Branly Stéphane (Matricule 2232279)
Guichard Amaury (Matricule 2227083)
Equipe DarWinMax

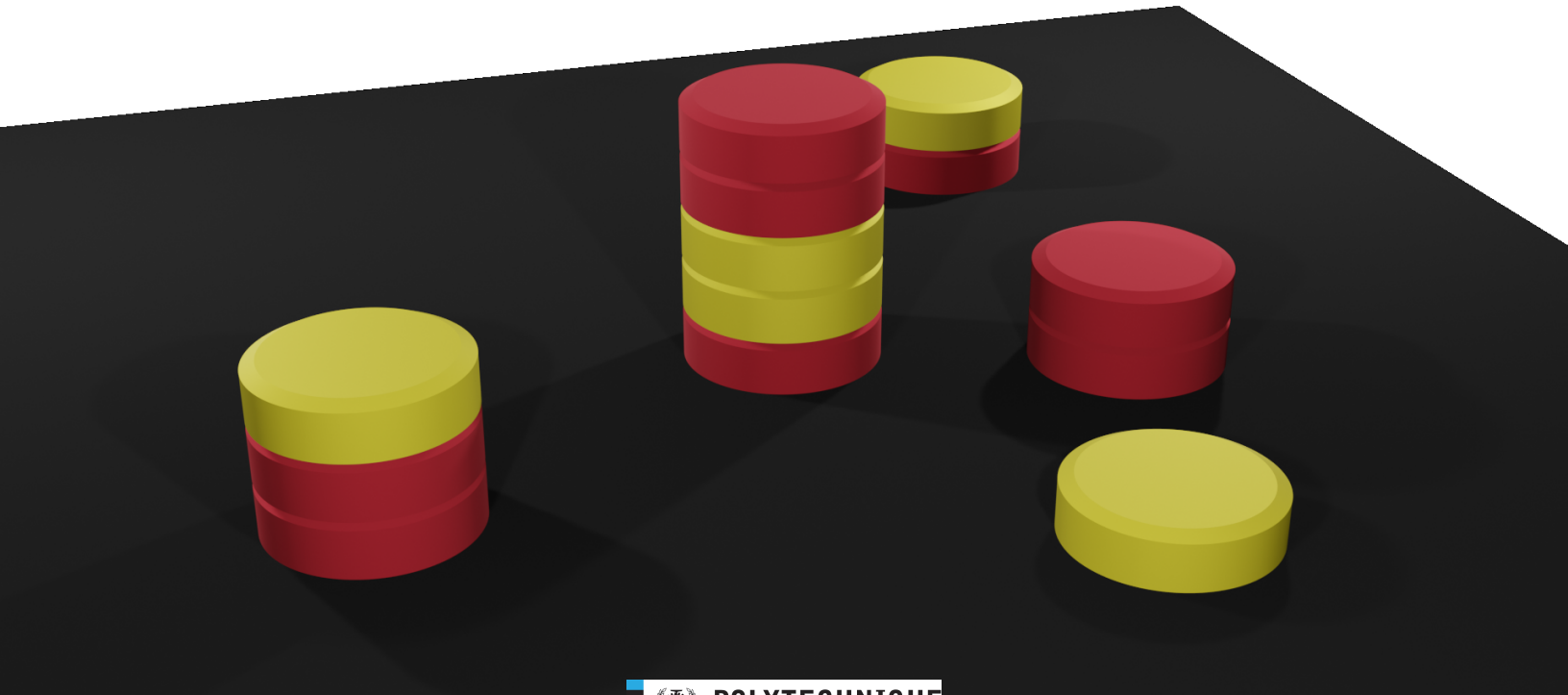
Rapport de projet

Agent intelligent pour le jeu Avalam

Montréal, 8 décembre 2022

Résumé

Dans le cadre de l'UV INF8215 (Intelligence artificielle : méthodes et algorithmes), un projet prend place dans le but d'appliquer les notions vues en cours sur un jeu concret. Le jeu sélectionné est le jeu de plateau Avalam qui se joue en 1 contre 1. La finalité du projet est un agent intelligent capable de jouer à ce jeu. Ce document montre le cheminement intellectuel que nous avons mené pour développer notre intelligence artificielle. L'ensemble du code est disponible sur le repository Github Avalam-AI. Durant ce projet, nous avions un objectif précis. Vérifier si un algorithme génétique permettait de s'affranchir des connaissances expertes approfondies dans la création d'une heuristique. Nous sommes parvenus en 8^e de finale avec une faible puissance de calcul et considérons donc notre objectif rempli.



POLYTECHNIQUE MONTRÉAL

0 | Introduction

0.1 | Le jeu Avalam

Le jeu Avalam est un jeu de plateau qui se joue 1 joueur contre 1 autre joueur. Chaque joueur est représenté par des pions lui appartenant (symbolisés par une couleur par exemple). Les actions possibles consistent à assembler 2 tours (constituées de pions) adjacentes pour en former une seule dont la hauteur ne peut exéder 5 pions. La partie se termine lorsqu'il n'y a plus d'actions possibles.
Le score est déterminé par la différence du nombre de tours possédées (dont le pion le plus haut appartient au joueur) par chaque joueur, en cas d'égalité, le joueur ayant le plus de tours de hauteur 5 gagne. Voir l'annexe A pour mieux visualiser le jeu.

0.2 | L'environnement de développement

Nous avons fait le choix de développer notre agent en utilisant git et en stockant notre code sur github. Cela permet de correctement versionner, examiner et déployer du code tout en ajoutant des actions qui génèrent de la documentation.

Des statistiques sur nos agents ainsi que sur le déroulement des parties permettent de mieux comprendre le comportement des agents ainsi que les évolutions des parties. Nous avons ainsi pu enregistrer les données suivantes :

— Nombre de coups disponibles par étape de la partie
— Évolution du plateau et du score durant la partie (avec génération de `.gif` comme nous pouvons le voir en annexe B.1)
— Résultats des parties

0.3 | La méthodologie de développement de l'agent

Pour développer notre agent, nous avons décider de le décomposer en trois parties

1. La stratégie de recherche
2. L'heuristique
3. L'importance des étapes / temps de calcul

Nous avons travaillé couche par couche sur notre agent pour l'enrichir. Le développement en utilisant la programmation orientée objet (diagramme de classes C) nous a permis de constituer les différentes briques qui peuvent être assemblées pour constituer un agent mais aussi reconstituer d'anciennes versions d'agents (à noter que les agents remis ont été "minimisés" et toute une partie Orientée Objet utilisée pour le développement a été retirée).

Tout d'abord la priorité de réflexion et développement a été mise sur les heuristiques et stratégies de recherche. Ainsi nous avons commencé par développer des heuristiques basiques s'interessant au plateau à une évolution d'un seul coup, puis deux coups. Heuristiques couplées dans un premier temps à une stratégie prenant le meilleur coup puis à une stratégie AlphaBeta. Notre but initial était de trouver cette heuristique via un algorithme génétique. L'idée était la suivante, nous développions des "fonctions d'observation" qui rendent compte d'une particularité du plateau. par exemple une fonction `enemy_isolated_tower3()` retournerait le nombre de tours de hauteur 3 isolées de l'enemi. L'heuristique serait alors une combinaison linéaire de ces fonctions d'observation : $Heuristic(x) = \sum_i \alpha_i f_i(x)$ où $f_i(x)$ est une fonction d'observation et α_i est son paramètre trouvé par génétique.

Ensuite nous nous sommes penchés sur l'implémentation d'une stratégie MonteCarlo Tree Search. Cette dernière ainsi qu'AlphaBeta nous ont petit à petit menés vers une réflexion sur la détermination de l'importance des étapes du jeu. L'importance des étapes étant cruciale pour savoir à quel moment il est judicieux de consacrer plus de temps de calculs.

Lors de la conception des briques, nous avons noté certaines optimisations pouvant étre apportées sur la représentation du plateau de jeu. Ainsi nous avons pu modifier le calcul du score du plateau, l'accès aux actions disponibles mais aussi remplacer des *clone*() de plateau par une représentation du plateau contenant l'historique des actions effectuées. Cela permet une exploration des arbres plus rapides comme le montre le tableau 0.1 avec une exploration naive de tous les états sur une certaine profondeur depuis l'état initial. Le coeur de ces optimisations réside sur le fait que le plateau n'est changé (score, tours, actions disponibles, fonctions d'observation) que localement par rapport à une action effectuée.

depth	états visités	*Board*	*ImprovedBoard*
2	81388	1.414s	0.503s
3	21711440	374.645s	127.296s

Table 0.1 – Temps d'exploration d'une partie de l'arbre en fonction du type de plateau

Nous allons maintenant voir dans les prochaines sections les trois parties qui composent notre agent.

1 | Stratégies de recherche

1.1 | Meilleur coup

Le plus simple des agents est le premier que nous avons implémenté (algorithme en annexe D), il parcourt les actions qu'il peut faire et effectue la meilleure selon son heuristique. Bien qu'en théorie limité, il nous a été très utile pour entrainer nos heuristiques avec un algorithme génétique.

1.2 | AlphaBeta

Notre algorithme minimax/alphabeta suit l'implémentation du cours (algorithme en annexe E) avec un certain nombre d'optimisations :

- les actions sont triées suivant l'heurisitique pour optimiser le pruning,
- les états évalués sont sauvegardés dans une hash map, ce qui permet de réduire le nombre d'états à visiter,
- le plateau a une symétrie centrale. Cependant, parcourir un état qui est une transposition d'un état déja visité est extremement rare voir impossible donc non laissé dans l'implementation finale,
- si la profondeur maximale autorisée est atteinte ou si le temps alloué pour ce coup est coulé, on retourne la valeur heuristique,
- si l'on se trouve dans un état final, on retourne $get\_score() * player * 10000$. Cela permet de retourner un score positif en cas de victoire, supérieur à la valeur heuristique et plus grand plus le score est en notre faveur.

1.3 | MonteCarlo Tree Search

La stratégie Monte Carlo Tree Search (algorithme en annexe F) a été développée dans le but de voir si elle pouvait concurrencer un AlphaBeta avec une heuristique assez élementaire. Nous avons pu faire la première version et rapidement nous avons été confronté à un problème : le facteur de branchement est très élevé (voir G.1), ce qui rend l'exploration MTCS en profondeur beaucoup plus longue.

En développant notre algorithme, nous nous sommes aperçus de quelques soucis que nous avons essayé de résoudre :

- **Exploration non équilibrée de l'arbre**. En effet, initialement nous renvoyions $get\_score() * player$ comme score de simulation. Mais le score UCT utilisé pour la sélection avec la constante $c = \sqrt{2}$ n'est adapté que pour un score de simulation compris entre 0 et 1. Ainsi nous avons décidé de renvoyer en simulation 0 en cas de défaite, 0.5 en cas d'égalité et 1 en cas de victoire. Cela a permis d'avoir une exploration plus équilibrée tout en ayant plus de simulations sur les noeuds plus prometteurs.
- **Nombre de simulations**. Tout l'intérêt de MCTS est de pouvoir faire de nombreuses simulations afin d'avoir une tendance de résultat pour une action sélectionnée. Il se trouve que malgré nos optimisations appliquées à *ImprovedBoard*, nous avions que peu de simulations en utilisant une heuristique de simulation plutôt élaborée. Tandis qu'avec des simulations Random, nous pouvions en faire plus mais le MCTS devenait théoriquement moins performant.

1.4 | Autres idées

Nous avions pour projet d'implémenter une version hybride de MCTS et AlphaBeta où notre agent adopterait une strategie jusqu'à une certaine step puis une autre ensuite. En théorie, MCTS performe mieux que AlphaBeta lorsque le facteur de branchement est élevé. Dans notre cas, MCTS devrait être meilleur en début de partie que AlphaBeta. Cependant dû a la spécificité d'Avalam, MCTS est très mauvais en début de partie. En effet, le début de partie est à la fois peu important (une action ne déterminera pas l'issue de la partie) que décisif (un début de partie raté empêche de gagner). MCTS détermine si un état a une forte probabilité d'amener à la victoire, or celle-ci est sensiblement identique pour tous les états du début de partie. À cette etape du jeu, il ne faut pas choisir un état qui mène à la victoire mais une action qui nous fait gagner des points/prendre l'avantage, ce qui est plus facile avec un algorithme guidé par une heuristique telle qu'AlphaBeta.
Nous avons donc voulu tester la configuration suivante : BestMove en début de partie, MCTS en milieu de partie, AlphaBeta en fin de partie. Mais le résultat ne fut pas concluant.

1.5 | Stratégie finale

Au final, nous avons implémenté un AlphaBeta en recherche IDS. Cela nous nous permet un "pseudo parcours en largeur". En effet, minimax parcours en profondeur, il faut donc fixer une profondeur maximal d'exploration pour avoir un résultat pertinent. IDS permet de nous assurer d'avoir parcouru la profondeur donnée entièrement. Pour les 5 premières step, nous avons fixé la profondeur maximale à 1 pour plusieurs raisons :

- Au début, explorer plus d'une profondeur est très couteux pour un gain faible.

— Du fait de l'entrainement de notre heuristique, celle ci performe mieux en vision à un coup que à plusieurs profondeurs (si celle-ci est faible).
— Le temps gagné pour les premiers coups est récuperé pour allouer plus de temps aux prochains coups et donc explorer l'arbre à des profondeurs plus elevées à des moments critiques de la partie.

2 | Heuristiques

Nous venons de voir dans la section précédente les stratégies développées. Nous allons maintenant nous interesser aux heuristiques utilisées par ces stratégies. Elles ont toutes étaient entrainées par génétique (sauf 2.1.5)

2.1 | Les différents types d'Heuristiques

2.1.1 | Réseau neuronal élémentaire

La première heuristique conçue consistait à un réseau neuronal qui prenait en entrée les tailles des tours appartenant au voisinage ($3*3$ ou $5*5$) d'une action analysée. La sortie correspondait donc au score de l'heuristique. Très mauvaise, elle ne gagnait que 4% des matchs contre greedy.

2.1.2 | Fonctions d'observation

Ensuite, nous avons commencé à concevoir des fonctions d'observation (des exemples de fonctions d'observation seront donnés dans la section 2.2) qui avaient pour objectif de pouvoir être utilisées pour évaluer le plateau de jeu sans prendre en compte le coup que nous allions jouer. Ces fonctions d'observation prenaient donc en entrée uniquement le plateau ainsi que le joueur que nous jouons. Le score de l'heuristique renvoyé correspond à une somme pondérée des fonctions d'observation.

$$Heuristic(p,j) = \sum_{i}^{n} \alpha_i f_i(p,j) \tag{2.1}$$

Avec $f_i(p,j)$ fonction d'observation pour un plateau p et le joueur j. α_i son poids de pondération associé et n le nombre de fonctions d'observation total.

2.1.3 | Fonctions d'observation "SingleLoop"

Nous avons remarqué que nos fonctions d'observation parcouraient chacune au moins 1 fois complètement le plateau. Ainsi, plutôt que de faire x parcours de plateau pour les n fonctions d'observation. Nous avons préféré faire 1 seul parcours du plateau qui serait commun à toutes les fonctions d'observation.

2.1.4 | Réseau neuronal avec fonctions d'observation en entrée

Comme nous avons pu le voir avec la formule du calcul de l'heuristique 2.1, il s'agit d'une combinaison linéaire des fonctions d'observation. Or, une heuristique optimale pour nos fonctions d'observation conçues pouvait être une combinaison non linéaire de ces fonctions d'observation. C'est pourquoi nous avons aussi créée une heuristique qui est un réseau neuronal prenant en entrée les fonctions d'observation et renvoyant en sortie un score du plateau. Cependant, entrainer un réseau neuronal par génétique ne fonctionne pas bien surtout avec une faible puissance de calcul.

2.1.5 | Réseau neuronal, estimation du pourcentage de victoire

Pour cette heuristique, nous nous sommes inspirés de la stratégie MCTS. De manière aléatoire, nous avons généré des milliers d'états du plateau possibles du jeu. Et pour chacun de ces états, nous avons effectués des simulations (comme l'algorithme MCTS) dans le but d'établir un % de victoire à cet état du plateau. Enfin, nous avons entrainé un réseau neuronal de manière "classique" afin qu'il puisse déterminer ce % en prenant uniquement en entrée le plateau et le joueur. Malgré que le réseau neuronal soit très précis pour évaluer le % calculé, cette heuristique utilisée avec une stratégie AlphaBeta n'était pas concluante. En effet il aurait fallu générer beaucoup plus d'états du plateau étant donné le facteur de branchement, mais aussi effectuer plus de simulations avec une heuristique de meilleur qualitée.

2.1.6 | Fonctions d'observation calculées dans l'*ImprovedBoard*

Enfin, nous sommes restés sur la combinaison linéaire de fonctions d'observation. Afin d'optimiser leur calcul, nous avons décidé de faire en sorte que leur score soit directement calculé lorsque des actions sont jouées. En effet, si nous sommes à un état n et que nous jouons une action a, nos fonctions d'heuritiques allaient avoir un score qui change pour l'état $n+1$ que de par une variation locale due à l'action jouée a. Cela signifie qu'un parcours entier du plateau n'est pas nécessaire pour connaître le nouveau score. Une petite variante a été faite pour la formule de l'heuristique qui est maintenant la suivante :

$$Heuristic(p,j) = \sum_{i}^{n} \alpha_i f_i(p) * j \tag{2.2}$$

La différence se trouve que par défaut, on considère les poids α_i associés à un seul joueur, si nous voulons utiliser

l'heuristique pour le joueur opposé, il suffira de multiplier par -1 (par j) (le meilleur coup du joueur -1 correspond au pire coup du joueur 1).

2.2 | Définition des fonctions d'observation

Nous avons beaucoup évoqué les fonctions d'observation. Nous allons maintenant en lister quelques unes.

- `board_score` qui renvoie le score du plateau
- `board_tower5` qui renvoie le nombre de tours de 5 du joueur 1. Déclinable pour les autres tailles de tour (de 1 à 5) et pour le joueur -1.
- `board_isolated_tower3` qui renvoie le nombre de tours de 3 du joueur 1 qui sont isolées. Déclinable pour les autres tailles de tour (de 1 à 5) et pour le joueur -1.
- `board_towers_links_2_2` qui renvoie le nombre d'actions possibles permettant d'assembler une tour de taille 2 avec une tour de taille 2 (les deux tours appartiennent au même joueur 1). Déclinable pour les autres actions possibles (1-1, 1-2, 1-3, 1-4, 2-2, 2-3 ; nous rangeons par taille de tour absolue croissante) ainsi que pour deux tours au joueur -1 et deux tours de signes différents.

Nous avons tenté une normalisation de ces fonctions d'observation dans le but de pouvoir comparer les poids associés aux fonctions d'observation.

2.3 | Définitions des paramètres

Comme dit précédemment, les paramètres des heuristiques (et les poids/biais des réseaux neuronaux) ont été trouvés via un entrainement génétique (voir algorithme H). Aucune bibliothèque n'a été utilisée, ce qui nous permettait de tout contrôler. Du fait de notre faible puissance de calcul :

- chaque génération avait entre 30 et 50 individus,
- entre 10 et 30% des meilleurs individus étaient gardés d'une génération sur l'autre,
- le taux de mutation était de l'ordre de 1%
- pour notre heuristique finale, nous avons calculé 400 générations,
- chaque individu combattait un unique match contre tous les autres individus (pour 50 individus cela donne 1225 matchs, on ne faisait pas de matchs retour pour gagner du temps, l'assignation joueur 1 et -1 était aléatoire),
- nous avons utilisé 2 fonctions de scores, au départ la somme des score des matchs mais nous avions peur que cela encourage des agents qui perdent souvent mais gagne parfois avec beaucoup de points. Puis, nous selectionnions les agents avec le plus de victoires,
- les agents utilisaient la stratégie BestMove,
- le fichier `game.py` a été adapté pour pouvoir gérer l'entrainement génétique (un seul lancement du script permettait d'effectuer toutes les générations voulues, des fichiers `.json` étaient sauvegardés à chaque génération).

3 | Importance des étapes et gestion du temps

Dans cette partie, nous allons évoquer la gestion du temps appliquée (visible sur B.1) pour notre agent AlphaBeta.

De nos observations, nous avons remarqué 2 points importants. Premièrement, le début de partie est à la fois peu important et décisif (voir 1.5), il faut donc s'assurer que nous jouons des actions qui ne nous mettent pas en defaut, mais pas besoin de sélectionner la meilleure action. Ensuite, pour limiter les dégats d'une heuristique imparfaite, il faut, le plus tôt possible, être en capacité d'explorer l'arbre en entier. On arrive la dans un dilemme : si on alloue plus de temps à la fin de partie, on pourra effectivement explorer l'arbre en entier plus tôt mais lors du milieu de partie notre agent aura peu de temps et pourrait prendre de mauvaises décisions. De plus, parcourir l'arbre à la fin est beaucoup plus rapide que lors du milieu de partie. Par exemple à la step 15, explorer 5 profondeurs nous prend 1000 secondes contre 100 à la step 22 (I.1). Grâce à notre implementation IDS, nous pouvons allouer un temps d'exécution précis à chaque step.
Voici donc les différentes optimisations (en plus des optimisations des méthodes de la *ImproveBoard* et du calcul de l'heuristique) que nous avons mis en place pour gérer notre temps :

- jusqu'a la step 5, on ne cherche que dans la première profondeur (voir 1.4),
- on considère qu'un match dure 34 steps, pour chaque step supérieure à 5 et inférieur à 34, le temps alloué est $\frac{time\_remaining}{34/2}$ ainsi chaque coups a en théorie le même temps d'exécution. Cela nous assure de ne pas dépasser les 15 minutes de crédits,
- supposons que nous avons 60 secondes pour jouer un coup. En 40 secondes, AlphaBeta explore jusqu'à la profondeur 3. On sait qu'il ne pourra pas explorer jusqu'à la profondeur 4 dans les 20 secondes restantes, nous pouvons donc arrêter le calcul, et les 20 secondes seront distribuées dans les steps suivantes. Nous appellons cette fonctionnalité le `trimming` : si l'on sait qu'on aura pas le temps de calculer la profondeur suivante, on s'arrête. Nous avons donc simulé

des centaines de parties (I.1) afin de calculer pour toutes les steps et pour différentes profondeurs des `trimming_ratio` qui représentent le ratio de temps nécessaire entre un parcours à une profondeur n et une profondeur $n+1$. Par exemple, supposons que pour la step 8 et profondeur 3 le trimming ratio est de 32 si le temps qu'il nous reste apres le calcul de la step 3 est inférieur à 32 fois le temps que le calcul de la profondeur 3 nous a demandé, nous n'avons pas besoin de calculer la step 4.
Pour calculer ces trimming ratio, nous avons simulé plusieurs centaines de parties et enregistré le temps de parcours de différentes profondeurs. Nous avons ensuite calculé les ratios et pris celui correspondant au premier quartile afin de ne pas trop trimm (si l'on avait pris le max par exemple) et de ne pas manquer des trimm (en prennant le min). *(Ce choix de premier quartile est un peu arbitraire, cette optimisation ayant été faite en fin de projet nous n'avons pas eu le temps de trouver le meilleur ratio).*

— Les steps 18 à 28 sont des steps charnières : le ratio de passage de 3 à 4 est très grand (17 pour la step 18) et à partir de 18, il manque parfois quelques dizaines de secondes pour explorer une profondeur supérieure voir calculer l'arbre en entier pour les step 25 à 28. Nous avons donc multiplié le temps alloué pour ces step par 1.75 *(valeur elle aussi arbitraire car amélioration de dernière minute).*

— Enfin nous pensions sauvegarder la dernière profondeur maximale de recherche pour commencer la recherche IDS à celle-ci car nous savions que, si on avait le temps de visiter jusqu'à la profondeur x pour la step n, alors nous aurions le temps de visiter à la même profondeur au minimum à la step $n+2$. Nous ne l'avons pas implementé car nous voulions que notre agent soit capable de jouer plusieurs parties en même temps et ainsi éviter de partager des informations qui concernent une seule partie mais qui seraient utilisées pour plusieurs parties et qui engendreraient des pertes de performance voir des mauvaises actions.

4 | Création et évolution des agents

Nous avons expliqué dans les sections précédentes comment nous avons développé nos briques au fur et à mesure pour créer les agents. Voici sur la table 4.1 des résultats réalisés avec des agents intermédiaires et l'agent final AlphaBeta IDS "Links".

	Joueur	Greedy	BM	MCTS	SL	Links
Role	Advs					
	Greedy		-7	-2	-9	-9
	BM	6		-1	-2	-4
	MCTS	4	3		-1	-4
1	SL	10	4	3		-1
	Links	6	4	2	1	
	Greedy		6	4	10	6
	BM	-7		3	4	4
	MCTS	-2	-1		3	2
-1	SL	-9	-1	-1		1
	Links	-9	-4	-4	-1	
	%Vict	0	25	50	75	**100**

Table 4.1 – Résultats des matchs réalisés avec les agents remis

Note : SL est un AlphaBeta utilisant une heuristique type single loop (moins optimisée et moins entrainée que l'agent de la compétition, Links qui utilise les fonctions d'observation calculées dans *ImprovedBoard*). BM est un BestMove utilisant la même heuristique que l'agent de la compétition. MCTS utilise une fonction random de simulation.

5 | Avantages et limites

Stratégie	Avantages	Inconvenients
BestMove	Heuristique rapide à entraîner Temps de calcul par coup négligeable	Faible vision du jeu
AlphaBeta	Vision à plusieurs profondeurs avec sûreté via IDS	Dépend d'une heuristique non optimisée pour AlphaBeta
MCTS	Influence de l'heuristique plus faible	Dépendance forte au nombre de simulations effectuées

Nous utilisions la stratégie BestMove pour entrainer l'heuristique. Cela nous permettait de faire 1225 matchs en quelques minutes et nous pensions que l'heuristique pourrait être généralisée à AlphaBeta. Malheureusement ce n'est pas le cas, BestMove, s'interesse à un coup tandis qu'AlphaBeta s'interesse à un état, ce qui entraine des poids pour les fonctions d'observations potentiellement différents. Malgré cela, nous avons gardé cette methode d'entrainement car l'heuristique trouvée était meilleur que celle que nous pouvions faire à la main. Avec plus de temps ou plus de puissance de calcul nous aurions pu entrainer sur AlphaBeta voir même approfondir l'heuristique décrite en 2.1.4.

A | Jeu Avalam

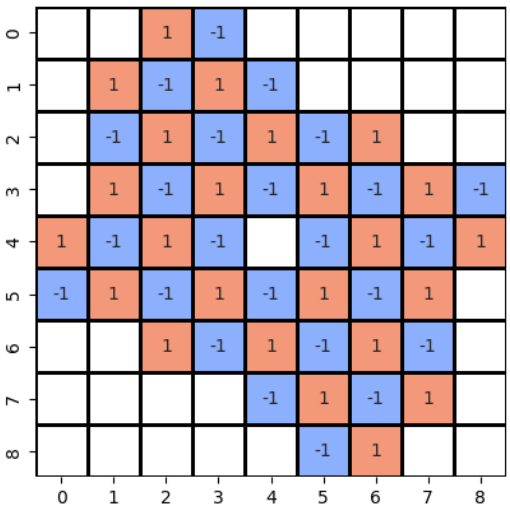


FIGURE A.1 – État initial du jeu

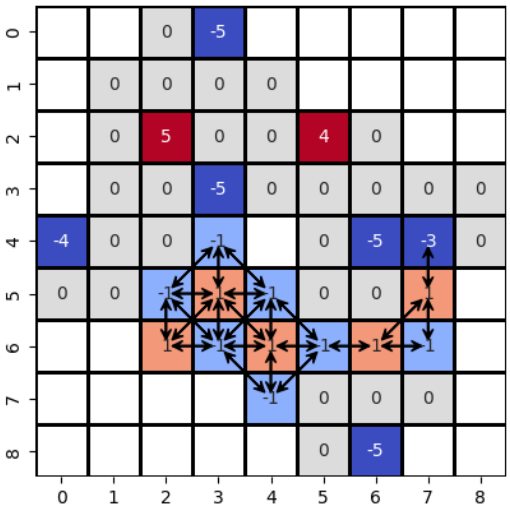


FIGURE A.2 – Actions possibles à un état avancé du jeu

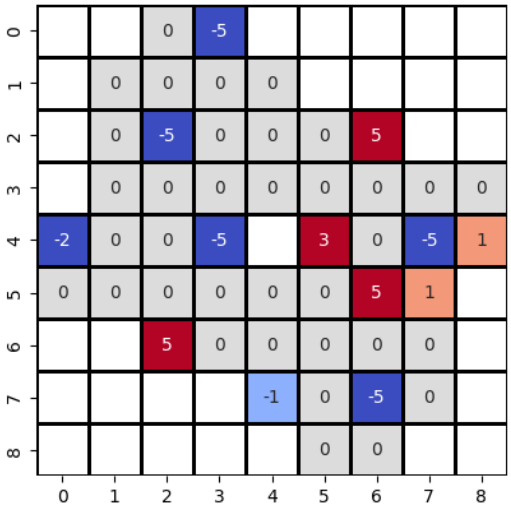


FIGURE A.3 – État final d'une partie (score -1)

B | Fichier gif généré

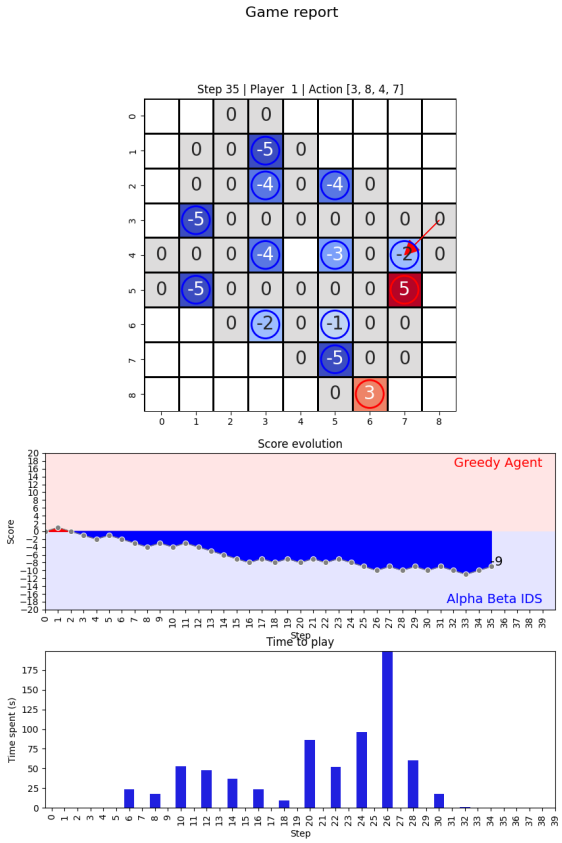


FIGURE B.1 – Dernière frame du gif généré pour une partie de Greedy contre AlphaBeta IDS

C | Diagramme de classes

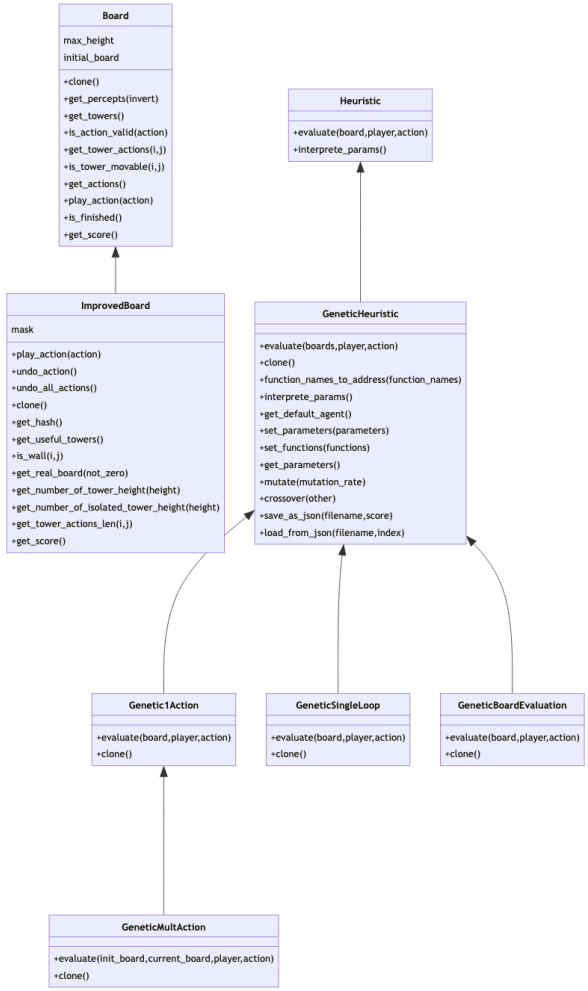


FIGURE C.1 – Board et Heuristiques

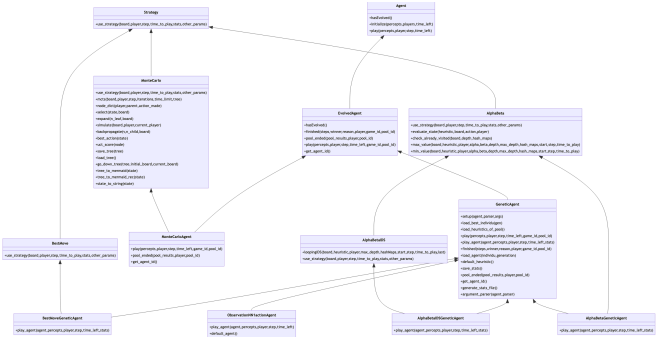


FIGURE C.2 – Agents et Stratégies

D | Algorithme BestMove

```
def best_move(board):
    max = -inf
    best_action = None
    for a in actions:
        value = heuristic(board, a)
        if value > max:
            max = value
            best_action = a
    return best_action
```

E | Algorithme AlphaBeta IDS

```
def alphabeta_IDS(board):
    while can_continue():
        _, action = max(board, -inf, inf)
    return action

def max(board, alpha, beta):
    v, m = trivial_case(board)
    if v:
        return (v, m)

    for a in sorted_actions:
        nV, _ = min(board, alpha, beta)
        if nV > v:
            v = nV
            m, alpha = a, min(alpha, v)
        if v >= alpha:
            return (v, m)

    return (v, m)

def min(board, alpha, beta):
    v, m = trivial_case(board)
    if v:
        return (v, m)

    for a in sorted_actions:
        nV, _ = min(board, alpha, beta)
        if nV < v:
            v = nV
            m, beta = a, min(beta, v)
        if v <= alpha:
            return (v, m)

    return (v, m)

def trivial_case(board):
    if is_finished(board):
        return score(board)
    if leaf(board):
        return heuristic(board)
    if visited(board)
        return hash_value(board)
    return (None, None)
```

F | Algorithme MonteCarlo Tree Search

```
def mcts(board):
    tree = init_tree(board)
    while can_continue():
        n_leaf = select(tree)
        n_child = expand(n_leaf)
        v = simulate(n_child)
        backpropagate(v, n_child)
    return best_action(tree)
```

G | Actions disponibles par step

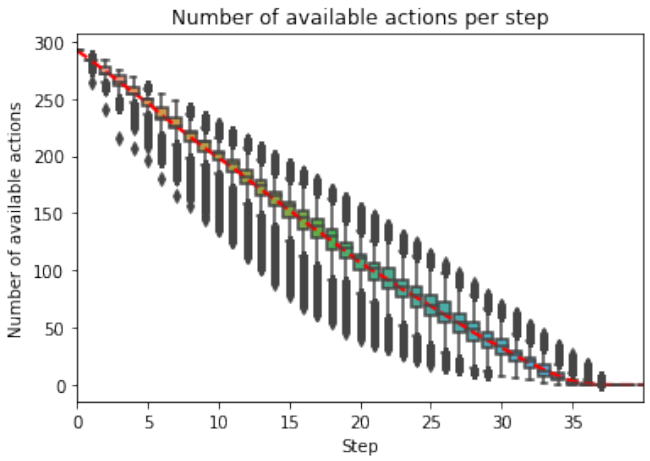


FIGURE G.1 – Nombres d'actions disponibles en fonction de la step

H | Algorithme d'entrainement génétique

```
def genetic(n, ite, s_rate, m_rate):
    p = random_individus(n)
```

```
        for i in range(ite):
            evaluate(p)
            best_individus = select(p, s_rate)
            p = crossover(best_individus)
            mutate(p, m_rate)
            i+=1
        return p
```

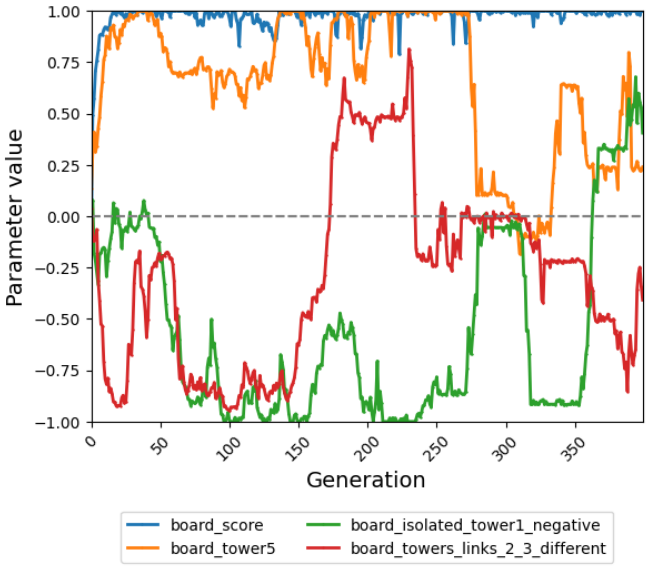


FIGURE H.1 – Valeurs des paramètres associés aux fonctions d'observation (moyenne de la génération) en fonction de la génération

Nous pouvons observer que certains paramètres semblent converger (comme celui associé à `board_score()`) tandis que d'autres sont toujours instables après 400 générations. De plus certains semblent être corrélés (ils varient en même temps), ce qui pourrait expliquer les instabilités et donc une non convergence.

I | Temps de calcul par step

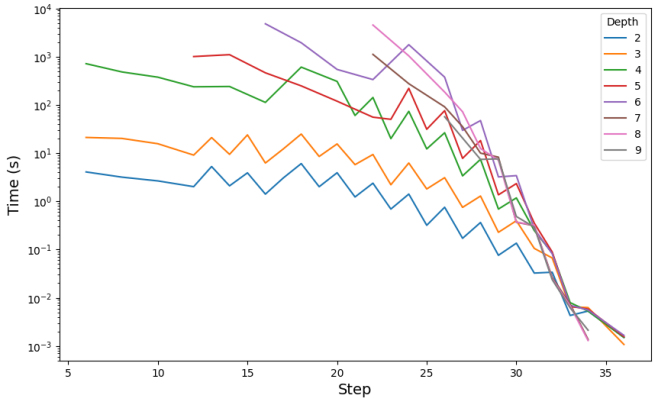


FIGURE I.1 – Temps de calcul pour une profondeur (échelle logarithmique) en fonction de la step