INF8175 : Projet
Agent Intelligent pour le jeu Abalone

LesFrenchies

Antoine Leblanc (2310186)

Clément Garancini (2315136)

Introduction

Ce projet vise à construire un agent logique pouvant jouer au jeu de plateau Abalone. L'ensemble du code du projet peut être retrouvé sur le github : https://github.com/ClemGarancini/Abalone_sources_projet.git . Ce repository contient les différents agents dont les fonctionnements sont détaillés dans ce rapport.

1 – Algorithme Initial

1.1 – MiniMax

Afin d'implémenter notre agent, nous avons choisi d'utiliser l'algorithme minimax. Il repose sur le fait que l'on souhaite maximiser notre score et que le joueur adverse cherche à faire de même. La mise en place de l'algorithme se décompose en deux fonctions (*maxValue* et *minValue*) qui vont respectivement chercher à obtenir le meilleur score atteignable depuis le nœud évalué pour **l'agent** (maximisation du score) et pour **l'adversaire** (minimisation du score).

1.2 – Alpha-Beta Prunning

Le premier ajout que l'on a fait à MiniMax est l'Alpha-Bêta prunning. C'est un algorithme très classique dans la recherche adversarielle en arbre. Il repose sur le principe évoqué précédemment. En effet, si dans nos états max possibles déjà parcourus le score maximal est de v, alors les états suivants qui possèderont des branches d'état min inférieurs à a, alors il sera inutile de poursuivre la recherche des autres branches min de ce même état max.

L'implémentation de l'algorithme est assez classique et il n'y a pas de changement majeur par rapport au pseudo-code, hormis les scores minimal et maximal que l'on peut espérer obtenir qui sont de -6 et de 0 respectivement. Le score est calculé selon le nombre de boules que l'on a en dehors du terrain.

2 – Améliorations

L'une des contraintes des règles du jeu est la limite de 15 min de *« réflexion »* qui est fixée à notre agent. Vu les ordres de grandeurs du nombre de coups possibles à chaque tour (empiriquement évalué à 60), le nombre de nœuds explose très rapidement (presque 1 000 000 000 d'états possibles après 5 coups joués) il est impératif de grandement améliorer cette recherche en graphe presque exhaustive. Voici les diverses améliorations que nous avons implémentées dans notre projet.

2.1 – Gestion du temps

La première adaptation que l'on a été forcé de réaliser a été de gérer le temps de calcul à chaque coup pour respecter la contrainte évoquée plus haut. L'idée initiale est la plus simple, une partie se déroulant en 25 tours maximum et 15 min sont allouées pour la partie totale, l'agent possède 15/25 = 0.5 min par coup soit 30 secondes. Cependant, à cause de la récurrence des méthodes *maxValue* et *minValue*, faire remonter l'information de l'action optimale prend un temps non négligeable. Il nous est apparu comme plus simple d'évaluer empiriquement le temps de calcul

en fonction du nombre de nœuds visités. Après divers essais, la valeur que nous avons retenue est de 100 000 nœuds visités par exploration.

2.2 – Heuristique

On a utilisé une heuristique pour approximer la valeur de certains états. Cette heuristique doit nous permettre d'évaluer assez précisément la valeur d'un état mais elle représente aussi notre stratégie pour gagner. Dans notre heuristique nous voulions prendre en compte 6 facteurs différends, pour une facilité de compréhension on fait l'hypothèse que l'on joue les blanc (W) et notre adversaire les noirs (B) :

- Notre score, noté *score_W* : On prend en compte le nombre de nos boules qui sont sorties du terrain
- Le score adverse, noté *score_B* : On prend en compte le nombre des boules adverses sorties du terrain
- À quel point nos boules sont resserrées, noté *paquet_W* : On a pu remarquer qu'avoir des paquets de boules est intéressant pour pouvoir pousser les boules de l'adversaire. Pour cela on parcourt l'ensemble des lignes, colonnes et des diagonales (diagonale Sud-Est et Sud-Ouest) et on compte le nombre de fois que 2 boules blanches sont côte à côte.
- À quel point les boules adverses sont disséminées, noté *paquet_B* : On a pu remarquer que des boules isolées pour l'adversaire est avantageux pour nous car on peut facilement les sortir du terrain. Le calcul est similaire à celui pour calculer *paquet_W*.
- À quel point nos boules sont proches du centre, noté *distance_W_centre* : On a aussi remarqué qu'avoir ses boules au centre du terrain nous place en position de force par rapport à l'adversaire. Pour calculer *distance_W_centre*, on calcule la somme des distances euclidiennes entre nos boules et le centre du terrain.
- À quel point les boules adverses sont éloignées du centre du terrain, noté *distance_B_centre* : Il est en effet intéressant pour nous d'essayer de faire en sorte que les boules noires se trouvent en périphérie du terrain. Le calcul est similaire à celui de *distance_W_centre*.

Ainsi, on a choisi de rendre compte de toutes ces données dans l'heuristique H selon la formule suivante :

$$H = score\_W - \alpha_1 * score\_B + \alpha_2 \frac{distance\_W\_centre - distance\_B\_centre}{distance\_W\_centre + distance\_B\_centre} + \alpha_3 \frac{paquet\_W - paquet\_B}{paquet\_W + paquet\_B}$$

Avec $\alpha_1, \alpha_2, \alpha_3$ des nombres réels compris entre 0 et 1. Après plusieurs expérimentations, nous avons convenu que les valeurs optimales de ces hyperparamètres sont :

$$\alpha_1 = 0{,}5;\ \alpha_2 = 0{,}25;\ \alpha_3 = 0{,}25.$$

On note que l'on a rajouté un facteur k pour tenir compte du fait que notre heuristique pouvait prédire un score en dehors de [-6,6] (intervalle de score final pour notre heuristique

Ainsi, nous obtenons une heuristique qui va être utilisé pour notre agent. Il faut que l'on voie avant ça si notre heuristique respecte les deux propriétés fondamentales des heuristiques : admissibilité et consistance. Concernant l'admissibilité, c'est assez compliqué de savoir si notre heuristique respecte cette propriété. En effet, on ne connaît pas la valeur réelle des états du jeu. Par exemple, dans ce jeu le fait que nos boules soient resserrées ou disséminées n'entraîne pas la même valeur pour un état (avec une différence de score équivalente) et même plus : on ne connaît pas de combien la valeur de l'état aux boules resserrés et supérieur à celui avec les boules disséminées. Ainsi, la seule chose dont on peut être sûr est que notre heuristique prédit correctement la valeur d'un état final (sans troncage, c'est-à-dire 6 boules blanches ou noires sorties) puisque son intervalle de valeur est de [-6,6] grâce au facteur k (qui vaut 6.5/6). Cependant

on a opté pour une heuristique avec une prédiction qui avoisine à plus ou moins 1 la différence de score entre les boules blanches et noires, ce qui fait que les valeurs des états prédites ne sont pas totalement aberrantes et on peut faire l'hypothèse que notre heuristique est quasi-admissible sans en être totalement convaincu (ce qui est impossible en pratique).

En ce qui concerne la consistance, on sait que la transition entre deux états où une boule est sortie vaut soit 1 soit -1. En plus de cela, selon le changement d'agencement des boules un joueur peut se retrouver en position de force car il possède plus de boules resserrées par exemple. Ainsi les véritables valeurs de transition peuvent être supérieurs à 1 ou inférieurs à -1. Pour notre heuristique on voit que l'on peut évaluer parfois un gain supérieur à ce qu'il en est réellement, et inversement, parce qu'on approxime ce qui nous semble être une bonne manière de calculer la valeur des états. Ainsi, on ne peut garantir la consistance car on ne sait pas objectivement quels sont les coûts de transition mais comme notre la valeur calculée par notre heuristique ne varie que de très peu pour les facteurs de distances et de nombres de boules resserrés, on peut supposer une quasi-consistance de notre heuristique.

Finalement, on ne peut pas garantir que notre heuristique soit admissible ou consistante par le fait de ne pas connaître les véritables valeurs des états, mais comme on a essayé de calculer des facteurs importants pour gagner, on peut supposer une quasi-admissibilité et une quasi-consistance de notre heuristique.

2.3 – Tables de transposition

Le jeu d'Abalone ne possédant qu'un seul type de pièces et un espace de déplacement réduit, lors d'une recherche en arbre des états atteignables, de nombreux nœuds doublons existent sur des chemins distincts. Une des propriétés d'un Processus de Décision de Markov tel que le jeu d'Abalone étant la dépendance de la distribution des états suivant à l'état concerné et seulement à celui-ci, deux nœuds représentant deux configurations du jeu identiques donneront des sous-arbres identiques. Il est donc possible de ne pas étendre les nœuds ayant déjà été visités sans prendre le risque de rater une solution optimale. Pour cela, nous proposons d'utiliser des tables de transpositions.

Hachage de Zobrist

Afin de vérifier rapidement l'identité entre deux configurations de jeu, il est bon d'utiliser une fonction de hachage. Une méthode facile à implémenter et adaptée à un jeu de plateau est la fonction de hachage de Zobrist.

Le principe est d'attribuer un entier, appelé *clef*, aléatoire assez grand (64 bits par exemple) pour chaque combinaison de case et d'états possibles de cette case (boule blanche, boule noire, case vide). Une fois cela fait, afin de calculer le *hash* d'une configuration, il suffit de prendre les *clefs* qui correspondent aux états des case dans cette configuration et d'appliquer successivement l'opération logique XOR sur la représentation en binaire de ces *clefs*.

Ce hachage est implémenté dans les méthodes *init_zobrist()* et *zobrist_hash(state)* qui respectivement initialise les clefs de hachage et calcule le hash d'une configuration donnée (*state*). Il ne reste plus, lors de la recherche en graphe de stocker dans un dictionnaire les *hash* des états visités et de vérifier si le hash d'un état est présent dans ce dictionnaire avant de le visiter.

2.4 – Sélections des actions à étendre

Jusqu'à présent, l'algorithme effectue un parcours en profondeur classique sans choisir au préalable les actions à visiter. Or nous disposons d'une fonction permettant d'évaluer rapidement et d'une façon assez conservative la qualité d'un état. Une idée que nous avons donc implémentée ensuite est la sélection des « meilleures » actions à explorer dans un état donné.

Avant d'explorer les nœuds fils, on les évalue et classe avec notre heuristiques puis l'on explore les meilleurs. Intuitivement, cela permet d'éviter de perdre du temps de recherche sur des actions qui sont à l'évidence mauvaises (au début, de nombreuses actions induisent instantanément la perte d'une boule par exemple).

3 - Analyse et résultats

3.1 – Premier algorithme

Notre algorithme initial, comprenant donc la méthode d'Alpha-Prunning, le calcul de l'heuristique et la prise en compte de la limite de temps, bat l'agent aléatoire facilement. Ce résultat, certes attendu, nous permet tout de même d'observer que le calcul de l'heuristique a bien les conséquences souhaitées en termes de stratégie (voir fig.1) :

- Nos billes sont rapprochées
- Nos billes sont proches du centre
- Les billes de l'adversaire sont éparpillées
- Les billes de l'adversaire sont loin du centre

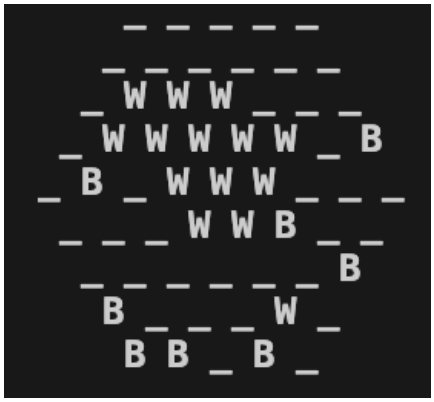

Figure 1: Résultat d'une partie de l'agent initial face à l'agent aléatoire

3.2 – Table de transpositions

Avec l'ajout de tables de transpositions, l'agent peut visiter plus de branches de l'arbres durant son temps de réflexion grâce à la suppression des doublons dans l'arbre. En effet, sur une partie, grâce aux tables, en moyenne 660 doublons ne sont pas étendus par recherche. Le nouvel agent bat son prédécesseur, ce qui est parfaitement logique, puisque l'ensemble des états visités par le premier agent est inclus dans celui des états visités par le second. Sa recherche sera plus exhaustive et son score meilleur.

Figure 2: Résultat d'une partie de l'agent avec tables de transposition face à l'agent initial et nombre moyen de doublons élagués par coup

3.3 – Sélection des actions

Même si éliminer des doublons permet d'améliorer notre agent, notre problème reste l'explosion de nœuds ce qui rend l'amélioration des tables de transposition assez faible. L'idée intuitive, que l'on applique naturellement en tant qu'humain à cause de notre faible capacité à gérer de nombreux cas en parallèle, est donc de ne sélectionner pour l'expansion de l'arbre que les nœuds qui semblent intéressant à première vue.

Outre l'élimination de chemins peu prometteurs, cette modification permet aussi indirectement d'améliorer l'efficacité du *prunning*. En effet, si l'heuristique est bonne, les chemins prometteurs auront de meilleures chances de donner de bons scores, ce qui augmentera théoriquement le nombre de nœuds élagués et ainsi permettra de visiter encore plus de nœuds.

Ce nouvel agent bat à chaque coup le précédent.

4 – Pistes d'améliorations

4.1 – Améliorations de MiniMax

D'autres méthodes auraient pu être implémentées afin d'améliorer directement notre algorithme. Nous avons tout d'abord l'heuristique qui est une piste constante d'améliorations. Dans notre cas nous ne prenons en considération que des caractéristiques globales de la configuration (proximité des pièces entre elles par exemple). Mais nous pourrions considérer des situations locales comme les menaces directes d'une éjection. D'une manière générale, travailler et retravailler l'heuristique améliorera l'algorithme à coup sûr puisqu'elle nous sert de base pour estimer la qualité d'une configuration.

Concernant la gestion du temps, une méthode qui peut fonctionner est la recherche par profondeur itérative. Il s'agit d'augmenter successivement la profondeur maximale de recherche jusqu'à la fin du temps imparti pour la recherche.

4.2 – Autres algorithmes

D'autres méthodes de recherche adversarielle peuvent être également implémentées telles que la recherche en arbre de Monte-Carlo qui simule des parties afin d'obtenir empiriquement des scores pour les états et les actions possibles.

Il est aussi possible d'entraîner des réseaux de neurones afin de pouvoir évaluer la qualité d'une configuration donnée, ce qui peut remplacer l'heuristique dans certaines parties de l'algorithme et ainsi améliorer la performance globale.