

**IA02 - Projet Prolog**

Chicago Stock Exchange



Sommaire

[Description du projet](#h.e4xsdw94uid1)

1. [**Présentation des principaux prédicats**](#h.mylffvh050wg)
2. [Initialisation et affichage du jeu](#h.z10pn8yhm9l3)
3. [Exécution du jeu](#h.xlnd827u71gs)
4. [**Description de l’IA**](#h.4bhr5cnrreop)
5. [**Difficultés rencontrées et améliorations possibles**](#h.5uc2h54tp1ze)

# 

# 

# Description du projet

L’objectif de ce projet est d’implémenter en Prolog le jeu “Chicago Stock Exchange” de façon à permettre une partie H/H, M/H ou M/M.

Le principe du jeu est le suivant :

Chicago Stock Exchange (CSE) est un jeu de société sur le thème de la bourse et de la finance. Tout ce qui est rare est cher ! A chaque tour, les joueurs en déplaçant le pion « trader» et opèrent une récolte de 2 marchandises. Ils peuvent ensuite opter pour un achat (garder l’une des récoltes) ou une vente (vendre une des récoltes) sur la place boursière. Une vente baisse d’un cran la valeur des marchandises sur le tableau des cours. Il vous faudra donc vous enrichir en appauvrissant l’adversaire. Simple ? Le souci est que l’adversaire en fait autant !

# 

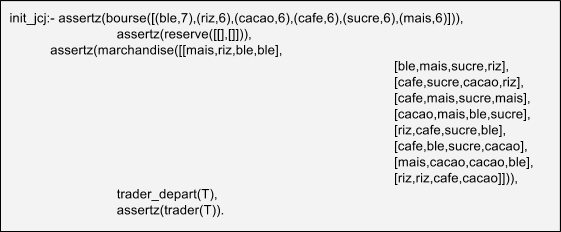
# 

# Présentation des principaux prédicats

Voici la liste non-exhaustive des prédicats que nous avons utilisés dans notre projet.

## Initialisation et affichage du jeu

Nous avons tout d’abord construit un prédicat d’**initialisation** des différentes parties du jeu :



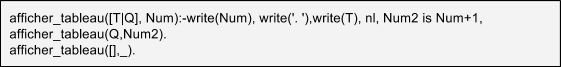
Ce prédicat utilise la clause **assertz** qui permet d’ajouter des clauses au programme de manière dynamique. On va ainsi ajouter la bourse, les marchandises, les réserves des joueurs et la position du trader.

**La bourse** est un tableau de 6 sous-tableaux. Chacun des sous-tableaux est de la forme suivante :

(nom\_marchandise, valeur\_marchandise)

**Les** **marchandises** sont quant à elles stockées dans un tableau de 9 sous-tableaux. Ces sous tableaux représentent les 9 piles du jeu. La tête des sous-tableaux étant la marchandise du haut de la pile.

L’utilisation de tableau étant omniprésente dans notre programme, nous avons réalisé un prédicat pour **afficher les tableaux** :



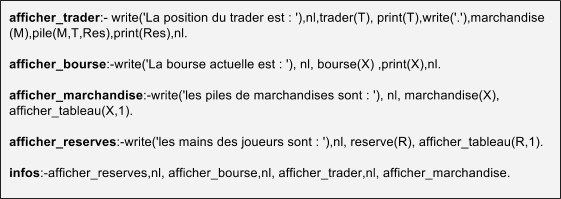
Ainsi que des prédicats pour **agir sur les tableaux**. A savoir : retourner le Nième élément d’une pile et retourner le premier élément d’une pile :



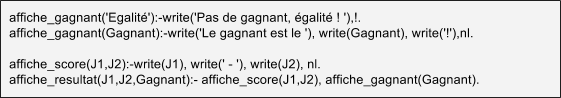
Enfin deux prédicats pour **trouver la position d’un élément** d’un élément dans une liste, et pour afficher la **différence de position** d’un élément entre une ancienne valeur et un nouveau tableau :



Nous avons également les prédicats **d’affichage de la position du trader**, de la bourse, des réserves des joueurs, de la bourse. Tous ces prédicats étant appelés par **infos :**

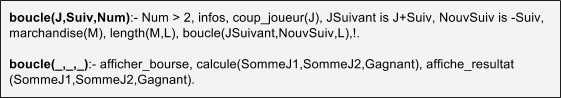


Enfin un prédicat **d’affichage du gagnant** et du score des joueurs :



## Exécution du jeu

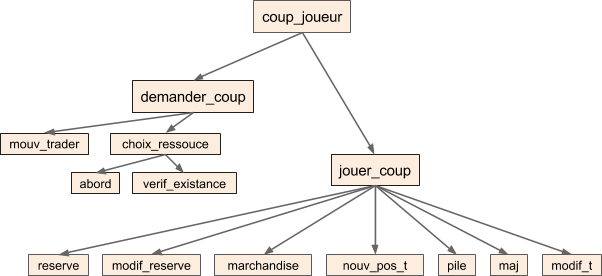
**La boucle** de jeu se déroule de la manière suivante : *Les joueurs jouent tour à tour jusqu’à ce qu’il n’y est plus que 2 piles*. Nous avons donc construit le prédicat boucle de la manière suivante :



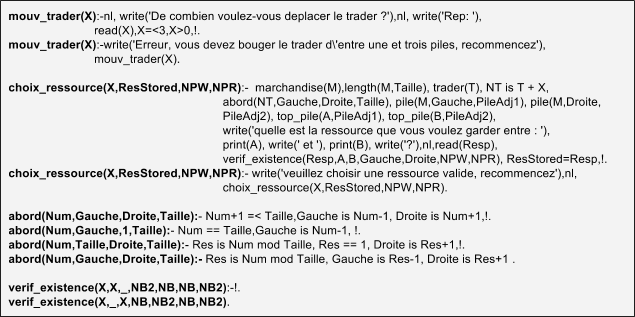
Ce prédicat commence par vérifier que le nombre de pile est strictement supérieur à 2, puis fait jouer successivement le joueur 1 et le joueur 2.

Si la clause Num > 2 échoue, alors le prédicat boucle(\_,\_,\_) est unifié, il déclenche alors l’affichage de la bourse, et le calcul et l’affichage du gagnant.

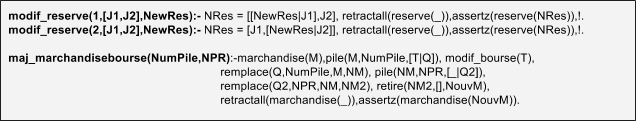
**Les coups de chaque** joueur sont implémentés comme suit.

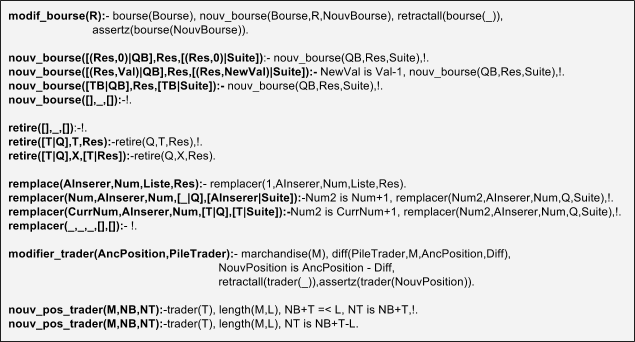


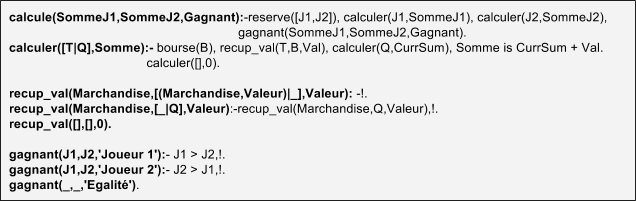
Tout d’abord, le prédicat **coup\_joueur.** Celui-ci va faire le lien entre le coup demandé à l’utilisateur, et le coup effectif joué.

Le prédicat **demander\_coup**, va ensuite demander à l’utilisateur de combien de case celui-ci veut se déplacer avec le prédicat **mouv\_trader**. Puis il va demander à l’utilisateur quelle marchandise il veut prendre et garder. Tout ceci avec l’aide des prédicats de vérification d’existance, pour respecter les contraintes du jeu. (Déplacement par coup, marchandise prise/rejetée,…)

Une fois le coup du jeu “enregistré”, c’est au tour du prédicat **jouer\_coup** de rentrer en action.

Celui-ci va **mettre à jour tout le plateau** (bourse et marchandise) mais aussi la position du joueur et sa réserve. Pour cela de nombreux prédicats ont été implémentés pour remplacer, retirer et mettre à jour les listes.



Enfin, les prédicats permettant de **déterminer le gagnant**. Ceux-ci calculent en premier le score des joueurs en fonction de la bourse et de la main de chaque joueur. Puis le prédicat **gagnant** renvoi le joueur gagnant sous forme de chaîne de caractère.

# 

# 

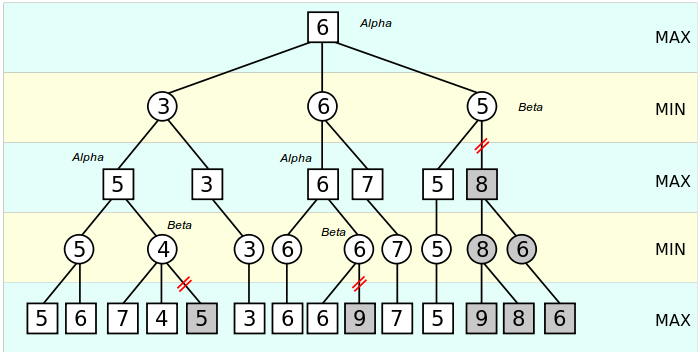
# Description de l’IA

Pour implémenter l'intelligence Artificielle nous avons choisi l’algorithme MinMax Alpha-Beta. En effet nous cherchons d’une part à minimiser les gains de l’adversaire et d’une autre par à maximiser nos propres gains.

Cet algorithme nous semble le plus efficace car il est capable de choisir les branches à explorer. Il ne se contente pas d’explorer le graphe en entier en faisant une profondeur d’abord mais il “élague” intelligemment les branches qui ne donneront toute façon pas de bonne solution.

On a un facteur de branchement possible de 6 pour chaque nœud, du fait que l’on puisse bouger le trader de 3 valeurs différentes et que pour chacune d’entre elle, le joueur a deux possibilité de coup, soit 216 coups possibles sur une profondeur de 3.

Grâce à l’algorithme Minimax Alpha-Beta, en supposant un ordre d’expansion maximum, on arrive à environ 29 branchements à même profondeur, ce qui est considérablement plus petit et donc permet un temps de calcul bien moins élevé.



Pour décrire le processus technique :

* On prend donc la position courante du trader.
* On obtient la liste des coups possibles à partir de cette position.
* On applique l’algorithme de recherche du meilleur coup sur cette liste.

Grâce à celui-ci, on obtient :

* La valeur du coup optimal et le coup associé.
* En remontant l’arbre récursif, le meilleur coup possible à partir de la position courante.

Ainsi, en ayant l’IA qui choisit toujours le meilleur coup sur 3 niveaux de profondeurs, on obtient un ordinateur robuste difficile à contourner.

Toujours d’un point de vue technique, on réutilise les prédicats définis dans la partie d’avant en modifiant légèrement leur signature pour qu’ils retournent les éléments du plateau dans des variables plutôt que dans des assert. En effet, on doit tester le coup optimal sans modifier l’ordre réel du jeu.

# Difficultés rencontrées et améliorations possibles

La plus grosse **difficulté** de ce sujet a été l’implémentation de l’IA. En effet, nous avons mis du temps à choisir l’algorithme le plus performant. (MinMax simple ? MinMax AlphaBeta) Puis une fois ce choix fait, il nous a fallu transposer cet algorithme dans notre programme, et donc l’adapter à du Prolog.

Une autre grande difficulté a été le maintien à jour du plateau en permanence. En effet, un joueur ne peut pas jouer correctement si le plateau n’est pas à jour. Nous devions donc à chaque fois faire remonter les informations modifiées. C’est pour cela que nous avons choisi de scinder le coup\_joueur, par exemple, en deux parties. La première partie, s’occupe de recueillir le coup du joueur, en vérifiant que celui-ci est correct. Et la deuxième partie s’occupe juste de jouer le coup à proprement dit, c’est à dire la mise à jour du plateau.

Cette dissociation permet également d’ajouter de la robustesse au code : en opérant ainsi, on s’assure que l’utilisateur rentre des données qui sont correctes au fur et à mesure que nous les lui demandons, ainsi, le prédicat jouer\_coup s’occupe juste d’effectuer les actions sans « se poser de questions ».

Une **amélioration** possible de notre programme serait d’afficher le plateau de jeu de façon un peu plus visuelle.

Il serait également possible de régler une fonction permettant de régler la “force” du joueur artificielle.