Représentation de la Connaissance
Complément Pratique

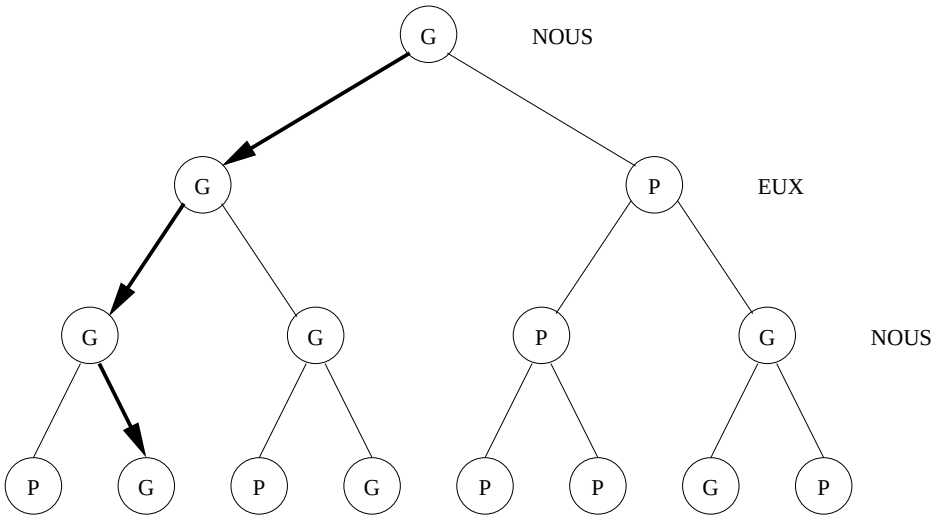
7 novembre 2006

Les jeux en Prolog

Introduction

Nous nous intéressons aux jeux à deux joueurs pour lesquels la situation du jeu est connue des deux joueurs à tout instant. Un exemple classique est le jeu d'échecs.

Prenons le cas d'un jeu ne pouvant avoir que deux issues : partie gagnée (G) ou perdue (P). On peut théoriquement représenter toutes les parties possibles de ce jeu par un *arbre de jeu* : chaque noeud correspond à l'état du jeu à un instant donné, les arcs correspondent à des déplacements valides, le noeud racine représente la position initiale du jeu et les feuilles toutes les positions finales possibles. Les joueurs jouent tour à tour, la profondeur de l'arbre détermine donc qui doit jouer ("nous" ou "eux").



Un tel arbre de jeu est un arbre AND/OR :

– Si c'est à nous de jouer, il suffit qu'*un* successeur soit gagnant pour gagner la partie (noeud OR).

– Si c'est à l'adversaire de jouer, il faut que *tous* les successeurs soient gagnants pour qu'on soit sûr de gagner dans tous les cas (noeud AND).

Un programme simple pour déterminer le "meilleur coup" :

```
won( Pos) :-
  terminalwon( Pos).

won( Pos) :-
  not terminallost( Pos),
  move( Pos, Pos1),
  not ( move( Pos1, Pos2),
        not won( Pos2)).
```

Il va de soit que ce code est beaucoup trop naïf en pratique. Pour le jeu d'échecs par exemple, le nombre de positions terminales est de l'ordre de 10^{120} et il est donc impossible d'obtenir le meilleur coup absolu du jeu en construisant l'arbre complet. De plus, ce code ne teste pas les cycles et va donc tourner en rond très rapidement (vu qu'il recherche en profondeur d'abord). Il est donc nécessaire de procéder de manière moins brutale.

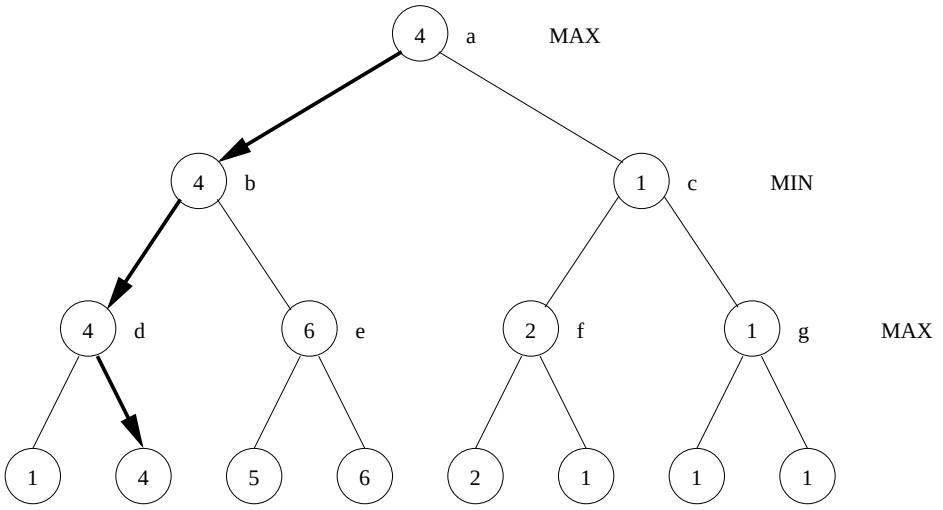
L'algorithme minimax

Dans les années 50, Shannon et Turing ont introduit un algorithme dont le principe de base est de limiter l'arbre de jeu à une certaine profondeur. Dans ce cas, les feuilles de l'arbre ne sont plus nécessairement des positions terminales.

On utilise une fonction d'évaluation $f(n)$ qui prend comme argument un noeud de l'arbre et retourne une valeur entière. Cette valeur représente une *estimation* de l'état de la partie. Une valeur élevée indique que le joueur "MAX" a plus de chances de gagner, dans le cas contraire c'est le joueur "MIN" qui a l'avantage.

Etant donné que les joueurs veulent sans cesse améliorer leur position, le joueur "MAX" va à tout moment vouloir obtenir des positions qui maximisent la valeur de la fonction d'évaluation et le joueur "MIN" des positions qui la minimise. D'où le nom de l'algorithme : minimax.

Illustrons la détermination du meilleur coup en partant d'un arbre simple limité à une profondeur de trois coups dans une position où c'est à "MAX" de jouer.



Le calcul de la fonction d'évaluation ne se fait qu'aux feuilles de l'arbre. Ces valeurs sont ensuite propagées vers le haut jusqu'à la racine. Les fils du noeud d par exemple ont respectivement comme étiquettes 1 et 4 et ce sera la valeur 4 qui sera attribuée à d vu qu'on est à une hauteur paire par rapport à la racine et qu'il s'agit donc du tour de "MAX". Pour la même raison, les noeuds e, f et g seront respectivement étiquetés par 6, 2 et 1.

Les noeuds b et c par contre seront respectivement étiquetés par les minimums des étiquettes de leurs fils, à savoir 4 et 1. Finalement la racine est étiquetée par 4 ce qui indique que "MAX" à l'avantage (en supposant que la valeur 0 représente une position égale) et que son meilleur coup consiste à rejoindre le noeud b.

Remarques :

→ Une bonne fonction d'évaluation de complexité acceptable n'est pas toujours facile à définir.
→ La fonction d'évaluation n'est appliquée qu'aux feuilles de l'arbre.

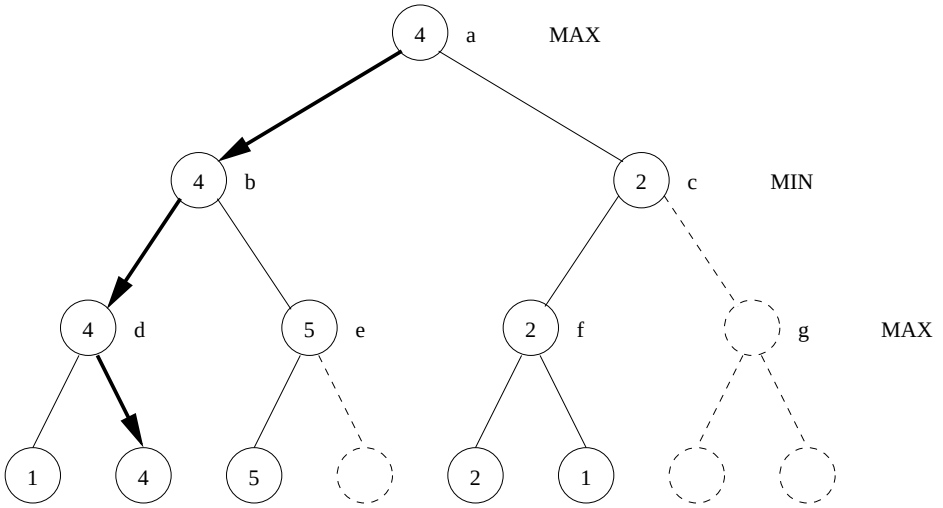
Une implémentation en Prolog

```
% minimax( Pos, BestSucc, Val):
%    Pos is a position, Val is its minimax value;
%    best move from Pos leads to position BestSucc

minimax( Pos, BestSucc, Val)  :-
  moves( Pos, PosList), !, % Legal moves in Pos produce
                           % PosList
  best( PosList, BestSucc, Val)
   ;
   staticval( Pos, Val).   % Pos has no successors:
                           % evaluate statically

best( [ Pos], Pos, Val)  :-
  minimax( Pos, _, Val), !.

best( [Pos1 | PosList], BestPos, BestVal)  :-
  minimax( Pos1, _, Val1),
  best( PosList, Pos2, Val2),
  betterof( Pos1, Val1, Pos2, Val2, BestPos, BestVal).

% Pos0 better than Pos1:
betterof( Pos0, Val0, Pos1, Val1, Pos0, Val0)  :-
  min_to_move( Pos0),   % MIN to move in Pos0
  Val0 > Val1, !        % MAX prefers the greater value
  ;
  max_to_move( Pos0),   % MAX to move in Pos0
  Val0 < Val1, !.       % MIN prefers the lesser value

% Otherwise Pos1 better than Pos0
betterof( Pos0, Val0, Pos1, Val1, Pos1, Val1).
```

L'algorithme alpha-beta

La méthode alpha-beta de base a été introduite dans les années 60 et permet d'obtenir exactement le même résultat que la méthode minimax mais sans construire complètement l'arbre. La méthode fait intervenir deux nouvelles notions :

Alpha : valeur minimale (garantie)
Beta : valeur maximale (qu'on ne dépassera jamais)



Lorsqu'on évalue le noeud e, nous savons que le noeud b aura au maximum la valeur 4 ($Beta = 4$). Le premier successeur de e a la valeur 5. C'est à MAX de jouer, on choisira donc le successeur de valeure maximale, ce qui veut dire que e aura au minimum la valeur 5. La valeur de e n'influencera donc pas celle de b puisqu'elle sera d'office plus grande que d et nous pouvons ignorer les successeurs restants. On appelle cela une coupure *Beta*.

Par analogie, on constate facilement que le noeud c aura une valeur au maximum égale à 2. Comme le noeud b a une valeur de 4 il n'est plus nécessaire de déterminer la vrai valeur du noeud c puisqu'on sait déjà qu'elle sera plus petite ou égale à 2 et donc à b. On appelle cela une coupure *Alpha*.

Dans les années 70, on a même découvert que les coupures ne devaient pas se limiter à éliminer des noeuds du niveau directement inférieur au niveau courant mais pouvaient aussi éliminer en profondeur des noeuds de niveau impair par rapport au niveau courant.

L'efficacité de l'algorithme dépend de l'ordre dans lequel les positions sont évaluées. Dans le meilleur des cas, le facteur de branchement n de l'arbre de recherche est réduit à $\sqrt{n}$.

Une implémentation en Prolog

```
alphabeta( Pos, Alpha, Beta, GoodPos, Val)  :-
  moves( Pos, PosList), !,
  boundedbest( PosList, Alpha, Beta, GoodPos, Val);
  staticval( Pos, Val).  % Static value of Pos

boundedbest( [Pos | PosList], Alpha, Beta,
             GoodPos, GoodVal)  :-
  alphabeta( Pos, Alpha, Beta, _, Val),
  goodenough( PosList, Alpha, Beta, Pos, Val,
              GoodPos, GoodVal).

% No other candidate
goodenough( [], _, _, Pos, Val, Pos, Val)  :-  !.

goodenough( _, Alpha, Beta, Pos, Val, Pos, Val)  :-
  min_to_move( Pos), Val>Beta, ! % Maximizer attained
                                 % upper bound
  ;
  max_to_move( Pos), Val<Alpha, !.% Minimizer attained
                                  % lower bound

goodenough( PosList, Alpha, Beta, Pos, Val,
            GoodPos, GoodVal)  :-
  newbounds( Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest( PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof( Pos, Val, Pos1, Val1, GoodPos, GoodVal).

% Maximizer increased lower bound
newbounds( Alpha, Beta, Pos, Val, Val, Beta)  :-
  min_to_move( Pos), Val > Alpha, !.

% Minimizer decreased upper bound
newbounds( Alpha, Beta, Pos, Val, Alpha, Val)  :-
   max_to_move( Pos), Val < Beta, !.

% Otherwise bounds unchanged
newbounds( Alpha, Beta, _, _, Alpha, Beta).
```

```
% Pos better than Pos1
betterof( Pos, Val, Pos1, Val1, Pos, Val)  :-
  min_to_move( Pos), Val > Val1, !
  ;
  max_to_move( Pos), Val < Val1, !.

% Otherwise Pos1 better
betterof( _, _, Pos1, Val1, Pos1, Val1).
```

Améliorations :

1. Incrémenter progressivement la profondeur de l'arbre de recherche (*progressive deepening*) :
   - Permet de contrôler le temps
   - Les résultats de l'itération précédente permettent d'ordonner les positions pour l'itération suivante. On aide ainsi l'algorithme alpha-beta en commençant la recherche par les déplacements plus prometteurs.

2. La fonction d'évaluation peut retourner une valeur non représentative de l'état du jeu pour certaines positions.
   Solution : prolonger le développement de l'arbre de recherche pour ces positions.