Introduction à la théorie des jeux à deux

Table des matières

1 Arbres d'évaluation

A) Minimax

L'algorithme Minimax permet de choisir un "meilleur" successeur dans un arbre où chaque niveau est successivement à maximiser et à minimiser. L'algorithme parcourt l'arbre en profondeur d'abord.

```
function MINIMAX(N) is
begin
   if N is a leaf then
        return the estimated score of this leaf
   else
        Let N1, N2, .., Nm be the successors of N;
        if N is a Min node then
           return min{MINIMAX(N1), .., MINIMAX(Nm)}
        else
           return max{MINIMAX(N1), .., MINIMAX(Nm)}
end MINIMAX;
```

Cet algorithme s'adapte aux jeux en considérant que la maximisation est le choix du meilleur coup possible et que la minimisation représente le choix de l'adversaire en supposant que celui-ci choisit aussi la meilleure position pour lui (d'après nous). C'est cette dernière supposition qui est très forte.

```
double ami (const Plateau& Jeu, int profondeur ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    // Attention : éviter la copie suivante
    Plateau::possibles Liste = Jeu.liste_coup();
    double eval = -INFINI;
    for( Plateau::possibles::const_iterator l = Liste.begin();
         l != Liste.end();
         ++l ) {
         Jeu.bouger( *l );  // je joue
         eval = MAX ( eval, ennemi ( Jeu, profondeur-1 ) ) ;
         Jeu.remettre ( *l ) ;
    }
    return eval;
}
```

```
double ennemi (const Plateau& Jeu, int profondeur ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    Plateau::possibles Liste = Jeu.liste_coup();
    double eval = +INFINI;
    for( Plateau::possibles::const_iterator l = Liste.begin();
         l != Liste.end();
         ++l ) {
         Jeu.bouger( *l ); // l'autre joue
         eval = MIN ( eval, ami ( Jeu, profondeur-1 ) ) ;
         Jeu.remettre ( *l ) ;
    }
    return eval;
}

int main(int argc, char ** argv) {
    ...
    ami ( Jeu, profondeur_max )
    ...
}
```

B) α-β

ALPHA-BETA est une méthode de "Branch and Cut" réduisant le nombre de noeuds explorés par la stratégie "Minimax" : pour chaque noeud, elle calcule la valeur de la position ainsi que deux valeurs, alpha et beta.

Alpha : une valeur **toujours plus petite** que la vraie évaluation. Au début du parcours Alpha vaut -INFINI pour un noeud quelconque et la valeur de l'évaluation pour une feuille. Ensuite, pour un noeud à maximiser, alpha vaut la plus grande valeur de ses successeurs et, pour un noeud à minimiser, alpha est celui du prédécesseur.

Beta : une valeur **toujours plus grande** que la vraie évaluation. Au début du parcours beta vaut INFINI pour un noeud quelconque et la valeur de l'évaluation pour une feuille. Ensuite, pour un noeud à maximiser, beta est celui du prédécesseur et, pour un noeud à minimiser, beta vaut la plus petite valeur de ses successeurs.

Ce qui est garanti :
- La valeur d'un noeud ne sera jamais plus petite que alpha et jamais plus grande que beta.

- alpha ne décroît jamais, beta ne croît jamais.
- Quand un noeud est visité en dernier, sa valeur est celle de alpha si ce noeud est à maximiser, et celle de beta sinon.

```
double ami_ab (const Plateau& Jeu, int profondeur, double A, double B ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    alpha = A; beta = B;
    Plateau::possibles Liste = Jeu.liste_coup();
    for( Plateau::possibles::const_iterator l = Liste.begin();
         l != Liste.end();
         ++l ) {
         Jeu.bouger( *l ); // je joue
         alpha = MAX( alpha, ennemi_ab ( Jeu, profondeur-1, alpha, beta ) );
         Jeu.remettre ( *l ) ;
         if (alpha >= beta) return beta;
    }
    return alpha;
}

double ennemi_ab (const Plateau& Jeu, int profondeur, double A, double B ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    alpha = A; beta = B;
    Plateau::possibles Liste = Jeu.liste_coup();
    for( Plateau::possibles::const_iterator l = Liste.begin();
         l != Liste.end();
         ++l ) {
         Jeu.bouger( *l ); // l'autre joue
         beta = MIN( beta, ami_ab ( Jeu, profondeur-1, alpha, beta ) );
         Jeu.remettre ( *l ) ;
         if (alpha >= beta) return alpha;
     }
    return beta;
}

int main(int argc, char ** argv) {
    ...
    ami ( Jeu, profondeur_max, -INFINI, INFINI )
    ...
}
```

2 Variantes et optimisations d'Alpha-Beta

En général, les améliorations de l'algorithme alpha–beta sont de trois types :

Trier : pour améliorer l'ordre d'examen des noeuds. Cela permet de faire beaucoup plus de coupes.

Réduire : plus la fenêtre de recherche $(\beta - \alpha)$ est petite, plus il y a de coupes.

Réutiliser : sauvegarde des résultats pour le cas où ils ré-apparaîtraient (parce qu'une position est accessible par différents mouvements, parce que l'on a relancé alpha–beta, avec des paramètres différents, etc.).

Dans la suite, le **facteur de branchement** désigne la taille moyenne de la liste des coups à chaque étape (environ 35 aux échecs, de 8 à 10 aux dames, etc.).

La **profondeur** d'un algorithme de recherche, est le nombre de demi-coups que celui-ci a exploré en avance (aux échecs les meilleurs programmes regardent actuellement 8/9 demi-coups, cela monte à 11/12 pour le jeu othello, entre 9 et 12 pour les dames internationales 10×10).

3 Méthodes de tri

A) Méthode de l'ordonnancement quantitatif

On explore en premier les situations dont le nombre de coups légaux issus est le plus faible.

B) Descente itérative

Avec un facteur de branchement proche de 8, le coût d'un alpha–beta de profondeur $d+1$ est environ de 3 à 4 fois plus élevé que le coût de l'algorithme de profondeur d. Ainsi, il ne coûte pas tellement plus cher d'exécuter l'algorithme à une profondeur d, puis à une profondeur $d+1$.

L'**alpha–beta itératif** est donc en au moins deux phases : une première passe permet d'ordonner les coups du meilleur au plus mauvais à une profondeur k, l'exécution à une profondeur $k+j$ produit alors plus de coupes que si elle avait été lancée directement.

C) Descente itérative et gestion du temps

Un autre avantage est pour une utilisation à temps limité : on effectue un premier alpha–beta à une profondeur faible k, puis à toutes les profondeurs suivantes, jusqu'à avoir dépassé la limite de temps (ou jusqu'à ce que l'estimation prédise un dépassement au prochain appel).

4 Tables

A) Tables de transposition

Il est possible d'atteindre une même position par plusieurs chemins différents. On stocke alors les positions, leur évaluation et la profondeur associée dans des tables. Retrouver un noeud déjà évalué dans les tables permet alors les optimisations suivantes :

- Si le noeud a déjà été évalué à une profondeur au moins aussi grande que celle désirée, sa valeur devient celle stockée.

- Si il a été évalué à une profondeur moins grande, cette valeur sert pour ordonner les noeuds du meilleur au moins bon.

Pour permettre une recherche rapide dans les tables, celle-ci sont en général implémentées par des tables de hachage (cf section C)). Il est en outre possible de stocker une table par profondeur évaluée.

B) Tables de refutation

Le problème des tables précédentes est leur taille importante.

Avec un shéma itératif, il est possible de stocker plutôt des tables de réfutation. C'est-à-dire que pour les mauvais coups, on garde en mémoire le chemin qui indique que c'est un mauvais coup, puis on en explore la suite à une profondeur plus grande. Si le coup reste mauvais, cela suffit pour le rejeter.

C) Utilisation des tables de hachage de la STL

```
struct Plateau {
   long long _pb, _pn, _rb, _rn;
```

```
   double eval();
};

struct Hasher {
#define _MUL 950706376UL
#define _MOD 2147483647UL
   size_t operator()( const Plateau& _p ) const {
       size_t tmp = (_p._pb * _MUL) % _MOD;
       tmp = ((tmp + _p._pn) * _MUL) % _MOD;
       tmp = ((tmp + _p._rb) * _MUL) % _MOD;
       return  ((tmp + _p._rn) * _MUL) % _MOD;
   }
};

typedef hash_map< Plateau, double, Hasher > Hash_t;

void Memory_ab(Plateau& Jeu, Hash_t& Table, int profondeur ) {
  ...
  Hash_t::const_iterator trouve = Table.find( Jeu );
  if (trouve != Table.end() )
      return (*trouve).second;
  else
      return Table[ Jeu ] = alpha_beta(Jeu, profondeur);
}
```

Des références sur les tables de hachage sur le net :
- http ://www.sgi.com/tech/stl/AssociativeContainer.html
- http ://www.sgi.com/tech/stl/HashedAssociativeContainer.html
- http ://www.sgi.com/tech/stl/hash_map.html

5 Fenêtres

A) PVS

L'algorithme PVS - Principal Variant Search - est un exemple typique d'heuristique. Il fait l'hypothèse d'une certaine "continuité" des valeurs des successeurs d'une position ; en effet, en général les successeurs d'une position ont des valeurs proches. On se sert donc de la valeur du premier successeur pour définir les bornes d'une fenêtre de recherche serrée, ce qui permettra beaucoup de coupures. Cette heuristique peut ne pas marcher : si la valeur d'un successeur tombe en dehors de la fenêtre prévue, il faut refaire le calcul.

En pratique : supposons que l'on ait à maximiser et que $\alpha = 3$, $\beta = +\infty$. On effectue le prochain appel, pour le coup à évaluer suivant, par ennemi_ab avec 3 et $3+1$. Si le résultat vaut 3 cela veut dire que la vraie valeur est 3 ou moins, elle ne sert donc pas. Si le résultat vaut 4, cela veut dire que la vraie valeur est 4 ou plus, il faut donc relancer ennemi_ab avec $\alpha = 4$ et le précédent β.

B) Limitation a priori de la fenêtre de recherche

Par exemple autour de la valeur de la profondeur précédente, ou bien encore autour de la valeur de la fonction d'évaluation appliquée à la situation à étudier, plutôt qu'autour de la valeur de la situation précédente (PVS).

C) Recherche par aspiration

L'alpha-beta commence avec les valeurs $-\infty$ et $+\infty$. Une connaissance du jeu (nombre de pions restants, etc.) permet de restreindre la taille de la fenêtre de départ.

Si la valeur tombe dans l'intervalle choisi, alors cet intervalle était correct. Sinon, il faut relancer l'alpha-beta.

6 MTD(f)

A) A. Plaat

La méthode MTD(f) [A. Plaat, http ://www.cs.vu.nl/~aske/mtdf.html] combine les idées précédentes : réduire la fenêtre au maximum (de taille 1), même si il faut relancer l'alpha-beta plusieurs fois, et utiliser des tables pour stocker les valeurs déjà calculées (si plusieurs alpha-beta ré-évaluent les mêmes positions).

```
double MTDf(Plateau& Jeu, int depth, double f) {
  double g = (f>MINFTY)?f:0;
  double lowerbound = MINFTY;
  double upperbound = INFTY;
  double beta; int count = 0;
  for(; lowerbound < upperbound ; ++count)  {
    if (count > 5) {
      g = Memory_ennemi_ab(Jeu, depth, lowerbound, upperbound);
      break;
    }
    if (g == lowerbound)
      beta = g+1;
```

```
    else
      beta = g;
    g = Memory_ennemi_ab(Jeu, depth, beta - 1, beta);
    if (g < beta)
      upperbound = g;
    else
      lowerbound = g;
  }
  return g;
}
```

B) Variante à deux pas

```
double MTDFdouble(Plateau& Jeu, int depth, double f) {
  double g = (f>MINFTY)?f:0;
  double lowerbound = MINFTY;
  double upperbound = INFTY;
  double beta; int count = 0;
  for(  double diffbound = DINFTY;
        diffbound > 1 ;
        ++count, diffbound = upperbound-lowerbound)  {

    if (count > 4) {
      g = Memory_ennemi_ab(Jeu, depth, lowerbound, upperbound);
      break;
    }
    if (g == lowerbound)
      beta = g+1;
    else if (g == upperbound)
      beta = g-1;
    else
      beta = g;
    g = Memory_ennemi_ab(Jeu, depth, beta - 1, beta + 1);
    if (g < beta)
      upperbound = g;
    else if (g > beta)
      lowerbound = g;
    else
      break;

  }
  return g;
```

```
}
```

7 Accélerer la victoire

Pour terminer la partie, il est possible de donner une évaluation différente aux victoires suivant le nombre de pions restants, le nombre de coups qu'il faut pour atteindre cette position, etc.

8 Heuristiques

A) Profondeur selective

Dans un cadre itératif, chaque coup peut être examiné avec une profondeur qui lui est propre, suivant différents critères : sa précédente évaluation était très rapide donc on incrémente la profondeur en peu plus, et vice-versa ; les coups avec une forte probabilité d'être bons sont examinés avec une profondeur faible tandis que les autres sont examinés avec une plus grande profondeur

Cette heuristique rend, en général, l'ordinateur plus humain !

B) La recherche focalisée

Il s'agit ici d'un shéma itératif dans lequel à partir des derniers niveaux, seules les meilleures possibilités (80/90 %) sont explorées, les autres sont abandonnées. Attention, il s'agit d'une heuristique à manipuler avec précaution.

C) Passe

Si une position semble suffisament bonne à une profondeur k, l'ordinateur peut voir ce qui se passerait si il ne jouait pas (il passe) à la profondeur $k + 1$. L'idée est qu'en général ce choix est mauvais. Ainsi, si le résultat obtenu reste bon, la position doit être vraiment bonne, dans le cas contraire, il faut faire une recherche normale.