INF8175 – Intelligence artificielle : méthodes et algorithmes

Automne 2024

Rapport - Projet final

Groupe 1

2145565 – Sidney Gharib

2141302 – Lammali Mounir

Soumis à : M. Quentin Cappart

7 décembre 2024

Table des matières

2. Nom d'équipe

Nous avons décidé de ne pas participer au tournoi Challonge pour des raisons qui seront détaillées dans ce rapport. Sur Abyss, notre nom est Thon de l'Atlantide et l'équipe est composée de Mounir Lammali (2141302) et de Sidney Gharib (2145565).

3. Méthodologie

Notre agent a subi plusieurs itérations au cours du semestre. Finalement, notre agent utilise l'algorithme Minimax, en implémentant de l'élagage alpha-bêta et une table de transposition.

Au début du projet, nous avions choisi d'utiliser l'algorithme Minimax avec de l'élagage alpha-bêta. Par la suite, nous nous sommes concentrés sur l'implémentation d'heuristiques avancées. Par exemple, nous avons développé plusieurs fonctions pour identifier l'ensemble des Divercités possibles pour un état donné du jeu, d'autres pour compléter ces Divercités, ou encore pour les bloquer. Cependant, une semaine avant la compétition Challonge, nous avons découvert que nous avions mal compris la manière d'intégrer une heuristique dans un algorithme. Concrètement, au lieu de permettre à Minimax de déterminer lui-même les meilleures actions et de leur attribuer un score élevé en fonction de leur pertinence, nous contraignions l'algorithme à exécuter des actions spécifiques via nos fonctions heuristiques. Cela allait à l'encontre du principe de Minimax, qui consiste justement à explorer les actions possibles et à évaluer leur impact pour maximiser les points. Cette prise de conscience nous a obligés à repenser entièrement notre approche et à reconstruire notre algorithme sur des bases solides, ce qui a été très chronophage. Par conséquent, notre agent n'était pas prêt à performer lors de la compétition Challonge, ce qui nous a amenés à prendre la décision de ne pas y participer.

La profondeur maximale de l'algorithme Minimax est ajustée en fonction du tour auquel la partie se trouve. Ainsi, plus la partie progresse, plus nous pouvons augmenter la profondeur de recherche. Cependant, si le temps commence à manquer, nous réduisons la profondeur maximale pour garantir que l'algorithme puisse toujours retourner une action dans les délais impartis.

Pour nos heuristiques, nous avons commencé avec une version améliorée de l'approche gloutonne. Celle-ci consiste à maximiser la différence entre nos points et ceux de l'adversaire, afin de garantir que nos actions nous avantagent davantage qu'elles n'aident l'adversaire. Nous avons également implémenté une fonction permettant de détecter les ressources uniques disponibles. Cela nous incite à conserver une ressource de chaque type au lieu de jouer toutes les ressources d'une seule couleur. Cette stratégie favorise la création de Divercités tout en nous assurant de pouvoir bloquer celles de l'adversaire si nécessaire. Par ailleurs, nous avons développé une heuristique qui évalue le nombre de Divercités bloquées. En effet, certains mouvements, en plus de nous rapporter des points, réduisent le score potentiel de l'adversaire en empêchant la création de ses Divercités. De plus, nous avons mis en place une heuristique qui priorise la création de Divercités utilisant des ressources dont nous disposons en abondance. Cela garantit que ces actions n'épuisent pas notre stock d'une ressource spécifique, nous permettant ainsi de maintenir une réserve stratégique pour les tours suivants.

Pour la table de transposition, nous devions ajouter un mécanisme visant à limiter la taille de celle-ci, afin de respecter la limite de mémoire imposée sur l'agent (4Gb de RAM). En effet, en utilisant l'approche d'une table de transposition, l'agent doit calculer un hash pour chaque état de jeu exploré et la stocker dans la table de transposition. Considérant le nombre très important d'états explorés et stockés à mesure que la partie avance, il est possible de dépasser les 4Gb de RAM, ce qui causerait à notre agent d'être désactivé. Pour remédier à ce problème, la table de transposition possède une classe qui hérite d'un dictionnaire ordonné, afin que l'on puisse conserver l'ordre d'insertion des éléments dans celle–ci. Ainsi, il est possible de retirer les éléments les plus anciens lorsque l'on atteint la capacité maximale de mémoire allouée à la table de transposition. Dans cette classe, nous utilisons

aussi le principe « premier arrivé, premier sorti » (file) pour retirer des états lorsque la mémoire est pleine, puisque dans le jeu Divercité, chaque joueur ne peut qu'ajouter des pièces à la planche; il est impossible de retirer ou déplacer des pièces déjà mises. Donc, lorsque la partie avance et que plusieurs pièces ont été placées, il est inutile de conserver des états du jeu qui sont incompatibles avec l'état actuel de la planche. Le dictionnaire possède également une taille maximale qui correspond à la variable « TABLE_MAX_SIZE », et dont la taille a été déterminée selon des tests et calculs basés sur la manière dont chaque état est encodé.

4. Résultats et évolution de l'agent

Dans cette section, les performances de notre agent seront évaluées à l'aide de plusieurs critères. Le pourcentage de victoires sur la plateforme Abyss permettra de mesurer son efficacité contre une variété d'adversaires externes, tandis que le pourcentage de victoires entre nos agents locaux servira à comparer les différentes versions de nos agents et à identifier les stratégies les plus robustes. Nous prendrons également en compte le temps moyen pris par tour, un indicateur pour évaluer l'efficacité temporelle de notre agent le plus performant, ainsi que la profondeur maximale de recherche atteinte durant une partie, qui reflète la capacité de nos agents à réfléchir plusieurs coups à l'avance.

Nous allons d'abord décrire certaines versions de notre agent qui étaient « hardcodées », une approche que nous avons adoptée au début du projet. Par exemple, nous avions une fonction permettant d'identifier toutes les Divercités possibles nécessitant une seule ressource supplémentaire pour être complétées. Une autre fonction était dédiée à la réalisation de ces Divercités. À l'inverse, nous avions également une fonction qui bloquait une Divercité adverse lorsque celle-ci était à une ressource près d'être complétée. Une version spécifique de notre agent comportait une table d'ouverture qui plaçait systématiquement toutes les villes avant de jouer une ressource. Ces villes étaient positionnées de manière à regrouper les couleurs similaires, maximisant ainsi le potentiel des ressources de même couleur une fois placées. Par ailleurs, les villes étaient prioritairement placées au centre du plateau, car nous présumions que l'essentiel des interactions se concentrerait au centre plutôt qu'aux extrémités du plateau. Cette approche visait à optimiser les actions futures et à mieux contrôler la partie dès son début.

Les versions suivantes de notre agent ont considérablement évolué, car, comme mentionné précédemment, nous avons commencé à implémenter correctement les heuristiques. C'est à ce moment-là, sur Abyss, que nous avons remporté notre première victoire. Bien que nos agents actuels soient encore loin d'être parfaits, ils surpassent largement ceux qui se reposent presque exclusivement sur des tables d'ouverture. Cette transition marque un tournant dans notre approche, avec une amélioration notable des performances globales.

Table 1: Pourcentage de victoires des agents sur la plateforme Abyss

Agent	Pourcentage de victoire sur Abyss
NoNameAgent (MinMax)	0%
GoMJGoMJ (City Placer)	0%
Improved (Heuristic + transposition)	30%
Improved_v2 (Heuristic_v2 + transposition)	45%

Dans la table 1, le pourcentage de victoire est affiché et en parenthèse il s'agit de la stratégie utilisée et priorisée par chaque agent.

Table 2: Résultats de parties entre nos agents (algorithmes) locaux

Premier Agent/ Deuxième Agent	MinMax	Hardcoded	City Placer	Heuristic_v2 + transposition	Heuristic + transposition	Divercity
MinMax	Égalité	Victore MinMax	Victoire city placer	Heuristic_v2 + transposition	Égalité	Victoire Divercity
Hardcoded	Victore MinMax	Égalité	Victoire city placer	Heuristic_v2 + transposition	Égalité	Victoire Divercity
City Placer	Victoire city placer	Victoire city placer	Égalité	Heuristic_v2 + transposition	Victoire city placer	Victoire city placer
Heuristic_v2 + transposition	Heuristic_v2 + transposition	Heuristic_v2 + transposition	Heuristic_v2 + transposition	Heuristic_v2 + transposition	Heuristic_v2 + transposition	Heuristic_v2 + transposition
Heuristic + transposition	Victore MinMax	Égalité	Victoire city placer	Heuristic_v2 + transposition	Égalité	Victoire Divercity
Divercity	Victoire Divercity	Victoire Divercity	Victoire city placer	Heuristic_v2 + transposition	Victoire Divercity	Égalité

Note importante: Dans la table précédente nous avons décidé d'afficher la stratégie utilisée plutôt que le nom de l'agent pour que la comparaison soit plus facile.

Dans la Table 2, on observe que l'agent « City Placer » a remporté presque toutes ses parties. Ce qui est particulièrement intéressant, c'est que cet algorithme ne performe pas très bien sur Abyss, ce qui indique que sa stratégie est efficace principalement contre nos propres agents. Autrement dit, cet agent généralise mal face à une population variée d'adversaires, mais il excelle en tant que spécialiste contre nos stratégies spécifiques. Par ailleurs, l'agent spécialisé dans les Divercités s'avère également très performant. Il a remporté toutes ses parties, sauf contre « City Placer ». Bien que cet agent ait été conçu au début du projet, il continue de bien performer localement, démontrant ainsi la robustesse de sa stratégie face à une majorité d'opposants. Finalement le meilleur agent est « my player » qui est très similaire à l'agent « Heuristic + transposition », mais qui a des poids différents sur chaque heuristique.

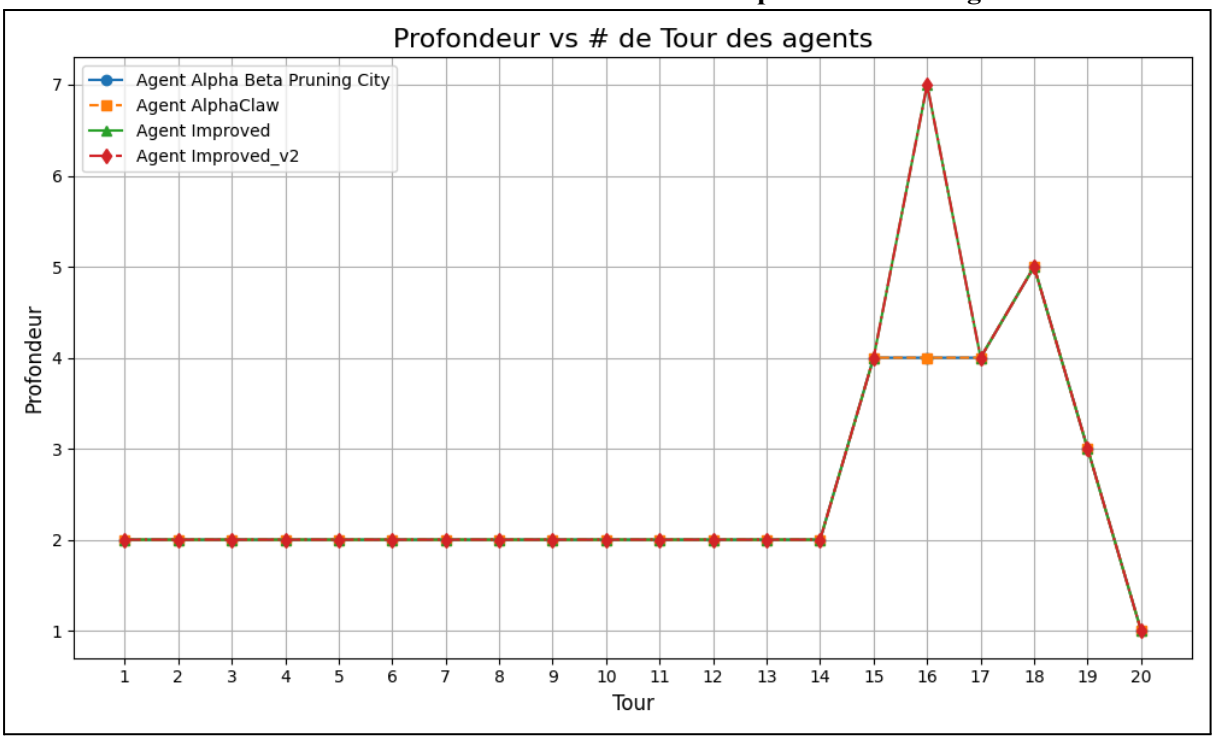
Table 3: Temps moyen pris par tour par notre meilleur agent selon l'utilisation d'une table de transposition

	Temps moyen par tour (en secondes)
Sans table de transposition	10.40
Avec une table de transposition	15.86

Nous avons calculé le temps moyen par tour pour notre agent le plus performant, avec et sans l'utilisation de la table de transposition, sur un échantillon de 10 parties. À notre grande surprise, le temps moyen sans table de transposition s'est révélé inférieur à celui avec une table de transposition. Nous nous attendions à l'inverse, car la table de transposition permet d'éviter de recalculer les états

déjà explorés, et devrait donc réduire le temps moyen par tour. Les facteurs susceptibles d'expliquer ce résultat inattendu seront détaillés dans la prochaine section.

Table 4: Profondeur de recherche selon le tour pour différents agents



Nous avons constaté qu'avec l'amélioration de la logique de notre algorithme Minimax et de la gestion de la profondeur de recherche, nos agents les plus performants sont désormais capables d'atteindre une profondeur maximale de 7. Cela représente une amélioration significative par rapport à nos agents précédents, qui étaient limités à une profondeur maximale de 4. Cette progression permet à nos agents d'explorer davantage de scénarios potentiels, renforçant ainsi leur capacité à prendre des décisions stratégiques optimales.

5. Discussion

D'après nos analyses présentées dans la section précédente, nous avons identifié les avantages et les limites de notre agent final, ainsi que plusieurs pistes d'amélioration potentielles.

Dans le cas de la gestion de mémoire pour notre table de transposition, il serait judicieux d'ajouter un mécanisme pour retirer tous les états incompatibles avec l'état actuel de la planche. Comme mentionné plus tôt, un joueur ne peut qu'ajouter des pièces à la planche, donc pour économiser de la mémoire, il serait possible de rajouter une routine qui supprimerait tout état qui ne coïnciderait pas avec l'état actuel de la planche. En ce moment, notre table de transposition ne retire des états que lorsque la mémoire devient pleine, en retirant que les états les plus anciens. Ceci est valide, mais il y a place à amélioration comme mentionné précédemment. Une autre amélioration possible pour la table de transposition consisterait à améliorer la technique de hashing utilisée pour encoder l'état du jeu. Notamment, il serait intéressant de plutôt utiliser le hashing de Zobrist. En effet, grâce au hashing de Zobrist, l'agent serait en mesure de calculer le hash d'un état de la planche beaucoup plus rapidement en effectuant une mise à jour du hash actuel grâce à une opération XOR. Sans compter que chaque état occuperait un moins grand espace en mémoire (soit un entier en 64 bits par état), comparativement à la méthode de hashing utilisée actuellement, qui encode l'état actuel sous un string, et où le string grandit à mesure que des pièces sont ajoutées sur la planche.

Pour la gestion du temps, nous avons une limite de recherche en profondeur dans l'arbre d'actions possibles selon à quel tour nous sommes rendus dans la partie. Par exemple, au premier tour nous limitons la profondeur à 2 alors que lorsqu'il reste que 10 tours, nous allons à une profondeur maximale de 7. Dans le cas advenant où il reste peu de temps, nous allons retourner à une profondeur de 2, car nous ne voulons pas perdre par disqualification. Une amélioration possible pour la gestion du temps serait d'allouer dynamiquement le temps selon la partie jouée. En effet, actuellement l'agent n'exploite pas optimalement les 15 minutes allouées par partie. Pour améliorer cela, nous pouvons répartir le temps alloué plus efficacement selon un moment précis durant la partie. Par exemple, les premiers tours se verraient attribuer un temps moindre étant donné qu'ils sont moins critiques que des tours de milieu ou fin de partie. De plus, si un tour prend moins de temps que prévu, le temps économisé peut être ajouté à un tour plus critique. Aussi, durant les appels à l'algorithme Minimax, si l'on remarque que le temps d'exploration des nœuds prendrait trop de temps, nous pouvons ajuster à la baisse la profondeur de recherche de l'algorithme. Avec cette amélioration, nous aurions sans doute obtenu des résultats différents quant à la performance de notre agent lorsqu'il utilise une table de transposition, car puisque le temps n'est pas utilisé de manière optimale, cela limite la profondeur de recherche et réduit les opportunités d'exploiter pleinement la table de transposition. Cela empêche cette dernière de jouer son rôle de manière efficace, puisqu'elle n'est pas suffisamment sollicitée dans les recherches.

Nous avons décidé de ne pas utiliser l'algorithme Monte Carlo Tree Search (MCTS). Bien que MCTS puisse être très performant dans de nombreux contextes, nous avons appris durant le cours qu'il peut parfois commettre de petites erreurs. La gravité de ces erreurs dépend du jeu. Par exemple, dans un jeu comme Starcraft, ne pas effectuer le mouvement de troupes le plus optimal peut être toléré sans impact majeur sur la partie. Cependant, dans Divercité, une seule erreur, comme le fait de ne pas bloquer une Divercité critique, peut suffire à perdre une partie. Cela rend MCTS moins adapté à notre problématique, où chaque action doit être soigneusement calculée pour éviter des erreurs coûteuses.

Pour un futur agent, il aurait été intéressant d'adopter une toute nouvelle méthodologie, comme l'apprentissage par renforcement, une méthode particulièrement puissante pour exceller dans les jeux de société ou les jeux vidéo. Malheureusement, ce concept était optionnel dans le cadre du cours, et les ressources pour l'explorer ont été mises à disposition tardivement dans la session. Toutefois, si nous avions l'opportunité de refaire ce projet avec des connaissances supplémentaires en apprentissage par renforcement, ce serait une approche que nous envisagerions sérieusement. Cette méthode aurait pu permettre à notre agent d'apprendre des stratégies complexes et d'évoluer de manière autonome face à des adversaires variés, renforçant ainsi sa capacité à généraliser et à performer dans divers contextes.