Table des matières

Méthodologie

Algorithme Minimax

Nous avons choisi l'algorithme Minimax, renforcé par l'Alpha-Beta Pruning, comme base pour notre agent, car il est parfaitement adapté aux jeux à deux joueurs avec adversité directe, tels que Divercité. Cette approche permet à notre agent d'anticiper les actions de l'adversaire tout en maximisant ses propres gains.

Dans Divercité, les joueurs placent des pièces influençant directement les scores en fonction des ressources environnantes. Minimax est idéal pour ce type de jeu compétitif, car il simule les interactions stratégiques entre deux joueurs en évaluant plusieurs niveaux de coups possibles.

Alpha-Beta Pruning

L'ajout de l'Alpha-Beta Pruning optimise considérablement Minimax en réduisant le nombre de branches explorées, ce qui est essentiel dans le cadre de nos contraintes de temps (15 minutes par partie). Cet élagage élimine les chemins inutiles qui n'affectent pas la décision finale, rendant ainsi l'algorithme plus efficace. De plus, pour maximiser l'efficacité de l'algorithme Alpha-Beta, il est crucial de trier les actions pour explorer en priorité les plus prometteuses, augmentant ainsi les chances de couper rapidement les branches inutiles. En stockant les évaluations avant le tri, nous évitons les recalculs inutiles, optimisant ainsi les performances globales.

actions.sort(key=lambda action: self.evaluate_state(action.get_next_game_state()), reverse=maximizing_player).

Zobrist Hashing

Le Zobrist Hashing génère un identifiant unique pour chaque état du plateau de jeu. Il utilise des opérations binaires exclusives pour refléter rapidement les changements d'état sans recalculer entièrement le hash. Cette méthode est idéale car elle est rapide, légère en ressources et adaptée aux mises à jour incrémentales.

Table de Transposition

La table de transposition stocke les états de jeu déjà évalués, associant chaque hash à des informations soit le score calculé et la profondeur de recherche. Avant d'évaluer un état, l'algorithme vérifie si son hash est déjà présent dans cette table. Si c'est le cas, l'évaluation stockée est utilisée immédiatement, économisant ainsi un temps précieux. Après l'évaluation, les résultats sont enregistrés pour une réutilisation future.

if state_hash in self.transposition_table: return self.transposition_table[state_hash]

self.transposition_table[state_hash] = {'score': evaluated_score, 'depth': current_depth}

En réutilisant les évaluations des états déjà explorés, cette méthode réduit le nombre de branches à analyser dans l'arborescence de Minimax, permettant une exploration plus profonde et rapide. Cela aide l'agent à anticiper davantage les mouvements adverses tout en respectant les contraintes de temps. Le temps économisé grâce à cette optimisation est essentiel pour maximiser la profondeur d'analyse, améliorant ainsi la qualité des décisions prises. En complément de la table de transposition, plusieurs caches spécifiques sont mis en

place pour optimiser davantage les performances. Ces caches permettent de stocker les résultats des calculs fréquents ou critiques, évitant ainsi leur répétition et réduisant la charge de calcul global.

Contrainte de temps

Nous n'avons pas implémenté de système dédié à la gestion des 15 minutes allouées par partie. Au lieu de cela, nous avons choisi une approche basée sur la limitation de la profondeur de recherche, ajustée en fonction de la progression de la partie. En début de partie, l'agent est limité à une profondeur maximale de 2, permettant des calculs rapides dans une situation où les décisions stratégiques ont encore peu d'impact. Au milieu de la partie, cette profondeur est augmentée à 3 pour explorer un plus grand éventail de possibilités. Enfin, en fin de partie, lorsque les choix deviennent critiques pour l'issue du jeu, l'agent peut atteindre une profondeur maximale de 9. Ces profondeurs ont été choisies en grande partie par essais-erreurs, afin de trouver un équilibre optimal entre exploration et respect des contraintes de temps.

L'intégration de Numpy pour les calculs a joué un rôle clé dans cette optimisation. Grâce à ses capacités de traitement rapide et efficace des données, nous avons considérablement réduit le temps nécessaire pour effectuer les calculs complexes impliqués dans l'évaluation des états de jeu et la simulation des coups. Cette amélioration de performance a rendu inutile la gestion explicite d'un temps limite pour chaque coup individuel. En se limitant uniquement sur la profondeur de recherche, l'agent a pu effectuer un nombre important de calculs sans dépasser le temps total imparti de 15 minutes par partie.

Heuristique

La fonction *evaluate_state* analyse l'état du jeu en se concentrant sur deux aspects clés : les scores actuels et le potentiel de formation de divercités, soit une approche équilibrée entre gains immédiats et opportunités stratégiques futures. La différence entre les points de l'agent et ceux de l'adversaire offre une vision instantanée de la situation. Une valeur positive l'encourage à adopter une stratégie agressive pour maintenir son avantage. Une valeur négative, en revanche, incite l'agent à adopter une posture défensive pour réduire l'écart. De plus, l'agent évalue non seulement son potentiel de divercité, mais aussi celui de l'adversaire, pondérant fortement les configurations proches de marquer des points. Nous présenterons dans cette section les différentes fonctions qui implémentent notre stratégie, en détaillant comment elles priorisent les actions de l'agent en fonction des opportunités évaluées. Chaque stratégie est associée à un poids, reflétant son importance relative dans le processus de prise de décision. Ces priorités sont illustrées dans le tableau présenté plus bas, qui résume les valeurs clés et montre comment elles contribuent à maximiser les chances de succès.

Stratégie de potentiel de Divercité

La fonction *count_divercite_potential* analyse chaque cité pour déterminer son avancement vers une divercité complète. Cette évaluation permet de prioriser les actions qui maximisent rapidement les opportunités de score, garantissant que l'agent se concentre sur des configurations prometteuses et minimise les pertes de temps sur des opportunités peu réalistes.

Tableau 1 - Résumé des états de potentiel de divercité

Valeur Potentielle	Description	Priorité pour l'IA
0	Divercité déjà atteinte	Basse
1	1 couleur unique supplémentaire nécessaire	Haute
2	2 couleurs uniques supplémentaires nécessaires	Moyenne
3	3 couleurs uniques supplémentaires nécessaires	Basse
4	Aucune couleur unique ou toutes identiques	Très basse

Adaptation selon qui commence

L'agent ajuste sa stratégie selon qu'il commence ou joue en second, déterminé par le booléen *is_first_player*. Lorsque l'agent commence, il adopte une stratégie agressive en plaçant rapidement des ressources dans des positions clés autour des cités prioritaires pour sécuriser un avantage initial tout en réduisant les options adverses. Par exemple, il peut concentrer ses efforts sur des cités à haut potentiel de divercité, obligeant l'adversaire à réagir. En revanche, lorsque l'agent joue en second, il observe les premiers mouvements adverses pour contrer leurs plans tout en exploitant les opportunités laissées ouvertes. Cette approche défensive-réactive lui permet de maintenir un équilibre tout en perturbant les stratégies adverses.

Blocage de l'adversaire

La fonction *evaluate_blocking_potential* se concentre sur l'identification des cités adverses proches de compléter une divercité. Contrairement à d'autres fonctions, elle cible spécifiquement les actions adverses et cherche à les neutraliser. Par exemple, si une cité adverse nécessite une ressource unique pour achever une divercité, l'agent privilégiera ce placement pour bloquer cette opportunité. Cette stratégie défensive vise à limiter les gains adverses tout en maintenant un contrôle stratégique sur le plateau.

Placement stratégique des ressources

La fonction *evaluate_resource_placement_potential* identifie les emplacements optimaux pour maximiser les avantages de l'agent tout en créant des opportunités stratégiques à long terme. Contrairement au blocage pur, cette fonction favorise les positions qui permettent à l'agent de progresser vers des divercités ou de consolider des configurations utiles. Par exemple, dans une zone densément peuplée, l'agent cherchera un placement qui optimise ses propres chances tout en minimisant les opportunités adverses. Cette stratégie assure un équilibre entre la progression de l'agent et la limitation des options adverses.

Stratégie de fin de partie

La fonction *evaluate_final_move_control* intervient exclusivement en phase finale, lorsque les derniers emplacements vides influencent fortement l'issue du jeu. À ce stade, si l'emplacement final complète une divercité adverse, l'agent priorise son occupation pour bloquer cet avantage. En revanche, si cet emplacement offre à l'agent une opportunité significative, comme compléter une divercité ou renforcer sa position défensive, il le choisira stratégiquement. Cette gestion des moments critiques permet à l'agent de maximiser son impact sur l'issue de la partie.

Stratégie de vision à long terme

Enfin, la fonction evaluate_winning_patterns valorise les configurations favorables à l'agent, telles que les divercités complètes ou les cités sécurisées. Elle récompense les positions qui garantissent un avantage durable, comme les cités déjà entourées de quatre ressources uniques ou celles complètement sécurisées, empêchant toute intervention adverse.

Résultats et évolution de l'agent

Nous avons développé notre propre plateforme *Abyss Clone*, basée sur le code de la véritable plateforme *Abyss*. Cette initiative visait à surmonter les limitations imposées par la plateforme officielle, notamment les restrictions de fréquence des tests. *Abyss Clone* nous a permis d'exécuter des tests automatisés de manière fréquente et efficace, éliminant ainsi le besoin de tests manuels en local, moins productifs. Sur cette plateforme, chaque version d'agent a été testée un nombre égal de fois afin d'obtenir des résultats comparables. Cependant, certaines versions d'agents ont rencontré des dysfonctionnements, entraînant des disqualifications et des scores inégaux malgré un nombre similaire de parties. Ces tests nous ont aidés à identifier et corriger les faiblesses spécifiques de chaque agent.

Nous avons développé trois agents principaux, chacun ayant une orientation stratégique distincte. *Syme*, notre agent axé sur la performance, a bénéficié d'optimisations constantes de l'algorithme Minimax. En y intégrant des techniques avancées comme Alpha-Beta Pruning, Zobrist, et des améliorations dans la gestion du temps de calcul, *Syme* est devenu de plus en plus compétitif dans des environnements de jeu complexes. En parallèle, nous avons conçu *Julia*, un agent basé sur des heuristiques. *Julia* servait principalement d'agent expérimental pour tester des stratégies nouvelles et créatives, en fonction des règles du jeu. Cet agent se distinguait par son exploration de concepts non optimisés algorithmiquement, ce qui a permis d'enrichir notre compréhension des dynamiques stratégiques.

Le tableau suivant montre les résultats obtenus de nos tests dans *Abysse Clone*. À noter que le dernier chiffre du nom de l'agent représente s'il commençait la partie (1) ou pas (2).

Tableau 2 - Performance des agents *Syme* et *Julia* par version

Identifiant (version)	Victoires	Défaites
SymeV0_1	0	17
SymeV0_2	16	0
SymeV1_1	0	16
SymeV1_2	17	0
SymeV3_1	0	16
SymeV3_2	0	16
SymeV4_1	16	16
SymeV4_2	16	16
SymeV5_1	16	16
SymeV5_2	16	16
SymeV6_1	16	16
SymeV6_2	16	16
SymeV7_1	27	5
SymeV7_2	27	5
JuliaV2_1	5	11
JuliaV2_2	5	11

À chaque nouvelle version, *Syme* a bénéficié d'améliorations ciblées sur son algorithme de calcul. Ces évolutions ont conduit à une progression constante des performances, visible dans les résultats obtenus. À partir des versions V6+, nous avons fusionné *Julia* dans *Syme*. Cette intégration a permis de créer un modèle hybride combinant les capacités d'optimisation et de prise de décision rapide de *Syme* avec l'heuristique avancée de *Julia*.

Enfin, notre troisième agent, *O'Brien*, était dédié à l'apprentissage automatique. L'objectif était de lui permettre de jouer contre lui-même et ainsi apprendre à jouer mieux pour adapter sa stratégie en fonction de l'expérience. Il a donc bâti ses propres connaissances en jouant contre lui-même sur 100 000 parties. *O'Brien* a initialement affiché des performances médiocres, perdant fréquemment contre des agents simples tels que *Random* ou *Greedy*. Cette sous-performance s'explique par un manque de données initiales, car *O'Brien* ne disposait d'aucune règle de base pour guider ses premières décisions, et des besoins élevés en entraînement, incompatibles avec les contraintes de notre environnement. Pour surmonter ces obstacles, nous avons réorienté *O'Brien* en l'intégrant à *Syme* et *Julia*. Nous avons exploité le concept des poids, un concept central dans l'apprentissage automatique, en les considérant comme des paramètres d'entraînement ajustables. Grâce à *Abyss Clone*, nous avons testé différentes combinaisons de poids stratégiques générées par l'IA (*ChatGPT*), permettant à *O'Brien* de jouer plusieurs versions avec des ajustements progressifs. Les résultats ont montré que la version 0 offrait les meilleurs résultats. Cette stratégie, caractérisée par une concentration agressive sur le score, a été adoptée pour *O'Brien_V1* sur la plateforme *Abyss*, où il a obtenu un excellent Elo, démontrant sa compétitivité face à des agents avancés.

Tableau 3 - Performance des poids de *O'Brien* par version

Version	Poids	Stratégie	V	D	Ratio
0	Premier joueur : [4.0, 1.5, 0.5, 1.0, 1.0, 1.0] Autre cas : [2.5, 1.0, 0.5, 0.8, 1.0, 1.0]	Concentration agressive sur le score avec une attention minimale aux autres caractéristiques.	74	4	18.50
1	Premier joueur : [2.0, 2.0, 1.0, 1.0, 1.5, 1.5] Autre cas : [1.5, 1.5, 1.0, 1.0, 1.2, 1.2]	Importance équilibrée entre toutes les caractéristiques. Aucune caractéristique n'est fortement priorisée.	60	19	3.16
2	Premier joueur : [1.5, 1.5, 0.8, 1.0, 2.5, 2.5] Autre cas : [1.0, 1.2, 0.8, 1.0, 2.0, 2.0]	Concentration sur le contrôle du dernier coup avec les schémas gagnants comme priorité secondaire.	58	20	2.90
3	Premier joueur : [3.0, 1.5, 0.8, 1.0, 1.2, 1.5] Autre cas : [2.5, 1.2, 0.6, 1.0, 1.0, 1.2]	Stratégie offensive simple sans accent particulier sur le contrôle ou la défense.	37	13	2.85
4	Premier joueur : [2.5, 2.0, 1.0, 1.2, 1.8, 2.2] Autre cas : [1.8, 1.6, 0.8, 1.0, 1.5, 2.0]	Approche équilibrée avec une légère inclinaison offensive vers le score et les schémas gagnants.	34	16	2.13
5	Premier joueur : [1.2, 1.0, 0.5, 1.0, 1.0, 3.0] Autre cas : [1.0, 1.0, 0.5, 1.0, 0.8, 2.5]	Les schémas gagnants dominent l'évaluation, avec un faible poids pour les autres caractéristiques.	33	17	1.94
6	Premier joueur : [3.5, 1.0, 0.8, 2.5, 1.2, 1.0] Autre cas : [3.0, 0.8, 0.6, 2.0, 1.0, 0.8]	Accent important sur le score et le placement pour dominer le jeu positionnel.	31	19	1.63
7	Premier joueur : [3.0, 1.2, 1.0, 1.5, 2.5, 1.0] Autre cas : [2.5, 1.0, 0.8, 1.2, 2.0, 1.0]	Score agressif et reconnaissance des schémas tout en maintenant un placement modéré.	28	22	1.27
8	Premier joueur :	Met l'accent sur le placement des ressources et le contrôle	22	28	0.79

	[1.8, 1.5, 0.7, 2.5, 2.0, 1.5] Autre cas : [1.2, 1.2, 0.6, 2.0, 1.8, 1.2]	du dernier coup pour des avantages à long terme.			
9	Premier joueur : Random [0.5, 3.0] Autre cas : Random [0.5, 2.5]	Pondération exploratoire / aléatoire pour l'expérimentation.	22	28	0.79
10	Premier joueur : Random [1.0, 3.0] Autre cas : Random [0.8, 2.5]	Mode exploration avec des limites aléatoires élargies pour l'expérimentation.	22	28	0.79
11	Premier joueur : [2.0, 2.0, 2.0, 2.0, 2.0, 2.0] Autre cas : [1.5, 1.5, 1.5, 1.5, 1.5, 1.5]	Pondérations uniformes amplifiées pour une évaluation globale agressive.	19	31	0.61
12	Premier joueur : [1.0, 1.0, 1.0, 1.0, 1.0, 1.0] Autre cas : [0.8, 0.8, 0.8, 0.8, 0.8, 0.8]	Pondérations uniformes ; chaque caractéristique contribue de manière égale au score.	19	31	0.61
13	Premier joueur : [1.5, 2.8, 2.2, 1.0, 1.0, 2.5] Autre cas : [1.0, 2.5, 2.0, 1.0, 1.0, 2.0]	Axé sur la défense avec une forte attention à la diversité et à la reconnaissance des schémas.	17	33	0.52
14	Premier joueur : [1.0, 2.0, 0.8, 1.0, 1.2, 3.0] Autre cas : [0.8, 1.8, 0.6, 0.8, 1.0, 2.5]	Combine diversités et schémas gagnants, équilibrant stratégies à long et court terme.	14	36	0.39
15	Premier joueur : [1.0, 3.0, 2.0, 1.0, 0.8, 1.2] Autre cas : [1.0, 2.5, 1.8, 1.0, 0.8, 1.0]	Stratégie défensive avec un fort accent sur la diversité et le blocage.	18	60	0.30
16	Premier joueur : [1.0, 1.0, 0.6, 3.0, 1.2, 1.0] Autre cas : [1.0, 1.0, 0.6, 2.5, 1.0, 1.0]	Priorise le placement des ressources pour gagner un potentiel avantage.	9	39	0.23
17	Premier joueur : [1.0, 3.0, 1.0, 2.5, 1.0, 1.0] Autre cas : [0.8, 2.5, 0.8, 2.0, 1.0, 1.0]	La diversité et le placement dominent, visant à surpasser stratégiquement l'adversaire.	8	42	0.19
18	Premier joueur : [1.0, 1.5, 2.8, 3.0, 1.2, 1.0] Autre cas : [0.8, 1.2, 2.5, 2.5, 1.0, 0.8]	Blocage élevé et coordination des ressources pour contrôler le plateau.	3	47	0.06
19	Premier joueur : [1.0, 1.5, 3.0, 1.2, 2.5, 1.0] Autre cas : [0.8, 1.2, 2.5, 1.0, 2.0, 1.0]	Priorise le blocage des coups adverses tout en maintenant un contrôle modéré du dernier coup.	2	48	0.04

Tableau 4 - Performance de l'agent final sur *Abyss*

Identifiant (version)	**Victoires**	**Défaites**	**Peak Elo**
O'brien_V1	13	5	1314

Discussion

Notre agent final présente plusieurs avantages majeurs qui en font un compétiteur efficace. L'implémentation de Alpha-Beta Pruning optimise l'algorithme Minimax en évitant les calculs inutiles en ne visitant pas les branches de l'arbre de décision qui n'affecteront pas le résultat final. De plus, en limitant la profondeur de recherche via la variable self.max_depth, l'agent se concentre sur un nombre stratégique de coups à l'avance, équilibrant entre profondeur et pertinence des calculs. Un autre atout est l'intégration de la mémoire via le hachage de Zobrist et une table de transposition. Couplées à nos optimisations de calcul (comme Numpy), ces améliorations ont permis à notre stratégie d'aller plus loin dans l'arbre de décision, augmentant ainsi la profondeur et la qualité des analyses.

Un des points forts réside dans notre analyse approfondie et notre gestion des poids stratégiques. L'agent priorise les actions grâce à un tri optimisé n'étendant en priorité que les actions les plus prometteuses. Cette approche réduit le facteur de branchement et permet de focaliser les calculs sur les options offrant le meilleur potentiel. Notre méthodologie, basée sur des tests exhaustifs des poids attribués à chaque stratégie, nous a permis de tirer parti d'une grande variété d'heuristiques tout en les optimisant de manière méthodique. Si l'idée de base était prometteuse, l'optimisation de ces stratégies a constitué la véritable clé du succès, permettant à l'agent d'exploiter pleinement son potentiel.

Cependant, l'agent final présente encore certaines limites. En reposant entièrement sur l'algorithme Minimax, il ne bénéficie pas des avantages de méthodes plus avancées, comme la recherche Monte Carlo Tree Search, qui pourrait offrir une approche plus adaptative et stochastique. De plus, des éléments d'heuristique comme la gestion des non-correspondances de couleurs, ne sont pas pleinement prises en compte.

Pistes d'amélioration

Comme mentionné précédemment, nous pourrions intégrer une stratégie de recherche comme la recherche Monte Carlo Tree Search. Cette méthode permettrait à l'agent d'explorer de manière aléatoire certaines branches de l'arbre de décision. En testant des actions qui pourraient être ignorées par l'élagage Alpha-Beta, l'agent pourrait découvrir des options gagnantes inattendues. De plus, la gestion des non-correspondances de couleurs pourrait être améliorée. Actuellement, l'agent ne prend pas en compte les opportunités de contrer efficacement l'adversaire en empêchant certaines configurations de couleurs.

Une autre piste majeure d'amélioration réside dans l'optimisation des poids stratégiques. Bien que nous ayons déjà testé différentes combinaisons de poids pour équilibrer les priorités des stratégies, notre analyse s'est limitée à une vingtaine de configurations. Une exploration plus poussée, avec un plus grand nombre de combinaisons, pourrait permettre de découvrir des poids encore plus optimaux, renforçant ainsi les performances globales de l'agent.

Enfin, une amélioration significative passerait par l'intégration d'un système d'apprentissage automatique. Comme nous avions initialement envisagé avec *O'Brien*, un modèle capable d'ajuster ses décisions en fonction des stratégies adverses rencontrées offrirait une adaptabilité inégalée. Avec davantage de temps et un accès étendu aux parties jouées sur *Abyss*, le modèle pourrait s'entraîner de manière plus approfondie. En apprenant des résultats de chaque partie et en conditionnant ses choix à maximiser ses chances de victoire, l'agent deviendrait encore plus compétitif face à des joueurs aux styles variés.