École Polytechnique de Montréal

INF8175 – Intelligence artificielle : Méthodes et algorithmes

Automne 2023

Projet Abalone

Remis par :

Léo Valette - 2307835

Oida Kon Fataki - 1930768

Nom d'équipe : ***TheAbs***

Soumis à : Quentin Cappart

Date de remise : Le 7 décembre

Tables des matières

Le but de ce projet est de concevoir et implémenter un agent d'intelligence artificielle capable de jouer au jeu de plateau Abalone de manière autonome et compétente. Ce jeu à information complète et à somme non nulle, dépourvu d'éléments aléatoires, constitue un terrain idéal pour explorer les capacités des systèmes d'intelligence artificielle. Notre approche vise à utiliser l'algorithme Alpha-Beta Pruning couplé à une fonction heuristique pour explorer efficacement l'espace de recherche du jeu et identifier les actions optimales.

Nous détaillerons la méthodologie employée ainsi que les différentes évolutions et choix opérés au cours du projet. Enfin, nous discuterons des résultats obtenus et des limites de notre agent.

Méthodologies

Dans cette section, nous détaillerons notre approche méthodique pour développer un agent d'intelligence artificielle performant dans le contexte du jeu Abalone. Fondée sur l'algorithme Alpha-Beta Pruning, notre méthode intègre une fonction heuristique dédiée, visant à guider efficacement les décisions de l'agent.

Algorithme de recherche

Nous avons opté pour l'utilisation de l'algorithme Alpha-Beta Pruning pour plusieurs raisons. Tout d'abord, il est recommandé et a démontré d'excellentes performances par le passé. De plus, notre familiarité avec cet algorithme nous a permis de mieux comprendre les résultats obtenus et de trouver des solutions aux problèmes rencontrés rapidement. De plus, cet algorithme est assez simple et permet l'ajout facile de nombreuses améliorations.

Une première amélioration capitale consiste à introduire la profondeur de recherche. Ce léger changement dans l'implémentation l'algorithme implique l'utilisation d'une fonction heuristique qui permet d'estimer un état sans nécessiter une exploration complète de l'arbre de décision. Cette modification est indispensable dans la pratique pour notre projet, car Abalone présente un nombre d'états extrêmement élevé, rendant une exploration complète de l'arbre de décision impraticable. Les performances de notre agent dépendent alors grandement de la qualité de l'heuristique utilisée, nous décrirons la nôtre en détail dans la partie suivante.

Pour accélérer la recherche, nous avons introduit une première amélioration à l'algorithme de base, centrée sur l'ordonnancement des mouvements. L'objectif est d'optimiser l'utilisation de la fonction de pruning de l'algorithme en priorisant les actions jugées les plus prometteuses. Notons qu'il est important que cette fonction soit très rapide car elle est appelée souvent et peut vite devenir contreproductive. À cette fin, nous faisons appel à la fonction *utils.getOrderScore()*, spécifiquement conçue pour évaluer l'intérêt d'une action dans le contexte d'Abalone. En comparant les différences de score entre l'état actuel et l'état suivant généré par une action, cette fonction favorise les actions éliminant les pièces de l'adversaire tout en pénalisant celles menant à la perte de nos propres pièces.

Une des complexités de ce projet réside dans la gestion du temps alloué à notre agent. Nous avons choisi d'instaurer une limite de temps évolutive pour chaque coup que l'agent doit jouer. Concrètement, le temps restant est réparti équitablement entre tous les coups à venir. Cette approche permet d'ajuster dynamiquement le temps alloué à chaque recherche. Si une recherche prend moins de temps que la limite fixée, les autres recherches bénéficieront d'un temps supplémentaire ; autrement, la recherche est interrompue dès que la limite est atteinte. Bien que cette interruption prématurée puisse ne pas conduire à l'identification du coup optimal, le risque est réduit grâce à l'ordonnancement des mouvements.

La version finale de l'agent intègre ces améliorations spécifiques, mais notre exploration ne s'est pas limitée à celles-ci. D'autres tentatives ont été étudiées, et nous les détaillerons dans les sections suivantes sur l'Évolution et la Discussion. Avant d'explorer ces aspects, plongeons dans les mécanismes de la fonction heuristique, élément crucial qui influence grandement les performances de notre agent.

Fonction heuristique

Pour concevoir notre heuristique, nous avons privilégié une approche basée sur la position des billes sur le plateau de jeu. Nous avons attribué une valeur plus élevée aux billes du joueur lorsqu'elles se trouvent au centre, tout en les pénalisant légèrement lorsqu'elles occupent les bords. Par opposition, les billes de l'adversaire ont été valorisées sur les côtés et pénalisées au centre. Nous avons également pris en compte les scores des joueurs avec un facteur d'échelle élevé pour que l'agent puisse concrétiser son avantage positionnel en poussant les billes adverses en dehors du plateau.

La figure ci-dessous présente les scores associés aux différentes cases du plateau :

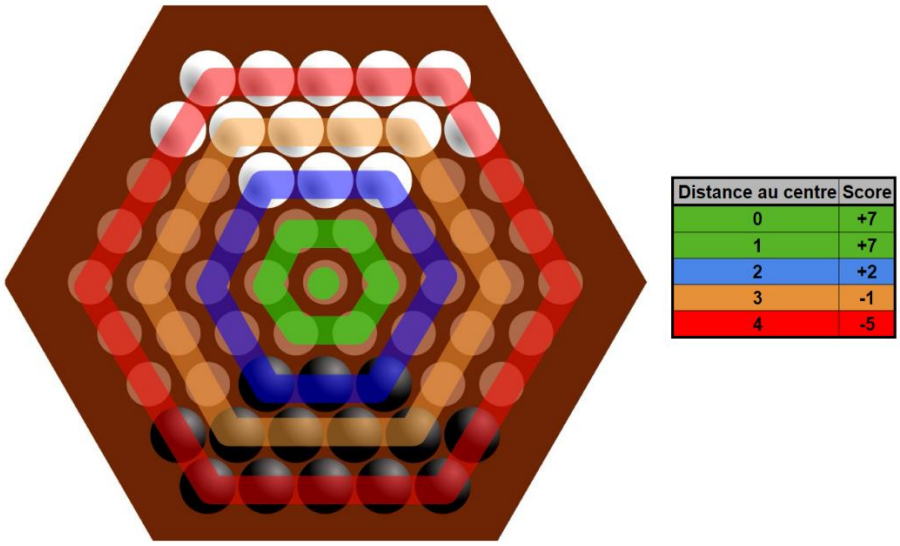

Figure 1: Scores associés aux cases du plateau

Trouver des valeurs appropriées pour refléter l'importance de la position des billes par rapport au centre du plateau a constitué un défi. Nous avons introduit un facteur de score

ajustable pour moduler l'impact du score des joueurs sur l'évaluation globale. Un facteur élevé accentue l'importance du score, tandis qu'un facteur bas le diminue. De plus, des pénalités ont été introduites pour les billes isolées du joueur, tandis que des récompenses ont été attribuées aux billes isolées de l'adversaire. Cette modification visait à encourager l'agent à former des groupes de billes connectées tout en le motivant à disperser ou isoler les pièces adverses.

Enfin, nous nous sommes assuré que l'évaluation du score était symétrique. Cela signifie que l'évaluation d'un état par un joueur doit être l'opposé de l'évaluation de ce même état par l'adversaire, garantissant ainsi à l'heuristique une impartialité vis-à-vis des joueurs.

Résultats et évolution de l'agent

Dans cette section, nous examinons comment notre approche a évolué, tant du côté de l'algorithme de recherche que de la fonction heuristique. Ces ajustements reflètent notre démarche pour améliorer les performances de notre agent dans le jeu Abalone.

Dans les premières itérations, notre approche reposait sur un Minimax simple. Confrontés à des performances décevantes, nous avons introduit le pruning avec l'algorithme Alpha-Beta. Cette évolution a considérablement réduit le nombre de nœuds explorés, améliorant significativement les performances. L'ajout de l'ordonnance des actions décrite en détail dans la section *Algorithme de recherche*, a grandement renforcé l'efficacité du pruning. Par exemple, pour une recherche de profondeur 4, nous sommes passés en moyenne de 70 000 états évalués à 30 000. Ces améliorations combinées ont permis d'accroître la profondeur de recherche de l'agent, augmentant ainsi ses performances globales.

Évaluer une position dans Abalone pose la difficulté de l'effet de myopie, où une position considérée avantageuse peut rapidement devenir perdante au tour suivant. Pour atténuer cet effet, nous avons forcé l'arrêt de la recherche seulement sur des états jugés "stables". Un état est défini comme stable si aucune pièce ne peut être poussée hors du plateau au tour suivant. Cela vise à garantir que la position ne change pas significativement juste après l'arrêt de la recherche. Cependant, cette approche, bien qu'intéressante en théorie, s'est avérée peu réalisable en pratique. Le calcul de la stabilité d'un état exige d'explorer tous les coups suivants pour s'assurer qu'aucun ne pousse une bille hors du plateau. Cette exigence a entraîné un coût de calcul élevé, prolongeant les temps d'exécution. De plus, la recherche pouvait descendre profondément sans trouver d'état stable. Face à ces contraintes, nous avons préféré ne pas utiliser cette méthode, bien que des ajustements pourraient la rendre viable (voir Discussion).

Dans la version finale, nous avons intégré une dimension temporelle pour deux objectifs principaux. Tout d'abord, nous avons réparti le temps de calcul disponible pour chaque coup, permettant ainsi une gestion plus efficace des ressources. Aussi, la gestion du temps revêtait une importance particulière en prévision de futures modifications que nous envisagions, notamment l'implémentation de l'itérative deepening, détaillée dans la section *Perpectives d'amélioration*.

En ce qui concerne l'heuristique, nous avons initialement mis en place une approche simple et passive, où l'agent se contentait de compter ses propres billes. Cette approche ne l'incitait pas à pousser les billes de l'adversaire, mais simplement à éviter de faire tomber les siennes. Cependant, il est vite devenu évident que cela ne suffisait pas pour encourager l'agent à adopter une stratégie offensive.

Dans une deuxième itération, nous avons introduit la notion de l'adversaire en inversant les scores, incitant ainsi l'agent à pousser les billes adverses vers l'extérieur. Malgré cela, il ne montrait toujours pas la capacité d'attaquer efficacement, car il n'avait pas une vision suffisamment profonde pour pousser les billes adverses hors du plateau.

Pour remédier à cela, nous avons introduit plusieurs notions qui ont été présentées en détails dans la partie *Fonction heuristique.*

En ce qui concerne les résultats, l'image ci-dessous présente un tableau récapitulatif des performances de chaque agent, intégrant les évolutions mentionnées précédemment. Ces résultats ont été cruciaux pour identifier la version stable que nous avons ensuite adoptée comme la version finale de notre agent.

	Total Score (V-D)	*Agent 1*	*Agent 2*	*Agent 3*	*Agent Final*	*Agent Transposition*
Agent 1	1 - 7		**0** - *2*	**1** - *1*	**0** - *2*	**0** - *2*
Agent 2	4 - 4			**2** - *0*	**0** - *2*	**0** - *2*
Agent 3	1 - 7				**0** - *2*	**0** - *2*
Agent Final	7 - 1					**1** - *1*
Agent Transposition	7 - 1					

Figure 2: Résultats des différents agents

- **Agent 1** : première implémentation de l'algorithme de recherche et première heuristique
- **Agent 2** : première implémentation de l'algorithme de recherche et deuxième heuristique
- **Agent 3** : algorithme de recherche incluant les états stables et heuristique final
- **Agent Final** : algorithme de recherche final et heuristique final
- **Agent Transposition** : algorithme de recherche incluant la table de transposition et heuristique final

Discussion

Après avoir abordé en détail l'implémentation de notre agent, nous allons mettre en lumière ses forces tout en identifiant franchement les domaines où des améliorations sont nécessaires.

Avantages et limites

Au final, notre agent réussi sa principale mission : jouer efficacement à Abalone. Il surpasse même tous les humains qui l'ont affronté, démontrant ainsi une performance supérieure à celle du joueur humain moyen. L'utilisation d'un algorithme simple et bien connu offre une

explicabilité accrue et facilite les modifications, contrairement à des méthodes plus complexes, comme les réseaux de neurones, qui sont souvent plus opaques et moins adaptables.

Cependant, bien qu'il soit robuste, il montre des limitations en termes d'efficacité pratique pour atteindre de grandes profondeurs. Cette difficulté se traduit par des erreurs et une vulnérabilité face à des adversaires capables d'anticiper sur plusieurs coups, ce qui a probablement contribué à notre défaite prématurée dans la phase finale du concours.

Pour pallier cette faiblesse, notre fonction heuristique se révèle être un atout. Elle encourage l'agent à déloger l'adversaire du centre du plateau et à pousser ses billes vers les bords. Cette approche permet à l'agent de déterminer des coups gagnants sans nécessiter une exploration en profondeur excessive.

Ces constats nous conduisent naturellement à explorer des perspectives d'amélioration qui permettrait de renforcer davantage les performances de notre agent.

Perspective d'amélioration

Parmi les perspectives d'améliorations explorées, la plus prometteuse est sans doute la table de transposition. Le principe est simple : stocker au fur et à mesure des recherches les évaluations des états explorés afin de pouvoir les réutiliser directement si on retombe dessus. D'après nos recherches, les gains potentiels sont très importants.

Après avoir affiné notre agent "classique", nous nous sommes consacrés à l'implémentation d'une table de transposition utilisant un hachage de Zobrist. Ensuite, nous avons introduit un nouvel algorithme de recherche nommé itérative Deepening, consistant en plusieurs recherches successives de plus en plus profondes. Grâce à la table de transposition, en pratique, cet algorithme permet d'atteindre des profondeurs bien plus importantes qu'une recherche classique. Malheureusement, des problèmes de fuite de mémoire, que nous n'avons pas réussi à résoudre, nous empêchent de soumettre cet agent. En effet, bien que les performances soient au rendez-vous (profondeur de recherche presque doublée), cet agent ne libère pas la mémoire et peut consommer jusqu'à 30 Go d'octets de RAM (le maximum disponible sur nos machines). Une fois à court de RAM, il se bloque et perd en raison du temps. Néanmoins, nous avons fourni le code de cet agent pour que notre travail n'ait pas été totalement inutile, et nous pensons qu'avec quelques modifications dans la manière dont nous stockons les états dans la table, l'agent pourrait être fonctionnel.

Une autre perspective d'amélioration concerne les états stables discutés dans la partie *Résultats et évolution de l'agent*. Bien que notre implémentation actuelle n'ait pas été retenue en raison du ralentissement de la recherche, nous sommes convaincus qu'en assouplissant la définition des états stables, cette fonctionnalité pourrait devenir plus pratique. Ces ajustements pourraient permettre à l'agent d'évaluer de manière plus efficace les états, représentant ainsi une piste prometteuse d'amélioration.

Une dernière amélioration à laquelle nous avons pensé est la *Search Extension*. Cette approche consiste à explorer plus en profondeur les états jugés prometteurs. Concrètement, nous

pourrions nous appuyer sur notre fonction d'ordonnancement des actions, qui offre un aperçu du potentiel de chaque coup, pour ajuster dynamiquement la profondeur de recherche. Cependant, il est essentiel de veiller à ce que cette modification n'entraîne pas des recherches trop profondes et pas assez larges, ce qui pourrait limiter l'exploration globale de l'arbre d'états.

Conclusion

En conclusion, nous sommes satisfaits des résultats obtenus avec notre agent, démontrant l'efficacité de l'algorithme Alpha-Beta pruning associé à une heuristique simple. Malgré les défis liés à la gestion de la mémoire qui ont limité nos versions finales, ce projet nous a permis de consolider nos connaissances du cours et de développer des compétences organisationnelles et de travail d'équipe.

Références

Ces 2 vidéos ont été une grande source d'inspiration:

- ***Coding Adventure: Chess***
- ***Coding Adventure: Making a Better Chess Bot***