Agent d'intelligence artificielle pour le jeu Abalone

Quentin Lao
Polytechnique Montréal
quentin.lao@polytechnique.org

1 INTRODUCTION

Ce projet a été réalisé dans le cadre du cours INF8175 - Méthodes et algorithmes d'intelligence artificielle. Il s'intéresse au célèbre jeu de plateau Abalone. Abalone se joue en 1 contre 1 au tour par tour où chaque joueur dispose initialement du même nombre de billes sur le plateau et cherche, pour gagner, à éjecter le plus de billes adverses. L'objectif de ce projet est de concevoir un modèle d'intelligence artificielle (un agent) capable de jouer à Abalone.

Figure 1: Jeu d'Abalone

2 CHOIX DE L'ALGORITHME : MINIMAX

2.1 Nature du jeu Abalone

Abalone est un jeu à somme nulle qui oppose deux joueurs qui jouent à tour de rôle. Ainsi, à leur tour de jeu, chaque joueur va essayer de prendre la meilleure décision pour améliorer sa position ce qui revient à choisir l'action qui place son adversaire dans la moins bonne situation. On peut alors placer Abalone dans la même classe de jeu que les Échecs ou le jeu de Go.

2.2 Comparaison à d'autres jeux

L'environnement du jeu qui nous a été fourni pour cette compétition est une version légèrement simplifiée de la version originale avec certaines actions officielles qui sont interdites ici (les mouvements de côté par exemple). Empiriquement[1], cette version du jeu compte en moyenne 60 actions possibles pour un état donné. Pour se donner un ordre d'idée, le nombres d'actions possibles aux Échecs avoisinent les 40 coups contre 300 pour le jeu de Go [2]. La complexité d'Abalonne semble se rapprocher de celle des Échecs. La célèbre intelligence artificielle des Échecs Stock Fish repose sur un algorithme Minimax très optimisé.

[1] en effectuant plusieurs parties et moyennant le nombre d'actions possibles

Ainsi, compte tenu de la nature du jeu Abalone et de sa complexité, il semble intéressant d'implémenter un algorithme Minimax pour la résolution du projet.

3 DÉROULÉ DU PROJET

L'algorithme Minimax seul ne permet pas d'avoir des résultats satisfaisants en terme de temps de calcul et d'efficacité. Après avoir **implémenter la forme de base de l'algorithme Minimax**, c'est-à-dire parcourir tous les états jusqu'à une profondeur donnée, la majeure partie du projet est consacrée à optimiser cette dernière. On distingue deux types d'optimisations : les optimisations d'efficacité qui visent à réduire le temps de calcul d'un agent et les optimisations de qualité qui permettent à l'agent de prendre de meilleures décisions. Le projet se divise alors en deux parties pour chacun de ces types d'optimisations. Il convient tout d'abord d'améliorer l'efficacité de l'agent avant de s'intéresser aux qualités de ses décisions.

Tout d'abord, il est possible de rendre l'algorithme de Minimax plus rapide grâce à la classique optimisation *alpha-beta pruning* qui réduit considérablement les noeuds à explorer. **La première étape du projet est alors d'implémenter un *alpha-beta pruning***. Une autre importante optimisation consiste à parcourir les actions possibles d'un même état dans un certain ordre de sorte à tester en priorité les actions qui ont du potentiel. Cette technique utilise une fonction pour estimer chaque action et permet en fait de rendre l'*alpha-beta pruning* encore plus efficace. **La deuxième étape est alors d'implémenter l'ordonnance des actions en priorisant celles qui ont du potentiel**. En l'état, il est possible que l'algorithme parcourt plusieurs fois les mêmes noeuds. Il est alors intéressant de garder en mémoire les états déjà parcourus pour éviter de les évaluer plusieurs fois. **La troisième étape du projet se concentre sur la conception d'une table de transposition qui permet de stocker les états déjà visités**. Ces étapes permettent de réduire le temps de recherche de l'agent.

Une fois l'efficacité de l'agent optimisé, on peut améliorer sa performance de jeu, c'est-à-dire la qualité de ces décisions. L'algorithme Minimax s'appuie sur une stratégie simple : en tant que joueur, pour maximiser mon gain potentiel, je dois minimiser celui de mon adversaire en supposant que lui aussi réfléchit comme moi. Pour guider cette stratégie, il est important de concevoir une fonction d'utilité (ou fonction d'évaluation) qui détermine "la valeur" d'un état (i.e un peu/beaucoup avantage noir ? ou un peu/beaucoup avantage blanc ?). Une bonne fonction d'utilité peut considérablement améliorer la qualité des décisions d'un agent. Il convient donc **d'optimiser la fonction d'utilité pour la rendre plus représentative de la valeur d'un état**. Néanmoins, la fonction d'utilité ne suffit pas. Elle ne reflète que la valeur présente de l'état. Il se pourrait que l'état soit "instable" (par exemple, si l'adversaire peut enchainer

des prises aux prochains tours) ce qui mènerait à grandement dégrader la position du joueur. Il est alors intéressant de n'arrêter la recherche que lorsqu'un état dit "tranquille" ou *quiescent* est atteint. Dans notre cas, un état sera *quiescent* si aucune bille ne peut être éjectée. **L'étape suivante vise ainsi à intégrer ce mécanisme d'arrêt de la recherche sur des états tranquilles**.

Enfin, **la dernière partie du projet est consacrée à la gestion du temps** pour que l'agent puisse organiser au mieux son temps dans les 15 minutes imparties.

4 OPTIMISATIONS D'EFFICACITÉ

L'objectif est de partir d'un algorithme Minimax simple et d'intégrer petit à petit des optimisations pour le rendre plus rapide.

4.1 Alpha-beta pruning

Pour réduire le nombre d'états visités, on peut avoir recours à l'*alpha-beta pruning*. Cela consiste à sans cesse garder deux bornes α et β lors du parcours de l'arbre pour discriminer des branches qui ne valent pas le coup d'être explorées. L'*alpha-beta pruning* est une optimisation classique et exacte (i.e qui donne le même résultat qu'un algorithme Minimax de base).

4.2 Ordre des actions visitées

L'ordre avec lequel un agent va évaluer chaque action a un impact sur sa rapidité à prendre une décision. En effet, si l'agent parcourt dès le début la meilleure action, alors l'*alpha-beta pruning* va être d'autant plus efficace car aura tendance à couper beaucoup de branches dans la suite de la recherche. Pour cela, il est nécessaire d'implémenter une autre fonction d'évaluation pour détecter rapidement les actions qui ont du potentiel. Cette fonction est nommée `guess_value` dans la classe `MyPlayer`. Cette nouvelle fonction d'évaluation rapide privilégie les actions qui :

- n'éjectent pas bêtement des billes alliées
- poussent des billes adverses
- sécurisent une bille qui était placée sur le bord
- mènent à un état où les billes alliées sont plus compactes. On utilise la fonction *Geometric score* proposée par Aichholzer, O., Aurenhammer, F., & Werner, T. [4]

4.3 Table de transposition

Lors de l'exploration, l'agent a de forte chance de tomber sur un état qu'il a déjà visité : un état peut être atteint avec des ordres de coups différents. Lorsqu'un état est visité et évalué pour la première fois, il est intéressant de le garder en mémoire dans une table de transposition pour éviter de le réévaluer une seconde fois. Cette table prend la forme d'un dictionnaire[2] où chaque clé correspond à un hash d'un état et la valeur à son évaluation. La fonction de hash utilisée est la fonction Zobrist [3]. L'implémentation d'une telle table dans le contexte de l'algorithme Minimax avec *alpha-beta pruning* est un peu délicate. Il faut comprendre que l'*alpha-beta pruning* peut arrêter l'exploration d'un noeud à n'importe quel moment et donc on ne connaitra pas la valeur exacte de ce noeud, on aura juste une borne inférieure ou supérieure. Par exemple, si un des fils d'un noeud MIN a une valeur inférieure à β, alors le noeud MIN sera coupé (par *alpha-beta pruning*); on n'a alors qu'une borne supérieure de ce noeud MIN (qui est la valeur de ce fils). Il faut tenir compte de ça dans l'implémentation de la table. Une implémentation en C est donnée dans le contexte de l'algorithme Minimax [1].

[2]Un dictionnaire permet d'accéder à une valeur plus rapidement qu'une liste

4.4 Bilan de ces optimisations

La table 1 présente la durée moyenne d'un coup pour chaque optimisation avec comme référence un algorithme basique Minimax avec *alpha-beta pruning* (i.e on considère sa durée moyenne de 1 et on compare toutes les optimisations avec celle-ci)[3]. Ces résultats ont été générés en mesurant la durée de 600 coups, ce qui permet de calculer la durée moyenne d'un coup et un intervalle de confiance à 95% pour chaque optimisation.

Table 1: Durée moyenne d'un coup des optimisations ($\alpha\beta$ = *alpha-beta pruning*; OA = Ordonnance des actions; TT = Table de transposition)

	$\alpha\beta$	$\alpha\beta$+OA	$\alpha\beta$+TT	$\alpha\beta$+OA+TT
Durée par coup	1 ± 0.098	0.767 ± 0.069	0.722 ± 0.061	0.54 ± 0.048
***p*-valeur avec $\alpha\beta$**	1	$< 10^{-3}$	$< 10^{-5}$	$< 10^{-14}$

On constate une nette amélioration en rapidité de l'agent tout au long des étapes d'optimisation. Les faibles p-valeurs (toutes inférieures à 10^{-2}) témoignent de la significativité de ces écarts de temps. Avec toutes les optimisations faites ($\alpha\beta$+OA+TT), un agent met quasiment deux fois moins de temps à jouer que sans elles.

5 FONCTION D'UTILITÉ

L'algorithme Minimax nécessite la conception d'une fonction d'utilité $\mathcal{U}$ qui évalue l'état d'un plateau de jeu à un instant donné. Ainsi, on dira que l'état s est *avantage blanc* si $\mathcal{U}(s) > 0$, *avantage noir* si $\mathcal{U}(s) < 0$, et *équilibré* si $\mathcal{U}(s) = 0$. Cette fonction est utilisée pour évaluer les états-feuilles de l'arbre de recherche de l'agent. Pour un état terminal (i.e pour lequel la partie s'arrête), on attibue une utilité de 100 si les blancs gagnent et -100 si les noirs gagnent. Néanmoins, il est possible qu'une feuille ne soit pas terminal, auquel cas il faut estimer une valeur pour cet état. Il faut alors concevoir une fonction heuristique $\mathcal{H}$.

5.1 Fonction heuristique

Pour définir une telle fonction heuristique, on détermine N caractéristiques numériques f_i du plateau. L'état s d'un plateau est alors évalué en faisant une somme pondérée de ses caractéristiques :

$$\mathcal{H}(s) := \sum_{i=1}^{N} \omega_i f_i(s) = {}^t\Omega\mathbf{F}(\mathbf{s})$$

[3]En fait, mon but était de chronométrer chaque agent à chaque optimisation pour voir le gain de temps. Mais à ce stade-là du projet, j'avais une fonction d'utilité plus simple que celle retenue à la fin du projet et donc plus rapide à exécuter. Donc ça n'a pas vraiment de sens de savoir que telle optimisation permet de gagner tant de secondes (gain de temps absolu). Il me parait plus pertinent d'observer le gain de temps relatif, à savoir quel est poucentage de temps gagné grâce à telle optimisation

avec $\Omega := (\omega_1, \ldots, \omega_N), \mathbf{F(s)} := (f_1(s), \ldots, f_N(s)) \in \mathbb{R}^N$.

Tout l'enjeu est de trouver des caractéristiques pertinentes et de déterminer les bons poids Ω.

5.2 Les caractéristiques utilisées

Pour un état s, on note $\mathcal{W}_s$ (resp. $\mathcal{B}_s$) l'ensemble des billes blanches (resp. noires) du plateau. $N(b)$ désigne l'ensemble des voisins de même couleur d'une bille $b \in \mathcal{W}_s \cup \mathcal{B}_s$, c le centre du plateau et d la distance de Manhattan. En pratique, on passe en coordonnées cubiques [5] particulièrement adaptées aux hexagones. Ces coordonnées facilitent le calcul de la distance de Manhattan et l'accès aux voisins d'une case.

Le nombre de billes de différences. L'intuition derrière cette caractéristique est simple. Il s'agit simplement de voir quel joueur a éjecté le plus de billes adverses. Par exemple, si les blancs ont moins de billes sur le plateau, alors c'est plutôt a priori avantage noir. On définit alors la différence de nombre de billes des deux joueurs.

$$f_1(s) := |\mathcal{W}_s| - |\mathcal{B}_s| \in [-5, 5]$$

Le nombre de billes sur les bords. Avoir des billes sur les bords est mauvais car cela rend le joueur très vulnérable. Il peut facilement se faire éjecter une bille. On peut alors calculer la différence des proportions de billes sur les bords de chaque joueur.

$$f_2(s) := \frac{|\{b \in \mathcal{B}_s \ : \ d(b,c) = 4\}|}{|\mathcal{B}_s|} - \frac{|\{b \in \mathcal{W}_s \ : \ d(b,c) = 4\}|}{|\mathcal{W}_s|} \in [-1, 1]$$

Centralité des billes. On dit souvent de ce genre de jeu qu'il est important de contrôler le centre. En effet, dans Abalone, contrôler le centre permet de jouer à la fois défensif (car difficile de se faire éjecter des billes) et offensif (car incite l'adversaire à se placer sur les bords ce qui peut créer des opportunités d'attaque). On définit alors la différence des distances moyennes au centre de chaque joueur.

$$f_3(s) := \frac{1}{4}\left(\frac{1}{|\mathcal{B}_s|}\sum_{b\in\mathcal{B}_s} d(b,c) - \frac{1}{|\mathcal{W}_s|}\sum_{b\in\mathcal{W}_s} d(b,c)\right) \in [-1, 1]$$

Cohésion des billes. Garder ses billes collées les unes des autres fait partie des stratégies de bases d'Abalone. Avoir des amas de billes crée des avantages numériques ce qui donne du fil à retordre à l'adversaire pour pousser nos billes (aspect défensif) et nous facilite à pousser les siennes (aspect offensif). On propose deux caractéristiques pour représenter la cohésion. Ne sachant pas vraiment laquelle choisir à ce stade, on conserve les deux. L'une se fonde sur le nombre moyen de voisins de chaque bille et l'autre sur une distance intra-couleur.

$$f_4(s) := \frac{1}{6}\left(\frac{1}{|\mathcal{W}_s|}\sum_{b\in\mathcal{W}_s} |N(b)| - \frac{1}{|\mathcal{B}_s|}\sum_{b\in\mathcal{B}_s} |N(b)|\right) \in [-1, 1]$$

$$f_5(s) := \frac{1}{2|\mathcal{B}_s|(|\mathcal{B}_s|-1)}\sum_{b,b'\in\mathcal{B}_s} d(b,b') - \frac{1}{2|\mathcal{W}_s|(|\mathcal{W}_s|-1)}\sum_{b,b'\in\mathcal{W}_s} d(b,b')$$

Menaces. Cette caractéristique est purement offensive. Elle est définie par les différences de billes menacées (i.e qui peuvent se faire expulser au prochain tour de jeu de l'adversaire) de chaque joueur.

$$f_6(s) := |\{b \in \mathcal{B}_s \ : \ b \text{ menacé en 1 coup}\}| - |\{b \in \mathcal{W}_s \ : \ b \text{ menacé en 1 coup}\}| \in [-10, 10]$$

5.3 Optimisations des poids Ω

Les poids Ω quantifient l'importance relative des caractéristiques f_i. Ces poids sont cruciaux car influencent grandement le comportement de l'agent. Par exemple, si la caractéristique "menaces" est beaucoup trop forte par rapport à "nombre de billes de différences", alors on peut imaginer un agent qui ne ferait que créer des menaces en permanence sans jamais éjecter de billes adverses. Ce ne serait pas une bonne heuristique car *in fine*, c'est le nombre de billes éjectées qui est comptabilisé.

Une grande partie de ce projet est alors consacré à déterminer une configuration de poids optimale Ω^* pour maximiser les chances de victoires. On limite nos recherches à $\Omega \in ([0, 2] \cap 0.125\mathbb{N})^N$. On va procéder à une recherche locale pour déterminer des configurations candidates (figure 2).

5.3.1 Recherche locale. Une recherche locale consiste à partir d'une configuration de poids initial Ω_0 et de choisir de manière itérative une configuration voisine qui améliore la configuration courante. On définit le **voisinage** d'une configuration de poids $\Omega \in ([0, 2] \cap 0.125\mathbb{N})^N$ comme étant tous les Ω' tels que $\|\Omega' - \Omega\|_\infty \leq \alpha$ i.e dont les poids diffèrent d'au plus α. De plus, on munit chaque voisinage d'un préordre $\succ$ défini par "$\Omega' \succ \Omega$ *si et seulement l'agent de configuration* Ω' *bat en moyenne l'agent de configuration* Ω". En particulier, on fait l'hypothèse forte de transitivité locale : si 3 configurations voisines Ω, Ω' et Ω'' sont telles que $\Omega \succ \Omega'$ et $\Omega' \succ \Omega''$, alors $\Omega \succ \Omega''$. Pour notre recherche locale, on **sélectionne la première configuration rencontrée qui bat la configuration courante** (au sens de la relation $\succ$).

En pratique, pour déterminer si $\Omega \succ \Omega'$, on fait affronter deux agents de configuration respective Ω et Ω' sur 5 matchs : 2 matchs avec des positions de départ classiques, 2 matchs avec des positions de départ de type Alien et 1 match avec une position de départ aléatoire (classique ou Alien). Une victoire rapporte 3 points à un agent, un match nul 1 point et une défaite 0. L'agent avec le plus de points à l'issue des 5 matchs est considéré comme meilleur au sens de $\succ$.

Le risque de la recherche locale est de rester bloqué dans un maximum local. Pour palier ce problème, on utilise une **méthode de redémarrage** ou *restart* qui consiste à arrêter la recherche pour en relancer une nouvelle avec une autre configuration initiale. Dans notre cas, parcourir tous les voisins d'une configuration de poids prendrait beaucoup trop de temps. On considère alors qu'une configuration est un maximum local si l'agent courant a battu (toujours au sens de $\succ$) de manière consécutive 5 de ses voisins, auquel cas on peut *restart* avec une autre configuration initiale. De plus, si on dispose d'une pool de configurations "pas trop mauvaises", on peut

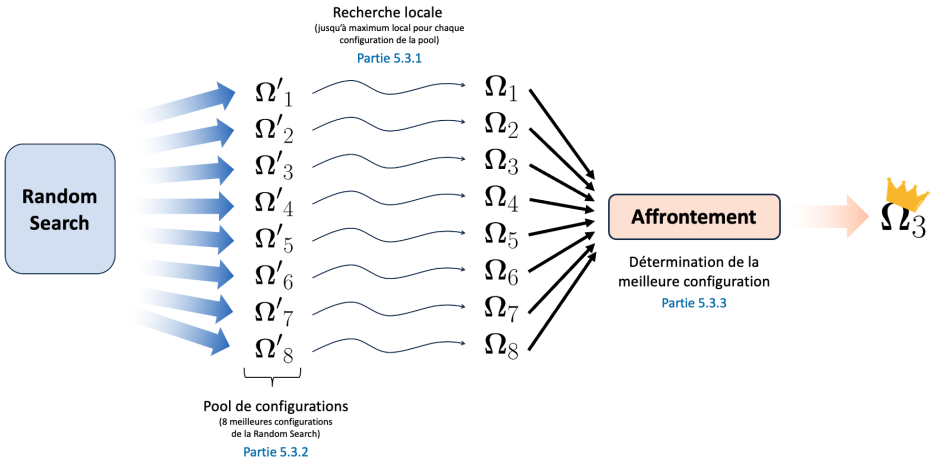


Figure 2: Les différentes étapes de l'optimisation des poids

choisir à chaque *restart* une configuration initiale parmi cette pool. La partie suivante explique comment on crée une telle pool.

L'algorithme 1 en appendix présente le pseudo-code de la recherche locale implémentée. En pratique, $\alpha = 0.125$.

5.3.2 Pool de configurations. On cherche à présent à construire une pool de configurations qui est utilisée dans la recherche locale. Dans cette pool, on aimerait que les configurations ne soient pas trop mauvaises de sorte à fournir à la recherche locale des configurations initiales pertinentes au moment des *restart*. En s'inspirant du tuning d'hyperparamètres en apprentissage supervisé, on procède à une Random Search. Le principe consiste à tester des configurations aléatoires. Chaque configuration aléatoire Ω effectue 10 matchs contre d'autres configurations tirées aléatoirement et on mesure le score de Ω (avec le même comptage : victoire = 3 points, nul = 1 point et défaite = 0 point). Dans le projet, 40 configurations aléatoires ont été testés, les 8 meilleures ont été retenues pour consitituer la pool.

L'algorithme 2 en appendix présente le pseudo-code de construction de la pool.

5.3.3 Sélection de la configuration de poids optimale. À l'issue de la recherche locale, on obtient ainsi 8 configurations candidates qui dérivent chacune de l'une des configurations de la pool. La table 2 resense les configurations obtenues. Pour chacun d'entre elles, on marque en gras les caractéristiques qui dominent ce qui est révélateur de leur stratégie. Voici quelques observations intéressantes de la table 2 :

- La majorité des configurations de poids porte une grande importance à la centralité.
- La plupart des agents ont une stratégie défensive en gardant une grande cohésion, en évitant les billes aux bords et en accordant peu d'imporantance à la création de menaces : Ω_1, Ω_3, Ω_5, Ω_6.
- Il y aussi 2 agents aggressifs qui veulent créer beaucoup de menaces : Ω_4 et Ω_8.
- Les caractéristiques "Cohésion 1" et "Cohésion 2" n'ont jamais des poids tous les deux élévés en même temps. Cela peut être du à une redondance : l'agent choisit d'utiliser soit l'une, soit l'autre.

Table 2: Les 8 configurations candidates Ω_i après recherche locale

	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6
	Nombre de billes	Billes au bord	Centralité	Cohésion 1	Cohésion 2	Menaces
Ω_1	0.875	0.5	**2**	0.5	**1.625**	0.125
Ω_2	**1.125**	0.125	**1.125**	0.25	0.125	0.05
Ω_3	**1.125**	**1.375**	**1.75**	0.05	**1.25**	0.25
Ω_4	**1.25**	0.375	**1.25**	0.25	0.05	**1**
Ω_5	**1.25**	0.375	**1.375**	**1.125**	0.05	0.05
Ω_6	**1.625**	**1.5**	**2.0**	0.625	**1.75**	0.375
Ω_7	**2**	0.625	0.375	0.625	0.05	0.375
Ω_8	0.25	0.125	**1.75**	1	0.5	**1.75**

La dernière étape est de choisir la meilleure d'entre elles. La manière la plus simple est de les faire s'affronter les unes contre les autres et analyser les taux de victoires. On fait affronter deux configurations sur 10 matchs : 4 avec une position départ classique, 4 avec une position départ de type Alien et 2 aléatoires. On comptabilise les points de chaque agents de la même manière (3 points pour une victoire, 1 pour un match nul et 0 pour une défaite). Chaque configuration joue alors 70 matchs et on peut calculer leur score moyen. La table 3 montre le détails de chaque affrontement.

Dans la table 3, on observe plusieurs choses intéressantes :

- Les meilleurs poids sont Ω_3 avec un score total moyen de 2.4. En se référant à la table 2 pour l'expression de Ω_3, on constate que l'agent accorde une grande importance à la

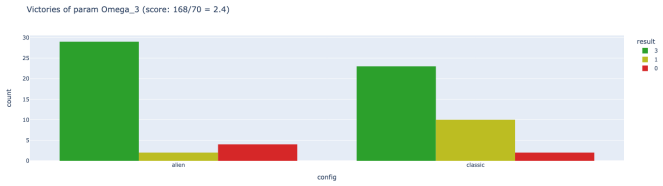
Table 3: Résultats des affrontements des 8 configurations candidates. Chaque case en ligne i et colonne j est de la forme *"(victoires de Ω_j, matchs nuls, victoires de Ω_i) : score de Ω_j contre Ω_i"*

	Ω_1	Ω_2	Ω_3	Ω_4	Ω_5	Ω_6	Ω_7	Ω_8
Ω_1	$\emptyset$	(3, 2, 5) : 1.1	(5, 3, 2) : 1.8	(4, 2, 4) : 1.4	(2, 1, 7) : 0.7	(6, 4, 0) : 2.2	(3, 4, 3) : 1.3	(0, 3, 7) : 0.3
Ω_2	(5, 2, 3) : 1.7	$\emptyset$	(8, 2, 0) : 2.6	(2, 5, 3) : 1.1	(4, 4, 2) : 1.6	(7, 3, 0) : 2.4	(0, 10, 0) : 1.0	(0, 6, 4) : 0.6
Ω_3	(2, 3, 5) : 0.9	(0, 2, 8) : 0.2	$\emptyset$	(2, 0, 8) : 0.6	(0, 3, 7) : 0.3	(2, 3, 5) : 0.9	(0, 1, 9) : 0.1	(0, 0, 10) : 0.0
Ω_4	(4, 2, 4) : 1.4	(3, 5, 2) : 1.4	(8, 0, 2) : 2.4	$\emptyset$	(0, 10, 0) : 1.0	(5, 2, 3) : 1.7	(1, 9, 0) : 1.2	(0, 7, 3) : 0.7
Ω_5	(7, 1, 2) : 2.2	(2, 4, 4) : 1	(7, 3, 0) : 2.4	(0, 10, 0) : 1.0	$\emptyset$	(3, 7, 0) : 1.6	(2, 4, 4) : 1.0	(0, 3, 7) : 0.3
Ω_6	(0, 4, 6) : 0.4	(0, 3, 7) : 0.3	(5, 3, 2) : 1.8	(3, 2, 5) : 1.1	(0, 7, 3) : 0.7	$\emptyset$	(0, 0, 10) : 0.0	(0, 0, 10) : 0.0
Ω_7	(3, 4, 3) : 1.3	(0, 10, 0) : 1	(9, 1, 0) : 2.8	(0, 9, 1) : 0.9	(4, 4, 2) : 1.6	(10, 0, 0) : 3.0	$\emptyset$	(0, 0, 10) : 0.0
Ω_8	(7, 3, 0) : 2.4	(4, 6, 0) : 1.8	(10, 0, 0) : 3.0	(3, 7, 0) : 1.6	(7, 3, 0) : 2.4	(10, 0, 0) : 3.0	(10, 0, 0) : 3.0	$\emptyset$
Total	(28, 19, 23) : 1.47	(12, 32, 26) : 0.97	(52, 12, 6) : 2.4	(14, 35, 21) : 1.1	(17, 32, 21) : 1.19	(43, 19, 8) : 2.11	(16, 28, 26) : 1.09	(0, 19, 51) : 0.27

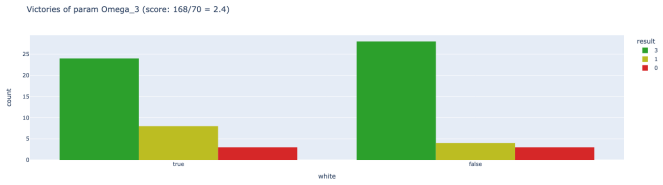
centralité, à la cohésion des billes et à limiter ses billes au bord et relativement peu d'importance à la création de menaces.

- Les stratégies de contrôle du centre sont les meilleures : les 3 meilleures configurations (Ω_1, Ω_3 et Ω_6) ont leur centralité ω_3 qui domine.
- Les stratégies purement offensives ne sont pas performantes : les configurations Ω_4 et Ω_8 accorde une grande importance aux menaces (ω_6) mais n'obtiennent que de faible score. Néanmoins, la configuration Ω_4 est un peu plus équilibrée que Ω_8 car elle tient compte de la différence du nombre de billes et fait plus attention à ses billes au bord; Ω_4 a d'ailleurs un score plus élevé que Ω_4.

On choisit alors la configuration de poids Ω_3. Un tel agent est assez équilibré avec un taux de victoires légèrement supérieur s'il joue noir et avec la position de départ Alien comme le montre la figure 3. **Il bat l'agent random avec 100% de victoire.**



(a) selon la position de départ



(b) selon la couleur de Ω_3

Figure 3: Nombres victoires, matchs nuls et défaites de Ω_3

6 GESTION DU TEMPS DE L'AGENT

La dernière tâche du projet est la gestion du temps. Dans le cadre de ce projet, chaque joueur dispose de 15 minutes pour jouer 25 coups. Si aucun joueur n'a ejecté 6 billes de l'adversaire à l'issue des 2×25 coups, la partie s'arrête et le joueur avec le plus de billes sur le plateau gagne. Cette contrainte de temps permet de borner la profondeur de recherche. Ainsi, l'agent ne va pas chercher à la même profondeur s'il lui reste 200 secondes ou 1000 secondes pour jouer ses 10 coups restants. Dans la suite, pour un joueur donné, on note c le nombre de coups restants pour atteindre les 25 coups et t le temps restant (en secondes) pour jouer ses coups.

Dans notre cas, à chaque tour de jeu, l'agent devra estimer s'il peut se permettre de chercher à profondeur 4. Sinon, il jouera son coup à profondeur 3. On note $\bar{t_3}$ et $\bar{t_4}$ la durée moyenne de recherche respectivement à profondeur 3 et 4. Au début d'un tour de jeu i, l'idée est de se projeter un coup après en estimant le temps qu'il resterait si le joueur cherche à profondeur 4 ($t - \bar{t_4}$ secondes restantes après le i-ème tour) puis comparer ce temps à celui que le joueur prendrait s'il jouait les coups restants à profondeur 3 ($(c-1)\bar{t_3}$ secondes). S'il a encore le temps de jouer à profondeur 3, alors l'agent peut se permettre de joueur à profondeur 4 au tour de jeu i. Autrement dit, s'il reste t secondes à jouer pour c coups restants, l'agent recherchera à une profondeur $p(c,t)$ définie par :

$$p(c,t) := \begin{cases} 4 & \text{si } (c-1)(\bar{t_3} + 3\sigma_3) \leq t - \bar{t_4} - 3\sigma_4 \\ 3 & \text{sinon} \end{cases}$$

On remarque qu'on a ajouté les écart-types des durées de recherche à profondeur 3 et 4 qu'on note respectivement σ_3 et σ_4, afin d'avoir une marge de sécurité. Afin d'accélerer encore plus, si aucune bille de couleur différente se touche sur le plateau, alors l'agent effectue une recherche seulement à profondeur 3. On juge que dans ce cas-là, il n'est pas nécessaire de chercher à profondeur 4 car si le joueur ne choisit pas la meilleure action, cela n'aura pas un grand impact sur le reste de la partie.

La figure 4 trace la fonction p en y faisant apparaitre la frontière de décision entre 3 et 4 ainsi qu'un exemple de partie de jeu en position de départ classique. La partie de jeu commence par une phase de profondeur à 3 (aucune bille ne touche une bille adverse), puis une longue phase de profondeur à 4 suivie de phases alternées rapides de profondeur 3 et 4. La partie se termine bien dans le temps imparti.

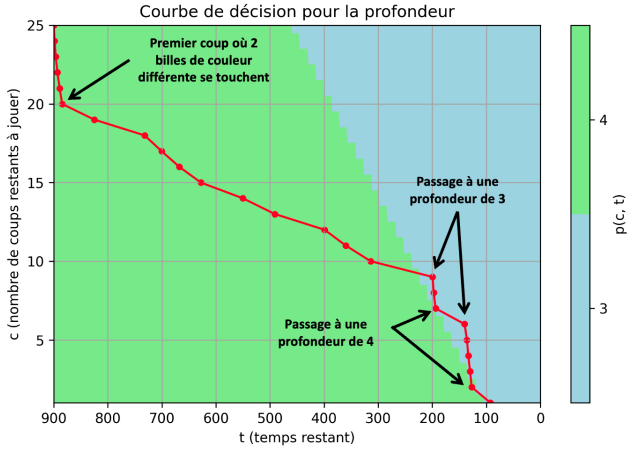


Figure 4: Choix de la profondeur en fonction du temps restant (t) et du nombre de coups restants (c). En rouge, le déroulé d'une partie. ($\bar{t}_3 = 3.025$; $\bar{t}_4 = 40$; $\sigma_3 = 2.35$; $\sigma_4 = 20$)

7 DISCUSSION

L'agent ainsi développé présente un certain nombre d'avantages. Il utilise un algorithme basique (Minimax) efficace où son temps d'exécution a été optimisé par le biais plusieurs procédés comme l'*alpha-beta pruning* et les tables de transpositions. Il est également performant avec une fonction heuristique avancée qui prend en compte plusieurs aspects du jeu pour évaluer les positions. Cette fonction a d'ailleurs été longuement optimisée permettant d'attribuer à chaque feature son meilleur poids. L'utilisation d'une fonction heurisitique permet une meilleure compréhension et interprétabilité des décisions de l'agent.

Une première piste d'amélioration possible serait d'adapter sa stratégie selon la position de départ ou le style de jeu de l'adverse. La section 5.3.1 a permis d'exhiber 8 configurations d'heuristiques, chacune avec un style de jeu différents et un taux de victoires différents selon l'adversaire. Une idée serait alors d'introduire un mécanisme d'adaptation, particulièrement intéressant en tournois : le premier match permet à l'agent de cerner le style de jeu de l'adversaire pour lui permettre de changer directement les poids de sa fonction heuristique. Une autre piste d'amélioration serait d'intégrer par exemple des stratégies d'ouverture fondée sur une base de données nourrie par des milliers et des milliers de parties réelles, à l'instar de Stockfish aux Échecs. On peut imaginer des ouvertures typiques à Abalone qui consisteraient directement à pousser le maximum de billes au centre. Pour optimiser au mieux, on pourrait aussi discuter avec des professionnels du jeu. Cette amélioration lui permettrait de gagner du temps en début de partie pour lui permettre d'allouer plus temps dans la suite.

Il ne faut pas oublier qu'on travaille sur une version simplifiée du jeu d'Abalone avec des actions possibles en moins. Ainsi, cet agent voit très vite ses limites quand il s'agira de l'adapter au jeu complet. Face à la complexité croissante de Minimax, on pourrait envisager d'autres algorithmes de recherche plus adaptés à des jeux d'envergure plus grande comme *Monte Carlo Tree Search*.

REFERENCES

[1] Web archive. 2007. https://web.archive.org/web/20071031100051/http://www.brucemo.com/compchess/programming/hashing.htm
[2] Bernard Victorri Bruno Bouzy. 1992. *LE JEU DE GO ET L'INTELLIGENCE ARTIFICIELLE*. https://helios2.mi.parisdescartes.fr/~bouzy/publications/AFIA92.article.pdf
[3] OpenGenus. .. https://iq.opengenus.org/zobrist-hashing-game-theory/
[4] Tino Werner Oswin Aichholzer, Franz Aurenhammer. 2002. *Algorithmic Fun - Abalone*. http://www.ist.tugraz.at/aichholzer/research/rp/abalone/tele1-02_aich-abalone.pdf
[5] Redblobgames. .. https://www.redblobgames.com/grids/hexagons/

APPENDIX

Algorithm 1 Recherche locale

```
α, pool
max_local ← ∅
while pool ≠ ∅ do
    Ω ← pool.pop()                          ▷ Configuration initiale
    N ← Voisinage_α(Ω)
    v ← 0                  ▷ Nb victoire de la configuration courante
    while v < 5 and N ≠ ∅ do                ▷ Doit battre 5 de ses voisins
        Ω' ← N.pop()
        if Ω' > Ω then                      ▷ Affrontement sur 5 matchs
            Ω ← Ω'
            N ← Voisinage_α(Ω)
            v ← 0
        else
            v ← v + 1
        end if
    end while
    max_local ← max_local ∪ {Ω}
end while
```

Algorithm 2 Pool de configurations

```
n                          ▷ Nombre de configuration aléatoire à tester
scores ← {}                ▷ Dictionnaire pour stocker les scores
for i = 0, ..., n − 1 do
    Ω ← get_Random_config()
    s ← 0                  ▷ Score de la configuration courante
    for j = 0, ..., 9 do
        Ω' ← get_Random_config()
        winner ← match(Ω, Ω')          ▷ Affrontement sur 1 match
        if winner = Ω then
            s ← s + 3
        else if winner ≠ Ω' then
            s ← s + 1
        end if
    end for
    scores[Ω] ← s
end for
scores ← sort_by_value(scores, desc_order)
```