

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	13
4	Discussion	14
5	Conclusion	15
A	Algorithmes et Code	I
1	Opérations Logiques	I
2	Trouver et jouer coups valides	III
3	Minimax	IV

Liste des Acronymes

IA Intelligence Artificielle

MSB Bit de Poids Fort

LSB Bit de Poids Faible

Table des figures

2.1	Exemple d'une configuration (non complète).	3
2.2	Configuration initiale du plateau de jeu. Source : eOthello.	4
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 Negamax avec AlphaBeta.	IV

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 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 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 :

```
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 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*.

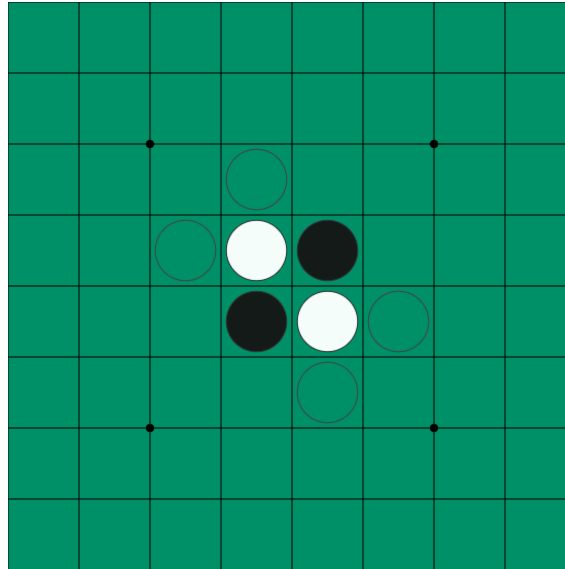


FIGURE 2.2 – Configuration initiale du plateau de jeu. Source : eOthello.

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 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 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 duplication des noeuds, un pour chaque joueur. Pendant la phase de stratégie, 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.

```

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 appendix 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 &  $\neg$  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 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

```

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 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 structure basées sur les noeuds.

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 implémenté 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`.

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).

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.

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.

Création du modèle

Entraînement du modèle

Chapitre 3

Validation

Chapitre 4

Discussion

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 :

```
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 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.6 – Algorithme Negamax avec AlphaBeta.

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).