
**Université de Technologie de Compiègne**

**IA02**

## Rapport de TP

# Prolog Stock Exchange

**Printemps 2015**

Damien MARIÉ - Kyâne PICHOU

*16 juin 2015*

# Table des matières

# 1 Introduction

Dans le cadre des TP de l'UV IA02, nous devons implémenter une version Prolog du jeu Chicago Stock Exchange, un jeu de plateau sur le thème de la bourse et de la finance. A chaque tour, les joueurs déplacent un pion « trader » et récoltent 2 marchandises. Ils en gardent une et jettent l'autre, faisant évoluer la valeur des ces marchandises. Le but étant d'avoir le stock de marchandises ayant le plus de valeur à la fin du jeu.

## 1.1 Structures de données

Les différentes structures de données qui composent ce jeu seront implémentées sous forme de listes.

**Bourse** La bourse est simplement une liste contenant 6 sous-listes (une par élément) formant un couple [marchandise,valeur].

**Marchandises** Le jeu est composé de 9 piles de 4 marchandises (à l'état initial). Chaque pile forme une sous-liste de 4 éléments et ces sous-listes composent une grande liste représentant l'état de toutes les piles du jeu.

**Trader** Le trader a une position qui est simplement représentée par le numéro de la pile où il se trouve.

**Réserve** Le joueur accumule des ressources tout au long de la partie. Chaque jouer a donc une liste dédié contenant toutes les marchandies qu'il possède.

**Coup de jeu** Lorsqu'un joueur joue, plusieurs paramètres sont à prendre en compte. Ces paramètres sont placés dans une liste contenant :

- le joueur qui effectue l'action
- la valeur du déplacement effectué (1,2 ou 3)
- la ressource qui est conservée
- la ressource qui est jetée

# 2 Principaux prédicats

## 2.1 Gestion du plateau

### 2.1.1 Affichage

### 2.1.2 Génération du plateau

En début de partie il faut générer les différentes structures qui composent un plateau. On crée donc un prédicat retournant un nouveau plateau.
Le prédicat va générer l'ensemble des piles en formant une liste, contenant des sous-listes de 4 éléments. Pour générer ces sous-listes, on tire un nombre au hasard, correspondant à l'indice d'un élément dans une grande liste contenant tout les jetons du jeu.

```
take_random(All,El,NewAll):-
       random_el(All,N),
       nth(N,All,El),
       select(El,All,NewAll)
```

Ensuite on génère la bourse, qui est toujours la même à l'état initial. On récupère donc simplement une liste hardcodée correspondant à l'état initial de la bourse.
De plus on génère un chiffre entre 1 et 9 correspondant à la position du trader.

## 2.2 Déroulement du jeu

### 2.2.1 Vérification de coup

Il est important de vérifier qu'un coup respecte les règles du jeu lorsqu'il est executé. Pour cela on utilise un prédicat *coup_possible* qui va prendre en entrée un plateau de jeu et qui va vérifier que le coup proposé est valide.
On commence par tester que la valeur du déplacement est bien de 1, 2 ou 3. Puis on effectue réellement le déplacement afin de récupérer la nouvelle position du trader. A partir de la nouvelle position du trader, on en déduit les éléments se trouvant sur le dessus des piles adjacentes. Enfin, on vérifie que les éléments à jeter et garder correspondent bien à ceux présents sur le haut des piles.
Concernant le déplacement du trader, on additionne simplement sa position avec le valeur du déplacement. On prend cependant garde à ne pas dépasser le nombre de pile, si c'est la cas alors on effectue un bouclage pour revenir au début du plateau.
Pour la vérification des éléments, on appelle un prédicat *choix* qui va retourner les 2 éléments disponibles en fonction de la position du trader. Ce prédicat récupère le nombre de pile puis la position des piles autour du trader. Puis il récupère simplement les éléments en haut des piles corespondantes.

## 2.3 Intelligence artificielle

Nous avons programmé plusieurs intelligences artificielles afin de pouvoir avoir un benchmark de celles-ci :

— Une intelligence purement aléatoire, qui choisit un coup aléatoire parmis les coups possibles (ai_random)

— Une intelligence qui regarde à un seul niveau les scores possibles (ai_simple_best)

— Une intelligence qui utilise l'algorithme minimax (ai_minimax)

Pour l'intelligence minimax, le seul problème est que la profondeur est limitée à 3 si l'on ne veut pas trop avoir de stackoverflow et à 2 si l'on veut pouvoir faire une partie complète. Nous avons aussi programmé des combats entre ces IA afin de tester leurs performances, voici les résultats de pourcentages de vainqueurs selon les IAs :

```
Random VS Random
j1 :73.5%
j2 :20.0%
j1 et j2 :6.5%

Random vs Simple best
j2 :94.0%
j1 :4.5%
j1 et j2 :1.5%

Simple best vs Random
j1 :97.5%
j1 et j2 :2.0%
j2 :0.5%

Simple best vs Simple best
j1 :67.0%
j2 :27.5%
j1 et j2 :5.5%
```

On voit bien qu'une intelligence de profondeur 1 est déja trés efficace par rapport à un jeu aléatoire. On peut donc facilement imaginer les gains d'une IA de profondeur supérieure.

On remarque que le joueur 1 à un avantage stratégique évident (70% de chance de gagner à coups alétoires, 67% dans le cas de coups moins aléatoires).

Voici ci-dessous l'essentiel du code de l'algorithme minimax.

```
%Pour un état donné, retourne le meilleur coup minimax
%State = [Plateau, CoupJoue, Joueur, Profondeur]
minimax([Plateau, CoupJoue, Joueur, Profondeur], BestNextState, Score) :-
    Profondeur >= 1, %Ici, Profondeur MAX est défini
    m_next_states([Plateau, CoupJoue, Joueur, Profondeur], NextStates),
    m_simple_best(NextStates, BestNextState, Score), !
.
minimax(State, BestNextState, Score) :-
    m_next_states(State, NextStates),
    m_best(NextStates, BestNextState, Score), !
.
minimax(State, _, Score) :-
    m_score_state(State, Score)
.

%applique un coup sur un état et renvoi le nouvel état
m_apply_coup([Plateau, _, Joueur, Profondeur], [_,Move,Keep,Drop],
```

```
        [NewPlateau, [_,Move,Keep,Drop], NewJoueur, NewProfondeur]):-
        jouer_coup(Plateau, [Joueur,Move,Keep,Drop], NewPlateau),
        joueur_suivant(Joueur, NewJoueur),
        NewProfondeur is Profondeur+1
.

%renvoi les nouveaux états possibles
m_next_states([Plateau, _, Joueur, Profondeur], [AP,BP,CP,DP,EP,FP]):-
    coups_possibles(Plateau, [A,B,C,D,E,F]),
    m_apply_coup([Plateau, _, Joueur, Profondeur], A, AP),
    m_apply_coup([Plateau, _, Joueur, Profondeur], B, BP),
    ....
.

%renvoi vrai si le score du joueur est àminimiser
m_min_to_move([_,_,_,P]):-
    Pm is P mod 2,
    Pm is 1
.

%retourne le score de l'état
m_score_state([Plateau, _, Joueur, _], Score):-
    score_joueur(Joueur, Plateau, Score1),
    joueur_suivant(Joueur,Joueur2),
    score_joueur(Joueur2, Plateau, Score2),
    Score is Score1-Score2
.

%retourne le meilleur état maximisant le score et le score associé en appelant r
    écursivement minimax
m_best([State], State, Score) :-
    minimax(State, _, Score), !
.
m_best([State1 | States], BestState, BestScore) :-
    minimax(State1, _, Score1),
    m_best(States, State2, Score2),
    m_better_of_two(State1, Score1, State2, Score2, BestState, BestScore)
.

%retourne le meilleur état sans appel récursif
m_simple_best([State], State, Score) :-
    m_score_state(State, Score), !
.
m_simple_best([State1 | States], BestState, BestScore) :-
    m_score_state(State1, Score1),
    m_simple_best(States, State2, Score2),
    m_better_of_two(State1, Score1, State2, Score2, BestState, BestScore)
.
```

# 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. L'implémentation *gprolog* n'est peut être pas assez évoluée, outillée et documentée pour permettre un dévelopement confortable mais celà ne nous à pas empécher à développer des fonctionnalités avancées.
Entre autre la gestion des marchandises, de la bourse, des réserves et de la position du trader sont implémentés, fonctionnels et sans failles. Notre intelligence artificielle reste cependant rudimentaire, c'est à dire qu'elle n'implémente pas l'algorithme *minimax* et ne fait ses calculs que en fonction de l'état courant du plateau. Dans un sens, cette technique est la plus logique étant donné que les joueurs ne peuvent savoir que la marchandise se trouvant sur le dessus de la pile, regarder en dessus et donc calculer les prochains états du jeu relève donc de la triche.
Néanmoins nous avons essayé d'implémenter l'algorithme *minimax*, mais celui-ci ne fonctionne qu'avec une profondeur de 1. Notre implémentation n'était donc pas opérationnelle, nous avons conservé notre IA basique.