# MOIA

*Awale en prolog*

23 mai 2006

Olivio Mariano & Anthony Vernier

# Table des matières

# Introduction

On présente ici un étude de système decisionel en Prolog pouvant s'interfacer avec une application TCP/Ip.Le prolog a été crée avec l'intention de faire de l'analyse et du traitement de langage. Dans un deuxième temps on s'est rendu compte que ce langage pouvait avoir un champ d'application beaucoup plus large. En utilisant la logique des prédicats, le prolog permet d'avoir un abord très puissant face à la notion d'intelligence. La possibilité de création d'une IA a permi la programmation de nombreux jeux. Dans le cadre de l'apprentissage de ce langage de programmation il nous a été demandé de créer un programme pouvant être decisionel et "intelligent". Les pages suivantes expliquent les règles et le déroulement du jeu, la façon dont nous avons implémenté le jeu et les differentes contraintes imposées.

## Première partie

# Le jeu

Le jeu modelisé, appelé awelé ou awalé, trouve son origine dans l'antiquité. On y joue sur tout le continent africain mais aussi aux Philippines, en Malaisie, au Ceylan, en Louisiane et au Brésil. Logiquement chaque pays possède sa façon d'interpreter le règles de ce jeu. Une chose, par contre, ne change jamais : l'awelé est basé sur le calcul.

## 1 Contraintes de jeu

- Le nombre de joueurs maximum : 2
- Le matériel admis est : un moncale de 2 côtés de 6 cases chacun et 48 graines
- L'objectif du jeu : récolter le maximum de graines

– La préparation du jeu : les graines son placées à raison de 4 graines par case.
– Chaque joueur utilise son camp et l'ensemble de 6 cases de la rangée située face à lui.

# 2 Déroulement d'une partie

Le jeu se déroule à tour de rôle. Pour effectuer un coup, le joueur prend toutes les graines de l'une des cases de son camp et les distribue une par une, dans le trous suivants du camp (personnel et adversaire) selon un enchaînement circulaire. Dans le casu où le nombre de graines est tel que le tour répasse par la case, le joueur saute la case de départ sans rien mettre dedans.

# 3 Type de prise

## 3.1 Prise simple

Si la dernière graine du coup est posée dans le camp adverse et si cette case contient 2 ou 3 graines, le joueur les prend.

## 3.2 Prise multiple

Si la case précédente à celle où il vient d'avoir lieu le coup de type prise simple est située dans le camp adverse et contient 2 ou 3 graines, le joueur les prend également. Cette règle vaut aussi pour un ensembles de cases qui sont contiguës et aussi ayant les conditions nécessaires pour une prise. La prise s'arrête quand les conditions de prise ne sont plus vérifiées.

## 3.3 Donner à manger

Logiquement il est impossible q'un joueur puisse jouer deux foix de suite. Donc si les cases de l'adversaire sont toutes vides, le joueur, doit joueur un coup qui replace au moins une graine

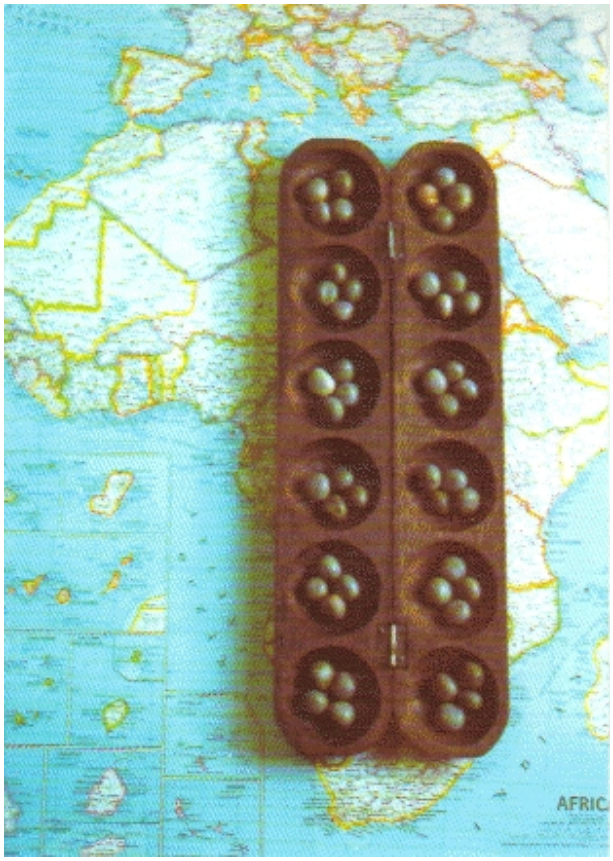


FIG. 1 – Le jeu de l'awele

dans le camp adverse. Il est impossible qu'un joueur puisse prendre toutes les graines du camp adverse (donc en laissant le joueur "affamé").

# 4 Conditions pour la fin de la partie

Le jeu pourra s'arrêter dans trois cas spécifiques :

1. Impossibilité de jouer - Un joueur ne peut pas jouer car il n'a plus de graines dans son camp.

2. Boucle infinie - Aucune prise n'est possible car le jeu cycle. Les mêmes positions se succédent indéfiniment.
3. Gain superieur à la moitié des graines - Le gain d'un joueur est supérieur ou égal a 25. (plus de la moitié des graines)
4. Gagner la partie - Le joueur qui possède le plus de graines a gagné la partie.

# 5 Démarches pour la création des fonctions

Il a été nécessaire de tester de manière intensive le jeu afin de comprendre ses contraines et ses limites. Il a fallu, ensuite, modéliser les prédicats généraux qui gèrent le cours du jeu, en fonction des règles de l'Awale. Il a été donc important de modéliser les régles d'arrêts suivantes :

1. Arrêter le jeu quand un des deux joueurs a gagné, ou quand aucun coup n'est possible.
2. Faire boucler le jeu tant qu'aucun joueur n'a gagné.
3. Donner la liste des possibilités de jeu au joueur : Comme il a été expliqué dans les règles, un joueur doit essayer de remplir le champ de son adversaire si celui-ci est vide.
4. Semer les graines, répartir la semence en respectant les règles.
5. Mettre à jour l'affichage de la nouvelle table de jeu.
6. Ramasser les graines si c'est possible, et mettre à jour les scores en conséquence.

# 6 L'intelligence artificielle

Plusieurs niveau d'intelligence peuvent être exprimés pour ce jeu. Pour nous assurer de pouvoir tester de manière correcte aussi la partie Systèmes Client / Serveur nous avons conçu un premier niveau qui n'est en fait pas du tout intelligent :

– une fonction (predicat) qui détermine le coup à jouer

– rédonne le maximum de graines à l'adversaire

– exécute le coup

Lorsque c'est au tour du joueur virtuel, l'algorithme commence par lancer une fonction spécifique

# 7 Les prédicats

1. Somme de tous les éléments d'une liste : avec ce prédicat on connaît le nombre de graines d'un côté.

```
somme([],0).
somme([X|L],Res) :-
somme(L,Res1),
Res is X + Res1.
```

2. Ce prédicat vérifie qu'on posséde le bon nombre de graines qui ont été prises et le bon nombre de graines sur le plateau.

```
plateau_valide(L1,L2,No,Nj) :-
somme(L1,Res1),
somme(L2,Res2),
somme([Res1,Res2,No,Nj],48).
```

3. Le prédicat vérifie que tous les éléments d'une liste sont égaux à zero.

```
tous_zero([]).
tous_zero([X|L]) :- X is 0,
   tous_zero(L).
```

4. – Renvoi [0] si A plus grand que B

- Renvoi 1 si est inférieur
- Renvoi 2 si égal

```
maximum(A,B,[0]) :- A>B.
maximum(A,B,[1]) :- A<B.
maximum(A,A,[2]).
```

5. Sous-prédicat de plus_grand

```
plus_grand3([],Acc,Acc).

plus_grand3([X|L],Acc,Res) :-
X < Acc,
plus_grand3(L,Acc,Res).

plus_grand3([X|L],Acc,Res) :-
plus_grand3(L,X,Res).
```

6. Renvoi la position : res est la position de val dans la liste l

```
position_liste(L,Val,Res) :-
pos_liste2(L,Val,1,Res).
```

7. Sous-prédicat de pos_liste2

```
pos_liste2([Val|L],Val,Acc,Acc).

pos_liste2([X|L],Val,Acc,Res) :-
Acc2 is Acc + 1,
pos_liste2(L,Val,Acc2,Res).
```

8. Détermine le coup qui rédonne le plus de graines à l'adversaire

```
plus_grand([X1,X2,X3,X4,X5,X6],NumCase) :-
Res1 is X1 + 0,
Res2 is X2 + 1,
Res3 is X3 + 2,
Res4 is X4 + 3,
Res5 is X5 + 4,
Res6 is X6 + 5,
plus_grand3([Res1,Res2,Res3,Res4,Res5,Res6],0,Val),
position_liste([Res1,Res2,Res3,Res4,Res5,Res6],Val,NumCase).
```

9. Etabli la fin du jeu. Les trois prédicats suivants permettent de savoir quand il faut s'arreter.

```
fin_jeu(L1,L2,No,Nj,1,G,IdJ) :-
plateau_valide(L1,L2,No,Nj),
tous_zero(L1),
tous_zero(L2),
maximum(No,Nj,G).


fin_jeu(L1,L2,No,Nj,1,G,1) :-
plateau_valide(L1,L2,No,Nj),
tous_zero(L1),
est_affame(L2),
somme(L2,S),
NewNj is Nj + S,
```

```
maximum(No,NewNj,G).

fin_jeu(L1,L2,No,Nj,1,G,0) :-
plateau_valide(L1,L2,No,Nj),
tous_zero(L2),
est_affame(L1),
somme(L1,S),
NewNo is No + S,
maximum(NewNo,Nj,G).


fin_jeu(_,_,No,Nj,Tc,0,IdJ) :- No > 24.
fin_jeu(_,_,No,Nj,Tc,1,IdJ) :- Nj > 24.
```

10. Sous_prédicat de remplacer

```
remplace([X|L],1,X,[0|L]).

remplace([X|L],N,Nbgr,[X|Lres]) :-
N2 is N - 1,
remplace(L,N2,Nbgr,Lres).
```

11. Remplacer utilise le prédicat remplace. Ce prédicat sert à remplacer une valeur par 0.

```
remplacer(LO,LJ,Res,Nb,NLO,LJ) :-
Res < 7,
remplace(LO,Res,Nb,NLO).
```

```
remplacer(LO,LJ,Res,Nb,LO,NLJ) :-
Res2 is Res - 6,
remplace(LJ,Res2,Nb,NLJ).
```

12. Retourne la valeur de l'élément situé à la position "Position" dans la liste.

```
retourElem([T|_],1,T):-!.
retourElem([_|Liste],Position,Retour):- P2 is Position - 1, retourElem(Liste,P2,Ret
```

13. Change la valeur d'un élément situé à la position "Position" dans la liste.

```
changeElem([],_,_,_).
changeElem([_|Liste],1,V2,[V2|Liste]):- !.
changeElem([T|Liste],Position,V2,[T|ListeRetour]):-
P2 is Position - 1,
changeElem(Liste,P2,V2,ListeRetour).
```

14. Sous-prédicat de semer

```
pose1graine(Lo,Lj,NumCase,0,NbGr,NLo,NLj,NCaseC) :-
pose1graine(Lo,Lj,NumCase,12,NbGr,NLo,NLj,NCaseC).

pose1graine(Lo,Lj,NumCase,CaseC,0,Lo,Lj,NCaseC) :-
NCaseC1 is CaseC - 1,
transforme(NCaseC1,NCaseC).

pose1graine(Lo,Lj,NumCase,NumCase,NbGr,NLo,NLj,NCaseC) :-
NextCase1 is NumCase + 1,
NextCase is NextCase1 mod 12,
```

```
pose1graine(Lo,Lj,NumCase,NextCase,NbGr,NLo,NLj,NCaseC).


pose1graine(Lo,Lj,NumCase,CaseC,NbGr,NLo,NLj,NCaseC) :-
CaseC < 7,
retourElem(Lo,CaseC,Val),
NVal is Val + 1,
changeElem(Lo,CaseC,NVal,NLo1),
NewCaseC is CaseC + 1,
NNbGr is NbGr - 1,
pose1graine(NLo1,Lj,NumCase,NewCaseC,NNbGr,NLo,NLj,NCaseC).
```

15. Le prédicat sert à passer d'un côté à l'autre du plateau.

```
transforme(0,12).
transforme(X,X).


pose1graine(Lo,Lj,NumCase,CaseC,NbGr,NLo,NLj,NCaseC) :-
CaseC > 6,
TrueCaseC is CaseC - 6,
retourElem(Lj,TrueCaseC,Val),
NVal is Val + 1,
changeElem(Lj,TrueCaseC,NVal,NLj1),
NewCaseC1 is CaseC + 1,
NewCaseC2 is NewCaseC1 mod 12,
transforme(NewCaseC2,NewCaseC),
NNbGr is NbGr - 1,
pose1graine(Lo,NLj1,NumCase,NewCaseC,NNbGr,NLo,NLj,NCaseC).
```

16. Utilise le prédicat pose1graines et gére l'action de distribuer les graines sur le plateau.

```
semer(Lo,Lj,NumCase,NbGr,NLo,NLj,NNumCase) :-
CaseC is NumCase + 1,
pose1graine(Lo,Lj,NumCase,CaseC,NbGr,NLo,NLj,NNumCase).
```

17. Utilise le prédicat semer et distribue les graines sur le plateau (vide la case qui est jouée).

```
distribue_graines([N1,N2,N3,N4,N5,N6],[L1,L2,L3,L4,L5,L6],Res,Resf,NewLO,NewLJ)
:-
remplacer([N1,N2,N3,N4,N5,N6],[L1,L2,L3,L4,L5,L6],Res,NbGr,NewLO1,NewLJ1),
semer(NewLO1,NewLJ1,Res,NbGr,NewLO,NewLJ,Resf).
```

18. Sous-prédicat de prise-graines.

```
enleve_graines(L,L,G,G,0).

enleve_graines(L,L,G,G,R) :-
retourElem(L,R,V),
V > 3.

enleve_graines(L,NL,G,NG,R) :-
retourElem(L,R,V),
V < 4,
changeElem(L,R,0,NL1),
NG1 is G + V,
R1 is R - 1,
enleve_graines(NL1,NL,NG1,NG,R1).
```

19. Détermine le nombre de graines qui sont prises : modifie donc l'état du plateau.

```
prises_graines(Lo,Lj,GrO,GrJ,R,Rf,GrO,NGrJ,NLo,Lj) :-
R > 6,
Rf < 7,
enleve_graines(Lo,NLo,GrJ,NGrJ,Rf).

prises_graines(Lo,Lj,GrO,GrJ,R,Rf,NGrO,GrJ,Lo,NLj) :-
R < 7,
Rf > 6,
Rf1 is Rf mod 6,
transforme(Rf1,Rf2),
enleve_graines(Lj,NLj,GrO,NGrO,Rf2).
```

20. Fonction principale de déroulement d'une partie.

```
jouer_ordi(L1,L2,NbgrO,NbgrJ,TypC,Gagnant) :-
tous_zero(L1),
NbJ is 48 - NbgrO,
maximum(NbgrO,NbJ,Gagnant).

jouer_ordi([N1,N2,N3,N4,N5,N6],[L1,L2,L3,L4,L5,L6],NbgrO,NbgrJ,TypC,NewNbgrO,NewNbg
:-
plus_grand([N1,N2,N3,N4,N5,N6],Res),
distribue_graines([N1,N2,N3,N4,N5,N6],[L1,L2,L3,L4,L5,L6],Res,Resf,NewLO1,NewLJ1),
prises_graines(NewLO1,NewLJ1,NbgrO,NbgrJ,Res,Resf,NewNbgrO,NewNbgrJ,NewLO,NewLJ).
```

# 8 Problèmes à résoudre

Il reste à mettre en place plusieurs points importants :

– détection du type de coup (rédonne, fin, sans-fin, normal).

– le code pourrait être optimisé pour eviter des redondances.

En dehors de ces points là il faudrait mettre en place une véritable héuristique capable de rendre le programme vraiment "intelligent".