IA02 : Résolution de Problèmes

Prolog Chicago Stock Exchange

Authors :

Antoine Pouillaude
GI05 Filière SRI

Frédéric Rouffineau
GI04

Université de Technologie de Compiègne
Rue Roger Couttolenc
60200 Compiègne

17 juin 2015

Table des matières

1 Introduction

Nous avons eu pour projet, dans le cadre de l'Unité de Valeur IA02 (Résolution de Problèmes et programmation logique), de coder le jeu Chicago Stock Exchange à l'aide du langage Prolog. Il s'agit d'un jeu de plateau pouvant se jouer à 2 joueurs ou plus durant lequel ils vont essayé de maximiser leur profit en vendant et en achetant des denrées alimentaires. Afin de simplifier l'implémentation du jeu sous Prolog, nous avons limité le nombre de joueurs à 2. Nous devions également créer une Intelligence Artificielle afin de pouvoir jouer contre l'ordinateur.

Lors d'un tour de jeu un joueur va déplacer un pion Trader sur des piles de denrées d'au moins 1 et d'au plus 3 piles. Il/elle va alors récolter les deux jetons situés sur le dessus des piles adjacentes du Trader, choisir d'en garder une et d'en jeter une. Jeter une denrée à pour effet de diminuer la valeur en bouse de cette-denière. Le joueur avec le stock de plus grande valeur gagne la partie.

1.1 Modèle

Le jeu sera majoritairement modéliser avec une liste représentant l'état de jeu courant. Cette liste sera de la forme :

```
[Liste des piles de denrées, État de la bourse, Position du Trader, Réserve du
   joueur 1, Réserve du joueur 2]
```

Les différentes listes qui composent l'état de jeu courant auront les formes indiquées ci-dessous.

Bourse L'état de la bourse sera modélisé par une liste contenant l'état des différentes denrées côtées à Chicago.

```
[Etat de la marchandise 1, Etat de la marchandise 2, ..., Etat de la marchandise
    n]
```

L'état d'une marchandise est quant à lui une simple liste avec, comme premier élément, le nom de la marchandise en question et en seconde position la valeur en bourse de ladite denrée.

```
[Marchandise,Valeur en bourse]
```

Piles de marchandises. Le jeu, dans son état initial est composée de 9 piles de 4 denrées différentes. Ces piles sont modélisées par des listes elle-mêmes inclues dans une grande liste contenant toutes les piles.

```
[[Pile1],[Denrée1,Denrée2,Denrée3,Denrée4],[Pile3],...,[PileN]]
```

Trader La position du Trader sera donnée à l'aide d'un chiffre entre 0 et le nombre de piles contenues dans l'état de jeu courant.

Réserve La réserve d'un joueur sera modélisé par une liste avec en tête de liste le nom du joueur en question suivi des denrées que ce joueur possède.

```
[Nom du joueur, Denrée1,Denrée2,...,DenréeN]
```

Coup de jeu Un coup de jeu doit contenir 4 paramètres :

- le nom du joueur
- la valeur du déplacement (1,2 ou 3)
- la ressource gardée
- la ressource vendue

Ces données seront regroupées dans une liste :

```
[Nom du joueur, Déplacement,Marchandise gardée,Marchandise vendue]
```

2 Principaux prédicats

2.1 Gestion du plateau

2.1.1 Affichage

Nous avons regroupé le code pour l'affichage du plateau de jeu dans le fichier *game_ display.pl*. Le prédicat d'affichage du plateau est le suivant :

```
%%%% display_game(+Game_State_To_Be_Displayed)
%% The following predicate displays a game state.
display_game(State) :-
              [Stacks,S,TP,RJ1,RJ2] = State,
              generate_board_from_stock(S,Board),
              display_board(Board),
              nl, nl, nl,
              display_stack(Stacks,TP),
              nl, nl,
              display_players([RJ1,RJ2]),nl,nl,
              evalState(State,Earnings),
              tab(5),display_earnings(Earnings),nl,nl,nl.
```

Dans la queue de cette règle on retrouve des appels aux prédicats :

- *generate_ board_ from_ stock* qui va, à partir d'un Template d'affichage générer une liste correspondant au plateau graphique à afficher.
- *display_ board* qui affiche le Template généré par la règle précédente.
- *display_ stacks* qui afficher le dessus de chaque piles de marchandises et le nombre de jetons restant dans ces piles.
- *display_ players* qui affiche l'état des stocks des différents joueurs.
- *display_ earnings* qui va afficher le gain en USD des deux joueurs en fonction de leur stocks.

Au final nous obtnons le rendu suivant :

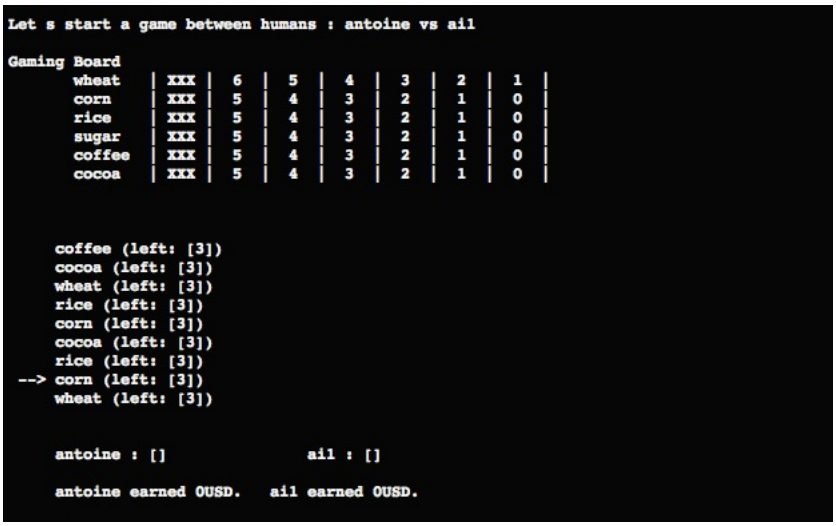

FIGURE 2.1 – The result of the gaming board display predicate.

2.1.2 Génération du plateau

Au début de la partie le plateau doit être généré. Nous faisons appelle à la règle de construction *starting_state* :

```
%%%% starting_state(?State_To_Be_Generated,+Player_1's_Name,+Player_1's_Name)
%% This predicate generates the initial state of the game.
starting_state(State,NameJ1,NameJ2) :-
              generating_stacks([[wheat,6],[corn,6],[rice,6],[sugar,6],[coffee
                  ,6],[cocoa,6]],Stacks,9),
              Bourse = [[wheat,7],[corn,6],[rice,6],[sugar,6],[coffee,6],[cocoa
                  ,6]],
              random(0,8,TP),
              State = [Stacks,Bourse,TP,[NameJ1],[NameJ2]].
```

Dans ce prédicat nous appelons la règle *generating_stacks*, qui va générer les piles de manières pseudo-aléatoire en faisant appel à la fonction built-in random3. Par la suite la position initiale du pion Trader est tirée de manière pseudo-aléatoire.

Il est très intéressant de remarquer le fait que les tirages se font de manière pseudo-aléatoire, à chaque fois le premier état de jeu généré après l'ouverture de l'interpréteur gprolog est toujours le même.

2.2 Déroulement du jeu

2.2.1 Vérification de coup

Avant de jouer un coup il est important de savoir si celui-ci existe. En effet, si le joueur rentre une mauvaise combinaison de déplacement et de pièces à garder, il faut alors empêcher le joueur de jouer la combinaison. C'est pour cela que nous avons créer le prédicat *is_possible* qui va vérifier si le coup de jeu que l'utilisateur souhaite jouer est conforme à l'état actuel du plateau de jeu.

```
%%%% is_possible(+List_Of_Stacks,+Current_Trader_Position,+
    Move_The_Player_Wants_To_Make,?Return_Value)
%%% Return_Value = 0 if the move is possible, 1 if the two given element are not
     on the two stacks adjacent ot the Trader position, 2 if the move applied to
     the Trader is not in [1,2,3].
%% This predicate will check if a move is possible in the current game
    configuration.
is_possible(Stacks,TP,[_,Pos,Keep,Sell],ReturnValue):- Pos < 4, Pos >0, !,
              get_indexes(Stacks,TP,Pos,[_,ISup,IInf]),
              nth0(ISup,Stacks,E1),
              nth0(IInf,Stacks,E2),
              test_comb(Keep,Sell,E1,E2,ReturnValue).
is_possible(_,_,_,2).

%%%% test_comb(+The_Element_The_Player_Wants_To_Keep,+
    The_Element_The_Player_Wants_To_Sell,+One_Of_The_Adjacent_Stack,+
    The_Other_Adjacent_Stack,?The_Return_Value)
%%% Return_Value = 0 if the two products are on top of the two stacks adjacent
    to the Trader's position, = 1 otherwise.
%% This rule test if the two elements given in the move are on top of the two
    adjacent stacks of the Trader stack.
test_comb(Keep,Sell,[Keep|_],[Sell|_],0) :- !.
test_comb(Keep,Sell,[Sell|_],[Keep|_],0) :- !.
test_comb(_,_,_,_,1).
```

Ci-dessus nous avons défini deux prédicats pour tester les combinaisons. La règle *test_ comb* sert à verifier si l'un des éléments donnés par l'utilisateur est bien sur l'une des piles adjacentes au Trader après déplacements. Le prédicat *get_ indexes* va chercher l'index des trois piles concernées par le coup de jeu.

2.2.2 Jouer un coup

Une fois que nous avons vérifié que le coup était viable, il faut maintenant le jouer et modifier l'état de jeu en conséquence. C'est pour cela que le prédicat *play* a été créé.

```
%%%% play(+Game_State,+Move_To_Apply,?New_State,-Return_Value)
%% This predicate will update the game state according to the move provided to
    the function.
play([Stacks,_,_,_,_],_,_,3) :- length(Stacks,Le),Le =< 2, !.
play([Stacks,S,TP,RJ1,RJ2],[Player,Pos,Keep,Sell],NewState,RetValue) :- length(
    Stacks,Le),Le > 2,is_possible(Stacks,TP,[Player,Pos,Keep,Sell],ExitStatus),
              play_ret([Stacks,S,TP,RJ1,RJ2],[Player,Pos,Keep,Sell],NewState,
                  RetValue,ExitStatus).

%%%% play_ret(+Game_State,+Move_To_Apply,?New_State,-Return_Value,+
    Value_Returned_By_The_is_possible_Predicate)
play_ret([Stacks,S,TP,RJ1,RJ2],[Player,Pos,Keep,Sell],NewState,0,0) :- !,
              get_indexes(Stacks,TP,Pos,[NTP,ISup,IInf]),
              nth0(ISup,Stacks,E1),
              nth0(IInf,Stacks,E2),
              pop(E1,NE1), pop(E2,NE2),
```

```
                update_i(Stacks,ISup,NE1,NTempStacks), update_i(NTempStacks,IInf,
                    NE2,NStacks),
                clean_list_and_update_TP(NStacks,NewStacks,NTP,NTPmod),
                length(NewStacks,Length), my_mod(NTPmod,Length,NewTP),
                add_to_player(Player,Keep,RJ1,RJ2,NRJ1,NRJ2),
                member_sec_order_e(S,Sell,Elt), [Name,Value] = Elt, NValue is
                    Value - 1,
                NElt = [Name,NValue], update_e(S,Elt,NElt,NS),
                NewState = [NewStacks,NS,NewTP,NRJ1,NRJ2].
play_ret([Stacks,S,TP,RJ1,RJ2],_,NewState,ExitStatus,ExitStatus) :- NewState =[
    Stacks,S,TP,RJ1,RJ2].
```

Il y a en fait deux prédicats différents avec le même nom. Le premier va récupérer l'entrée de l'utilisateur ou de l'Intelligence Artificielle et tester si le coup possible. La règle de test va retourner une valeur en fonction du résultat du test. Si cette valeur est 0 alors le coup viable et le second prédicat du même nom va alors modifier l'état du jeu en conséquence.

2.3 Intelligence artificielle

Afin de pouvoir seul contre l'ordinateur nous avons créé une intelligence artificielle basée sur l'algorithme de recherche Min-Max avec élagage Alpha/Beta. Cet algorithme va être implémenté par deux prédicats importants.

2.3.1 Trouver les coups possibles

Il faut d'abord trouver tous les coups que l'ordinateur pourra jouer. C'est à cela que sert le prédicat *all_possible_moves*.

```
%%%% all_possible_moves(+State_Of_The_Game,+Player_Who_Will_Make_The_Move,?
    Possible_Moves)
%% The following predicate will get all the possible moves that the ai could
    make.
all_possible_moves([Stacks,_,TP,_,_],Player,PossibleMoves) :- possible_moves(
    Stacks,TP,Player,[1,2,3],PossibleMoves).


%%%% possible_moves(+Stacks_Of_The_Game,+Trader_Position,+
    Player_Who_Will_Make_The_Move,+List_Of_Jumps_To_make,?List_Of_Moves)
%% possible_moves will get the list of the moves possible by applying all the
    jumps given in the fourth parameter. In that game the list should be
    [1,2,3].
possible_moves(_,_,_,[],[]) :- !.
possible_moves(Stacks,TP,Player,[H|T],ListMoves) :-
                possible_moves(Stacks,TP,Player,T,SListMoves),
                get_possible_list(Stacks,TP,H,ListProd),
                reverse(ListProd,RevList),
                concat([Player,H],ListProd, TempList),
                concat([Player,H],RevList, TempListRev),
                concat([TempList],[TempListRev],Temp),
                concat(Temp,SListMoves,ListMoves).
```

La règle *all_ possible_ moves* appelle le prédicat *possible_ moves* pour tous les déplacements possibles. *possible_ moves* va alors retourner toutes les combinaisons de produits gardés et de produits vendus possibles.

Le prédicat *all_ possible_ moves* va, à la fin, retourner la liste des coups possibles afin qu'ensuite l'Intelligence Artificelle puisse déterminer le meilleur coup possible.

2.3.2 Déterminer le meilleur coup

C'est ici que l'algorithme Min-Max entre en scéne. Le Minimax va parcourir l'arbre des états simulés en jouant les coups possibles précédement trouvés par la règle précédente. Un problème que nous avons rencontré concernant cette méthode est que l'algorithme ne peut pas descendre plus profond que 2 durant une partie entre deux Intelligences Artificelles sinon le système retournera une exception de stack overflow.

Nous avons utilisé l'élagage Alpha/Beta afin d'améliorer l'éfficacité de la recherche du meilleur coup possible. En ce qui concerne notre fonction d'évaluation des états, elle se base sur le gain potentiel des deux joueurs. Le gain du joueur en train de jouer sera compté positivement alors que le gain de l'adversaire sera compté négativement par la fonction d'évaluation.

Le nombre de prédicats servant à l'implémentation de Minimax étant trop grand ils ne seront pas affichés dans ce fichier. Vous pourrez les trouver dans le fichier *game_ ai.pl* de notre projet.

3 Conclusion

Nous avons bien réussi à implémenter une version Prolog du jeu Chicago Exchange. Le jeu est complet, c'est à dire qu'il se déroule correctement et répond au cahier des charges :

— Partie Humain vs Humain
— Partie IA vs Humain
— Partie IA vs IA

Prolog était un language trés adapté pour ce probléme. Cela nous a permis d'implémenter un jeu en "seulement" quelques lignes de codes qui, je suis sûr aurait eu besoin de plusieurs de plus de lignes si eût été codé dans un langage tel que C.
Notre Intelligence Artificielle a été certainement la partie la plus dure à coder mais nous avons tout de même réussi à obtenir quelque chose de satisfaisant. Nous avons tout deux joué une partie contre cette Intelligence Artificielle et celle-ci nous a battue. Cependant nous pourrions émettre des objections concernant le choix de l'algorithme Minimax. En effet cet algorithme parcours un arbre composé d'états de jeu simulés par l'ordinateur, c'est-à-dire que l'Intelligence va connaitre à l'avance les produits situés en-dessous des jetons placés au-dessus des piles. C'est une connaissance que le joueur humain n'a pas et ne peut acquérir. En bref, l'ordinateur triche en regardant le contenu des piles.
On pourrait alors modifier notre Intelligence afin de prendre en compte les pièces jouées et faire des statistiques quant aux pièces restant en jeu. L'ordinateur prendrait alors des décisions en fonction des probabilités de trouver une certaine marchandise en dessous des pièces manipulées. Une autre possibilité serait d'utiliser une heuristique afin de prévoir les états de jeu.