Université de Technologie de Compiègne

Génie Informatique

IA02 - P16

Groupe B

Projet du jeu de Khan - Rapport

Tanguy Lucci
Alexandre-Guillaume Gilbert

tanguy.lucci@etu.utc.fr
alexandre-guillaume.gilbert@etu.utc.fr

16 juin 2016

Structure du projet

Le code source du projet est disponible à l'adresse : http ://github.com/luccitan/KhanIA02

Le projet a été développé et testé sous l'interpréteur SWI-PROLOG.

Le projet contient 8 fichiers :

— **main.pl** : il inclut l'intégralité des autres fichiers. De plus, il contient les prédicats qui permettent de démarrer le jeu (lancement du jeu ainsi que la boucle de tour de jeu). Il initialise également le jeu avec une difficulté et en déclarant les prédicats dynamiques qui vont être utilisés.
— **print.pl** : Il contient les prédicats spécialisés dans l'affichage. Ceci inclut l'affichage du plateau ou encore les prédicats outils pour les affichages utilisateurs (lors de la prise de position). internaltools.pl : Il contient les différents prédicats outils permettant de manipuler les données du jeu et d'en récupérer des informations. ex : extraire le tuple de la cellule d'un tableau, connaître la couleur ennemie d'une autre couleur, etc... externaltools.pl : Il contient différents prédicats qui sont externes au projet, qui permettent de faire des manipulations basiques et/ou abstraites de données. ex : concaténer deux listes, connaître la longueur d'une liste, etc...
— **internaltools.pl** : Il contient les différents prédicats outils permettant de manipuler les données du jeu et d'en récupérer des informations. ex : extraire le tuple de la cellule d'un tableau, connaître la couleur ennemie d'une autre couleur, etc...
— **externaltools.pl** : Il contient différents prédicats qui sont externes au projet, qui permettent de faire des manipulations basiques et/ou abstraites de données. ex : concaténer deux listes, connaître la longueur d'une liste, etc...
— **init.pl** : Il contient tous les prédicats nécessaires à l'initialisation du jeu : interface/menu de choix de type de jeu (Homme vs Homme, Homme VS IA, IA vs IA), interface/menu du choix de plateau de départ, positionnement des pièces au démarrage, ... engine.pl : Il contient les prédicats permettant de générer des mouvements et/ou de générer un plateau à partir de données tels qu'un mouvement donné et un plateau initial.
— **minimax.pl** : Il contient les prédicats de l'IA. L'algorithme minimax y est implémentée, à l'aide d'une fonction score et de sous-fonctions scores.
— **dynamic.pl** : Il contient les prédicats permettant de modifier les prédicats dynamiques (un état de joueur, le plateau courant de jeu, la difficulté, la position courante du Khan)

Présentation des prédicats

Lancement du jeu

Le lancement du jeu s'effectue grâce au prédicat : initBrd(Brd). Ce prédicat permet de choisir le type de match que l'on souhaite jouer : "IA vs IA", "Humain vs IA", ou "Humain vs Humain". Lorsque le choix du type de jeu est effectué, le prédicat lance choseBrdLoop(InitialBrd) permettant de choisir et d'initialiser le plateau du jeu. positioningPhase(InitialBrd, Brd) lance le positionnement des pions et setBrd(Brd) ajoute dynamiquement le plateau à la base de faits.

Choix du plateau

Le choix du plateau s'effectue par l'intermédiaire du prédicat : choseBrdLoop(Brd).

showAllBaseBrds est un prédicat, défini dans "print.pl" permettant l'affichage des 4 orientations de plateaux possibles.

Positionnement des pièces

Le positionnement des pièces fait intervenir deux cas de figure :

— le positionnement des pièces par un joueur humain ;

— le positionnement des pièces par l'intelligence artificielle.

Lorsqu'il s'agit du joueur humain qui place les pièces, un choix de la position de la pièce à placer (sbire ou kalista) lui est demandé :

```
humanPositioningMenu(Brd, N, PlayerSide, ResBrd) :-
        repeat, nl, wSep(20), nl,
        showBrd(Brd, (0,0)),
        write("[Joueur"), write(PlayerSide), write("] => position du sbire "
        writeln(N), write("Inserez les coordonnees dans ce format : X,Y "),
        read(CHOICE), validPositioning(Brd, PlayerSide, CHOICE, CellPower),
        M is N + 1,
        pieceType(PlayerSide, sbire, Type),
        setCell(Brd, (CellPower, Type), CHOICE, SubBrd),
        humanPositioningMenu(SubBrd, M, PlayerSide, ResBrd).
```

humanPositioningMenu

Lorsqu'il s'agit de l'intelligence artificielle, le positionnement des pièces est fait de manière aléatoire : generateRandomStartPosition(rouge, (X,Y)) :- random(5,7,X), random(1,7,Y).

Boucle du jeu

— Le prédicat possiblesMoves permet de générer l'ensemble des coups possibles pour les pièces d'un joueur, tout en prenant en compte la position du khan.
— Le déplacement des pièces se fait par l'intermédiaire du prédicat createBrd(Brd, (Xstart, Ystart), (Xend, Yend),BrdRes) . (Xstart, Ystart) correspond aux coordonnées de la pièce que l'on souhaite déplacer vers la position (Xend, Yend)

Le maintien de la boucle du jeu se fait par l'intermédiaire du prédicat : gameRoundLoop() gameRoundLoop() unifie tout d'abord le plateau de jeu avec le prédicat Board(Brd), récupère le khan et déclare le joeur adverse gagnant si jamais la kalista du joueur jouant le coup n'est pas présente sur le plateau. Lorsqu'un joueur est gagnant, la partie s'arrête et gameRoundLoop() affiche le plateau final du jeu.

Si aucun joueur n'est gagnant, on génère l'ensemble des coups possibles et on demande au joueur de choisir une pièce et sa nouvelle position grâce au prédicat askTheMove(PossibleMoves, Move). Avant de valider le mouvement, on vérifie que la nouvelle position est correcte via le prédicat : moveAskedPossible(Cstart,Cend, PossibleMoves). On unifie le nouveau plateau généré avec le prédicat createBrd(Brd, StartPosition, EndPosition, BrdRes) et on place correctement le khan sur la dernière pièce jouée.

Structures de données

- **Plateau** : pour représenter le plateau, nous avons fait le choix d'utiliser une liste de listes. Chaque liste intérieure représente une ligne du plateau, contenant les différentes cellules de la ligne. L'avantage de cette représentation des données est de parcourir très facilement les différentes cellules d'un tableau tout en permettant une connaissances des coordonnées de celui-ci grâce à l'incrémentation de variables.
- **Les coordonnées** : nous avons fait le choix de représenter les coordonnées sous la forme X,Y avec X étant le numéro de ligne et Y le numéro de colonne. Cela peut paraître contre-intuitif si on se réfère au système abscisse/ordonnée qui fonctionne de manière opposée, mais notre plateau étant une liste de lignes, il nous paraissait évident que la première coordonnée désigne celles-ci.
- **Les cellules de la ligne** : elles sont représentées par un tuple , de la forme (Puissance, Contenu)
  - **Puissance** représente la puissance de la case. La valeur est soit de 1, de 2 ou de 3.
  - **Contenu** représente le contenu de la case. Il peut prendre une valeur de type PieceType (cf. ci-dessous).

  Nous appelons **PieceType** la donnée sur une pièce :

  - **ko** représente la Kalista ocre
  - **so** représente un sbire ocre
  - **kr** représente la Kalista rouge
  - **sr** représente un sbire rouge
  - **empty** représente une case vide.

```
[
    [ (2,empty),(2,empty),(3,empty),(1,empty),(2,empty),(2,empty) ],
    [ (1,empty),(3,empty),(1,empty),(3,empty),(1,empty),(3,empty) ],
    [ (3,empty),(1,empty),(2,empty),(2,empty),(3,empty),(1,empty) ],
    [ (2,empty),(3,empty),(1,empty),(3,empty),(1,empty),(2,empty) ],
    [ (2,empty),(1,empty),(3,empty),(1,empty),(3,empty),(2,empty) ],
    [ (1,empty),(3,empty),(2,empty),(2,empty),(1,empty),(3,empty) ]
 ]
```

Un tableau vide selon notre représentation des données

Le plateau courant du jeu est stockée à l'aide d'un prédicat dynamique.

Un prédicat dynamique est utilisé pour la position du Khan courant : cette information est donc séparée du plateau, et est représentée sous la forme X,Y. Cela nous évite d'ajouter une information supplémentaire au tuple pour savoir si cette cellule contient le Khan ou pas.

Des prédicats dynamiques sont utilisés pour stocker des informations sur les joueurs. Ces prédicats sont de la forme (CouleurJoueur, TypeJoueur).

— **CouleurJoueur** est soit ocre, soit rouge. Il permet de distinguer les deux joueurs
— **TypeJoueur** peut prendre la valeur homme ou la valeur ia. Cela permet de déterminer le type de joueur et sera utilisé lors du début de round pour savoir s'il faut générer le coup ou alors le demander via un interface.

Au début, nous avions rajouté une liste des pièces présentes pour chaque joueur. Cependant, on s'est aperçu que la liste était inutile car le plateau est parcouru de toute manière pour générer les mouvements possibles lors de la génération du meilleur coup pour l'IA ou lors de la vérification de la validité d'un coup proposé par un joueur humain.

Intelligence Artificielle

Afin d'implémenter un mode de jeu "Homme vs IA" ainsi qu'un mode de jeu "IA vs IA", il était nécessaire de mettre en place une intelligence artificielle capable de générer un coup qui lui est potentiellement favorable afin qu'elle gagne. Pour cela nous avons décidé d'implémenter l'algorithme minimax. Celui-ci consiste à utiliser une fonction score. Celle-ci, dans notre cas, évaluera l'état d'un plateau en fonction d'un côté (joueur ocre ou joueur rouge). L'algorithme minimax consiste à maximiser le score de son prochain coup, parmi tous les coups possibles. En prévoyant des coups ultérieurs, il va ainsi à chercher à prévoir le coup suivant en partant du principe que l'adversaire est le plus intelligent possible, ce qui revient à minimiser le score du coup suivant selon son point de vue, car l'adversaire va faire le coup qui l'arrange le moins. Toute la force de l'algorithme repose donc sur la précision de la fonction score, c'est-à-dire sa capacité à écarter les bons coups des mauvais, et ainsi en déduire le meilleur (ou le pire). Pour cela, nous avons considéré ces différents points stratégiques concernant le jeu de Khan :

- Il n'est pas favorable d'avoir des pions (sbires ou Kalista) sur une case de puissance 2 car elles ne permettent à ces pions de ne bouger que sur un seul type de case (la liberté stratégique qui en découle est donc fortement limité, l'adversaire peut prévoir facilement les coup qui nous seront possibles).
- Il est favorable d'avoir moins de pièces que l'adversaire. En effet, en ayant moins de pièces, il est plus facile d'être en dehors des limites du Khan concernant les mouvements possibles. Ainsi, il est plus probable d'avoir le droit de bouger le pion que l'on veut au lieu de 1 ou de pions à cause de la limitation du Khan
- Il est très fortement favorable d'avoir une Kalista seule (contrairement à une Kalista et un seul autre sbire, par exemple). Celle-ci sera complétement libre de ses mouvements (dans le respect de la puissance de sa case initiale bien évidemment). Dans ce cas présent, il est très difficile pour une Kalista seule d'être tué.
- Nous avons considéré en plus la stratégie triviale d'approcher le plus possible les pions de la Kalista ennemie. Nous prenons donc en compte la distance moyenne entre les premiers de la dernière.

À partir de ces points stratégiques, nous avons pu établir la fonction score :

- Nous vérifions d'abord que les cas extrêmes ne se manifestent pas : si la Kalista de la personne considérée pour l'évaluation est morte, alors un score potentiellement infini dans le négatif est renvoyé (ici -10000)
- Dans le cas opposée, c'est-à-dire la Kalista ennemie décédée, nous renvoyons un score infiniment positif.

Si aucun cas extrême ne se présente, nous partons de 0 et nous sommons des sous-scores en évaluant chacun des 4 cas présentés :

- Le score se voit perdre des points si des pions du joueur sont sur des cases deux.
- Nous ajoutons la différence de pions entre l'ennemi et le joueur (l'ordre est important : si le joueur a moins de pions, la différence sera positive et le score sera donc valorisé)

— Le score est fortement augmenté si la Kalista est seule
— La distance moyenne est retirée au score (ainsi, plus les pions sont proches, moins le score s'en voit défavorisée).

```
score(Brd, PlayerSide, Score)    :-
    subScoreOwnKalistaDead(Brd, PlayerSide, SC1),
    SC1 = -10000, Score is SC1, !;
    subScoreEnemyKalistaDead(Brd, PlayerSide, SC2),
    SC2 = 10000, Score is SC2, !;
    subScoreLonelyKalista(Brd, PlayerSide, SC3),
    subScoreLessPiecesThanEnemy(Brd, PlayerSide, SC4),
    subScoreDistanceFromEnemyKalista(Brd, PlayerSide, SC5),
    subScoreNumberOfPower2(Brd,PlayerSide, SC6),
    Score is SC3+SC4+SC5+SC6.
```

Prédicat score vérifiant les cas extrêmes, et dans le cas non échant calcule un score selon notre stratégie.

Grâce à l'algorithme minimax, nous avons pu mettre en place un système de difficulté (il n'y a pas d'interface à proprement parlé pour changer la difficulté, le changement se fait directement via l'appel d'un prédicat dans l'interpréteur). Cette difficulté correspond à la profondeur de l'algorithme minimax :

— Difficulté "FACILE" (profondeur = 1). L'IA va simplement chercher le coup de score maximal parmi ses mouvements possibles direct
— Difficulté "NORMALE" (profondeur = 2). L'IA va chercher le coup de score maximal en sachant que l'adversaire va minimiser son score au prochain coup (MAX - MIN dans l'arborescence des coups possibles)
— Difficulté "DIFFICILE" (profondeur = 3). L'IA va chercher à chercher le coup de score maximal, parmi ceux qui ont été maximisés malgré une minimisation de l'adversaire (MAX - MIN - MAX dans l'arborescence des coups possibles).
  En plus de l'algorithme minimax, nous avons implémenté un mécanisme en amont afin de pallier un souci de minimax. En effet, l'algorithme peut ne pas choisir un coup directement gagnant à cause d'une prévision trop lointaine. Si un coup lui est gagnant mais que le coup prochain - malgré le décès de la Kalista ennemie, lui fait mourir sa Kalista, un score négatif sera remonté dans l'arborescence et il choisira alors de ne pas jouer ce coup. Notre solution est donc de toujours tester la profondeur minimale en premier.
— (I) Si la Kalista ennemie est morte sur un des coups prévues, il privilégiera ce coup directement et arrêtera l'algorithme.
— (II) Sinon, si la profondeur maximale (c'est-à-dire la difficulté) n'est pas atteinte, alors réitérer le (I) sur une profondeur incrémentée de 1.

Bien que cela puisse allonger le temps de calcul, le résultat pratique montre un ralentissement

négligeable. La complexité de l'algorithme minimax étant k^N, avec 3 itérations de l'algorithme successivement, nous obtenons une complexité maximale de $k^{3+2+1} = k^6$. Bien que cela puisse être beaucoup, le nombre de ramifications (k), ainsi que la nature des traitements effectués permettent un calcul du coup maximal d'un dixième de seconde (approximativement).

Difficultés rencontrées et améliorations possibles

La difficulté principale a été d'ordre technique. En effet, n'étant pas familiers avec Prolog, il a fallu faire un effort particulier pour manipuler les différents changements de données et les différentes valeurs à retourner et à manipuler (lors des générations de mouvements ou lors de l'algorithme minimax, par exemple) pour obtenir un code fonctionnel.

Par manque de temps, et au vu des priorités que demandaient le projet, nous n'avons pas implémenté une règle (et c'est le seul point manquant aux règles dans le projet). Il est en effet impossible de réinsérer un pion lorsque le Khan est trop restrictif. Cela retire beaucoup de possibilités dans le jeu, il est vrai. Le temps nous manquant, et sachant qu'il existant toujours l'alternative de toutes les pièces amovibles, nous avons préféré avoir un algorithme minimax complétement fonctionnel plutôt qu'une règle implémenté sur une IA non performante, donc non pertinente.