Rapport de TP, Fondement IA: Analyse de différents algorithmes d'IA appliqués au jeu de plateau Othello

Eliott Georges, Izaak Aubert–Mécibah

4A ICY - 2024

Configuration matérielle utilisée pour le TP :

Les tests ont été réalisés avec un processeur AMD ryzen 7 7700x
Le programme python de l'Othello utilise un seul cœur du processeur, cadencé à 5.3GHz

Table des matières

Première partie

Introduction

Chapitre 1

Contexte

Cette série de Travaux Pratiques de l'enseignement **fondement de l'Intelligence Artificielle (IA)**, dirigée par **René Mandiau**, vise à approfondir les connaissances théoriques acquises lors des cours magistraux et des travaux dirigés. Elle se concentre sur plusieurs objectifs clés :

- Compréhension des enjeux de l'IA : Les étudiants apprendront l'importance et l'impact potentiel de l'IA dans divers domaines, soulignant l'importance de cette technologie dans les innovations contemporaines et futures.
- Décision et Action : Ce volet se concentre sur les principes et modèles de recherche en IA, incluant la recherche non informée et informée. Il aborde également la résolution de problèmes à travers des graphes d'états et des algorithmes comme A*, permettant aux étudiants de comprendre comment aborder et résoudre efficacement des problèmes complexes.
- Représentation des Connaissances et Raisonnement : Les participants apprendront à représenter et à raisonner sur des connaissances, une compétence cruciale pour le développement d'applications intelligentes. Cette partie met l'accent sur la manière dont l'IA peut modéliser le monde et prendre des décisions basées sur ces modélisations.

Au terme de cette Unité d'Enseignement (UE), nous auront acquis une compréhension des modèles existants en IA. Nous serons capables d'identifier si un problème donné peut et doit être résolu par une approche IA. En outre, nous aurons les compétences nécessaires pour modéliser et développer une application ciblée utilisant des approches et des outils IA existants, leur permettant de mettre en pratique les connaissances théoriques acquises dans des contextes réels et variés.

Chapitre 2

Othello

L'Othello se joue sur un tablier unicolore de 64 cases, 8 sur 8, appelé othellier. Les joueurs disposent de 64 pions bicolores, noirs d'un côté et blancs de l'autre. En début de partie, quatre pions sont déjà placés au centre de l'othellier : deux noirs, en e4 et d5, et deux blancs, en d4 et e5. Chaque joueur, noir et blanc, pose l'un après l'autre un pion de sa couleur sur l'othellier selon des règles précises. Le jeu s'arrête quand les deux joueurs ne peuvent plus poser de pion. On compte alors le nombre de pions. Le joueur ayant le plus grand nombre de pions de sa couleur sur l'othellier a gagné.

Les colonnes sont numérotées de gauche à droite par les lettres a à h ; les lignes sont numérotées de haut en bas par les chiffres 1 à 8.

Le joueur noir commence toujours la partie. Puis les joueurs jouent à tour de rôle, chacun étant tenu de capturer des pions adverses lors de son mouvement. Si un joueur ne peut pas capturer de pion(s) adverse(s), il est forcé de passer son tour. Si aucun des deux joueurs ne peut jouer, ou si l'othellier ne comporte plus de case vide, la partie s'arrête. Le gagnant en fin de partie est celui qui possède le plus de pions.

La capture de pions survient lorsqu'un joueur place un de ses pions à l'extrémité d'un alignement de pions adverses contigus et dont l'autre extrémité est déjà occupée par un de ses propres pions. Les alignements considérés peuvent être une colonne, une ligne ou une diagonale. Si le pion nouvellement placé vient fermer plusieurs alignements, il capture tous les pions adverses des lignes ainsi fermées. La capture se traduit par le retournement des pions capturés. Ces retournements n'entraînent pas d'effet de capture en cascade : seul le pion nouvellement posé est pris en compte.

Par exemple, la figure de gauche ci-dessous montre la position de départ. La figure centrale montre les quatre cases sur lesquelles Noir peut jouer, grâce à la capture d'un pion Blanc. Enfin, la figure de droite montre la position résultante si Noir joue en d3. Le pion Blanc d4 a été capturé (retourné), devenant ainsi un pion noir.

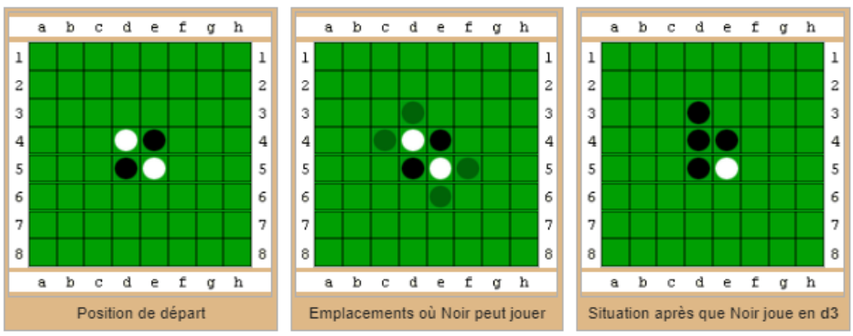


FIGURE 2.1 – Exemple Othello

Deuxième partie

Analyse

Chapitre 3

Structures de données

Pour implémenter le jeu Othello, nous avons choisi d'utiliser le langage de programmation *Python*. Simple à utiliser, peu contraignant, et donc parfait pour nous focaliser sur l'optimisation du code et l'implémentation de divers algorithmes d'intelligence artificielle.

La structure générale du projet est orientée objet, voici le détail :

```
src
├── main
│   └── fonctions gérant le déroulement de la partie
├── Board (class)
│   ├── update_adjacents: mise à jour des cases adjacentes
│   ├── update_state: mise à jour de l'état du plateau (= joue le coup)
│   ├── possible_moves: calcule la liste des coups possibles
│   ├── check_directions: vérifie pour chaque "ligne" (ou trajectoire) autour d'une case
│   │   si le coup est possible
│   ├── othe_type: retourne le type de pion opposé
│   └── copy: fait une copie de l'instance de la classe Board
├── Player (class)
└── AIPlayer (class)
    ├── play: retourne le coup joué en fonction de l'algorithme choisi
    ├── strat: retourne un résultat dépendant de la stratégie choisie
    └── les différents algorithmes de jeu (random, negamax, negamax-α-β...)
```

Détails importants sur la structure des données :

— Chaque coup possible *move* est représenté par une liste telle que : $move = [x, y, details]$ avec $details = [(dx, dy), (end\_x, end\_y)]$. dx et dy définissent la direction dans laquelle il faut retourner les pions, et $endx$, $endy$ sont les coordonnées de la pièce jusqu'à laquelle (exclue) il faut retourner les pions.

— le fichier constants.py comporte tous les paramètres d'exécution du jeu comme le type d'algorithme voulu, la profondeur maximale etc.

Chapitre 4

Principaux algorithmes

4.1 Algorithmes de l'Othello

Le jeu Othello comporte plusieurs règles, dont le respect de celles-ci est vérifié grâces a plusieurs algorithmes. Les algorithmes ci-dessous sont présents dans la classe *Board* du code source. C'est le code du plateau qui gère la bonne exécution de chaque coup joué. Pour optimiser le code, nous avons décidé d'utiliser une liste des cases adjacentes au pion. Cela permet d'aller plus vite lors de la vérification des coups possibles, et lors de l'actualisation de l'état du plateau.

4.1.1 Vérification des coups possibles

Afin de créer l'ensemble des coups possible à un instant donné, nous devons vérifier pour chaque case adjacente, chacune des directions. Si il existe au moins une direction jouable alors le coup est valide.

Algorithm 1: possible_moves

```
Data: type de pion : type, liste des adjacents : adjacents
Result: Ensemble des coups possible
moves ← [ ];
for adjacent in adjacents do
    check ← check_directions(adjacent, type);
    if au moins une direction est valide then
        moves.add(adjacent, check);
    end
end
return moves
```

Ici, pour chaque case adjacente, nous vérifions à l'aide de la fonction *check_directions* les directions valides. Chaque coup valide est ajouté à l'ensemble des coups possibles qui est retourné.

Voici le détail de la fonction *check_directions* :
Soit *directions* l'ensemble des directions autour d'un point (par exemple $(1, 0)$ représente un déplacement selon x uniquement, dans le sens positif).

Algorithm 2: check_directions

```
Data: type, x, y, adjacents, board
Result: Ensemble des coups possible
result ← [ ];
for k in [[0, ..., 7]] do
    dx, dy ← direction[k];
    current_x, current_y ← x + dx, y + dy;
    if current_x, current_y not in board or
     board[current_x, current_y] ≠ other_type(type) then
        continue;
    end
    while (current_x, current_y) in board do
        if board[current_x, current_y] = 0 then
            break;
        end
        if board[current_x, current_y] = type then
            result.add([(dx, dy), (current_x, current_y)]);
            break;
        end
        current_x ← current_x + dx;
        current_y ← current_y + dy;
    end
end
return result;
```

Dans cet algorithme, pour chaque direction possible (haut, bas, droite, gauche, diagonale, haut, droite...), nous vérifions si la case de jeu désirée de coordonnées (x, y) est adjacente à un pion de couleur adverse. Si c'est le cas, il faut alors vérifier si ce pion, et les autres qui le suivent dans cette même direction s'il y'en a, son "encadrés" par la couleur du joueur. Par exemple XOOOX est un encadrement de plusieurs O par X. Si cette condition est vérifiée, alors le coup est valide, et les pions adverses "encadrés" changeront de couleur.

4.1.2 Mise à jour du plateau

Après le calcul de l'ensemble des coups possible, le joueur doit jouer un coup parmi ceux-ci. Suite à ce jeu, le plateau est mis à jour.

La fonction *update_state* se charge de cette action, elle retourne les pions adverses entre le coup joué (x, y) et $move[1] = (end\_x, end\_y)$, suivant la direction $move[0] = (dx, dy)$.

4.2 Algorithmes de jeu du joueur IA

Les algorithmes en pseudo-code présents dans ce document sont une simplification des algorithmes présents dans le code source. Cela permet une meilleure compréhension.

Table 4.1 – Tableau des paramètres généraux liés aux algorithmes de jeu

Paramètres généraux	
$SIZE$: int	Dimensions du plateau
$ALGS$: [int, int]	Type d'algorithme pour les IA (noirs et blancs, 0 : random, 1 : negamax, 2 : nega-alpha-beta)
$STRATS$: [int, int]	Stratégie associée à chaque IA (noirs et blancs, 0 : positionnel, 1 : absolu, 2 : mobilité, 3 : mixte)
$GAME\_TYPE$: int	Type de partie (1 : joueur vs joueur, 2 : IA vs joueur, 3 : joueur vs IA, IA vs IA sinon)
MAX_INT : int	Entier le plus grand représentant ici l'infini
MAX_DEPTH : [int, int]	Profondeur maximum pour les algorithmes de recherche d'arbre
MAX_ITER : int	Nombre d'itérations maximum pour les algorithmes d'apprentissage par renforcement
$MC\_ROLLOUT\_METHOD$: [int, int]	type de rollout, random, probabiliste ou deterministe (voir 4.3.1)
$AVOID\_DUPLICATES$: bool	Détermine si on sauvegarde les états déjà traités pour éviter des doublons

Plusieurs algorithmes plus complexes et plus "intelligents" ont été implémentés, notamment en fonction de la stratégie a choisie.
Il existe quatre stratégies :

Table 4.2 – Tableau des stratégies envisageables par les algorithmes

Stratégies de jeu	
Positionnelle	Prise en compte des poids statique du tableau. L'évaluation est la différence entre les poids associés des deux joueurs
Absolue	Prise en compte de la différence du nombre de pions
Mobile	Maximise le nombre de coups possibles et minimise les coups de l'adversaire tout en essayant de prendre les coins
Mixte	Le jeu est divisé en trois phases où les stratégies peuvent différer : (i) en début de partie (20 à 25 premiers coups), le joueur IA choisit une stratégie de type 'positionnel', au milieu de partie, sélectionne une stratégie de 'mobilité' ; et enfin en fin de partie (10 à 16 derniers coups), sélectionne la stratégie 'absolu'

Voici le détail de quelques fonctions utilisées dans les algorithmes suivants :
— *strategy* : retourne la valeur associée à la stratégie
— *possible_moves* : retourne la liste des coups possibles
— *play* : joue le coup passé en paramètre et actualise le plateau de jeu

4.2.1 Negamax

Voici l'algorithme qui implémente Negamax (variante de min-max) :

Algorithm 3: negamax

Data: Plateau : board, profondeur de l'arbre : depth, liste des coups possibles : pm, type de pion : type
Result: Couple : (meilleur score, coup le plus rentable)
if $depth = \boldsymbol{MAX\_DEPTH}$ **then**
  return $strategy(board, pm)$;
end
$moves \leftarrow possible\_moves(type)$;
if $longueur(moves) = 0$ **then**
  return $strategy(board, pm, type)$;
end
$best \leftarrow -\mathbf{MAX\_INT}$;
$best\_moves \leftarrow [\,]$;
$score \leftarrow 0$;
for *move in moves* **do**
  $play(move, type)$;
  $score \leftarrow -negamax(board, depth + 1, moves, other\_type(type))[0]$;
  if $score = best$ **then**
    $best\_moves.add(move)$;
  end
  if $score > best$ **then**
    $best \leftarrow score$;
    $best\_moves = [move]$;
  end
end
$best\_move \leftarrow random(best\_moves)$;
return $best,\ best\_moves$

Cet algorithme, inspiré d'une publication de Tristant Cazenave [1], est une variante de *min-max*. L'explication de Tristan Casenave est la suivante : *"Plutôt que de tester si on est à un niveau pair ou impair pour savoir si on cherche à maximiser ou à minimiser l'évaluation, on peut inverser le signe des évaluations à chaque niveau, et toujours chercher à maximiser."*. Negamax a pour but de maximiser le "score" du joueur joué tout en minimisant celui du joueur affronté. Pour ce faire, il parcourt chaque coup possible jusqu'à une profondeur limite, et remonte le score atteint pour chaque chemin. Le coup menant au chemin avec le meilleur score est retenu.

4.2.2 Negamax-α-β

Voici l'algorithme qui implémente Negamax-α-β :

Algorithm 4: negamax_alpha_beta

```
Data: Plateau : board, profondeur de l'arbre : depth, liste des coups possibles : pm, alpha,
      beta, type de pion : type
Result: Couple : (meilleur score, coup le plus rentable)
if depth = MAX_DEPTH then
    return strategy(board, pm);
end
moves ← possible_moves(type);
if longueur(moves) = 0 then
    return strategy(board, pm, type);
end
best ← −MAX_INT;
best_moves ← [ ];
score ← 0;
for move in moves do
    play(move, type);
    score ← −negamax(board, depth + 1, moves, −alpha, −beta, other_type(type))[0];
    if score = best then
        best_moves.add(move);
    end
    if score > best then
        best ← score;
        best_moves = [move];
        if best > alpha then
            alpha ← best;
            if alpha > beta then
                break
            end
        end
    end
end
best_move ← random(best_moves);
return best, best_moves
```

Cet algorithme fut aussi inspiré de la publication de Tristan Casenave [1], il est une amélioration du negamax classique : pour chaque branche, il vérifie si le score justifie une exploration. Cela permet de couper les branches peu avantageuses, et donc de réduire la complexité temporelle et spaciale.

4.2.3 Monte-Carlo

Algorithme

Pour l'implémentation de cet algorithme, nous nous sommes inspirés de plusieurs explications sur le web [4], [2]. L'algorithme Monte-Carlo comporte quatre étapes majeures :

— Selection
— Expansion
— Simulation
— Back-propagation

Algorithm 5: Monte Carlo

```
Data: Ensemble des noeuds
Result: Meilleur coup selon l'algorithme
for iteration in ⟦1, MAX_ITER⟧ do
    if il existe un noeud non visité then
        res ← rollout(noeud);
        backpropagate(res, noeud);
    else
        for noeud depuis la racine do
            UCB1(noeud);
        end
        best ← max_UCB1();
        leaf ← best_leaf_from(best);
        extend(leaf);
    end
end
best_move ← max_UCB1();
return best_move
```

Algorithm 6: Rollout

Data: Noeud depuis lequel on rollout : node
Result: True si le noeud terminal atteint est une victoire, False sinon
while *node n'est pas terminal* **do**
  $node \leftarrow next\_node(node)$;
end
return $is\_win(node)$

Ici $next\_node(node)$ retourne le prochain noeud depuis *node* (en calculant les coups possibles depuis le noeud) de manière aléatoire, ou selon la méthode décrite plus bas.

Algorithm 7: Backpropagate

Data: Noeud depuis lequel on veut rétropropager : node, score : score
$nodes \leftarrow possible\_moves(node)$;
while *(parent :=node.parent) is not* $\emptyset$ **do**
  $parent.visits \leftarrow parent.visits + 1$;
  $parent.score \leftarrow parent.score + score$;
  $node \leftarrow parent$;
end

Algorithm 8: Expand

Data: Noeud depuis lequel on veut étendre : node, arbre : tree
$nodes \leftarrow possible\_moves(node)$;
for *node in nodes* **do**
  $tree.add(node)$;
end

L'objectif de l'algorithme de Monte-Carlo est d'effectuer un parcours sur un arbre, avec un nombre d'itérations limité. Chaque itération correspond à une visite de noeud avec une simulation ou une expansion. Ce parcours permet, au fil des itérations, de construire un coefficient de "réussite" à chaque noeud qui est relatif aux chances de gagner depuis ce noeud. Ainsi plus le nombre d'itérations est grand, plus le coefficient obtenu sur chaque noeud est proche de la réalité.

4.2.4 Random play

Le premier algorithme d' "intelligence artificielle" implémenté fut tout simplement un jeu aléatoire, qui retourne aléatoirement un coup parmi les coups possibles.

4.3 Ameliorations des algorithmes du joueur IA

4.3.1 Adaptation de Monte Calro à l'Othello

Le principe fondamental de la recherche Monte Carlo est la répétition d'une recherche aléatoire jusqu'à un état terminal. Or, cela implique qu'à chaque étape de simulation, l'algorithme choisi aléatoirement un coup parmis les coups possibles avec une probabilité suivant une loi uniforme. Pour maximiser les chances victoire, nous pouvons essayer de controller cet aléatoire pour le faire tendre vers des coup qui semblent à priori bon pour le joueur adverse (simuler un joueur adverse intelligent). Pour ce faire, on considère une matrice heuristique du jeu Othello (déjà utilisée dans α-β et *min-max*) $H \in \mathcal{M}_{8,8}(\mathbb{R})$ dont chaque coefficient est la valeur heuristique associée à la case. Ainsi, pour chaque liste de coups possibles, nous pouvons leur associer une valeur, et effectuer le procédé suivant :

(1) Soit pm la liste des coups possibles au noeud courant avec et $n = longueur(pm)$. Alors, lors de la simulation, dans la configuration de base on prend i aléatoirement dans $[\![0, n]\!]$ pour retourner $pm[i]$. Dans notre configuration, on remplace la loi uniforme par la suivante, avec $H(pm)$ la liste des valeures heuristiques associées à pm, et $S_H = \sum_{i=0}^{n} H(pm)[i]$:

TABLE 4.3 – Loi de probabilité pour un ensemble de coups possibles pm

Loi de probabilité				
X	$pm[0]$	$pm[1]$	$\dots$	$pm[n]$
$f_X(X)$	$\frac{H(pm)[0]}{S_H}$	$\frac{H(pm)[1]}{S_H}$	$\dots$	$\frac{H(pm)[n]}{S_H}$

Il existe en revanche un problème à contourner pour bien appliquer cette méthode : les valeurs heuristiques peuvent être négatives. Pour regler ce problème, il suffit de rammener les valeurs dans $[0, \infty[$.

(2) Une autre solution est de sélectionner à tout prix le meilleur coup d'après l'heuristique. Dans ce cas, on peut définir la fonction f_{max} qui sélectionne à chaque noeud de la simulation le meilleur coup :
Soit C l'ensemble de tous les coups possibles,

$$\begin{aligned} f_{max} : C^n &\longrightarrow C \\ u &\longmapsto max_{0 \leq i \leq n}(u[i]) \end{aligned}$$

Avec max au sens de la valeur de l'heuristique : $H(u)$.

Lorsque ce n'est pas au joueur adverse de jouer, nous prenons aléatoirement un coup parmis les coups possibles (comme la méthode classique).

En résumé, cela permet restreindre le choix des nœuds pendant une simulation, pour considérer que notre adversaire jouera de la manière la plus intelligente possible. Cela permet d'ajouter une hypothèse dans le jeu de l'adversaire et de peut-être mieux choisir les coups.

4.4 Détermination des états déjà traités

Afin de réduire l'ensemble des états parcourus par un algorithme, il est évident que supprimer les doublons est une méthode efficace.

Ainsi, dans nos algorithmes, l'option $AVOID\_DUPLICATES$ permet de stocker les états dans une liste, et si un état à déjà été visité auparavant, alors la recherche ne continue pas.

En revanche, notre architecture n'est clairement pas optimale et le stockage des états déjà traités ne donne pas de résultats interessants.

Troisième partie

Validation

Chapitre 5

Performances des algorithmes

5.1 Théorie

5.1.1 Negamax

L'algorithme Negamax effectue une recherche complète en parcourant absolument tout les nœuds jusqu'à la profondeur maximale indiquée. L'arbre de recherche grandit extrêmement rapidement, et à chaque nœud, il existe n nœuds enfants avec n le nombre de coups possibles. Or, le nombre de coups possibles n'est pas le même depuis chaque nœud. Pour simplifier les choses, nous allons définir un nombre de coups possibles moyen : $\overline{n_{pm}}$.

Ainsi, pour une profondeur maximale d, nous avons :

$$T_{minmax}(d) = \mathcal{O}(\overline{n_{pm}}^{\,d})$$

5.1.2 Negamax α-β

Cet algorithme est une amélioration de Negamax, il utilise une technique d'élagage afin d'éviter de parcourir certaines branches inutiles. De fait, la complexité est nettement plus faible.

Nous pouvons considérer que cet algorithme élague une branche sur 2, ce qui donne :

$$T_{\alpha\beta}(d) = \mathcal{O}(\overline{n_{pm}}^{\,\frac{d}{2}})$$

5.1.3 Monte-Carlo

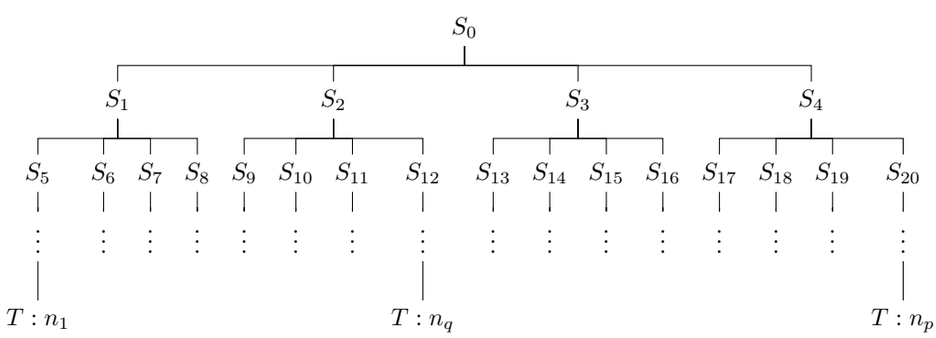
Rollout

possible_moves :
Il faut vérifier chaque directions pour chaque noeuds adjacent n_a. Dans le pire, 27 cases sont parcourues ce qui fait $27 \times n_a$. On définit $\overline{n_q}$ comme la distance moyenne entre le noeud racine et un noeud terminal. Pour chaque noeud à la couche i, on a donc $27 \times (\overline{n_q} - i) \times \overline{n_a}$, avec $\overline{n_a}$ le nombre moyen d'adjacents.

next_node :

— calcul des coup possibles ($= 27 \times (\overline{n_q} - i) \times \overline{n_a}$)
— choix aléatoire ($= \mathcal{O}(1)$)

Car on effectue $(\overline{n_q} - i)$ fois $next\_node(node)$ et $is\_win(.) = \mathcal{O}(1)$:



Back propagation

Si l'on se trouve à un noeud à la couche i, on a une complexité temporelle en $\mathcal{O}(i)$.

Expand

Il y a un appel de *possible_moves*, qui est un $\mathcal{O}(n_a)$ (à cause de possible_moves(node)).

Conclusion

En somme, nous pouvons estimer la complexité de cet algorithme en combinant celles de chaque étapes, pour un nombre d'itérations n :

couche 1 : $\overline{n_{pm}} \times (rollout + backprop) \longrightarrow$ calcul de $UCB1$

couche 2 : $\overline{n_{pm}} \times (rollout + backprop) + expand \longleftarrow$ utilisation de $UCB1$

couche 2 : $\overline{n_{pm}} \times (rollout + backprop) + expand \longleftarrow$ utilisation de $UCB1$

... Jusqu'à n

On a donc :

$$\overline{n_{pm}}(27(\overline{n_q} - 1)\overline{n_a} + \mathcal{O}(1)) + \sum_{i=2}^{n} m(i)\Phi(i)$$

avec :

$$m(i) = \left(\overline{n_{pm}}(27(\overline{n_q} - i)\overline{n_a} + \mathcal{O}(1)) \quad \mathcal{O}(1)\right)$$

et :

$$\Phi(i) = \begin{cases} \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \text{si } i = 2k,\ k \in \mathbb{Z} \\ \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \text{sinon} \end{cases}$$

Car on alterne entre $27(\overline{n_q} - i)\overline{n_a} + \mathcal{O}(1)$ et $\mathcal{O}(1)$ en fonction de la parité de i.

Nous pouvons dresser cette approximation :

$$T_{MCTS}(n) = \mathcal{O}(n \times \overline{n_{pm}} \times \overline{n_a} \times \overline{n_q}) = \mathcal{O}(n)$$

On remarque donc que le temps de calcul dépend linéairement du nombre d'itérations choisies.

5.2 Pratique

On fixe un nombre de parties N assez grand et fixe pour obtenir des moyennes, par exemple 50.
Ces données nous permettent de confirmer ou non le comportement asymptotique prévu théoriquement (bien que très simplifié).
Afin de mesurer les statistiques de victoires, tous les matchs sont contre un negamax-α-β de profondeur maximale 2.
Voici les résultats :

negamax :
(Inutile de s'interesser à la winrate de negamax prof 2 vs negamax-α-β car leur fonctionnement est le même, modulo les coupures alpha et beta)

```
[+] Moyenne:  0.05529668999988643 s
[+] Moyenne des coup du joueur noir:  0.000740269362163928 s
[+] Moyenne des coup du joueur blanc:  0.000998942245530113 s
[+] Victoires des noirs:  50 %
[+] Victoires des blancs:  50 %
[+] Matchs nuls:  0 %
```

FIGURE 5.1 – Résultats pour $i = 2$

```
[+] Moyenne:  2.5507291549998627 s
[+] Moyenne des coup du joueur noir:  0.0006561009811475145 s
[+] Moyenne des coup du joueur blanc:  0.07993823154375022 s
[+] Victoires des noirs:  15 %
[+] Victoires des blancs:  80 %
[+] Matchs nuls:  5 %
```

FIGURE 5.2 – Résultats pour $i = 4$

```
[+] Moyenne:  278.68995153999987 s
[+] Moyenne des coup du joueur noir:  0.000662333035657841 s
[+] Moyenne des coup du joueur blanc:  8.832773883392822 s
[+] Victoires des noirs:  15 %
[+] Victoires des blancs:  85 %
[+] Matchs nuls:  0 %
```

FIGURE 5.3 – Résultats pour $i = 6$

Nous pouvons essayer de comparer la complexité observée et la théorique. Pour la théorique, pour un coefficient de 2.5 qui représente le nombre moyen de coups possible $\times$ facteur relatif au CPU, nous avons :

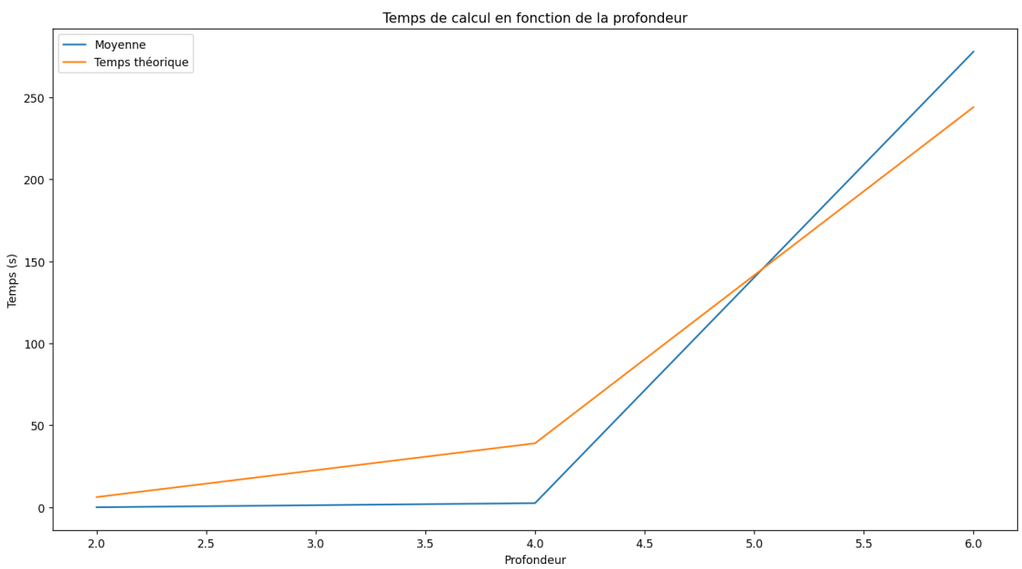


FIGURE 5.4 – Comparaison des résultats pratiques et théoriques

negamax-α-β :
(Inutile de s'interesser à la winrate de negamax-α-β prof 2 vs lui-même)

```
[+] Moyenne:  0.043982925000091204 s
[+] Moyenne des coup du joueur noir:  0.0007248300074206014 s
[+] Moyenne des coup du joueur blanc:  0.0006676212499960294 s
[+] Victoires des noirs:  55 %
[+] Victoires des blancs:  45 %
[+] Matchs nuls:  0 %
```

FIGURE 5.5 – Résultats pour $i = 2$

```
[+] Moyenne:  0.6501759750000019 s
[+] Moyenne des coup du joueur noir:  0.0006268961589913305 s
[+] Moyenne des coup du joueur blanc:  0.019947130163649 s
[+] Victoires des noirs:  25 %
[+] Victoires des blancs:  70 %
[+] Matchs nuls:  5 %
```

FIGURE 5.6 – Résultats pour $i = 4$

```
[+] Moyenne:  19.05113202500006 s
[+] Moyenne des coup du joueur noir:  0.0006321165376816914 s
[+] Moyenne des coup du joueur blanc:  0.6018034596899761 s
[+] Victoires des noirs:  20 %
[+] Victoires des blancs:  80 %
[+] Matchs nuls:  0 %
```

FIGURE 5.7 – Résultats pour $i = 6$

Pour le même coefficient vu dans la partie negamax, on a :

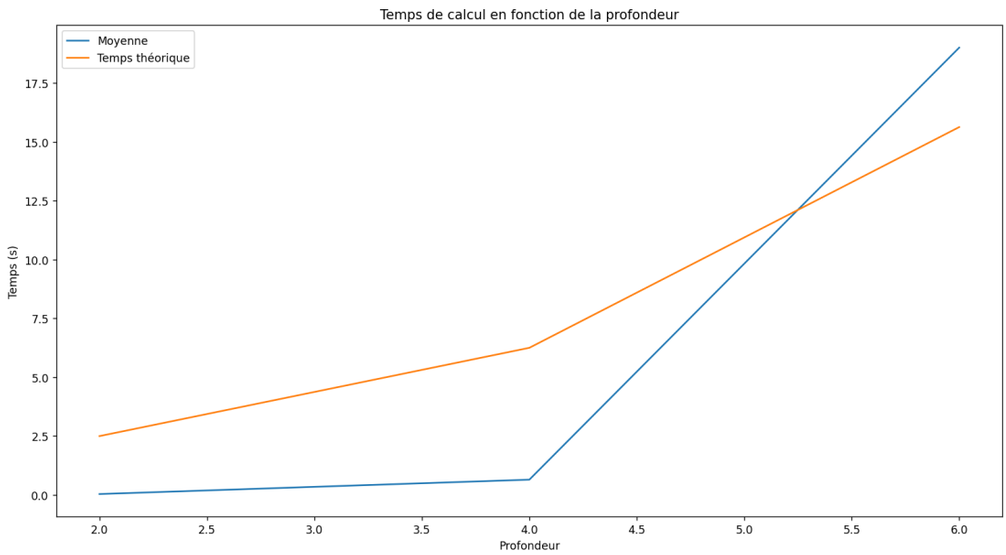


FIGURE 5.8 – Comparaison des résultats pratiques et théoriques

Les résultats pratiques sont sûrement biaisés par l'implémentation en python et le coût différent de $\mathcal{O}(1)$ de certaines opérations.

Monte-Carlo :

```
[+] Moyenne:  2.770787264999944 s
[+] Moyenne des coup du joueur noir:  0.000749999638337779 s
[+] Moyenne des coup du joueur blanc:  0.08630896539317674 s
[+] Victoires des noirs:  85 %
[+] Victoires des blancs:  15 %
[+] Matchs nuls:  0 %
```

FIGURE 5.9 – Résultats pour $i = 100$

```
[+] Moyenne:  8.475545284999953 s
[+] Moyenne des coup du joueur noir:  0.0007948232014633429 s
[+] Moyenne des coup du joueur blanc:  0.26656550338626667 s
[+] Victoires des noirs:  70 %
[+] Victoires des blancs:  25 %
[+] Matchs nuls:  5 %
```

FIGURE 5.10 – Résultats pour $i = 300$

```
[+] Moyenne:  12.732108464999918 s
[+] Moyenne des coup du joueur noir:  0.0007891068869179011 s
[+] Moyenne des coup du joueur blanc:  0.3981194122119725 s
[+] Victoires des noirs:  75 %
[+] Victoires des blancs:  25 %
[+] Matchs nuls:  0 %
```

FIGURE 5.11 – Résultats pour $i = 500$

```
[+] Moyenne:  17.133711529999843 s
[+] Moyenne des coup du joueur noir:  0.0007382044942587235 s
[+] Moyenne des coup du joueur blanc:  0.5400004181727809 s
[+] Victoires des noirs:  60 %
[+] Victoires des blancs:  40 %
[+] Matchs nuls:  0 %
```

FIGURE 5.12 – Résultats pour $i = 700$

```
[+] Moyenne:  23.197908604999885 s
[+] Moyenne des coup du joueur noir:  0.0007911929794046815 s
[+] Moyenne des coup du joueur blanc:  0.7294736562957564 s
[+] Victoires des noirs:  70 %
[+] Victoires des blancs:  25 %
[+] Matchs nuls:  5 %
```

FIGURE 5.13 – Résultats pour $i = 900$

Résultats :

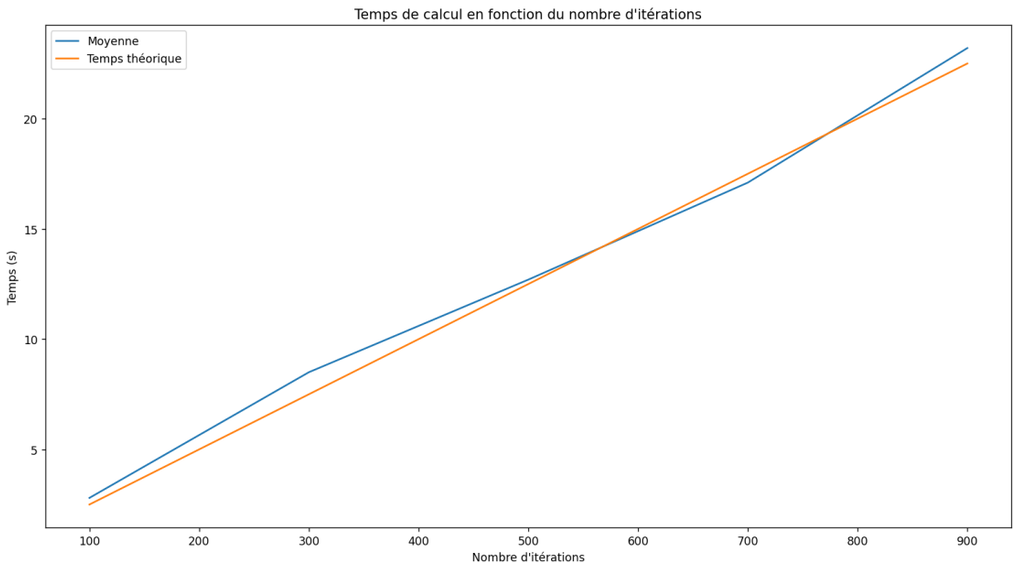


FIGURE 5.14 – Comparaison des résultats pratiques et théoriques, pour Monte-Carlo

On remarque bien le caractère linéaire de le complexité temporelle en fonction du nombre d'itérations.

Chapitre 6

Statistiques de jeu

6.1 Tournoi entre algorithmes

6.1.1 Matches

Pour comparer, nous voici un tournoi. Chaque algorithme aura un temps d'exécution limité à 10 secondes par coup. Les points attribués sont :

— $+(P_W - P_L)$ points par victoire, avec P_W le pourcentage de victoires sur toutes les parties et P_L le pourcentage de victoires de l'adversaire (et donc de défaites)
— $+0$ sinon

Phase 1 : affrontements entre algorithmes

Dans la première phase, nous allons organiser des matches entre les différents types d'algorithmes afin d'analyser les performances de chacun d'eux par rapport aux autres. Pour comparer équitablement chaque algorithme, nous n'allons plus utiliser une profondeur max ou un nombre d'itérations max pour nos algorithmes, mais un temps de calcul maximum.

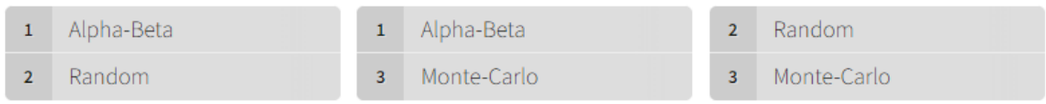


FIGURE 6.1 – Matches de la phase 1

Phase 2 : affrontements par stratégies

Dans cette deuxième phase, nous allons comparer la performance des différentes stratégies disponibles. Pour ce faire, nous fixons la profondeur (par exemple 4) et nous effectuons les matches suivants :

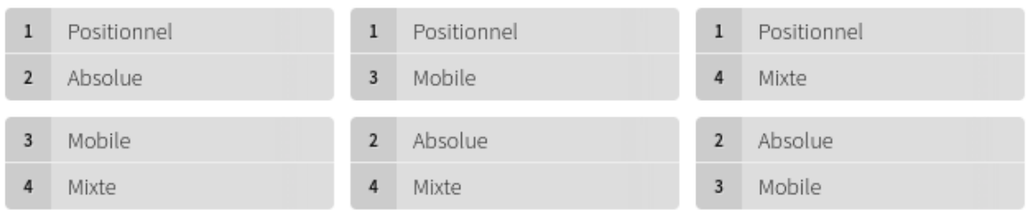


FIGURE 6.2 – Matches de la phase 2

Phase 3 : affrontements par heuristique

Afin de réduire le temps de calcul des algorithmes type minmax, il est nécéssaire de définir une ou plusieurs heuristiques pour évaluer le score d'un coup joué. Dans ce TP nous avons utilisé 2 heuristiques (figure 6.3) :

	a	b	c	d	e	f	g	h
1	500	-150	30	10	10	30	-150	500
2	-150	-250	0	0	0	0	-250	-150
3	30	0	1	2	2	1	0	30
4	10	0	2	16	16	2	0	10
5	10	0	2	16	16	2	0	10
6	30	0	1	2	2	1	0	30
7	-150	-250	0	0	0	0	-250	-150
8	500	-150	30	10	10	30	-150	500

	a	b	c	d	e	f	g	h
1	100	-20	10	5	5	10	-20	100
2	-20	-50	-2	-2	-2	-2	-50	-20
3	10	-2	-1	-1	-1	-1	-2	10
4	5	-2	-1	-1	-1	-1	-2	5
5	5	-2	-1	-1	-1	-1	-2	5
6	10	-2	-1	-1	-1	-1	-2	10
7	-20	-50	-2	-2	-2	-2	-50	-20
8	100	-20	10	5	5	10	-20	100

FIGURE 6.3 – Heuristiques utilisées pour l'Othello

Seul les algorithmes negamax et negamax-α-β utilisents ces heuristiques, nous allons donc les comparer avec negamax-α-β en profondeur 6.

Phase 4 : amélioration MCTS

Nous allons vérifier si l'amélioration proposée est fonctionnelle.

6.1.2 Résultats

Phase 1

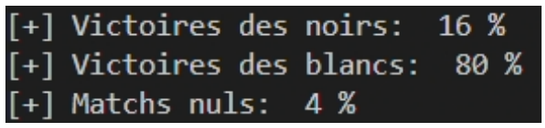


FIGURE 6.4 – Résultat : Random VS Alpha Beta

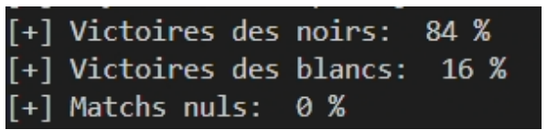


FIGURE 6.5 – Résultat : Alpha Beta VS Monte Carlo

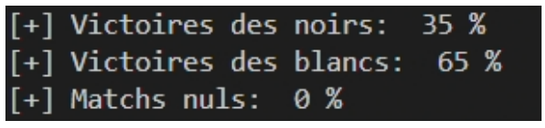


FIGURE 6.6 – Résultat : Random VS Monte Carlo

TABLE 6.1 – Scores finaux de chaque stratégie

Résultats de la phase 1			
	Random	Alpha Beta	Monte-Carlo
Random	/	64	30
Alpha Beta	0	/	0
Monte-Carlo	0	68	/
Totaux	0	132	30

L'algorithme de recherche alpha-beta (et donc la méthode minmax) domine random et Monte-Carlo.

Phase 2

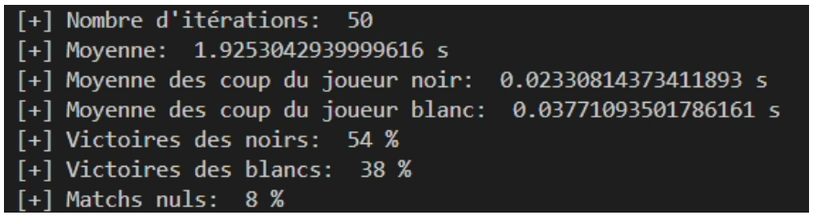


FIGURE 6.7 – Positionnelle VS Absolue

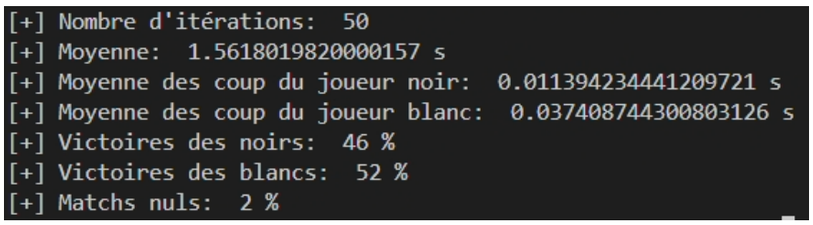


FIGURE 6.8 – Positionnelle VS Mobilité

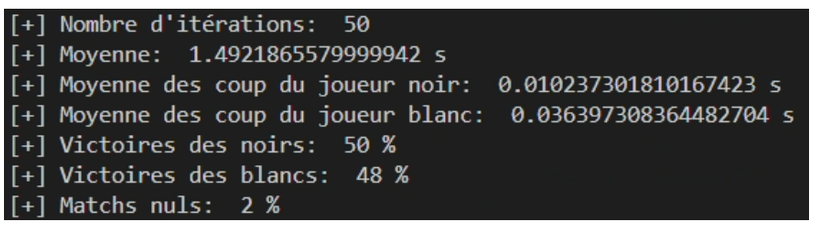


FIGURE 6.9 – Positionnelle VS Mixte

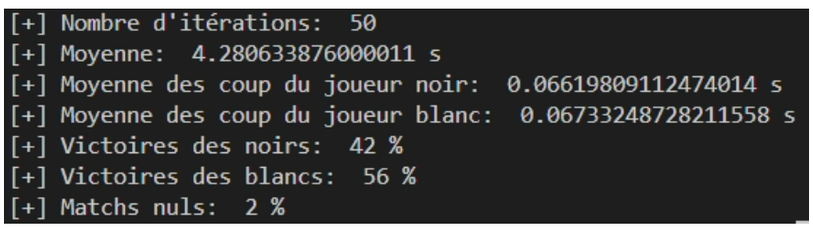


FIGURE 6.10 – Mobilité VS Mixte

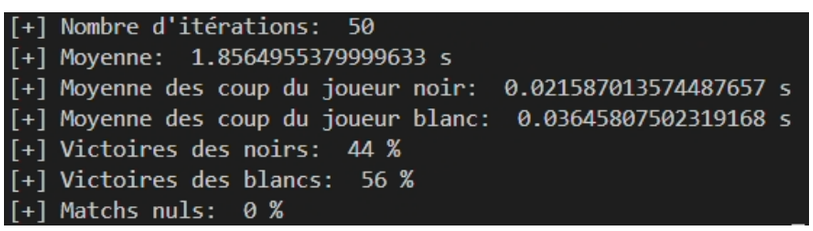


FIGURE 6.11 – Absolue VS Mixte

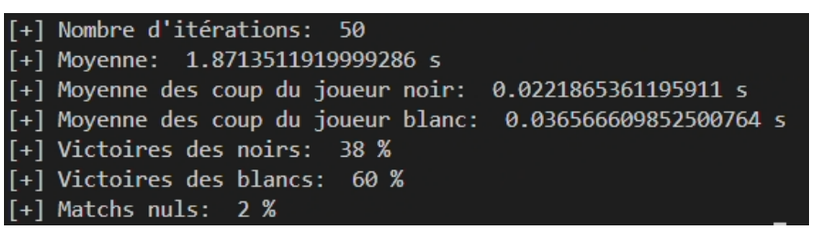


FIGURE 6.12 – Absolue VS Mobilité

TABLE 6.2 – Scores finaux de chaque stratégie

Résultats de la phase 2				
	Positionnelle	Absolue	Mobilité	Mixte
Positionnelle	/	0	6	0
Absolue	16	/	22	12
Mobilité	0	0	/	14
Mixte	2	0	0	/
Totaux	18	0	28	26

La stratégie Mobilité est la gagnante de ce tournoi, très proche des résultats de Mixte en termes de points. Si on analyse de plus près, Mixte à 14 pourcents de victoires en plus contre Mobilité. Ainsi, Mixte est peut-être au moins aussi performant que Mobilité, en fonction des conditions. Nous en parlerons dans le chapitre 7.

Phase 3

Pour 50 itérations, voici les résultats :

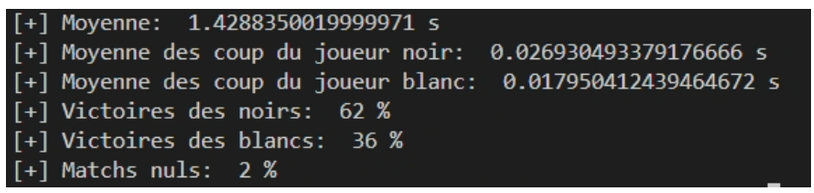


FIGURE 6.13 – Résultat : heuristique 2 (noir) VS heuristique 1 (blanc)

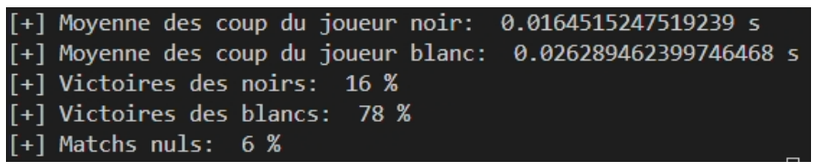


FIGURE 6.14 – Résultat : heuristique 1 (noir) VS heuristique 2 (blanc)

On remarque que l'heuristique 2 est bien meilleure.

Phase 4

Amélioration déterministe (2) :

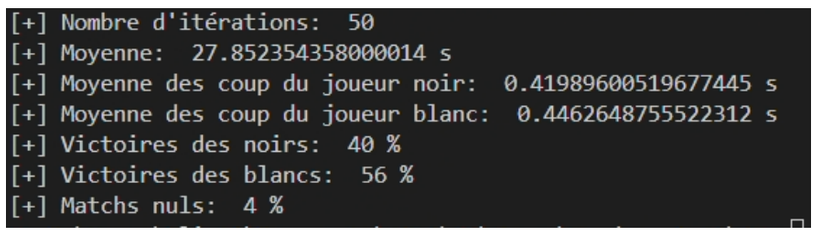


FIGURE 6.15 – Résultat pour MCTS classique VS MCTS avec amélioration (2)

Quatrième partie

Discussion

Chapitre 7

Discussion des résultats

7.1 Algorithmes

Les résultats précedents montrent une domination de Alpha-Beta sur Monte-Carlo et Random. Pour la domination sur random, cela est évident. En revanche, Alpha-Beta est meilleur que Monte-Carlo sur l'Othello pour des raisons de complexité du jeu. En effet le jeu Othello est assez simple au sens des règles, et l'ensemble des coup possibles reste petit pour chaque coups. De plus la dimension du plateau est assez petite, et les parties sont assez courtes (aux alentours de 63.5 coups en moyenne). En somme, Monte-Carlo n'utilise pas tout son potentiel sur ce jeux, contrairement au jeu de GO où alpha beta s'incline.

7.2 Stratégies

La stratégie positionnelle vise à jouer le meilleur coup à un instant précis, par rapport à l'heuristique. Il n'y a donc pas vraiment de prise en compte de l'adversaire, et le meilleur coup à un instant t n'appartient pas nécessairement à la meilleure suite de coups. Un coup moyen peut parfois débloquer des très bon coups.

La stratégie absolue prend en compte la différence entre le nombre de pions du joueur avec l'adversaire. Elle fait donc en sorte de maximiser l'écart et tente d'étendre cet écart. Le problème est que dans l'Othello, il n'est pas rare de voir des retournements de situations qui renverse le jeu proportionnellement à cet écart.

La stratégie de mobilité maximise le nombre de coups possibles et minimise ceux de l'adversaire. Il y a donc une restriction du choix de l'adervsaire ce qui réduit la probabilité de tomber sur un excellent coup. Cela permet de garder un avantage constant.

La stratégie mixte est un mix de positionnel en début de partie, de mobilité puis absolue en fin de partie.

Les stratégies mobilité et mixte se démarquent car elles conservent leur avantage grâce à la mobilité, et la mixte l'emporte contre mobilité car en début de partie, peu de pions sont posés donc le potentiel de la stratégie mobilité n'est pas utilisé. Le fait de vouloir maximiser son score avec la positionnelle donne un premier élan en début de partie en cherchant les coins. En fin de partie, la stratégie absolue permet d'empecher les retournements de situations qui peuvent avoir lieu dans une mobilité simple.

7.3 Heuristiques

Comme vu précédement, l'heuristique 2 est bien meilleure. Cela est sûrement dû au fait que l'heuristique 2 déplace le jeu sur les bords et l'anneau exterieur au centre. L'heuristique 1 force beaucoup les coins mais laisse beaucoup de coups joués au centre être effectués. L'heuristique 2 semble donc tendre plus vite vers des positions fortes.

7.4 Coefficient C de l'UCB1

Le coefficient C de la fonction UCB1 qui est au coeur de l'algorithme de Monte-Carlo semble affecter ses performances (table 7.1) :

TABLE 7.1 – Résultats Monte-Carlo (1000 itérations) contre α-β (profondeur 6)

C	1	1.5	2
Winrate (%)	28	34	20

Le coefficient C permet d'ajouter une balance entre le fait de rester dans la branche actuelle ou de changer de branche pour explorer si une meilleure solution existe ailleur. En effet, pour tout noeud i ayant comme paramètres n_i : nombre de visites totales, et w_i : nombre de victoires rencontrées depuis ce noeud, on a :

$$UCB1(node) = \frac{w_i}{n_i} + C\sqrt{\frac{\log(N)}{n_i}}$$

Avec N le nombre de visites totales de la racine.

Admettons que $n_0 = 1$ et $w_0 = 1$, et $n_1 = 1$ et $w_1 = 0$. Ainsi si $C = 0$ la valeur UCB1 du noeud 0 sera toujours plus élevée car elle sera toujours plus grande que 0. L'algorithme n'ira jamais explorer le noeud 1. Donc plus C est grand, plus il "force", au bout d'un moment (quand n_i augmente), à aller explorer les autres noeuds. Il faut alors trouver la bonne balance entre "explorer un peu partout" et "explorer au maximum une branche qui mène à une victoire".

Chapitre 8

Perspectives et améliorations

8.1 Amélioration de la stratégie des joueurs IA

Comme énoncé plus haut, une stratégie importante dans ce jeu est de minimiser le nombre de coups possible à notre adversaire. Ainsi, nous pouvons étendre certains algorithmes dans le but de se focaliser sur cette stratégie. De plus, il serait intéressant d'utiliser un algorithme d'apprentissage pour apprendre des stratégies/ouvertures connues à notre IA. Cela permettrait de construire une évaluation des coups bien plus solide qu'une simple heuristique.

8.2 Détermination des états déjà traités : amélioration de la complexité spaciale

Le jeu Othello est un jeu symétrique, et donc toute matrice heuristique sera symétrique également. Les symétries de l'Othello sont selon l'axe vertical, horizontal et les axes diagonaux. Ainsi, pour tout état dans l'ensemble des états possibles, il existe 3 autres états équivalents au sens du score heuristique.

Explications :
Soit $H \in \mathcal{M}_{8,8}$ tel que $H = (h_{i,j})_{1 \leq i,j \leq 8}$ la matrice de l'heuristique.
On a donc :

$$\forall i, j \in [\![1, 8]\!],\ h_{i,j} = h_{9-i,9-j} = h_{i,9-j} = h_{9-i,j}$$

Exemple d'heuristique :

	a	b	c	d	e	f	g	h
1	500	-150	30	10	10	30	-150	500
2	-150	-250	0	0	0	0	-250	-150
3	30	0	1	2	2	1	0	30
4	10	0	2	16	16	2	0	10
5	10	0	2	16	16	2	0	10
6	30	0	1	2	2	1	0	30
7	-150	-250	0	0	0	0	-250	-150
8	500	-150	30	10	10	30	-150	500

FIGURE 8.1 – Exemple d'heuristique

Donc peu importe dans quel état l'Othello est, si nous effectuons une rotation de l'ensemble des pions de $\theta = 0[\frac{\pi}{2}]$ le déroulement du jeu sera le même. Cela nous permet de représenter plus simplement les états, et de limiter la complexité spatiale en réduisant la liste d'états différents parcourus.

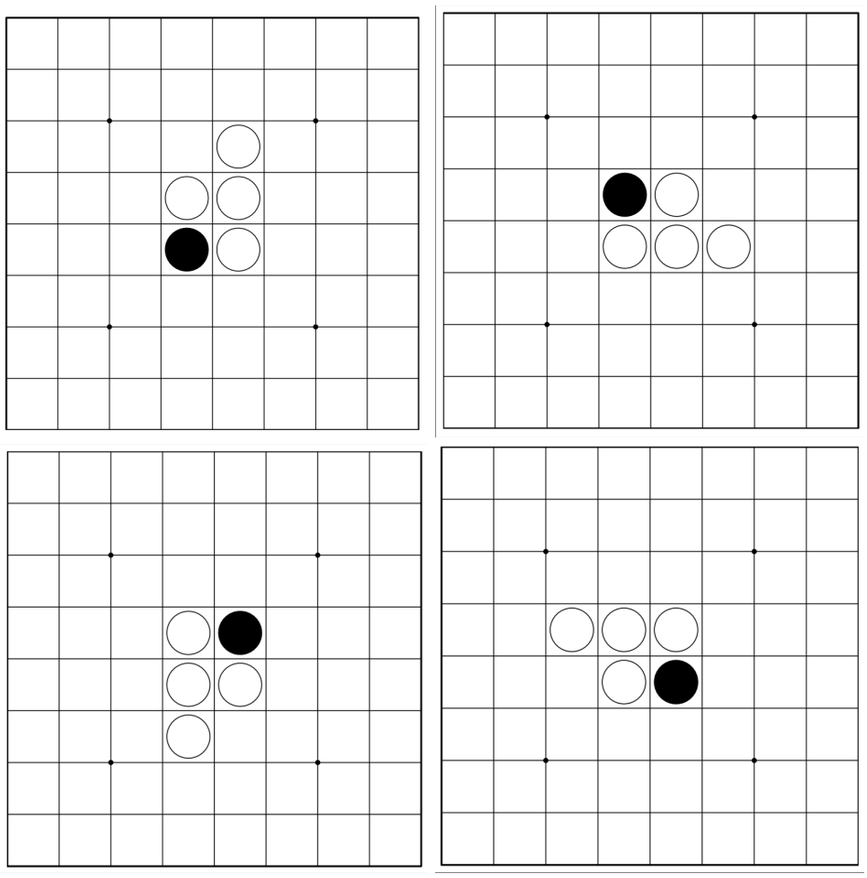


FIGURE 8.2 – Exemple d'états équivalents

En effet, si l'on stocke uniquement les états différents (en prenant en compte les similitudes précédentes) nous réduisons par 4 la taille de la liste des états connus.

8.3 Apprentissage par renforcement

Pour aller plus loin, il serai interessant d'implémenter un algorithme qui apprend de ses parties passées, autrement dit de l'apprentissage par renforcement. Il existe certains articles montrant une application de ce type d'algorithmes sur le jeu Othello [3].

Cinquième partie

Conclusion

Dans le cadre du cours de Fondements de l'Intelligence Artificielle, les analyses des différents algorithmes d'IA pour le jeu d'Othello a permis de mettre en lumière les performances comparatives et les domaines d'application spécifiques de chaque approche algorithmique. Les techniques de Negamax, Negamax-α-β, Monte-Carlo, et le jeu aléatoire ont été explorées, chacune offrant un aperçu précieu sur la stratégie et l'optimisation des processus de décision automatisée.

Les résultats obtenus montrent que l'algorithme Negamax-alpha-beta offre un équilibre optimal entre performance et coût calculatoire pour le jeu d'Othello, en raison de son efficacité à élaguer les branches non pertinentes dans l'arbre de jeu. En parallèle, l'approche Monte-Carlo, bien qu'efficace dans des contextes de jeux plus complexes comme le Go, s'est avérée moins adaptée à la simplicité relative d'Othello, mettant en évidence l'importance de choisir l'algorithme approprié au contexte spécifique du problème.

L'exploration des différentes stratégies de jeu, telles que la positionnelle, la mobilité et la mixte, a révélé que la stratégie de mobilité, qui maximise les options de jeu tout en limitant celles de l'adversaire, est particulièrement efficace dans les phases moyennes à tardives du jeu.

Finalement, ce projet nous a servi à mettre en application des théories et concepts appris en classe, facilitant une meilleure compréhension des enjeux de l'intelligence artificiel, notamment dans les jeux de stratégie et renforçant nos compétences en programmation et en analyse algorithmique. Les perspectives d'amélioration suggérées, notamment l'optimisation de l'évaluation des états de jeu et l'intégration de l'apprentissage par renforcement, ouvrent la voie à de futures recherches qui pourraient encore améliorer les capacités de décision automatisée dans Othello et au-delà.

Bibliographie

[1] Tristan Cazenave. *Des Optimisations de l'Alpha-Béta*. URL : https://www.lamsade.dauphine.fr/~cazenave/papers/berder00.pdf.

[2] Int8. *Monte Carlo Tree Search – beginners guide*. URL : https://int8.io/monte-carlo-tree-search-beginners-guide/#Policy_network_training_in_Alpha_Go_and_Alpha_Zero.

[3] Michiel van der Ree et Marco Wiering. *Reinforcement Learning in the Game of Othello : Learning Against a Fixed Opponent and Learning from Self-Play*. URL : https://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/paper-othello.pdf.

[4] Wikipedia. *Monte Carlo tree search*. URL : https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.