Rapport de projet
Agent intelligent pour le jeu Abalone

Équipe *Frabalon*

François MARIE
2309736

François GOYBET
2307249

POLYTECHNIQUE MONTRÉAL

INF8175 - Intelligence Artificielle : Méthodes et Algorithmes

16 April 2024

Sommaire

1. Introduction

Le but de ce projet est de développer un agent intelligent pour le jeu Abalone. Le jeu consiste en un plateau hexagonal où deux joueurs s'affrontent avec chacun 14 billes. Le but est de pousser les billes adverses hors du plateau dans un maximum de 50 tours. Certaines règles de déplacements ont été modifiées dans le sujet pour simplifier le jeu et réduire la complexité.

2. Analyse du jeu

Afin de développer un agent intelligent, il est d'abord nécessaire de comprendre les règles et stratégies du jeu. Nous connaissions déjà Abalone avant ce projet ce qui nous a permis de rapidement faire émerger des idées lors de nos parties :

- Plus une bille se trouve proche d'un bord, plus elle est en danger. Les cases sur les "sommets" de l'hexagone sont d'autant plus dangereuses car une bille peut se faire éjecter depuis trois directions différentes.
- Le centre du plateau, en particulier la case centrale, est un endroit stratégique qui permet de dominer l'adversaire en le forçant à disposer ses billes en croissant autour de la case centrale et donc de le pousser plus facilement.
- Les formations en "blob" sont plus faciles à protéger que les billes isolées ou les formation en ligne. En effet, cette formation offire le moins de surface d'attaque et permet de riposter dans plusieurs directions.
- Il est souvent contreproductif de chercher à ramener une bille isolées dans un groupe si cela prend plus de 1 ou 2 tours.
- Les priorités évoluent avec le temps. En début de partie, il est souvent préférable de chercher à contrôler le centre et construire une formation avec ses billes. Inversement, lors des derniers tours, il est préférable d'attaquer l'adversaire, quitte à perdre le centre ou déconstruire sa formation.

3. Agents développés

3.1. Alphabeta

3.1.1. Description

La version principale de l'agent de notre projet utilise l'algorithme alphabeta. C'est la version que nous avons utilisée pour le tournoi et la remise.
La première version de l'agent développée fut un agent minimax simple afin de tester les mécanismes du projet fourni. Nous avons ensuite rapidement implémenter l'élagage alphabeta pour améliorer les performances de l'agent et obtenir un meilleur niveau de profondeur.
Par la suite nous avons ajouté une heuristique basée sur les stratégies évoquées dans **Analyse du jeu**. L'heuristique a été modifiées et améliorée au fur et à mesure de l'avancement du projet. La version finale est décrite dans la section **Heuristique**.

3.1.2. Évolution

3.2. Monte Carlo Tree Search

3.2.1. Description

Nous avons également développé un agent qui se base sur l'algorithme Monte Carlo Tree Search (MCTS). Ce n'est pas l'agent que nous avons retenu pour le tournoi et la remise.
l'algorithme du MCTS se base sur des observations statistiques pour déterminer le meilleur coup à effectuer. Il se décompose en 4 phases : sélection, extension, simulation et backpropagation. Nous utilisons le upper confident bound applied to trees (UCT) comme règle de sélection. Lors de la phase de simulation, nous avons d'abord utiliser une approche complétement aléatoire. En effet, nous avons simulé des parties où les coups étaient décidés aléatoirement, et quant l'agent gagnait, on remontait sa victoire.
Par la suite, pour éviter de simuler des parties aléatoires, et perdre du temps à aller jusqu'à la fin des parties simulées, nous avons implémenter une heuristique, décrite dans la section **Heuristique**.

3.2.2. Évolution

4. Conception des agents

4.1. Heuristique

L'heuristique finale attribue un score pour l'agent à un état de la partie en fonction des données suivantes :

- Le score de base fourni par la méthode get_player_score() de la classe GameState est utilisé mais est ramené de l'intervalle $[-6, 6]$ à $[-1, 1]$.
- Un score des pièces présentes sur le plateau (pieces_score) représente l'avantage matériel d'un joueur sur un autre. Ce score soustrait le nombre de pièces adverses au nombre de pièces alliées puis est normalisé pour être compris dans l'intervalle $[-1, 1]$.
- Un score nommé center_score évalue simplement si la case centrale appartient à un joueur. La valeur est de 1 si une bille alliée est dessus, −1 si c'est un bille adverse et 0 sinon. Au-delà du tour 45, ce score vaut toujours 0 pour pousser l'agent à ignorer le centre et prioriser les autres scores d'attaque.
- Un score de danger (threat_score) représente le nombre de billes alliées qui sont en danger de se faire éjecter (*i.e.* qui sont sur le bord avec une bille adversaire à côté). Ce score est normalisé de sorte à être contenu dans l'intervalle $[-1, 0]$. L'importance de ce score diminue linéairement avec l'avancé du nombre de tour pour encourager l'agent à prendre des risques en fin de partie.
- Un score de clustering (cluster_score) évalue la formation des billes alliées pour favoriser les "blobs" et éviter les billes isolées et les lignes. Ce score compte pour chaque bille alliée le nombre de voisins directs alliés. La valeur est normalisé pour être dans l'intervalle $[0, 1]$. L'importance de ce score évolue selon une loi normale du nombre de tours centrée en 20 avec un écart-type de 5. Ces valeurs ont été choisi pour que l'agent ait le comportement suivant :

- En tout début de partie le contrôle du centre est à prioriser (quitte à avoir une mauvaise formation)
- En fin de première moitié de partie (vers le tour 20), l'agent se concentre à fortifier sa formation.
- En fin de partie, la formation n'est plus importante et l'agent doit prioriser l'attaque.

- Un score de distance au centre (distance_score) permet d'évaluer la disposition des billes sur le plateau à l'aide d'une valeur de dangerosité (*c.f.* Figure 1). Cette valeur est normalisée pour être contenue dans l'intervalle $[-1, 1]$. Elle augmente plus les billes alliées sont proche du centre et les billes adverses sont proches du bord. L'importance des billes alliée dans le calcul de ce score diminue linéairement avec l'avancé du nombre de tours pour que vers la fin de partie l'agent se concentre sur pousser les billes adverses sur le bord du plateau en prenant des risques. L'importance des billes adverses diminue également légérement pour que l'agent favorise l'éjection de billes à simplement pousser le plus de billes adverses en périphérie du plateau.

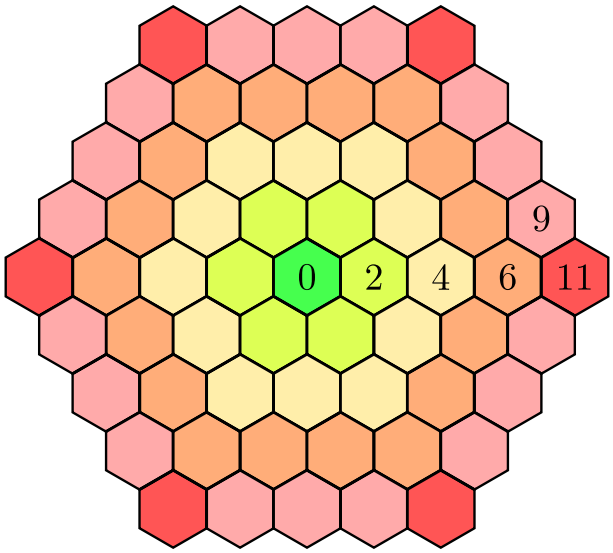

Figure 1: Représentation de la dangerosité des cases du plateau

Ces scores sont ensuite additionnés avec des pondérations pour connaître le score total de l'état. L'utilisation de pondérations a pour objectif de rapidement modifier le comportement d'un agent pour favoriser de manière générale l'attaque ou la défense. La somme brute (*i.e.* non pondérée) des scores est comprise dans $[-5, 5]$ ce qui assure que l'heuristique est admissible (car elle ne sort pas de l'intervalle possible de score de $[-6, 6]$). L'utilisation des pondérations entre 0 et 1 pour affiner le comportement l'agent assure également que l'heuristique reste admissible.

4.2. Optimisations

4.2.1. Précalculs

Le fichier utils.py contient des données précalculées qui permettent de gagner du temps d'exécution.

Distance au centre
Le système de coordonnées particulier du jeu rend difficile d'évaluer la distance au centre à partir d'une pièce. De plus nous souhaitons différencier les sommets du plateau des autres

cases du bord, comme en Figure 1. Nous avons donc décidé de crééer manuellement une hashmap constante DANGER contenant l'ensemble des positions possibles avec un niveau de danger personnalisé. Cet approche permet une grande flexibilité sur l'ajustement des niveaux de danger et évite le calcul de la distance au centre à chaque tour.

Distribution normale du facteur de clustering
L'importance du score de clustering évolue selon une loi normale (centrée en 20 avec un écart-type de 5). Dans un premier temps, nous calculions la valeur avec la librairie *scipy* à chaque tour. Cependant, cela a gravement nuit aux performances de l'agent et a quasiment doublé le temps d'exécution : sur Alphabeta avec une profondeur de 3, le temps est passé de 1min30 à presque 3min. Pour pallier à ce problème, les valeurs possibles ont été précalculées et stockées dans une liste constante.

4.2.2. Gestion du temps

Pour éviter de dépasser le temps maximal de 15min avec Alphabeta, le niveau de profondeur est ajusté dynamiquement lors de l'appel à la fonction compute_action() :

- Dans la quasi-première moitié de la partie (jusqu'au tour 19), la profondeur est fixée à 3. En effet l'agent doit seulement se concentrer sur le contrôle du centre et la formation de ses billes.
- Après cette phase terminée, la profondeur est augmentée à 4 pour attaquer l'adversaire et éjecter ses billes.
- Si le temps restant est inférieur à 2min30, la profondeur passe à nouveau à 3 pour éviter de dépasser le temps imparti et risquer de perdre la partie.
- Lors des 3 derniers tours, la profondeur est fixée au nombre de tours restants afin d'éviter de calculer des coups inutiles.

5. Résultats

5.1. Performances

Les versions finales de chaque agent (Alaphabeta et MCTS) battent l'agent greedy implémenté par défaut.
Pour départager la version qui serait envoyée au tournoi, nous les avons fait s'affronter sur l'ensemble des configurations :

5.2. Tournoi

Lors du tournoi, notre agent a réussi à finir 2ème de sa poule avec 2 victoires pour 1 défaite, avec un total de 37 points. Cependant, le premier round a été perdu 4-1 contre un autre agent. Bien que cette défaite soit un peu décevante, nous sommes satisfaits de la performance globale de notre agent. Le système de gestion de temps semble avoir fonctionner correctement sur le serveur de tournoi.

5.3. Limtations et évolutions possibles

Malgré de bonnes performances, certaines points restent limitant notamment concernant l'heuristique :

- Il serait possible d'identifier et favoriser certains patternes et disposition de billes particulier. Par exemple, lorsque l'adversaire forme une ligne il serait intéressant de prioriser les coups qui permettent de la casser pour séparer ses billes et l'affaiiblir.
- L'agent est conçu pour jouer uniquement sur des parties de 50 tours. Cependant, cette limitation n'est pas présente dans les règles du jeu et n'a été ajoutée que pour simplifier le projet. Il serait intéressant de rendre l'agent capbable de jouer sur des parties de longueur variable.
- D'autres approches n'ont pas été explorées par manque de temps. Il pourrait par exemple être intéressant de réaliser un agent à base d'apprentissage profond et le comparer à l'Alphabeta réalisé.

6. Conclusion