Rapport de projet

Programmation en Prolog

Par M. Es-Skali & Bertin

Introduction

Le but de ce projet est de modeliser le fameux jeu du TicTacToe et de creer un programme permettant de jouer de la facon la plus efficace possible. Aussi, nous devrons tout d' abord implementer l'algorithme connu sous le nom de Minimax.; nous l' ecrirons dans sa version la plus generale, et non pas dans une version specifique adaptee au TicTacToe. Ensuite, on tentera de l' optimiser , l' algorithme resultant s' appellera le Minimax_Alpha_Beta. Une fois ce travail realise, nous essaierons de les adapter au TicTacToe afin de determiner a chaque tour de jeu les meilleurs coups possibles .

Le Minimax

L'algorithme du Minimax.

Considerons un jeu a deux personnes dans lequel les deux joueurs ont des objectifs contradictoires. Il s'agit tout d' abord d' orienter la partie; nous distinguerons donc le joueur max 1 et le joueur min 2.
L' objectif fondemmental du minimax est de determiner a chaque tour de jeu, quel que soit le joueur, le meilleur coup possible. En fait , il s'agit de considerer tous les coups possibles et de simuler la partie recursivement en fonction de chacun de ces coups . Cette simulation est modelisable sous la forme d' un arbre. C'est pourquoi , l'algorithme du minimax est un algorithme base sur un parcours d' arbre et qui en fonction des valeurs de l' heuristique au niveau des feuilles permet de privilegier l' un des fils de la racine.

Il est developpe dans le fichier **Minimax1.pl.**

Notre version generale du minimax se distingue en particulier de sa version adaptee au tictactoe du fait qu' il ne construise pas l' arbre de lui-meme .Aussi , l'arbre est prealablemement construit dans le fichier **Exemple.pl.** Il est implemente de sorte que pour chaque noeud la liste des fils soit connue et qu'a chacune des feuilles soit associee la valeur de l' heuristique.
De plus notre version du minimax fait completement abstraction de la notion de profondeur ; c' est a dire que l' arbre est parcouru completement.

Cet algorithme repose donc sur un parcours recursif de l' arbre.
Appeler le *minimax* sur un noeud interne consiste en realite a appeler *minimax_liste* sur la liste des fils. Si la liste des fils est reduite a un element alors on rappelle minimax sur cet element sinon on applique minimax a la tete et minimax_liste a la queue. Il s'agit egalement d'indiquer a chaque iteration quel noeud et quelle valeur seront conserves d'ou le predicat *choix*.
Plus l' heuristique est elevee , plus la situation est favorable a max et plus elle faible , plus la situation est favorable a min. A chaque iteration , le predicat choix va ,en fonction du joueur a considerer et des valeurs de l' heuristique, determiner le coup optimal.

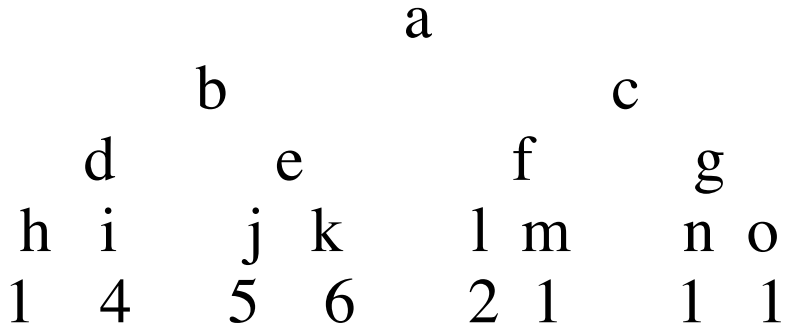
L'algorithme du Minimax modifie : le Minimax Alpha Beta.

L' algorithme precedent peut etre ameliore , dans le sens que l' on peut calculer le minimax d' un arbre de jeu en consultant moins de noeuds qu' avec l' algorithme precedent. L' idee est de transmettre recursivement une borne superieure et une borne inferieure que l' on nommera respectivement Alpha et Beta. Il s' agit en fait d' etablir, a chaque iteration, en fonction des valeurs de ces bornes et de la valeur resultant du *minimax_alpha_beta* applique au noeud , si oui ou non c' est la peine de parcourir les arborescences issues des freres. Cette evaluation, ainsi que l' eventuel CUT resultant , sont realises par le predicat *condition*.

Cet algorithme est developpe dans le fichier **Minimax2.pl.**

Exemple d'un arbre arbitraire et jeu de tests.

Nous allons maintenant donner l'arbre etudie dans le fichier **exemple.pl.** Ceci est un arbre binaire. On donne en bas, la valeur de l'heuristique pour chaque feuille de cet arbre:



Nous donnons en meme temps la syntaxe d'execution des deux programmes concus :
- minimax : minimax(Pos_initiale, Position, Valeur)

ce qui revient ici a lancer minimax(a, Position, Valeur)
resultat : Position = b ; Valeur = 4.

cela montre que le meilleur coup a jouer est d'aller vers la position b.

- minimax_alpha_beta : minimax_alpha_beta(Pos_initiale, Alpha, Beta, Position, Valeur)

on lance donc minimax_alpha_beta(a, 0, 50, Position, Valeur)
en fait, on a choisi 0 et 50 comme borne Sup et borne Inf des valeurs possibles de l'heuristique des feuilles.
on obtient exactement le meme resultat qu'auparavant, sauf que cette fois, l'algorithme n'a pas parcouru tout
l'arbre.

Et voici la preuve de ce qu'on avance, on donne ci dessus des extraits de la trace d'execution du programme :

```
6?- minimax_alpha_beta(a,0,50,P,V).
CALL   minimax_alpha_beta(a, 0, 50, P, V) (dbg)?-
.........
CALL   minimax_alpha_beta(b, 0, 50, _, Val) (dbg)?-
.........
CALL   minimax_alpha_beta(d, 0, 50, _, Val) (dbg)?-
.........
CALL   minimax_alpha_beta(h, 0, 50, _, Val) (dbg)?-
.........
CALL   minimax_alpha_beta(i, 1, 50, _, Val) (dbg)?-
.........
CALL   minimax_alpha_beta(e, 0, 4, _, Val) (dbg)?-
.........
CALL   minimax_alpha_beta(j, 0, 4, _, Val) (dbg)?-

/* la, on voit que minimax_alpha_beta ne sera pas appele sur k
car deja h(j) = 5 > 4, donc l'algorithme arrete l'exploration des
sous arbres, et remonte pour explorer le reste des noeuds
superieurs */
.........
CALL   minimax_alpha_beta(c, 4, 50, _, Val) (dbg)?-
.........
CALL   minimax_alpha_beta(f, 4, 50, _, Val) (dbg)?-
.........
.........
```

On pourra trouver l'integralite de la trace de cette execution dans le fichier **test.txt.**

En conclusion, les deux algorithmes, et plus particulierement le minimax_alpha_beta, marchent.
Nos allons maintenant appliquer ces deux algorithmes generaux au jeu du TicTacToe, dans la partie qui suit.

Le TicTacToe

L' algorithme du minimax adapte au Tictactoe.

L' algorithme du minimax presente dans la partie precedente etait depourvu de la notion de profondeur et s' appuyait sur la preexistence de l' arbre .

Ce nouvel algorithme , presente dans **Tictactoe1.pl,** se distingue donc tout d' abord par cette notion de profondeur, la profondeur devient un argument essentiel du predicat *t_minimax*. En effet, la profondeur indiquee, grace a un compteur decremente a chaque unification (ou iteration), permet de limiter l' exploration de l' arbre des coups possibles aux noeuds de moindre profondeur.

L' exploration n' est en realite pas un terme adapte a l' algorihme. En fait, l' arbre n' est cree qu' au fur et a mesure des iterations grace au predicat *t__arbre* contenu dans le fichier **Annexe.pl** . Danc cet algorithme , chacun des neuds correspond non seulement a un coup possible mais aussi a l' etat de la grille du Tictactoe sous la forme d' une liste .Cette liste, creee par linearisation de la grille par concatenation des lignes ,est constituee de 0 pour une case vide , de 1 pour le joueur max et de 2 pour le joueur min. La liste des fils obtenue par l' intermediaire du predicat t_arbre correspond a la liste des grilles resultant par le remplacement de un et un seul zero par 1 ou 2 en fonction du joueur auquel appartient le tour de jeu. On peut noter par ailleurs que le joueur correspondant est determine par les predicats *t_noeud_min* et *t_noeud_max* en fonction du nombre de zeros contenues dans la grille. (ces predicats reposent sur l' hypothese suivante: " le joueur max debute toujours la partie ") Cette construction recursive de l' arbre est egalement novatrice.

Dans un arbre de ce type , les feuilles correspondent a des coups terminaux :soient des grilles gagnantes , soient des grilles depourvues de 0 (match nul). t_minimax renvoie l' heuristique non seulement dans le cas ou il est applique a une feuille mais aussi lorsque il atteint un noeud de la profondeur indiquee lors du premier appel du predicat. Le calcul de cet heuristique est effectue grace au predicat *potentiel* contenu dans le fichier **Potentiel.pl**. Mis a part le fait que l' on occulte les appels recursifs au dela de cette profondeur les principes de recursivite developpes pour le minimax general sont conserves.

L' algorithme du minimax_alpha_beta adapte au tictactoe.

De meme que t_minimax , *t_minimax_alpha_beta* conserve les memes principes de recursion que sa version generale . Sa specificite est egalement issue de la profondeur et de la construction de l' arbre de jeu. C' est pourquoi , nous ne reprendrons donc pas les explications donnees ulterieurement.
Cet algorithme est developpe dans le fichier **Tictactoe2.pl** .

Demonstration et test.

Une demonstration du TicTacToe joue par l'ordinateur, est decrite dans le fichier **demo.pl.**

Conclusion

Ce projet nous a permis de mieux apprehender la programmation en Prolog sans contraintes. De par sa nature ludique, il nous a convaincu de son utilite . Pourquoi ne pas imaginer un programme en Prolog permettant de simuler un adversaire au Puissance 4 voire aux echecs ? La complexite ne reside plus que dans le choix d' une heuristique valable et dans la multiplication des coups possibles . Cependant la possibilite de modeliser un jeu ne signifie pas que ce dernier sera facilement exploitable pour les programmeurs.

Remarque : tous les fichiers requis pour ce projet se trouvent dans le repertoire : ***~es-skali/Prolog/Projet***