IA02 : Awalé - Rapport

BAUNE Florian - TALEB Aladin

Résumé

Le code source ainsi que le rapport peuvent être récupérés sur le GitHub
`https://github.com/frozarmony/Awaladin-florIAn`

1 Présentation du jeu

L'Awale dans sa version de base est un jeu qui se joue à 2 joueurs. Chaque joueur possède un plateau, avec 6 trous qui contiennent un certain nombre de graines.

On définit la situation initiale comme suit :
— Les scores des deux joueurs à 0
— 4 graines dans chaque trou de chaque plateau
Les joueurs jouent tour à tour. Voici le déroulement d'un tour :

1. Prise de toutes les graines d'un des trous non vides dans le plateau du joueur et distribution des graines dans le sens anti-horaire à partir du trou suivant le trou source. Le trou source ne doit pas être rempli.
   Une prise est valide si l'adversaire a au moins une graine dans son plateau à la fin de la distribution. Si aucune prise n'est possible, chaque joueur récole le contenu des graines de son plateau.
2. Récolte des graines à condition que :
   — La dernière graine distribuée soit dans le plateau du joueur adverse
   — Le dernier trou rempli contient 2 ou 3 graines.
   Dans ces cas, le joueur récolte les graines dans le trou, ainsi que les graines de trous précédents s'ils respectent également ces conditions
   — Le plateau adverse contient au moins une graine à la fin de la récolte. Si ce n'est pas le cas, on annule toute éventuelle récolte, sans annuler la distribution des graines.
   Les graines récoltées sont ajoutées au score du joueur.

Ces tours s'effectuent jusqu'à ce que :
— L'un des joueurs gagne, c'est à dire que son score est strictement supérieur à la moitié des graines du jeu
— Le plateau n'ait plus de graine, ce qui généralement signifie un jeu nul
— Le jeu soit cyclique, dans ce cas, les joueurs récupèrent les graines de leur camps. Dans les règles officielles, la cyclicité doit être observée et validée par les deux joueurs. Pour notre programme, nous définissons la cyclicité de manière simple et intuitive, quitte à être restrictive : si le jeu "boucle", donc que nous rencontrons un état déjà joué, alors le jeu est cyclique.

2 Hypothèses & Manière de travailler

Lors de notre premier TP de projet, nous avons cherché à comprendre les règles de l'Awalé à la fois par la théorie et par la pratique (il faut bien s'amuser un peu). Puis très vite, notre chargée de TD nous a proposé quelques pistes pour nous aider à démarrer. Ces pistes se présentaient sous la forme de prototypes de prédicats sensés apporter une première pierre à notre édifice. Simplement, nous manquions de recule sur l'architecture globale sous-jacente à ces prototypes. C'est donc dans une optique de clarté et de cohérence que nous avons décidé de faire table rase et de trouver une architecture par nos propres moyens.

2.1 Organisation du travail

De manière générale, nous avons adopté la même méthode pour chaque point critique du projet. Ainsi, nous nous réunissions autour d'un tableau afin de discuter de chacune des difficultés. Puis, au fur et à mesure de nos échanges, nous construisions l'architecture et décidions des meilleurs solutions à apporter à nos problèmes. Enfin, lorsqu'un point de conception était suffisamment détaillé, nous nous répartissions les prédicats à développer et nous donnions rendez-vous quelques jours plus tard pour la fusion et les tests de nos codes respectifs. En somme dans ce projet, nous avons conçu par le haut pour mieux développer et tester par le bas.

2.2 Cahier des charges

Le jeu devra, selon les imposés du projet, permettre d'arbitrer une partie suivant les règles officielles, entre deux joueurs, qu'ils soient humains, assistés par ordinateurs ou virtuels. Nous avons néanmoins souhaité généraliser le jeu en donnant la possibilité de faire des parties aux environnements plus exotiques. Ainsi, il sera possible de jouer avec des plateaux ayant autant de trous et de graines que proposé par l'utilisateur.

Dans cette version, nous limitons tout de même notre généralisation. Il ne sera possible de jouer qu'à deux joueurs, ceux-ci ayant le même nombre de trous dans leur plateau. Les diverses règles de jeux (distribution, capture, etc ...) restent immuables, et n'évoluent pas au cours de la partie.

3 Architecture

3.1 Structures de données

Nous avons basé notre travail sur deux structures de données principales. La première contenant les données sur les joueurs et la seconde représentant l'état d'une partie à un instant donné.

Le `PlayerState` est donc une liste avec deux éléments, chacun de ces éléments est lui-même une liste qui a pour premier éléments le type du joueur (Humain, Assisté ou IA). Le joueur est donc représenté par un type dans une liste afin de pouvoir ajouter des données complémentaires en lien avec le type du joueur. Par exemple pour une IA, cela pourrait être un nombre définissant le niveau de cette IA.

```
PlayerState :
[
    [PlayerType,...],
    [PlayerType,...]
]
```

```
PlayerType :
kHuman, kAssistedHuman, kComputer
```

Le `GameState` est aussi une liste avec cette fois trois éléments : la liste des scores des joueurs, la liste des camps (un camps contenant lui-même une liste de champs) et un entier déterminant le joueur actif (0 pour le joueur 1, 1 pour le joueur 2).

```
GameState :
[
    [Score1,Score2],
    [Board1,Board2],
    PlayerTurn,
]
```

Les champs sont indexés de deux manières :
— En absolu : $[Board1, Board2] = [1e, 2e, 3e, 4e, 5e, 6e], [7e, 8e, 9e, 10e, 11e, 12e]$
— En relatif : $[Board1, Board2] = [1e, 2e, 3e, 4e, 5e, 6e], [1e, 2e, 3e, 4e, 5e, 6e]$

3.2 Fonction principale

La fonction principale est le premier prédicat appelé par l'utilisateur. Il initialise tout d'abord le premier `GameState`, et demande à l'utilisateur quel type de partie il souhaite jouer : Joueur VS Joueur, IA VS Joueur, ... initialisant ainsi le `PlayerState`. Il lance ensuite la boucle principale du jeu qui effectuera les différents tours jusqu'à la fin du jeu. On affiche enfin l'état de fin de partie, ainsi que le gagnant s'il y en a un.

3.3 Initialisation

Le prédicat d'initialisation utilise un état de jeu initial afin de calculer les paramètres du jeu : le nombre total de graines, et le nombre de champs dans chaque camp. Ces paramètres sont stockés via un `assert` afin que tous nos prédicats y aient accès.

3.4 Boucle principale

La boucle principale sert de boucle "tant que" à l'aide du prédicat `repeat`. Elle appelle les tours des jeux successifs, tout en sauvegardant les différents états, jusqu'à ce que le jeu soit considéré comme fini.

3.5 Tour de jeu

Un tour de jeu s'effectue comme suit. On affiche tout d'abord l'état du jeu à l'aide d'un lot de prédicats montrant les scores et les plateaux. On récupère ensuite la liste de toutes les actions possibles et on soumet cette liste à un prédicat se chargeant de fournir l'action choisie. Le choix de cette action dépend du type de joueur :
— Un joueur humain n'aura qu'un simple choix parmi les actions possibles
— Un joueur humain assisté se verra proposé une action par l'IA, mais pourra tout de même choisir celle qu'il souhaite
— Un joueur IA fera la meilleure action selon son algorithme.
L'action choisie passe en argument d'un prédicat se chargeant de l'effectuer en appliquant la distribution puis la récolte.

Lorsque aucune action n'est possible, la liste est unifiée avec la liste vide. Le choix de l'action renvoie alors directement l'action 0, sans consulter l'IA ou le joueur, qui correspond au vidage des plateaux.

A la fin du tour de jeu, le nouvel état est ajouté à la liste des états du jeu.

3.6 Actions

Etant donné que cet algorithme allait être utilisé pour générer les états dans notre intelligence artificielle, nous avons fait notre possible pour réduire au maximum sa complexité.

3.6.1 Actions possibles

Nous définissons tout d'abord un prédicat listant les actions possibles, c'est à dire les distributions que l'on peut effectuer. Une action est considérée comme possible si :

— L'action correspond à un trou non vide
— Si le plateau ennemi est vide, la distribution laisse au moins une graine dans le champ adverse. C'est le cas si

$$Index_{TrouRécolté} + NbGraines_{TrouRécolté} > NbTrous$$

Néanmoins, aucune action n'est possible si le jeu est considéré comme cyclique

Les actions possibles sont ramenés sous forme de liste ayant la structure suivante :

$$X = [0,0,1,1,0,1]$$

où $X(i) = 1$ si l'action i est possible, et $X(i) = 0$ le cas échéant.

On peut ensuite traduire cette liste dans un modèle plus compréhensible par l'utilisateur, sous la forme :

$$X = [3,4,5]$$

3.6.2 Distribuer graines

Concernant la distribution des graines dans le plateau, lorsque le nombre de graines à distribuer est petit, le travail de l'algorithme est assez simple et il suffit d'ajouter récursivement une graine à chaque champs. Cependant lorsque le nombre de graines à distribuer devient plus grand, il faut faire plusieurs tours de plateau. En conséquence, on doit modifier la liste des champs plusieurs fois, ce qui augmente la complexité. Ainsi, nous avons mis au point un algorithme qui au maximum ne parcours qu'une seul fois chaque liste de champs. Son principe est le suivant : on parcours le camps du joueur actif, pendant la descente récursive, on récupère la quantité de graines à distribuer. Puis, lors de la remontée, on met à jours les valeurs des champs en se basant sur une formule arithmétique. Cette formule utilise la quantité de graine à distribué ainsi que l'index relatif du champ pour connaître directement le nombre de graines à lui ajouter,

$$(((NombreDeGraineADistribuer - IndexRelatif) \div (2 * NombreDeChamps - 1)) + 1)$$

. ensuite on parcours le camps du joueur passif si c'est nécessaire en se servant de la même formule. La seule différence étant que le nombre de graines à distribuer est déjà connu.

3.6.3 Récolter graines

Le prédicat de la distribution doit renvoyer le dernier champ rempli. A partir de cette donnée, le prédicat de récolte s'assure tout d'abord que ce dernier champ est bien dans le camp adverse à l'aide de la formule

$$EnemyIndex = (DernierChamp - 1) \div NombreDeChamps$$
$$EnemyIndex \neq PlayerIndex$$

La récolte d'un plateau consiste à le dérouler récursivement jusqu'à arriver au dernier champ rempli. Pour cela, on converti le dernier champ "absolu" en relative au plateau à récolter à l'aide de la formule

$$DernierChampRelatif = ((DernierChamp - 1) \mod NombreDeChamps) + 1$$

On récolte alors le dernier champ si c'est possible selon les règles et on unifie une variable indiquant que la récolte a été faite ou non. Ainsi, si une récolte n'a pas pu être effectuée, les trous précédents ne doivent pas être récoltés.

Il vérifie enfin s'il reste des graines dans le plateau adverse, afin d'être sûr que la récolte ne l'affame pas, et met à jour le `GameState` avec les nouveaux scores et plateaux.

3.6.4 Vider les plateaux

Nous avons besoin dans certains cas de vider les plateaux, par exemple lorsque le jeu est cyclique ou que l'un des joueurs ne peut plus nourrir l'autre. Dans ce cas là, les graines de chaque plateau sont capturés par leur joueur respectif.

Nous définissons donc un prédicat permettant de vider un plateau en rendant tous ses champs nuls, tout en faisant la somme des graines afin de mettre à jour le score.

3.7 IA

Pour réaliser l'IA de ce jeu, nous avons choisi d'implémenter l'algorithme MiniMax qui consiste à générer un arbre comprenant tous les états possibles avant d'en déterminer le meilleur. Cette technique consiste à récupérer la meilleure action, c'est à dire celle maximisant nos gains, tout en supposant que l'adversaire jouera de manière optimale en les minimisant.

Afin de générer, stocker et explorer notre arbre de recherche, nous avons utilisé les `ASSERT` de prolog. Ainsi, chaque noeud fermé de l'espace d'état est représenté comme suit :

```
gameStatesArc(*Identifiant de l'IA*,
*Etat du jeu parent*,
*Profondeur relative de l'état*,
*Liste des états fils*,
*Liste des actions pour arriver aux états fils*).
```

3.7.1 Génération d'arbre

La génération et/ou la mise à jours de l'arbre de recherche se fait en deux étapes :

— la première étape consiste à rechercher l'état courant dans l'arbre et à détruire tous les états antérieurs ou tous les états issus d'états antérieurs. Cette procédure à pour but d'optimiser la mémoire et par la même occasion, réduire le temps de recherche dans la base de connaissance dynamique.

— la seconde étape consiste à générer l'arbre jusqu'à une certaine profondeur (ou rang). Pour ce faire, il suffit de prendre chaque noeud du rang le plus profond et de générer ainsi le rang suivant. Et ainsi de suite jusqu'à arriver à la profondeur souhaitée.

3.7.2 Parcours d'arbre

Une fois l'arbre généré, on le parcours récursivement de bas en haut afin de pondérer chaque noeud suivant l'algorithme MiniMax :

— Les feuilles sont pondérées par une fonction d'évaluation qui sera définie plus tard

— Les noeuds Joueurs sont pondérés par la valeur maximum de leurs fils

— Les noeuds Ennemis sont pondérés par la valeur minimum de leurs fils.

Ce prédicat renvoie ensuite le mouvement permettant d'aller au `GameState` de pondération maximale parmi les fils de l'état actuel.

3.7.3 Fonction d'évaluation

Il ne reste plus qu'à définir la fonction d'évaluation. L'algorithme Minimax cherche le coup maximisant nos gains tout en supposant que l'adversaire tentera de les minimiser. Cette notion de gain peut être représentée par la différence entre les scores :

$$Gain = Score_{IA} - Score_{Ennemi}$$

Ainsi, l'IA s'assurera d'augmenter la différence des scores, donc ses chances de gagner tout en supposant que l'adversaire tentera de minimiser cette différence des scores, et donc de prendre l'avantage.

Nous aurions également pu vérifier si la partie était gagnante ou perdante, et ainsi appliquer des poids plus conséquents, mais nous avons préféré nous contenter d'une fonction d'évaluation très simple à calculer afin de diminuer au maximum son temps de calcul.

3.7.4 Génération et parcours simultanés avec Minimax $\alpha\beta$

Après avoir implémenté notre algorithme, nous avons vite été ramenés à la réalité : la complexité de l'IA était trop grande. La génération et le parcours étant en $o(b^n)$ avec b le facteur de branchement (6 la plupart du temps), et n le nombre de demi-coups, l'IA peinait à prévoir des coups avec plus de 5 demi-coups d'avance.

Après quelques recherches sur le sujet, nous avons découvert l'élagage $\alpha\beta$ consistant à trier à la volée les parties de l'arbre de recherche qui ne modifieront pas la pondération des noeuds.

α et β correspondent aux bornes, respectivement inférieure et supérieure, de l'intervalle dans lequel la pondération d'un noeud doit être afin que celle-ci ait un impact significatif. Ces paramètres se propagent et se mettent à jour durant le parcours de l'arbre en profondeur d'abord.

Il arrive parfois que $\beta \leq \alpha$, et donc qu'aucun noeud ne peut influencer l'algorithme. On pratique alors une coupure de l'arbre afin de ne pas explorer les parties inutiles. Ceci évite de générer inutilement des états de jeu, et de calculer leur fonction d'évaluation.

Nous avons réussi à implémenter cet algorithme, qui s'avère inefficace en début de partie mais très rapide en milieu de jeu. Nous pouvons atteindre des temps de recherches acceptables jusqu'à 12 demi-coups en profitant de la grosse allocation mémoire du programme SWI-Prolog.

4 Modularité & Evolutivité

Notre projet se veut assez souple et modulaire tant au niveau de l'organisation du programme que des possibilités de jeux qu'il offre. En effet, il permet par exemple à l'utilisateur de choisir l'état initiale de la partie, lui permettant ainsi de s'entrainer sur un état particulier ou de réécrire une cuisante défaite. Mais ce n'est pas tout ! Il peut aussi légèrement modifier les règles en changeant le nombre de champs de départ ainsi que le nombre de graines dans chaque champs. Pour ce faire il lui suffit d'utiliser le prédicat comme dans l'exemple suivant (avec 7 champs dans chaque camps) :

```
awale([[0,0],[[4,5,2,7,4,0,9],[2,7,6,1,6,8,9]],0])
```

Au niveau de la structuration du code, nous nous sommes organisés avec plusieurs fichiers :
- `main.pl` : qui contient le prédicat principal et les boucles principales.
- `action.pl` : qui contient le prédicat `doAction` qui est utilisé pour calculer un nouvel état à partir d'une action valide et d'un état parent
- `io.pl` : qui contient toutes les fonctions d'entrées/sorties. Cela pourrait notamment faciliter la traduction de notre programme, ou encore la création d'un autre style d'affichage...
- `IA.pl` : qui contient les fonctions nécessaire au fonctionnement de notre IA.
- `tools.pl` : qui contient toutes les petites fonctions utilitaires dont nous pouvons avoir besoin dans les différents fichiers, comme les prédicats de parcours de liste par exemple.

Tous les affichages de phrases sont assurés par des prédicats indépendants situés dans des fichiers distincts afin de faciliter la traduction. Nous avons pour l'instant deux fichiers contenant les phrases du logiciel

en Anglais (`english.pl`) et en Français (`francais.pl`), interchangeables en modifiant une simple ligne dans le fichier main.

La majeure partie de nos prédicats ont, dans un but de compréhension, des prototypes de la forme

```
%-------
%iaBestAction(IA_ID, CurrentGameState, PrePossibleActions, FinalRank, &BestAction)
%Fonction principale de l'algorithme MINIMAX
%-------
```

Les noms de variable ayant un '&' devant elles ont pour but d'être unifié à la fin de l'appel du prédicat.

Ainsi, notre conception laisse le champ libre à quelques évolutions et améliorations qui seront exposées dans la prochaine partie..

5 Conclusion

5.1 Résultats

Notre programme nous permet bien d'arbitrer une partie entre deux joueurs selon les règles officielles, tout en proposant la possibilité à l'utilisateur de joueur sur d'autres types de plateaux.

L'IA a été conçue afin de déterminer efficacement la meilleure action possible. Ce n'est pas une supercherie, elle cherche réellement à gagner contre l'adversaire. Nous l'avons améliorée petit à petit en espérant battre le niveau expert d'une version de l'Awale trouvée sur le net à l'adresse :

`http://s.helan.free.fr/awale/lejeu/jouer/awale.html`

Notre IA est capable de prévoir 12 demi-coups à l'avance dans des temps relativement acceptables, ce qui fut suffisant pour battre l'IA de ce site à plusieurs reprises.

5.2 Améliorations possibles

Nous sommes totalement conscients que notre programme n'est pas parfait, malgré tout le soin, le temps et la volonté que nous y avons apportés. Voici quelques pistes pour améliorer ce programme :

- Gestion des exceptions pour les entrées de l'utilisateur. Le programme plante si l'utilisateur ne rentre pas correctement des chiffres.
- Affichage optimisé du plateau, les champs n'étant pas prévus pour contenir un nombre à trois chiffres ou plus de graines.
- Charger/sauvegarder une partie, il suffirait de stocker la liste de tous les états puis de la relire en début d'une autre partie.
- Gestion de la difficulté de l'IA, car nous avons pour l'instant implémenté un Dieu de l'Awale dans le programme. Il suffirait de donner la possibilité à l'utilisateur de modifier le nombre de demi-coups explorés par l'IA.
- Amélioration de l'IA en début de partie en lui ajoutant une base de connaissance sur les meilleurs ouvertures de l'Awale (comme aux echecs), évitant ainsi une recherche inutile et couteuse d'un arbre.
- Amélioration du style de l'interface console un peu minimaliste.

5.3 Ce que nous avons appris

De part la structure particulière du Prolog, loin de nos habitudes des langages fonctionnels, nous avons été contraints de prendre beaucoup de recul et de réfléchir astucieusement en amont avant de passer au code.

Ceci fut très bénéfique pour nous en termes de gestion méthodique de projet, que l'on a pas forcément lorsque le langage nous est familier.

Nous avons pu nous rendre un peu plus compte de la puissance de Prolog, mais également de la difficulté à s'y habituer. On regrette assez souvent l'absence de structures conditionnelles et itératives, nous obligeant à créer de nouvelles fonctions spécialement dédiées à simuler ce type de structure.

Néanmoins, ce projet ne nous a pas permis d'exploiter réellement les capacités démonstratives de ce langage. Avec du recul, on se rend rapidement compte que tous nos prédicats ne font qu'implémenter un algorithme fonctionnel en mimant des structures venant de langages comme le C, mais presque aucun ne profite des structures propres au Prolog, comme la création et la recherche d'un arbre à partir d'une base de faits. Nous aurions pu réaliser ce programme aussi facilement, voir plus, avec un langage fonctionnel ; mais pour ce projet, l'intérêt du Prolog peine à se faire ressentir. Le Prolog est clairement supérieur à d'autres langages pour résoudre des problèmes complexes et abstraits en quelques lignes de code, mais n'est visiblement pas fait pour réaliser des jeux ou applications.