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

Liste des Acronymes

IA Intelligence Artificielle

MSB Bit de Poids Fort

LSB Bit de Poids Faible

TP Travaux Pratiques

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Dans le cadre de ce travail pratique, nous avons abordé la conception et l'implémentation d'une Intelligence Artificielle (IA) pour jouer au jeu d'Othello. Ce jeu de réflexion à deux joueurs sur un damier de 64 cases, avec des pions de deux couleurs, présente un défi intéressant pour l'IA en raison de la complexité de ses règles et de son espace de recherche combinatoire. L'objectif principal était de développer un algorithme capable de jouer contre un joueur humain ou contre une autre IA.

Nous explorerons dans un premier temps la modélisation du jeu et les structures de données utilisées pour représenter le plateau et les pions. Nous décrirons ensuite les algorithmes développés, en particulier l'algorithme Minimax, et plusieurs de ses variantes. Nous présenterons également une fonction d'évaluation pour mesurer la qualité des positions.

De plus, nous détaillerons la conception d'un classifieur pour évaluer la qualité d'une position. Nous expliquerons comment ce dernier a été entraîné et comment il se compare aux autres heuristiques.

Enfin, nous discuterons des résultats obtenus et des améliorations possibles.

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 PyTorch [1].

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 evaluation
└── 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 définies 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 :

```yaml
### Game Parameters ###
# Who plays against who.
# 0: human.
# 1: random (chooses randomly a possible move),
# 2: positional. (uses a heuristic table to define the quality of a position),
# 3: absolute (minimizes/maximizes the number of pieces for the opponent/player),
# 4: mobility (minimizes/maximizes the number of possible moves for the opponent/player),
# 5: mixed (uses a heuristic table to define the quality of a position).
# Mixed signifies using positional, then mobility, then absolute.
mode: [0, 2]

# Which MiniMax version to use.
# 0: NONE
# 1: Default MiniMax
# 2: Alpha-Beta pruning
# 3: Default Negamax
# 4: Alpha-Beta pruning Negamax
minimax_mode: [4, 4]

# Maximum depth of the search tree (minimax algorithm)
max_depth: [4, 4]

# Which heuristic table to use.
# 0: None
# 1: TABLE1
# 2: TABLE2
h_table: [0, 2]

```

```yaml
# Whether to display the board with a graphical interface
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 avions 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 $64 bits$. 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 notés 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 $e5$.

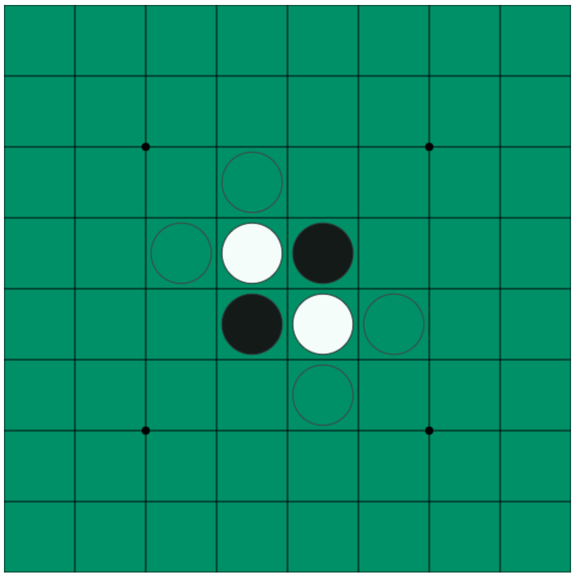

FIGURE 2.2 – Configuration initiale du plateau de jeu. Source : eOthello.

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 réécris en hexadécimal pour une meilleure lisibilité :

$$\begin{aligned} &0x00\_00\_00\_08\_10\_00\_00\_00 \quad (noir) \\ &0x00\_00\_00\_10\_08\_00\_00\_00 \quad (blanc) \end{aligned}$$

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

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

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 :

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

Avec $\ll$ l'opérateur de décalage à gauche, et | l'opérateur logique '`ou`'.

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

$$\text{pieces} = \text{noir} \quad | \quad \text{blanc}$$
$$\text{vides} = \quad \sim \text{pieces}$$

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 anaylyses formulés lors de tours précédants. 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 noeud 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 noeuds 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 dupplication des noeuds, un pour chaque joueur. Pendant la phase de statégie, seul le noeud 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 noeud.
5. **Gestion de la mémoire et des statistiques :**
   — Après chaque mouvement, les enfants du nœud précédent sont éventuellement 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 noeuds enfants des deux joueurs, et soit crée une copie du noeud de l'autre joueur, soit récupère le noeud enfant si 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 couteuse 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.

function NORD(x)
  return $x \gg 8$
end function
function SUD(x)
  return $(x \ \& \ 0x00ffffffffffffff) \ll 8$
end function
function EST(x)
  return $(x \ \& \ 0x7f7f7f7f7f7f7f7f) \ll 1$
end function
function OUEST(x)
  return $(x \ \& \ 0xfefefefefefefefe) \gg 1$
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 appendix 1 pour plus de détails).

2.2 Trouver les coups valides

Pour générer les coups valides à partir d'un 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 ←∼ (joueur ∨ ennemi)
3:     coupsUniques ← []
4:     sautsDir ← {}
5:     for all direction in [nord, sud, est, ouest, nord_ouest, . . . , sud_est] do
6:         compteur ← 0
7:         victimes ← direction(joueur) & ennemi
8:         if victimes = faux then
9:             continue
10:        end if
11:        for i ← 1 to taille do
12:            compteur ← compteur + 1
13:            prochainePiece ← direction(victimes) & ennemi
14:            if prochainePiece = faux then
15:                break
16:            end if
17:            victimes ← victimes ∨ prochainePiece
18:        end for
19:        captures ← direction(victimes) & casesVides
20:        while captures ≠ 0 do
21:            capture ← captures & − captures
22:            captures ← captures ⊕ capture
23:            if capture ∉ sautsDir then
24:                Ajouter capture à coupsUniques
25:                sautsDir[capture] ← []
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 transposition[2] 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 initiliser 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 effectués 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

```
function RéaliserCoup(joueur, ennemi, coupAJouer, directions)
    for all (direction, compte) in directions[coupAJouer] do
        victimes ← coupAJouer
        opposeeDirection ← directionsOpposees[direction]
        for i ← 1 to compte do
            victimes ← victimes | (opposeeDirection(victimes) & ennemi)
        end for
        joueur ← joueur ⊕ victimes
        ennemi ← ennemi ⊕ (victimes & ∼ coupAJouer)
    end for
    joueur ← joueur | coupAJouer
    return joueur, ennemi
end function

directionsOpposees ← {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 appendix 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 struture basées sur les noeuds.

Algorithm 4 Algorithme Negamax avec élagage AlphaBeta

```
function NegamaxAlphaBeta(noeud, heuristique, profondeur_max,
profondeur = 0, alpha = −MAX_INT, beta = MAX_INT, table = None)
    if profondeur = profondeur_max then
        noeud.valeur ← heuristique(noeud.pieces_joueur, noeud.pieces_ennemi,
                                    noeud.taille, table)
        return noeud
    end if
    if non noeud.visite then
        noeud.developper()
    end if
    if non noeud.coups then
        noeud.valeur ← heuristique(noeud.pieces_joueur, noeud.pieces_ennemi,
                                    noeud.taille, table)
        return noeud
    end if
    meilleur ← −MAX_INT
    indices ← []
    for i, coup in enumerate(noeud.coups) do
        if noeud.coups_vers_enfant[coup] = None then
            noeud.definir_enfant(coup)
        end if
        enfant ← noeud.coups_vers_enfant[coup]
        enfant.valeur ← −NegamaxAlphaBeta(enfant, heuristique, profondeur_max,
                                    profondeur + 1, −beta, −alpha, table).valeur
        if enfant.valeur > meilleur then
            meilleur ← enfant.valeur
            indices ← [i]
            if meilleur > alpha then
                alpha ← meilleur
                if alpha > beta then
                    return noeud.enfants[random.choice(indices)]
                end if
            end if
        else if enfant.valeur = meilleur then
            indices.append(i)
        end if
    end for
    return noeud.enfants[random.choice(indices)]
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 noeud parmi les meilleurs trouvé.

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 WTHOR [3]. 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 pourvoir 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 peu 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'entrainement, 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 impossiblité 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 obtimalles soient différentes.

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

— Nous avons décidé de ne pas supprimé 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 avions de toute façon déjà trop de données pour les ressources matérielles disponibles.

Création du modèle

Nous avions 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 suivant :

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 à la quantité immense des données, et au temps nécessaire pour l'entraînement. 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 un à une puisque le notebook s'arrêtait).

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 Travaux Pratiques (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.

Table 3.1 – NegamaxAlphaBeta/Negamax Analyse comparative du temps d'exécution d'une partie à travers les profondeurs et les stratégies en secondes

Depth	**Strategy**	**Mean**	**Std**
		Depth 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)
		Depth 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)
		Depth 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)

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

1.1 Nombre de nœuds explorés

Nous avons mesuré à chaque coup des joeurs, le nombre de nœuds explorés. Nous pouvons donc comparer chaque algorithme sur un graphe pour chaque profondeur donnée [1]. 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.

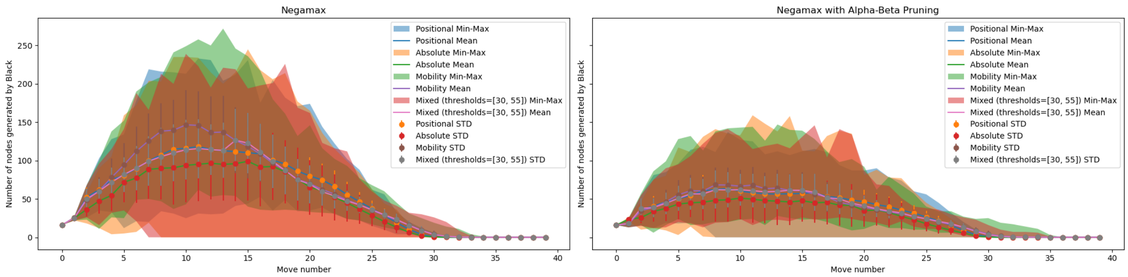

FIGURE 3.1 – Nombre de nœuds explorés par Minimax et Alpha-Beta en profondeur 2.

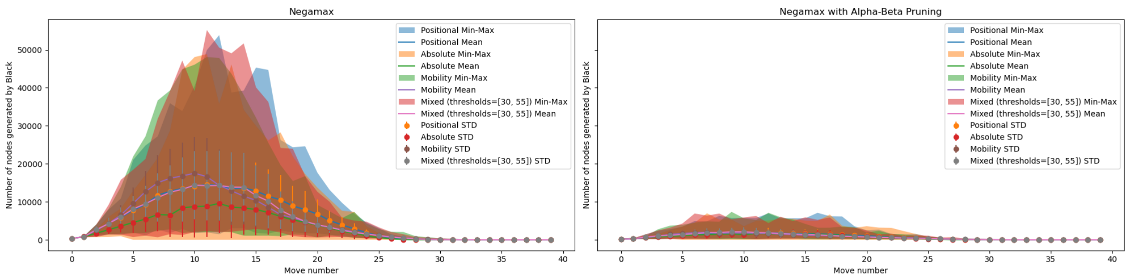

FIGURE 3.2 – Nombre de nœuds explorés par Minimax et Alpha-Beta en profondeur 4.

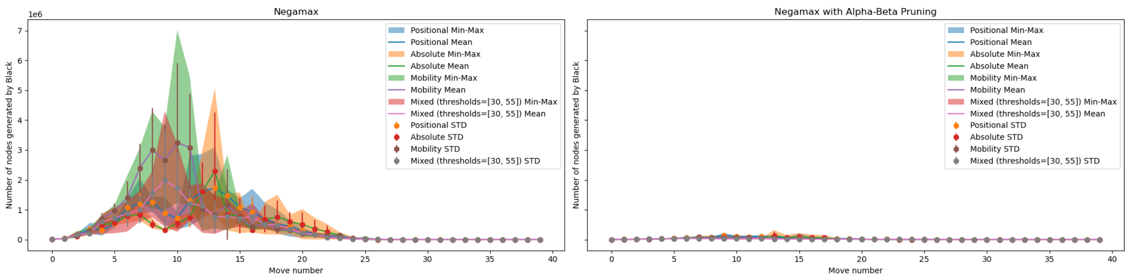

FIGURE 3.3 – Nombre de nœuds explorés par Minimax et Alpha-Beta en profondeur 6.

Dans un tableau 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 4.

TABLE 3.2 – NegamaxAlphaBeta (gauche)/Negamax(droite) Analyse comparative du nombre de nœuds explorés à travers les profondeurs et les stratégies en pourcentage

Depth	Strategy	Max of Maxs	Mean of Means	Mean of Stds
		Depth 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%
		Depth 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%
		Depth 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 – NegamaxAlphaBeta (gauche)/Negamax(droite) Analyse comparative du nombre de nœuds explorés à travers les profondeurs et les stratégies en valeur exacte

Depth	Strategy	Max of Maxs	Mean of Means	Mean of Stds
		Depth 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
		Depth 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
		Depth 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 algorithmes

Chapitre 4

Discussion

Dans ce chapitre, nous discutons des résultats obtenus dans le chapitre précédent. Nous commençons par discuter des 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 temporelle 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.

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 le Mobilité et le Mixe, 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 ($\approx$ 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 noeuds 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 couteuse. Cela est problement dû à la nature symétrique du plateau de jeu, qui implique que plusieurs coups entrainent 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 très élevé. 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 implématation actuelle prend en compte les noeuds déjà explorés, nous pourrions donc améliorer l'algorithme en ne considérant le noeud seulement si son symétrique n'a pas déjà été exploré.
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 $c \leq 8$ pour la profondeur 6 seulement. Pour les profondeurs inférieur, la réduction est bien plus faible, ce qui est attendu, puisqu'à ces niveaux, l'efficacité de l'élagage est moindre.

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.

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

Chapitre 5

Conclusion

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 :

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

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

```python
def set_state(bitboard: int, x: int, y: int, size: int):
    """Add a bit to the board by shifting a 1 to the left
    by x * size + y and performing a bitwise OR with the board"""
    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.

function NordEst(x)
    return $Nord(Est(x))$
end function
function NordOuest(x)
    return $Nord(Ouest(x))$
end function
function SudEst(x)
    return $Sud(Est(x))$
end function
function SudOuest(x)
    return $Sud(Ouest(x))$
end function

En Python :

```python
def N(x):
    return x >> 8
def S(x):
    return (x & 0x00ffffffffffffff) << 8
def E(x):
    return (x & 0x7f7f7f7f7f7f7f7f) << 1
def W(x):
    return (x & 0xfefefefefefefefe) >> 1
```

```python
def NW(x):
    return N(W(x))
def NE(x):
    return N(E(x))
def SW(x):
    return S(W(x))
def SE(x):
    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.

```python
def generate_moves(own, enemy, size) -> tuple[list, dict]:
    """Generate the possible moves for the current player using bitwise operations"""
    empty = ~(own | enemy)  # Empty squares (not owned by either player)
    unique_moves = []  # List of possible moves
    dir_jump = {}  # Dictionary of moves and the number of pieces that can be captured in each direction

    # Generate moves in all eight directions
    for direction in [north, south, east, west, north_west, north_east, south_west, south_east]:
        # We get the pieces that are next to an enemy piece in the direction
        count = 0
        victims = direction(own) & enemy
        if not victims:
            continue

        # We keep getting the pieces that are next to an enemy piece in the direction
        for _ in range(size):
            count += 1
            next_piece = direction(victims) & enemy
            if not next_piece:
                break
            victims |= next_piece

        # We get the pieces that can be captured in the direction
        captures = direction(victims) & empty
        # if there are multiple pieces in captures, we separate them and add them to the set
        while captures:
            capture = captures & -captures  # get the least significant bit
            captures ^= capture  # remove the lsb
            if capture not in dir_jump:
                unique_moves.append(capture)
                dir_jump[capture] = []
            dir_jump[capture].append((direction, count))

    return unique_moves, dir_jump
```

Figure A.4 – Trouver les coups valides.

```python
def make_move(own: int, enemy: int, move_to_play: int, directions: dict) -> tuple[int, int]:
    """Make the move and update the board using bitwise operations."""
    for direction, count in directions[move_to_play]:
        victims = move_to_play  # Init the victims with the move to play

        op_dir = opposite_dir(direction)  # opposite direction since we
        #go from the move to play to the captured pieces
        for _ in range(count):
            victims |= (op_dir(victims) & enemy)
        own ^= victims
        enemy ^= victims & ~move_to_play
    # because of the XOR, the move to play which is considered a victim
    # can be returned a pair number of times
    own |= move_to_play
    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.

```python
def minimax(node: Node, heuristic: callable, max_depth: int, depth: int = 0, table=None) -> Node:
    """MiniMax Algorithm"""
    # End of the recursion : Max depth reached or no more possible moves
    if depth == max_depth:
        node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
        return node

    if not node.visited:
        node.expand()
    if not node.moves:
        node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
        return node

    best = -MAX_INT if depth % 2 == 0 else MAX_INT
    indices = []
    for i, move in enumerate(node.moves):
        if node.moves_to_child[move] is None:
            node.set_child(move)
        child = node.moves_to_child[move]
        child.value = minimax(child, heuristic, max_depth, depth=depth + 1, table=table).value

        if depth % 2 == 0:
            if child.value > best:
                best = child.value
                indices = [i]
            elif child.value == best:
                indices.append(i)
        else:
            if child.value < best:
                best = child.value
                indices = [i]
            elif child.value == best:
                indices.append(i)
    return node.children[random.choice(indices)]
```

FIGURE A.6 – Algorithme Minimax.

```python
def minimax_alpha_beta(node: Node, heuristic: callable, max_depth: int, depth: int = 0,
                       alpha: int = -MAX_INT, beta: int = MAX_INT, table=None) -> Node:
    """MinMax Algorithm with alpha-beta pruning"""
    # End of the recursion : Max depth reached or no more possible moves
    if depth == max_depth:
        node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
        return node

    if not node.visited:
        node.expand()
    if not node.moves:
        node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
        return node

    best = -MAX_INT if depth % 2 == 0 else MAX_INT
    indices = []
    for i, move in enumerate(node.moves):
        if node.moves_to_child[move] is None:
            node.set_child(move)
        child = node.moves_to_child[move]
        child.value = minimax_alpha_beta(child, heuristic, max_depth,
                                         depth=depth + 1, alpha=alpha, beta=beta, table=table).value

        # Update best node and best score
        if child.value == best:
            indices.append(i)
        else:
            if depth % 2 == 0:
                if child.value > best:
                    best = child.value
                    indices = [i]
                alpha = max(alpha, best)  # Prune if possible
                if alpha >= beta:
                    return node.children[random.choice(indices)]
            else:
                if child.value < best:
                    best = child.value
                    indices = [i]
                beta = min(beta, best)  # Prune if possible
                if alpha >= beta:
                    return node.children[random.choice(indices)]
    return node.children[random.choice(indices)]
```

Figure A.7 – Algorithme Minimax avec AlphaBeta.

```python
def negamax(node: Node, heuristic: callable, max_depth: int, depth: int = 0, alpha: int = -MAX_INT,
            beta: int = MAX_INT, table=None) -> Node:
    """Negamax version of the MinMax Algorithm. Only works for pair depth."""
    if depth == max_depth:
        node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
        return node

    if not node.visited:
        node.expand()
    if not node.moves:
        node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
        return node

    best = -MAX_INT
    indices = []
    for i, move in enumerate(node.moves):
        if node.moves_to_child[move] is None:
            node.set_child(move)
        child = node.moves_to_child[move]
        child.value = -negamax(child, heuristic, max_depth,
                               depth=depth + 1, alpha=-beta, beta=-alpha, table=table).value

        if child.value > best:
            best = child.value
            indices = [i]
        elif child.value == best:
            indices.append(i)
    return node.children[random.choice(indices)]
```

Figure A.8 – Algorithme Negamax.

```python
def negamax_alpha_beta(node: Node, heuristic: callable, max_depth: int, depth: int = 0,
                       alpha: int = -MAX_INT, beta: int = MAX_INT, table=None) -> Node:
    """Negamax version of the MinMax Algorithm with alpha-beta pruning. Only works for pair depth."""
    # End of the recursion : Max depth reached or no more possible moves
    if depth == max_depth:
        node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
        return node

    if not node.visited:
        node.expand()
    if not node.moves:
        node.value = heuristic(node.own_pieces, node.enemy_pieces, node.size, table)
        return node

    best = -MAX_INT
    indexes = []
    for i, move in enumerate(node.moves):
        if node.moves_to_child[move] is None:
            node.set_child(move)
        child = node.moves_to_child[move]
        child.value = -negamax_alpha_beta(child, heuristic, max_depth,
                                          depth=depth + 1, alpha=-beta, beta=-alpha, table=table).value

        if child.value > best:
            best = child.value
            indexes = [i]
            if best > alpha:
                alpha = best
                if alpha > beta:
                    return node.children[random.choice(indexes)]
        elif child.value == best:
            indexes.append(i)
    return node.children[random.choice(indexes)]
```

Figure A.9 – Algorithme Negamax avec AlphaBeta.

4 Graphiques des noeuds explorés

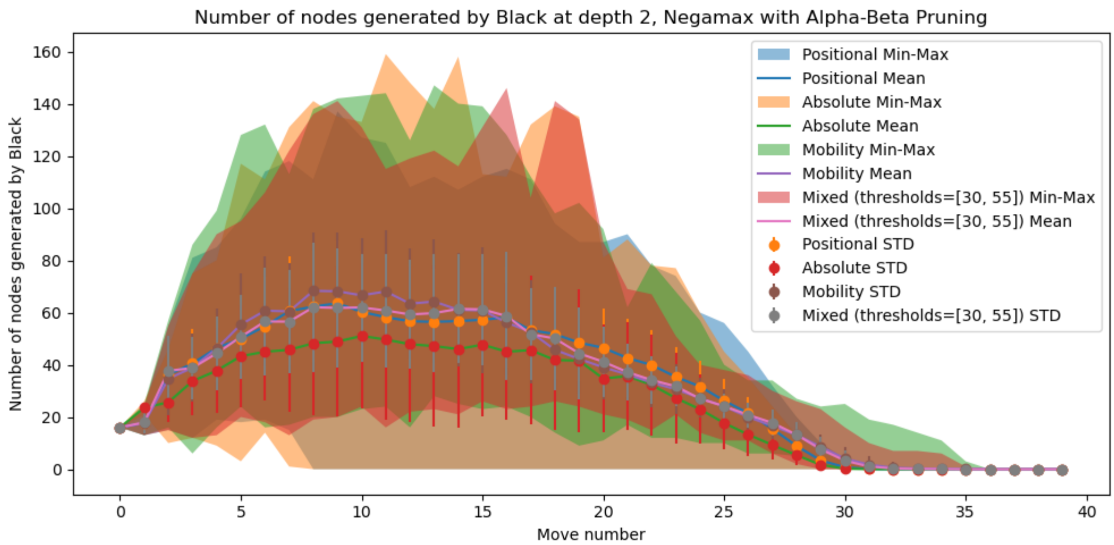

Figure A.10 – Nombre de noeuds explorés par coup pour la profondeur 2 avec Alpha-Beta.

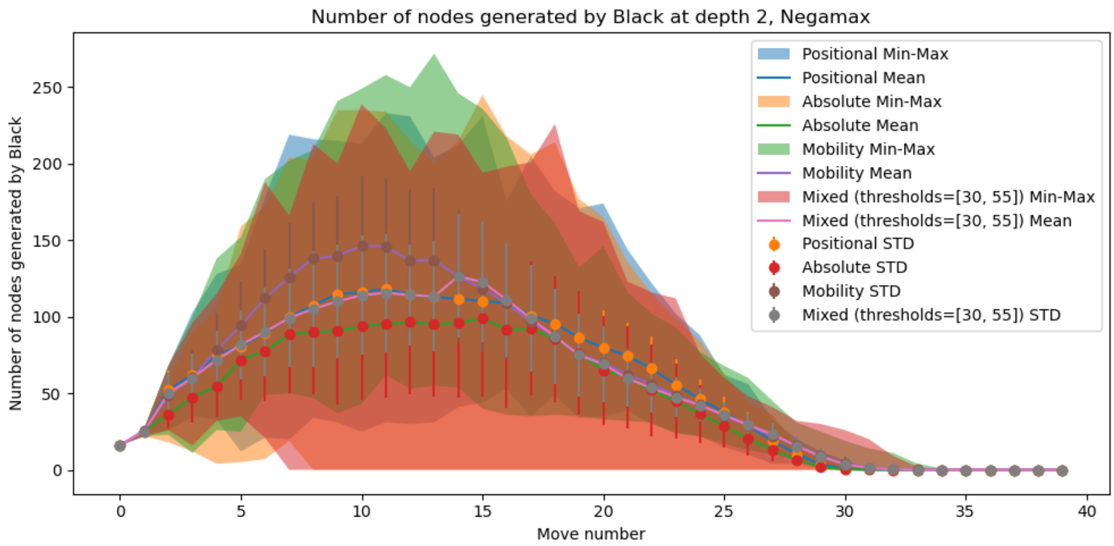

Figure A.11 – Nombre de noeuds explorés par coup pour la profondeur 2 sans Alpha-Beta.

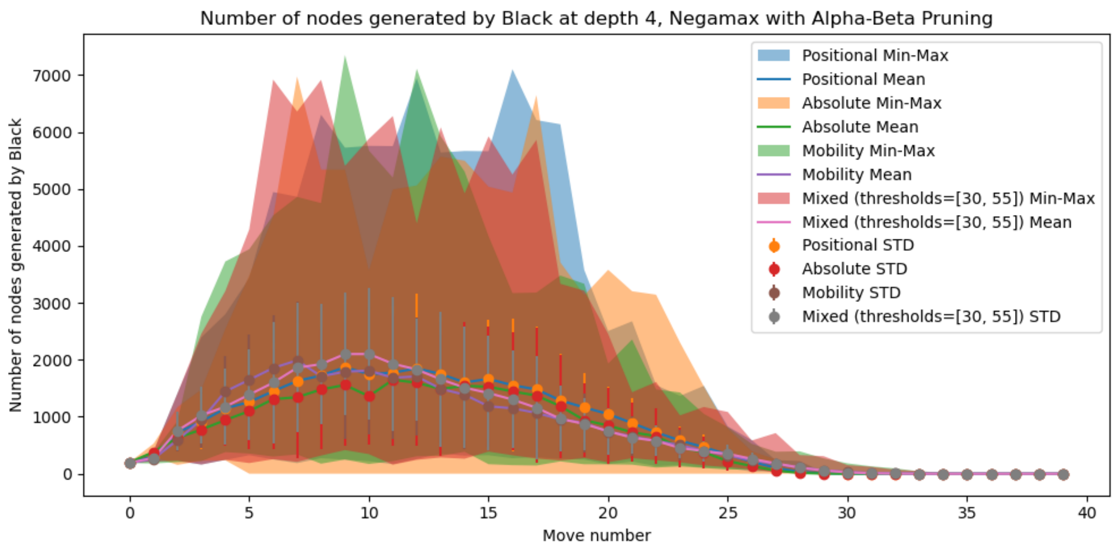

Figure A.12 – Nombre de noeuds explorés par coup pour la profondeur 4 avec Alpha-Beta.

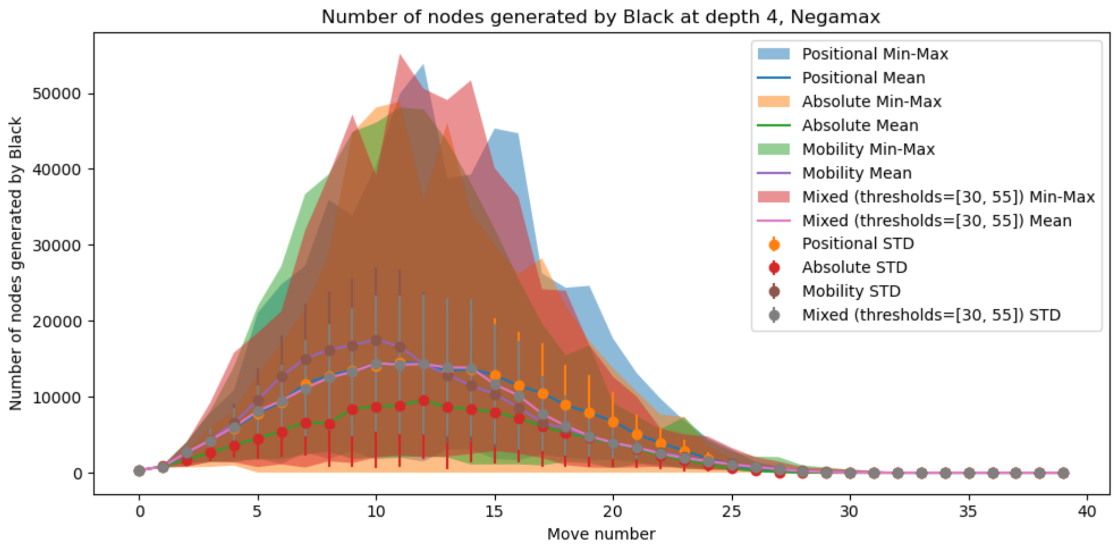

FIGURE A.13 – Nombre de noeuds explorés par coup pour la profondeur 4 sans Alpha-Beta.

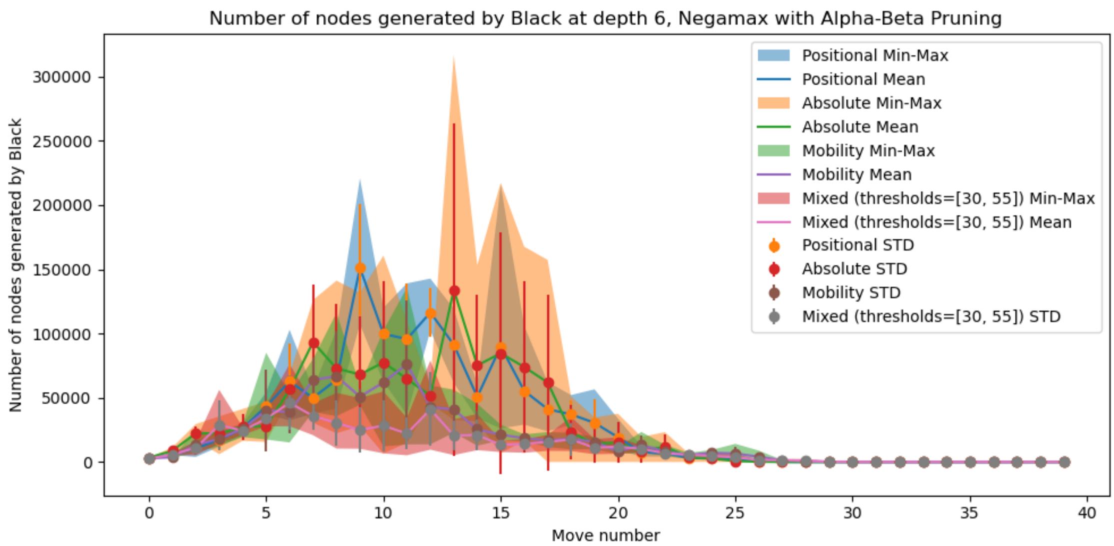

FIGURE A.14 – Nombre de noeuds explorés par coup pour la profondeur 6 avec Alpha-Beta.

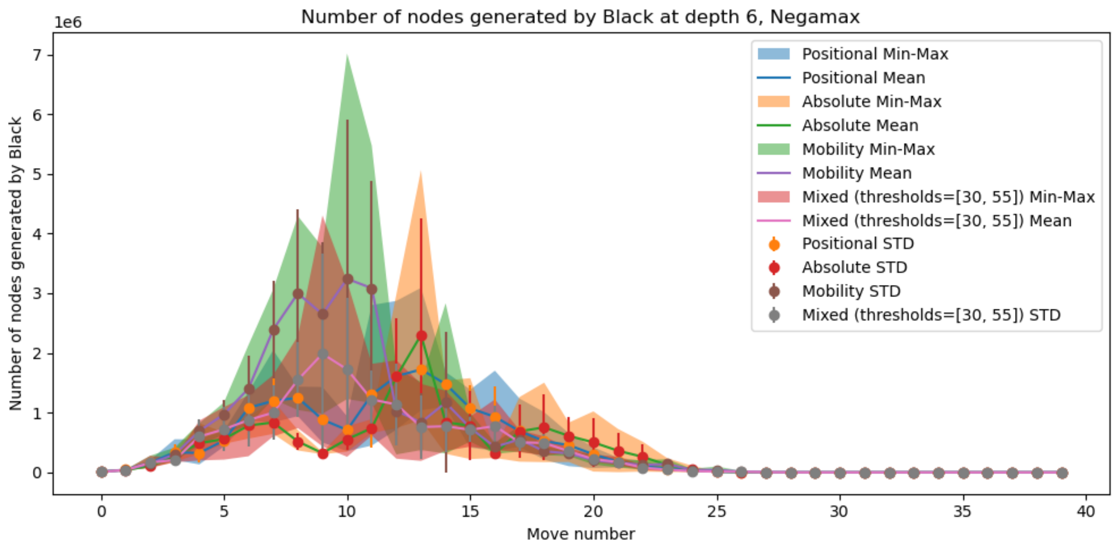

FIGURE A.15 – Nombre de noeuds explorés par coup pour la profondeur 6 sans Alpha-Beta.

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.