Projet de IA02 – Printemps 2016

Le Jeu de Khan en Prolog

Groupe Semaine A

Romain HUBER & Mathilde GAUDEFROY
16/06/2016

Table des matières

Introduction

Dans le cadre de l'UV IA02, nous avons dû implémenter le jeu de Khan. C'est un jeu qui se joue à deux adversaires. L'objectif de ce projet est de permettre une partie Humain vs Humain, Humain vs Machine et Machine vs Machine. Ceci, tout en respectant bien évidemment les règles du jeu. Le jeu doit pouvoir être joué jusqu'à la victoire d'un des deux joueurs.

La machine doit être capable de « gagner » une partie, ainsi, il faut réfléchir et implémenter une « intelligence artificielle » pour cette dernière.

Le but de ce jeu est de capturer la Kalista adverse avec sa propre Kalista ou l'un de ses cinq Sbires. Le jeu se déroule sur un plateau de 6*6 cases. Un pion du jeu peut se déplacer du nombre de cases correspondant au chiffre indiqué sur sa case de départ (ces chiffres correspondent à 1,2 ou 3). Les déplacements sont verticaux et horizontaux avec possibilité de changer de direction en cours de route. Il n'est pas possible de passer au-dessus d'un pion.

I. Présentation et description des prédicats principaux

1) Premier tour

Nous avons créé un prédicat spécialement pour le premier tour. En effet, lors du premier tour, le joueur peut déplacer n'importe quelle pièce. Il a donc le choix entre voir le plateau, voir les pièces qu'il peut déplacer ou choisir une pièce à déplacer.

```
premierTour :- write('Que voulez-vous faire ?'), nl,
write('1 pour voir les pièces que vous pouvez deplacer'), nl,
write('2 pour voir le plateau'), nl,
write('3 pour choisir une pièce'), nl,
read(X), choixPremierTour(X).
```

2) Boucle du jeu

Après le premier tour, la boucle de jeu est appelée. Dans cette boucle, nous avons trois cas de figures qui se présentent au joueur :

- Soit c'est la fin du jeu et on appelle donc le prédicat finDuJeu qui vérifie quel joueur a perdu sa Khalista et affiche un message de félicitations à l'autre joueur vainqueur.
- Soit ce n'est pas la fin du jeu et c'est donc un tour normal. On change donc de joueur et on regarde si ce dernier a des pièces à déplacer. Nous avons donc deux cas possibles en fonction du résultat de l'évaluation des prédicats `getPieceDispo(X, P)` et `\+isVide(P)`.

```
boucleTour :- finDuJeu, currentPlayer(X), write('Félicitation, le joueur '), write(X),  write(' a gagné'), !.
boucleTour :- nl,affichage_pion, nl, changementPlayer, currentPlayer(X), write('Au tour du joueur : '), write(X), nl, getPieceDispo(X, P), \+isVide(P), tourRespect.
boucleTour :- tourNonRespect.
```

a. Cas 1

Soit le joueur a des pièces à déplacer et on lui propose les choix suivants à l'aide du prédicat choixRespect :

- Voir le plateau
- Voir les pièces qu'il peut déplacer : Ceci en fonction du type de cas où il se trouve.
- Choisir la pièce qu'il veut déplacer : Dans ce cas, le prédicat choixPiece est appelé. Ce prédicat vérifie si la pièce choisie est bien déplaçable et que le joueur concerné est réellement en possession de cette pièce. Si ces conditions sont validées, on lui affiche les différents mouvements qu'il peut effectuer avec cette pièce, grâce au prédicat movesDispo. Ensuite, on appelle le prédicat permettant de déplacer une pièce, si ce prédicat a été correctement évalué, on modifie dynamiquement la position du Khan et c'est à l'autre joueur de jouer.

```
tourRespect :- write('Que voulez-vous faire ?'), nl,
write('1 pour voir les pièces que vous pouvez déplacer'), nl,
write('2 pour voir le plateau'), nl,
write('3 pour choisir une pièce'), nl,
read(X), choix(X).

choix(1) :- currentPlayer(P), getPieceDispo(P, Z), write(Z), nl, !, tourRespect.
choix(2) :- affichage_pion, !, tourRespect.
choix(3) :- write('Entrez la pièce que vous voulez déplacer'), nl, read(Y), choixPiece(Y), !.
choix(_) :- write('Chiffre incorrect'), nl, tourRespect.
```

b. Cas 2

Soit le joueur n'a plus de pièces à déplacer et les choix suivants lui sont donc proposés à l'aide du prédicat choixNonRespect :

- Voir le plateau : La grille est affichée.
- Voir les pièces qu'il peut déplacer : Ce choix permet de récupérer toutes les pièces disponibles pour le joueur. Ces pièces sont les mêmes qu'au premier tour, car le khan n'est pas « respecté », on utilise donc le prédicat effectuant le choix au premier tour.
- Voir ses pièces n'étant pas « en vie » : Les pièces affichées sont les pièces que l'on peut remettre en jeu, c'est-à-dire celles qui ne sont pas mortes.
- Remettre une pièce sur le plateau : Ce choix permet au joueur de choisir la pièce qu'il souhaite remettre sur le plateau, et le prédicat gérant cette remise est donc appelé.
- Déplacer une pièce : Ce choix permet au joueur de choisir la pièce qu'il souhaite déplace, et le prédicat gérant cette remise est donc appelé.

```
tourNonRespect :- write('Que voulez-vous faire ?'), nl,
write('1 pour voir les pièces que vous pouvez deplacer'), nl,
write('2 pour voir le plateau'), nl,
write('3 pour voir vos pièces mortes'), nl,
write('4 pour remettre une pièce sur le plateau'), nl,
write('5 pour deplacer une pièce'), nl,
read(X), choixNonRespect(X).

choixNonRespect(1) :- currentPlayer(P), getPieceDispoPremierTour(P, Z), write(Z), nl, !,
tourNonRespect.
choixNonRespect(2) :- affichage_pion, !, tourNonRespect.
choixNonRespect(3) :- currentPlayer(X), pieces(X,P), getPieceMorte(P,PM), write(PM), nl, !,
tourNonRespect.
choixNonRespect(4) :- write('Entrez la pièce que vous voulez remettre sur le jeu'), nl, read(Y),
putBackPiece(Y), !.
choixNonRespect(5) :- write('Entrez la pièce que vous voulez déplacer'), nl, read(Y),
choixPieceNonRespect(Y), !.
```

choixNonRespect(_) :- write('Chiffre incorrect'), nl, tourNonRespect.

La dernière ligne choixNonRespect(_) :- write('Chiffre incorrect'), nl, tourNonRespect. permet de vérifier que le joueur rentre bien un chiffre compris entre 1 et 5, sinon les choix lui sont reproposés jusqu'à ce qu'il rentre un chiffre correct. Le cut présent dans les autres lignes permet de ne pas passer dans cette dernière ligne, si le chiffre entré est correct.

3) AllMoves

Le prédicat allMoves est très utile et est donc utilisé dans de nombreux prédicats. Ce dernier renvoie, dans une variable Res, passée en paramètre, la liste des mouvements possibles pour la pièce concernée, aussi passée en paramètre.
Les mouvements possibles dépendent évidemment de plusieurs facteurs tels que la position du Khan, de combien de cases la pièce peut-elle avancer, les autres pièces présentent sur son chemin…etc.

```
allMove(Piece, Res) :- position(Piece, X, Y),
currentPlayer(Player),
get_mvmtsPiece(Piece, M),
 setof(S, movesDispo(Piece,M,Player,X,Y,S), Res).
```

Nous avons utilisé la fonction prolog setof qui permet de créer une liste triée et sans doublons.

4) Récupérer les pièces disponibles

Le prédicat getPieceDispo permet de savoir quelles pièces du joueur concerné sont disponibles, c'est-à-dire quelles sont les pièces que le joueur peut jouer. Elle prend en compte le khan, les pièces en vie du joueur et les mouvements que ses pièces peuvent faire. Une pièce est dite disponible si elle peut effectuer au moins un mouvement.

```
getPieceDispo(Player, Res) :- boardSelected(Board), khan(K), position(K, X, Y),
get_mvmts(Board, X, Y, M), pieces(Player, P), en_vies(P, V), piecesDispo(Board, M, V, Res).
```

II. Choix structure de données

1) Plateau de jeu

Voici ci-dessous les prédicats nous permettant de définir notre plateau de jeu avec les différents mouvements possibles.
Le prédicat board correspond au plateau de jeu. Initialement, il est vide. En effet, aucune pièce n'est sur le plateau au tout début. C'est au joueur de choisir ses emplacements, tout en respectant les conditions. Par exemple, au départ, il ne peut disposer ses pièces que sur les deux premières lignes. Nous avons choisi de représenter le fait qu'une case soit vide par un « 0 ».

```
board([[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0]]).
```

Les 4 prédicats grille suivant nous permettent de définir les mouvements possibles sur chaque case. Le premier paramètre correspond à la position du plateau. En effet, le joueur 1 peut choisir de jouer sur les deux premières lignes en haut ou en bas du tableau ou sur les deux premières colonnes à droite ou à gauche du tableau.

```
grille(1,[[3,1,2,3,3,1],[1,2,1,2,3,1],[3,1,2,1,3,3],[2,1,2,3,2,1],[3,3,2,1,1,2],[3,2,3,2,1,2]]).
grille(2,[[3,3,2,3,2,3],[2,3,1,1,1,1],[3,2,2,2,1,2],[2,1,3,1,2,3],[1,2,2,3,3,3],[2,1,1,3,1,1]]).
grille(3,[[2,1,2,3,2,3],[2,1,1,2,3,3],[1,2,3,2,1,2],[3,3,1,2,1,3],[1,3,2,1,2,1],[1,3,3,2,1,3]]).
grille(4,[[1,1,3,1,1,2],[3,3,3,2,2,1],[3,2,1,3,1,2],[2,1,2,2,2,3],[1,1,1,1,3,2],[3,2,3,2,3,3]]).
```

2) Différentes pièces

Nous avons choisi de représenter l'ensemble des pièces des deux joueurs sous forme de liste.
Ainsi notre prédicat pieces se décompose en deux parties.
La première partie correspond à une liste composée de toutes les pièces des deux joueurs.

```
pieces([sR1,sR2,sR3,sR4,sR5,kaR, sO1,sO2,sO3,sO4,sO5,kaO]).
```

La deuxième partie prend en paramètre le joueur concerné (1 ou 2) et y associe ses pièces. Le joueur 1 joue avec les pièces rouges et le joueur 2 joue avec les pièces ocres.

```
pieces(1, [sR1,sR2,sR3,sR4,sR5,kaR]).
pieces(2, [sO1,sO2,sO3,sO4,sO5,kaO]).
```

3) Faits dynamiques

Nous avons utilisés les faits dynamiques suivants :

- Position(nom_pion, positionX, positionY) : fait permettant de savoir la position de chaque pion et s'il est en vie ou non. Si sa position existe, il est en vie, si non, il est mort.
- Nom du khan : fait permettant de savoir sur quelle pièce est le Khan.
- Grille sélectionnée : fait permettant de savoir quelle grille le joueur a sélectionnée, c'est-à-dire le sens dans lequel le joueur a choisi de positionner le plateau.
- Joueur actuel : fait permettant de savoir si c'est le tour du joueur n°1 ou du joueur n°2.

Ainsi, nous pouvons modifier l'état du jeu (présence et position du khan, pièces sur le plateau…etc) facilement et rapidement.

Nous avons le prédicat remiseZero qui permet de remettre le jeu à zéro en « supprimant » la position du plateau choisie par le joueur, en « recréant » le khan associé au joueur 1 et en affectant le joueur n°1 au joueur actuel.

```
remiseZero :- asserta(boardSelected(1)), retractall(boardSelected(_)),
asserta(khan(1)), retractall(khan(_)),
asserta(currentPlayer(1)), retractall(currentPlayer(_)).
```

Grâce à notre structure de données et au prédicat *en_vies*, nous pouvons facilement savoir quelles pièces sont en vies et quelles pièces ne le sont pas. Ceci est fait en testant si le fait dynamique position associée à la pièce concernée existe ou non.

```
en_vies([], []).
en_vies([T|Q], [T|Res]) :- position(T,_,_), get_mvmtsPiece(T, M), en_vies(Q, Res), !.
en_vies([_|Q], V) :- en_vies(Q, V).
```

De même, il est également très facile d'accéder aux pièces mortes.

```
getPieceMorte([], []).
getPieceMorte([T|Q], [T|Res]) :- \+position(T,_,_), getPieceMorte(Q, Res), !.
getPieceMorte([_|Q], V) :- getPieceMorte(Q, V).
```

III. Description de l'IA

Nous avons choisi d'implémenter l'intelligence artificielle de la machine à l'aide de l'algorithme MINIMAX.

1) Fonctionnement algorithme MINIMAX

L'algorithme minimax est un algorithme qui s'applique à la théorie des jeux pour les jeux à deux joueurs à somme nulle (et à information complète) consistant à minimiser la perte maximum, c'est-à-dire dans le pire des cas.

Il amène l'ordinateur à passer en revue toutes les possibilités pour un nombre limité de coups et à leur assigner une valeur qui prend en compte les bénéfices pour le joueur et pour son adversaire.

Le meilleur choix est alors celui qui minimise les pertes du joueur tout en supposant que l'adversaire cherche au contraire à les maximiser. C'est pour cette raison que le jeu est dit « à somme nulle ».

2) Paramètres d'évaluation

Afin d'assigner une valeur à chaque coup, nous avons choisi de prendre en compte les paramètres suivants :

- Le nombre de pièces restants pour chacun des deux joueurs
- La distance des pions de la machine à la kalista du joueur
- La distance des pions du joueur à la kalista de la machine
- Si la khalista adverse a été mangé.

IV. Difficultés rencontrées et améliorations possibles

Prolog est un langage nouveau pour nous, dont nous avons pris connaissance il y a 6 mois. De plus, ce langage est très différent de ce que nous avons l'habitude d'utiliser. En effet, Prolog est basé sur le calcul de prédicats du premier ordre. Les concepts fondamentaux sont l'unification, la récursivité et le retour sur trace. Par conséquent, nous avons perdu du temps au début du projet pour prendre en main réellement ce langage et être capable d'implémenter un tel jeu.

En effet, nous avions déjà codé une partie du jeu sans réfléchir à la suite, et nous n'avions pas pris le temps de corriger cette partie ce qui a entrainé de nombreux disfonctionnement et une lourdeur au niveau du nombre de méthode. Nous avons dû par exemple faire un copier-coller d'une grande partie des méthodes pour l'intelligence artificielle afin de faire des tests virtuels. Cependant, une fois l'implémentation totale de l'intelligence artificielle finie, il aurait été possible de ne pas avoir à recréer tout le plateau « pour de faux » en recopiant toutes les positions, mais en utilisant le vrai plateau en pensant à bien le remettre en l'état à chaque test (chose qui est déjà faite sur le plateau virtuel).

Au niveau des améliorations, la fonction d'évaluation de l'IA n'est pas optimale et pourrait être revue avec une meilleure compréhension du jeu. Par exemple, elle ne prend pas ne compte le fait qu'elle puisse remettre une pièce en jeu. Elle décidera toujours de déplacer une pièce. De plus, bien qu'elle prenne en compte le fait qu'un joueur va pouvoir remettre une pièce en jeu dans un certain cas, elle ne réfléchit pas aux conséquences de cette remise en jeu.

L'algorithme MINIMAX que nous avons implémenté prend beaucoup de temps de calcul et cela empêche d'aller à une profondeur supérieure à deux coups. Pour régler ce problème et améliorer la recherche, il faudrait utiliser l'algorithme MINIMAX alpha-beta qui diminuerai grandement les temps de calcul.

Lors de la phase de placement, les pièces sont placées aléatoirement (en respectant les règles) pour la partie intelligence artificielle. Réfléchir au positionnement serait une meilleure idée, mais il faudrait pour cela connaître les stratégies du jeu.

Enfin, l'amélioration ultime serait de coupler ce programme avec un autre langage afin d'y inclure une interface graphique.