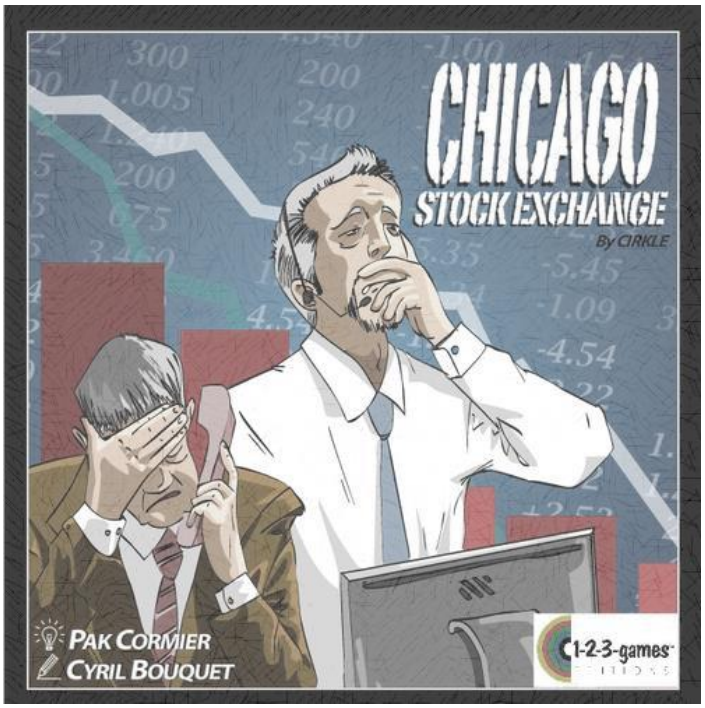
Projet Prolog

Chicago Stock Exchange



Florian GUYET & Charlotte HAAG

Introduction

Dans le cadre de l'UV IA02, il nous a été demandé de concevoir un jeu de société sur le thème de la bourse et de la finance, appelé Chicago Stock Exchange. Nous avons réalisé ce projet en Prolog de façon à permettre une partie « Humain VS Humain », « Humain VS Machine » et « Machine VS Machine », tout en essayant de respecter au mieux les règles du jeu énoncées.

Dans un premier temps, nous nous sommes occupés de l'affichage et de la création du plateau du jeu, tout en essayant d'implémenter une interface de jeu la plus claire et ergonomique. Le plateau utilisé durant le déroulement du jeu est donc constitué des marchandises, de la bourse, de la position du trader ainsi que des réserves des deux joueurs.

Nous avons par la suite implémenté le jeu humain, c'est-à-dire permettant à deux joueurs de s'affronter en suivant les différentes règles du jeu.

Pour finir nous avons créé la partie contenant l'intelligence artificielle qui permet d'effectuer des parties « Humain VS Machine » ou « Machine VS Machine ».

Dans un premier temps nous allons vous présenter les principaux prédicats que nous avons implémentés dans notre projet. Nous allons ensuite vous décrire l'IA ainsi que les choix faits lors de l'implémentation de l'IA et de l'évaluation des coups. Pour finir nous exposerons les différents problèmes que nous avons rencontrés durant l'élaboration de notre code.

I) Présentation des principaux prédicats

a) Affichage et création d'un plateau de jeu

Tout d'abord nous avons implémenté les prédicats permettant de générer de façon aléatoire un plateau et de pouvoir par la suite l'afficher.

```
plateau_depart([Marchandises, Bourse, PositionTrader, ReserveJ1, ReserveJ2]) :-
                                                    bourse(Bourse),
                                                    marchandises_depart(Marchandises),
                                                    random(1,9,PositionTrader),
                                                    reserveJoueur1(ReserveJ1),
                                                    reserveJoueur2(ReserveJ2).
```

Dans le prédicat ci-dessus, nous appelons les différents prédicats permettant de créer le plateau de départ. En effet ce plateau est une liste contenant les marchandises, la bourse, la position du trader ainsi que les réserve des joueurs. La bourse au départ du jeu est la liste suivante :

```
bourse([[ble,7],[riz,6],[cacao,6],[cafe,6],[sucre,6],[mais,6]]).
```

Le prédicat *marchandises_depart(Marchandises)* permet de former aléatoirement la liste des marchandises.

```
marchandises_depart(NewMarchandises) :- liste_marchandises_dispo(Liste),
                                        melanger_liste(Liste, [], Marchandises),
                                        creer_sous_liste(Marchandises, [], NewMarchandises).
```

Nous prenons une liste contenant tous les éléments (blé, mais, riz, sucre, cafe, cacao) de la forme suivante :

```
liste_marchandises_dispo( [ ble, ble, ble, ble, ble, ble, mais, mais, mais, mais, mais, mais,
                            riz, riz, riz, riz, riz, riz, sucre, sucre, sucre, sucre, sucre, sucre,
                              cafe, cafe, cafe, cafe, cafe, cafe, cacao, cacao, cacao, cacao, cacao,  cacao]).
```

Nous mélangeons ensuite cette liste grâce au prédicat *melanger_liste(Liste, [], Marchandises)* qui prend la liste et en créer une nouvelle mais avec les éléments mélangés. Le prédicat *creer_sous_liste(Marchandises, [], NewMarchandises)* permet ensuite de séparer cette liste mélangée, appelée *Marchandises*, en 9 sous-listes de 4 éléments.

La position du trader est générée de façon aléatoire grâce à un random.

Les deux prédicats *reserveJoueur1(ReserveJ1)* et *reserveJoueur2(ReserveJ2)* renvoient la liste correspondant à la réserve du joueur correspondant, qui est au départ est une liste vide.

L'autre prédicat qu'il a été nécessaire d'implémenter est celui gérant l'affichage du plateau au cours de la partie. Nous avons essayé d'afficher de façon claire tous les éléments permettant aux joueurs de visualiser facilement l'état du plateau au cours de la partie.

affiche_plateau([Marchandises, Bourse, PositionTrader, ReserveJ1, ReserveJ2]):-
affiche_bourse(Bourse),
affiche_marchandises(Marchandises),
affiche_position(PositionTrader),
affiche_joueur1(ReserveJ1),
affiche_score(j1,[Marchandises, Bourse, PositionTrader, ReserveJ1, ReserveJ2]),
affiche_joueur2(ReserveJ2),
affiche_score(j2,[Marchandises, Bourse, PositionTrader, ReserveJ1, ReserveJ2]).

Dans ce prédicat, nous passons donc en paramètre le plateau que l'on souhaite afficher. Ce prédicat fait appel à plusieurs autres prédicats, chacun chargé de l'affichage d'une partie du plateau. Ainsi *affiche_bourse(Bourse)* permet l'affichage de la bourse sous la forme suivante :

```
Voici la bourse :
-----------------
ble :   7
riz :   6
cacao : 6
cafe :  6
sucre : 6
mais :  6
```

Le prédicat *affiche_marchandises(Marchandises)* permet d'afficher les marchandises du plateau et *affiche_position(PositionTrader)* positionne le trader en dessous de la pile sur laquelle il se situe, pour plus de visibilité.

```
Voici les cartes au sommet de chaque pile :
-------------------------------------------
|cafe   |cacao  |cacao  |ble    |mais   |ble    |cafe   |cacao  |cacao  |
                                              T

La lettre T represente la position du Trader (position=7)
```

Les deux prédicats *affiche_joueur1(ReserveJ1)* et *affiche_joueur2(ReserveJ2)* affichent les réserves des deux joueurs. Grâce à *affiche_score(j1, Plateau)* et *affiche_score(j2,Plateau)* on affiche également en dessous de chaque réserve le score de chaque joueur, calculé en fonction des produits qu'il possède et leur valeur dans la bourse.

```
Voici la reserve du Joueur1 :
-----------------------------
ble mais
Score = 13

Voici la reserve du Joueur2 :
-----------------------------
mais
Score = 6
```

b) Jeu humain

Après avoir mis en place les différents éléments du plateau, nous nous sommes penchés sur comment permettre à un joueur de réaliser un coup, qui au préalable devra être vérifié afin de déterminer si ce coup est possible ou non.

```
jouer_coup(Plateau, Coup, NewPlateau) :-
        \+coup_possible(Plateau,Coup),
        write('\n\n/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\ \n'),
        write('/!\\ATTENTION : Le coup n\'est pas possible /!\\ \n'),
        write('/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\/!\\ \n'),
        changer_joueur,  NewPlateau = Plateau, !.


jouer_coup([Marchandises, Bourse, PositionTrader, ReserveJ1, ReserveJ2], [Joueur,Deplacement,Garder,Jeter],
        [NewMarchandises, NewBourse, NewPositionTrader, NewReserveJ1, NewReserveJ2]) :-
            coup_possible(    [Marchandises, Bourse, PositionTrader, ReserveJ1, ReserveJ2],
            [Joueur,Deplacement,Garder,Jeter]),
            write('Le coup est possible\n'),
            length(Marchandises,LongueurM),
            changer_position(PositionTrader, Deplacement, LongueurM, NewPositionTrader),
            write('New pos='), write(NewPositionTrader), nl,
            changer_bourse(Bourse,Jeter,NewBourse),
            ajouter_reserve(Joueur, Garder, ReserveJ1, ReserveJ2, NewReserveJ1, NewReserveJ2),
            changer_marchandises(Marchandises, NewPositionTrader,NewMarchandises),!.
```

Le prédicat *jouer_coup(Plateau, Coup, NewPlateau)* est assez complexe dans la mesure où il fait appel à d'autres prédicats importants pour la vérification du coup et pour la modification du plateau après un coup. Tout d'abord ce prédicat vérifie si le coup est possible ou non grâce au prédicat *coup_possible(Plateau,Coup)*, que nous présenterons par la suite. Si le coup n'est pas possible alors on revient au joueur qui vient de jouer afin qu'il puisse retenter un coup et on renvoie le plateau initial. Si le coup est possible alors on change la position du trader grâce au prédicat *changer_position(PositionTrader, Deplacement, LongueurM, NewPositionTrader)* qui prend en paramètres la position actuelle du trader avant le coup, le déplacement désiré par le joueur (1, 2 ou 3), le nombre de piles restantes dans la liste de marchandises et renvoie la nouvelle position du trader. On modifie ensuite la bourse grâce au prédicat *changer_bourse(Bourse,Jeter,NewBourse)* qui prend en paramètres la bourse à modifier, le produit que le joueur a décidé de jeter (et dont la valeur va donc être décrémentée dans la bourse) et renvoie la nouvelle bourse. On ajout ensuite le produit gardé par le joueur dans sa réserve via le prédicat *ajouter_reserve(Joueur, Garder, ReserveJ1, ReserveJ2, NewReserveJ1, NewReserveJ2)* qui gère la réserve de chacun des deux joueurs. On fait ensuite appel au prédicat *changer_marchandises(Marchandises, NewPositionTrader,NewMarchandises)* qui permet de modifier la liste des marchandises.

Nous allons vous présenter deux des prédicats utilisés ci-dessus qui ressortent par leur importance : *coup_possible(Plateau,Coup)* et *changer_marchandises(Marchandises, NewPositionTrader,NewMarchandises)*.

```
coup_possible([Marchandises, _, PositionTrader, _,_], [_,Deplacement,Garder,Jeter]) :-
        length(Marchandises, LongueurM),
        changer_position(PositionTrader, Deplacement, LongueurM, NewPosition ),
        recuperer_droite(NewPosition, Marchandises,LongueurM,ProduitDroite),
        recuperer_gauche(NewPosition, Marchandises, LongueurM, ProduitGauche),
        produits_a_cote(Garder, Jeter, ProduitGauche, ProduitDroite).
```

Le prédicat ci-dessus a pour but de vérifier si le coup souhaité par le joueur est possible ou non. Il prend en arguments le plateau ainsi que le coup. Après avoir récupéré la nouvelle position du trader après le déplacement, on récupère le produit situé en haut de la pile situé à droite de la nouvelle position du trader, ainsi que le produit situé en haut de la pile de gauche. On vérifie ensuite si ces deux produits correspondent à ceux que le joueur veut garder ou jeter.

```
changer_marchandises(Marchandises, NewPosition, NewMarchandises) :-
        length(Marchandises, LongueurM),
        position_droite(NewPosition, LongueurM, PositionPileD),
        position_gauche(NewPosition, LongueurM, PositionPileG),
        nth(PositionPileG, Marchandises, RecupGauche),
        supprimer_tete_liste(RecupGauche, NewRecupGauche),
        nth(PositionPileD, Marchandises, RecupDroite),
        supprimer_tete_liste(RecupDroite, NewRecupDroite),
        construct_marchandises(Marchandises, NewRecupGauche,
        NewRecupDroite, NewPosition, NewMarchandises).
```

Dans le prédicat ci-dessus, on modifie la liste des marchandises en retirant les deux produits que le joueur a gardés ou jetés. Pour cela on récupère les piles situées à droite et à gauche de la position du trader et on retire la tête de ces piles (qui sont en fait des sous-listes de la liste des marchandises). On reconstruit ensuite grâce au prédicat *construct_marchandises(Marchandises, NewRecupGauche, NewRecupDroite, NewPosition, NewMarchandises)* la nouvelle liste de marchandises avec les deux nouvelles piles obtenues. Ce prédicat prend en paramètres la liste des marchandises avant modification, les deux nouvelles piles auxquelles on a retiré le produit du dessus, la nouvelle position du trader et cela renvoie la nouvelle liste de marchandises. Nous avons dû gérer différents cas : le cas où la position du trader était comprise entre 2 et 8 et deux autres cas où la position du trader était à 1 ou à 9. Ces deux derniers ont été compliqués à réaliser dans la mesure où les nouvelles piles à récupérer étaient dans des dispositions différentes par rapport au cas général. En effet si la position du trader est 1 alors la pile de gauche à modifier est donc en position 9 (la dernière pile de la liste des marchandises). Si la position du trader est à 9 alors la nouvelle pile droite se trouve en position 1 (c'est-à-dire la première pile de la liste des marchandises).

c) Implémentation de l'interface

Afin de réaliser une véritable interface de jeu, nous avons implémenté plusieurs prédicats ayant pour rôle d'implémenter l'interface du jeu. Tout d'abord il nous avait été demandé de pouvoir lancer 3 types de parties de jeu : une partie « Humain VS Humain », une partie « Humain VS Machine » et une partie « Machine VS Machine ». Lors du lancement du jeu on a donc le lancement d'une nouvelle partie dont on peut choisir le type via l'interface.

```
jeu :- retract(joueurEnCours(_)), jeu,!.

jeu :- nl, nl, write('NOUVELLE PARTIE !'),
       nl, nl,demander_type_jeu(Choix),
      executer_jeu(Choix).
```

Cela nous donne par exemple, dans le cas où l'on sélectionne le jeu « Humain VS Humain » :

```
NOUVELLE PARTIE !


Quel mode voulez vous executer ?

1 - Humain vs Humain.
2 - Humain vs Machine.
3 - Machine vs Machine.
1.

Jeu Humain vs Humain selectionne
```

En fonction du type sélectionné cela lance différent prédicats : *jeuHH* pour une partie « Humain VS Humain », *jeuHM* pour une partie « Humain VS Machine » et *jeuMM* pour une partie « Machine VS Machine ». Par exemple, voici ci-dessous le prédicat chargé de lancer et de gérer une partie entre deux joueurs humains :

```
jeuHH :- nl, nl, write('Jeu Humain vs Humain selectionne'), nl, nl,asserta(joueurEnCours(j2)), plateau_depart(P),
boucle_jeuHH(P).

boucle_jeuHH([Marchandises,Bourse,PositionTrader,R1,R2]) :-
        length(Marchandises, LongueurM), LongueurM < 3,nl,
         affiche_marchandises(Marchandises),nl,
        write('LE JEU EST TERMINE'), affiche_gagnant([Marchandises,Bourse,PositionTrader,R1,R2]),
        retract(joueurEnCours(_)),!.

boucle_jeuHH(Plateau) :-
        affiche_plateau(Plateau), changer_joueur,
        demander_coup(Coup,Plateau), jouer_coup(Plateau, Coup, NewPlateau), boucle_jeuHH(NewPlateau).
```

Le jeu se termine lorsqu'il ne reste plus que deux piles de jetons dans la liste des marchandises. Le prédicat *affiche_gagnant(PlateauFin)* affiche le gagnant de la partie en calculant et comparant les scores des deux joueurs.

Dans le cas où c'est la machine qui doit jouer le coup, il n'y a plus d'interaction avec l'utilisateur. Par contre nous avons dû mettre en place divers prédicats pour que l'utilisateur puisse jouer un coup, notamment pour qu'il puisse choisir le déplacement du trader, ou les produits qu'il veut garder ou jeter.

```
demander_coup([Joueur,Deplacement,Garder,Jeter], [Marchandises, _, PositionTrader, _, _]) :-
        joueurEnCours(Joueur),
        write('\nC\'est au tour de '), write(Joueur), write(' de jouer !'), nl,
        demander_deplacement(Deplacement),
        length(Marchandises, LongueurM),
        changer_position(PositionTrader, Deplacement, LongueurM, NewPosition),
        affiche_marchandises_top(Marchandises),
        affiche_position(NewPosition),
        demander_garder(Garder),
        demander_jeter(Jeter).
```

Le prédicat *demander_coup(Coup, Plateau)* permet de récupérer le déplacement que le joueur souhaite faire via la console. Des vérifications sont faites afin de faire en sorte que le programme soit robuste aux erreurs de saisies courantes. Ainsi le joueur ne peut rentrer que 1, 2 ou 3 pour le déplacement, dans le cas contraire il lui est demandé de recommencer sa saisie. Voici l'affichage obtenu sur l'interface en cas d'erreur de saisie :

```
C'est au tour de j2 de jouer !

Quel deplacement souhaitez vous faire ? 45.
Veuillez rentrer un deplacement egal a 1, 2 ou 3

Quel deplacement souhaitez vous faire ?
```

Les prédicats *demander_garder(Garder)* et *demander_jeter(Jeter)* permettent de récupérer les produits que l'utilisateur veut garder ou jeter. De la même façon les réponses de l'utilisateur sont vérifiées afin qu'aucune erreur de saisie ne vienne perturber le programme.

```
Quel produit voulez vous garder ? caco.
Veuillez rentrer un produit existant,
en minuscule sans accent (ble, riz, cacao, cafe, sucre, mais)

Quel produit voulez vous garder ?
```

II) Description de l'Intelligence Artificielle

Après avoir réalisé un mode de jeu « Humain vs Humain », nous nous sommes attardés sur le développement et l'implémentation de l'Intelligence Artificielle.

Le but de cette partie était de permettre à une personne seule de jouer contre une IA, assez développé pour qu'il existe un réel challenge à battre cette dernière.

a) Recherche des coups possibles et évaluation de ces coups

Nous avons commencé par implémenter le prédicat *coups_possibles(Plateau,ListeCoupsPossibles),* qui permet de récuperer la liste des coups possibles selon un Plateau donné en paramètre .

```
coups_possibles([Marchandises|_],[]) :-     length(Marchandises, LongueurM), LongueurM <3.

coups_possibles(Plateau,ListeCoupsPossibles) :-     coups_par_deplacement(Plateau, 1, CoupsEn1),
                                                    coups_par_deplacement(Plateau, 2, CoupsEn2),
                                                    coups_par_deplacement(Plateau, 3, CoupsEn3),
                                                    append(CoupsEn1, CoupsEn2, CoupsEn12),
                                                    append(CoupsEn12,CoupsEn3, ListeCoupsPossibles).
```

Si le nombre de pile dans Marchandises est inférieur à 3, alors aucun coup n'est possible, et nous récupérons une liste vide. Sinon, le prédicat divise la recherche des coups possibles selon les déplacements réalisables. La liste des coups possibles selon un certain déplacement est récupérée avec le prédicat suivant :

```
coups_par_deplacement([Marchandises,_, PositionTrader, _, _], Deplacement, CoupsDeplacement):-
        length(Marchandises, LongueurM),
        NewPositionTrader is (PositionTrader + Deplacement),
        recuperer_gauche(NewPositionTrader, Marchandises, LongueurM, ProduitGauche),
        recuperer_droite(NewPositionTrader, Marchandises, LongueurM, ProduitDroite),
        joueurEnCours(Joueur),
        Coup1 = [Joueur, Deplacement,ProduitGauche, ProduitDroite],
        Coup2 = [Joueur, Deplacement,ProduitDroite, ProduitGauche],
        append([Coup1],[Coup2], CoupsDeplacement).
```

Après avoir récupéré la liste des coups possibles, afin de décider quel est le meilleur coup, et donc quel sera le coup joué, nous devons évaluer chacun de ces coups. C'est le prédicat *scores_coups_possibles(Plateau,[[Joueur|QueueCoup]|Q], ListeCoupsPossiblesScores)* qui nous permet de faire cela.

```
scores_coups_possibles(Plateau,[[Joueur|QueueCoup]|Q], ListeCoupsPossiblesScores):-
        jouer_coup_machine(Plateau,[Joueur|QueueCoup],NewPlateau),
        calcul_score_joueur(Joueur,NewPlateau, Score),
        append([[Joueur|QueueCoup]], [Score], ScoreCoup),
        scores_coups_possibles(Plateau,Q, ListeCoupsPossiblesScoresQueue),
        append(ListeCoupsPossiblesScoresQueue,[ScoreCoup] ,ListeCoupsPossiblesScores).
```

Le score d'un coup est simplement donné par le score du joueur qui joue le coup, après application de ce dernier sur le plateau en cours. Le score du joueur, quand à lui, est la somme des valeurs des produits qu'il possède dans sa réserve. Enfin, la valeur d'un produit est récupérée dans la bourse du plateau en cours.

b) Choix du meilleur coup : profondeur 1.

Nous avons commencé à définir le meilleur coup par le coup permettant au joueur de gagner le plus de point, juste après l'application de ce coup (profondeur 1).

La liste récupérée par le prédicat *scores_coups_possibles* est de la forme suivante : *[[Coup, Score],[Coup2, Score2], …].* Il est donc facile de récupérer par la suite le meilleur coup (celui dont le score est le plus grand), avec le prédicat *coup_score_max*. On initialise d'abord la boucle récursive avec un Score maximum très petit. Ensuite, nous cherchons un coup qui améliorera ce score, par récursivité :

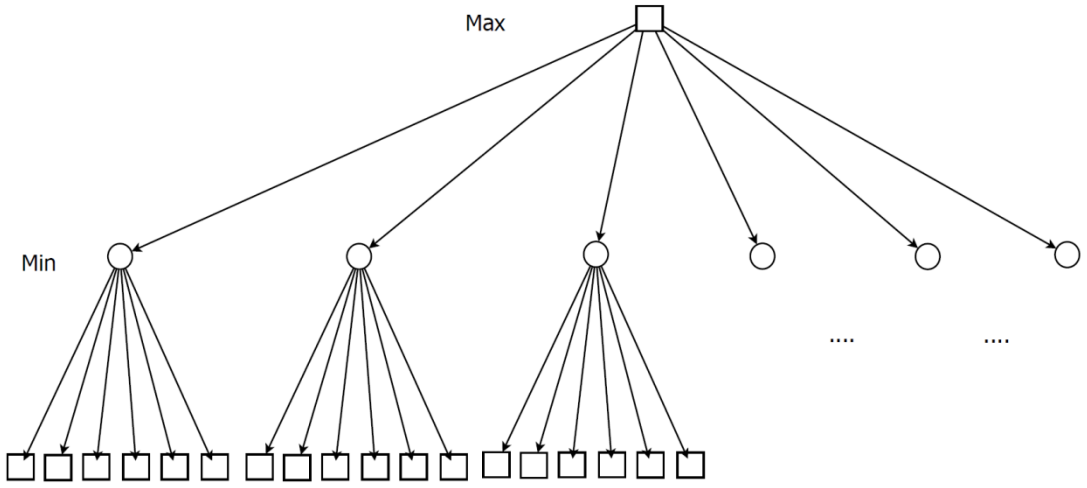
```
coup_score_max(ListeScore,CoupMax):- coup_score_max(ListeScore,[_,-2000000],CoupMax), !.  %initialisation

coup_score_max([], CoupMax, CoupMax):-!.

coup_score_max([[Coup, Score]|Q], [_, ScoreMaxPrecedent], CoupMax) :-
                                        Score > ScoreMaxPrecedent,
                                        coup_score_max(Q,[Coup, Score], CoupMax),!.

coup_score_max([_|Q], CoupScorePrecedent, CoupMax) :-
                                        /*Score <= ScoreMaxPrecedent,*/
                                        coup_score_max(Q,CoupScorePrecedent, CoupMax),!.
```

Le meilleur coup est donc donné par le prédicat suivant :

```
meilleur_coup(Plateau, MeilleurCoup):-
        coups_possibles(Plateau, ListeCoupsPossibles),
        scores_coups_possibles(Plateau,ListeCoupsPossibles, ListeCoupsPossiblesScores),
        coup_score_max(ListeCoupsPossiblesScores,[MeilleurCoup,_]).
```

c) Le minimax : profondeur 2 et 4.

Afin d'amélioré notre IA, nous avons implémenté l'algorithme du minimax. Comme expliqué dans le cours de IA02, celui-ci a pour but de minimiser le score de l'adversaire sous l'hypothèse que ce dernier jouera parfaitement.



Voici l'algorithme utilisé dans le prédicat minimax1 (minimax de profondeur 2) :

```
minimax1([Marchandises|Reste], MeilleurCoup):-
        length(Marchandises, LongueurM), LongueurM < 5,
        meilleur_coup([Marchandises|Reste], MeilleurCoup).

minimax1(Plateau, MeilleurCoup,ScoreMax):-
        coups_possibles(Plateau, ListeCoupsPossibles),
        max_min_opposant(Plateau, ListeCoupsPossibles, [MeilleurCoup,ScoreMax]).
```

Si le nombre de piles dans Marchandise est inférieur à 5, il se peut qu'il y ait moins de 3 piles en un seul coup (on retire deux produits dans des piles ne contenant qu'un produit) et le jeu doit s'arrêter. Afin d'éviter que l'IA ne réfléchisse sur des coups qu'il ne sera pas autorisé à faire, lorsqu'il y a moins de 5 piles dans marchandises, minimax1 récupère le MeilleurCoup avec le prédicat *meilleur_coup* (profondeur 1) expliqué précédemment.

Sinon, le coup renvoyé est celui choisi par application du minimax, de profondeur 2 : nous recherchons le score maximum du score minimum que l'opposant pourra effectuer.

```
max_min_opposant(Plateau, ListeCoupsPossibles, [MeilleurCoup, ScoreMin]):-
                max_min_opposant(Plateau, ListeCoupsPossibles, [_,-30000], [MeilleurCoup, ScoreMin]),!.

max_min_opposant(       Plateau,[[_,Deplacement, Garder, Jeter]|Q] ,
                        [_, ScoreMinPrecedent], [MeilleurCoup, ScoreMin]):-

                        joueurEnCours(Joueur),
                        jouer_coup_machine(Plateau, [Joueur,Deplacement, Garder, Jeter], NewPlateau),
                        changer_joueur,
                        min_score_opposant( NewPlateau ,ScoreMinCoup),
                        changer_joueur,
                        ScoreMinCoup > ScoreMinPrecedent,
                        max_min_opposant(       Plateau, Q, [[Joueur,Deplacement, Garder, Jeter],
                                                ScoreMinCoup], [MeilleurCoup, ScoreMin]),!.


max_min_opposant(       Plateau,[[_,Deplacement, Garder, Jeter]|Q] ,[MeilleurCoupPrecedent,
                        ScoreMinPrecedent], [MeilleurCoup, ScoreMin]):-

                        joueurEnCours(Joueur),
                        jouer_coup_machine(Plateau, [Joueur,Deplacement, Garder, Jeter], NewPlateau),
                        changer_joueur,
                        min_score_opposant( NewPlateau ,_),
                        changer_joueur,
                        /*ScoreMinCoup <= ScoreMinPrecedent, */
                        max_min_opposant(       Plateau, Q, [MeilleurCoupPrecedent, ScoreMinPrecedent],
                                                [MeilleurCoup, ScoreMin]),!.
```

L'explication du la recherche de la valeur maximale est analogue à la recherche du coup de valeur maximale. Sauf que dans ce cas, nous récupérons précédemment le score minimum que pourra avoir l'opposant, avec le prédicat min_score_opposant.

Nous avons aussi développé un prédicat minimax2 qui permet de faire cette recherche sur 4 coups successifs. Son fonctionnement ressemble au minimax1, avec une profondeur et un temps de recherche plus important.

III) Difficultés rencontrées et améliorations possibles

Lors de l'implémentation de ce projet nous avons rencontré plusieurs difficultés. Tout d'abord nous avons eu du mal à appréhender au départ la façon dont nous devions nous y prendre pour réaliser ce projet : n'ayant jamais fait de prolog avant l'UV de IA02, ce type de langage entièrement récursif nous a posé au départ quelques problèmes, mais nous nous sommes rapidement familiarisés avec ce langage grâce à l'implémentation de ce projet. Nous avons dû cependant revenir sur certains prédicats car nous nous sommes rendus compte par la suite que nous n'exploitions pas complètement les propriétés récursives du Prolog.

D'autres prédicats ne marchaient pas pour une raison purement algorithmique ou syntaxique comme par exemple les prédicats chargés de calculer les position du trader ou des piles de jetons situés à droite et à gauche du trader. Au début de l'implémentation du code, nous avions également généré un tableau de départ statique. Nous avons donc dû revenir dessus afin de trouver un moyen de le générer de façon aléatoire.

Le prédicat *construct_marchandises(Marchandises, NewRecupGauche, NewRecupDroite, NewPosition, NewMarchandises)* nous a aussi posé quelques problèmes en raison des différents cas à traiter (position du trader sur la première pile de jetons ou la dernière de la liste marchandises).

La partie du projet la plus difficile à implémenter a été la partie IA. En effet nous n'avions pas bien compris la logique de l'algorithme du Minimax, et nous avions commencé à faire une IA en essayant de nous inspirer du Minimax α/β, mais nous nous sommes rendu compte que la tâche était trop ardue, et nous avons donc recommencé afin de faire un algorithme Minimax.

Si nous pouvions apporter une amélioration à notre projet ce serait justement de mettre en place un algorithme Minimax α/β afin de rendre notre code plus performant. Il faudrait aussi modifier l'algorithme du Minimax pour qu'il ne cherche pas seulement à minimiser les coups de l'opposant, mais aussi à maximiser ses propres coups.

L'implémentation de ce projet en Prolog nous a permis de voir comment fonctionne ce langage en créant un jeu à la fois concret et ludique. La partie du projet liée à la mise en place de l'IA était très intéressante dans la mesure où nous avons testé différents types d'intelligence artificielle, mais aussi c'était aussi la partie la plus difficile à réaliser.