

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DES HAUTS-DE-FRANCE



Département d'Informatique et de Cybersécurité

RAPPORT DES TRAVAUX PRATIQUES

INTELLIGENCE ARTIFICIELLE APPLIQUÉE AU JEU D'OTHELLO

Date : 15 avril 2024

Professeur : René Mandiau - HDR

Elias BOULANGER
INSA Hauts-de-France - ICY
Université Polytechnique Hauts-de-France
elias.boulanger@uphf.fr

Table des matières

1	Introduction	1
2	Analyse	2
1	Modélisation et structures de données	2
1.1	Structure du projet	2
1.2	Bitboards	3
1.3	Structure Node	5
1.4	Boucle de jeu et séparation des connaissances	6
2	Algorithmes	7
2.1	Décaler un bitboard	7
2.2	Trouver les coups valides	7
2.3	Jouer un coup	9
2.4	Minimax et Alpha-Beta	10
2.5	Évaluateur de la qualité d'une position avec PyTorch	11
3	Validation	15
1	Mesure de complexité : Exploration de l'arbre de jeu	15
2	Championnat : Comparaison des configurations	18
4	Discussion	20
1	Complexité et exploration de l'arbre de jeu	20
1.1	Analyse temporelle	20
1.2	Exploration de l'arbre de jeu	21
2	Performance des agents et analyse des parties	22
2.1	Performance des Heuristiques et par Couleur	22
2.2	Comparaison des Stratégies et Avantages par Couleur	22
5	Conclusion	24
A	Algorithmes et Code	I
1	Opérations Logiques	I
2	Trouver et jouer coups valides	III
3	Minimax	IV
B	Résultats et statistiques	VIII
1	Graphiques individuels des nœuds explorés	VIII

Liste des Acronymes

IA Intelligence Artificielle

LSB Bit de Poids Faible

MSB Bit de Poids Fort

PdV Point de Vue

TP Travail Pratique

Table des figures

2.1	Exemple d'une configuration (non complète).	3
2.2	Configuration initiale du plateau de jeu. Source de l'image : eOthello.	4
3.1	Nombre de nœuds explorés par Minimax et Alpha-Beta en profondeur 2. . .	16
3.2	Nombre de nœuds explorés par Minimax et Alpha-Beta en profondeur 4. . .	16
3.3	Nombre de nœuds explorés par Minimax et Alpha-Beta en profondeur 6. . .	16
A.1	Opérations logiques pour obtenir l'état du plateau.	I
A.2	Opérations logiques pour définir l'état du plateau.	I
A.3	Opérations de décalage pour les coups valides.	II
A.4	Trouver les coups valides.	III
A.5	Jouer un coup.	III
A.6	Algorithme Minimax.	IV
A.7	Algorithme Minimax avec AlphaBeta.	V
A.8	Algorithme Negamax.	VI
A.9	Algorithme Negamax avec AlphaBeta.	VII
B.1	Nombre de nœuds explorés par coup pour la profondeur 2 avec Alpha-Beta. . .	VIII
B.2	Nombre de nœuds explorés par coup pour la profondeur 2 sans Alpha-Beta. . .	IX
B.3	Nombre de nœuds explorés par coup pour la profondeur 4 avec Alpha-Beta. . .	IX
B.4	Nombre de nœuds explorés par coup pour la profondeur 4 sans Alpha-Beta. . .	X
B.5	Nombre de nœuds explorés par coup pour la profondeur 6 avec Alpha-Beta. . .	X
B.6	Nombre de nœuds explorés par coup pour la profondeur 6 sans Alpha-Beta. . .	XI

Liste des tableaux

2.1	Résumé du modèle pour une taille de batch de 256.	13
3.1	Analyse comparative du temps d'exécution d'une partie en fonction des profondeurs et stratégies (rapport NegamaxAlphaBeta(gauche)/Negamax(droite))	15
3.2	Analyse comparative du nombre de nœuds explorés en fonction les profondeurs et les stratégies en pourcentage (rapport NegamaxAlphaBeta/Negamax)	17
3.3	Analyse comparative du nombre de nœuds explorés en fonction des profondeurs et stratégies (NegamaxAlphaBeta (gauche) vs Negamax(droite)) . . .	17
3.4	Analyse comparative des scores pour le Joueur Noir en fonction des profondeurs et stratégies (rapport Table d'heuristique 2 (gauche) / Table d'heuristique 1 (droite) et valeurs exactes entre parenthèses)	18
3.5	Analyse comparative des scores pour le Joueur Blanc en fonction des profondeurs et stratégies (rapport Table d'heuristique 2 (gauche) / Table d'heuristique 1 (droite) et valeurs exactes entre parenthèses)	18
3.6	Analyse comparative des scores en fonction des profondeurs et stratégies pour toutes les configurations du Joueur Noir	19
3.7	Analyse comparative des scores en fonction des profondeurs et stratégies pour toutes les configurations du Joueur Blanc	19

Chapitre 1

Introduction

Dans ce projet de Travail Pratique (TP), nous nous sommes attachés à développer et à évaluer des algorithmes d'Intelligence Artificielle (IA) pour le jeu d'Othello, un jeu de plateau impliquant une stratégie complexe malgré des règles apparemment simples. Othello se joue sur un damier de 64 cases avec des pions bicolores, où l'objectif est de capturer le plus grand nombre de pions adverses. Cette simplicité apparente masque une profondeur stratégique et une explosion combinatoire qui représentent un défi de taille pour la conception d'algorithmes efficaces.

Ce rapport détaille le processus de conception, d'implémentation et d'évaluation de notre système d'IA. Nous avons adopté une approche méthodique pour modéliser le jeu, en utilisant des structures de données adaptées pour représenter efficacement le plateau et les pions. L'accent a été mis sur l'élaboration et l'amélioration de l'algorithme Minimax, ainsi que sur ses variantes, notamment les algorithmes Alpha-Beta, qui réduisent le nombre de nœuds explorés et optimisent les performances de calcul.

Nous avons également développé une fonction d'évaluation pour mesurer la qualité des positions sur le damier, intégrant diverses heuristiques comme la position des pions, la mobilité et le contrôle du plateau. Ces heuristiques ont été testées dans différentes configurations, permettant une comparaison directe de leur efficacité à travers des matchs simulés entre notre IA et des joueurs humains (surtout quelques amis et moi), ainsi que contre d'autres configurations d'IA. En outre, ce travail inclut la conception d'un classifieur pour évaluer la qualité des positions.

Ce rapport présente donc non seulement les aspects techniques de notre approche, mais aussi une discussion sur les résultats obtenus, les défis rencontrés et les potentielles améliorations.

Nous concluons sur ce que nous avons appris de ce projet, sur les perspectives et les leçons tirées de cette expérience, avec finalement un point plus personnel.

Chapitre 2

Analyse

Notre implémentation comprend notamment une représentation binaire, aussi appelé Bitboard, les algorithmes de recherches Minimax et Alpha-Beta, et un classifieur pour prédire le résultat d'une partie en cours. Nous détaillons également les structures de données utilisées, et les algorithmes de décalage pour les coups valides. Nous avons réalisé ce projet en Python, pour des raisons de simplicité et de rapidité de développement, notamment à propos de la visualisation, et du développement du modèle de deep learning via PyTorch¹.

1 Modélisation et structures de données

1.1 Structure du projet

```
Othello-Reversi/
├── config.yaml : fichier de configuration des paramètres globaux du projet
├── main : point d'entrée du programme, permet de réaliser des tests, lancer
    des parties, faire jouer des algorithmes, etc
├── game.py : contient la logique de la boucle de jeu, permet de lancer une
    partie selon les paramètres donnés
├── node.py : contient la classe Node, encapsulant un état du jeu, et les informations
    nécessaires pour l'exploration de l'arbre de recherche, ou pour rejouer
    une partie
├── strategies.py : contient les algorithmes de recherche, et retourne le plateau
    après jouer un coup selon la stratégie choisie
├── next.py : contient les fonctions de calcul des coups valides, et celles
    pour jouer un coup
├── heuristics.py : contient les fonctions d'évaluation basées sur des heuristiques
├── utils/
│   └── Ensemble de fonctions utilitaires telles que des tables d'heuristiques,
│       la visualisation, les opérations logiques, etc
├── model_pipeline.ipynb : notebook Jupyter pour data preprocessing, model training,
    et évaluation
└── understanding_bitboards.ipynb : notebook Jupyter pour comprendre les bitboards,
    de nombreux exemples illustrent les opérations logiques
```

1. PyTorch est une bibliothèque de tenseurs optimisée pour l'apprentissage profond. Voir Documentation PyTorch.

Une partie peut être lancée depuis le programme `main.py`, ce dernier prendra les paramètres définis dans le fichier `config.yaml`. Par exemple, pour lancer une partie avec *Interface* entre un joueur *Humain* et un joueur *Negamax-AlphaBeta*, en utilisant une stratégie *Positionnelle* pour l'évaluation avec une profondeur d'exploration maximale de 4 coups, nous obtenons le fichier de configuration suivant :

```
1  ### Game Parameters ###
2  # Who plays against who.
3  # 0: human.
4  # 1: random (chooses randomly a possible move),
5  # 2: positional. (uses a heuristic table to define the quality of a position),
6  # 3: absolute (minimizes/maximizes the number of pieces for the opponent/player),
7  # 4: mobility (minimizes/maximizes the number of possible moves for the opponent/player),
8  # 5: mixed (uses a heuristic table to define the quality of a position).
9  # Mixed signifies using positional, then mobility, then absolute.
10 mode: [0, 2]
11
12 # Which MiniMax version to use.
13 # 0: NONE
14 # 1: Default MiniMax
15 # 2: Alpha-Beta pruning
16 # 3: Default Negamax
17 # 4: Alpha-Beta pruning Negamax
18 minimax_mode: [4, 4]
19
20 # Maximum depth of the search tree (minimax algorithm)
21 max_depth: [4, 4]
22
23 # Which heuristic table to use.
24 # 0: None
25 # 1: TABLE1
26 # 2: TABLE2
27 h_table: [0, 2]
28
```

```
1  # Whether to display the board with a graphical interface
2  display: False
```

FIGURE 2.1 – Exemple d'une configuration (non complète).

Plus de paramètres sont disponibles, accompagnés de leurs valeurs par défaut, ainsi que des descriptions et commentaires.

1.2 Bitboards

Une représentation classique d'un plateau de jeu d'Othello est une matrice de 8x8, où chaque case peut contenir trois valeurs : par exemple (-1, 0, 1) pour les pions noirs, vides et blancs respectivement. Celle-ci est intuitive, et est la première approche que nous avons envisagée. Cependant, nous avons finalement opté pour une meilleure alternative, plus efficace en termes de temps et d'espace : les bitboards.

En effet, au lieu de contenir 64 entiers de 32bits chacun (soit 2048bits), nous pouvons encoder l'ensemble du plateau de jeu dans 2 entiers de 64bits chacun (soit 128bits) : ce qui

est 16 fois moins lourd !

De plus, les opérations logiques sur les bitboards sont très rapides, un avantage supplémentaire pour les algorithmes de recherche.

Comment encoder un pion, un coup, ou un plateau de jeu ?

Nous pouvons en fait tous les encoder de la même manière, à travers un entier sur *64bits*. Chaque bit représente une case du plateau, sa valeur égale à 1 si elle est occupée par un pion, ou si le coup est valide. Le Bit de Poids Fort (MSB) correspond à la case *h8*, et le Bit de Poids Faible (LSB) à la case *a1*. Les positions sont usuellement notées de *a1* à *h8*, où *a* est la colonne la plus à gauche, et 1 la ligne la plus haute. [Ros05]

Par exemple, la configuration initiale du plateau de jeu est la suivante :

- Les pions noirs sont en *d5* et *e4*.
- Les pions blancs sont en *d4* et *e5*.

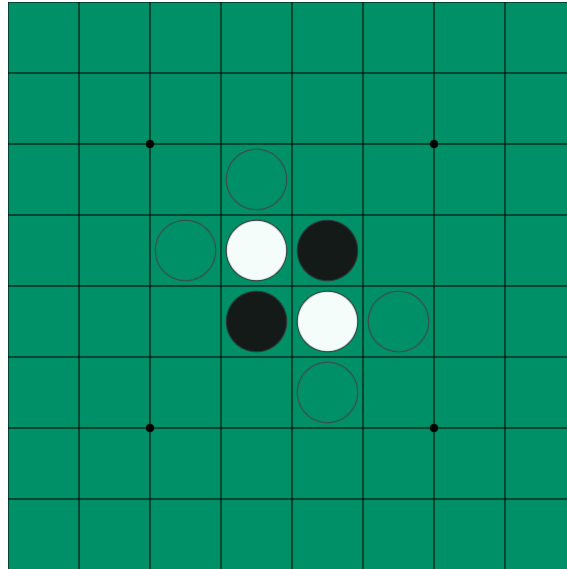


FIGURE 2.2 – Configuration initiale du plateau de jeu. Source de l'image : eOthello.

Cette dernière s'encode comme suit, avec dans l'ordre, le plateau noir, puis blanc :

```
0b00000000_00000000_00000000_00001000_00010000_00000000_00000000_00000000
0b00000000_00000000_00000000_00010000_00001000_00000000_00000000_00000000
```

Ces deux entiers peuvent être réécrits en hexadécimal pour une meilleure lisibilité :

`0x00_00_00_08_10_00_00_00` (*noir*)

`0x00_00_00_10_08_00_00_00` (*blanc*)

De fait, il est possible de récupérer la valeur de la case $c_{i,j}$ d'un bitboard b en utilisant la formule suivante :

$$(b \gg (8 \times i + j)) \ \& \ 1, \quad \forall i, j \in \{0, \dots, 7\}$$

Avec i la ligne, j la colonne, \gg l'opérateur de décalage à droite, et $\&$ l'opérateur logique 'et'.

Similairement, nous pouvons définir la case $c_{i,j}$ d'un bitboard b en utilisant la formule suivante :

$$b \mid (1 \ll (8 \times i + j)), \quad \forall i, j \in \{0, \dots, 7\}$$

Avec \ll l'opérateur de décalage à gauche, et \mid l'opérateur logique 'ou'.

Nous obtenons toutes les pièces posées et toutes les cases vides avec les opérations logiques suivantes :

$$\begin{aligned} \text{pieces} &= \text{noir} \mid \text{blanc} \\ \text{vides} &= \sim \text{pieces} \end{aligned}$$

Avec \sim l'opérateur logique 'non'. Les programmes équivalents en python sont donnés dans l'appendice 1. De manière générale, n'importe quel pion peut être ajouté au plateau avec l'opérateur logique 'ou'. Aussi, 8 peut être remplacé par n'importe quel entier représentant la taille du plateau, ici 8x8.

1.3 Structure Node

Dans le contexte de ce projet, nous aimerions pouvoir rejouer des parties, déterminer les états déjà traités, et reprendre les analyses formulés lors de tours précédents. Pour cela, nous avons défini une structure de données nommée **Node**, qui encapsule un état du jeu, et les informations nécessaires pour l'exploration de l'arbre de recherche. Nous l'avons défini dans le fichier `node.py` via une classe Python.

Attributs :

- **parent** : Référence au nœud parent dans l'arbre de jeu.
- **pièces_joueur** : Représente les pièces du joueur actuel (bitboard).
- **pièces_ennemi** : Représente les pièces de l'ennemi. (bitboard)
- **tour** : Indique à qui est le tour de jouer (-1 pour noir, 1 pour blanc).
- **taille** : La taille du plateau de jeu.
- **enfants** : Liste des nœuds enfants générés par expansion.
- **coups** : Liste des coups possibles à partir de ce nœud.
- **directions** : Dictionnaire associant à chaque coup les directions des pièces à capturer.
- **coups_vers_enfant** : Dictionnaire reliant les coups aux nœuds enfants correspondants.
- **valeur** : Valeur évaluative du nœud pour les algorithmes de recherche.
- **visité** : Booléen indiquant si le nœud a été visité lors de la recherche.

Méthodes :

- **développer()** : Génère tous les coups possibles et prépare les structures pour l'expansion.
- **définir_enfant(coup)** : Crée un nœud enfant à partir d'un coup spécifié et l'ajoute à la liste des enfants. Si le nœud enfant existe déjà, il est simplement retourné.

- **ajouter_autre_enfant(autre)** : Ajoute un autre nœud enfant qui n'est pas généré par un coup direct. Permet de transférer des nœuds d'un joueur à l'autre.
- **inverser()** : Échange les pièces du joueur et de l'ennemi et inverse le tour de jeu.
- **compter_tous_les_enfants()** : Compte récursivement le nombre total de nœuds enfants.

1.4 Boucle de jeu et séparation des connaissances

La boucle de jeu est le cœur de notre implémentation. Elle gère les interactions entre les différents composants du jeu, tels que les joueurs, les nœuds, les stratégies, et les paramètres de configuration. La boucle de jeu suit un processus itératif qui se déroule en plusieurs étapes, comme décrit ci-dessous :

1. Initialisation :

- Le jeu commence avec un plateau de 8x8 cases par défaut, équipé initialement de deux pièces noires et deux pièces blanches au centre.
- Les joueurs sont initialisés avec un système de bitboards qui gère les positions des pièces sur le plateau.
- Deux instances de joueurs (nœuds) sont créées, représentant le joueur actuel et son adversaire.

2. Déroulement du jeu :

- La boucle de jeu continue tant que des coups sont possibles.
- Pour chaque tour, le jeu vérifie si le joueur actuel a des coups disponibles. Si c'est le cas, il les génère en utilisant la méthode d'expansion du nœud.
- Si le joueur actuel ne peut pas jouer (aucun coup possible), le jeu vérifie si l'autre joueur peut jouer. Si aucun des deux joueurs ne peut jouer, le jeu se termine.

→ Séparation des connaissances : il y a duplication des nœuds, un pour chaque joueur. Pendant la phase de stratégie, seul le nœud du courant est utilisé.

3. Affichage :

- Si l'affichage est activé, le plateau est mis à jour visuellement après chaque coup pour montrer la progression du jeu.

4. Stratégie de jeu :

- Une stratégie détermine le coup suivant du joueur en cours en fonction des configurations définies.
- La stratégie peut intégrer différentes approches et niveaux de complexité selon les règles de jeu ou les paramètres d'IA spécifiés.
- Si le joueur actuel est un humain, le jeu attend un clic de souris sur un coup valide pour continuer.
- L'algorithme de stratégie retourne le prochain nœud.

5. Gestion de la mémoire et des statistiques :

- Après chaque mouvement, les enfants du nœud parent (qui ne sont pas le nœud actuel) sont nettoyés pour optimiser l'utilisation de la mémoire.
- Des statistiques de jeu peuvent être collectées et enregistrées, notamment le nombre de pièces jouées et le nombre de nœuds générés.

→ Préservation des connaissances : à chaque fin d’itération, lorsque les joueurs doivent être inversés, la méthode `ajouter_autre_enfant(autre)` compare les nœuds enfants des deux joueurs, et soit crée une copie du nœud de l’autre joueur, soit récupère le nœud enfant s’il existe déjà.

6. Fin du jeu :

- Le jeu se termine lorsque plus aucun coup n’est possible pour les deux joueurs. À ce stade, on détermine le vainqueur en comptant le nombre de pièces pour chaque joueur.
- Les résultats finaux, y compris les statistiques et l’état final du jeu, peuvent être enregistrés ou affichés selon les configurations.

2 Algorithmes

Le calcul des coups valides est une étape cruciale pour le jeu, c’est par ailleurs la plus coûteuse en temps de calcul. La représentation en bitboards nous permet de réaliser ces opérations de manière très efficace, comparable à une vectorisation. Il est en effet possible de calculer simultanément les coups valides qui sont dans la même direction, en utilisant des masques prédéfinis.

2.1 Décaler un bitboard

Pour réaliser cela, nous devons être capable de décaler un plateau entier dans les 8 directions cardinales. Nous y parvenons tel que suit :

Algorithm 1 Opérations de décalage pour les coups valides.

```

1: function NORD( $x$ )
2:   return  $x \gg 8$ 
3: end function
4: function SUD( $x$ )
5:   return  $(x \& 0x00ffffffffffff) \ll 8$ 
6: end function
7: function EST( $x$ )
8:   return  $(x \& 0x7f7f7f7f7f7f7f7f) \ll 1$ 
9: end function
10: function OUEST( $x$ )
11:   return  $(x \& 0xfefefefefefefefe) \gg 1$ 
12: end function
```

Les masques permettent d’éviter un débordement ou *overflow*, une sortie de plateau, et de préserver les bords. Ils consistent à mettre à 0 les bits qui sont des positions sensibles dans la direction donnée. Nous obtenons ensuite *Nord – Est*, *Nord – Ouest*, *Sud – Est*, *Sud – Ouest* en combinant les opérations précédentes. (Voir appendice 1 pour plus de détails).

2.2 Trouver les coups valides

Pour générer les coups valides à partir d’une position donnée, nous avons besoin de la position des pions du joueur, des pions de l’adversaire, et de la taille du plateau. L’algorithme 2 illustre la génération des coups valides pour un joueur donné, en utilisant les opérations de décalage définies précédemment.

Algorithm 2 Génération des Coups Valides avec Bitboards

```
1: function GÉNÉRERCOUPS(joueur, ennemi, taille)
2:   casesVides  $\leftarrow \sim (joueur \vee ennemi)$ 
3:   coupsUniques  $\leftarrow []$ 
4:   sautsDir  $\leftarrow \{\}$ 
5:   for all direction in [nord, sud, est, ouest, nord_ouest, ..., sud_est] do
6:     compteur  $\leftarrow 0$ 
7:     victimes  $\leftarrow direction(joueur) \& ennemi$ 
8:     if victimes = faux then
9:       continue
10:    end if
11:    for i  $\leftarrow 1$  to taille do
12:      compteur  $\leftarrow compteur + 1$ 
13:      prochainePiece  $\leftarrow direction(victimes) \& ennemi$ 
14:      if prochainePiece = faux then
15:        break
16:      end if
17:      victimes  $\leftarrow victimes \vee prochainePiece$ 
18:    end for
19:    captures  $\leftarrow direction(victimes) \& casesVides$ 
20:    while captures  $\neq 0$  do
21:      capture  $\leftarrow captures \& \sim captures$ 
22:      captures  $\leftarrow captures \oplus capture$ 
23:      if capture  $\notin sautsDir$  then
24:        Ajouter capture à coupsUniques
25:        sautsDir[capture]  $\leftarrow []$ 
26:      end if
27:      Ajouter (direction, compteur) à sautsDir[capture]
28:    end while
29:  end for
30:  return coupsUniques, sautsDir
31: end function
```

L'objectif de cette fonction est de remplir une liste de coups possible, mais également de stocker les informations nécessaires pour réaliser ces coups. À cette fin, nous remplissons dans une table de transposition² pour chaque coup trouvé, une liste de couples 'nombre de sauts' et 'direction'.

Nous pouvons décomposer l'algorithme 2 en plusieurs étapes :

1. Calculer les cases vides et initialiser les 2 structures à retourner (lignes 2-4).
2. Pour chaque direction, trouver les pions de l'adversaire qui sont des candidats pour être capturés (lignes 7-10). Un ensemble de pions est candidat si son intersection avec les pions adverses n'est pas vide.
3. Tant qu'au moins un pion est candidat, continuer de décaler dans la direction donnée, et ajouter les candidats à la liste des victimes, en mettant à jour la métrique du nombre de saut (lignes 11-18).
4. Vérifier si les candidats peuvent être capturés en vérifiant l'intersection avec la liste des cases vides (lignes 19).

2. un simple de dictionnaire en Python

À ce niveau, nous avons obtenu tous les coups valides, mais si nous voulons les jouer, nous devons les séparer. Cette opération, ainsi que les ajouts aux structures sont effectuées dans les lignes 20-29.

1. Tant qu'il existe des pions à capturer, nous les isolons un par un en faisant un 'ou exclusif' avec le LSB (lignes 21-22)
2. Le pion obtenu est ajouté à la liste des coups, et une liste vide est ajoutée à la table de transposition pour ce pion si aucun couple n'a été ajouté (lignes 23-25).
3. Pour chaque direction, le couple 'nombre de sauts' et 'direction' est ajouté à la liste de la table de transposition du pion capturé. (lignes 27).

2.3 Jouer un coup

Algorithm 3 Réalisation d'un Coup en Othello

```

1: function RÉALISERCOUP(joueur, ennemi, coupAJouer, directions)
2:   for all (direction, compte) in directions[coupAJouer] do
3:     victimes  $\leftarrow$  coupAJouer
4:     opposeeDirection  $\leftarrow$  directionsOpposees[direction]
5:     for i  $\leftarrow$  1 to compte do
6:       victimes  $\leftarrow$  victimes | (opposeeDirection(victimes) & ennemi)
7:     end for
8:     joueur  $\leftarrow$  joueur  $\oplus$  victimes
9:     ennemi  $\leftarrow$  ennemi  $\oplus$  (victimes &  $\sim$  coupAJouer)
10:  end for
11:  joueur  $\leftarrow$  joueur | coupAJouer
12:  return joueur, ennemi
13: end function
14:
15: directionsOpposees  $\leftarrow$  {nord : sud, sud : nord, est : ouest, ouest : est,
    nord_ouest : sud_est, nord_est : sud_ouest,
    sud_ouest : nord_est, sud_est : nord_ouest}
```

Nous initialisons d'abord les victimes avec le coup à jouer. Puis, nous déterminons la direction opposée à celle dans laquelle le coup a été effectué. Cette étape est cruciale car, au lieu de commencer par nos pierres et de se diriger vers une direction candidate, nous partons du coup joué et nous orientons dans la direction opposée afin de capturer les pièces adverses. Nous poursuivons ensuite par la récupération des pièces adverses pouvant être capturées dans cette direction, jusqu'à atteindre le nombre requis.

La mise à jour du plateau se fait ensuite via l'opérateur XOR :

1. Nous ajoutons le coup joué à nos propres pièces.
2. Nous retirons les pièces adverses capturables de l'ensemble des pièces de l'ennemi.

Étant donné que plusieurs directions sont envisageables pour un coup spécifique, l'opérateur XOR peut retourner une pièce plusieurs fois. Nous contournons cette problématique en employant l'opérateur OR vers la fin, pour garantir la présence du coup joué parmi nos pièces. Pour une raison analogue, nous excluons le coup joué des pierres capturées lors de l'application de l'opérateur XOR entre l'ennemi et les pièces capturables.

Voir appendice 2 pour les équivalents en Python.

2.4 Minimax et Alpha-Beta

Dans cette section, nous parcourons les différents algorithmes de recherche utilisés pour la prise de décision. Puisque les codes se ressemblent beaucoup, nous ne développerons que l'algorithme Negamax avec l'élagage Alpha-Beta. Toutes les versions de Minimax sont disponibles dans l'appendice 3.

Notre implémentation est inspirée de [Caz09], auquel nous avons ajouté notre structure basées sur les nœuds.

Algorithm 4 Algorithme Negamax avec élagage AlphaBeta

```

1: function      NEGAMAXALPHABETA(noeud, heuristique, profondeur_max,
   profondeur = 0, alpha = -MAX_INT, beta = MAX_INT, table = None)
2:   if profondeur = profondeur_max then
3:     noeud.valeur  $\leftarrow$  heuristique(noeud.pieces_joueur, noeud.pieces_ennemi,
   noeud.taille, table)
4:     return noeud
5:   end if
6:   if non noeud.visite then
7:     noeud.developpeur()
8:   end if
9:   if non noeud.coups then
10:    noeud.valeur  $\leftarrow$  heuristique(noeud.pieces_joueur, noeud.pieces_ennemi,
   noeud.taille, table)
11:    return noeud
12:  end if
13:  meilleur  $\leftarrow$  -MAX_INT
14:  indices  $\leftarrow$  []
15:  for i, coup in enumerate(noeud.coups) do
16:    if noeud.coups_vers_enfant[coup] = None then
17:      noeud.definir_enfant(coup)
18:    end if
19:    enfant  $\leftarrow$  noeud.coups_vers_enfant[coup]
20:    enfant.valeur  $\leftarrow$  -NegamaxAlphaBeta(enfant, heuristique, profondeur_max,
   profondeur + 1, -beta, -alpha, table).valeur
21:    if enfant.valeur > meilleur then
22:      meilleur  $\leftarrow$  enfant.valeur
23:      indices  $\leftarrow$  [i]
24:      if meilleur > alpha then
25:        alpha  $\leftarrow$  meilleur
26:        if alpha > beta then
27:          return noeud.enfants[random.choice(indices)]
28:        end if
29:      end if
30:    else if enfant.valeur = meilleur then
31:      indices.append(i)
32:    end if
33:  end for
34:  return noeud.enfants[random.choice(indices)]
35: end function

```

Voici les étapes clés de l'algorithme Negamax avec élagage AlphaBeta :

Étape 1 : Fin de la Récursion : La récursion se termine si la profondeur maximale est atteinte ou s'il n'y a plus de coups possibles à jouer.

Étape 2 : Génération des Coups : Les coups possibles sont générés uniquement si cela n'a pas déjà été fait pour le nœud courant. Cette étape évite la redondance et optimise la performance de l'algorithme.

Étape 3 : Parcours en Profondeur : L'algorithme explore en profondeur chaque coup possible, en simulant le coup et en passant au nœud enfant correspondant.

Étape 4 : Élagage AlphaBeta : Durant le parcours, l'algorithme utilise l'élagage AlphaBeta pour couper les branches de l'arbre de jeu qui ne peuvent pas affecter le résultat final. Cela permet de réduire considérablement l'espace de recherche.

→ Pendant l'élagage, nous gardons une trace des meilleurs coups rencontrés. Ainsi, nous nous pouvons choisir parmi plusieurs coups équivalents, et éviter le déterminisme.

Étape 5 : Stockage des Meilleurs Coups : L'algorithme garde une trace des meilleurs coups rencontrés. Si un coup a une valeur supérieure à l'actuelle meilleure valeur (alpha), cette valeur est mise à jour.

Étape 6 : Retour du Meilleur Coup : Une fois tous les coups explorés, l'algorithme retourne aléatoirement un nœud parmi les meilleurs trouvés.

2.5 Évaluateur de la qualité d'une position avec PyTorch

Nous avons également développé un évaluateur de la qualité d'une position, basé sur un modèle de deep learning. Nous avons utilisé PyTorch pour construire un réseau de neurones convolutif, entraîné sur la base de données WTHOR³. Nous avons ensuite utilisé ce modèle pour prédire le résultat d'une partie en cours, en évaluant la position actuelle du jeu.

L'entière pipeline permettant d'obtenir le modèle qui va suivre est disponible dans le notebook `model_pipeline.ipynb`.

Nous nous sommes inspirés de [LJK18] pour le développement de ce modèle.

Extraire les données de la WTHOR database

Les fichiers .wtb sont organisés comme suit :

- **En-tête de 16 octets** qui contient :
 - 1 octet pour le siècle de la création du fichier.
 - 1 octet pour l'année de la création du fichier.
 - 1 octet pour le mois et le jour de la création du fichier.
 - 4 octets pour le nombre de parties contenues dans le fichier (jusqu'à 2 147 483 648 parties).
 - 2 octets inutilisés dans ce contexte.
 - 1 octet pour l'année des jeux.
 - 1 octet pour la taille du plateau (0 ou 8 pour 8x8, 10 pour 10x10).
 - 1 octet pour le type de jeu (0 pour le jeu normal, 1 pour le "solitaire").
 - 1 octet pour la profondeur du jeu.
 - 1 octet réservé.

3. WTHOR est une base de données de parties d'Othello, disponible sur ffothello.org.

- **Informations de partie** pour chaque jeu incluant :
 - 2 octets pour l'étiquette du tournoi.
 - 2 octets pour l'identifiant du Joueur Noir.
 - 2 octets pour l'identifiant du Joueur Blanc.
 - 1 octet pour le score réel du Joueur Noir.
 - 1 octet pour le score théorique du Joueur Noir.
- **60 octets de mouvements** par partie, listant les coups joués.

Décodage des fichiers

1. Les fichiers sont lus par le biais d'une fonction génératrice qui parcourt un répertoire et extrait les informations de chaque fichier .wtb.
2. Pour chaque fichier, l'en-tête est lu pour vérifier la taille du plateau et le nombre de jeux.
3. Les informations de chaque partie sont ensuite lues suivies par la liste des mouvements.
4. Chaque mouvement est décodé de la représentation en octets à une représentation sur le plateau de jeu, utilisant les coordonnées (x, y).

Les données d'un fichier .wtb ne contiennent que les coups joués dans l'ordre, et non qui est le Joueur Noir ou blanc. Pour pouvoir obtenir ces informations et obtenir correctement chaque partie, il nous est donc nécessaire de la rejouer dans son intégralité à partir de chaque coup, et de déterminer par exemple les tours où certains joueurs n'ont pas pu jouer.

De ce processus, nous avons pu reconstruire 130,458 parties.

Prétraitement des données

Pour pouvoir entraîner notre modèle, il nous faut transformer les données extraites. Nous avons considéré pour l'entraînement, l'ensemble des positions de jeu, que nous étiquetons avec le coup suivant. Nous avons séparé cette base en deux, une pour le Joueur Noir, et une pour le Joueur Blanc. Ainsi, nous avons pu extraire plus de 6 millions positions de jeu, exactement 3,847,000 pour le Joueur Noir, et 3,682,000 pour le Joueur Blanc. Ces positions proviennent tous de parties respectivement gagnées par le premier joueur ou le second.

Nous notons que ce type de données implique une impossibilité d'obtenir 100%⁴ de précision, car il existe plusieurs coups possibles pour chaque position. Ce n'est pas un problème ici, car son application est vouée à l'évaluation de la qualité des coups possibles par exemple, en comparant les différentes probabilités de victoire de chaque coup suivant pour une position donnée. Nous retenons cependant que cela pourrait introduire un biais, et une difficulté pendant la phase d'apprentissage.

Différences notables avec [LJK18] :

- Il n'est pas commun de séparer les données de cette manière, mais nous faisons l'hypothèse que jouer en premier ou en deuxième devraient être traité différemment. Puisque ces deux derniers ne commencent pas avec la même configuration, il est possible que des stratégies optimales soient différentes.

4. [LJK18] ont estimé qu'un classifieur parfait pourrait obtenir 91.1% de précision au mieux

- Nous avons décidé de ne pas supprimer les duplicatas, car nous avons jugé que cela pourrait être une information utile pour le modèle. En effet, statistiquement, une position qui se répète dans les parties gagnantes devraient être plus considéré qu’une position rencontrée une unique fois.
- Il est usuel de considérer les symétries du plateau, mais nous avons manqué de temps pour implémenter cette fonctionnalité. Nous avons de toute façon déjà trop de données pour les ressources matérielles disponibles.

Création du modèle

Nous avons quelques idées intéressantes pour cette partie, telles que construire un noyau spécialisé (convolution) pour chaque direction, ou utiliser des réseaux de neurones récurrents pour prendre en compte l’historique des coups. Cependant, nous avons décidé de rester simple pour cette première itération, et avons construit un réseau de neurones convolutif classique, en essayant de reproduire les résultats de [LJK18] (ou de s’en approcher).

Ainsi, nous définissons un réseau de neurones convolutif à huit niveaux avec le résumé disponible dans le tableau 2.1.

TABLE 2.1 – Résumé du modèle pour une taille de batch de 256.

Layer (type)	Output Shape	Param #
Conv2d-1	[256, 64, 8, 8]	640
BatchNorm2d-2	[256, 64, 8, 8]	128
Conv2d-3	[256, 64, 8, 8]	36,928
BatchNorm2d-4	[256, 64, 8, 8]	128
Conv2d-5	[256, 128, 8, 8]	73,856
BatchNorm2d-6	[256, 128, 8, 8]	256
Conv2d-7	[256, 128, 8, 8]	147,584
BatchNorm2d-8	[256, 128, 8, 8]	256
Conv2d-9	[256, 256, 8, 8]	295,168
BatchNorm2d-10	[256, 256, 8, 8]	512
Conv2d-11	[256, 256, 8, 8]	590,080
BatchNorm2d-12	[256, 256, 8, 8]	512
Conv2d-13	[256, 256, 8, 8]	590,080
BatchNorm2d-14	[256, 256, 8, 8]	512
Conv2d-15	[256, 256, 8, 8]	590,080
BatchNorm2d-16	[256, 256, 8, 8]	512
Linear-17	[256, 128]	2,097,280
Linear-18	[256, 64]	8,256
Total params		4,432,768
Trainable params		4,432,768
Non-trainable params		0
Input size (MB)		0.06
Forward/backward pass size (MB)		352.38
Params size (MB)		16.91
Estimated Total Size (MB)		369.35

Entraînement du modèle

Nous avons entraîné le modèle comme nous avons pu sur Kaggle ⁵, en utilisant un GPU P100. Nous avons utilisé la fonction de perte **CrossEntropyLoss**, et l'optimiseur **Adam**. Nous avons entraîné le modèle pendant 100 époques, avec un taux d'apprentissage de 0.01, et avec Early Stopping. Nous avons utilisé une taille de batch de 256.

Plus de précision sont disponibles dans le notebook `model_pipeline.ipynb`.

Malheureusement, nous n'avons pas pu obtenir de réels résultats, dû au temps nécessaire pour l'entraînement ; lui-même dû à la très grande quantité de données à traiter. Kaggle ou Colab ⁶ ne fournissant pas de GPU suffisamment longtemps pour notre tâche, nous avons dû nous arrêter après quelques époques (en les répétant une à une puisque le notebook s'arrêtait à la fin de chaque époque).

5. <https://www.kaggle.com/>

6. <https://colab.research.google.com/>

Chapitre 3

Validation

Nous désirons mesurer deux aspects de notre programme : la qualité des stratégies décrites dans le sujet de TP et la complexité de chaque algorithme. Pour cela, nous avons mis en place un championnat qui oppose différentes configurations, ainsi que des mesures de complexité pour chaque algorithme. Les statistiques sont obtenues sur 100 itérations, sauf pour certaines profondeurs 6 pour des raisons de temps de calcul. Les résultats sont présentés dans les sections suivantes.

1 Mesure de complexité : Exploration de l'arbre de jeu

Nous comparons ici la complexité en temps et en nombre de nœuds explorés. Nous avons mesuré ces deux aspects sur 24 configurations. Elles correspondent à un affrontement entre soit Minimax, soit Alpha-Beta contre un joueur aléatoire avec une profondeur de recherche de 2, 4 et 6, sur différentes stratégies.

Temps de calcul

Nous avons mesuré le temps de calcul pour chaque partie jouée. L'unité des valeurs entre parenthèses est la seconde.

TABLE 3.1 – Analyse comparative du temps d'exécution d'une partie en fonction des profondeurs et stratégies (rapport NegamaxAlphaBeta(gauche)/Negamax(droite))

Depth	Strategy	Mean (%)	Std (%)
2	Positional	74.10 (0.03 vs 0.04)	198.71 (0.02 vs 0.01)
	Absolute	86.94 (0.01 vs 0.02)	208.26 (0.02 vs 0.01)
	Mobility	54.55 (0.06 vs 0.12)	83.37 (0.02 vs 0.03)
	Mixed (thresholds=[30, 55])	67.22 (0.06 vs 0.08)	140.54 (0.04 vs 0.03)
4	Positional	14.56 (0.90 vs 6.21)	14.38 (0.30 vs 2.06)
	Absolute	24.09 (0.51 vs 2.10)	16.92 (0.19 vs 1.15)
	Mobility	13.04 (1.50 vs 11.52)	11.27 (0.49 vs 4.38)
	Mixed (thresholds=[30, 55])	15.93 (1.17 vs 7.34)	12.67 (0.40 vs 3.18)
6	Positional	8.45 (35.20 vs 416.65)	2.60 (2.82 vs 108.49)
	Absolute	12.23 (20.14 vs 164.64)	18.39 (13.95 vs 75.87)
	Mobility	2.83 (39.32 vs 1390.49)	1.03 (7.65 vs 745.69)
	Mixed (thresholds=[30, 55])	3.17 (17.34 vs 546.98)	4.36 (5.04 vs 115.61)

Nombre de nœuds explorés

Nous avons mesuré à chaque coup des joueurs, le nombre de nœuds explorés. Nous pouvons donc comparer chaque algorithme sur un graphe pour chaque profondeur donnée¹. Nous avons également calculé le nombre moyen de nœuds explorés pour chaque configuration. Les résultats sont présentés dans les tableaux 3.2 et 3.3.

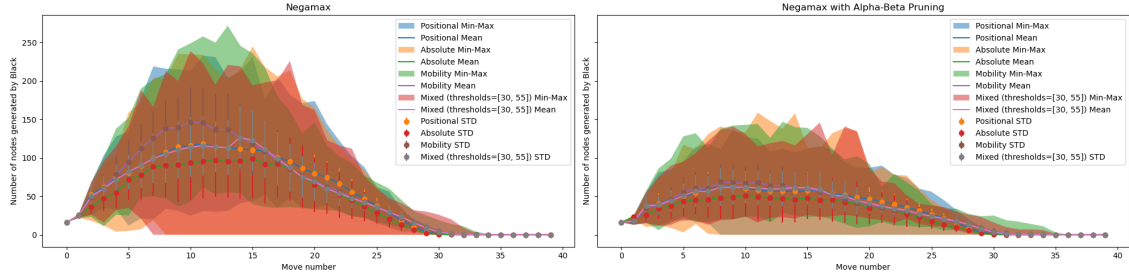


FIGURE 3.1 – Nombre de nœuds explorés par Minimax et Alpha-Beta en profondeur 2.

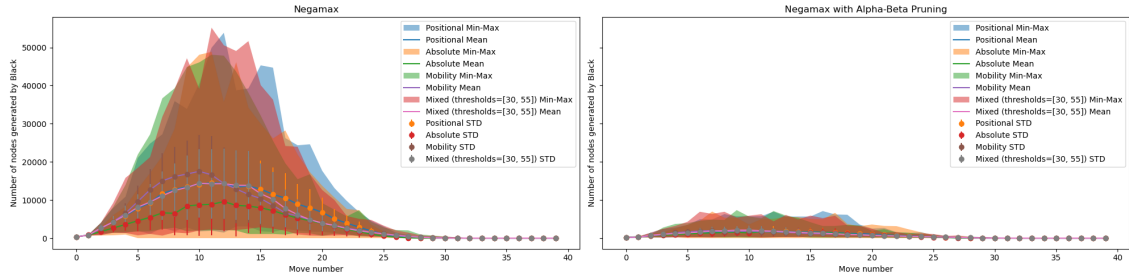


FIGURE 3.2 – Nombre de nœuds explorés par Minimax et Alpha-Beta en profondeur 4.

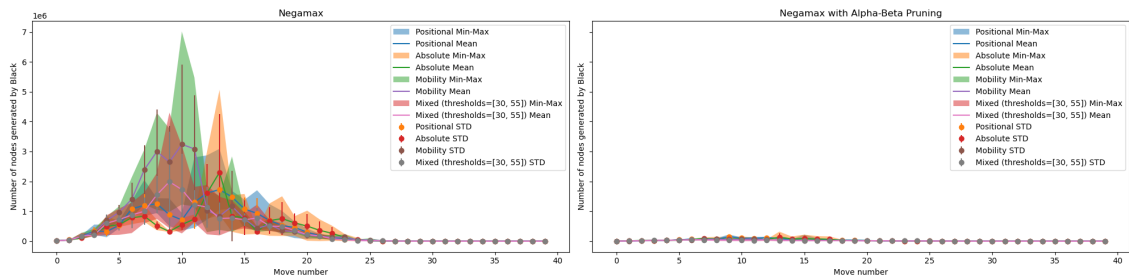


FIGURE 3.3 – Nombre de nœuds explorés par Minimax et Alpha-Beta en profondeur 6.

Formulons un récapitulatif, nous pouvons observer le nombre moyen de nœuds explorés par Minimax et Alpha-Beta pour chaque profondeur. Les résultats sont présentés dans le tableau 3.2. Pour les valeurs exactes, voir le tableau 3.3, avec pour unité 10^4 le nombre de nœuds explorés.

1. Chaque figure est disponible séparément dans les appendices 1.

TABLE 3.2 – Analyse comparative du nombre de nœuds explorés en fonction les profondeurs et les stratégies en pourcentage (rapport NegamaxAlphaBeta/Negamax)

Depth	Strategy	Max (%)	Mean (%)	Std (%)
2	Positional	58.80	57.52	65.20
	Absolute	64.90	55.79	63.92
	Mobility	54.04	53.95	60.26
	Mixed (thresholds=[30, 55])	61.09	58.35	62.72
4	Positional	13.19	14.77	16.35
	Absolute	14.27	21.19	17.92
	Mobility	15.29	14.06	14.30
	Mixed (thresholds=[30, 55])	12.54	15.76	15.56
6	Positional	7.15	6.96	6.66
	Absolute	6.26	7.67	10.21
	Mobility	1.93	2.98	2.56
	Mixed (thresholds=[30, 55])	1.84	3.07	2.77

TABLE 3.3 – Analyse comparative du nombre de nœuds explorés en fonction des profondeurs et stratégies (NegamaxAlphaBeta (gauche) vs Negamax(droite))

Depth	Strategy	Max (10^4)	Mean (10^4)	Std (10^4)
2	Positional	0.0137 vs 0.0233	0.003193 vs 0.005551	0.001091 vs 0.001673
	Absolute	0.0159 vs 0.0245	0.002579 vs 0.004622	0.001411 vs 0.002208
	Mobility	0.0147 vs 0.0272	0.003256 vs 0.006036	0.001102 vs 0.001830
	Mixed (thresholds=[30, 55])	0.0146 vs 0.0239	0.003178 vs 0.005445	0.001087 vs 0.001734
4	Positional	0.7103 vs 5.3861	0.078708 vs 0.532977	0.046724 vs 0.285685
	Absolute	0.6975 vs 4.8871	0.068316 vs 0.3224	0.046545 vs 0.259710
	Mobility	0.7352 vs 4.8096	0.073940 vs 0.525755	0.040591 vs 0.283905
	Mixed (thresholds=[30, 55])	0.6921 vs 5.5201	0.077103 vs 0.489311	0.044621 vs 0.286737
6	Positional	22.08 vs 309.04	2.96 vs 42.50	1.13 vs 16.93
	Absolute	31.68 vs 506.37	2.77 vs 36.06	1.98 vs 19.34
	Mobility	13.56 vs 701.55	1.79 vs 60.19	0.779 vs 30.44
	Mixed (thresholds=[30, 55])	7.90 vs 430.26	1.24 vs 40.39	0.582 vs 20.99

2 Championnat : Comparaison des configurations

Dans un premier temps, comparons les scores en fonctions de l'heuristique utilisée. Les tableaux correspondent à ceux donnés dans le cours. Les scores sont tous simplement le nombre de points obtenues en moyenne par chaque joueur. Nous avons mesuré les scores pour chaque configuration, et les résultats sont présentés dans les tableaux 3.4 (Point de Vue (PdV) du Joueur Noir) et 3.5 (PdV du Joueur Blanc).

Scores en fonction de l'heuristique et par couleur

TABLE 3.4 – Analyse comparative des scores pour le Joueur Noir en fonction des profondeurs et stratégies (rapport Table d'heuristique 2 (gauche) / Table d'heuristique 1 (droite) et valeurs exactes entre parenthèses)

Depth	Strategy	Mean (%)	Std (%)
2	Positional	119.81 (40.34 vs 33.67)	74.22 (8.31 vs 11.20)
	Mixed (thresholds=[30, 55])	111.36 (31.26 vs 28.07)	118.48 (10.94 vs 9.23)
4	Positional	123.77 (39.84 vs 32.19)	99.24 (9.33 vs 9.40)
	Mixed (thresholds=[30, 55])	102.79 (33.56 vs 32.65)	119.15 (11.94 vs 10.02)
6	Positional	146.67 (39.60 vs 27.00)	53.31 (6.41 vs 12.02)
	Mixed (thresholds=[30, 55])	116.36 (38.40 vs 33.00)	185.24 (12.56 vs 6.78)

TABLE 3.5 – Analyse comparative des scores pour le Joueur Blanc en fonction des profondeurs et stratégies (rapport Table d'heuristique 2 (gauche) / Table d'heuristique 1 (droite) et valeurs exactes entre parenthèses)

Depth	Strategy	Mean (%)	Std (%)
2	Positional	128.22 (38.85 vs 30.30)	83.51 (9.34 vs 11.19)
	Mixed (thresholds=[30, 55])	110.02 (31.28 vs 28.43)	115.42 (10.44 vs 9.05)
4	Positional	97.70 (31.06 vs 31.79)	199.38 (18.76 vs 9.41)
	Mixed (thresholds=[30, 55])	90.55 (29.52 vs 32.60)	218.67 (18.74 vs 8.57)
6	Positional	114.05 (42.20 vs 37.00)	76.89 (9.24 vs 12.02)
	Mixed (thresholds=[30, 55])	136.42 (47.20 vs 34.60)	140.46 (12.98 vs 9.24)

Scores globaux et par couleur

Nous voulons maintenant comparer les stratégies et l'avantage potentiel d'une couleur sur l'autre. Les résultats sont présentés dans les tableaux 3.6 et 3.7.

TABLE 3.6 – Analyse comparative des scores en fonction des profondeurs et stratégies pour toutes les configurations du Joueur Noir

Depth	Strategy	Min	Max	Mean	STD
2	Positional	0.00	58.00	35.46	9.63
	Absolute	0.00	62.00	24.29	10.10
	Mobility	0.00	62.00	26.97	12.16
	Mixed (thresholds=[30, 55])	0.00	62.00	34.09	10.99
4	Positional	0.00	59.00	32.70	9.04
	Absolute	0.00	61.00	22.44	10.04
	Mobility	0.00	63.00	30.71	11.98
	Mixed (thresholds=[30, 55])	0.00	64.00	34.94	11.65
6	Positional	0.00	52.00	29.57	10.61
	Absolute	2.00	59.00	26.33	8.54
	Mobility	0.00	61.00	30.35	12.61
	Mixed (thresholds=[30, 55])	11.00	63.00	37.98	9.91

TABLE 3.7 – Analyse comparative des scores en fonction des profondeurs et stratégies pour toutes les configurations du Joueur Blanc

Depth	Strategy	Min	Max	Mean	STD
2	Positional	0.00	58.00	34.95	9.87
	Absolute	0.00	58.00	24.77	9.96
	Mobility	0.00	62.00	29.33	11.07
	Mixed (thresholds=[30, 55])	0.00	62.00	33.52	11.56
4	Positional	0.00	57.00	31.91	11.55
	Absolute	0.00	63.00	23.34	10.26
	Mobility	0.00	64.00	27.86	11.19
	Mixed (thresholds=[30, 55])	0.00	62.00	34.29	12.56
6	Positional	7.00	59.00	32.97	8.77
	Absolute	0.00	58.00	21.99	12.14
	Mobility	0.00	60.00	29.56	11.58
	Mixed (thresholds=[30, 55])	0.00	60.00	34.70	11.15

Chapitre 4

Discussion

Dans ce chapitre, nous discutons des résultats obtenus dans le chapitre précédent. Nous commençons par expliciter les résultats de complexité temporelle et d'exploration de l'arbre de jeu. Ensuite, nous discutons des résultats de la performance des agents et l'analyse des parties.

1 Complexité et exploration de l'arbre de jeu

L'analyse comparative entre les algorithmes Negamax et Alpha-Beta montre des différences significatives dans leur performance à travers différentes stratégies et profondeurs de recherche.

1.1 Analyse temporelle

À la profondeur 2, l'algorithme Alpha-Beta surpasse déjà le Negamax traditionnel. Le moins bon résultat, la stratégie Positionnelle avec élagage représente 86.94% du temps de calcul de Negamax, indiquant une efficacité remarquable de l'Alpha-Beta dans la limitation de l'espace de recherche. Cette tendance est constante à travers les stratégies, comme la Mobilité et le Mixte, où les réductions de temps sont substantielles (54.55% et 67.22% du temps de Negamax respectivement). On remarque cependant que les Écart-types sont bien plus élevés (≈ 1.5 à 2 fois plus) pour l'Alpha-Beta que pour le Negamax. Cela suggère que l'Alpha-Beta est plus sensible à la configuration de l'arbre de jeu, et le gain à chaque itération peut varier considérablement. Il serait potentiellement possible de réduire cet écart s'il on parvenait à explorer les nœuds dans un ordre plus optimal, ce qui est une des améliorations possibles de l'algorithme Alpha-Beta.

En ce qui concerne la profondeur 4, les avantages de l'Alpha-Beta deviennent encore plus prononcés. Ici, les pourcentages de réduction du temps moyen de calcul sont impressionnants, allant jusqu'à un rapport de 13.03% pour la stratégie Mobilité. Cela suggère que l'effet de l'élagage Alpha-Beta est particulièrement bénéfique à des profondeurs plus grandes, où la complexité exponentielle du Negamax le rendrait autrement impraticable. De plus, les écarts-types sont devenues plus stables. Nous remarquons également que la stratégie Absolue est très coûteuse. Cela est probablement dû à la nature symétrique du plateau de jeu, qui implique que plusieurs coups entraînent le même score, ce qui rend une discrimination difficile pour l'algorithme. Nous pouvons faire l'hypothèse que pour Absolue, le nombre d'élagage est très inférieur à celui des autres stratégies.

Enfin, à la profondeur 6, les avantages de l'Alpha-Beta sur le Negamax traditionnel sont encore plus renforcés. Ceci est cohérent avec les observations précédentes : à mesure

que la profondeur augmente, les optimisations apportées par l'élagage Alpha-Beta sont essentielles pour maintenir une performance acceptable. Une remarque notable est l'écart-type de la stratégie Absolue, qui est particulièrement élevée. Une amélioration potentielle de l'algorithme serait de ne pas considérer les branches symétriques, ce qui réduirait le nombre de nœuds explorés et potentiellement le temps de calcul, ainsi que les variations de performance. Notre implémentation actuelle prend en compte les nœuds déjà explorés, nous pourrions donc améliorer l'algorithme en ne considérant le nœud seulement si son symétrique n'a pas déjà été exploré.

1.2 Exploration de l'arbre de jeu

Nos statistiques d'exploration des nœuds suivent globalement la même tendance que les temps de calcul.

Les graphiques illustrant le nombre de nœuds explorés par coup montrent clairement que l'Alpha-Beta est bien plus performant que le Negamax. Comme illustré dans les figures 3.1, 3.2, et 3.3, les aires sous les courbes pour l'Alpha-Beta sont nettement plus petites, indiquant moins de nœuds explorés par rapport à Negamax, et ce, à tous les niveaux de profondeur. Plus encore, l'amélioration est plus significative à mesure que la profondeur augmente, ce qui est conforme à la théorie sous-jacente de l'Alpha-Beta. En effet, la réduction théorique du nombre de nœuds explorés est de l'ordre de $\sqrt{b^d}$, où b est le facteur de branchement et d la profondeur de recherche. Nos mesures, quant à elles trouvent une réduction similaire d'environ $\frac{1}{c}\sqrt{b^d}$, avec $1 \leq c \leq 8$ pour la profondeur 6 seulement. Pour les profondeurs inférieures, la réduction est bien plus faible. Il est probable que si nous testions sur des Joueurs Aléatoires, la réduction serait plus proche de la théorie, car les heuristiques utilisées ont un impact sur la réduction effective du nombre de nœuds explorés.

Les tableaux 3.2 et 3.3 fournissent une vue d'ensemble quantitative de cette analyse. Les valeurs exactes des réductions des nœuds explorés, permettent une évaluation précise et détaillée des performances algorithmiques pour chaque stratégie. Notamment, la stratégie Mobilité à la profondeur 6 révèle la plus grande réduction proportionnelle, soulignant potentiellement une synergie entre l'élagage et cette stratégie en particulier qui a pour objectif d'optimiser la différence du nombre de coups possibles entre Joueur actuel et adverse.

Conclusions

Dans l'ensemble, ces résultats justifient grandement l'utilisation de l'élagage Alpha-Beta, en particulier pour des jeux avec un grand espace d'état comme c'est le cas pour de nombreux jeux de plateaux. L'efficacité accrue de l'Alpha-Beta se traduit par des temps de calcul plus rapides et moins de ressources nécessaires, ce qui est crucial dans des scénarios en temps réel ou des systèmes avec des contraintes de ressources.

C'est pourquoi, dans la partie suivante, nous nous concentrons sur les performances des agents, en utilisant l'Alpha-Beta comme algorithme de recherche.

2 Performance des agents et analyse des parties

Dans cette section, nous discutons des résultats obtenus lors du championnat inter-configuration, en mettant l'accent sur les performances du Joueur Noir et blanc à diverses profondeurs.

À noter, la stratégie Mixte utilise ici pendant les 30 premiers coups la stratégie Positionnelle, puis jusqu'à 50 la Mobilité, en finissant par la stratégie Absolue. Voyons pourquoi cette stratégie est la plus performante.

2.1 Performance des Heuristiques et par Couleur

Tel que nous l'observons dans les tableaux 3.4 et 3.5, les performances des heuristiques varient significativement de l'une à l'autre. En général, quelle que soit la couleur, la table d'heuristiques 2 offre un meilleur gain¹.

Nous remarquons par ailleurs que le gain augmente avec la profondeur (pour Positionnelle), commençant autour de 120%, puis 124% et finalement 147% pour les profondeurs 2, 4 et 6 respectivement pour le Joueur Noir. Cette corrélation nous fait supposer que la table d'heuristiques a été obtenue sur la base de parties du PdV d'un Joueur Noir, ou que les valeurs de la table d'heuristiques sont plus adaptées à ce dernier. Sinon, cela pourrait être dû à la nature du jeu, où le premier joueur aurait un avantage dans le cas où une bonne heuristique lui est fourni, ce qui lui permettrait alors de contrôler le plateau et de limiter les options de l'adversaire.

Également, nous remarquons que l'écart-type pour la stratégie Positionnelle est toujours bien plus faible que celle de Mixte. ce qui suppose qu'utiliser une stratégie dédiée avec heuristique est plus stable que celles sans connaissances préalables ou mixtes.

Cependant, il est difficile d'interpréter la variation de scores pour le Joueur Blanc. Plus de statistiques et une matrice de corrélation entre chaque paramètre pourraient être nécessaires pour comprendre les raisons de ces variations.

2.2 Comparaison des Stratégies et Avantages par Couleur

L'analyse des scores globaux nous révèle les tendances suivantes :

- Les stratégies les plus performantes sont de loin Mixte et Positionnelle, avec des gains par rapport aux autres entre +5 et +10 pions. Cela démontre la supériorité d'une stratégie avec connaissances préalables sur une sans heuristique.
- Absolue est la moins performante, avec des gains négatifs par rapport aux autres stratégies ; autour de -5 à 10 environ.
- Les Joueur Noir et blanc sont relativement plutôt équilibrés, avec des différences très inférieures à leur écart-type respectif. La couleur ne semble pas avoir un impact significatif sur l'issue de la victoire. Si l'on inclue un joueur aléatoire dans les statistiques, nous nous rendons compte que pour les matches le concernant, un avantage se profile pour le Joueur Blanc. Il est difficile de dire si cela est dû à la nature de ces heuristiques et stratégies en particulier, ou si cet avantage disparaît lorsque le niveau de jeu augmente.

1. sauf pour la profondeur 4 du Joueur Blanc qui est plus équilibré autour de 97.7% et 90.55% pour la stratégie Positionnelle et Mixte respectivement

Conclusions

En conclusion, les stratégies mixtes montrent un potentiel accru à des profondeurs d'analyse plus élevées, tandis que les stratégies positionnelles sont avantageuses pour le Joueur Noir à des profondeurs moins élevées. Mixte permet de combiner les points forts, en début de partie, il est intéressant de prendre des positions solides, d'obtenir des pions stables ; en milieu de partie, réussir à minimiser le nombre de coups possibles de l'adversaire et maximiser les siens permet de contrôler le déroulé de la partie ; finalement, le but du jeu étant de maximiser le nombre de pions, la stratégie Absolue est utilisée pour maximiser le score à la fin de la partie.

Chapitre 5

Conclusion

Ce projet a permis d'explorer en profondeur les mécanismes des algorithmes min-max et Alpha-Beta dans le contexte du jeu d'Othello, un environnement aux règles simples mais à la complexité combinatoire élevée. Nous avons développé et évalué plusieurs stratégies d'intelligence artificielle, notamment les approches positionnelle, absolue, mobilité et mixte, en les faisant jouer les unes contre les autres. Nos résultats montrent que l'algorithme Alpha-Beta, en particulier, offre des avantages significatifs en termes de réduction de la complexité temporelle et de l'espace de recherche, ce qui le rend préférable pour des profondeurs de recherche élevées.

L'analyse de la performance des différents agents a révélé que les stratégies mixtes, qui adaptent leur approche selon la phase du jeu, étaient particulièrement efficaces, combinant les avantages des approches positionnelles, de mobilité et absolues au fur et à mesure que la partie progresse. Cela souligne l'importance d'une adaptation stratégique flexible en réponse à l'évolution de l'état du jeu. Aussi, les stratégies positionnelles, prédominantes dans Mixte, ne peuvent pas être négligées, car elles sont elles-mêmes très efficaces. La connaissance préalable du jeu est donc un atout majeur pour l'IA.

Les défis rencontrés, notamment liés à la gestion de la complexité exponentielle et l'ordre d'exploration des nœuds, nous ont conduit à introduire plusieurs optimisations, telles que la représentation en BitBoard, un algorithme efficace de calcul des coups valides, et l'implémentation de l'élagage Alpha-Beta. Ces techniques ont permis de réduire considérablement le temps de calcul nécessaire pour explorer l'arbre de jeu, sans compromettre la qualité des décisions prises par l'IA.

D'autres améliorations futures sont envisageables, comme l'exploration de techniques de tri préalable des coups ou l'intégration de l'apprentissage par renforcement pour améliorer la capacité d'adaptation et d'apprentissage de l'IA. Il serait par ailleurs très intéressant de finir l'implémentation du modèle classifieur et de le comparer aux heuristiques existantes, ainsi qu'à l'état de l'art tel qu'Edax.

En conclusion, ce projet a non seulement renforcé notre compréhension des algorithmes de jeu à deux joueurs et de leurs applications pratiques mais a également ouvert la voie à de futures recherches sur l'optimisation des stratégies d'IA dans des jeux de complexité similaire. Nous avons pu démontrer que des avancées significatives sont possibles, même dans des contextes où les ressources de calcul sont limitées, par une utilisation judicieuse des techniques d'élagage et de l'analyse heuristique.

Sur un plan plus personnel, j'ai particulièrement aimé ce projet, et vais continuer à mettre à jour le dépôt GitHub associé. J'espère pouvoir continuer à faire plus de Théorie des Jeux et surtout plus d'IA, peut-être sur d'autres jeux de plateau. J'ai appris beaucoup de choses, et j'espère que ce rapport vous a plu. Merci de m'avoir lu.

Annexe A

Algorithmes et Code

1 Opérations Logiques

En 1.2, nous avons vu comment encoder en pseudo-code un pion et un plateau.

En Python :

```
1 def get_state(bitboard: int, x: int, y: int, size: int):
2     """Return the state of the cell by shifting the board
3     to the right by x * size + y and taking the LSB"""
4     return (bitboard >> (x * size + y)) & 1
```

FIGURE A.1 – Opérations logiques pour obtenir l'état du plateau.

```
1 def set_state(bitboard: int, x: int, y: int, size: int):
2     """Add a bit to the board by shifting a 1 to the left
3     by x * size + y and performing a bitwise OR with the board"""
4     return bitboard | (1 << (x * size + y))
```

FIGURE A.2 – Opérations logiques pour définir l'état du plateau.

En 2.1, nous avons vu comment décaler un bitboard dans 4 directions cardinales. Voyons maintenant comment décaler un bitboard dans les 4 directions diagonales, et leur équivalent en python.

Algorithm 5 Opérations de décalage en diagonales.

```
1: function NORDEST( $x$ )
2:   return  $Nord(Est(x))$ 
3: end function
4: function NORDOUEST( $x$ )
5:   return  $Nord(Ouest(x))$ 
6: end function
7: function SUDEST( $x$ )
8:   return  $Sud(Est(x))$ 
9: end function
10: function SUDOUEST( $x$ )
11:   return  $Sud(Ouest(x))$ 
12: end function
```

En Python :

```
1  def N(x):
2      return x >> 8
3  def S(x):
4      return (x & 0x00ffffffffffffff) << 8
5  def E(x):
6      return (x & 0x7f7f7f7f7f7f7f7f) << 1
7  def W(x):
8      return (x & 0xfefefefefefefefe) >> 1
```

```
1  def NW(x):
2      return N(W(x))
3  def NE(x):
4      return N(E(x))
5  def SW(x):
6      return S(W(x))
7  def SE(x):
8      return S(E(x))
```

FIGURE A.3 – Opérations de décalage pour les coups valides.

2 Trouver et jouer coups valides

En 2.2, nous avons vu comment trouver les coups valides et les jouer. Voyons maintenant en Python comment se traduit les pseudo-codes.

```
1 def generate_moves(own, enemy, size) -> tuple[list, dict]:
2     """Generate the possible moves for the current player using bitwise operations"""
3     empty = ~(own | enemy) # Empty squares (not owned by either player)
4     unique_moves = [] # List of possible moves
5     dir_jump = {} # Dictionary of moves and the number of pieces that can be captured in each direction
6
7     # Generate moves in all eight directions
8     for direction in [north, south, east, west, north_west, north_east, south_west, south_east]:
9         # We get the pieces that are next to an enemy piece in the direction
10        count = 0
11        victims = direction(own) & enemy
12        if not victims:
13            continue
14
15        # We keep getting the pieces that are next to an enemy piece in the direction
16        for _ in range(size):
17            count += 1
18            next_piece = direction(victims) & enemy
19            if not next_piece:
20                break
21            victims |= next_piece
22
23        # We get the pieces that can be captured in the direction
24        captures = direction(victims) & empty
25        # if there are multiple pieces in captures, we separate them and add them to the set
26        while captures:
27            capture = captures & -captures # get the least significant bit
28            captures ^= capture # remove the lsb
29            if capture not in dir_jump:
30                unique_moves.append(capture)
31                dir_jump[capture] = []
32            dir_jump[capture].append((direction, count))
33
34    return unique_moves, dir_jump
```

FIGURE A.4 – Trouver les coups valides.

```
1 def make_move(own: int, enemy: int, move_to_play: int, directions: dict) -> tuple[int, int]:
2     """Make the move and update the board using bitwise operations."""
3     for direction, count in directions[move_to_play]:
4         victims = move_to_play # Init the victims with the move to play
5
6         op_dir = opposite_dir(direction) # opposite direction since we
7         #go from the move to play to the captured pieces
8         for _ in range(count):
9             victims |= (op_dir(victims) & enemy)
10        own ^= victims
11        enemy ^= victims & ~move_to_play
12    # because of the XOR, the move to play which is considered a victim
13    # can be returned a pair number of times
14    own |= move_to_play
15    return own, enemy
```

FIGURE A.5 – Jouer un coup.

3 Minimax

En 2.4, nous avons vu comment implémenter l'algorithme Negamax avec AlphaBeta. Voyons maintenant en Python comment se traduit le pseudo-code, et les autres versions de l'algorithme.

```
1 def minimax(node: Node, heuristic: callable, max_depth: int, depth: int = 0, table=None) -> Node:
2     """MiniMax Algorithm"""
3     # End of the recursion : Max depth reached or no more possible moves
4     if depth == max_depth:
5         node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
6         return node
7
8     if not node.visited:
9         node.expand()
10    if not node.moves:
11        node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
12        return node
13
14    best = -MAX_INT if depth % 2 == 0 else MAX_INT
15    indices = []
16    for i, move in enumerate(node.moves):
17        if node.moves_to_child[move] is None:
18            node.set_child(move)
19            child = node.moves_to_child[move]
20            child.value = minimax(child, heuristic, max_depth, depth=depth + 1, table=table).value
21
22        if depth % 2 == 0:
23            if child.value > best:
24                best = child.value
25                indices = [i]
26            elif child.value == best:
27                indices.append(i)
28        else:
29            if child.value < best:
30                best = child.value
31                indices = [i]
32            elif child.value == best:
33                indices.append(i)
34    return node.children[random.choice(indices)]
```

FIGURE A.6 – Algorithme Minimax.

```

1  def minimax_alpha_beta(node: Node, heuristic: callable, max_depth: int, depth: int = 0,
2      alpha: int = -MAX_INT, beta: int = MAX_INT, table=None) -> Node:
3      """MinMax Algorithm with alpha-beta pruning"""
4      # End of the recursion : Max depth reached or no more possible moves
5      if depth == max_depth:
6          node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
7          return node
8
9      if not node.visited:
10         node.expand()
11     if not node.moves:
12         node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
13         return node
14
15     best = -MAX_INT if depth % 2 == 0 else MAX_INT
16     indices = []
17     for i, move in enumerate(node.moves):
18         if node.moves_to_child[move] is None:
19             node.set_child(move)
20         child = node.moves_to_child[move]
21         child.value = minimax_alpha_beta(child, heuristic, max_depth,
22             depth=depth + 1, alpha=alpha, beta=beta, table=table).value
23
24     # Update best node and best score
25     if child.value == best:
26         indices.append(i)
27     else:
28         if depth % 2 == 0:
29             if child.value > best:
30                 best = child.value
31                 indices = [i]
32             alpha = max(alpha, best) # Prune if possible
33             if alpha >= beta:
34                 return node.children[random.choice(indices)]
35         else:
36             if child.value < best:
37                 best = child.value
38                 indices = [i]
39             beta = min(beta, best) # Prune if possible
40             if alpha >= beta:
41                 return node.children[random.choice(indices)]
42     return node.children[random.choice(indices)]

```

FIGURE A.7 – Algorithme Minimax avec AlphaBeta.

```

1  def negamax(node: Node, heuristic: callable, max_depth: int, depth: int = 0, alpha: int = -MAX_INT,
2      beta: int = MAX_INT, table=None) -> Node:
3      """Negamax version of the MinMax Algorithm. Only works for pair depth."""
4      if depth == max_depth:
5          node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
6          return node
7
8      if not node.visited:
9          node.expand()
10     if not node.moves:
11         node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
12         return node
13
14     best = -MAX_INT
15     indices = []
16     for i, move in enumerate(node.moves):
17         if node.moves_to_child[move] is None:
18             node.set_child(move)
19             child = node.moves_to_child[move]
20             child.value = -negamax(child, heuristic, max_depth,
21                 depth=depth + 1, alpha=-beta, beta=-alpha, table=table).value
22
23             if child.value > best:
24                 best = child.value
25                 indices = [i]
26             elif child.value == best:
27                 indices.append(i)
28     return node.children[random.choice(indices)]

```

FIGURE A.8 – Algorithmme Negamax.

```

1  def negamax_alpha_beta(node: Node, heuristic: callable, max_depth: int, depth: int = 0,
2      alpha: int = -MAX_INT, beta: int = MAX_INT, table=None) -> Node:
3      """Negamax version of the MinMax Algorithm with alpha-beta pruning. Only works for pair depth."""
4      # End of the recursion : Max depth reached or no more possible moves
5      if depth == max_depth:
6          node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
7          return node
8
9      if not node.visited:
10         node.expand()
11     if not node.moves:
12         node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
13         return node
14
15     best = -MAX_INT
16     indexes = []
17     for i, move in enumerate(node.moves):
18         if node.moves_to_child[move] is None:
19             node.set_child(move)
20             child = node.moves_to_child[move]
21             child.value = -negamax_alpha_beta(child, heuristic, max_depth,
22                 depth=depth + 1, alpha=-beta, beta=-alpha, table=table).value
23
24         if child.value > best:
25             best = child.value
26             indexes = [i]
27             if best > alpha:
28                 alpha = best
29                 if alpha > beta:
30                     return node.children[random.choice(indexes)]
31         elif child.value == best:
32             indexes.append(i)
33     return node.children[random.choice(indexes)]
34

```

FIGURE A.9 – Algorithme Negamax avec AlphaBeta.

Annexe B

Résultats et statistiques

1 Graphiques individuels des nœuds explorés

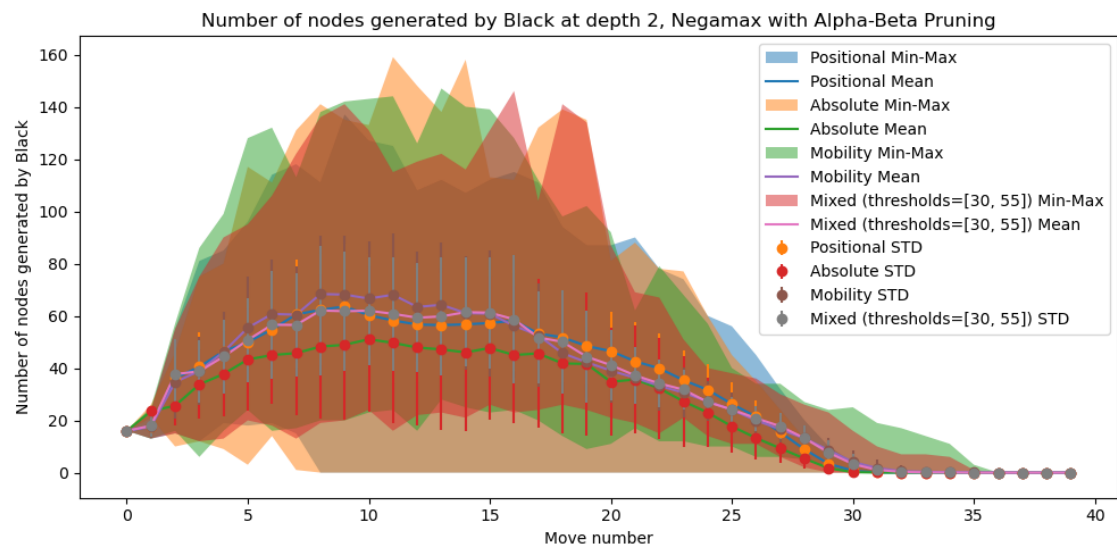


FIGURE B.1 – Nombre de nœuds explorés par coup pour la profondeur 2 avec Alpha-Beta.

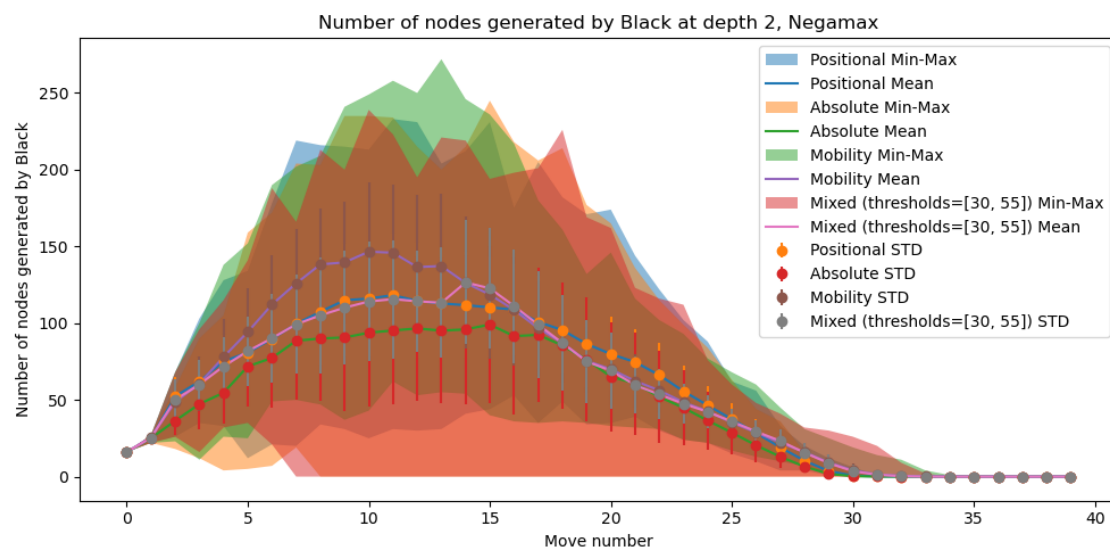


FIGURE B.2 – Nombre de nœuds explorés par coup pour la profondeur 2 sans Alpha-Beta.

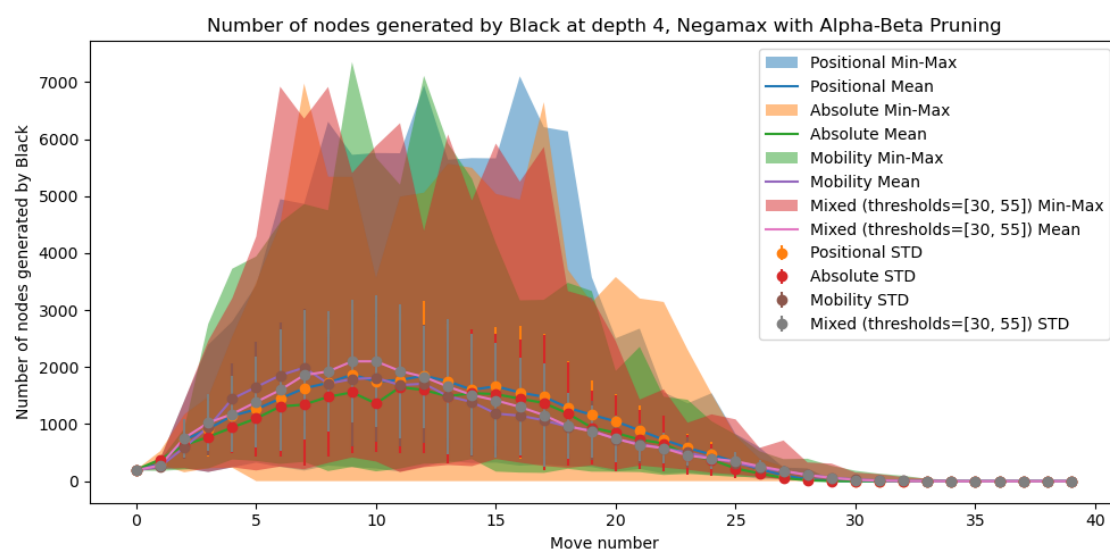


FIGURE B.3 – Nombre de nœuds explorés par coup pour la profondeur 4 avec Alpha-Beta.

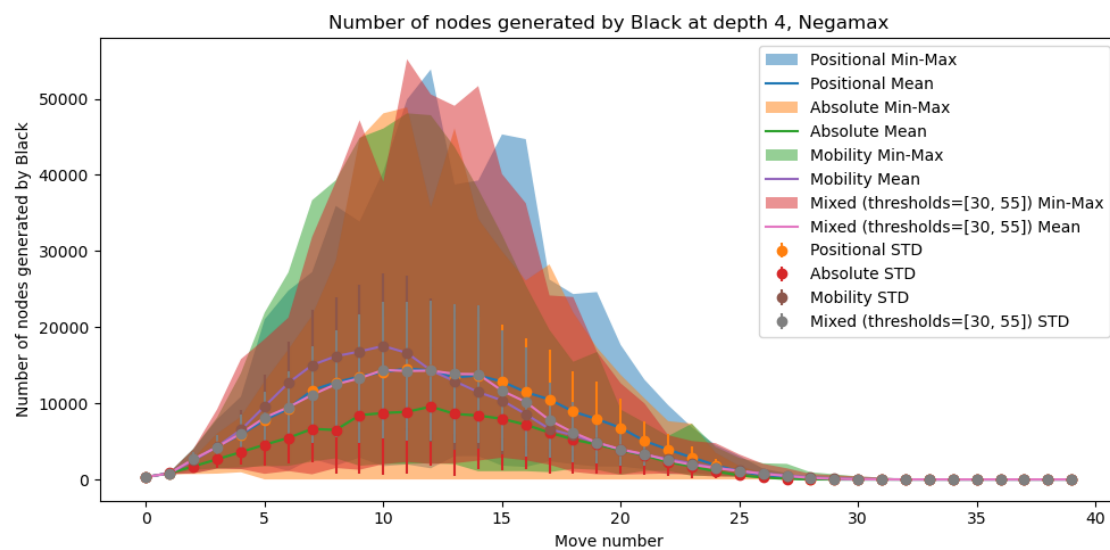


FIGURE B.4 – Nombre de nœuds explorés par coup pour la profondeur 4 sans Alpha-Beta.

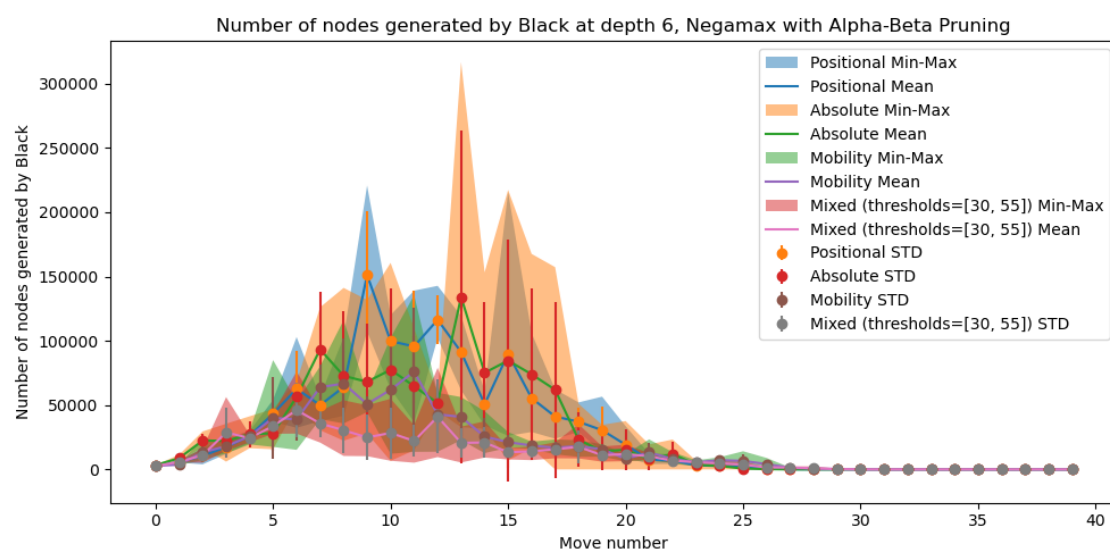


FIGURE B.5 – Nombre de nœuds explorés par coup pour la profondeur 6 avec Alpha-Beta.

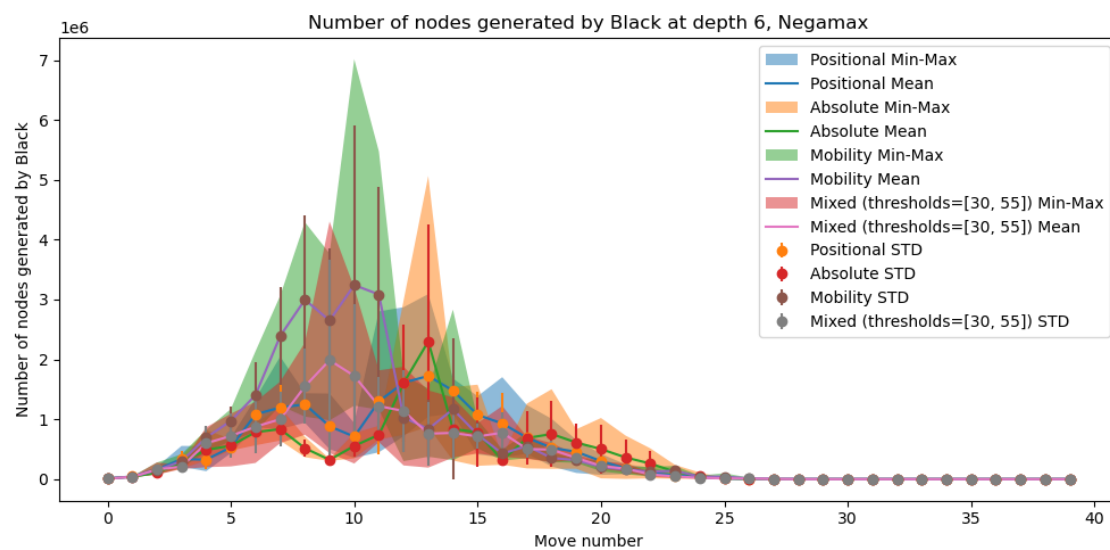


FIGURE B.6 – Nombre de nœuds explorés par coup pour la profondeur 6 sans Alpha-Beta.

Bibliographie

- [Ros05] Brian ROSE. *Othello : A Minute to Learn, A Lifetime to Master*. 2005. URL : <https://www.ffothello.org/livres/othello-book-Brian-Rose.pdf>.
- [Caz09] Tristan CAZENAVE. *Des Optimisations de l'Alpha-Beta*. 2009. URL : <https://www.lamsade.dauphine.fr/~cazenave/papers/berder00.pdf> (visité le 13/04/2024).
- [LJK18] Pawel LISKOWSKI, Wojciech JASKOWSKI et Krzysztof KRAWIEC. « Learning to Play Othello With Deep Neural Networks ». In : *IEEE Transactions on Games* 10.4 (déc. 2018), p. 354-364. ISSN : 2475-1510. DOI : 10.1109/tg.2018.2799997. URL : <http://dx.doi.org/10.1109/TG.2018.2799997>.