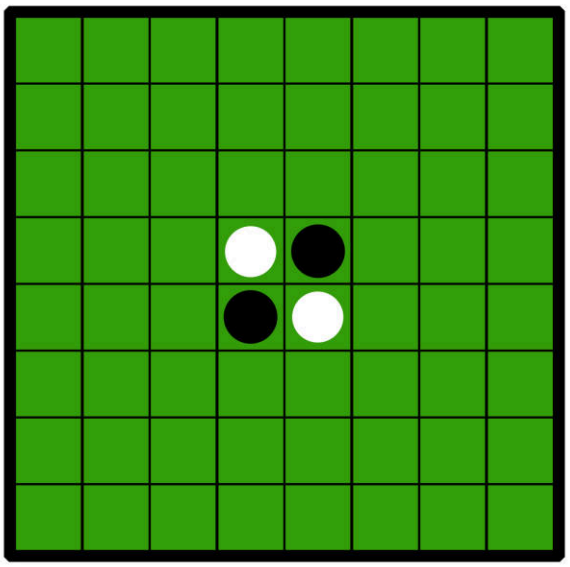
L3 MIASHS
15/12/2025

Projet IA
Reversi



Dehaene Noah n°12205415
Hajj Assaf Joe n°12208709
El Fejer Amine n°11612847

Table des matières:

I/Présentation

Le but de ce projet est d'implémenter un jeu en utilisant des techniques d'intelligence artificielle, et d'évaluer les performances de ces codes.

Nous avons choisi le Reversi qui est un jeu de stratégie pour deux joueurs sur un plateau de 8×8 cases. Chaque pion est bicolore : un côté Noir, un côté Blanc. Le joueur Noir commence toujours.

À chaque tour, un joueur pose un pion de sa couleur de façon à encercler une ou plusieurs lignes de pions adverses. Tous les pions ainsi pris sont retournés et changent de couleur. Les encerclements peuvent être horizontaux, verticaux ou diagonales. Si un joueur ne peut pas jouer, il passe son tour. La partie se termine quand plus aucun joueur n'a de coup valide.

Le gagnant est celui qui possède le plus de pions de sa couleur à la fin de la partie.

Nous avons implémenté notre jeu en utilisant les techniques vues dans le cours , notamment :

- Élagage alpha-beta
- Élagage alpha-beta optimisé
- A*
- BFS
- Greedy BFS
- DFS
- Dijkstra
- Dijkstra optimisé
- Monte-Carlo

II/Algorithmes

Tous nos bot Java héritent de la classe BotPlayer.java.

Élagage alpha-beta :

En principe , on élimine les branches de l'arbre de décision qui ne peuvent pas influencer le résultat final, rendant la recherche plus rapide sans changer le résultat.
Notre classe *AlphaBetaBot* implémente une stratégie de jeu se basant sur l'algorithme du Minimax avec élagage alpha beta. Il simule son coup puis les réponses possibles de l'adversaire, puis sa propre réponse et ainsi de suite jusqu'à une certaine profondeur. La profondeur (*maxDepth*) est définie au lancement du programme.

Quand c'est le tour de notre bot, il cherche le coup qui lui donnera le score le plus élevé (Maximisation).
Quand c'est le tour de l'adversaire , le bot suppose qu'il choisira un coup qui donne au bot le score le plus bas (Minimisation).
L'élagage alpha-beta est une optimisation de vitesse, si le bot se rend compte qu'il est en train d'examiner une séquence de coup qui mène à une situation pire qu'une autre déjà étudiée alors il arrête d'explorer cette branche. Cela permet de ne pas perdre de temps à explorer des coups non intéressants.

En clair, une fois que le bot a simulé les coups jusqu'à la profondeur , il évalue la qualité du plateau obtenu. Il utilise donc une méthode appelée *evaluateComplex* qui combine trois critères.

1. **Le contrôle des coins :** Au Reversi , les coins sont imprenables , nous avons donc accordé un coefficient 100 au fait de posséder des coins.

2. **La mobilité :** Le bot préfère les situations qui l'amène à avoir davantage de choix possible dans ses coups et où l'adversaire en a peu (coefficient 10).

3. **Le score :** Il regarde enfin le score de la partie qui est le critère d'évaluation le moins important durant la partie (coefficient 1) car dans ce jeu avoir beaucoup de pions au début est souvent un désavantage.

Le déroulement d'un tour:

1. Le bot regarde tous les coups qu'il a le droit de jouer.
2. Pour chaque coup, il crée une copie du plateau (pour ne pas toucher au vrai jeu).
3. Il lance l'algorithme Alpha-Beta sur cette copie.
4. Il retient le coup qui a obtenu la meilleure note finale et l'exécute sur le vrai plateau.

Ce bot privilégie donc la possession des coins et la mobilité plutôt que de simplement prendre le plus de pions possible immédiatement.

A* :

Le Bot A* évalue les gains immédiats et les gains espérés dans le futur. Pour chaque coût valide sur le plateau, le bot calcule une note finale **F** qui est la somme de 2 valeurs :

- **G** (le gain immédiat) : “qu'est ce que ce coup m’apporte concrètement tout de suite ?”
- **H** (le gain futur) : “Est ce que ce coût me place dans une situation favorable pour la suite de la partie ?”

Il choisit ensuite le coût qui a la note **F** la plus élevée.

Pour calculer le bénéfice immédiat (*calculateActualBenefit*), le score G, le bot regarde 4 critères concrets :

- **Le gain matériel (pions retournés):** Il compte combien de pions adverses sont retournés. C'est un facteur important (multiplié par 2,5).
- **La capture de coin:** Si le coût permet de prendre un coin, il reçoit un bonus énorme (+15 points), car cela est souvent la clé de la victoire au Reversi.
- **La stabilité:** Il vérifie si le pion posé est “stable” (sur un bord ou un coin), c'est à dire difficile à reprendre pour l'adversaire (souvent avec des d’autres pions “allié” aux alentours.
- **La mobilité :** Il calcule comment ce coût modifie le nombre de possibilités du jeu. Il cherche à augmenter ses propres options tout en réduisant celles de l'adversaire.

Pour estimer le potentiel futur (*calculateHeuristicEstimate*), le score H, le Bot se projette un peu plus loin, il simule le coup sur un plateau imaginaire et évalue la situation résultante, selon 5 critères stratégiques:

- **Contrôle des coins :** Il regarde s'il possède déjà des coins ou s'il a accès à un coin au prochain tour.
- **Contrôle des bords :** Il évalue sa présence sur les côtés du plateau par rapport à l'adversaire.
- **Avantage de mobilité:** Il vérifie s'il y aura plus de liberté de mouvement que l'adversaire dans cette future configuration.
- **Potentiel de stabilité :** Il favorise les positions où ses pions sont connectés au coin (imprenable).
- **Simulation :** Il fait une petite simulation rapide sur une profondeur de 3 coups supplémentaire. Cela lui permet d'éviter des pièges évidents, même s'il ne calcule pas toutes les possibilités de façon exhaustive pour gagner du temps.

En clair, le bot A* (*AstarBot*) est plutôt agressif sur les coins. Il cherche aussi une stabilité sur les bords et ne joue pas de manière “impulsive", Il accepte de gagner moins de pions dans l'immédiat, si cela lui garantit une meilleure position (mobilité et contrôle) pour la suite de la partie.

BFS (Breadth-First Search)：

Contrairement à l’*AstarBot* qui suit une intuition précise (heuristique), le *BFSBot* procède par niveaux. Concrètement, on donne à ce Bot une profondeur d'exploration (*maxDepth*). Il regarde tous les scénarios possibles à 1 coup, puis tous ceux à 2 coups, puis à 3 coups, etc… jusqu'à *maxDepth*.

Sa logique de réflexion :

1. **Exploration par niveau (queue)**: Il utilise une file d'attente (*queue*). Il empile les états du plateau les uns après les autres. Il ne traite les états du tour 3 que lorsqu'il a fini d'analyser tous les états possibles du tour 2. Et il fait ça jusqu'au tour maxDepth.
2. **Évaluation simple:** Ce Bot n'a pas de stratégie complexe, il compte simplement la différence de pions (Mon score - score adverse). Il garde en mémoire le meilleur score (*bestscore*) rencontré n'importe où dans ses recherches pour chaque coup de départ.

Déroulement typique de son tour:

1. Le Bot regarde le plateau et liste tous les coups qu'il a le droit de jouer. S'il n'y a aucun coup, il passe son tour. S'il n'y a qu'un seul coup, il le joue immédiatement sans réfléchir (gain de temps).
2. Pour un coup potentiel **A,** le bot copie le plateau et joue A sur cette copie. Il vérifie s'il gagne immédiatement la partie avec ce coup et si oui, il le joue immédiatement sans chercher plus loin.

3. Si la partie continue, il lance l'algorithme BFS pour le coup A avec une profondeur limite *maxDepth* (=6).

   Concrètement, il place le plateau (après le coup A) dans une file d'attente.

   Il démarre une boucle de traitement dans laquelle il prend le premier plateau de la file, regarde à qui c'est le tour, génère tous les coups possibles depuis cet état, crée tous les nouveaux plateaux résultant et les ajoutent à la fin de la file d'attente pour être traité plus tard (niveau après niveau).

   A chaque fois qu'il examine un plateau dans la file, que ce soit une profondeur 2, 4 ou 6, il évalue le score. Il compare ensuite ce score avec son score actuel (*bestscore*) et garde le meilleur des deux.

   Il arrête de parcourir une branche si la partie est finie ou si la profondeur maximale est atteinte.

4. Une fois qu'il a exploré, tout l'arbre des possibles pour le coup A, il fait pareil pour B, C et tous les autres coups valides. Le coup finalement joué sera celui qui l'amènera à l'état où il a le meilleur score possible.

En clair, le *BFSBot* est un opportuniste, il n'a pas de défense spécifique, il choisit simplement le chemin qui peut l'amener au plus gros gain.

GreedyBFS：

Le *GreedyBFSBot* n'évalue pas plusieurs coups à l'avance. Il regarde simplement sa liste de coups disponibles et sélectionne celui qui lui rapporte le plus de jetons.

Logique de réflexion:

1. Sa priorité est de retourner un maximum de pions, c'est son critère avec le poids le plus fort (x3).
2. Il prend en compte la haute valeur des coins (+20 points) et la dangerosité des cases adjacentes (-10 points).
3. Il prend en compte les coûts qui réduisent le choix de l'adversaire (poids x 1.5).
4. Il classe tous ses coups possibles dans une file d'attente et fait monter en haut de la liste les coups avec les plus gros scores.
5. À la fin, il joue le coup situé en haut de la file d'attente.

En clair, le *GreedyBFSBot* est un joueur opportuniste, mais il n'a pas une vision à long terme.

DFS (Depth-First Search)：

Contrairement au *BFSBot* (qui regardent tout ce qui se passe au tour 1, puis au tour 2), le *DFSBot* choisit un chemin et le suit jusqu'au bout avant de revenir en arrière pour en essayer un autre. Il explore l'arbre des possibilités branches par branches et ne s'arrête pas tant qu'il n'a pas atteint la profondeur maximale (*maxDepth*) ou la fin de la partie.

Cet algorithme repose sur une récursivité dans sa méthode *performDFS.* Il choisit l'un de ces coups possibles, descend immédiatement au niveau suivant, prend le premier coup possible de l'adversaire, descend encore au niveau suivant et ainsi de suite jusqu'à atteindre la profondeur. Lorsqu'il a exploré tous les coups possibles, les siens ou ceux de l'adversaire (en fonction de à qui est le tour à la profondeur maxDepth), il remonte d'un niveau pour recommencer et ainsi de suite jusqu'à avoir simulé tous les coups.

Comme pour le BFS Bot, il va sélectionner le coup qui l'amène au meilleur score à la fin d'une des branches.

Dijkstra :

L'objectif du DijkstraBot est de trouver le "chemin le plus court" vers la victoire.

- Un état avantageux (beaucoup de points, bonne position) est considéré comme "très proche" (distance négative).
- Un état désavantageux est "très loin".

En minimisant cette distance, il cherche mathématiquement à maximiser son avantage. Contrairement aux bots BFS et DFS précédents, celui-ci accorde une valeur à la position des pions.

Logique de réflexion:

1. **Évaluation de la situation** *(evaluateBoardAdvantage):* Il note la qualité du plateau selon 3 critères :
    - **Une matrice de poids** attribue une valeur aux pions selon leur position. Par exemple, un coin vaut +100 et les cases juste devant valent -20 ou -50 car risquées.
    - **La mobilité** qui signifie qu'il est plus favorable de jouer un coup qui lui accorde davantage de possibilités futures.
    - **Le score** évalue la différence de score entre lui et l'adversaire.
2. **La file prioritaire** *(priorityQueue)***:** Il explore d'abord les pistes qui sont les plus prometteuses. Si le coup B semble bien, il va creuser les conséquences de B avant de regarder les autres coups.
3. **Mémoire des états** *(distances Map)*: Si 2 séquences de coups différentes mènent à la même disposition de plateaux, Il garde le chemin le plus efficace et évite de recalculer ce qu'il sait déjà.

Déroulement typique de son tour:

1. Il regarde ses coups possibles: A, B, C etc… et attribue un coût initial à chacun (par exemple, si A prend un coup à x distance, il sera très faible. Si B joue dans une zone dangereuse, sa distance sera très élevée). Il met ensuite tout ça dans sa liste de priorités.

2. Il prend le meilleur coup actuel, simule une réponse de l'adversaire, calcule le nouveau score et met à jour le coût total de ce chemin.

   Si le chemin reste prometteur, il continue de l'explorer, sinon il le met de côté et va voir si le coup suivant n'était pas une meilleure idée.

3. Il continue l'exploration jusqu'à avoir atteint la profondeur égale à maxdepth (ou jusqu'à la fin de la partie).

4. Enfin, il regarde quel chemin a le coût final le plus bas, donc l'avantage le plus grand, remonte le fil d'Ariane pour retrouver quel était le premier coup de ce chemin jugé idéal et le joue.

En clair, le *dijkstraBot* agit comme un expert de la position. Il ne se fait pas avoir en donnant un coin car il sait que c'est une catastrophe grâce à la matrice de poids. Il est également plus efficace que le *BFSBot* et *DFSBot* car il perd moins de temps à analyser des coûts jugés visiblement mauvais.

Monte Carlo :

Le *MonteCarloBot* évalue simplement les probabilités pour définir son prochain coup. C'est un statisticien, il ne connaît aucune stratégie. Il se pose simplement la question: "Si je joue ce coup et que le reste de la partie se joue complètement au hasard, quelles sont mes chances de gagner ?".

Logique de réflexion:

1. Pour chacun de ses coups valides, il va simuler 1000 parties jusqu'à la fin avec des coups aléatoires *(simulateRandomGame)*.

2. Il utilise ensuite le principe de la loi des grands nombres. Même si une partie aléatoire ne veut rien dire, la moyenne de 1000 parties aléatoire donne une indication très précise de l'utilité d'une position. Si une position permet de gagner 80% du temps, même en jouant au hasard, c'est que cette position est très utile.

Le *Monte-CarloBot* est imprévisible et redoutable en fin de partie. Au début de la partie, il peut sembler jouer étrangement (l'arbre des possibles est trop grand). Mais à la fin de la partie il devient presque imbattable car il reste peu de cases vides et ses simulations couvrent presque toutes les fins de partie.

III/Tests

Les deux catégories d'IA

Avant d'effectuer les tests, il est important de souligner que ces bots sont classées dans deux catégories différentes.

Nous avons d'une part les bots déterministes. Ces bots là jouent toujours exactement le même coup si on leur présente le même plateau (avec la même disposition de pions).
Dans cette catégorie, on retrouve:

-Le BFSBot

-Le DFSBot

-Le DijkstraBot

-Le AstarBot (A*)

-Le GreedyBFSBot

À l'inverse, dans l'autre catégorie, nous avons les bots non déterministes parmi lesquels:

-Le RandomBot (plutôt logique)

-Le MonteCarloBot

Il est important de savoir cela car pour les tests, il ne sert à rien de faire jouer plusieurs fois l'un contre l'autre deux bots déterministes. Nous obtiendrons toujours le même score, quel que soit le nombre de parties jouées.

Difficultés rencontrées

Tous les bots commencent par demander la liste des coups valides. Or, dans la classe *ReversiPlateau*, cette liste est construite avec une double boucle for qui parcourt toujours le plateau de la case (0,0) à la case (7,7) dans le même ordre. La conséquence de cela est que si les coups possibles sont A, B et C, ils seront toujours présentés au bot dans l'ordre ce qui le forcera, si le plateau est le même qu'une situation déjà rencontrée, à évaluer les possibilités toujours dans le même ordre et donc à sélectionner les même coups (si élagage ou si le bot compare les scores et garde l'ancien *bestScore* en cas d'égalité).

Dans les bots un peu plus stratégiques comme *DikjstraBot* ou *Astarbot*, l'évaluation de la qualité d'un coup repose sur des constantes fixes, des poids qui ne changent jamais. Donc, si les plateaux sont identiques, le score calculé sera toujours le même.

Il y a plusieurs solutions qui permettent de ne pas avoir toujours le même résultat lorsque l'on fait jouer des bots déterministes.

1. La première est de faire jouer ces bots contre des bots non déterministes.
2. Une autre solution est d'ajouter un peu de hasard dans les choix des bots déterministes.

En soi, l'une des solutions qui pourrait régler tous ces problèmes serait de mélanger la liste des coups valides avec une *Collections.shuffle(validMoves);*. C'est une méthode qui mélange aléatoirement les éléments d'une liste. Cela permettrait de rendre les algorithmes un peu plus imprévisibles.

Deuxième difficulté, les tests peuvent prendre beaucoup de temps. En effet, dans notre fichier *Reversiplateau.java*, le plateau est stocké sous forme d'un tableau à 2 dimensions. Pour copier ce tableau, l'ordinateur doit effectuer une série d'actions répétitives et chronophages:

- L'ordinateur doit exécuter une boucle qui s'arrête sur chacune des 64 cases du jeu.
- Pour chaque case, il doit lire la valeur (Noir, Blanc, Vide) puis écrire cette valeur dans la case correspondante de la copie.

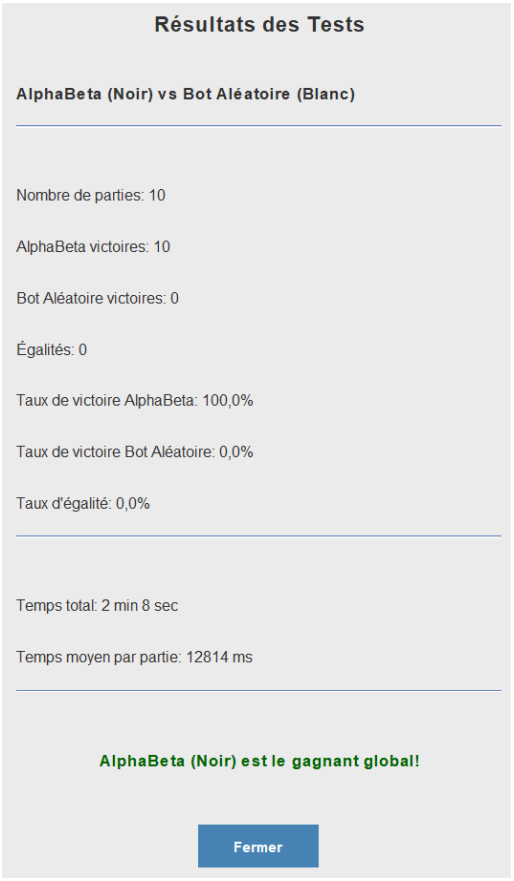
Or en Java, un tableau à 2 dimensions est dispersé dans la mémoire. Pour passer d'une ligne à l'autre, l'ordinateur doit chercher l'adresse de la nouvelle ligne en mémoire, ce qui prend du temps supplémentaire. L'accumulation de ces 64 opérations de lecture/écriture rend l'algorithme extrêmement lent.

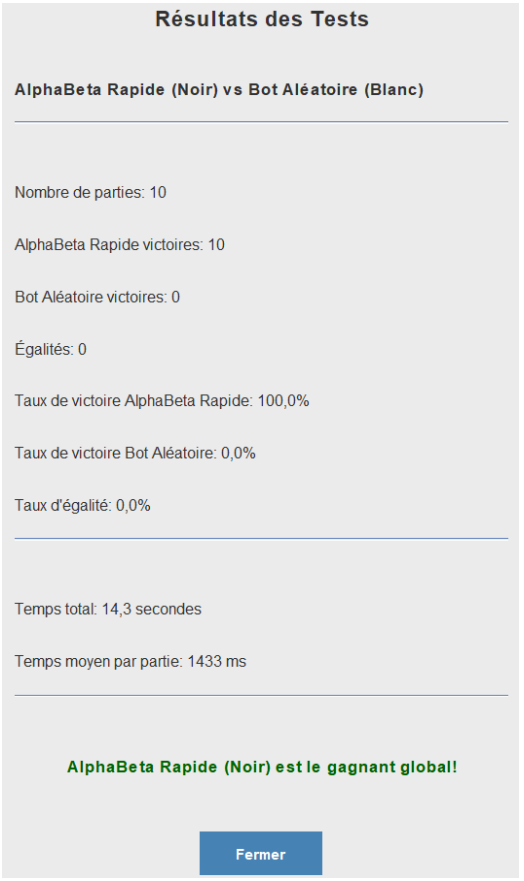
La solution qui a été trouvée est la création d'un nouveau fichier appelé *FastReversiBoard.java*. Le plateau est stocké sous la forme de 2 nombres entiers. Le principe est que ces nombres contiennent toute l'information du jeu (position de tous les pions).

Au lieu de faire 64 opérations dispersées, il n'en fait qu'une seule, ce qui rend la copie quasiment instantanée.

Les bots qui fonctionnent avec ce plateau portent le nom *nomdel'algorithme*Rapide.java. Nous avons laissé les anciens fichiers qui fonctionnent sur l'ancien plateau pour la comparaison.

Par exemple, si on compare AlphaBeta.java à AlphaBetaRapide.java sur 10 parties contre le *RandomBot .*





On voit bien que le temps moyen par partie pour le bot rapide est significativement inférieur à celui de l'autre bot (12814>1433 ms).

Résultats

Nous avons fait jouer chaque bot contre le *RandomBot*.

	Pourcentage De Victoires	Temps Moyen	Nombre De Jeux
BFS	74%	86 140 ms	100
DFS	83%	105 254 ms	100
Dijkstra	80%	3 419 ms	100
Greedy BFS	78%	3 ms	10 000
A*	84%	31 ms	10 000
Alpha-Beta	90%	2 192 ms	100
Monte Carlo	100%	22 923 ms	100

Observations:

- BFS et DFS ne sont pas optimisés, ils ont un très grand temps moyen, et un plus petit pourcentage de victoires comparés aux autres Bots.
- Greedy BFS est très rapide, avec un pourcentage de victoires approximativement similaire aux BFS et DFS.
- Dijkstra est un peu lent même avec l'optimisation, ce qui est normal car il simule un nombre important de parties.
- A* est très rapide avec un pourcentage de victoires satisfaisant.
- Alpha-Beta est le meilleur algorithme, il prend un peu de temps pour jouer mais compense cela avec un pourcentage de victoires très élevé.
- Monte Carlo ayant un pourcentage de victoires de 100% est surprenant, on le compare alors avec Alpha-Beta pour voir lequel est le meilleur pour le jeu de Reversi.

	Pourcentage De Victoires	Temps Moyen	Nombre De Jeux
Alpha-Beta | Monte Carlo	71% | 29%	16 667 ms	100

On pourrait donc conclure que l'algorithme Alpha-Beta est le meilleur en confrontations directes parmi ceux présents dans notre projet, mais est-ce vraiment le cas ?

Essayons de vérifier cela en faisant jouer tous les algorithmes entre eux. Nous avons ajouté la ligne de code qui permet de mélanger la liste des coups valides afin que les résultats soient moins prévisibles entre les bots déterministes.
Le tableau suivant donne le pourcentage de victoire de l'algorithme de la ligne contre celui de la colonne sur 100 parties (si l'addition des valeurs ne donne pas 100%, le reste est le pourcentage de match nul).

	A*	**BFS**	**DFS**	**Dijkstra**	**Monte Carlo**	**Alpha-Beta**
A*		87%	83%	100%	17%	0%
BFS	12%		48%	51%	0%	34%
DFS	14%	52%		51%	0%	31%

Dijkstra	0%	46%	47%		0%	56%
Monte Carlo	79%	100%	100%	100%		27%
Alpha-Beta	100%	62%	62%	44%	72%	

Donc si on fait un classement par rapport à la moyenne des pourcentages de victoires, on a:

1. **Monte Carlo**: 81.2% de victoires.
2. **Minimax avec élagage Alpha-Beta**: 68% de victoires.
3. **A***: 57,4% de victoires.
4. **Dijkstra**: 29,8% de victoires.
5. **DFS**: 29,6% de victoires.
6. **BFS**: 29% de victoires.

L'algorithme de Monte-Carlo se place donc comme grand gagnant de ce tournoi. Il domine totalement BFS, DFS et Dijkstra avec 100% de victoires. C'est assez logique car ces bots ont une vision du jeu à long terme plutôt limitée, avec des heuristiques rigides. Monte Carlo, lui, simule des centaines de parties pour chaque coup et par conséquent, ne fait quasiment aucune erreur en fin de partie (moment décisif).
Pourtant, il se fait tout de même battre par l'algorithme avec élagage Alpha-Beta. Cela vient probablement du fait que ce dernier propose une défense solide qui laisse peu d'occasions à Monte Carlo de prendre l'avantage.

L'algorithme de Dijkstra se fait battre dans une majorité des cas. Néanmoins, il arrive à battre Alpha-Beta. Il semble que le jeu très positionnel de Dijkstra, avec la matrice de poids, soit la meilleure stratégie à employer contre Alpha-Beta.

Les bots avec algorithme BFS et DFS semblent arriver à un taux de victoire de 50% chacun à mesure que le nombre de match entre eux augmente.

Dans l'ensemble, les résultats semblent montrer une sorte de boucle entre les quatre premiers algorithmes du classement. En effet on voit que:

1. Alpha-Beta l'emporte sur Monte Carlo (72% contre 27%)
2. Monte Carlo bat A* (79% contre 17%)
3. A* bat Dijkstra (100% contre 0%)
4. Dijkstra bat Alpha-Beta (56% contre 44%)

IV/Conclusions

Sur la base de nos tests qui n'ont simulé que 100 parties pour chaque match (nos résultats auraient sûrement été plus précis avec un nombre plus important de parties jouées), nous avons pu voir que l'IA probabiliste Monte Carlo était globalement imbattable contre les autres IA déterministes.

L'IA utilisant l'algorithme du minimax avec élagage Alpha-Beta semble être une bonne solution si l'on veut gagner rapidement face à un adversaire qui joue au hasard.

La conclusion est qu'aucun de nos bots n'est invincible. Chacun peut espérer gagner contre un adversaire. Une IA capable de reconnaître la stratégie de l'adversaire et d'adapter sa propre stratégie en conséquence, aurait très certainement trouvé sa place sur le haut du podium.

Cette conclusion est valable pour la façon dont nous avons paramétré nos algorithmes. Nous pouvons admettre des résultats différents si par exemple, nous modifions la valeur des poids des heuristiques chez certains algorithmes.

V/Annexe

Guide d'utilisation de l'interface:

L'interface est facile à utiliser. Sur Eclipse, après l'exécution de la classe Main, une fenêtre s'ouvre pour sélectionner le mode de jeu pour chacun des deux joueurs, ce qui permet de jouer contre l'ordinateur, utilisant l'algorithme choisi, ou de tester le jeu entre deux algorithmes (ordinateur contre lui-même).
Il suffit donc de sélectionner l'algorithme voulu pour chaque joueur et cliquer sur "**commencer la partie**". La partie sera visible sur un plateau affiché sur l'écran.
À la fin de la partie, une fenêtre s'ouvre pour annoncer le joueur gagnant, et les scores. Il y a deux boutons, l'un pour rejouer avec les mêmes choix des joueurs (rejouer), l'autre pour rejouer en choisissant de nouveau le mode de jeu et les joueurs, comme au début (nouvelle partie).

Si on veut simuler plusieurs parties entre 2 bots, il faut appuyer sur le bouton "**Simuler plusieurs parties**". Cela ouvrira une nouvelle fenêtre où vous pourrez sélectionner les bots dont vous voulez simuler l'affrontement, ainsi que le nombre de parties que vous voulez simuler.
La progression de la simulation sera visible en bas de cette fenêtre, elle affichera la partie simulée et le temps mis pour terminer chaque partie.
À la fin de la simulation, une nouvelle fenêtre s'affiche faisant un récapitulatif, avec le taux de victoire pour chaque bot, le taux d'égalité, le total du temps mis pour faire la simulation, le total du temps moyen par partie, et pour finir, l'annonce du gagnant global (celui qui a remporté le plus de partie).

Sources:

-Wikipedia contributors. (2025, 9 décembre). *Monte Carlo method*. Wikipedia. https://en.wikipedia.org/wiki/Monte_Carlo_method

-Fédération Française d'Othello. (2008). *Algorithme MinMax – élagage AlphaBeta*. https://pageperso.lis-lab.fr/~liva.ralaivola/teachings20062005/reversi/MinMaxL2.pdf

-Wikipedia contributors. (2025a, octobre 4). *Alpha–beta pruning*. Wikipedia. https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning?utm_source=chatgpt.com

-Contributeurs aux projets Wikimedia. (2025, 25 novembre). *Algorithme de parcours en largeur*. https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur#:~:text=L%27algorithme%20de%20parcours%20en,non%20explor%C3%A9s%20des%20successeurs%2C%20etc.

-Contributeurs aux projets Wikimedia. (2025a, novembre 24). *Algorithme de parcours en profondeur*. https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur

-Contributeurs aux projets Wikimedia. (2024, 13 septembre). *Algorithme a**. https://fr.wikipedia.org/wiki/Algorithme_A*

-Contributeurs aux projets Wikimedia. (2025a, novembre 14). *Algorithme De Dijkstra*. https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra

-*W3Schools.com*. (s. d.). https://www.w3schools.com/java/

-*W3Schools.com*. (s. d.-b). https://www.w3schools.com/html/html_basic.asp

-*L'intelligence artificielle pour les développeurs, concepts et implémentation en Java* (2e éd.). (2019). ENI.

-Cloner de github à Eclipse : https://sos-cer.github.io/projects/git-guide/git-clone.html

Distribution du travail :

Joe :

- Amélioration de l'interface graphique tel hints pour joueur humain, et fenêtre de résultat
- Ajout des algorithmes : A*, BFS, DFS, Dijkstra, Greedy BFS
- Implémentation des commentaires Javadoc + génération du site
- Format et positionnement des fichiers sur github pour suivre les conventions
- Ajout des tests + son interface graphique
- Tableaux des tests contre RandomBot et Alpha-Beta vs Monte Carlo dans rapport + idées principales liées
- Rédaction du README.md
- Petites modifications et corrections du rapport

Noah :

- Rédaction du rapport
- Création du jeu Reversi de base
- Ajout des algorithmes RandomBot, AlphaBetaBot, AlphaBetaBotRapide.
- Génération du tournoi général, chaque algorithme joue les uns contre les autres.
- Ajout des résultats du tournoi général.
- Modifications et corrections des fichiers Java.

Amine :

- Aide au brainstorming initiale du projet