IA02 : Logique et Résolution de problèmes par la recherche

CHIDIAC Kevin

DE MAGONDEAUX Estelle

SLAM Zineb

P2015

Sommaire

1. Introduction

Dans le cadre de l'UV IA02, il nous est demandé d'implémenter le jeu Chicago Stock Exchange. Il s'agit d'un jeu de société sur le thème de la bourse basé sur un principe : « Tout ce qui est rare est cher !» Les règles sont assez simples et le score de chaque joueur dépend des marchandises qu'il garde en stock et de celles vendues sur le marché.

2. Les prédicats

2.1 Les objets du jeu

Le plateau :

Le plateau est le support du jeu : c'est lui qui contient toutes les marchandises, les piles où elles sont réparties, le Trader, la bourse et les réserves de marchandises accumulées par chaque joueur.

Pour le modéliser nous avons donc choisi une représentation sous forme de liste contenant chaque pièce du jeu : [Bourse, Marchandises, PositionTrader, ReserveJ1, ReserveJ2].

Où la bourse représente la liste des différentes marchandises et leur valeur actuelle sous la forme d'une liste où chaque élément est la sous-liste [Marchandise, Valeur]. Les ReserveJX représentent les marchandises que chaque joueur a souhaité garder au long de la partie. Les Marchandises sont toutes les ressources disponibles du jeu, elles représentent les différentes piles qui composent le plateau.

Le coup :

Un coup est une liste dans laquelle les éléments sont les données essentielles pour jouer un coup, à savoir :

- Le joueur qui réalise le coup
- Le déplacement (1, 2 ou 3)
- La marchandise gardée et ajoutée à la réserve
- La marchandise vendue et dont la valeur boursière diminue.

Le Trader

Le Trader est le pion que se partagent les joueurs. Bien qu'en réalité le plateau soit circulaire (permettant de ne pas tenir compte de sa position absolue), nous avons défini sa position comme étant comprise entre 1 et N (où N est le nombre de piles) sachant que N+1 = 1 et que si la N+1-ème pile est vide, N+1 = N+2.

2.2 L'initialisation

Le Trader :

generer_trader(Pos):- random(1,9,Pos).

Ce prédicat permet de placer aléatoirement le trader sur une des neuf piles du plateau.

Les piles :

generer_pile(Marchandise,[Elem|Y],I,Newmarch,Real).

Ce prédicat permet de générer des piles de I élément en récupérant un élément au hasard de Marchandise (Elem) et en rappelant ensuite le prédicat sur la nouvelle liste de marchandises Newmarch.

Les marchandises :

generer_marchandise(Marchandise,[T|Q]).

Ce prédicat utilise **generer_pile** afin de pouvoir créer les différentes piles du plateau. En effet, **generer_pile** crée des piles de I élément, il y a donc N-I éléments dans Marchandise à la fin du prédicat, **generer_marchandise** permet de faire des piles tant qu'il reste des éléments dans la liste Marchandise. L'appel du prédiact se fait en unifiant Marchandise avec une liste ordonnée ou non de toutes les cartes du jeu : 6 de cacao, café, maïs, sucre, riz, et 7 de blé.

Le plateau :

plateau_depart([bourse,Marchandises,1,[],[]]).

Ce prédicat permet d'initialiser le plateau avec la bourse (qui est la même au début de chaque partie), la liste des piles obtenues grâce à **generer_marchandise**, la position du trader définie en 1 (à cause du besoin de pouvoir se situer de manière absolue tout au long de la partie) et les réserves de chaque joueurs, initialement vide.

2.3 L'affichage

Les marchandises :

imprime_liste_liste([T|Q]).

Les marchandises étant représentées par des sous-listes contenant les piles, nous avons choisi d'afficher les piles par colonne. Ce prédicat appelle donc l'affichage d'une pile , **imprime_liste(T)**, qui affiche chaque carte de la pile, en allant à la ligne après chaque fin de pile.

Le plateau :

affiche_plateau([Bourse,Marchandises,PositionTrader,ReserveJ1,ReserveJ2]).

Ce prédicat permet l'affichage des différents états du plateau au fil des tours. En utilisant des « nl », nous obtenons un affichage clair. L'affichage des scores n'est pas géré par ce prédicat.

2.4 Le jeu

Menu d'accueil :

menu.

Pour donner à choisir un des trois modes de jeu, nous avons créé un prédicat de menu, qui ne prend aucun paramètre, et qui permet de lancer le mode de jeu demandé. Le joueur envoie 1, 2 ou 3, correspondant au modes « multijoueur », « joueur VS ordinateur », et « mode automatique ». Si l'utilisateur envoie un mauvais chiffre, l'affichage du menu boucle après avoir affiché « vous avez mal choisi ».

Début du jeu :

start_HH.

start_HM.

start_MVSM.

Ces trois prédicats apellent la generation du plateau initial, et ensuite appellent le début du jeu à partir du plateau généré.

Evolution du jeu :

Jouer([B,M,Pos,R1,R2],1).

A chaque tour, le prédicat jouer affiche les données de jeu à l'instant donné : la bourse, la position du trader, les piles et leurs cartes. Ce prédicat demande aussi au joueur, quand c'est son tour d'entrer la valeur de son déplacement, la marchandise gardée et celle vendue, c'est-à-dire le coup qu'il souhaite effectuer.

L'affiche et les données entrées étant différentes selon le mode de jeu, il est nécessaire d'écrire trois prédicats de jeu.

Les données nécessaires étant maintenant disponibles, le coup peut être joué grâce au orédiact jouer_coup.

Après avoir implémenté le coup, les nouvelles données du plateau sont affichées et on appelle l'adversaire à jouer à son tour.

Jouer le coup :

jouer_coup([B,M,Pos,R1,R2],[J,Deplacement,Gardee,Jetee],
[NewB,NewM,NewPos,NewR1,NewR2]).

- B, NewB : bourse avant et après le coup
- M, NewM : liste des marchandises et des piles avant et après le coup
- R1, R2, NewR1, NewR2 : réserve des joueurs 1 et 2 avant et après le coup
- Pos, NewPos : position du Trader avant et après le coup.

Tout d'abord, il nous faut vérifier si le coup est possible. Une fois le coup validé, on met à jour les piles où une marchandise a été enlevée. On ajoute celle gardée à la réserve du joueur, on enlève celle jetée de la liste des marchandises en décrémentant sa valeur boursière. Ensuite, on met à jour la position du Trader. Le plateau est ensuite également mis à jour : les piles devenues vides sont enlevées du plateau et la position du trader décrémente alors de 1.

Déplacer le Trader :

deplacer_trader(Marchandises,PositionTrader,Deplacement,NewPosition).

On commence tout d'abord par calculer le nombre de piles du plateau en comptant le nombre d'éléments de la liste Marchandises. En effet, Marchandises est une liste contenant les différentes piles du plateau. Ensuite, il suffit de mettre à jour la position après déplacement dans NewPosition en ajoutant la position actuelle (PositionTrader) et le déplacement (Deplacement). Nous prenons en compte le fait que le plateau est circulaire, donc la position du trader revient à la 1ère pile quand il dépasse la N^{ème} pile (avec N piles sur le plateau).

Tester si un coup est possible :

coup_possible([J,Marchandises,PositionTrader,R1,R2],[Joueur,Deplacement,Gardee,Jetee]).

Dans ce prédicat réside une série de test afin de savoir si le coup rentré par le joueur est valide. Tout d'abord nous vérifions que le joueur a été déclaré correctement, puis si le déplacement est bien compris entre 1 et 3. Pour finir, nous regardons la validité des marchandises Gardee et Jetee grâce à **marchandise_valide**, explicité ci-dessus.

Si ces 3 conditions sont correctes, le coup est considéré comme valide.

Tester si le jeton à vendre ou à garder existe :

marchandise_valide(Marchandises,Gardee,Jetee,Pos1,Pos2).

positions_autour_trader(_,PositionTrader,Pos1,Pos2).

Les marchandises qui subissent une modification de statut (vente ou gain) sont les premiers éléments des piles. Nous étudions donc les piles situées juste avant (Pos1) et juste après (Pos2) grâce au prédicat positions_autour_trader.

on peut savoir grâce à la fonction nth(Pos1,Marchandises,Pile_pos1) de prolog (qui récupère le Pos1-ème élément de la liste Marchandises et le place dans Pile_pos1) et nth0 (0,Pile_pos1,Carte) si les marchandises choisies par les joueurs sont valides ou non.

Mettre à jour la bourse des joueurs :

garder(1,Gardee,R1,NewR1,R2,R2).
garder(2,Gardee,R1,NewR1,R2,R2).

Ce prédicat permet la mise à jour de la réserve du joueur en ajoutant la marchandise Gardee à sa réserve. Nous indiquons que seulement la réserve du joueur dont c'est le tour est modifiée.

Mettre à jour le plateau :

ajourner_march (Marchandises,PositionTrader,NewMarchandise,NewPos).

Ce prédicat assure la mise à jour du plateau à la fin de chaque tour. En effet, il est appelé à la fin de **jouer_coup**. Ainsi, la liste des marchandises (sans les éléments gardés et jetés par le joueur) est stockée dans NewMarchandise et la nouvelle position du Trader dans NewPos. Le prochain appel de **jouer_coup** s'effectuera donc sur NewMarchandise et NewPos.

2.5 Evolution du score

La valeur des marchandises :

valeur_March([[Jetee|Valeur]|_],Jetee,Valeur).

La bourse étant une liste dont les éléments sont de cette forme : [Marchandise|Valeur], récupérer la valeur d'une marchandise consiste en déterminer le 2ème élément de la sous-liste qui contient la marchandise recherchée. Ce prédicat permet donc de récupérer pour chaque marchandise la valeur qui lui correspond suite au déroulement du jeu.

Le score du joueur :

score_joueur(Bourse,[X|Y],N).

Ce prédicat prend en paramètre l'état de la bourse entrée en paramètre, afin de calculer la valeur des marchandises, la réserve du joueur et renvoie le score cumulé. Ainsi en appelant ce prédicat récursivement sur toutes les marchandises accumulées par le joueur, on obtient aisément le score du joueur. Le score de chaque joueur est affiché à chaque tour, car les deux scores peuvent être modifiés à chaque tour (et c'est l'intérêt du jeu).

2.6 Fin du jeu

Le jeu s'arrête lorsqu'il ne reste plus que deux piles sur le plateau. Le prédicat **jouer** appelle alors le calcul des scores totaux et affiche le gagnant .

3. L'intelligence artificielle

Pour implémenter une intelligence artificielle, il faut tenir compte de toutes les possibilités afin d'obtenir la solution optimale à un état du jeu. Ici, à chaque tour, il y a 6 possibilité de coups (2 choix de marchandise gardée/jetée par déplacement et 3 déplacements possibles : 1,2 et 3). Il faut alors choisir parmi ces 6 coups lequel est optimal pour l'IA. Ce choix se fait en comparant les scores à la fin de chaque coup, le meilleur coup étant celui qui permet de faire le plus gros gain de score tout en faisant perdre le plus possible à l'adversaire (les valeurs des marchandises fluctuant selon la disponibilité de celles-ci). Le coup optimal est donc celui avec la meilleure valeur Gain + Perte.

3.1 Les prédicats de l'IA

Les coups possibles :

coups_possibles ([Bourse,Marchandises,PositionTrader,R1,R2],Joueur,X,Y,Z).

Devant faire face à plusieurs éventualités dans le déplacement, nous avons choisi d'implémenter un autre prédicat pour faire la liste des coups valides que peut jouer l'IA. On liste donc grâce à « findall » les coups possibles pour chaque déplacement. Les variables X, Y et Z sont alors unifiées avec la liste des 2 coups possibles pour les valeurs de déplacement 1, 2 et 3.

Simulation de jeu :

simulation_jouer([J,Deplacement,Gardee,Jetee],[Bourse,_,_,R1,R2],Pertinence).

Ce prédicat prend en paramètre un coup possible. Il nous permet de tester un coup et d'en récupérer sa pertinence. La pertinence est calculée comme étant la somme du gain de score du joueur (ici l'ordinateur) et de la perte de score de l'adversaire.

Choix du meilleur coup :

meilleur_coup([Bourse,Marchandises,PositionTrader,R1,R2],Joueur,Sol,Coupafaire).

Après avoir récupérer la liste des coups possibles et effectuer une simulation dessus afin d'obtenir la pertinence de chaque coup, ce prédicat nous permet de récupérer le coup ayant la meilleure pertinence.

Après avoir simulé chaque coup possible et calculé sa pertinence, il cherche le coup

Ce prédicat réunit les prédicats précédents pour parvenir à unifier avec Coupafaire le coup le plus pertinant. Il récupère donc la liste des coups possibles dans 3 listes, simule chaque coup de celles-ci, cherche la pertinence maximum, et unifie Coupafaire avec le coup correspondant.

Coupafaire étant de la forme [Joueur,Déplacement,Gardée,Jetée], il est directement réutilisable par le prédicat jouer_coup.

4. Les prédicats utilitaires

- **element(X, L).**

Récupère l'élément X de la liste L

- **nb_element(L, N).**

Récupère le nombre d'éléments de la liste L

- **retirer_element(N,[T|Liste],[T|Newliste]).**

Récupère le N-ième élément de la liste Liste

- **imprime(Liste).**

Affiche à l'écran la liste Liste

- **supprime_first([T|Q], Q, T).**

Récupère le 1er élément d'une liste et renvoie la liste sans cet élément

- **retire(N,[T|Q],[T|Res]).**

Retire le N-ième élément d'une liste et renvoie la liste sans cet élément.

- **recup_nieme(N, Liste, Element).**

Récupère le N-ième élément d'une liste

- **add_element([T|Y],X,NewL).**

Ajoute X en tête de liste et renvoie la nouvelle liste dans NewL

- **remplace(X,Y,L,Res) :**

remplace l'élement X de la Liste L par l'élément Y et renvoie la nouvelle liste dans Res.

- **elem_max(L,X)** :

renvoie le maximum de la Liste L dans la variable X.

- **elem_max(L,X)**

unifie X avec l'élément de la liste L de valeur maximale.

5. Difficultés

- Dans le prédicat **générer_marchandise** on a eu des difficultés au début pour avoir autant d'éléments des marchandises de départ que dans la marchandise des plateaux (c'est à dire ne pas avoir 10 blé au lieu de 7) et donc il fallait, à chaque fois qu'on générait une pile, renvoyer dans la liste de marchandises entrée en paramètre, une nouvelle liste de marchandises disponibles (sans les éléments de la pile générée) jusqu'à ce qu'il ne reste qu'un seul élément.

- Une autre difficulté rencontrée est d'avoir à remettre à jour les marchandises à chaque fois et supprimer les piles vides car comme notre plateau était circulaire il fallait penser à décrémenter la position du trader si la pile supprimée est située avant, de plus il fallait penser au cas ou durant un seul un tour on supprime 2 piles à la fois.

- Un des problèmes rencontrés et la manipulation de données qui évoluent à chaque coup. Les données du plateau doivent être modifiées et sauvegardées à chaque tour, et c'est à partir de ces nouvelles données que les prédicats de jeu doivent s'effectuer.

- Il nous a fallu penser et prendre en compte les données non valides, comme les piles vides qui doivent être supprimées du plateau, la position du trader qui change selon le nombre de piles, et décrémente à chaque pile vide, parfois 2 fois en un seul tour. La prise en compte de toutes les situations à risque du jeu était une tâche difficile, pourtant elle conditionne l'évolution et le fonctionnement du jeu.

- Retranscrire et afficher sur l'interface de jeu les données de chaque tour n'était pas évident, car cela implique de les unifier avec une variable, et ensuite l'afficher de manière compréhensible.

- L'écriture des prédicats d'intelligence artificielle pure était compliquée dans le sens où il nous faut retranscrire en langage Prolog une stratégie de jeu. Comprendre quels paramètres doivent être pris en compte par le prédicat **coups_possibles** et **meilleurs_coups** demande à l'auteur d'analyser sa logique de réflexion. On cherche à traduire tous les éléments conditionnant un « bon » coup, mais on réalise alors que tous

les paramètres additionnés ne produisent pas une heuristique appropriée. Il faut alors trouver une condition englobant toutes nos idées, et pertinente.

- La décomposition de nos prédicats nous a permis de créer et de tester plus facilement les prédicats plus compliqués, et certains sont utilisés par plusieurs prédicats.

6. Améliorations possibles :

6.1 Meilleur coup :

- L'idéal dans le calcul du meilleur coup est de calculer le score pour chaque coup et pour chaque coup on simule les coups que pourra faire l'adversaire et donc en fonction des ces coups on choisira le meilleur coup en étant le score maximisant le score du joueur et minimisant le score du joueur adverse quelque soit son déplacement.

Algorithme :

Pour Déplacement allant de 1 à 3

Trouver tous les coups possibles.

Fin pour .

Pour chaque Coup (6 au total) calculer le score => garder les 3 meilleurs scores : [M_Coup1,M_Coup2,M_Coup3]

Pour chaque CoupX ϵ [M_Coup1, M_Coup2,M_Coup3]

Simuler le jeu avec CoupX

Trouver tous le coup possibles du joueur adverse .
Calculer le Score de Chaque Coup.

Choisir le CoupX qui minimise tous les Coups de l'adverse (c'est à dire que le joueur adverse se déplace par 1 , 2 ou 3 son score restera toujours inférieur et dans le pire des cas égal au score du Joueur)

- De plus pour faciliter le calcul du meilleur coup on pourra à chaque tour trier la liste de bourse en valeur croissante et donc on réduira le nombre de choix de coups en ne simulant que les coups étant susceptibles d'être plus pertinents que le plus pertinent calculé jusque-là.

6.2-Factoriser le code :

- Pour factoriser le code on pourrait remplacer tous les affichages au début de chaque tour par des appels au prédicat **afficher_plateau** , et pour les autres affichages (comme les réserves et score) on pourrait éventuellement penser à faire un prédicat **afficher_tour**.

- Il nous serait aussi avantageux de générer le plateau via le prédicat **generer_plateau**, qui serait appelé lors du prédicat **start**.

7. Conclusion

L'élaboration d'un jeu nous a permis de comprendre le mécanisme de fonctionnement du logiciel Prolog. Il est parfois laborieux de comprendre les erreurs de compilation, il faut alors entrer dans chaque sous-prédicat pour comprendre « la réaction » du logiciel. La prise en compte de l'ordre des règles et des buts, ainsi que les paramètres entrés, sont primordiaux.

Par ailleurs, pouvoir traduire une stratégie de jeu sous langage informatique est très intéressant, Et il est assez surprenant de voir que notre ordinateur obtient un score supérieur au nôtre à tous les coups. On ne peut s'empêcher de penser que celui-ci est intelligent, or nous savons bien que c'est une combinaison de règles qui induisent la réponse de l'ordinateur.