PROJET IA02 – Le jeu KHAN

Rapport Final

Table des matières

Avant Propos

Nous avons développé notre programme sous *Swi-Prolog*. Nous vous conseillons donc de le lancer avec *Swi-Prolog* pour éviter tout problème éventuel de compatibilité.

De plus, nous avons séparés les fichiers et nous incluons tous les fichiers au lancement du jeu grâce au prédicat *include_sources*. Pour que cela marche, nous vous conseillons de lancer *Swi-Prolog* depuis le terminal avec la commande swipl et en étant dans le dossier sources, puis de faire [main] et enfin appeler le prédicat *main*.

I – Principaux prédicats du jeu et fonctionnement interne

I.1 – Les faits statiques

I.1.1 – Liste des faits statiques

Faits servant à définir les numéros des joueurs

- *player1(1)*
- *player2(2)*

Liste des pions

- *pawnList(['S1', 'S2', 'S3', 'S4', 'S5', 'K'])*

Plateau de jeu

- *chooseBoardDisplay(Clé, PlateauVouluSelonClé)*
- Fait statique qui permet au joueur de choisir la disposition du plateau qu'il veut

I.1.2 – Utilisation des faits statiques

Avoir la valeur d'une case ayant pour coordonnées X et Y

- *get_cell_value(X, Y, CellValue)*

Récupérer le code de l'autre joueur

- *get_other_player(ActualPlayer, Player_2)*
- *get_other_player(ActualPlayer, Player_1)*

Vérifier qu'une case de coordonnées X et Y soit sur le plateau

- *cell_in_board(X,Y)*

I.2 – Les faits dynamiques

Le plateau de jeu « actif »

- *:-dynamic(activeBoard/1)*
- Permet de sauvegarder et récupérer la disposition du plateau choisie par le joueur au début du jeu

Indices I et J qui servent au déplacement sur le plateau de jeu au cours de la partie

- *:-dynamic(i/1)*
- *:-dynamic(j/1)*

Prédicat pawn/4 qui sert à gérer les pions des joueurs sur le plateau

- *:-dynamic(pawn/4)*
- S'utilise de la façon suivante : *pawn(Indice_ligne, Indice_colonne, Pion, Joueur)*
- Permet énormément de choses :
    - En ayant la ligne et la colonne, on peut savoir s'il y a un pion sur la case et si oui, quel est ce pion et à quel joueur il appartient
    - En ayant le pion et le joueur, on peut savoir si le pion du joueur est sur le plateau et si oui, à quels indices il se situe
    - En ayant la ligne, la colonne et le joueur, on peut savoir si la case est vide, possède un pion adverse ou bien un pion du joueur.
- C'est un fait dynamique qui nous sera utile tout au long du jeu pour les déplacements des pions et le calcul des coups possibles
    - Permet de calculer la liste de tous les pions d'un joueur actuellement sur le plateau, de même que la liste des pions d'un joueur sortis du plateau

Prédicat *khan/2* permettant de gérer la position du Khan

- *:-dynamic(khan/2)*
- Les deux arguments sont les indices de ligne et de colonne
- Fait dynamique qui est utile pour :
  - Récupérer la valeur de la case actuelle du Khan
  - Vérifier qu'une case possède la même valeur que la case actuelle du Khan

III – L'affichage du plateau de jeu

Affichage du plateau de jeu grâce au prédicat *dynamic_display_board(Board)* qui affiche la variable *Board* en utilisant les fait dynamiques *I* et *J*.
Nous avons aussi mis en place un un affichage des pions des joueurs en couleur afin de bien les distinguer entre eux.

IV – Placement des pions au tout début du jeu

IV.1 - Prédicats de contrôle

Prédicats de contrôle servant à vérifier les informations entrées par l'utilisateur.

pawn_in_initial_lines(X,Y,Player)

- Vérifie, selon que *Player* soit le joueur 1 ou 2, que les coordonnées X et Y correspondent à une case située sur les deux premières lignes faisant face au joueur

initial_pawn_placement_correct(Player, X, Y)

- Appelle le prédicat *pawn_in_initial_lines(X,Y,Player)*
- Vérifie que le joueur *Player* ne possède pas déjà un pion sur la case de coordonnées X et Y

IV.2 - Prédicats de gérant le placement des pions

ask_one_player_initial_pawns_placement (Player, ListeDesPions)

- Demande à un joueur de placer l'ensemble des pions présents dans la liste *ListeDesPions*.
- Se charge de vérifier que le placement de chaque pion est correct
- On place un pion grâce au prédicat *place_pawn(X, Y, Pawn, Player)* qui fait appel au fait dynamique *pawn*
- Affiche le plateau de jeu à chaque nouveau placement

ask_initial_pawns_placement()

- Demande aux deux joueurs de placer leurs pions (*ask_one_player_initial_pawns_placement*) puis affiche le tableau une fois tous les pions placés

V – Mouvements des pions au cours du jeu

V.1 - Prédicats de contrôle

check_good_coordonates(X,Y,MoveList)

- Prédicat qui vérifie que les coordonnées X et Y correspondent bien à des coordonnées présentes dans la liste MoveList représentant l'ensemble des coups possibles

move_right(X, Y, NY)

- Prédicat qui vérifie qu'une pièce peut bouger vers la droite. Si c'est le cas, alors NY représente la valeur de Y après le mouvement
- Les prédicats suivants ont un fonctionnement similaire :
  move_left(X, Y, NY)
  move_up(X, Y, NX)
  move_down(X, Y, NX)

finish_right(X, Y, NY, Player)

- Prédicat qui vérifie qu'une pièce peut bouger et terminer son mouvement vers la droite. Si c'est le cas, alors NY représente la valeur de Y après le mouvement. A la différence de *move_right* qui vérifie si la case de droite est vide, il faut ici seulement vérifier que la case ne possède pas de pion du joueur. En effet, ce dernier ne peut pas manger ses propres pions mais peut manger ceux de l'adversaire.
- Les prédicats suivant ont un fonctionnement similaire :
  finish_left(X, Y, NY, Player)
  finish_up(X, Y, NX, Player)
  finish_down(X, Y, NX, Player)

V.2 - Prédicats de générations de coups et pions possibles à jouer

get_all_cells_according_to_khan_cell_value(CellList)

- Prédicat qui va remplir la variable *CellList* avec l'ensemble des cases du plateau qui ont la même valeur que la case actuelle du Khan. De plus, toutes les cases contenues dans *CellList* à l'issue de l'appel seront aussi des cases vides.
- Ce prédicat est utilisé lorsqu'un joueur peut remettre un pion en jeu : il lui faut alors le remettre sur une case vide possédant la même valeur que la case actuellement occupée par la Khan.

possible_moves(X, Y, Player, Range, AlreadySeens, MoveList)

- Prédicat qui permet de générer tous les coups possibles pour une case, un joueur et la distance en nombre de cases (1, 2 ou 3 selon la case de départ).
- La liste des coups possibles sera unifiée avec la variable *MoveList*.
- La variable *AlreadySeens* est là pour contenir l'ensemble des cases que l'on a déjà visité au cours de la recherche. En effet, un pion, lors d'un déplacement, ne peut pas revenir sur ses pas.
- La liste complète *MoveList* des cases de destination possibles en partant de la case X,Y et avec un déplacement d'une valeur de *Range* est obtenue en utilisant *setof* sur le prédicat *possible_moves* :
  setof(ML, possible_moves(I,J,Player,CellValue,[(I,J)],ML), ML),
  flatten(ML, MoveList).

get_possible_pawn(Player, UsedPawnList, PossiblePawnList)

- Prédicat qui retourne l'ensemble des pions qu'un joueur peut joueur en tenant compte de la position actuelle du Khan : un joueur est obligé de jouer une pièce se situant sur une case de même valeur que la case actuelle du Khan.
- Si le joueur n'a aucun pion sur une case de même valeur que celle du Khan, alors il pourra joueur l'ensemble de ses pions, y compris les pièces précédemment mangées par l'adversaire.

V.3 - Prédicats gérant le déplacement des pions

ask_pawn_new_position(I,J,MoveList,Pawn,Player)

- Prédicat qui permet à un joueur de déplacer une pièce.
- La liste des coups possibles contenue dans *MoveList* est affichée
- Il lui est demandé de rentrer les nouvelles coordonnées
- Si les coordonnées sont mauvaises, on les lui redemande. Sinon, on bouge le pion en appelant le prédicat *move_pawn(I, J, X, Y, Pawn, Player)*.
  I et *J* sont les anciennes coordonnées, *X* et *Y* les nouvelles. *Pawn* représente le nom du pion à déplacer et *Player* contient la valeur (1 ou 2) du joueur.

ask_pawn_new_placement(MoveList,Pawn,Player)

- Prédicat qui demande à un joueur de choisir la nouvelle position d'une pièce dans le cas où il remet la pièce sur le plateau alors qu'elle était sortie du jeu
- Fonctionnement identique au prédicat *ask_pawn_new_position(I,J,MoveList,Pawn,Player)* à la différence qu'on a pas besoin des coordonnées d'origine et que *MoveList* contiendra la liste des cases obtenues via le prédicat *get_all_cells_according_to_khan_cell_value(CellList)*.
- De plus, on place le pion non plus avec *move_pawn* mais avec le prédicat *place_pawn(X,Y,Pawn,Player)*

ask_movement_to_player(Player)

- Principal prédicat permettant à un joueur donné (représenté par la variable *Player*) de choisir un pion à déplacer et sa destination.
- Utilisation du prédicat *get_possible_pawn* pour définir la liste des pions que le joueur peut jouer
- Si elle est vide, utiliser la liste des pions complète avec le fait statique *pawnList*.
  Sinon garder la liste retournée par le prédicat *get_possible_pawn*.
- Demander quelle pièce veut jouer le joueur et vérifier que le joueur rentre bien le nom d'une pièce contenue dans la liste des pièces possibles à jouer.
- Si le joueur a choisi de replacer un pion, appeler le prédicat *ask_pawn_new_placement* avec la liste des cases possibles obtenues via le prédicat *get_all_cells_according_to_khan_cell_value*.
- Si je joueur a choisi de bouger un pion présent sur le plateau, appeler le prédicat *ask_pawn_new_position* avec les liste des mouvements possibles obtenue via le *setof* sur le prédicat *possible_moves*.

VI – Boucle de jeu et fin de la partie

La boucle de jeu permettant aux joueur de jouer l'un après l'autre jusqu'à que l'un deux prenne la Kalista adverse fait appel à trois prédicats.

player_win(Player)

- Prédicat qui renvoi vrai si *Player* a gagné et faux sinon
- Il récupère la valeur du joueur adverse en utilisant le prédicat *get_other_player(Player,OtherPlayer)*
- Il construit la liste des pions non utilisés de *OtherPlayer*, c'est à dire les pions en dehors du plateau
- Et il teste enfin si la Kalista fait partie de cette liste de pions hors jeu

human_vs_human_game_loop(Player)

- Prédicat permettant de demander au joueur *Player* le mouvement voulu
- Il teste ensuite si ce joueur vient de gagner
- Si c'est le cas, le jeu s'arrête et le gagnant est félicité
- Sinon on rappelle récursivement *human_vs_human_game_loop* sur le joueur adverse

human_vs_human_launch_game()

- Prédicat permettant de lancer le jeu en appellant *human_vs_human_game_loop* sur le joueur 1.

II – Intelligence artificielle

II.1 – La théorie

L'intelligence artificielle est une adaptation de l'algorithme du min-max. On retrouve les prédicats eval, min, max et joue.

Le prédicat eval permet d'attribuer un score à la situation du plateau. Plus ce score est bas, plus le joueur est proche de gagner. Cette fonction utilise une heuristique pour calculer ce score.

L'heuristique choisie a besoin d'être revue car elle renvoie trop souvent la valeur 0. Elle consiste à compter le nombre de pions qui menacent la Kalista adverse et le nombre de pions qui menacent notre Kalista. La différence entre les deux est le résultat de l'évaluation.

Les prédicats min et max servent à simuler les futurs coups de l'adversaire ou de l'IA. Ces prédicats parcourent un arbre en profondeur d'abord afin de parcourir toutes les situations pouvant découler de la situation initiale du plateau de jeu. La profondeur de l'algorithme (*ie*, le nombre de coups à l'avance que l'IA anticipe) est réglable.

Enfin le prédicat joue permet de lancer la récursivité des prédicats min-max. Quand le parcours de l'arbre est terminé, le meilleur coup est joué.

II.2 – Représentation de l'information

Pour réaliser l'algorithme min-max, il a fallu revoir la façon de représenter un coup afin que celui-ci contienne suffisamment d'information pour être annulé.

Il existe 3 types de coups :

- Le déplacement d'une pièce vers une case vide
- Les déplacement d'une pièce vers une case occupée par un pion adverse
- Le placement d'un pion sorti sur le plateau de jeu

Un coup est représenté par un tuple de 9 valeurs : (Xini, Yini, Xfin, Yfin, P, Iini, Jini, Ifin, Jfin)

- Lorsqu'un coup est un déplacement vers une case vide, alors Xini et Yini représentent la case d'origine de la pièce. Xfin, Yfin la case de destination. Le champ P prends la valeur 'V' pour vide et les 4 champs suivants sont 0. Le prédicat permettant de tester ce mouvement est le suivant : *is_simple_move((_,_,_,_,'V',0,0,0,0))*

- Lorsqu'un coup est un déplacement vers une case occupée par un pion de l'adversaire, alors Xini et Yini représente la case d'origine de la pièce. Xfin et Yfin la case de destination. Le champ P prends la valeur du pion qui est sorti du jeu et les champs Iini et Jini prennent les coordonnées de la case où était ce pion sorti du jeu. Les autres champs sont nuls. Le prédicat permettant de tester ce mouvement est le suivant : *is_kill_pawn_move((_,_,X,Y,_,X,Y,0,0))*

- Lorsqu'un coup est un placement d'un pion sur une case alors les quatre premiers champs prennent la valeur 0. Le champ P prends la valeur du pion qui rentre dans la partie. Les champs Iini et Jini prennent la valeur 0. Enfin les champs Ifin et Jfin prennent les coordonnées de la destination du pion. Le prédicat permettant de tester ce mouvement est le suivant : *is_placement_move((0,0,0,0,_,0,0,_,_))*

II.3 – Prolog

Différents prédicats ont été nécessaires pour l'élaboration de l'algorithme d'intelligence artificielle.

Tout d'abord le prédicat *is_that_pawn_in_range(MoveList, Pawn, JoueurAdverse, X, Y)* qui retourne vrai si le pion passé en paramètres et qui place dans X et Y les coordonnées de la case du pion adverse.

Le prédicat *get_IAMoveFormat (JoueurActif, JoueurAdverse, X, Y, MyPawn, (I,J), Move)* permet de transformer un mouvement tel qu'il se présente après la recherche des mouvements possibles par le prédicat *possible_moves* en un mouvement d'un type de mouvement exploitable par l'intelligence artificielle.

Pour récupérer la liste de tous les mouvements possibles d'un pion dans le format de mouvement de l'intelligence artificielle, il faut appeler le prédicat *get_pawn_moves(JoueurActif, JoueurAdverse, Pawn, MoveList)*.

Les prédicats *apply_move* et *revert_move* permettent respectivement d'appliquer un mouvement du format de mouvement propre à l'intelligence artificielle ou de l'annuler.

III – Difficultés rencontrées et améliorations possibles

I.1 – Difficultés rencontrées

La principale difficulté que nous avons rencontrée est la prise en main du langage de programmation *Prolog*. Ce dernier est à l'opposé de ce que l'on a l'habitude de faire en termes d'habitudes de réflexion et de conception d'un programme informatique.
On ne compte plus les heures passées à bloquer sur un problème de syntaxe extrêmement simple au final.
Le débogage sur *Prolog* est aussi un problème. Nous n'avons pas réussi à faire fonctionner les outils de debug qui apparemment existent sur *Swi-Prolog* et avons dû débugger à la main, grâce au prédicat *write*.

Une autre difficulté rencontrée fut de mettre en place des boucles de contrôle des saisies faites par l'utilisateur. Extrêmement simple à faire en programmation procédurale classique, cela s'est révélé un peu plus compliqué sur *Prolog*, même si nous avons finalement réussi. L'astuce était d'éviter d'être piégé dans le *backtracking*.

I.2 – Amélioration

Nous sommes déçus de ne pas avoir eu le temps de finir l'implémentation de l'intelligence artificielle. Celle-ci marche pour un temps mais plus la profondeur de recherche du meilleur coup est grande, plus l'IA renvoie faux rapidement et met fin au jeu.
De plus l'heuristique de l'IA est à améliorer.