Printemps 2017 IA02

Rapport projet Arimaa

Antoine MALLARD Mewen MICHEL

Objectif

Le but de ce projet est de développer une Intelligence Artificielle en Prolog pour le jeu Arimaa. Arimaa est un jeu de plateau proche du jeu d'échecs dont les règles sont simples, mais dont la nature le rend très intéressant pour le développement d'une IA. Nous avons donc été chargés d'implémenter le jeu et d'implémenter une IA pour jouer contre un humain. Le détail du sujet se trouve sur Moodle.

Réalisation

Afin de pouvoir tester notre IA sur l'interface Javascript prévue, nous avons forké le dépôt de M. Lacaze sur Github (github.com/mmewen/Prolog-Arimaa). Notre code commenté se trouve dans *./ArimaaIA02/apps/arimaa/test.pl*. Nous avons choisi de fonctionner de manière incrémentale : tout d'abord intégrer les règles du jeu à notre IA pour lui permettre de jouer de manière aléatoire puis intégrer une intelligence plus intéressante.

Conventions

Dans tout notre code, les paramètres de retour sont les premiers paramètres d'un prédicat, les paramètres d'entrées étant à la fin. Nous nous sommes aperçu trop tard que la convention était en fait inverse.
De plus, nous avons pris quelques convention pour le contenu des variables. Ces structures ont été choisies afin de faciliter les échanges avec l'interface graphique et reprennent donc certains choix. Les principales structures utilisées sont donc les suivantes :

- Une variable au pluriel correspond à une liste de ce qu'elle désigne
- *Gamestate* :
  [*side*, *Pieces*]
- *Board* :
  [*Pieces*]
- *Piece* (ou *Occupant* ou *Opponent*) désigne une pièce telle que définie dans la *Board* :
  [Row, Col, Type, Color]
  par exemple :
  [0, 1, rabbit, silver]
- *Position* (ou *Pos*) comportent uniquement la ligne et la colonne ($0 \leq X \leq 7$) de la case du plateau dont on parle :
  [*Row*, *Col*]
- *Move* est une liste de longueur 2 donnant la position initiale et la position finale d'une pièce après un seul mouvement :
  [Pos1, Pos2]
  par exemple :
  [[0, 1], [1, 1]]
- *State* est une liste de longueur 3 donnant la suite de mouvements à effectuer pour arriver au *Gamestate* et à la *Board* donnés :

[Moves, NewGamestate, NewBoard]

- *ScoredState* est une liste de longueur 4 donnant le score de l'état final après un certain nombre de mouvements :

  [Score, Moves, NewGamestate, NewBoard]

  Le score est un entier positif, plus il est faible, meilleur est l'état final. Si *Score* vaut 0, l'état final est gagnant.

IA aléatoire

Pour commencer, nous avons défini plusieurs prédicats qui nous seront utiles par la suite : *concat*, *nth*, *replace*, … Puis nous avons ajouté des faits nous permettant de connaître la force d'un type de pièce (*strength*) ainsi que la position des pièges (*is_on_trap*). Ces derniers sont définis de manière à permettre la vérification (*?- strength(X, cat).* ou *?- is_on_trap([2, 2]).*) ainsi que l'énumération (*?-is_on_trap(Pos).*).

Pour retourner 4 mouvements aléatoires (*four_random_moves*), on tire donc 4 fois un mouvement au hasard (*move_one_random*). On calcule donc tous les coups possibles (*all_possible_moves*) avant d'en tirer un (*one_random_move*) et de l'appliquer au terrain actuel (*apply_move*). Pour trouver tous les coups possibles, on calcule tous les mouvements possibles de chaque pièce de notre couleur sur le terrain (*possible_moves*). Ce prédicat fait appel à des prédicats plus simples tenant compte des bords du terrain, des pièces présentes dans les cases voisines ainsi que du fait que la pièce soit gelée ou non.

On note donc que cette version tient compte des piège dans ses calculs mais n'inclut pas de possibilité de les éviter. De plus, nous n'avons pas implémenté de concept de pousser ou tirer un adversaire car il ne nous semblait pas primordial.

IA par meilleur score après N coups

Dans un second temps, nous avons implémenté une IA basé sur un scoring du plateau. Nous avons choisis de prendre en compte différents facteurs tel que :

- La distance qui sépare chaque lapin du bout du plateau (*get_dist_from_goal*)
- La distance qui sépare chaque lapin de la case du bout du plateau vide la plus proche (*get_dist_to_freedom*)
- Les pièces prises (*get_taken_piece_score*)

Chacun de ces facteurs est multiplié par un coefficient d'importance que l'on peut faire varier pour modifier le comportement de l'IA. Plus une position est à notre avantage, plus la fonction de scoring renvois un petit score. Ainsi, lorsqu'un lapin arrive au bout du plateau (*win*), la fonction renvoie 0.

Pour retourner les 4 meilleurs coups selon nos critères (*best_state*), on garde les q meilleurs coups possibles suivant la position actuelle en itérant l'opération 4 fois (*q_best_n_moves,* avec $n = 4$ et $q \leq 5$ dans nos tests). Pour décider des meilleurs coups possibles nous les classons par score (*sort_states_by_score*), ensuite il nous suffit de garder les q premiers (*keep_q_first*). La fonction de scoring (*get_score*) est donc appelée pour chaque coups possible dans un premier temps, puis pour chaque coup possible après les q meilleurs

coups et ainsi de suite. *q_best_n_moves* retourne donc une liste de n^q états possibles en réalisant n mouvements à partir de l'état courant.

Commentaires

Déroulements du projet

En développant ce programme, nous avons réussi à nous tenir à notre conception malgré quelque difficultés d'adaptation au langage. Nous avons été surpris par sa puissance en terme de rapidité de calcul mais aussi en terme de concision. En effet, obtenir une IA capable de "comprendre" un jeu et de proposer des coups en quelques centaines de lignes de code nous paraissait improbable. Malgré sa puissance, nous avions peur d'avoir une consommation CPU ou mémoire vive démesurée avec notre exploration de tous les coups possible, c'est pourquoi nous nous sommes limité à une sélection des *q* meilleurs coups.

Améliorations

Si nous avions eu plus de temps, nous serions concentrés sur les améliorations possibles suivantes.

Dans un premier temps, nous aurions souhaité faire en sorte que nos fonctions soient utilisables pour la couleur adverse. Cela ne demanderai que quelque modification et permettrai d'étudier les coups jouable par l'adversaire pour prévoir et ainsi s'adapter. Maintenant que nous avons pu observer qu'il est possible de lire 4 coups à l'avance en un temps acceptable, cette amélioration est désormais très utile.

D'autre part, l'IA que nous avons développé ne base son intelligence que sur des facteurs choisis. Elle n'est pas capable de résoudre le jeu. Il faudrait pour cela réussir à explorer la quasi-totalité des coups possibles. Nous avons bien peur que cela prenne trop de temps de calcul. Cela dit, afin d'améliorer l'intelligence nous pourrions rajouter des heuristiques tel que "protéger les cases départ", "protéger ses lapins" ou encore "ne pas être trop proche d'une pièce plus forte".

Enfin, il aurait été préférable de développer la possibilité de tirer et pousser les pièces. Notre IA ne dispose donc pour l'instant d'aucun moyen pour prendre les pièces adverses, ce qui l'empêche quasiment totalement de gagner.

Conclusion

Nous avons donc réussi à coder une intelligence artificielle en programmation logique, bien qu'elle soit assez basique et ne prenne en compte pour l'instant que peu de facteurs. Ce projet nous a permis de nous rendre compte de la puissance de Prolog et d'en tester plusieurs fonctionnalités intéressantes.