Agent intelligent pour le jeu Abalone

Équipe MonkeyAbananelone

Saint-Hillier, Achille
2081283
Génie informatique et logiciel
École Polytechnique Montréal
achille.saint-hillier@polymtl.ca

Benoit-Paraschivoiu, Mathéo
2079549
Génie informatique et logiciel
École Polytechnique Montréal
matheo.benoit-paraschivoiu@polymtl.ca

I. Mise en contexte

Le présent rapport expose un agent intelligent spécialisé dans une version altérée du jeu Abalone. Ce jeu se déroule sur un plateau hexagonal composé de 61 cases. Les joueurs disposent de 50 coups pour remporter la partie, soit en éjectant le plus grand nombre possible de pièces ennemies du plateau (avec un maximum de six pièces), soit en se rapprochant le plus possible du centre. Le tableau I illustre la complexité de ce jeu dans un contexte logiciel. On peut ainsi constater qu'il offre moins de possibilités que le Go, mais davantage que les Échecs pour un coup moyen. Il est donc évidemment impossible d'explorer tous les états du jeu. Pour développer un agent intelligent capable de jouer à ce jeu, nous avons donc choisi d'implémenter un algorithme de type ***Minimax***, dont nous discuterons plus en détail ultérieurement.

Les contraintes de ce projet sont les suivantes : l'agent doit être implémenté en *Python* en utilisant la librairie *SeaHorse*, il doit jouer en dessous de 15 minutes par partie et il doit utiliser moins de 4 Go de RAM.

Dans la suite de ce rapport, nous examinerons en détail notre implémentation, mettant en lumière les heuristiques sélectionnées et mentionnant diverses d'approche d'optimisation réalisée. Nous présenterons alors l'agent final sélectionné et les résultats quantitatifs obtenus. Finalement, nous discuterons des améliorations envisageables et conclurons sur le projet.

TABLE I
Statistiques sur divers jeux de plateau [1] [2]

	Échecs	**Abalone**	**Go**
Facteur de branchement	35	50	250
Nombre de coups moyen par partie	80	50	150
log(taille de l'arbre de jeu)	123	86	360

II. Implémentation

Tel que dit plus haut, nous utilisons un algorithme de type *Minimax*. *Minimax* est un algorithme de recherche utilisé dans les jeux à deux joueurs, alternant entre un joueur cherchant à maximiser son score et son adversaire cherchant à le minimiser [2]. Il explore récursivement un arbre de jeu, évalue les positions finales, et choisit le meilleur coup en considérant les réponses possibles de l'adversaire. En évaluant tous les coups possibles jusqu'à une certaine profondeur, l'algorithme détermine la meilleure séquence de coups en supposant que l'adversaire joue de manière optimale également. Plus précisément, nous avons implémenté un *Heuristique Minimax*, puisque nous évaluons le score et la qualité des états avec des heuristiques.

Notre implémentation s'est concentrée donc sur deux axes : le développement d'heuristiques, et la quête de vitesse/performance afin d'optimiser notre algorithme.

A. Heuristiques

Nous avons implémenté plusieurs heuristiques tout au long de la session, pour finalement en garder quatre qui améliorerait la performance de notre agent intelligent. Nous allons ici expliquer les heuristiques gardées.

1) ***Centre de masse****:* Notre heuristique principale repose sur un mécanisme de centre de masse [3]. Tout d'abord, nous calculons la case du plateau représentant le centre de masse de chaque groupe de billes, comme illustré à l'étape 1 de la figure 1. Ensuite, nous définissons une nouvelle case, résultant d'une moyenne pondérée entre les positions des centres de masse et le centre du plateau. Le coefficient attribué au centre du plateau peut varier pour ajuster le comportement de l'agent intelligent. Un coefficient élevé favorise les actions se rapprochant du centre, tandis qu'un coefficient bas favorise une approche plus agressive en se rapprochant de

l'adversaire. Cette étape est visualisée en deuxième dans la figure 1. Enfin, nous évaluons la distance moyenne par rapport à cette case précédemment définie. En cas d'élimination d'une bille, une distance élevée lui est associée pour appliquer une pénalité significative aux actions entraînant la perte d'une bille. Cette dernière étape est représentée en dernier dans la figure 1. Cette heuristique est polyvalente et génère de bonnes actions à elle seule. Elle englobe les deux caractéristiques principales du jeu : ne pas se faire éliminer, et se rapprocher du centre.

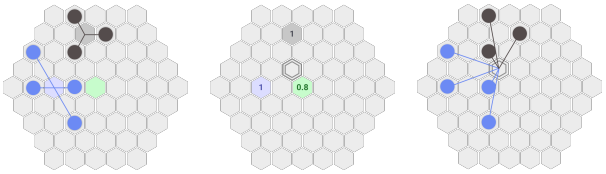

Fig. 1. Illustration de l'heuristique "centre de masse"

*2) **Supériorité numérique**:* Nous avons également mis en place une heuristique de supériorité numérique. Pour chaque ligne et chaque diagonale présente sur le plateau, nous évaluons le nombre de positions où une supériorité numérique peut être établie, définie comme une position dans laquelle un joueur peut pousser une bille adverse. Nous calculons la différence de supériorités numériques entre nos billes et celles de l'adversaire, cherchant à maximiser cette disparité. Cette heuristique nous permet d'adopter des positions plus avantageuses et d'adopter une stratégie plus agressive. Une représentation visuelle est disponible dans la figure 2.

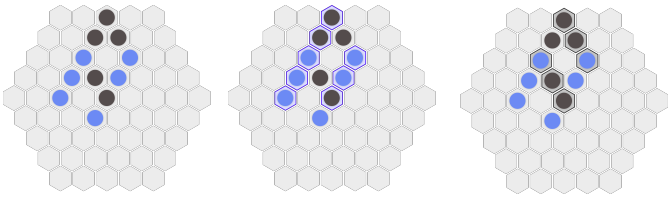

Fig. 2. Illustration de l'heuristique "supériorité numérique"

*3) **Formation regroupée**:* De plus, nous avons incorporé une heuristique visant à favoriser une disposition regroupée. Pour chaque bille dans notre ensemble, nous évaluons le nombre de voisins immédiats de la même couleur. Cette évaluation génère un score compris entre zéro et six pour chaque bille, que nous cherchons à maximiser. Bien que ce score soit spécifique à chaque bille, l'effet de groupe favorise la formation d'un voisinage complexe et regroupé, créant ainsi des positions avantageuses. En effet, dans ce jeu, l'union fait la force. La représentation graphique de cette heuristique est accessible via la figure 3.

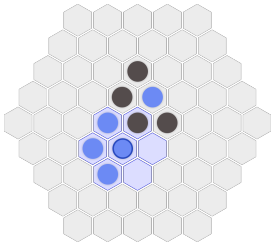

Fig. 3. Illustration de l'heuristique "formation regroupée"

*4) **Pièces sur le bord**:* Finalement, nous avons également mis en œuvre une heuristique très simple qui consiste à compter et retourner le nombre de nos pièces positionnées sur le bord du plateau, lesquelles sont considérées comme étant dans des positions dangereuses, exposées à une possible élimination. Cette approche nous permet de minimiser les situations dans lesquelles nos pièces pourraient être facilement capturées.

Il est important de noter que toutes les heuristiques mentionnées plus haut sont calibrées par des poids (hyperparamètres). L'objectif de ces poids est d'attribuer une importance différenciée à chaque heuristique, tout en normalisant les valeurs de toutes les heuristiques sur une échelle commune. Ces poids ont été initialement configurés de manière intuitive, puis ajustés manuellement suite à des simulations de parties afin de garantir leur efficacité. Une amélioration a cette façon de faire sera proposé ultérieurement.

B. Quête de performance/vitesse

Afin d'optimiser notre algorithme *Minimax* pour visiter moins d'états et rendre l'exécution de notre agent plus rapide, nous avons eu recours à trois techniques : l'élagage alpha-bêta, l'utilisation de tables de transpositions et le classement des actions possibles.

L'**élagage alpha-bêta** est une technique classique abordée dans le cours, et notre implémentation reflète les enseignements dispensés au sein du cours [2]. Par conséquent, nous n'en discuterons pas davantage dans ce rapport.

Nous avons également mis en œuvre une **table de transposition**, laquelle recense tous les états déjà parcourus, dans le but d'éviter de revisiter des états déjà explorés [2]. Pour ce faire, nous avons tenté d'utiliser le *hachage de Zobrist*. Cependant, étant donné que la librairie *SeaHorse* utilisée renvoie un état complet plutôt

qu'une différence de mouvement, nous n'avons pas pu mettre en œuvre ce *hachage* de la manière recommandée par la théorie (en $O(1)$). Nous avons plutôt dû *hacher* l'état complet, y compris le plateau dans son intégralité (en $O(N)$) et garder ces *hashs* en mémoire à l'aide d'un dictionnaire. Malgré cette approche moins optimale, elle a néanmoins contribué à optimiser notre algorithme, comme le montre les tableaux II et III, et nous avons donc choisi de la conserver.

Pour optimiser davantage l'élagage alpha-bêta afin de couper davantage de sous-arbres, il est nécessaire de procéder au **classement des actions possibles**, privilégiant ainsi l'analyse des actions susceptibles de générer un score plus élevé en premier lieu. Pour déterminer quelles actions présentent un potentiel prometteur, nous avons mis en œuvre une certaine logique. En premier lieu, nous accordons la priorité aux actions qui déplacent trois billes tout en éliminant simultanément une bille adverse. Ensuite, nous privilégions les actions impliquant le déplacement de deux billes, avec également l'élimination d'une bille adverse. Les actions suivantes les plus prioritaires sont celles où nous déplaçons trois billes sans éliminations, suivies des actions impliquant le déplacement de deux billes, et enfin, celles impliquant le déplacement d'une seule bille. L'illustration de cette logique est disponible dans la figure 4.

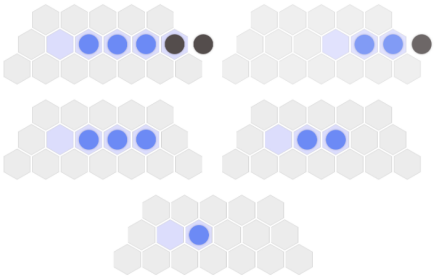

Fig. 4. Ordre de priorité des actions

Les améliorations de vitesse des approches précédentes en fonction des positions sont illustrées dans les tableaux II et III pour une profondeur de trois. On peut ainsi constater que les gains de vitesse et la réduction du nombre d'états visités sont énormes. La durée par coup est approximativement 46 fois plus rapide pour la position classique et 100 fois plus rapide pour la position alien. De plus, le nombre d'états visités est réduit de 133 fois pour la position classique et 234 fois pour la position alien.

TABLE II
ÉTATS VISITÉS ET DURÉE MOYENNE PAR COUP SELON LE TYPE D'OPTIMISATION POUR UNE POSITION CLASSIQUE EN PROFONDEUR DE TROIS

Type d'optimisation	**États visités**	**Durée (s) par coup**
Minimax de base	3373177	650.2
Minimax élagage α/β	104019	41.4
α/β, *hachage*	82654	37.1
α/β, *hachage* et ordonnancement	25368	14.2

TABLE III
ÉTATS VISITÉS ET DURÉE MOYENNE PAR COUP SELON LE TYPE D'OPTIMISATION POUR UNE POSITION ALIEN EN PROFONDEUR DE TROIS

Type d'optimisation	**États visités**	**Durée (s) par coup**
Minimax de base	4914129	857.5
Minimax élagage α/β	141296	41.7
α/β, *hachage*	106373	28.4
α/β, *hachage* et ordonnancement	20998	8.6

Nous nous limitons à une profondeur de trois dû aux contraintes de la compétition énoncées plus tôt. En effet, en considérant que le jeu comporte un maximum de 50 coups, et que la plupart du temps tous les coups sont joués, notre agent doit jouer 25 coups dans un intervalle de 15 minutes, ce qui fait une moyenne de 36 secondes par coups. Cette contrainte de temps restreint ainsi la profondeur de notre arbre de recherche. Selon nos heuristiques et nos optimisations détaillées ici, et compte tenu de notre environnement de développement (*Python*), il apparaît en pratique qu'une profondeur égale ou supérieure à quatre coups n'est pas réalisable dans le temps imparti. On voit effectivement que le temps pris pour une profondeur de quatre (voir tableau IV) dépasse amplement le 36 secondes par coup allouées. Cette limite de profondeur consiste en une stratégie de recherche de Type **A** [2] dans laquelle la recherche s'arrête à une profondeur maximale.

TABLE IV
ÉTATS VISITÉS ET DURÉE MOYENNE PAR COUP SELON LE MODE DE JEU POUR UNE PROFONDEUR DE QUATRE AVEC TOUTES LES OPTIMISATIONS

Mode de jeu	**États visités**	**Durée (s) par coup**
Classique	269315	85.7
Alien	229242	79.9

Nous avons aussi introduit une profondeur dynamique dans notre agent afin d'accélérer son jeu. Les six premiers coups dans le jeu classique sont analysés à une profondeur de 1, permettant ainsi à notre agent de jouer plus rapidement. Cette optimisation est rendue possible

par la simplicité des débuts de parties classiques, qui se concentrent souvent sur une course vers le centre et ne nécessitent pas une analyse approfondie des coups futurs. De plus, si nous détectons qu'il nous reste moins de 100 secondes pour jouer, nous réduisons également la profondeur à 1 pour limiter le temps pris par l'agent. Pour le reste du jeu, notre profondeur demeure fixée à trois.

Pour finir, nous avons tenté d'ajouter à cette profondeur dynamique avec un principe d'agent défensif. En effet, si l'arbre a une profondeur impaire, nous aboutissons à l'étude de l'une de nos actions, tandis que dans le cas d'une profondeur paire, c'est une action de l'adversaire qui est étudiée. Selon le cas, dû à l'algorithme *Minimax* utilisé, l'agent sera plus agressif ou défensif, respectivement. Ainsi, nous avons certains agents qui en plus d'une profondeur dynamique ont une profondeur défensive, c'est-à-dire tombe en profondeur deux lorsqu'il gagne afin de jouer de manière plus défensive.

III. Résultats

Nous avons implémenté plusieurs agents en fonction des combinaisons d'heuristiques et de profondeurs d'arbre (voir tableau V). Il est important de noter que la performance des divers agents varie en fonction des combinaisons d'heuristiques utilisées, ainsi que de la profondeur de notre arbre de recherche. Ces disparités de performance seront étudiées dans cette section.

TABLE V
Les différents agents implémentés et leurs spécificités

Agent	Profondeur	Heuristiques
A	1	Aléatoire
B	1	Les 3 premières
C	2	Les 3 premières
D	3 (Dynamique)	Les 3 premières
E	3 (Défensif)	Toutes
F	3 (Dynamique)	Toutes
G	3 (Défensif)	Les 3 premières

Afin de tester l'implémentation de plusieurs agents intelligents afin de trouver le meilleur, nous avons développé un simulateur de parties. Celui-ci simule plusieurs parties (ce nombre est un paramètre au choix entré par l'utilisateur) entre deux joueurs et exporte ensuite les résultats dans un fichier *Excel*. Pour ce faire, deux parties sont simulées en position classique et deux en position alien, chacune des deux avec un joueur différent qui commence. Nous ne simulons pas plus que deux parties puisque nos agents sont déterministes dus à l'ordonnancement de nos actions, et donc une partie avec les deux mêmes agents sur le même plateau sera identique. Les parties suivantes sont simulées sur des positions de plateau aléatoires, mais symétriques (voir figure 5). Encore une fois, deux fois pour chaque configuration, avec chaque joueur commençant une fois.

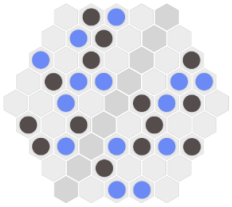

Fig. 5. Exemple de plateau aléatoirement généré

Afin d'obtenir de bonnes statistiques et une idée objective du meilleur agent, nous avons simulé 20 parties entre chaque combinaison de joueurs possibles (21 affrontements au total), pour un total de 420 parties simulées. Les résultats moyens de ces simulations sont publiés dans le tableau VI.

TABLE VI
Résultats moyens des simulations

Agent	Score	Temps restant	Pourcentage de victoire
A	0	899.98	0.00
B	1.86	897.2	44.33
C	2.78	859.54	66.67
D	2.58	537.91	69.44
E	2.08	501.63	41.67
F	2.67	558.64	**100.0**
G	2.47	518.67	52.78

Une autre façon de visualiser ces résultats, plus spécifiquement les victoires, est sous forme de graphe, tel que vu ci-dessous dans la figure 6. Une flèche de A vers B signifie que A a perdu contre B en moyenne (donc sur 20 parties simulées, il en perd 11 ou plus).

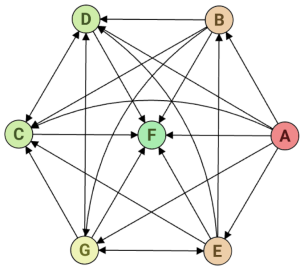

Fig. 6. Victoires et défaites des différents agents

En analysant ces résultats, on voit plusieurs choses. Premièrement, la stratégie défensive ne semble pas fonctionner. En effet, pour un agent avec les mêmes heuristiques, la version défensives est moins bonne que la version dynamique. L'agent E perd environ 28% de taux de victoire sur le D et perd contre lui, alors que l'agent G diminue le taux de victoire d'environ de moitié sur l'agent F. Ensuite, nos agents ont amplement le temps de finir leur partie dans le temps alloué, ce qui est sécurisant. Finalement, on remarque que le meilleur agent est l'agent **F**, c'est-à-dire un agent avec une profondeur dynamique (mais pas défensive) avec toutes les heuristiques expliqués plus haut. Cet agent a gagné toutes ces parties et a un score moyen de 2.67, tout en jouant en moins de 400 secondes par partie en moyenne (temps restant de 558.64 secondes sur 900).

IV. Discussion

Nous nous sommes concentrés sur le développement de plusieurs agents, ce qui nous a permis d'explorer plusieurs théories, d'isoler des modifications pour évaluer leurs impacts, et enfin, de déterminer notre meilleur agent. Notre approche systématique nous permet d'identifier de manière certaine le meilleur parmi nos agents, ce qui constitue un avantage significatif. De plus, nous sommes d'avis que notre façon de faire, ancrée dans une méthodologie quantitative et scientifique, réduit considérablement la marge d'incertitude. Nous sommes également satisfaits de nos optimisations de vitesse, nous permettant de jouer confortablement bien au-dessus du seuil de temps alloué. Cependant, plusieurs éléments pourraient être bénéfiques pour améliorer nos agents.

Tout d'abord, comme discuté précédemment, nous attribuons des poids à nos heuristiques de manière relativement aléatoire, basée sur notre intuition, ce qui n'est évidemment pas idéal. Nous sommes convaincus qu'un avantage notable pour notre approche aurait été d'apprendre ces poids. Si nous avions disposé de suffisamment de données (ce qui a finalement été le cas avec un grand nombre d'environ 1500 parties simulées au cours de la session), nous aurions pu apprendre les poids grâce à un apprentissage supervisé. Cette approche aurait impliqué des changements systématiques de poids, l'évaluation des scores résultants, ainsi que la détermination du succès du match. Bien que cette méthode aurait pris du temps et aurait nécessité la conversion des parties simulées en données exploitables, elle aurait probablement considérablement amélioré les performances de nos agents. De plus, l'implémentation d'une telle méthode nous semble intéressante.

Ensuite, une autre amélioration possible pour nos agents aurait été de tenter de développer un agent réactif, c'est-à-dire un agent capable de détecter le style de jeu adverse et d'ajuster ses heuristiques en conséquence. Bien que cela soit difficile à mettre en œuvre, nous voyons un potentiel d'amélioration considérable. En effet, lors de la compétition du cours, notre agent n'a jamais perdu lors de la phase de poule pour finalement être éliminé dès le premier tour de la ronde finale. De plus, une équipe contre laquelle nous avions gagné en phase de poule a atteint le top quatre. Ainsi, il est évident que les propriétés de victoire de notre agent ne sont pas transitives, une victoire contre un agent ne garantissant pas forcément une victoire contre les agents ayant perdu contre lui, par exemple. Un agent capable de s'adapter, au moins partiellement, au style de jeu adverse pourrait ainsi peut-être accroître cette transitivité et éviter de perdre contre un agent dont le style de jeu est incompatible avec le nôtre.

Finalement, en prenant en compte le temps restant en moyenne pour nos agents, il aurait pu être intéressant d'approfondir la recherche jusqu'à une profondeur de quatre en cas de prise de pièces ou d'autres états prometteurs. Cette approche s'inscrit dans la lignée de la théorie des stratégies de type B [2], visant à explorer plus en profondeur les états nécessitant une analyse approfondie.

V. Conclusion

En conclusion, nous avons présenté le développement d'un agent intelligent spécifiquement conçu pour le jeu d'Abalone. Nous avons élaboré la théorie sous-jacente à sa performance et détaillé notre méthodologie systématique, laquelle inclut la simulation d'un grand nombre de parties. En analysant quantitativement ces données ainsi que le comportement de nos agents, nous avons réussi à identifier un agent surpassant ses pairs en termes de performance. Nous sommes satisfaits du développement de notre agent, en particulier en ce qui concerne la méthodologie adoptée, laquelle limite significativement le recours au hasard. Les enseignements tirés de cette compétition ont été précieux, et la création d'un agent performant dans le contexte d'Abalone représente une réussite gratifiante. Ce premier pas dans la conception d'agents intelligents spécialisés pour des jeux spécifiques a suscité en nous le désir de poursuivre ces efforts et de développer d'autres agents.

REFERENCES

[1] Wikipédia. (2023). Game Complexity. [En ligne]. Disponible : https://en.wikipedia.org/wiki/Game_complexity

[2] Cappart, Q. (2023). INF8175 - Intelligence artificielle : méthodes et algorithmes. [En ligne]. Disponible : https://moodle.polymtl.ca/course/view.php?id=2667

[3] Aichholzer, O., Aurenhammer, F., & Werner, T. (2002). Algorithmic fun - Abalona. Telematik. [En ligne]. Disponible : http://www.ist.tu-graz.ac.at/staff/aichholzer/research/rp/abalone/tele1-02_aich-abalone.pdf