Des Optimisations de l'Alpha-Béta

Tristan Cazenave

Laboratoire d'Intelligence Artificielle
Département Informatique, Université Paris 8,
2 rue de la Liberté, 93526 Saint Denis, France.

cazenave@ai.univ-paris8.fr

Résumé. Nous passons en revue les principales optimisations apportées à l'algorithme Alpha-Béta. Nous présenterons tout d'abord le minimax, puis le négamax, ensuite l'Alpha-Béta lui même. Nous évoquerons ensuite les optimisations classiques comme l'approfondissement itératif, les tables de réfutations, la quiescence, l'ordonnancement, les coups nuls, les fenêtres nulles, les tables de transpositions. Nous en viendrons alors a décrire nos propres optimisations qui permettent des gains très conséquents: la recherche abstraite de preuves et l'élargissement itératif.

Mot clé: Alpha-Béta, recherche abstraite de preuve, élargissement itératif.

1 Introduction

Nous nous intéressons à l'optimisation de la recherche arborescente et plus particulièrement des arbres Min-Max. Les techniques développées pour cette sort d'arbre peuvent se révéler directement utiles, ou être à la source d'autres idées pour des systèmes qui développent d'autres formes d'arbres comme les démonstrateurs de théorèmes, les résolveurs de problèmes, ou les systèmes de satisfaction de contraintes..
L'algorithme Minimax et ses améliorations est utilisé dans la plupart des jeux à deux joueurs, à somme nulle et à information complète. Cet algorithme a été proposé il y a plus de cinquante ans par von Neumann et Morgenstern [Neumann et Morgenstern 1944] pour trouver un coup aux Echecs. Alan Turing [Turing 1953] a proposé des stratégies de recherche basée sur le principe du Minimax, et une amélioration importante fut l4alpha-Beta. Une forme affaiblie d'Alpha-Béta est apparue dans les premiers programmes d'Echecs comme NSS, de Newell, Shaw et Simon [Newell et al. 1958].
On se place maintenant dans le cadre des jeux à deux joueurs. L'algorithme MiniMax, ses variantes et améliorations sont utilisés dans la majorité des programmes de jeux à deux joueurs, à somme nulle et à information complète.

2 Améliorations usuelles du Minimax

On rappelle qu'une fonction d'évaluation prend en entrée une position dans un jeu et donne en sortie une évaluation numérique pour cette position. L'évaluation est d'autant plus élevée que la position est meilleure.
Si une fonction d'évaluation est parfaite, il est inutile d'essayer de prévoir plusieurs coups à l'avance. Toutefois, pour les jeux un peu complexes comme les Echecs, on ne connaît pas de fonction d'évaluation parfaite. On améliore donc un programme si à partir d'une bonne fonction d'évaluation on lui fait prévoir les conséquences de ses coups plusieurs coups à l'avance. L'hypothèse fondamentale du MiniMax est que l'adversaire utilise la même fonction d'évaluation que le programme.
Notre but est de trouver le coup qui **maximise** la fonction d'évaluation, alors que le but de l'adversaire est de choisir le coup qui **minimise** la fonction d'évaluation. Or les deux adversaires jouent chacun leur tour et en général c'est le joueur ami qui joue en premier puisqu'on cherche le meilleur coup à jouer pour le joueur ami. On va donc choisir les coups maximisant aux niveaux pairs et les coups minimisant aux niveaux impairs de l'arborescence, si on fait l'hypothèse que le premier niveau est le niveau 0.

Algorithme MiniMax, sachant qu'on appelle MiniMax(Position,0) pour trouver le meilleur coup :

```
#define PROFMAX 5 // Marche pour tous les niveaux
#define INIFNI MAXINT
#define odd(a) ((a)&1)
int MiniMax(char *Position,int profondeur)
   {
    int valeur,Best,i,N;
```

```
    char *PositionSuivante[100];

    if (profondeur==PROFMAX)
      return Evaluation(Position);

    N=TrouveCoupsPossibles(Position,PositionSuivante);

    Best=-INFINI;
    for (i=0; i<N; i++)
       {
               valeur=MiniMax(PositionSuivante[i],profondeur+1)
               if (odd(profondeur)) // niveaux impairs, on minimise
          {
           if (valeur<Best)
                     Best=valeur;
                  }
               else if (valeur>Best) // niveaux pairs, on maximise
                 Best=valeur;
              }
    return Best;
   }
```

2.1 Le NegaMax

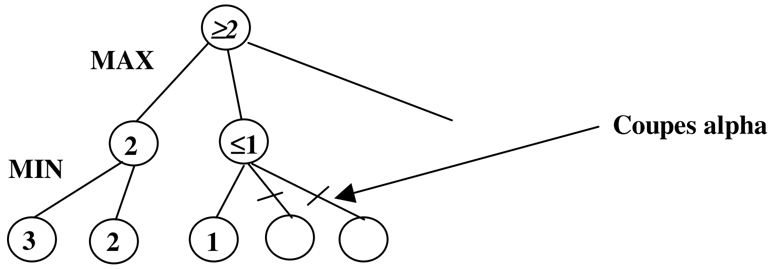
Plutôt que de tester si on est à un niveau pair ou impair pour savoir si on cherche à maximiser ou à minimiser l'évaluation, on peut inverser le signe des évaluations à chaque niveau, et toujours chercher à maximiser. On a alors l'algorithme NegaMax :

```
#define PROFMAX 4 // Ne marche que pour les niveaux pairs
#define INIFNI MAXINT
int NegaMax(char *Position,int profondeur) {
    int valeur,Best,i,N;
    char *PositionSuivante[100];

    if (profondeur==PROFMAX)
      return Evaluation(Position);

    N=TrouveCoupsPossibles(Position,PositionSuivante);

    Best=-INFINI;
    for (i=0; i<N; i++) {
               valeur=-NegaMax(PositionSuivante[i],profondeur+1)
               if (valeur>Best)
                 Best=valeur;
            }
    return Best;
}
```

2.2 L'algorithme Alpha Beta

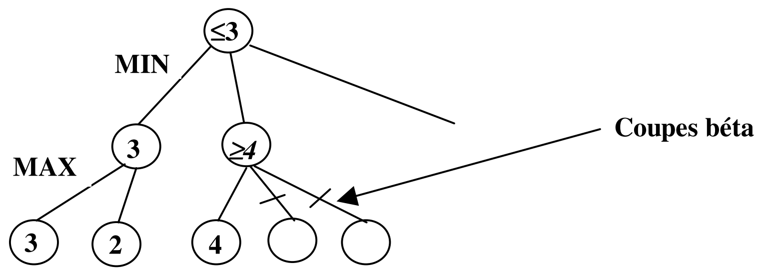
La coupure alpha se fait aux niveaux Min. Elle est basée sur l'observation que si la valeur d'un niveau Min est plus petite que la valeur du niveau Max supérieur, quelques soient les valeurs des nœuds suivants le niveau Min, ils ne changeront pas la valeur du niveau Max supérieur.

Exemple :



La coupure béta est la coupure symétrique de la coupure alpha pour les niveaux Max.

Exemple :



Propriété : lorsque le minimax trouve un coup en n nœuds, l'alpha béta trouve ce même coup en 1+2√(n) nœuds si les coups sont bien ordonnés [Knuth 75].L'ordre des coups est important car du bon ordre dépend le nombre de coupes.

Algorithme AlphaBeta, on appelle la fonction avec AlphaBeta (Position,0,-INFINI,INFINI) :

```
#define PROFMAX 4 // Ne marche que pour les niveaux pairs
#define INFINI MAXINT
#define odd(a) ((a)&1)
int AlphaBeta(char *Position,int profondeur,int alpha,int beta) {
    int valeur,Best,i,N;
    char *PositionSuivante[100];

    if (profondeur==PROFMAX)
      return Evaluation(Position);

    N=TrouveCoupsPossibles(Position,PositionSuivante);

    if (odd(profondeur)) Best=INFINI;
    else Best=-INFINI;
    for (i=0; i<N; i++) {
        valeur=AlphaBeta(PositionSuivante[i],profondeur+1,alpha,beta);
        if (odd(profondeur)) { // on minimise
    if (valeur<Best) {
       Best=valeur;
       if (Best<beta) {
          beta=Best;
          if (alpha>beta) return Best; // coupure alpha
       }
    }
    else if (valeur>Best) { // on maximise
      Best=valeur;
```

```
      if (Best>alpha) {
      alpha=Best;
       if (alpha>beta) return Best; // coupure beta
      }
    }
  }
  return Best;
}
```

2.3 Le NegaMax avec coupures Alpha Béta

On peut utiliser des coupures Alpha et Béta dans le negamax. On obtient alors l’algorithme suivant :

```
#define PROFMAX 4 // Ne marche que pour les niveaux pairs
#define INFINI MAXINT
int NegaAlphaBeta(char *Position,int profondeur,int alpha,int beta) {
    int valeur,Best,i,N;
    char *PositionSuivante[100];

    if (profondeur==PROFMAX)
      return Evaluation(Position);

    N=TrouveCoupsPossibles(Position,PositionSuivante);

    Best=-INFINI;
    for (i=0; i<N; i++) {
        valeur=-NegaAlphaBeta(PositionSuivante[i],profondeur+1,-beta,-
alpha);
        if (valeur>Best) {
          Best=valeur;
          if (Best>alpha) {
            alpha=Best;
            if (alpha>beta) return Best;
          }
        }
    }
    return Best;
}
```

2.4 L'effet d'horizon, approfondissement sélectif et quiescence

Il y a deux inconvénients au fait qu'on doive fixer une profondeur maximum. Le premier étant que le programme ne peut prévoir les effets d'un coup à une profondeur dépassant la profondeur maximum. Le second est que cela introduit des effets pervers dans les choix du programme : le programme fera toutes les menaces qu'il peut et qui sont pourtant inutiles pour repousser au-delà de son horizon un événement qui lui est défavorable. C'est ce qu'on appelle l'effet d'horizon : le programme repousse les événements défavorables au-delà de son horizon.

La solution utilisée dans Deep Blue comme parade à l'effet d'horizon est d'utiliser une méta-fonction d'évaluation qui n'évalue pas la position mais plutôt qui évalue le type de la position. Cette méta-fonction d'évaluation évalue si une position est stable (essentiellement, s'il reste des pièces en prise). Elle est utilisée par Deep Blue pour savoir si une position est évaluable ou si on doit continuer à la développer. Une position stable a une évaluation en laquelle on peut avoir confiance alors qu'une position instable est mal évaluée. Deep Blue utilise ce mécanisme de quiescence pour continuer de développer les positions basses de l'arbre jusqu'à des positions stables.

Les options de quiescence couramment utilisés dans les programmes d’Echecs sont les coups de capture, les coups de promotions (sauf quand le Roi est en Echec).

On peut aussi utiliser **l'élagage de futilité** lorsqu'on développe les positions instables : si la valeur de la position est inférieure à alpha-V alors seuls les captures et les échecs sont considérés.

Selective deepening

Si un coup semble intéressant, on continue à chercher à une plus grande profondeur que la profondeur habiuelle. Si un coup semble mauvais, on arrête de chercher à une profondeur plus petite que la profondeur habituelle.

Par exemple, si Chinook analyse un coup qui perd 3 pions, plutôt que de continuer à analyser la position jusqu'à une profondeur 10, il va réduire son analyse à seulement 5 coups à l'avance en faisant l'hypothèse qu'il y a de bonnes chances que le coup soit très mauvais. Par contre, si le programme joue un coup qui paraît très bon, il augmentera la profondeur de l'analyse de 10 à 12 coups à l'avance.

D'après J. Schaeffer, c'est une décision d'investissement : on investit son capital (le temps d'analyse) là ou on espère avoir le meilleur bénéfice (on cherche le plus d'information possible).

Un autre mécanisme de selective deepening est utilisé dans Deep Blue. Il analyse la structure de l'arborescence pour savoir si une branche de l'arbre développé est uniforme ou pas. Ainsi une branche dans laquelle beaucoup de coups sont bons est considérée comme ayant une évaluation sûre. Alors qu'un branche pour laquelle 1 seul coup parmi 30 est bon est considérée comme peu sûre. Deep Blue développe alors plus que les autres la partie de l'arborescence qui ne contient qu'un seul bon coup sur 30.

On peut considérer les évaluations utilisées pour activer le selective deepening comme des méta-fonctions d'évaluation.

Recherche de Quiescence

Un exemple de procédure de recherche de quiescence est donné ci-dessous. La fonction TrouveCoupsPossibles ne selectionne que les coups liés à la quiescence (captures et promotions). A chaque feuille de l'arboresence Alpha-Beta principale, on appelle la fonction quiescence pour évaluer la position.

```
int TableauCoups[PROFMAX][NB_COUPS_MAX];
char Position[TAILLE_POSITION];
int Quiescence(int profondeur,int alpha,int beta) {
    int valeur,Best,i,N;
    int *CoupsPossibles=TableauCoups[profondeur];

    Best=-Evaluation(-beta,-alpha);

    N=TrouveCoupsPossibles(Position,CoupsPossibles);

    for (i=1; i<CoupsPossibles[0]+1; i++) {
        JoueCoup(Position,CoupsPossibles[i]) ;
        valeur=-Quiescence(profondeur+1,-beta,-alpha);
        DejoueCoup() ;
        if (valeur>Best)
   Best=valeur;
}
    return Best;
   }
```

2.5 Le coup nul

Un coup nul (null move) consiste à changer le tour de jeu, ce qui est équivalent pour un joueur à passer son tour. Le coup nul est un coup légal au jeu de Go où un joueur peut passer, c’est même de cette façon que la fin de partie est décidée : lorsque les deux joueurs passent consécutivement. Aux Echecs par contre, ce n’est pas un coup légal, et il existe un petit nombre de positions pour lesquelles cela pourrait être utile. Ce sont les positions de Zugzwang, le premier joueur qui joue dans une position de ce type a perdu.

L'heuristique du coup nul permet de détecter des coups inutiles et de gagner du temps en ne développant pas des arborescences inutiles. On suppose que jouer le coup améliore la position (pas de Zugzwang). Toutefois cette heuristique peut parfois masquer une arborescence qui si elle avait été explorée aurait changer le résultat de l'Alpha-Béta. Contrairement aux coupes de l'Alpha-Béta qui ne changent pas le résultat du MiniMax, les coupes dues au coup nul peuvent changer le résultat de l'Alpha-Béta.

Le joueur joue deux coups de suite et effectue une recherche à un profondeur inférieure. Si le résultat de la recherche est inférieur à alpha, alors le coup n'est pas étudié plus profondément. On utilise un facteur de réduciton R pour choisir la profondeur de recherche du coup nul. Si on est dans un nœud pour lequel on s'apprête à faire une recherche de profondeur P, la profondeur de la recherche associée au coup nul sera P-R. On choisit en général R=2 ou R=3.

2.6 L'approfondissement itératif

L'approfondissement itératif commence par effectuer une recherche de profondeur 1, puis recommence avec une recherche complète à profondeur 2, et continue ainsi à faire des recherches à des profondeurs de plus en plus grandes jusqu'à ce qu'une solution soit trouvée.

Puisqu'une telle recherche n'engendre jamais un nœud tant que les nœuds de profondeurs plus petites n'ont pas été engendrés, elle trouve toujours la solution la plus courte. De plus si l'algorithme se termine à une profondeur d, sa complexité en espace est en O(d), c'est à dire linéaire en fonction de la profondeur de recherche.

A priori, l'approfondissement itératif perd beaucoup de temps dans les itérations précédant la solution. Toutefois, ce travail supplémentaire est généralement beaucoup plus petit que la dernière itération. S'il y a n coups possibles en moyenne pour chaque position, le nombre de positions à la profondeur d est n^d. Le nombre de nœuds à la profondeur d-1 est de n^{d-1} pour la d-1$^{\text{ème}}$ itération et chaque nœud est engendré deux fois, une fois pour l'itération finale, une fois pour l'avant dernière itération. Le nombre de nœuds engendré est donc de $n^d + 2n^{d-1} + 3n^{d-2} + 4n^{d-3} + \ldots + dn$. Ce qui est en $O(n^d)$. Si n est assez grand, le premier terme est nettement plus grand que les autres, c'est donc la dernière itération qui prend la plus grande partie du temps. Par rapport à la recherche en largeur d'abord, l'approfondissement itératif engendre n/(n-1) fois plus de nœuds. Si on considère l'asymptote, l'approfondissement itératif est un algorithme de recherche optimal aussi bien en temps qu'en espace.

Les algorithmes de recherche en profondeur d'abord ont un autre avantage important sur les algorithmes en meilleur d'abord ou en largeur d'abord : il permettent d'utiliser à moindre coût des informations incrémentales sur la position. L'ordre de recherche des positions permet de connaître les informations sur la position avant le coup qui a mené à cette position. On peut utiliser ces informations pour recalculer plus rapidement les propriétés d'une position, en ne calculant que la différence avec la position précédente induite par le coup.

2.7 Les tables de réfutation

Une heuristique qui fonctionne bien avec l'approfondissement itératif est de retenir à chaque itération la suite de coups optimaux calculés par l'Alpha-Béta en réponse à chaque coup à la racine. Chaque coup à la racine est donc associé à une séquence de coups qui le suivent et qui constitue sa réfutation. On a ainsi les séquences de coups qui sont jugée les meilleures pour les deux joueurs à une profondeur donnée. L'heuristique consiste à essayer les coups de la table de réfutation en premier dans l'arborescence. Ce sont ceux qui ont le plus de chances d'être les meilleurs puisqu'ils étaient les meilleurs à la profondeur précédente. Or on a vu qu'essayer les meilleurs coups en premier permet de maximiser le nombre de coups Alpha Béta, et donc de minimiser le temps de recherche.

2.8 Le coup qui tue et l'heuristique de l'historique

L'ordre dans lequel on considère les coups a une grande influence sur l'efficacité de l'algorithme Alpha Béta. Un coup qui marche beaucoup mieux que les autres coups du même niveau dans un sous-arbre a de bonnes chances de bien marcher dans un autre sous-arbre. On va donc essayer ce 'killing move' en priorité dans les sous-arbres suivants. Cette heuristique est utilisée dans de nombreux programmes de jeux, entre autres Deep Blue et

GoTools (résolution de problèmes de vie et de mort au jeu de Go). En général, les programmes d'Echecs utilisent cette heuristique en mémorisant le meilleur coup ou les deux meilleurs coups pour chaque profondeur de recherche.

Une généralisation des coups qui tuent est l'heuristique de l'historique (*history heuristic* [Schaeffer 83,89]) : Une note est mise à jour pour chaque coup légal rencontré dans l'arbre de recherche. A chaque fois qu'un coup est reconnu comme le meilleur dans une recherche, sa note est ajustée d'un montant proportionnel à la profondeur du sous-arbre exploré. On ajoutel $2^{ProfondeurMax-p}$ à la note du meilleur coup, p étant la profondeur à laquelle a été essayé le coup. On ordonne les coups à tester dans l'Alpha-Béta en fonction de leur note. J. Schaeffer a montré que l'heuristique de l'historique associée aux tables de transposition est responsable de 99% des réductions de recherche dans l'Alpha-Béta [Schaeffer 89] .

2.9 La recherche aspirante

Plutôt que d'appeler l'Alpha-Beta avec des valeurs initiales de –INFINI pour alpha et +INFINI pour beta, on peut lui permettre de couper plus de branches, si on augmente (resp. diminue) la valeur initiale de alpha (resp. beta). Si la valeur finale trouvée est comprise entre les alpha et beta initiaux, le résultat sera tout de même juste, bien que l'on ait coupé plus de branches inutiles qu'avec des valeurs infinies.

La recherche aspirante permet de régler les valeurs initiales pour alpha et beta en prenant en compte les résultats de la recherche précédente. Au début de chaque itération, les valeurs maximales (resp. minimales) sont initialisées avec les valeurs remontées de l'itération précédentes, additionnées (resp. diminuées) de la valeur d'un pion. Si la recherche échoue (les valeurs ne sont pas comprises dans la fenêtre), la fenêtre est ajustée à (-INFINI,valeur) si on échoue vers le bas, ou (valeur,+INFINI) si on échoue vers le haut.

2.10 La fenêtre minimale, La variation principale, La fenêtre nulle

En poussant cette idée jusqu'au bout, on obtient l'idée de la fenêtre nulle. Cela consiste à appeler l'alpha-béta avec une fenêtre (valeur,valeur+1), sachant que la fonction d'évaluation est entière avec des différences d'évaluation minimales de un point. Les arbres développés sont alors plus petits, et on peut ajuster vers le haut ou vers le bas la valeur en fonction de ce que retourne l'arbre. Il a été montré que cet algorithme, qui associé aux tables de transposition s'appelle MTD(f), développe les feuilles de l'arbre dans le même ordre qu'un algorithme en meilleur d'abord qui a été prouvé meilleur qu'alpha-béta: SSS*.

2.11 Les Tables de Transposition

Une table de transposition sert à stocker les positions déjà rencontrées dans une table de hachage. Lorsqu'on rencontre une position, on commence par vérifier si elle a déjà été vue. Pour cela, on a un problème de mémoire : coder la position sur 32 ou 64 bits. De plus, la rapidité de codage est importante. La méthode la plus fréquemment utilisée est le hachage de Zobrist : on associe un nombre aléatoire pour chaque position de pièce possible, et pour le joueur qui a la main.
Au Go cela revient a avoir : 2 nombres aléatoires par intersection + couleur qui joue + Ko. Le hachage de la position est le XOR de tous les nombres présent sur la position.
Au Echecs on a 12 nombres aléatoires par case (un par pièce différente) + 4 nombres pour les droits de roques + 8 nombres pour les captures en passant + 1 pour la couleur qui joue. Soit 64*12+4+8+1=781 nombres aléatoires.
Il y a deux avantage à utiliser le XOR pour coder une position :

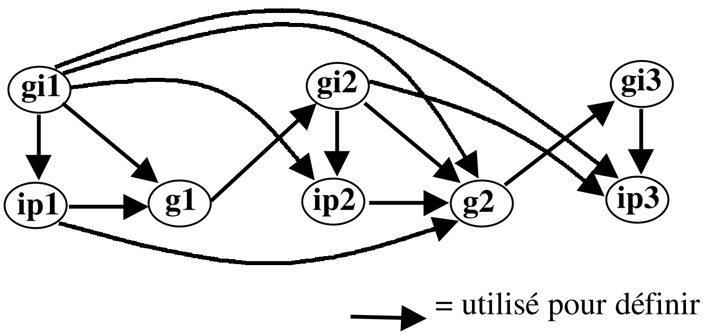
- Le XOR est une opération très rapide sur les bits.
- La valeur de hachage d'une position peut être calculée incrémentalement.

3 La Recherche Abstraite de Preuves

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

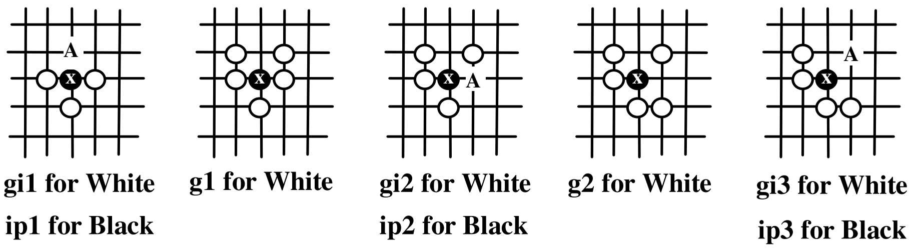
3.1 Les Ensembles de Coups Abstraits

Les coups qui permettent de changer l'issue d'une recherche peuvent être facilement trouvés lorsque le but est presque atteint. Toutefois, lorsque le but n'est plus à un ou deux coups, cela devient plus difficile. Nous traiterons dans cette section de la sélection de l'ensemble complet des coups possibles utiles à examiner lorsque le but ne peut pas être directement atteint. Par exemple, si une chaîne de pierres au jeu de Go peut être capturée en 5 coups, nous voulons trouver tous les coups abstraits qui sont susceptibles d'empêcher cette chaîne d'être capturée. Un coup abstrait est un coup qui est défini en utilisant des propriétés abstraites des chaînes ou du damier comme par exemple la liberté d'une chaîne.
Nous utiliserons des noms pour les différents états de jeux. Le nom des jeux est suivi par un nombre qui indique le nombre minimal de coup que l'attaquant doit jouer pour atteindre le but. Un jeu qui peut être gagné si l'attaquant joue est appelé 'gi'. Un jeu ou le défenseur doit jouer un coup pour ne pas être battu est appelé 'ip'. Un jeu gagné pour l'attaquant est appelé 'g. Un jeu est toujours associé à un joueur et dans le cas des jeux 'gi' et 'ip' à des coups de ce joueur.



Les dépendances entre jeux

Cette figure donne les dépendances entre les différents jeux. Un jeu peut être défini en utilisant les jeux d'indice inférieurs. Par exemple le jeu g1 pour Blanc est défini par : le jeu est ip1 pour Noir, et tous les coups ip1 amènent à des jeux gi1 pour Blanc.



Exemples de jeux

La figure ci dessus donne des exemples de jeux pour le jeu de la prise. Un exemple de la façon dont les coups abstraits pour un jeu sont trouvés est par exemple le passage des jeux gi aux jeux ip : le seul coup qui peut modifier une intersection vide au Go est de jouer dessus. Toutes les intersections vides utilisées pour définir un jeu gi, sont donc des coups possibles du jeu ip associé. Un jeu ip est associé à l'ensemble des coups qui peuvent prévenir un jeu gi.
Ces ensembles abstraits de coups sont trouvés automatiquement par Introspect, mais le code engendré est très volumineux. Il serait intéressant de développer des outils permettant de définir plus simplement (avec des programmes plus concis) ces ensembles.

3.3 Résultats Expérimentaux

Les test comparatifs de l'Alpha-Béta et de la recherche de preuve abstraite ont été fait sur des ensembles de problèmes standards [Kano 1985a,b,1987] sur un ordinateur muni d'un processeur K6-2 450 MHz. Prevent-ip3-1s-10000N correspond à l'Alpha-Béta utilisant toutes ses optimisations, stoppé au bout d'une seconde ou de 10000 nœuds. La fonction de sélection des coups est la même que celle de la recherche abstraite de preuve à l'exception près que les définitions de jeux ne sont pas utilisées pour sélectionner encore plus les coups. Preventip3-1s revient au même sans la barrière des 10000 nœuds. ip3-1s-10000N est l'algorithme de recherche abstraite de preuves.

Algorithm	Total time	Number of nodes	% of problems
Preventip3-1s-10000N	19.79	109117	99.12%
Preventip3-1s	19.79	109117	99.12%
ip3-1s-10000N	11.82	10340	99.12%

Table 1. Résultats pour ggv1

Algorithm	Total time	Number of nodes	% of problems
Preventip3-1s-10000N	113.20	836387	78.47%
Preventip3-1s	118.60	870938	77.78%
ip3-1s-10000N	34.13	42382	88.19%

Table 2. Résultats pour ggv2

Algorithm	Total time	Number of nodes	% of problems
Preventip3-1s-10000N	65.61	449987	65.28%
Preventip3-1s	74.25	483390	65.28%
ip3-1s-10000N	21.13	27283	73.61%

Table 3. Résultats pour ggv3

Des test ont aussi été effectués pour voir l'évolution de l'algorithme quand plus de temps lui était donné. Les gains sont alors encore plus grands par rapport à l'Alpha-Béta.

Algorithm	Total time	Number of nodes	% of problems
Preventip3-10s-100000N	635.20	4607171	79.17%
ip3-10s-100000N	63.57	81302	90.28%

Table 4. Résultats pour ggv2 avec plus de temps et de nœuds

Algorithm	Total time	Number of nodes	% of problems
Preventip3-10s-100000N	726.40	4319840	70.83%
ip3-10s-100000N	23.97	33936	73.61%

Table 5. Résultats pour ggv3 avec plus de temps et de nœuds

Des test ont aussi été effectués pour tester l'intérêt de l'heuristique du coup nul. Ces tests se révèlent concluants puisque le temps de recherche est inférieur. Toutefois l'utilisation de l'heuristique du coup nul enlève la propriété de preuve à l'algorithme. Les résultats peuvent alors être faux. En pratique, d'après les résultats expérimentaux, cela n'a toutefois pas l'air de diminuer ses performances.

Algorithm	Book	Total time	nodes	%
Preventip3-1s-10000N-NM	ggv1	13.34	69582	98.25%
Preventip3-1s-10000N-NM	ggv2	66.55	518398	77.08%
Preventip3-1s-10000N-NM	ggv3	30.50	230724	65.28%
ip3-1s-10000N-NM	ggv1	10.58	9401	99.12%
ip3-1s-10000N-NM	ggv2	31.57	39220	88.89%
ip3-1s-10000N-NM	ggv3	16.93	20902	73.61%

Table 6. Results with null move forward pruning

3.4 Conclusion sur la recherche abstraite de preuve

La recherche abstraite de preuve a deux grands avantage sur l'Alpha-Béta classique : ses résultats sont fiables, et il sont calculés plus rapidement. Avec les mêmes contraintes de temps, elle résout plus de problèmes, et l'algorithme réagit mieux que l'Alpha-Béta à l'augmentation des capacités CPU. Cet algorithme marche pour un grand nombre de jeux. Les seul inconvénient étant de définir une méthode pour trouver les ensembles de coups abstraits associés à chacun des jeux.

4 L'Elargissement Itératif

4.1 L'algorithme

La recherche sélective consiste à ne regarder qu'une partie des coups possibles. Un problème des algorithmes de type Alpha-Béta est l'ordonnancement des coups : on veut essayer en priorité les coups qui simplifie la situation et qui marchent souvent.
L'élargissement itératif consiste à ne considérer que les coups simples pour une première recherche, et s 'ils ne marchent pas envisager des coups moins standards. Pour cela, on définit des ensembles de coups abstraits : $S1 \subset S2 \subset \ldots \subset Sn$. L'élargissement itératif consiste à faire une recherche avec approfondissement itératif pour S1. Si elle échoue, recommencer avec S2, ainsi de suite jusqu'à Sn, ou jusqu'à ce que le temps imparti soit écoulé.
Au niveaux ET de l'arbre de preuve, il est naturel de définir les ensembles de coups abstraits à partir des définitions de jeux. Les coups S1 aux nœuds ET (appelés AND1) seront donc les coups ip1 et ip2, alors que les coups de S2 (appelés AND2) seront les coups ip1, ip2 et ip3. Au nœuds OU, S1 (OR1) sera l'ensemble des libertés de la chaîne à capturer et S2 (OR2) tous les coups intéressants y compris les libertés.

L'élargissement des coups peut se faire dans différents ordre, nous avons tetsté ceux-ci :

- OR2-2AND2-2: C'est l'algorithme original sans élargissement itératif. Les coups OR2 sont utilisés au nœuds OU et les coups AND2 aux nœuds ET.
- OR1-2AND2-2: L'algorithme commence avec une recherche utilisant les ensembles OR1 et AND2 , et si la recherche échoue, il en effectue une autre avec les ensembles OR2 et AND2.
- OR2-2AND1-2: L'algorithme commence avec une recherche utilisant les ensembles OR2 et AND1 , et si la recherche échoue, il en effectue une autre avec les ensembles OR2 et AND2.
- AND1-2OR1-2: OR1, AND1 => OR1, AND2 => OR2, AND2.
- OR1-2AND1-2: OR1, AND1 => OR2, AND1 => OR2, AND2.
- ORAND1-2: OR1, AND1 => OR2, AND2.
- OR1-2ANDOR1-2: OR1, AND1 => OR2, AND1 => OR1, AND2 => OR2, AND2.
- ORAND1-2AND1-2: OR1, AND1 => OR1, AND2 => OR12, AND1 => OR2, AND2.

4.2 Résultat Expérimentaux

Les expériences ont été effectuées sur un Pentium 266 MHz.

Algorithm	Time	Nodes	Problem
OR2-2AND2-2	18.15	4809	99.12%
OR1-2AND2-2	17.67	2667	99.12%
OR2-2AND1-2	12.81	4291	99.12%
AND1-2OR1-2	12.26	2576	99.12%
OR1-2AND1-2	12.38	3044	99.12%
ORAND1-2	12.11	2730	99.12%
OR1-2ANDOR1-2	12.31	2913	99.12%
ORAND1-2AND1-2	12.13	2587	99.12%

Table 7. Résultats pour ggv1

Algorithm	Time	Nodes	Problems
OR2-2AND2-2	62.96	30182	86.81%
OR1-2AND2-2	47.99	19096	86.11%
OR2-2AND1-2	32.62	28008	86.81%
AND1-2OR1-2	39.74	19721	86.11%
OR1-2AND1-2	37.15	24244	87.50%
ORAND1-2	39.57	19566	87.50%
OR1-2ANDOR1-2	35.99	23450	87.50%
ORAND1-2AND1-2	45.85	19544	85.42%

Table 8. Résultats pour ggv2

Algorithm	Time	Nodes	% Problems
OR2-2AND2-2	41.43	21226	78.67%
OR1-2AND2-2	30.03	15526	77.33%
OR2-2AND1-2	23.70	22647	81.33%
AND1-2OR1-2	23.78	15073	77.33%
OR1-2AND1-2	20.68	16281	81.33%
ORAND1-2	25.11	13206	78.67%
OR1-2ANDOR1-2	21.85	18106	80.00%
ORAND1-2AND1-2	32.74	13844	74.67%

Table 9. Résultats pour ggv3

On peut voir qu'un algorithme général et indépendant du jeu permet des gains substantiels: c'est le OR2-2AND1-2. Il ne dépend que des définitions des jeux ip pour l'élargissement, ce qui est une heuristique très générale. Le temps de recherche est pratiquement divisé par deux par rapport à l'algorithme sans élagissement.

5 Conclusion

L'optimisation de l'alpha-béta est encore un sujet de recherche active. Notamment en ce qui concerne la recherche sélective. Nous avons décrit les optimisations usuelles de l'alpha-béta ainsi que deux optimisations très efficaces pour les jeux a but simples et avec un grand facteur de branchement. Les évolutions futures de l'Alpha-Béta peuvent venir de plusieurs horizons : l'ajout (automatique ?) de connaissances, une sélectivité accrue, une plus grande rapidité pour les optimisations déjà existante, l'utilisations d'informations incertaines et probabilistes, ainsi que une possible liaison avec des algorithmes de recherche en meilleur d'abord. Enfin de nombreuse autres optimisations sont sans doute possibles, et plus particulièrement dans les jeux avec un grand nombre de coups possibles comme le Go ou le Phutball. De plus l'utilisation de techniques proches de celles de l'alpha-béta peuvent être à la source d'idées dans des domaines connexes comme la satisfaction de contraintes ainsi que l'a montré la réutilisation d'Iterative Broadening [Ginsberg 1992] par Meseguer et Walsh [Meseguer et Walsh 1998] pour la satisfaction de contraintes.

Bibliographie

Breuker T.: *Memory versus Search in Games.* PhD thesis, Maastricht. 1999.

Cazenave T.: *Metaprogramming Forced Moves*. Proceedings ECAI98 (ed. H. Prade), pp. 645-649. John Wiley & Sons Ltd., Chichester, England. ISBN 0-471-98431-0. 1998.

Cazenave T.: *Intelligence Artificielle: Une Approche Ludique*. Notes de cours. 1999.

Cazenave T.: *Generating Search Knowledge in a Class of Games.* submitted. http://www.ai.univ-paris8.fr/~cazenave/papers.html. 2000.

Cazenave T.: *Iterative Widening*. Workshop of the 2000 computer Olympiad, Londres. 2000.

Cazenave T.: *Abstract Proof Search.* Proceedings of the Computer and Games 2000 conference, publiée en LNCS, 2001.

Ginsberg M. L., Harvey W. D. : *Iterative Broadening.* Artificial Intelligence 55 (2-3), pp. 367-383. 1992.

Ginsberg M.: *Partition Search.* AAAI 97.

Kano Y.: *Graded Go Problems For Beginners. Volume One.* The Nihon Ki-in. ISBN 4-8182-0228-2 C2376. 1985.

Kano Y.: *Graded Go Problems For Beginners. Volume Two.* The Nihon Ki-in. ISBN 4-906574-47-5. 1985.

Kano Y.: *Graded Go Problems For Beginners. Volume Three.* The Nihon Ki-in. ISBN 4-8182-0230-4. 1987.

Marsland T. A., Björnsson Y.: *From Minimax to Manhattan.* Games in AI Research, pp. 5-17. Edited by H.J. van den Herik and H. Iida, Universiteit Maastricht. ISBN 90-621-6416-1. 2000.

Meseguer P., Walsh T. : *Interleaved and Discrepancy Based Search.* Proceedings ECAI98 (ed. H. Prade). John Wiley & Sons Ltd., Chichester, England. ISBN 0-471-98431-0. 1998.

Schaeffer J.: *Search Ideas in Chinook.* Games in AI Research. Edited by H.J. van den Herik and H. Iida, Universiteit Maastricht. ISBN 90-621-6416-1. 2000.

Thomsen T.: *Lambda-search in game trees – with application to go.* CG 2000. LNCS, 2001.