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

18 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 compléxité.

2. Analyse du jeu

Afin de développer un agent intelligent, il est d'abord nécessaire de comprendre les règles et stratégies du jeu. Lors de nos parties nous avons pu identifier ces points clés :

- Plus une bille se trouve proche d'un bord, plus elle est en danger. Les "sommets" de l'hexagone sont les plus dangereux car on peut se faire éjecter depuis 3 directions.
- 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.
- Les formations en "blob" sont plus faciles à protéger. En effet, cette formation offre peu de surface d'attaque et permet de riposter dans plusieurs directions.
- Il est souvent contreproductif de chercher à ramener une bille isolée dans un groupe si cela prend plus de 1 ou 2 tours.
- Les priorités évoluent avec le temps. En début de partie, il est préférable de contrôler le centre et de solidifier sa formation. Lors des derniers tours, il est préférable d'attaquer l'adversaire, quitte à perdre le centre ou à déconstruire sa formation.

3. Agents développés

3.1. Minimax avec alphabeta pruning

3.1.1. Description

La version finale de l'agent utilise l'algorithme minimax avec alphabeta pruning. L'agent utilise également une heuristique pour évaluer les états de la partie et sélectionner la meilleure action (voir **Heuristique** pour une description de son fonctionnement). Le niveau de profondeur de l'arbre de recherche est dynamique et change en fonction de l'avancement de la partie et du temps restant (voir **Optimisations** pour plus de détails).

3.1.2. Évolution

La première version développée est un minimax simple (sans pruning ou heuristique) afin de tester le projet fourni. Cette version est désignée par "v0" et n'arrive pas à battre systématiquement l'agent *Greedy*.

Par la suite, nous avons rapidement implémenté l'élagage alphabeta afin d'améliorer les performances. Une première heuristique basée uniquement sur le nombre de pièces et leur distance au centre a également été ajoutée. Cette version "v1" bat l'agent *Greedy*. Nous avons ensuite réécrit une v2 pour nettoyer le code et ajouter de nouvelles statistiques à notre heuristique (voir **Heuristique** pour la version finale). Cette version bat *Greedy* et la v1. Enfin, pour le

tournoi, nous avons affiner les coefficients de chaque composante de l'heuristique envourager l'attaque (v2.1).

L'évolution des performances face à *Greedy* est illustrée Figure 1. On constate bien une amélioration du nombre de points moyen avec la montée de version ainsi qu'un diminution du score de *Greedy* à partir de la v1. La version finale (v2.1) a également été testée sur les agents précédents (*c.f* Figure 2) et les bat systématiquement. On constate bien également que l'écart est plus sérré avec la montée de version, ce qui montre que les agents sont de plus en plus forts.

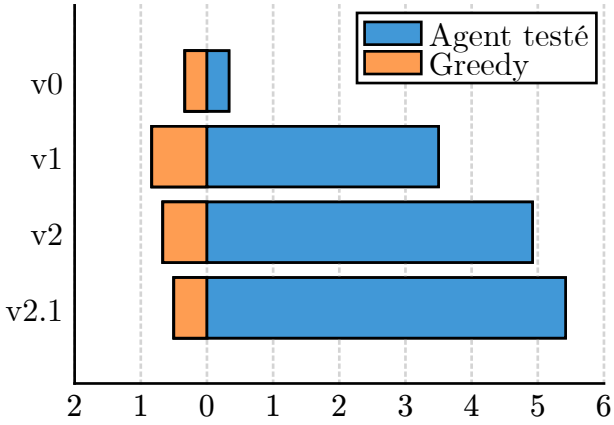

Figure 1: Nombre de points moyen par partie de l'agent contre Greedy.

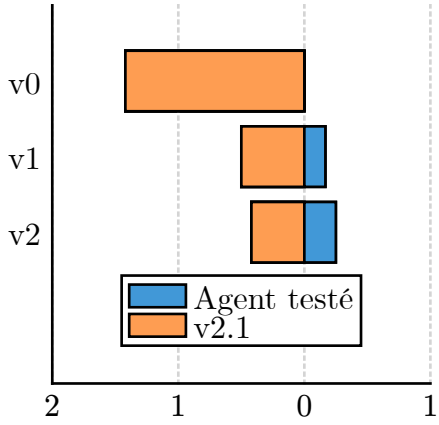

Figure 2: Nombre de points moyen par partie de l'agent contre la v2.1

Les points moyens sont calculés sur 12 matchs avec 3 parties par configuration.

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 avec comme paramètre $c = \sqrt{2}$.
Le temps de simulation entre chaque coup est déterminé à l'avance. Nous avons d'abord mis un temps fixe, puis un temps variable.
Nous avons développé une partie où les simulations sont aléatoires, puis une version l'heuristique décrite dans la section **Heuristique**.

3.2.2. Évolution

Pour la première version de notre agent nous avons d'abord utilisé une approche complétement aléatoire. En effet, nous avons simulé des parties où les coups étaient décidés aléatoirement, et quand l'agent gagnait sa victoire était remontée. Nous avons laissé également 5 secondes de simulation par tour. Il s'agit de "v0".
Par la suite, nous avons utilisé le même agent, mais avec un temps variable, c'est à dire que le temps suit une loi normale centrée au coup 20, de sorte que l'agent prenne 15 minutes au total pour ses coups. Il s'agit de "v0.1".

Nous avons ensuite implémenté une première heuristique. Le temps reste variable. Il s'agit de "v1". Enfin nous avons affiné nos coefficents de notre heuristique pour avoir la version finale de notre agent MCTS. Il s'agit de "v1.1".

Figure 3: Nombre de points moyen par partie de l'agent contre Greedy.

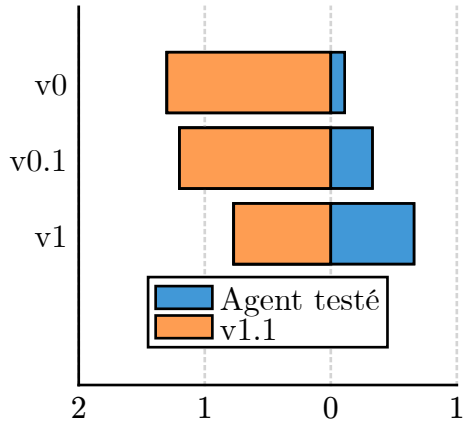

Figure 4: Nombre de points moyen par partie de l'agent contre la v1.1

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 $[-1, 0]$.
- 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 normalisée 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 choisies 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 à l'aide d'une valeur de dangerosité (*c.f.* Figure 5). Ce score est contenu dans $[-1, 1]$ et 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 diminue linéairement avec le nombre de tours pour que vers la fin pour que l'agent prenne plus de risques. L'importance des billes adverses diminue aussi légérement pour favoriser les éjections plutôt que de pousser le plus de billes adverses en périphérie du plateau.

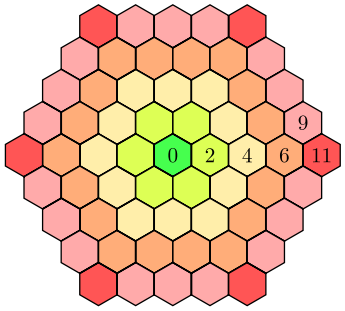

Figure 5: Dangerosité des cases du plateau

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 $[-6, 6]$ 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 5. Nous avons donc décidé de créeer 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. Choix de l'agent

Les versions finales de chaque agent (Alaphabeta et MCTS) battent l'agent greedy implémenté par défaut.
Pour départager quelle version envoyer au tournoi, nous les avons fait s'affronter sur l'ensemble des configurations. Au total nous avons réalisé 12 parties avec 3 parties par configuration.
Dans toutes les parties, l'agent Alphabeta a gagné contre l'agent MCTS avec 6 billes MCTS éjectées avant la fin de la partie et aucune bille Alphabeta éjectée. L'agent Alphabeta a donc été choisi comme version finale.

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. Limitations 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 dispositions de billes particuliers. 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'affaiblir.
- 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

Ce projet nous a permis d'explorer, de comprendre et d'expérimenter plusieurs méthodes et algorithmes d'intelligence artificielle appliqués au jeu Abalone. La grande liberté d'implémentation nous a permis d'étudier, examiner et tester en profondeur les techniques de minimax et de MCTS. Bien que notre agent final montre des performances prometteuses, d'autres techniques plus avancées existent, et restent à être explorées. Ce projet constitue un étape importante dans notre compréhension des applications pratiques de l'IA.