Projet d'Intelligence Artificielle : J'ai Awalé de Travers

Mathis Saillot

Avril 2020

1 Introduction

Pour ce projet d'intelligence artificielle, l'objectif est de coder un agent capable de prendre une décision sur le prochain coup à jouer lors d'une partie d'Awalé. Appartenant à la grande famille Mancala, l'Awalé est un jeu combinatoire abstrait d'origine africaine. Deux joueurs s'affrontent avec pour but de ramasser plus de graines que son adversaire à la fin de la partie.

Dans ce rapport je vais commencer par brièvement présenter les règles de l'Awalé ainsi que les contraintes spécifiques à ce projet. Ensuite, je vais parler des prémisses de mon travail, mes différentes recherches et les premières idées pour mon agent. Je vais rapidement parle des solutions utilisant des algorithme de la famille des Monte Carlo Tree Search (MCTS), avec lesquels je n'ai pas eu beaucoup de succès. Et enfin je vais présenter mon travail autour de l'algorithme MinMax. Les différentes optimisations que j'ai implémenté pour pouvoir exploré plus profondément l'arbre de décision, les différentes heuristiques utilisées et comment essayer de trouver les valeurs optimales pour ces dernières, dans le but d'aboutir à mon agent final : J'ai Awalé de Travers.

Sommaire

2 Les débuts du projet

2.1 Les règles de l'Awalé

Avant de commencer ce projet je ne connaissais le jeu de l'Awalé que de nom, et n'y avait jamais joué. La première étape pour moi a donc tout naturellement consisté en l'apprentissage des règles de ce jeu. L'Awalé est un jeu à information complète, sans hasard - comme les échecs et le go - il en existe de nombreuses variantes, et il possède de nombreux noms différents. Les règles utilisées pour le projet sont basés sur la variante la plus répandue.

Le jeu se joue sur un plateau de 12 trous répartis en deux rangées représentant le camps de chaque joueur. Au début de la partie, 4 graines sont placées dans chacun des trous, soit un total de 48 graines. Ensuite, chacun leur tour, les joueurs vont choisir un trou de leur camps, ramassés toutes les graines qui y sont présentes et les déposer une par une dans les trous suivants en tournant toujours dans le même sens. Si il y avait plus de 11 graines dans le trou choisi, le joueur n'y dépose pas de graine et continue normalement dans les trous suivant jusqu'à ce que sa main soit vide. À la fin de l'égrenage, si le trou recevant la dernière graine est situé dans le camp adversaire et qu'il contient alors exactement 2 ou 3 graines, le joueur les ramasse, puis si le trou précédent est aussi un trou adverse et qu'il contient 2 ou 3 graines elles sont ramassées également. On procède ainsi jusqu'à arrivé dans notre propre camps ou dans un trou contenant un nombre de graines différent. Les graines ainsi amassées sont retirées du jeu et constituent les points du joueur. Le premier joueur ayant 25 points ou plus met fin à la partie et gagne. Une égalité est possible si les deux joueurs ont 24 points à la fin du jeu.

À cela, deux règles spéciale s'ajoutent :

- si un coup devait permettre à un joueur de ramasser toutes les graines présentes dans le camps adversaire, le coup est autorisé mais aucune graine n'est prise.
- si l'adversaire n'a plus de graines dans son camps, le joueur est obligé de jouer un coup donnant au moins une graine à son adversaire. Si aucun des coups ne le permet, alors le joueur ramasse toutes les graines présente de son côté et la partie prend fin.

2.2 Les contraintes du projet

Dans le contexte de ce projet, différentes contraintes sont imposées, à la fois informatiques mais aussi sur les règles de l'Awalé, qui sont légèrement modifiées. En plus des règles énoncées plus haut, une partie s'arrête dès qu'il reste 6 graines ou moins sur le plateau de jeu. Les graines restantes ne sont attribuées à personne, le score de chaque joueur correspond au total des graines amassées jusque là.

Pour évaluer les agents rendus par tous les groupes d'étudiants, un tournoi est organisé. Chaque agent a le droit à une heure d'apprentissage lors de son instanciation. Une fois tous les agents instanciés, le tournoi commence : chaque agent affronte chaque autre agent dans une série de 100 manches. Chaque manche consiste en deux parties d'Awalé où le premier joueur est alterné. Le score obtenu lors de ces deux parties est additionné pour former le score de la manche. Le joueur avec le plus grand score remporte la manche et gagne ainsi 3 points. En cas d'égalité les deux joueurs marquent 1 point. Le gagnant du tournoi est l'agent ayant obtenu le plus de points.

Le langage de programmation imposé est Java, le moteur du jeu de l'Awalé est déjà codé, et les agents participant au tournois doivent hériter de la classe abstraite *CompetitorBot*. Pour chaque tour de jeu, l'agent dont c'est le tour a 100 millisecondes maximum pour donner sa décision sous la forme d'un tableau de 6 poids correspondant aux 6 trous du joueur. Le trou avec le plus gros poids associé est choisi pour être le coup à jouer, en cas d'égalité un coup aléatoire est choisi parmi ceux ayant les poids maximum dans le tableau. En plus de la contrainte de temps, une contrainte de mémoire utilisée par l'agent est imposée : la limite par agent est à 64 Mio.

Par soucis d'équité, le multithreading est interdit, ainsi que tout autre technique de calcul partagé. Il faut donc être attentif à l'optimisation de son code, 100 millisecondes cela passe très vite pour prendre une décision aussi complexe que jouer à l'Awalé. Il est également interdis d'effectuer de l'apprentissage avant le tournoi, tout doit se faire en moins d'une heure lors de l'instanciation. C'est avec toutes ces contraintes en tête que j'ai commencé mes recherches pour essayer de trouver ce qui a déjà été fait et qui pourrait être utilisé pour mon agent.

2.3 Awalé et Intelligence Artificielle

Il faut savoir que l'Awelé est un jeu qui a été résolu en 2002 par Romein and Bal [10] en déterminant le score des 889.063.398.406 positions possibles du jeu. Ils ont ainsi prouvé que quelque soit le joueur qui commence, si les deux jouent parfaitement, le jeu se termine par une égalité. Un agent jouant parfaitement à l'Awalé a alors été créé, il possède une base de donnée contenant toutes les positions du jeu, avec leur score ce qui lui permet de toujours choisir le meilleur coup. Malheureusement, je n'ai pas réussi à trouver l'interface web [10] qui devait permettre de l'affronter et qui semble avoir disparue. La base de donnée à aussi été utilisée afin d'analyser les parties du tournoi tenu à la Computer Olympiad en 2000 [9], durant lequel les deux meilleurs agents jouant à l'Awalé (MARVIN et SOFTWARI) se sont affrontés [7].

Les deux programmes possédaient une base de donnée avec toutes les positions contenant 34 graines ou moins. Leur jeu était donc optimal une fois ce nombre de graine atteint. Les deux programmes utilisaient un opening book [8], à savoir des stratégies pour le début de la partie construit aux début du tournoi. D'après l'analyse de Romein and Bal [9], les deux programmes, sur l'ensemble des parties, ont joués les coups optimaux environ 85% du temps.

L'utilisation de base de donnée contenant les situations de jeux m'a paru peu réalisable au vu des contraintes de mémoire et de temps d'apprentissage imposées pour ce projet. Par contre l'utilisation d'un opening book m'a beaucoup intéressé, et l'algorithme utilisé par Lincke [8] me semblait accessible. Je n'ai malheureusement jamais pris le temps de l'implémenter car je me suis concentré sur d'autres stratégies pendant le reste du projet. J'ai cependant retenu autre chose concernant le jeu optimal : le seul coup optimal pour le premier joueur est le trou le plus à droite. Le deuxième joueur a alors trois coups optimaux possible le deuxième, cinquième et sixième trou dans le sens de rotation du jeu. Tout autre coup provoque une défaite si l'adversaire joue de façon optimale. Bien entendu, pour ce projet aucun agent ne sera capable de joué de façon optimale, mais respecter ces premiers coups pourrait donner un léger avantage dès le début de la partie.

2.4 Monte Carlo Tree Search (MCTS)

En théorie des jeux, un arbre est souvent utilisé pour représenter les différents états ainsi que les coups permettant de passer d'un état de jeu à un autre. Dans le cas de l'Awalé, l'arbre est assez facile à formaliser : chaque noeud représente une situation de jeu - le nombre de graines dans chaque trou, le score de chaque joueur et qui est le joueur actif - et les branches de l'arbre indiquent qu'un coup légal permet de passer d'un état à une autre. Pour l'Awalé, l'arbre possède une seule racine, le jeu commence toujours avec la même répartitions de graines. Les feuilles de l'arbre correspondent à des situations de jeux qui marquent la fin de la partie, par la victoire de l'un des joueurs ou sur une égalité. Toute partie d'Awalé peut être représentée par un parcours en profondeur depuis la racine jusqu'à une feuille de l'arbre.

Utiliser un algorithme d'exploration d'arbre afin de prendre une décision me semble donc être un choix judicieux pour ce projet. Le terme MCTS représente une famille d'algorithme de décisions qui construisent un arbre par itération successives et de façon asymétrique. Une politique de sélection est utilisée à chaque itération pour trouver le prochain noeud à étendre, puis une simulation est effectuée à partir de ce noeud et le résultat de la simulation est utilisée pour mettre à jour les informations de l'arbre. La simulation est effectuée en accord avec une deuxième politique. À partir de ce principe, de nombreux algorithmes ont été créés, plus ou moins performants, répondant à des

besoins spécifiques à certains domaines d'expertises ou au contraire essayant d'être le plus général possible [1]. Suite à mes recherches, j'ai sélectionné trois types d'algorithme MCTS que j'ai essayé d'implémenter pour ce projet :

- la version la plus répandue [6] du MCTS l'Upper Confidence Bound for Trees (UCT)
- l'algorithme Playout Policy Adaptation with move Features with Memorization (PPAFM) [2]
- un algorithme combinant un réseau de neurones et MCTS, inspiré par le travail de Silver et al. [11] sur AlphaGo le programme ayant vaincu Lee Sedol au jeu de Go

Malheureusement, je n'ai pas réussi à obtenir les résultats espérés avec ces algorithmes, je pense notamment que c'est lié à la contrainte de temps de prise de décision. Pour l'UCT, la politique de simulation est répartition uniforme aléatoire. Il faut effectuer un très grand nombre de simulations afin d'avoir un bon résultat. Ce qui n'est pas vraiment compatible avec le temps limité imposé. Pour l'algorithme PPAFM ça a été encore pire, il s'est avéré qu'il est beaucoup plus coûteux en terme de calculs, le rendant totalement inutilisable en un temps si court.

Pour le dernier algorithme que j'ai essayé d'implémenter se basant sur un MCTS, c'est le réseau de neurone qui m'a posé problème. Dans cette méthode, la politique de simulation est remplacé par un appel au réseau de neurone afin d'estimer le score à la fin de la partie si le noeud est atteint. J'ai décidé d'utiliser un Multi Layer Perceptron (MLP) avec 14 neurones d'entrée, un pour chaque trou, plus le score des joueurs, 8 couches cachées de 64 neurones et 7 neurones de sortie, soit un neurone de sortie pour chaque coup possible plus le neurone d'estimation du score. La fonction d'activation utilisée est ReLU, avec un dropout de 0.5 lors de l'apprentissage qui se fait par rétro-propagation du gradient. Malgré mes recherches, je n'ai jamais réussi à entraîner correctement le MLP car au bout d'un certain temps d'entraînement, le réseau de neurone se corrompt et certains poids prennent des valeurs infini ou NaN. J'ai donc décidé de ne pas plus insister sur cet algorithme pour me concentrer sur le MinMax.

3 MinMax

3.1 Principe général

L'algorithme MinMax ou MiniMax est un algorithme d'exploration d'arbre en profondeur d'abord. Mais contrairement aux algorithmes de type MCTS, l'arbre doit être étendu jusqu'à la racine afin de pouvoir évaluer les noeuds. L'algorithme MinMax est particulièrement bien adapté aux jeux à deux joueurs à somme nulles comme l'Awalé. Chaque joueur cherche à maximiser son score, ce qui revient à vouloir minimiser celui de l'adversaire. Dans le cas de l'Awalé, il est possible de modéliser cela en donnant 1 point au gagnant de la partie, -1 au perdant et 0 en cas d'égalité entre les deux joueurs. Ainsi à la fin de la partie la somme des scores des deux joueurs sera toujours nulle.

Dans l'algorithme MinMax, l'arbre exploré est une alternance de noeuds min et de noeuds max, chacun représentant l'un des deux joueurs. Le joueur dont c'est le tour est le joueur max, la racine de l'arbre représente la situation de jeu actuelle. Le but est d'associer à chaque noeud le score que peut obtenir le joueur max à la fin de la partie si la situation venait à être rencontrée. Le score de chaque noeud est calculé grâce à l'équation suivante [4] :

$$f(n) = \begin{cases} score(n), & \text{si n est une feuille} \\ max\{f(c) | c \text{ est un noeud fils de } n, & \text{si n est un noeud Max} \\ min\{f(c) | c \text{ est un noeud fils de } n, & \text{si n est un noeud Min} \end{cases} \quad (1)$$

Grâce à l'équation 1, on voit que le score de chaque noeud est calculé en fonction de celui de ses fils, sauf pour les feuilles où l'évaluation se fait grâce à $score(n)$. Cela veut dire que l'arbre doit être exploré en entier avant de pouvoir donner un score à la racine. Ce n'est bien entendu pas réalisable

en temps raisonnable dans le cadre de ce projet. C'est pour cette raison que l'algorithme MinMax n'est jamais utilisé tel quel, mais plutôt une variante permettant de réduire le nombre de noeuds à explorer afin d'obtenir un résultat sous une contrainte de temps d'exécution. La variante la plus utilisée est l'algorithme AlphaBeta [5] associé à une limite de la profondeur de l'arbre.

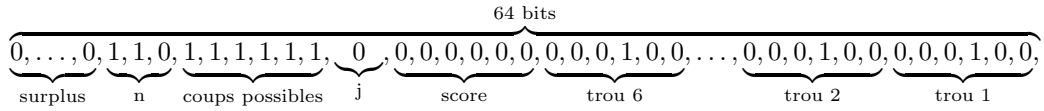
```
1  AlphaBeta(Noeud : Node, α : réel, β : réel, EstMax : bool, Profondeur : entier) : réel
2      si EstTerminal(Noeud) ou Profondeur = ProfondeurMax alors
3          retourner Evaluation(Noeud)
4      si EstMax alors
5          valeur ← −∞;
6          pour chaque Fils : NoeudFils de Noeud faire
7              res ← AlphaBeta(NoeudFils,α, β, false, Profondeur + 1) ;
8              si res > valeur alors
9                  valeur ← res;
10                 si valeur ≥ β alors
11                     retourner valeur
12             si valeur > α alors
13                 α ← valeur;
14     sinon
15         valeur ← +∞;
16         pour chaque Fils : NoeudFils de Noeud faire
17             res ← AlphaBeta(NoeudFils,α, β, true, Profondeur + 1) ;
18             si res < valeur alors
19                 valeur ← res;
20                 si valeur ≤ α alors
21                     retourner valeur
22             si valeur < β alors
23                 β ← valeur;
24     retourner valeur
```

Algorithme 1 : AlphaBeta avec limite de profondeur

3.2 Alpha-Beta

Cette variante du MinMax permet de couper des branches de l'arbre lors de l'exploration sans changer le résultat de l'évaluation. L'algorithme (1) maintient deux valeurs, α et β qui représentent respectivement le score minimum que peut avoir actuellement le joueur Max et le score maximum que peut avoir le joueur Min. À partir du moment où le score maximum du joueur Min devient supérieur au score minimum du joueur Max, l'exploration du noeud en cours peut être abandonné, car il ne sera plus possible d'améliorer le score du joueur Max en continuant.

L'efficacité de l'algorithme dépends grandement de l'ordre dans lequel les noeuds sont explorés. Dans le pire des cas il visite tous les noeuds, ce qui est équivalent à l'algorithme MinMax. Malheureusement, appliquer uniquement l'algorithme Alpha-Beta ne suffit toujours pas pour l'Awalé. Il faut arrêter l'exploration de l'arbre une fois une certaine profondeur p atteinte, dans le but de limiter le nombre de noeuds à explorer et ainsi pouvoir prendre une décision dans le temps impartit. Mais cela à un énorme désavantage, l'algorithme qui était exact demande maintenant l'utilisation d'une heuristique, ce qui peut fausser le résultat du MinMax. La performance de l'agent dépends donc maintenant de la capacité de l'heuristique à estimer la valeur des noeuds, ainsi que de la profondeur à laquelle l'algorithme arrête l'expansion de l'arbre.



n: nombre de coups possibles j: joueur actif

Figure 1: Répartition des données sur les 64 bits du *long*

3.3 Optimisations

Parmi les exemples d'agents contenus dans les sources du projet, l'un d'entre eux est basé sur un MinMax avec une profondeur limite et l'heuristique suivante : l'évaluation d'un noeud correspond au score du joueur Max sur ce noeud moins le score du joueur Min. Cette heuristique, bien que basique permet d'obtenir de très bons résultats, notamment lorsque la profondeur est augmentée. En activant la coupe Alpha-Beta, la profondeur maximale pouvant être atteinte et permettant de respecter la limite de 100 ms par coups est 8. Dans la suite du projet, cette heuristique sera mentionnée sous le nom de *DiffScore* et le bot l'utilisant : *MinMaxDiffScore*.

Afin de coder un agent capable de battre *MinMaxDiffScore*, je voulais gagner en profondeur pour l'Alpha-Beta. Pour cela, j'ai décidé de ne pas utiliser les classes fournies dans le projet permettant de simuler l'Awalé, mais d'écrire mon propre moteur du jeu. J'ai donc réfléchis à une structure de donnée légère permettant de faire des calculs plus rapides pour pouvoir représenter un état de jeu de l'Awalé. Après plusieurs itérations et ajustement, la version finale de ma structure de donnée est constitué d'un tableau contenant deux *long*. En Java, un *long* est un type primitif permettant de stocker un entier sur 64 bits, soit deux fois plus de bits que le type couramment utilisé pour stocker les entiers en Java : *int*. Chaque *long* dans le tableau est utilisé de la même façon afin de représenter les informations spécifiques à chaque joueur (Figure 1). Dans ma structure de donnée, les deux joueurs sont numérotés 0 et 1, et c'est toujours le joueur 0 qui commence.

Le nombre de graine de chaque trou ainsi que les scores des joueurs sont chacun codé sur 6 bits, ce qui permet de représenter des valeurs de 0 à 63, ce qui est suffisant puisque qu'il y a 48 graines dans le jeu. Le $43^{\text{ème}}$ bit représente le joueur actif, seulement le *long* du joueur 0 est utilisé pour stocker cette information, il n'est pas utilisé sur le *long* du joueur 1. Les 6 bits suivants représentent 6 booléens indiquant si le coup correspondant est jouable. Par exemple $0, 0, 0, 0, 0, 1$ indique que le joueur ne peux jouer que depuis le trou 1. Les 3 bits suivant codent le nombre de coups légaux que le joueur peut effectuer. Ce nombre est redondant avec les 6 bits précédents, mais il permet d'accélérer les calculs lors de la manipulation de la structure de donnée. Seulement 52 bits sur les 64 sont utilisés, il reste donc de la place pour potentiellement ajouter d'autres informations.

Dans mon projet, la classe *AwariLong* est dédiée à la manipulation de cette structure de donnée. Elle permet de faire toutes les actions principales nécessaires à la simulation de l'Awalé comme récupérer le score d'un joueur, jouer un coup, connaître si une situation de jeu est terminale, ou savoir le nombre de graines présentes dans un trou. Pour accéder et modifier efficacement ces informations, j'ai utilisé les opérateurs bit à bit de Java, à savoir le *and* (&), le *or* (|), le *xor* ($\wedge$) et les bitshifts ($\ll$ et $\gg$). Tous les algorithmes que j'ai implémenté pour se projet se basent sur cette structure de donnée pour faire leurs calculs - ou sur *Awari*, la version antérieure à *AwariLong* qui utilisait un tableau de trois *int*.

3.4 Profondeur Améliorée et Classement des Coups

Profondeur Améliorée

Afin d'améliorer les performances de l'Alpha-Beta, j'ai décidé d'apporter une modification au système de profondeur classique. En effet, lors de l'appel à l'algorithme, il faut préciser la profondeur maximale que l'on souhaite lui imposé. Cette profondeur a été choisi en fonction de tests afin de respecter la limite de temps de prise de décision. Or, le temps de prise de décision moyen sur une partie d'Awalé peut prendre des valeurs assez différentes d'une partie à l'autre. Entre le début de la partie et la fin, le nombre de noeuds visités diffère énormément. En utilisant un Alpha-Beta à une profondeur de 12 avec l'heuristique *DiffScore*, une exécution de l'algorithme peut entraîner l'exploration d'un arbre de plusieurs millions de noeuds en début de partie à moins de quelques milliers en fin de partie. Cela s'explique facilement par le fait que le *branching factor* évolue en fonction de la partie. Moins il y a de graines sur le plateau, moins les joueurs auront de choix sur le coup à jouer ensuite. Une profondeur maximale de 12 n'a donc pas du tout la même signification au début de la partie et à la fin.

Une solution serait de choisir une profondeur en fonction du nombre de graines encore présentes sur le plateau, mais ce n'est pas très précis et cela demanderait beaucoup de tests pour trouver les bonnes valeurs de profondeur associées au bon nombre de graines. J'ai donc modifié l'algorithme AlphaBeta en transformant la notion de profondeur maximum en notion de budget en nombre de noeuds. À chaque noeud exploré, le budget diminue de 1 et une variable globale comptant le nombre de noeuds explorés est incrémentée. Lors de l'appel récursif, le budget est divisé en fonction du nombre de fils du noeud actuel et d'un hyperparamètre $\delta > 1$. Ce paramètre est choisi en fonction de l'efficacité de la coupe AlphaBeta, plus le nombre de branche coupée est grand, plus δ doit être grand. Si δ est trop grand, ou que la coupe AlphaBeta élimine moins de branches que prévus, le budget initial risque d'être dépassé. Il faut donc choisir un budget et une valeur de δ permettant de respecter la contrainte de temps du projet. Ces paramètres dépendent surtout de l'heuristique utilisée pour évaluer les noeuds.

Grâce à cet algorithme, j'arrive à mieux stabiliser le nombre de noeuds explorés à chaque tour. J'ai ainsi un meilleur contrôle sur le temps de décision de l'algorithme. Cela me permet d'utiliser au mieux le temps disponible pour explorer un maximum de noeuds, à la fois au début et à la fin d'une partie d'Awalé.

Classement des Coups

L'efficacité de l'algorithme AlphaBeta - à savoir le nombre de branches coupées - est dépendant directement de l'ordre dans lequel les fils sont explorés lors du parcours en profondeur. Si l'algorithme les parcours du plus mauvais au meilleur, aucune coupe AlphaBeta ne peut avoir lieu, et on est alors dans le cas d'un algorithme MinMax classique. Si au contraire, les noeuds sont traités dans l'ordre inverse, alors un nombre minimal de noeuds est exploré. Bien entendu si cet ordre été connu, l'algorithme n'aurait pas besoin d'être exécuté. Dans l'algorithme de base, il n'est pas précisé l'ordre dans lequel les noeuds sont explorés, cela dépends surtout de la structure de donnée utilisée et de comment les informations y sont stockées et accédées. Dans le cas de l'Awalé, l'ordre logique est de commencé par le fils correspondant à un coup joué sur le trou le plus à gauche, puis sont voisin de droite et ainsi de suite. Si on considère que chaque trou est un coup optimal le même nombre de fois dans l'ensemble de tous les coups possible dans l'Awalé, alors cet ordre ne donnera pas de meilleurs résultats en moyenne que de choisir un ordre aléatoire.

Le but est donc d'essayer de trier les différents coups possibles pour savoir dans quel ordre il faut explorer les noeuds. Pour ce faire, il est possible d'utiliser une autre heuristique, mais cette solution ne m'a pas intéressé car elle me semble trop coûteuse et difficile à mettre en place. À la place, j'ai décidé de m'inspirer de l'algorithme PPAFM [2] pour créer une hybridation avec l'algorithme AlphaBeta.

```
1  AlphaBetaTriCoups(Noeud : Node, α : réel, β : réel, EstMax : bool, Profondeur : entier)
   : réel
2      si EstTerminal(Noeud) ou Profondeur = ProfondeurMax alors
3          retourner Evaluation(Noeud)
4      si EstMax alors
5          valeur ← −∞;
           /* On effectue le tri des fils en fonction des scores des catégories
              des coups associés                                                  */
6          TriDesFils(Noeud);
7          i ← 0;
8          pour chaque Fils : NoeudFils de Noeud faire
9              res ← AlphaBetaTriCoups(NoeudFils,α, β, false, Profondeur + 1) ;
10             si res > valeur alors
11                 valeur ← res;
                   /* Quand la valeur du noeud est mise à jour, on met à jour les
                      scores                                                      */
12                 AjoutScore(Categorie(Coup(Noeud, NoeudFils)), i);
13                 pour chaque Fils : F de Noeud avant NoeudFils faire
14                     RetireScore(Categorie(Coup(Noeud, F)), 1);
15                 si valeur ≥ β alors
                       /* Une coupe AlphaBeta a lieu, on met à jour les scores si
                          besoin                                                  */
16                     AjoutScore(Categorie(Coup(Noeud, NoeudFils)), i);
17                     pour chaque Fils : F de Noeud avant NoeudFils faire
18                         RetireScore(Categorie(Coup(Noeud, F)), 1);
19                     retourner valeur
20             si valeur > α alors
21                 α ← valeur;
22             i ← i + 1;
23     sinon
           /* Les mêmes modifications sont apportées dans le cas du joueur Min  */
24     retourner valeur
```

Algorithme 2 : Alpha-Beta avec tri sur l'ordre des noeuds

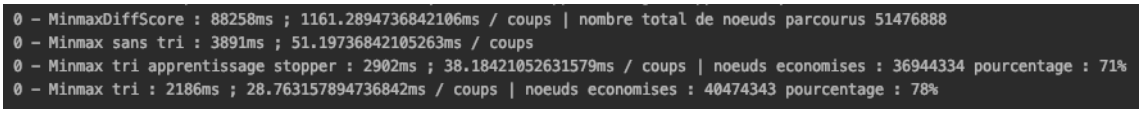


Figure 2: Illustration de la différence de temps d'exécution de l'algorithme AlphaBeta à une profondeur limite de 12

Dans un premier temps, chaque coup est associé à une catégorie de coup, suivant ses caractéristiques. Chaque catégorie possède un certain score, ce dernier permet ensuite de classer les noeuds de catégories différentes par ordre de priorités. Pour connaître la catégorie auquel un coup appartient, l'équation 2 est utilisée. Les coups sont ainsi classés en fonction du trou de départ du coup, du nombre de graines dans ce trou et du nombre de graines dans le trou d'arrivée du coup. Le trou d'arrivée n'a pas besoin d'être inclut dans l'équation car il peut être déduit grâce au trou de départ et au nombre de graines dans le trou. Le joueur dont c'est le tour n'a pas d'importance, le plateau étant symétrique. J'ai essayé d'ajouter à l'équation le nombre total de graines présentes sur le plateau, mais le gain lié à cette amélioration n'a pas suffit à compenser l'augmentation du temps de calcul lié a cet ajout. Le trou de départ étant pris en compte dans l'équation, chaque coups légal à une certaine situation de jeu est forcément classé dans une catégorie différente de tous les autres.

$$categorie(c) = TrouDepart(c) + 6 \times GraineTrouDepart(c) + 6 \times 48 \times GraineTrouArrive(c) \quad (2)$$

Pour savoir quel score donner à chacune des catégories de coups, il faut procéder à un apprentissage. L'idée principale derrière cet technique est que l'apprentissage se fait en même temps que l'AlphaBeta. L'algorithme 2 explique son fonctionnement général. Le tableau de score est initialisé avec des score de zéro. Puis au fur et à mesure des parties, les scores sont mis à jour afin de correspondre au mieux à l'ordre optimal des noeuds.

L'efficacité des optimisations peut être observé sur la Figure 2. La première est l'algorithme MinMax fourni avec le projet, les trois lignes suivantes sont des algorithmes utilisant la nouvelle structure de donnée. Avec la version trié de l'algorithme sur la dernière ligne, on voit que le temps de prise de décision est de 30ms environ, contre 1160ms pour *MinMaxDiffScore* ce qui est presque 40 fois plus rapide. On peut voir aussi que le nombre de noeuds économisés grâce au tri est de l'ordre de 80%. L'algorithme AlphaBeta avec tri sur les noeuds et l'algorithme de profondeur amélioré sont compatible et peuvent être combinés. C'est ce que j'utilise comme algorithme de MinMax pour mon agent.

3.5 Recherche d'Heuristiques

Le deuxième paramètre du MinMax qui a un impact sur les performances de l'algorithme, en plus de la profondeur de recherche, est l'heuristique servant à évaluer une situation de jeu. L'heuristique est sans doute plus importante même que la profondeur [3]. Tout au long du projet, j'ai cherché à implémenter plusieurs types d'heuristiques afin de les comparer pour ensuite garder la meilleure. Toutes les heuristiques que j'ai testé sont construites de la même manière ou presque : un vecteur de *features* est constitué à partir de la situation de jeu, ce vecteur est ensuite multiplié par un vecteur de poids afin de constituer l'évaluation du noeud. Les features sont sensées représenter des éléments importants liés à des stratégies de jeu.

Heuristique 1 La première heuristique que j'ai codé est une simple version modifiée de *DiffScore*. Les deux *features* sont mon score et celui de l'adversaire, mais les poids peuvent être différents de 1

```
Rangs :
1. MinMaxH3.2 : 20.63
2. MinMaxH4 : 16.67
3. Awariness : 16.29
4. MinMaxH6 : 15.53
5. MinMaxH1 : 14.57
6. MinMaxH5.3 : 12.88
7. MinMaxH2 : 7.18
8. MinMax : 2.52
9. Le crocodile crétin : 1.19
```

Figure 3: Les résultats d'un tournois entre les différentes Heuristiques

et -1.

Heuristique 2 Pour ma deuxième heuristique, j'ai décidé d'ajouter 2 nouvelles *features* par joueur. La première est le nombre de trou possédant moins de 3 graines. Ces trous sont importants d'après moi car ce sont les seuls trous permettant à l'adversaire de marquer des points. La deuxième *feature* est directement liée à la première, elle représente la plus grande séquence de trous consécutifs contenant moins de 3 graines. Les séquences de trous de 1 ou 2 graines sont parfait pour marquer un maximum de points.

Heuristique 3 La troisième heuristique est celle m'ayant donné les meilleurs résultats pendant longtemps (Figure 3). Elle reprends toutes les *features* précédentes et en ajoute une seule par joueur : le nombre de graine dans les trous en position de krou de chaque joueur. À l'Awalé, un krou est un trou possédant 12 graines ou plus. Lorsqu'il est joué, le joueur va effectué un tour complet avant d'aller potentiellement placée sa dernière graine chez l'adversaire. Un krou joué au bon moment permet de marquer beaucoup de points.

Heuristique 5 Je ne mentionne pas la 4 car c'est une copie conforme de la 3 mais avec des poids différents. Pour cette heuristique, j'ai voulu essayé quelque chose de différent qui n'a pas du tout fonctionné. Les *features* de l'heuristique 3 sont d'abord soustraites aux mêmes *features* mais calculée sur la racine du MinMax. Puis le résultat de la différence est multiplié par un vecteur de poids. L'idée été d'ajouté un effet de contexte à l'heuristique, mais les résultats n'ont pas été très bon.

Heuristique 6 Cette heuristique se base sur le travail de Daoud et al. [3], à savoir un vecteur de 12 *features*. Mais je n'ai pas réussi à avoir de très bon résultats avec cette heuristique, je ne m'attarde donc pas dessus. J'ai néanmoins bien aimé deux *features* en particulier : le nombre de trou vide et le nombre de trou possédant plus de 12 graines (qui est légèrement différent du nombre de graines dans les krous de l'heuristique 3).

Pour chacune de ces heuristiques, il me faut un vecteur de poids. Plutôt que d'essayer de le remplir au hasard, j'ai décidé de faire appel à un algorithme génétique. Dans un premier temps, j'ai commencé par optimiser les poids en dehors du temps d'apprentissage et de reporté ensuite les poids en dur dans le code. Je savais pertinemment que ces bots n'allaient pas être valide pour le tournoi, mais le but été surtout de tester les différentes heuristiques et comment réussir un algorithme génétique efficace en 1 heure seulement. La Figure 3 montre les résultats d'un tournoi opposant toutes les heuristiques après leur apprentissage. On peut voir que toutes se classent mieux que Le crocodile crétin, le champion de l'année dernière et que le *MinMaxDiffScore* arrivé $8^{\text{ème}}$.

3.6 Apprentissage Génétique

La dernière étape afin d'avoir un agent complet, est de réaliser l'apprentissage génétique pendant l'heure prévue à cet effet et non en amont. Toutes les heuristiques de la section précédente ont eu un apprentissage qui a duré plusieurs heures, s'entraînant contre les autres heuristiques. En me

basant sur les travaux de Daoud et al. [3], j'ai implémenté l'algorithme suivant pour l'apprentissage de l'heuristique :

1. Initialisation aléatoire d'une population de I individus. Les gènes des individus représentent le tableau de poids à optimiser pour l'heuristique. Les poids sont entiers et tirés au hasard entre une borne min et max. L'un des individus est initialisé avec des valeurs naïves correspondant à un choix humain de valeur de poids pour l'heuristique en question. Cet individus est considéré comme le meilleur de la population 0.

2. C individus sont sélectionnés pour être les combattants de la génération en cours. Le meilleur individus de la génération précédente est toujours sélectionné pour faire partie des combattants.

3. Le fitness de chaque individus est évalué lors d'une série de m matchs contre chaque combattant. Chaque victoire rapporte 200 points, une égalité 100 points et le score à la fin de la partie est également ajouté. L'individu affronte aussi un "boss", un combattant spécial qui n'est pas lié à la population et qui rapporte deux fois plus de points.

4. Le meilleur individu de la population précédente est lui aussi réévalué de la même façon.

5. La population est triée en fonction du fitness des individus et le meilleur individu est remplacé si son fitness est moins bon que celui du premier

6. La meilleure moitié de la population est sélectionnée afin de générer la population suivante. Deux individus sont choisis au hasard parmi le groupe sélectionné. Leur gène sont répartis aléatoirement entre leur deux descendants. Cette opération est recommencée jusqu'à ce que une nouvelle population de I individus soit créée

7. Chaque nouvel individu a un certain pourcentage de chance p de faire muté l'un de ses gènes. Si un individus est mutant, un gène est choisi au hasard ainsi que sa nouvelle valeur entre les bornes min et max.

8. Le processus recommence avec la nouvelle génération à l'étape 2. jusqu'à ce que le budget temps soit épuisé.

3.7 J'ai Awalé de Travers

L'agent que j'ai décidé de faire combattre lors du tournoi utilise l'algorithme AlphaBeta avec Tri des Coups et la Profondeur Améliorée. L'heuristique se base sur un vecteur de 8 *features*, soit quatre pour chacun des joueurs. Les quatre *features* sont : le score, le nombre de trou vide, le nombre de trou avec 1 ou 2 graines et le nombre de trous avec 12 graines ou plus. Les poids pour le score du joueur et le score de l'adversaire sont fixés à 10 et -10. Les 6 autres poids sont appris pendant l'apprentissage grâce à l'algorithme génétique dont les paramètres sont les suivants :

- une population de $I = 42$ individus
- les bornes pour les poids sont min $= -20$ et max $= 20$
- le nombre de combattants $C = 5$ et le nombre de match joués $m = 2$
- le taux de mutation $p = 0.2$
- le "boss" est un AlphaBeta avec Tri des Coups de profondeur 8 utilisant *DiffScore*

Une fois l'apprentissage des poids effectué, 100 parties sont jouées afin d'entraîner le score des catégories de coups. Le budget pour la profondeur améliorée est de 450000 noeuds, et $\delta = 1.5$. J'ai Awalé de Travers joue toujours son premier coup depuis le trou le plus à droite.

4 Conclusion

Depuis le début de mes études en informatique, le domaine de l'intelligence artificielle m'a toujours attiré et fasciné, en particulier appliqué à la théorie des jeux. Ce projet m'a vraiment beaucoup plu et je me suis investi à fond pour essayé de produire le meilleur agent possible. Même si je suis satisfait de ce que je rend, il y a deux choses que je n'ai pas eu le temps de faire et qui aurait pu l'améliorer je pense. Tout d'abord, c'est la génération d'un openning book, qui, si c'est bien fait, peut être extrêmement fort. La deuxième idée aurait été d'enregistrer les parties les plus récentes dans le but de reproduire plus facilement les bons coups menant à la victoire, et éviter de refaire deux fois la même erreur.

Bibliographie

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez Liebana, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 03 2012. doi: 10.1109/TCIAIG.2012.2186810.

[2] T. Cazenave and E. Diemert. *Memorizing the Playout Policy*, pages 96–107. 02 2018. ISBN 978-3-319-75930-2. doi: 10.1007/978-3-319-75931-9_7.

[3] M. Daoud, N. Kharma, A. Haidar, and J. Popoola. Ayo, the awari player, or how better representation trumps deeper search. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, volume 1, pages 1001–1006. IEEE, 2004.

[4] A. De Bruin, W. Pijls, and A. Plaat. Solution trees as a basis for game-tree search. *ICGA Journal*, 17(4):207–219, 1994.

[5] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4): 293–326, 1975.

[6] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[7] T. Lincke and R. van der Goot. Marvin wins awari tournament. *ICGA Journal*, 23(3):173–174, 2000.

[8] T. R. Lincke. Strategies for the automatic construction of opening books. In *International Conference on Computers and Games*, pages 74–86. Springer, 2000.

[9] J. Romein and H. Bal. Awari is solved. *ICGA Journal*, 25(3):162–165, 09 2002. doi: 10.3233/ICG-2002-25306.

[10] J. Romein and H. Bal. Solving the game of awari using parallel retrograde analysis. *Computer*, 38(10):26–33, 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1236468.

[11] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.