpion et « seulement » de 10^{50} aux échecs. L'exploration exhaustive des parties possibles est totalement hors de portée de la puissance de calcul des ordinateurs.

À partir d'une position donnée, les programmes de go – comme ceux d'échecs – doivent se contenter d'analyser les coups possibles jusqu'à une profondeur limitée, qui dépend des ressources de calcul disponible. Il est pratique de représenter les successions de coups par un type de graphe appelé arbre, dont la racine représente la position [illegible] feuilles sont les positions résultantes après les séquences de coups explorées (les branches), les nœuds de l'arbre étant les coups intermédiaires.

Un programme dispose de deux outils pour déterminer le meilleur coup possible. Le premier est une fonction dite heuristique, qui estime, de façon approchée, la valeur des positions de jeu atteintes au terme du déploiement de l'arbre (les feuilles). Une heuristique intègre en général des connaissances du jeu acquises par des experts humains. Aux échecs, par exemple, l'expérience a montré qu'une dame vaut environ huit pions, une tour, cinq, et le fou et le cavalier, trois. La valeur réelle des pièces dépend bien sûr de leur position sur l'échiquier, mais le simple bilan des

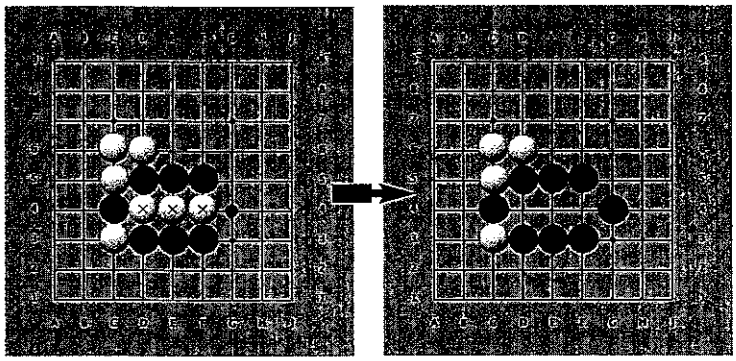
Les bases du jeu de go



Le go se joue à deux sur un plateau où est tracée une grille de 19 × 19 intersections, le *goban*, avec un nombre illimité de pions noirs et blancs, les pierres. Des goban de taille 9 × 9 et 13 × 13 sont aussi utilisés pour l'apprentissage *(ci-dessous)*. Les joueurs posent successivement et à tour de rôle une pierre de leur couleur sur les intersections libres. Le but du jeu est de constituer des territoires.

1

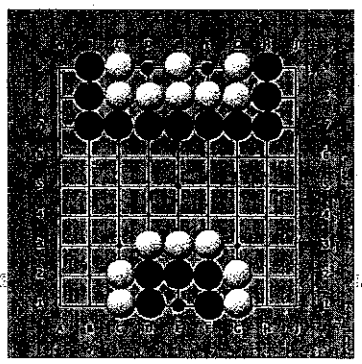
Chaînes et libertés. Les pierres situées sur deux intersections adjacentes sont « connectées » (les diagonales ne comptent pas). Un ensemble de pierres connectées de proche en proche forme une « chaîne ». Les intersections libres adjacentes à une chaîne (ou à une pierre isolée) sont ses « libertés ». Ci-dessus, les pierres blanches marquées d'une croix forment une chaîne qui a quatre libertés *(points rouges)*. Les pierres situées en D7 et en G6 ne sont pas connectées à cette chaîne.



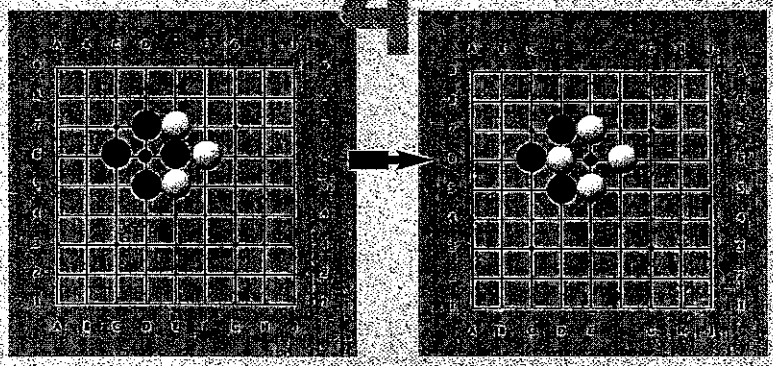
Capture. Lorsqu'un joueur occupe la dernière liberté d'une chaîne adverse, il capture celle-ci et retire du goban toutes les pierres correspondantes. Ci-dessus, la chaîne blanche marquée de croix n'a plus qu'une liberté *(point rouge, à gauche)*. Si Noir joue en G4, il supprime cette liberté et capture la chaîne *(à droite)*.

3

Chaînes vivantes et chaînes mortes. Un joueur ne peut pas poser une pierre qui supprime la dernière liberté d'une de ses propres chaînes, sauf si par ce coup il capture une chaîne adverse. Ci-dessous, Blanc peut jouer en E1, car il capture la chaîne noire. En revanche, Noir ne pourra jamais jouer en D9 ou en F9, car il ne capturerait pas la chaîne blanche. Cette chaîne blanche délimite un territoire comportant plusieurs intersections distinctes, des « yeux » ; elle est dite vivante *(en haut)*. La petite chaîne noire n'a qu'un seul œil, elle est dite morte *(en bas)*.



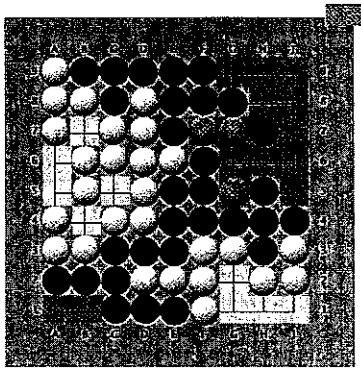
4

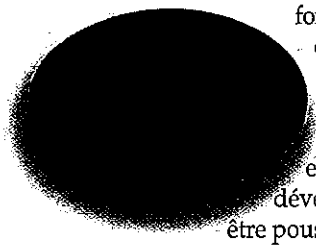


Ko. La règle de *Ko* (éternité en japonais) évite les parties infinies en interdisant de jouer un coup tel que la position après capture est exactement identique à la position précédente. Ci-dessus, si Blanc joue en D6, il capture la pierre noire en E6 *(à gauche)*. Noir ne peut rejouer immédiatement sur cette intersection et reprendre la pierre en D6 *(à droite)*. Le ko est fréquent dans une partie et, en obligeant à jouer ailleurs plutôt que de répondre à la menace, il permet qu'une position locale influence le jeu dans son ensemble.

5

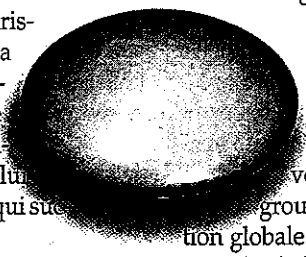
Fin de partie. Les noirs entament la partie. Celle-ci se termine lorsque les deux joueurs passent leur tour. Après s'être mis d'accord sur les chaînes mortes et les avoir enlevées du goban, les joueurs procèdent au décompte des territoires, c'est-à-dire de l'ensemble des intersections complètement entourées par des pierres d'une même couleur. Les règles usuelles comptent un point par intersection libre et un point par pierre adverse capturée. Pour compenser l'avantage que procure aux noirs le fait de commencer la partie, les blancs reçoivent un bonus variant entre 5,5 et 7,5 points, le *komi*. Dans la partie ci-contre, si Noir jouait en G6, il capturerait les pierres blanches marquées d'une croix, et si Blanc jouait lui-même en G6, Noir jouerait en H6 et les capturerait quand même : ces pierres sont mortes. Les territoires étant par ailleurs bien délimités, les deux joueurs décident de passer leur tour et la partie s'arrête. Les pierres mortes sont enlevées, puis on compte les territoires *(en grisé)*: 15 pour Noir, 9 pour Blanc. Noir ajoute 3 points pour les pierres capturées, et Blanc un komi de 5,5 points. Noir gagne par 18 à 14,5.





forces en présence fournit une première estimation raisonnable d'une position. L'évaluation fournie par une heuristique est la plupart du temps d'autant plus fiable que la position est proche de la fin de la partie. Le développement de l'arbre gagne donc à être poussé aussi loin qu'il est raisonnable.

Le second outil est un algorithme de recherche arborescente. Il calcule, à partir des valeurs des positions atteintes au terme de l'exploration estimées par l'heuristique, la valeur de la position en cours (la racine) et de tout ou partie des positions intermédiaires. En d'autres termes, l'algorithme remonte des feuilles vers la racine pour estimer cette dernière. Le meilleur coup est celui qui conduit au nœud de plus grande valeur qui succède immédiatement à la racine.



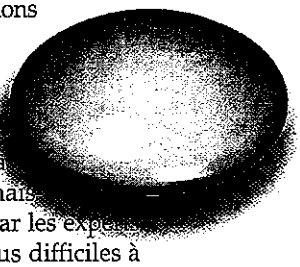
Explorer l'arbre des feuilles à la racine

L'algorithme de recherche arborescente le plus ancien et le plus répandu est l'algorithme min-max. Il consiste à remonter dans l'arbre en calculant de façon récursive à partir des feuilles la valeur des positions précédentes, et ainsi de suite jusqu'à la racine *(voir l'encadré page 32)*. L'algorithme min-max reproduit un raisonnement naturel. Lorsque c'est à lui de jouer, le programme cherche le coup qui va le conduire dans la meilleure position. Il est logique de supposer que son adversaire va en faire de même, c'est-à-dire chercher à atteindre la position la moins bonne pour l'ordinateur. Le programme calcule ainsi la valeur d'une position en prenant le maximum des valeurs des positions résultantes pour tous les coups possibles si c'est à lui de jouer, et le minimum si c'est au tour de l'adversaire. Par construction, l'algorithme min-max suppose que les deux adversaires jouent de façon rationnelle et optimale, et conduit toujours au meilleur coup possible pour une heuristique donnée. Cependant, pour choisir le meilleur coup dans une position au rang n, il faut connaître la valeur de toutes les positions au rang $n + 1$. Cette récursivité implique de développer toutes les branches de l'arbre jusqu'à la profondeur fixée, et d'évaluer toutes les feuilles à l'aide de l'heuristique. L'algorithme min-max est donc peu économique en termes de temps de calcul.

Introduit par le mathématicien John von Neumann en 1928, cet algorithme a connu depuis de nombreuses améliorations, qui, tout en restant fondées sur le principe de récursivité, permettent de réduire le nombre de calculs à mener. La plus connue, l'algorithme alpha-bêta, supprime les branches de l'arbre qui ne vont pas influer sur la valeur min-max de la racine en imposant deux bornes sur la valeur recherchée, l'une minimale et l'autre maximale. Cet élagage peut mener à une réduction impressionnante du nombre de feuilles évaluées et, partant, du nombre d'appels à la fonction heuristique. Dans les meilleurs cas, le gain en temps de calcul correspondant permet de développer l'arbre à une profondeur double.

Que ce soit pour jouer aux échecs ou au go, la stratégie des programmes informatiques classiques s'appuie donc sur le schéma suivant : construction de l'arbre des coups possibles jusqu'à une profondeur donnée, évaluation des positions terminales résultantes, et calcul de la valeur de la position actuelle par récursivité. Cette approche, aujourd'hui très efficace pour le jeu d'échecs, trouve néanmoins ses limites avec le jeu de go. Dans ce dernier cas, elle se révèle même moins efficace que les stratégies des joueurs humains.

La première difficulté concerne l'évaluation d'une position. Une fonction heuristique doit prendre en compte à la fois des attributs locaux (tactiques) et globaux (stratégiques) de la position. Aux échecs, ces deux niveaux sont relativement imbriqués, car les interactions d'un petit groupe de pièces influent directement sur la situation globale du jeu. En revanche, dans le jeu de go, le lien entre l'échelon tactique – estimation de la viabilité d'une chaîne, possibilités de capture à court terme, etc. – et le niveau stratégique – zones d'influence des pierres, interactions à long terme entre chaînes situées dans des régions éloignées du goban, etc. – est moins direct.



L'avantage matériel, c'est-à-dire la différence entre les nombres de pierres des deux joueurs présentes sur le goban, constitue par exemple un indicateur très médiocre de la valeur de la position. Par ailleurs, les connaissances tactiques et stratégiques élaborées par les expert du jeu de go sont moins nombreuses et plus difficiles à mettre en œuvre que dans le cas des échecs. En conséquence, les heuristiques développées pour le jeu de go sont encore très imprécises et gourmandes en temps de calcul.

Une explosion combinatoire

Le second écueil est la complexité combinatoire. Dans une position de go quelconque, il existe en moyenne de l'ordre de 200 coups possibles, contre une quarantaine tout au plus aux échecs. Le facteur de branchement de l'arbre des possibilités – le nombre de branches sortant d'un nœud – est ainsi plus grand, l'arbre, plus dense, et le nombre de feuilles, beaucoup plus élevé. Dans les deux cas, le nombre de feuilles de l'arbre croît exponentiellement avec sa profondeur, mais avec un exposant de 200 pour le go contre 40 pour les échecs. Quatre coups seulement peuvent ainsi mener à environ 1,5 milliard de positions différentes. Or, avec l'algorithme min-max, chaque position atteinte est évaluée. Même en réduisant le nombre d'appels à l'heuristique par des améliorations de type alpha-bêta, le temps de calcul devient vite colossal avec la profondeur d'analyse (le temps de calcul de la valeur des nœuds est négligeable par rapport au temps nécessaire à l'évaluation des feuilles).

En pratique, la complexité du go est telle que les méthodes usuelles de recherche arborescente ne permettent pas d'aller au-delà d'une profondeur de trois à quatre coups, contre plus d'une dizaine aux échecs. Or la profondeur d'analyse requise pour déterminer un coup au go est souvent très élevée. Alors qu'aux échecs, il est rare de devoir calculer

plus d'une dizaine de coups à l'avance pour explorer une combinaison précise, certaines situations du jeu de go doivent être explorées dans plusieurs directions pendant plus de 30 coups avant qu'une évolution ne se dessine clairement. Compte tenu de toutes ces difficultés propres au jeu de go, il n'est pas étonnant que les performances des programmes soient décevantes.

A contrario, il est relativement facile pour un joueur humain d'évaluer une situation sur un goban. Notre système visuel très performant, configuré pour identifier les formes et les régularités, est particulièrement adapté pour appréhender les motifs à grande échelle formés par les chaînes de pierres sur le goban, ainsi que pour en compléter les lacunes ou pour en prolonger le développement en pensée. Quelques pierres suffisent à notre cerveau pour imaginer le territoire qu'elles délimitent. Dans une moindre part, cette vision géométrique est aussi l'un des atouts des grands maîtres d'échecs, mais elle est alors moins naturelle. Ainsi, malgré des capacités de calcul formel limitées, les joueurs humains ont une capacité d'anticipation bien supérieure à celle des programmes informatiques.

Goemate, *Go ++*, *The many faces of Go* et *KCC Igo* sont aujourd'hui considérés comme les programmes les plus forts sur des goban de taille 19 × 19. Le logiciel libre *GnuGo* est d'un niveau un peu inférieur à celui de ces programmes commerciaux. Pour donner une idée du chemin à parcourir : en 1997, Janice Kim, joueuse professionnelle de premier dan, a battu le programme *Hand-Talk* (prédécesseur de *Goemate*) malgré un handicap de 25 pierres (c'est-à-dire 25 coups d'avance), et en 1998, Martin Müller, sixième dan amateur, a battu *The many faces of go* malgré un handicap de 29 pierres !

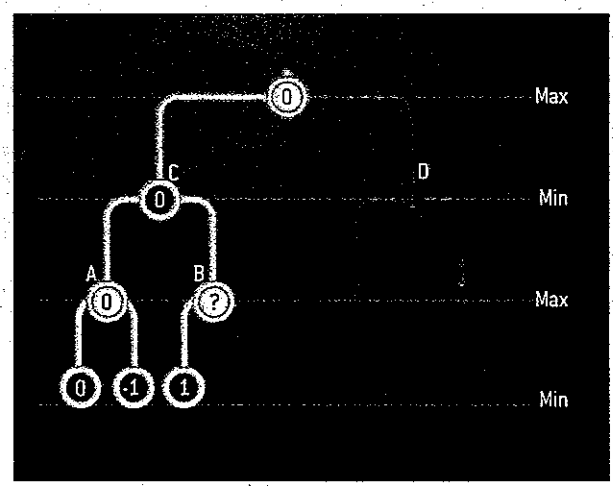
Se fier au hasard pour mieux jouer

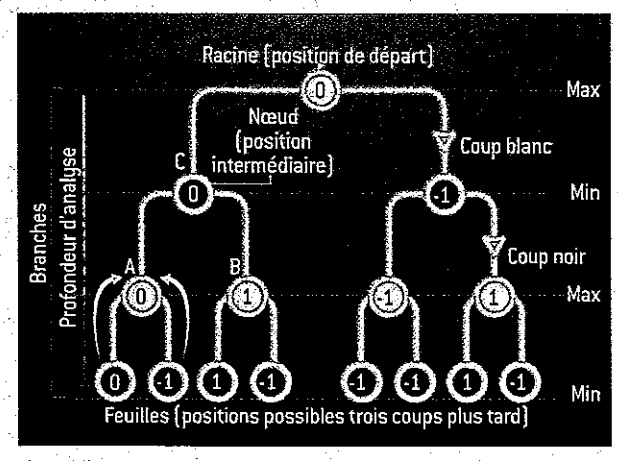
Pour autant, les programmes de go ont progressé ces dernières années, tant dans le domaine des heuristiques que dans celui de la recherche arborescente. La conception d'heuristiques a pris un tour nouveau en 1993, lorsque Bernd Bruegmann de l'Université de Californie à San Francisco eut l'idée d'utiliser des techniques aléatoires, les simulations Monte-Carlo, dans son programme *Gobble*. Le principe de cette technique est d'utiliser le hasard pour simuler plusieurs réalisations d'une quantité que l'on sou-

Pour choisir le meilleur coup dans une position donnée, l'ordinateur explore les coups successifs possibles jusqu'à une certaine profondeur. La valeur des différentes positions atteintes est estimée par une fonction dite heuristique. Ces informations sont représentées dans un arbre : la racine représente la position courante ; les feuilles, les positions atteintes au terme de l'exploration ; les branches,

Min-max. L'algorithme le plus général est nommé min-max. La valeur de la racine est estimée de façon récursive. À partir des feuilles, dont la valeur est connue, on remonte dans l'arbre en calculant la valeur d'une position à partir de celles qui lui succèdent. On suppose que les deux joueurs jouent de façon optimale : le programme attribue à une position le maximum parmi les valeurs des positions résultantes pour tous les coups possibles si c'est à lui de jouer (nœud max), et le minimum si c'est à l'adversaire de jouer (nœud min).

Ci-dessous, le nœud A est une position où l'ordinateur doit jouer. Sa valeur est donc égale au maximum de celle des feuilles qui le suivent, soit A = max (0, –1) = 0. De même, on trouve B = 1. Le nœud C vaut min (A, B) = min (0, 1) = 0. De proche en proche, on remonte ainsi à la racine, qui vaut 0. L'ordinateur choisit le coup qui le mène dans la meilleure position suivante, soit C. L'algorithme min-max conduit à un résultat optimal. Cependant, il explore l'arbre de façon exhaustive et évalue toutes les feuilles, ce qui est coûteux en temps de calcul.





Alpha-bêta. Une amélioration de l'algorithme min-max, nommée alpha-bêta, permet de simplifier l'exploration de l'arbre en supprimant les branches qui n'influent pas sur la valeur de la racine. Imaginons que les nœuds et les feuilles soient évalués au fur et à mesure que les branches sont développées, de gauche à droite. On a par exemple C = min (A, B) = min [0, max (1, *x*)], où *x* est la valeur de la quatrième feuille. Soit *x* est plus petit que 1, auquel cas B = 1, soit *x* est plus grand que 1, et alors B = *x* > 1, d'où C = 0, quelle que soit la valeur de *x*. Ce raisonnement permet de se dispenser de l'évaluation de la quatrième feuille. De même, on peut supprimer la branche de droite sous le nœud D. En pratique, l'algorithme alpha-bêta réalise ces suppressions grâce à la mise à jour en chaque nœud de deux bornes α et β encadrant la valeur recherchée. Cette réduction du nombre de feuilles à évaluer permet parfois de doubler la profondeur d'analyse.

haite évaluer, et prendre ensuite la moyenne de ces réalisations. Il s'applique à d'innombrables domaines scientifiques, telles la physique des particules ou l'évaluation numérique d'intégrales. Au go, évaluer une position *via* une simulation Monte-Carlo consiste, dans la version la plus simple, à terminer la partie en jouant des coups au hasard. Seuls les coups intrinsèquement mauvais, comme combler les degrés de liberté internes d'une de ses propres chaînes, sont proscrits. La partie terminée, on détermine le score, puis on répète l'opération un grand nombre de fois – de quelques dizaines à quelques centaines de milliers de fois selon les programmes. La moyenne des scores obtenus fournit une estimation de la valeur de la position de départ.

L'heuristique fondée sur les simulations Monte-Carlo présente de nombreux avantages par rapport à celles fondées sur les connaissances expertes. Ne nécessitant que des connaissances du jeu très sommaires – coups valides, capture et décompte des scores suffisent –, elle est facile à implémenter. Autre conséquence de cette simplicité, elle est peu gourmande en temps de calcul : sur un goban de taille 19 × 19, elle réalise jusqu'à 5000 évaluations globales par seconde, contre environ 200 pour une heuristique classique (même si celle-ci peut analyser des situations tactiques limitées à un rythme 1000 fois plus élevé). En outre, cette méthode prend en compte à la fois le caractère global et local d'une position. Seul point faible, l'heuristique Monte-Carlo est d'autant moins performante que la fin de la partie est éloignée. Chaque coup joué dans la simulation augmente ou diminue en effet aléatoirement la valeur de la position, de sorte que plus on enchaîne de coups, plus les probabilités de dévier par rapport à la valeur réelle de la position sont élevées.

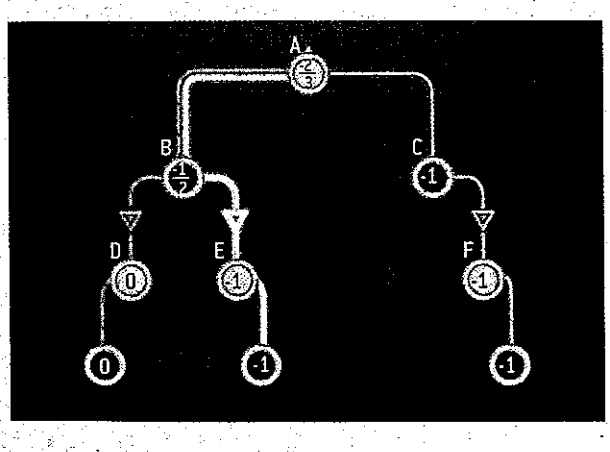
L'évaluation par simulation Monte-Carlo s'est révélée très efficace. Elle s'est largement imposée pour le jeu sur goban de taille 9 × 9. Le programme *CrazyStone* mis au point à la fin de 2005 par Rémi Coulom, de l'Université de Lille, qui utilise cette technique, a par exemple obtenu en 2006 la médaille d'or aux Olympiades informatiques dans cette catégorie. En taille 19 × 19, la méthode est moins performante, mais les progrès sont rapides.

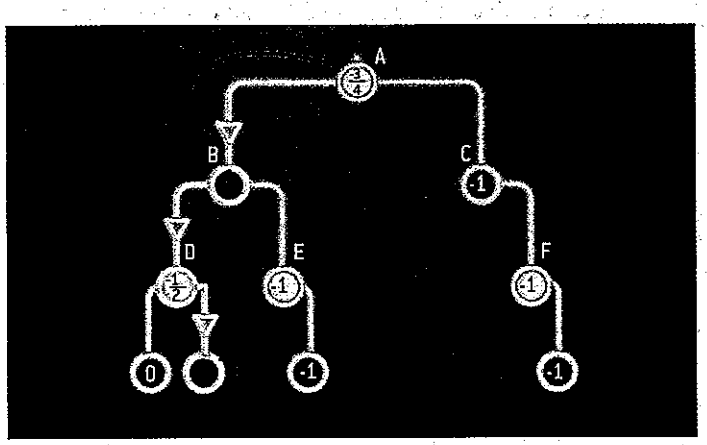
Il peut sembler étonnant qu'introduire du hasard dans un jeu totalement déterministe donne d'aussi bons résultats. C'est pourtant ce que nous faisons tous les jours lorsque nous devons résoudre des problèmes de nature déterministe dont

Les algorithmes de recherche arborescente

les successions de coups intermédiaires (ou nœuds). Il existe plusieurs méthodes pour calculer le meilleur coup connaissant la situation quelques coups plus tard. On détaille ici les trois principales. Par commodité, on a représenté un arbre de profondeur égale à trois coups, où seules deux options sont possibles à chaque coup et où c'est aux blancs de jouer.

UCT. En pratique, la complexité du go est telle que la profondeur d'analyse des algorithmes précédents est limitée à environ quatre coups. L'algorithme probabiliste UCT représente une alternative performante. L'arbre est développé et exploré de façon asymétrique en réalisant un compromis entre l'approfondissement des branches qui semblent meilleures compte tenu des feuilles déjà explorées, et l'exploration des branches encore peu empruntées. Des trajectoires sont lancées de la racine vers les feuilles. En chaque nœud rencontré, s'il reste des successeurs (ou nœuds fils) vierges, l'algorithme en choisit un au hasard. Si tous les nœuds fils ont été visités au moins une fois, l'algorithme choisit le nœud i qui maximise la somme V de deux termes. Le premier est la valeur moyenne x_i des feuilles précédemment atteintes depuis ce nœud (terme correspondant au nœud le plus rentable). Le second terme est un écart à la moyenne $\sqrt{[(\log n)/n_i]}$, où n_i est le nombre de trajectoires passées par ce nœud et n est le nombre total de trajectoires (ce terme favorise la branche la moins explorée). Une fois une feuille atteinte, sa valeur est intégrée dans la valeur moyenne des nœuds traversés.





Imaginons que trois trajectoires ont déjà été lancées *(ci-contre)*, affectant une valeur moyenne aux nœuds traversés (de A à F). Par exemple, les feuilles 0 et –1 ont été atteintes à partir du nœud B, de sorte que sa moyenne vaut –1/2. Détaillons la quatrième trajectoire *(ci-dessus, en jaune)*. À partir de la racine, les nœuds B et C ont déjà été explorés, deux et une fois respectivement. On évalue donc $V(B) = -1/2 + \sqrt{[(\log 4)/2]} \approx 0{,}048$ et $V(C) = -1 + \sqrt{[(\log 4)/1]} \approx -0{,}027$. V(B) > V(C), donc l'algorithme choisit le nœud B. Ses successeurs D et E ont aussi tous deux été parcourus une fois. Comme $V(D) = 0 + \sqrt{[(\log 4)/1]} > V(E) = -1 + \sqrt{[(\log 4)/1]}$, le nœud D est choisi. Ensuite, c'est la branche de droite sortant de D, encore vierge, qui est sélectionnée. La valeur de la feuille correspondante, soit –1, est incorporée en remontant de proche en proche dans la valeur moyenne des nœuds visités *(flèches rouges)* : $D = (0 - 1)/2 = -1/2$; $B = [2 \times (-1/2) - 1]/3 = -2/3$; enfin, la racine A vaut $[3 \times (-2/3) - 1]/4 = -3/4$.

L'algorithme UCT explore au début largement l'arbre en essayant uniformément tous les coups, puis, petit à petit, il privilégie les branches qui ont souvent mené à des parties gagnées. La valeur des nœuds ainsi évalués converge vers leur valeur min-max.

l'analyse est trop complexe. Pour réussir un service au tennis, par exemple, au lieu de calculer le geste parfait *via* la connaissance des lois physiques et des paramètres de la situation, nous procédons par tâtonnements en apprenant à rectifier nos gestes jusqu'à trouver une solution approchée.

Nos recherches actuelles autour du programme *MoGo*, à la suite des travaux de Bruno Bouzy, de l'Université Paris 5, visent à introduire une dose de connaissances expertes dans la méthode Monte-Carlo pour faire apparaître des séquences de coups réalistes dans les simulations aléatoires. Les résultats obtenus par *MoGo* depuis août dernier montrent que ce panachage améliore de façon considérable la justesse de l'évaluation d'une position.

Explorer ou exploiter ?

Le hasard est aussi derrière les progrès réalisés en matière d'algorithmes de recherche arborescente. En 2006, l'un d'entre nous (R. Munos) et Levente Kocsis et Csaba Szepesvári, de l'Université de Budapest, ont eu indépendamment l'idée d'un algorithme qui développe l'arbre de façon asymétrique, en privilégiant les branches qui paraissent les meilleures et en laissant provisoirement de côté les branches les moins prometteuses. Cette stratégie revient à réaliser un compromis entre l'exploration exhaustive de l'arbre et l'exploitation des informations déjà acquises. Un tel compromis apparaît dans de nombreuses disciplines mêlant apprentissage et optimisation.

Imaginons une machine à sous comme on en trouve dans les casinos, mais munie de plusieurs bras, dont l'action retourne des récompenses aléatoires. Sur quels bras tirer successivement pour optimiser le gain moyen ? Les lois de probabilité des gains de chaque bras étant initialement inconnues, à chaque tour, le joueur a le choix entre deux possibilités : exploiter l'information acquise lors des coups précédents en tirant le bras qui a fourni les gains les plus élevés jusque-là, ou explorer le jeu en choisissant un bras encore peu essayé, dont la probabilité de gain est mal connue, mais qui pourrait se révéler plus intéressant.

Parmi les nombreuses stratégies inventées pour résoudre ce dilemme, l'algorithme UCB (*Upper Confidence Bounds*, ou Bornes supérieures de l'intervalle de confiance) introduit en 2002 par Peter Auer, de l'Université de Graz, Nicolò Cesa-Bianchi, de l'Université de Milan, et Paul Fischer, de l'Université de Dortmund, donne de bons résultats. Il consiste à choisir à chaque tour, parmi tous les bras, celui qui maximise la somme d'un terme d'exploitation (la moyenne des gains obtenus jusqu'ici avec ce bras) et d'un terme d'exploration (d'autant plus grand que ce bras a été peu tiré par le passé). Maximiser le premier terme revient à privilégier le bras qui paraît le meilleur ; maximiser le second, en revanche, revient à préférer un bras encore mal connu. La combinaison de ces termes permet d'établir un compromis proche de l'optimum : lorsqu'on joue un grand nombre de fois, le pourcentage de chances de choisir un autre bras que le bras optimal tend vers zéro (comme $(\log n)/n$, où n est le nombre de parties jouées). Mieux, on sait qu'il n'existe pas d'algorithme qui tende plus rapidement vers ce comportement. Appliquer l'algorithme UCB revient à choisir le coup pour lequel, compte tenu des informations antérieures, il existe une possibilité qu'il soit le meilleur. C'est « l'optimiste dans l'incertain ».

Cet algorithme est optimal pour résoudre le compromis entre exploitation et exploration lorsque le gain survient immédiatement après la décision. Cependant, dans une partie de go, le gain associé à un choix n'est pas immédiat : seule la victoire ou la défaite en fin de partie permet de juger. Et encore : comment déterminer, dans une branche de l'arbre dont l'exploration conduit à une défaite, quel est le coup perdant qui aurait dû être évité ? Choisir entre l'exploration et l'exploitation d'une branche de l'arbre est difficile si l'on ne sait pas quelles actions ont conduit au résultat final. Comment dès lors adapter la stratégie d'optimisme dans l'incertain au jeu de go ?

R. Munos, L. Kocsis et C. Szepesvári ont eu l'idée d'utiliser l'algorithme UCB de façon récursive pour estimer la valeur des nœuds et de la racine dans l'arbre d'analyse d'une position. Le résultat est un algorithme nommé UCT (*Upper confidence bounds for trees*, soit Bornes supérieures de l'intervalle de confiance appliquées aux arbres). À chaque nœud de l'arbre est associé un algorithme de type UCB, qui choisit entre le coup qui paraît le meilleur au vu des explorations passées et un coup qui conduit à une branche encore peu explorée. Le choix d'une séquence de coups est ainsi décomposé en plusieurs problèmes plus simples en utilisant l'algorithme UCB de façon répétée.

Un développement asymétrique

L'algorithme UCT explore l'arbre en construisant de façon répétée des trajectoires (des suites de coups) de la racine vers les feuilles. En chaque nœud rencontré, l'algorithme vérifie d'abord s'il existe des nœuds non visités parmi les nœuds suivants (ou nœuds fils), auquel cas il en choisit un au hasard. Ce procédé est destiné à initialiser l'algorithme. Si tous les nœuds fils ont été visités au moins une fois, l'algorithme UTC fait appel à l'algorithme UCB pour choisir le nœud suivant. Plus précisément, le nœud choisi est celui qui maximise la somme de deux termes. Le premier terme est la valeur moyenne des feuilles précédemment atteintes à partir de ce nœud. Le second terme est d'autant plus petit que le nombre de trajectoires passées par ce nœud est grand. Lorsque la trajectoire atteint une feuille, celle-ci est évaluée grâce à une heuristique Monte-Carlo et le résultat est propagé à tous les nœuds traversés, et actualise leur valeur moyenne.

Le choix des coups tout au long d'une trajectoire résulte ainsi d'un compromis entre le coup qui paraît le meilleur au vu des trajectoires déjà explorées (maximisation du premier terme) et le coup qui conduit à explorer les

branches les moins visitées (maximisation du second terme). Concrètement, l'algorithme optimiste UCT explore au début largement l'arbre en essayant uniformément tous les coups possibles, puis, petit à petit, il privilégie l'exploration des branches qui ont souvent mené à des parties gagnées. L'arbre croît ainsi peu à peu de façon asymétrique tandis que des trajectoires successives explorent des branches nouvelles.

La stratégie ainsi appliquée est optimiste. Le terme d'exploration peut en effet être considéré comme la largeur d'un intervalle de confiance sur la valeur moyenne du nœud évalué. La somme des deux termes est la borne supérieure de cet intervalle, tandis que la différence définit sa borne inférieure. Le coup finalement retenu par l'algorithme UCT est celui qui maximise la borne supérieure de l'intervalle de confiance. Par analogie, apprendre à skier en suivant cette stratégie reviendrait à essayer des enchaînements de gestes en n'envisageant que les meilleures conséquences possibles et à apprendre par déceptions successives. Cet espoir indéfectible permettrait de trouver rapidement les gestes optimaux – à ceci près que les chutes étant bien réelles, et non de simples évaluations, la stratégie est quelque peu téméraire pour un skieur ! *A contrario*, une stratégie « pessimiste dans l'incertain », qui consisterait à maximiser la borne inférieure de l'intervalle de confiance, privilégierait les branches les plus explorées – notre skieur, par peur d'une chute, se contenterait de reproduire les gestes qu'il maîtrise déjà. Cette approche aboutirait à une stratégie robuste, où le risque de jouer un coup très mauvais est faible, mais qui ne convergerait pas vers le meilleur coup.

L'algorithme UCT pose quelques problèmes d'ordre théorique. Certaines hypothèses sous lesquelles l'algorithme UCB converge vers le choix optimal ne sont plus vérifiées. En particulier, l'indépendance des gains d'un coup au suivant n'est pas assurée, puisque les valeurs des parties issues d'un nœud donné dépendent des coups joués par la suite. L. Kocsis et C. Szepesvári ont néanmoins prouvé que les valeurs des nœuds calculées par l'algorithme UCT convergent vers la valeur optimale, c'est-à-dire la valeur min-max. R. Munos et C. Szepesvári ont par ailleurs établi que la vitesse de convergence de l'algorithme UCT dépend non pas de la profondeur réelle de l'arbre, mais de sa profondeur effective, c'est-à-dire du nombre de coups à partir duquel les estimations de l'heuristique Monte-Carlo rejoignent la valeur min-max exacte.

L'algorithme UCT présente de nombreux avantages par rapport à l'algorithme min-max. Au lieu d'explorer l'arbre de façon exhaustive, il répartit spontanément les ressources de calcul de façon adaptative, en passant plus de temps sur les coups les plus prometteurs en fonction de la profondeur effective locale de l'arbre. En d'autres termes, il passe plus de temps à examiner les positions tactiquement complexes, ce qui est très efficace dans le cas du jeu de go. Par ailleurs, la valeur des nœuds de l'arbre étant actualisée à chaque trajectoire, l'algorithme peut fournir une évaluation de la position à tout moment, qui sera affinée si du temps supplémentaire est accordé (si le temps alloué est suffisant, l'arbre finira par être entièrement exploré). L'adaptabilité a cependant un prix : lorsque l'arbre ne présente aucune régularité particulière – si le meilleur coup est « caché » dans une séquence qui semble de prime abord peu prometteuse –, l'algorithme UCT est certainement moins efficace qu'un algorithme alpha-bêta classique.

Homme et ordinateur : l'écart se resserre

Depuis les années 1980, des confrontations entre programmes de jeu de go sont organisées pour encourager les recherches dans ce domaine et évaluer les progrès accomplis. Les Olympiades informatiques qui se tiennent tous les quatre ans comportent ainsi un tournoi de go. Chaque année, au Japon, le *Gifu challenge* récompense le meilleur programme de jeu sur un goban 19 × 19. Sur Internet, des serveurs permettent de faire jouer les programmes et les joueurs humains ensemble sur différentes tailles de goban, et d'établir des classements. Chaque mois, sur le serveur KGS *(Kiseido go server)*, un tournoi de logiciels est organisé.

Pour la première fois, au regard des résultats obtenus ces deux dernières années par la nouvelle génération de logiciels utilisant la méthode Monte-Carlo et l'algorithme UCT, les humains ont du souci à se faire. Renforcé par une bibliothèque d'ouvertures, c'est-à-dire une base de données des coups classiques joués en début de partie, le programme *CrazyStone* a battu en août 2006 un joueur professionnel sur un goban 7 × 7. Les meilleurs programmes semblent désormais jouer de façon optimale dans cette taille. Sur un goban 9 × 9, les programmes *Valkyria* et *MoGo* ont battu plusieurs amateurs de bon niveau, un résultat inattendu il y a quelques années encore ; et très récemment, *MoGo* a battu pour la première fois des programmes utilisant des méthodes classiques sur un véritable goban 19 × 19.

En raison du niveau limité des programmes actuels, les joueurs de go professionnels se sentent à l'abri de toute menace pour encore de nombreuses années. Combien de temps ce *statu quo* va-t-il durer ? Les progrès sur des goban 9 × 9 ont été fulgurants ces dernières années. Le chemin est encore long, mais la marge de progression pour des techniques aussi récentes est élevée. Gageons que l'intérêt du go, jeu millénaire, n'en sera que grandi le jour où les machines découvriront, peut-être, une autre façon d'y jouer.

Sylvain GELLY est doctorant INRIA-CNRS au Laboratoire de recherche en informatique de l'Université de Paris-Sud, à Orsay. **Rémi MUNOS** est directeur de recherche à l'Institut national de recherche en informatique et automatique (INRIA), projet SEQUEL.

Auteurs & Bibliographie

S. GELLY, Y. WANG, R. MUNOS et O. TEYTAUD, *Modification of UCT with patterns in Monte-Carlo go*, Rapport de recherche INRIA RR-6062, 2006.

L. KOCSIS et C. SZEPESVÁRI, *Bandit based Monte-Carlo planning*, in *15th European Conference on Machine Learning*, pp. 282-293, 2006.

P. AUER, N. CESA-BIANCHI et P. FISCHER, *Finite-time analysis of the multiarmed bandit problem*, in *Machine Learning Journal*, vol. 47(2-3), pp. 235-256, 2002.

Serveur de go Kiseido : http://www.gokgs.com/

Logiciels de go : http://senseis.xmp.net/?GoPlayingPrograms