Rapport du projet IA02 : L'Awalé

*

P14 - Par Antoine Giraudmaillet et Anaig Maréchal

Sommaire

1. Introduction

Dans le cadre de l'UV IA02, il nous a été demandé de réaliser un Awalé, un jeu de plateau Mancala, en utilisant Prolog et les notions abordées en cours et TD.

2. Description du jeu Prolog

Respect des règles du jeu :

Au sein de notre programme, nous avons intégré les règles originales de l'Awalé.

- Le plateau se compose de 12 trous et de 48 graines (représentés par des chiffres entre parenthèses).
- Les graines se distribuent circulairement, dans le sens contraire des aiguilles d'une montre.
- Les joueurs s'alternent à chaque coup.
- Les graines du dernier trou semé sont récupérées si celui-ci se trouve dans le champ adverse et en contient deux ou trois, ainsi que les graines de tous les trous précédents respectant cette contrainte.
- Si un joueur capture au moins 25 graines, le jeu s'arrête et le joueur est gagnant.

Respect des exceptions :

Nous avons également pris en compte les coups interdits et les limites du jeu.

- Il est interdit de jouer une case vide.
- Si le champ adverse est vide, il est obligatoire d'y introduire au moins une graine. Le jeu s'arrête si l'on n'en a pas la possibilité.
- Il est interdit de vider le champ adverse en récupérant les graines potentiellement gagnées.
- Si le jeu cycle, la partie s'arrête et chacun récupère les graines de son champ

Fonctionnalités du programme :

- Lancement du jeu grâce au prédicat init/0.
- Choix du type de partie : humain/humain, humain/IA ou IA/IA.
- Possibilité de quitter le jeu à tout moment en connaissant le vainqueur.
- Possibilité de se faire conseiller sur les coups à jouer.

3. Explication des choix de conception

Nous avions deux possibilités : travailler sur un plateau de 2x6 cases (liste composée de 2 sous-listes de taille 6) ou un plateau de 1x12 cases (une liste de 12 éléments).
Nous avons choisi de travailler sur 12 cases, ce qui était plus facile pour distribuer les graines de façon circulaire. Pour distinguer les joueurs, nous distinguons les cases d'indice inférieur à 6, qui appartiennent au joueur 1, et celle d'indice supérieur à 7, qui appartiennent au joueur 2. Cependant, nous avons noté que travailler sur deux sous-listes aurait pu permettre de

réduire le nombre de prédicats en évitant les tests sur les joueurs (et donc la duplication du code pour chaque exécution symétrique).
Pour l'affichage, nous avons systématiquement converti le plateau en 2 listes de 6 cases que nous retournons selon le joueur courant. Cette méthode nous permet de travailler facilement sur 2 lignes pour le plateau. Elle n'a pas un impact trop important sur l'exécution du programme car la conversion est faite au plus une fois par tour de jeu.

4. Explication de la stratégie IA

Notre IA a une prévision à court terme : ses calculs portent uniquement sur son prochain coup. Elle joue la case qui lui rapporte immédiatement le plus grand nombre de points. Si plusieurs coups lui permettent d'obtenir le même nombre de points (notamment si aucun coup ne lui permet d'en gagner), la case à jouer est choisie aléatoirement entre ces coups « équivalents ».
L'avantage de cette méthode est que les coups de l'IA sont moins prévisibles. De plus, cela permet d'avoir des parties différentes à chaque jeu IA contre IA.
Ce jeu IA contre IA nous a d'ailleurs permis de détecter des bugs, et donc d'améliorer notre programme, en nous faisant tomber sur des cas spécifiques sans avoir à jouer.
La difficulté de notre IA est moyenne.
Sa faiblesse est de ne va pas chercher à prévoir ce que le joueur adverse va jouer. En effet, si elle prévoyait les 2 coups suivants le sien (celui de son adversaire puis le sien au prochain tour), cela reviendrait à calculer l'état de 6^3 plateaux dont il faudrait ensuite faire la moyenne des meilleurs scores pour chacun des 6 coups initiaux. Nous avons donc estimé que la difficulté de l'IA était convenable telle quelle sans avoir besoin d'augmenter les besoins en mémoire et en temps de calcul.

5. Problèmes rencontrés et solutions apportées

Au cours de la conception de l'Awalé, plusieurs questions se sont posées. Nous avons cherché à y apporter les meilleures réponses possibles sans rendre le programme trop complexe.

Tout d'abord, nous ne savions pas comment déterminer précisément quand le jeu cycle étant donné que la longueur d'un cycle peut varier. De plus, on peut avoir l'impression de tomber dans un cycle à cause des choix d'un joueur puis le quitter si celui-ci décide de changer de stratégie et de jouer une case en rupture avec ses coups précédents. Pour y remédier, de manière assez naïve, nous avons estimé qu'au bout de 12 coups sans nouveau point marqué, – c'est-à-dire un tour de plateau dans le cas extrême où il ne reste que 2 billes séparées du même nombre de cases vides – la partie stagne et est potentiellement entrée dans un cycle.

Ensuite, pour générer des états de plateau atteignables, notre programme Prolog teste successivement toutes les cases de 1 à 6 en mettant éventuellement à jour le meilleur score : pour cela un prédicat échoue pour toutes les cases sauf la dernière. Mais un problème se pose si la dernière case du joueur est nulle, car ce cas va aussi échouer (en effet

une case à valeur nulle ne peut pas être jouée). L'ensemble des possibilités de jeu va donc s'arrêter sur un *fail* : la partie s'arrête donc alors qu'il est toujours possible de jouer. Pour résoudre ce problème, nous trouvons d'abord l'indice de la dernière case non nulle du plateau et faisons en sorte que le prédicat testant cette case n'échoue pas. Nous avons donc introduit le prédicat *derniere_case_non_nulle*.

6. Conclusion

La réalisation de ce projet nous a permis de mettre en application l'ensemble des notions Prolog vues en cours. Nous avons pu réaliser qu'il est possible de représenter la plupart des problèmes sous forme récursive même si ce n'est pas forcément une solution qui apparaît immédiatement.

Nous avons découvert une manière de programmer complètement différente de ce que nous avons connu jusqu'ici. Le choix d'utiliser Prolog pour ce projet parait cohérent car le jeu nécessite de générer de nouveaux tests automatiquement et Prolog est adapté à ce type de test. Cependant, un autre paradigme de programmation aurait probablement permis d'obtenir un résultat similaire avec moins de ligne de code. Très peu de fonctions utilitaires sont prédéfinies sur Prolog ce qui peut expliquer cette longueur de code.