INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES HAUTS-DE-FRANCE

Université Polytechnique HAUTS-DE-FRANCE

14/04/2024

Développement d'une intelligence artificielle sur le jeu Othello

Rapport de projet

Auteurs : Quentin Espanet et Bastien Ubassy

Professeur : René Mandiau

Fig. 1. – Ot'hello World (Capture d'écran)

Table des matières

1.Introduction 2
1.1. Préambule 2
1.2. Objectif du projet 2
1.3. Choix techniques 3
2. Analyse 3
2.1. Algorithmes de recherches 3
2.1.1. MiniMax 3
2.1.1.1. Pseudocode 3
2.1.1.2. Implémentation en Python 3
2.1.2. NegaMax 4
2.1.2.1. Pseudocode 4
2.1.2.2. Implémentation en Python 5
2.1.3. Alpha-Beta (MiniMax) 5
2.1.3.1. Pseudocode 5
2.1.3.2. Implémentation en Python 6
2.1.4. Alpha-Beta (NegaMax) 7
2.1.4.1. Pseudocode 7
2.1.4.2. Implémentation en Python 7
2.2. Heuristiques 8
2.2.1. Comparaison du nombre de pions 8
2.2.2. Matrice d'évaluation 1 8
2.2.3. Matrice d'évaluation 2 8
3. Validation 9
3.1. Implémentation de l'algorithme de jeu aléatoire en Python 9
3.2. Tests 9
3.2.1. Comparaison des algorithmes de recherche 9
3.2.1.1. Profondeur 1 9
3.2.1.2. Profondeur 2 10
3.2.1.3. Comparaison des algorithmes Alpha-Beta (MiniMax) et Alpha-Beta (NegaMax) ... 11
3.2.2. Comparaison des heuristiques 12
3.2.3. Comparaison des profondeurs de recherche 13
3.2.4. Comparaison heuristique mixte et heuristique non mixte 14
3.2.5. Comparaison avec mémoire et sans mémoire 15
3.3. Bilan 16
4. Discussion 17
4.1. Mémorisation des coups 17
4.2. Heuristiques 18
4.3. Profondeur 18
4.4. Interface graphique 18
5. Conclusion 18
6. Annexes 19
6.1. Bibliothèques python externes utilisées 19

1.Introduction

1.1. Préambule

Othello est un jeu de société combinatoire abstrait qui se joue à deux joueurs adverses.

C'est un jeu de plateau qui se joue sur un damier de 8x8 cases. Les pions sont bicolores, noirs d'un côté et blancs de l'autre. Le but du jeu est de capturer le plus de pions de l'adversaire en les retournant pour les transformer en pions de sa propre couleur. Les pions noirs commencent la partie.

Le jeu est terminé lorsque le plateau est rempli ou que les deux joueurs ne peuvent plus jouer. Le joueur ayant le plus de pions de sa couleur sur le plateau remporte la partie.

L'avantage d'Othello est la simplicité de ses règles qui permettent tout de même de proposer très rapidement une explosion combinatoire.

1.2. Objectif du projet

L'objectif de ce projet était de développer plusieurs intelligences artificielles capable de jouer à Othello contre un humain ou contre une autre IA. Quatres méthodes de recherche ont été étudiées :

- MiniMax
- Alpha-Beta (sur MiniMax)
- NegaMax
- Alpha-Beta (sur NegaMax)

Nous avons également utilisé trois heuristiques différentes pour évaluer les plateaux de jeu :

- Comparaison du nombre de pions
- Utilisation de la première matrice vue en cours

	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

Fig. 2. – Première matrice d'évaluation

- Utilisation de la deuxième matrice vue en cours

	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

Fig. 3. – Seconde matrice d'évaluation

Pour améliorer nos résultats, nous avons également utilisé une heuristique mixte qui utilise une matrice d'évaluation en début de partie et la comparaison du nombre de pions en fin de partie.

1.3. Choix techniques

Nous avons décidé d'utiliser le langage Python pour ce projet. En effet, ce langage que nous maîtrisons bien nous permettait de rapidement mettre en place le jeu d'Othello avec une interface graphique. Nous avons utilisé le framework Flask pour créer une application web qui sert d'interface graphique pour jouer à Othello.

Nous avons également prévu une partie sans interface graphique afin de faire nos smulations plus rapidement.

2. Analyse

2.1. Algorithmes de recherches

2.1.1. MiniMax

2.1.1.1. Pseudocode

```
MiniMax(nœud, profondeur, joeurMaximisant):
si profondeur = 0 ou nœud est une feuille alors
    retourner la valeur heuristique de nœud
si joeurMaximisant alors
    valeur := −∞
        pour chaque fils de nœud faire
            valeur := max(valeur, MiniMax (fils, profondeur − 1, FALSE))
    sinon
        valeur := +∞
            pour chaque fils de nœud faire
                valeur := min(valeur, MiniMax (fils, profondeur − 1, TRUE))
    retourner valeur
```

2.1.1.2. Implémentation en Python

```
def minimax(self, depth, time_left, maximizingPlayer, heuristic):
    if depth == 0 or time_left < 0 or self.go:
        return self.heuristic_choice(heuristic), []

    time_begin = time.perf_counter()

    if maximizingPlayer:
        best_value = -math.inf
        best_move = []
        for x, y in self.shadow_pawn:
            sim_board = self.copy()
            again = sim_board.make_move(x, y)
            if again:
               child_value,_ = sim_board.minimax(depth - 1, time_left - (time.perf_counter()
- time_begin), True, heuristic)
            else:
               child_value,_ = sim_board.minimax(depth - 1, time_left - (time.perf_counter()
- time_begin), False, heuristic)
            if child_value > best_value:
                best_value = child_value
                best_move = [x, y]
        return best_value, best_move

    else:
        best_value = math.inf
        best_move = []
        for x, y in self.shadow_pawn:
            sim_board = self.copy()
            again = sim_board.make_move(x, y)
            if again:
               child_value,_ = sim_board.minimax(depth - 1, time_left - (time.perf_counter()
- time_begin), False, heuristic)
            else:
               child_value,_ = sim_board.minimax(depth - 1, time_left - (time.perf_counter()
- time_begin), True, heuristic)
            if child_value < best_value:
                best_value = child_value
                best_move = [x, y]
        return best_value, best_move
```

2.1.2. NegaMax

2.1.2.1. Pseudocode

NegaMax(nœud, profondeur, couleur):
1 si profondeur = 0 ou nœud est une feuille alors
2 retourner couleur × la valeur heuristique de nœud
3 valeur := −∞
4 pour chaque fils de nœud faire
5 valeur := max(valeur, -NegaMax (fils, profondeur − 1, −couleur))
6 retourner valeur

2.1.2.2. Implémentation en Python

```
def negamax(self, depth, time_left, color, heuristic):
    if depth == 0 or time_left < 0 or self.go:
        return self.heuristic_choice(heuristic) * color, []

    best_value = -math.inf
    time_begin = time.perf_counter()
    best_move = []

    for x, y in self.shadow_pawn:
        sim_board = self.copy()
        sim_board.make_move(x, y)
        child_value,_ = sim_board.negamax(depth - 1, time_left - (time.perf_counter()
- time_begin), -color, heuristic)
        if -child_value > best_value:
            best_value = -child_value
            best_move = [x, y]
    return best_value, best_move
```

2.1.3. Alpha-Beta (MiniMax)

2.1.3.1. Pseudocode

```
Alpha-Beta_MiniMax(node, depth, maximizingPlayer):
si profondeur = 0 ou nœud est une feuille alors
    retourner la valeur de nœud
sinon
    si nœud est de type Min alors
        v = +∞ pour tout fils de nœud faire
            v = min(v, Alpha-Beta_MiniMax (fils, α, β)) si α ≥ v alors  // coupure alpha
                retourner v
            β = min(β, v)
    sinon
        v = -∞ pour tout fils de nœud faire
            v = max(v, Alpha-Beta_MiniMax (fils, α, β)) si v ≥ β alors  // coupure beta
                retourner v
            α = max(α, v)
retourner v
```

2.1.3.2. Implémentation en Python

```
def alphabeta_minimax(self, depth, time_left, maximizingPlayer, heuristic, alpha,
beta):
    if depth == 0 or time_left < 0 or self.go:
        return self.heuristic_choice(heuristic), []

    if maximizingPlayer:
        best_value = -math.inf
        best_move = []
        time_begin = time.perf_counter()
        for x, y in self.shadow_pawn:
            sim_board = self.copy()
            sim_board.make_move(x, y)
              child_value,_ = sim_board.alphabeta_minimax(depth - 1, time_left -
(time.perf_counter() - time_begin), False, heuristic, alpha, beta)
            if child_value > best_value:
                best_value = child_value
                best_move = [x, y]
            alpha = max(alpha, best_value)
            if beta <= alpha:
                break
        return best_value, best_move

    else:
        best_value = math.inf
        best_move = []
        time_begin = time.perf_counter()
        for x, y in self.shadow_pawn:
            sim_board = self.copy()
            sim_board.make_move(x, y)
```

```
            child_value,_ = sim_board.alphabeta_minimax(depth - 1, time_left -
(time.perf_counter() - time_begin), True, heuristic, alpha, beta)
            if child_value < best_value:
                best_value = child_value
                best_move = [x, y]
            beta = min(beta, best_value)
            if beta <= alpha:
                break
        return best_value, best_move
```

2.1.4. Alpha-Beta (NegaMax)

2.1.4.1. Pseudocode

Alpha-Beta_NegaMax(nœud, profondeur, α, β, couleur):

```
si profondeur = 0 ou nœud est une feuille alors
    retourner couleur × la valeur heuristique de nœud
valeur := −∞
pour chaque fils de nœud faire
    valeur := max(valeur, -Alpha-Beta_NegaMax (fils, profondeur − 1, −β, −α, −couleur))
    α := max(α, valeur)
    si α ≥ β alors
        retourner valeur
retourner valeur
```

)

2.1.4.2. Implémentation en Python

```
def alphabeta_negamax(self, depth, time_left, color, heuristic, alpha, beta):
    if depth == 0 or time_left < 0 or self.go:
        return self.heuristic_choice(heuristic), []

    best_value = -math.inf
    time_begin = time.perf_counter()
    best_move = []

    for x, y in self.shadow_pawn:
        sim_board = self.copy()
        sim_board.make_move(x, y)
        child_value,_ = sim_board.negamax(depth - 1, time_left - (time.perf_counter()
- time_begin), -color, heuristic, -beta, -alpha)
        if -child_value > best_value:
            best_value = -child_value
            best_move = [x, y]
            alpha = max(alpha, best_value)
            if beta <= alpha:
                break

    return best_value, best_move
```

2.2. Heuristiques

2.2.1. Comparaison du nombre de pions

Cette heuristique consiste à comparer le nombre de pions de chaque joueur sur le plateau de jeu. Plus un joueur a de pions, plus il est en position de force. Voici l'implémentation de cette heuristique en Python :

```
def count_pawn(self):
    black = np.where(self.board == self.B)
    white = np.where(self.board == self.W)

    return len(white[0]) - len(black[0])
```

2.2.2. Matrice d'évaluation 1

Cette heuristique utilise une matrice d'évaluation pour évaluer le plateau de jeu. Cette matrice attribue une valeur à chaque case du plateau en fonction de sa position. Voici l'implémentation de cette heuristique en Python :

```
def compare_point_board1(self):
    black = np.where(self.board == self.B)
    white = np.where(self.board == self.W)
    black_score = np.sum(self.point_board1[black[0], black[1]])
    white_score = np.sum(self.point_board1[white[0], white[1]])

    return white_score - black_score
```

Avec la matrice déclarée au préalable :

```
self.point_board1 = np.matrix([[500, -150, 30, 10, 10, 30, -150, 500],
                               [-150, -250, 0, 0, 0, 0, -250, -150],
                               [30, 0, 1, 2, 2, 1, 0, 30],
                               [10, 0, 2, 16, 16, 2, 0, 10],
                               [10, 0, 2, 16, 16, 2, 0, 10],
                               [30, 0, 1, 2, 2, 1, 0, 30],
                               [-150, -250, 0, 0, 0, 0, -250, -150],
                               [500, -150, 30, 10, 10, 30, -150, 500]])
```

2.2.3. Matrice d'évaluation 2

Comme pour la matrice d'évaluation 1, cette heuristique utilise une matrice d'évaluation pour évaluer le plateau de jeu. Voici l'implémentation de cette heuristique en Python :

```
def compare_point_board2(self):
    black = np.where(self.board == self.B)
    white = np.where(self.board == self.W)
    black_score = np.sum(self.point_board2[black[0], black[1]])
    white_score = np.sum(self.point_board2[white[0], white[1]])

    return white_score - black_score
```

Avec la matrice déclarée au préalable :

```
self.point_board2 = np.matrix([[100, -20, 10, 5, 5, 10, -20, 100],
                               [-20, -50, -2, -2, -2, -2, -50, -20],
                               [10, -2, -1, -1, -1, -1, -2, 10],
```

```
        [5, -2, -1, -1, -1, -1, -2, 5],
        [5, -2, -1, -1, -1, -1, -2, 5],
        [10, -2, -1, -1, -1, -1, -2, 10],
        [-20, -50, -2, -2, -2, -2, -50, -20],
        [100, -20, 10, 5, 5, 10, -20, 100]])
```

3. Validation

Pour valider nos intelligences artificielles, nous avons mis en place une série de tests mettant à l'épreuve chacun des algorithmes mis en place. Chaque algorithme de recherche effectue un certain nombre de parties contre une autre IA simulant des coups joués aléatoirement sur la grille parmi les coups possibles. Cette IA Aléatoire est nécessaire car les IA que nous avons développées sont déterministes, donc si on en faisait jouer une contre une autre, la même partie se jouerait en boucle. Nous procédons ensuite à une analyse statistique et graphique des temps de calcul (temps moyen pour jouer un coup, temps du plus lent coup joué) et du pourcentage de victoire pour déterminer quel algorithme fournit les meilleurs résultats.

3.1. Implémentation de l'algorithme de jeu aléatoire en Python

```python
def random_ai(self, color):
    if (self.current_player == (-color)):
        return self.board
    x, y = self.shadow_pawn[np.random.randint(len(self.shadow_pawn))]
    self.make_move(x, y)
    return self.board
```

3.2. Tests

Nous avons fait différents tests statistiques puis nous les avons représenté à l'aide de graphiques pour comparer les performances des différents algorithmes de recherche. Ces graphiques nous les avons réalisés à l'aide de la bibliothèque QUICKCHART.IO de Python.

3.2.1. Comparaison des algorithmes de recherche

Nous avons effectué 50 parties pour chaque algorithme de recherche contre l'IA aléatoire d'abord en profondeur 1 puis en profondeur 2. L'heuristique choisie est la mixte : d'abord la matrice d'évaluation 1 puis la comparaison du nombre de pions au quarantième coup joué.

3.2.1.1. Profondeur 1

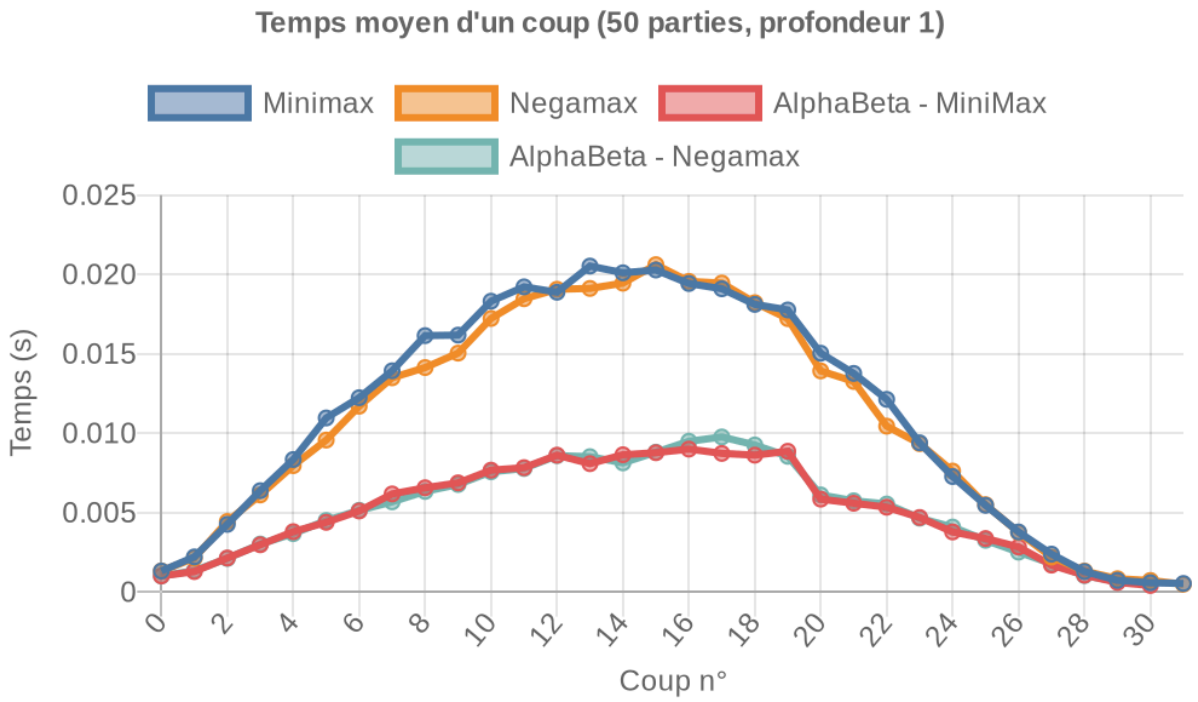

Fig. 4. – Temps moyen de recherche à un numéro de coup donné (profondeur 1, heuristique mixte)

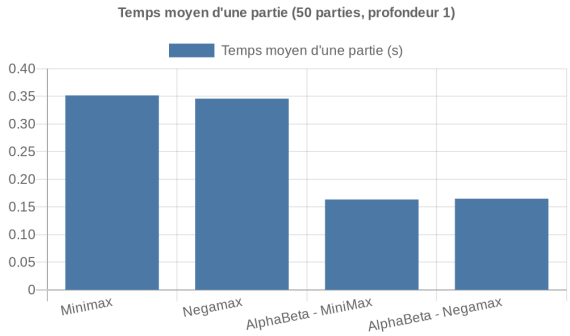

Fig. 5. – Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 1, heuristique mixte)

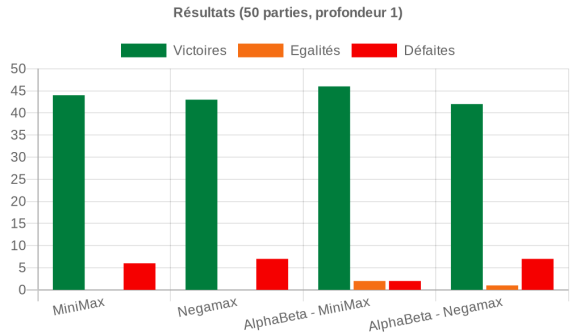

Fig. 6. – Nombre de victoire en fonction de l'algorithme de recherche (profondeur 1, heuristique mixte)

Ce premier test nous permet déjà de voir un écart entre les algorithmes MiniMax, NegaMax et leurs variantes Alpha-Beta respectives. Il est difficile de conclure quoi que ce soit avec une profondeur de 1, intéressons-nous donc à la profondeur 2.

3.2.1.2. Profondeur 2

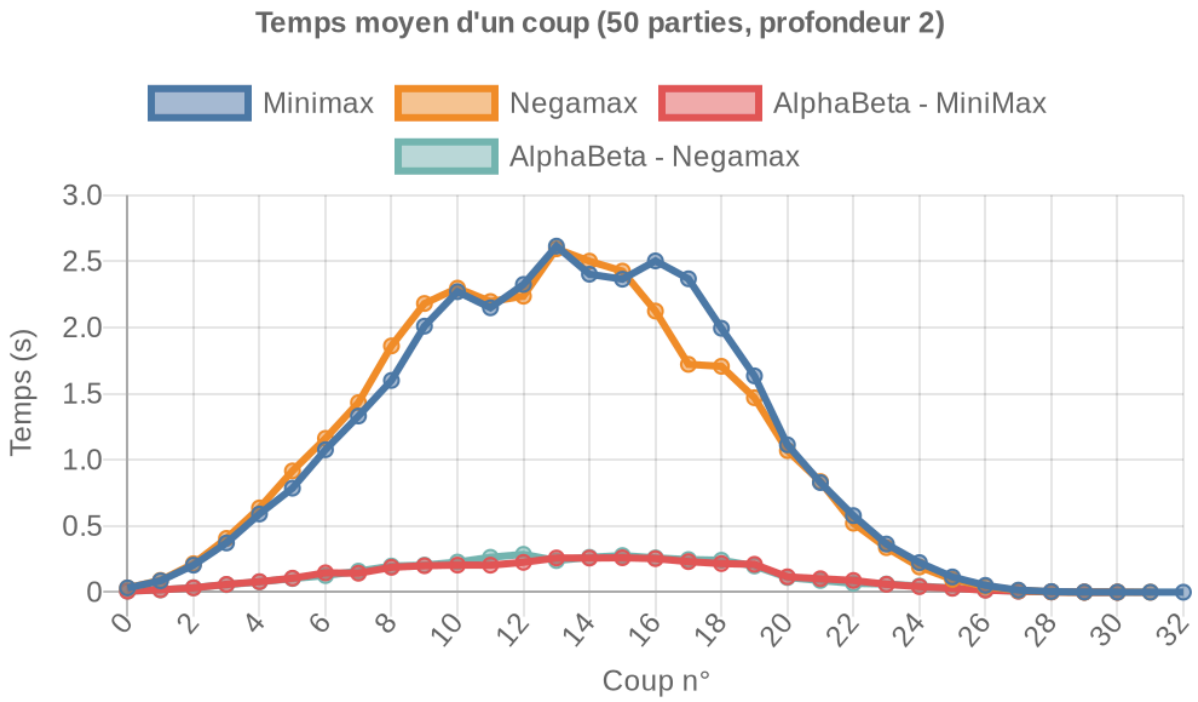

Fig. 7. – Temps moyen de recherche à un numéro de coup donné (profondeur 2, heuristique mixte)

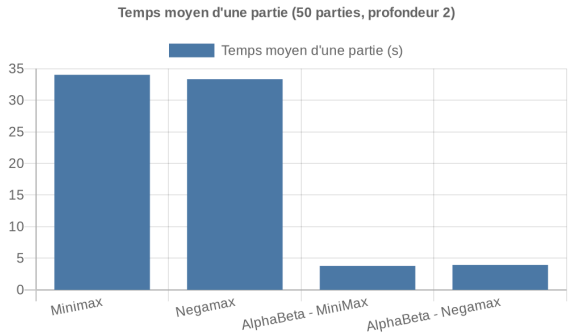

Fig. 8. – Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 2, heuristique mixte)

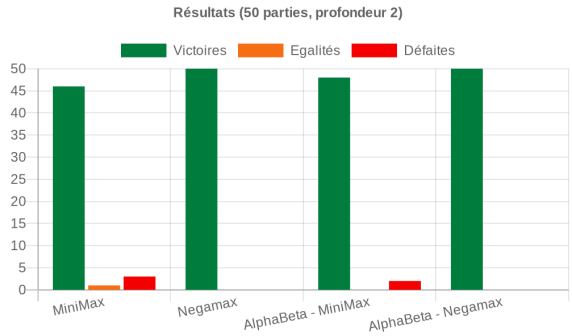

Fig. 9. – Nombre de victoire en fonction de l'algorithme de recherche (profondeur 2, heuristique mixte)

Nous pouvons voir que l'algorithme Alpha-Beta (MiniMax) est le plus performant des quatre algorithmes testés. En effet, il a le temps de calcul moyen le plus faible et le pourcentage de victoire le plus élevé. Ce résultat est étonnant car en théorie l'algorithme Alpha-Beta (NegaMax) devrait être plus performant. Nous pouvons au contraire constater que l'algorithme NegaMax ets plus performant que l'algorithme MiniMax. Nous avons donc fait d'autres tests pour être sûr de nos résultats.

3.2.1.3. Comparaison des algorithmes Alpha-Beta (MiniMax) et Alpha-Beta (NegaMax)

Nous avons effectué 200 parties pour chaque algorithme de recherche contre l'IA aléatoire en profondeur 1 et 2. L'heuristique choisie est la mixte : d'abord la matrice d'évaluation 1 puis la comparaison du nombre de pions au quarantième coup joué. Nous ne nous intéressons ici qu'au temps moyen pour un coup joué et au temps moyen pour une partie.

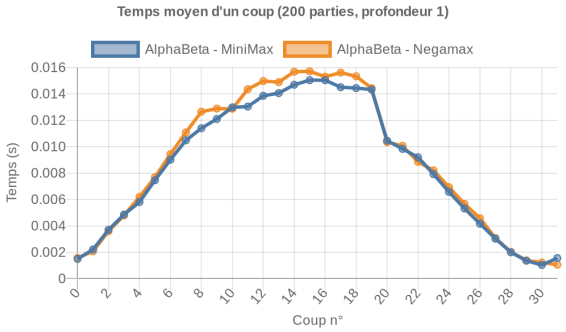

Fig. 10. – Temps moyen de recherche à un numéro de coup donné (profondeur 1, heuristique mixte)

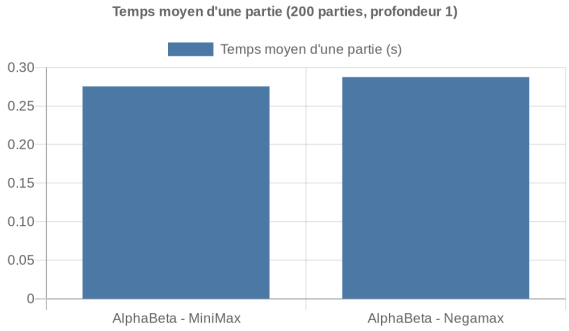

Fig. 11. – Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 1, heuristique mixte)

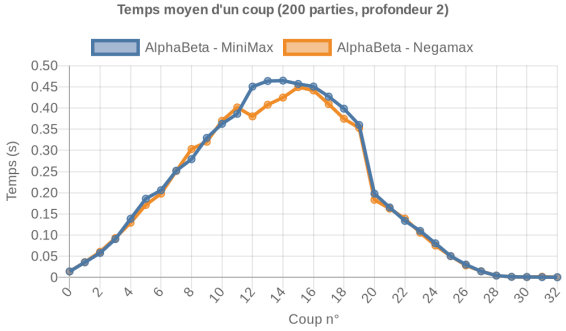

Fig. 12. – Temps moyen de recherche à un numéro de coup donné (profondeur 2, heuristique mixte)

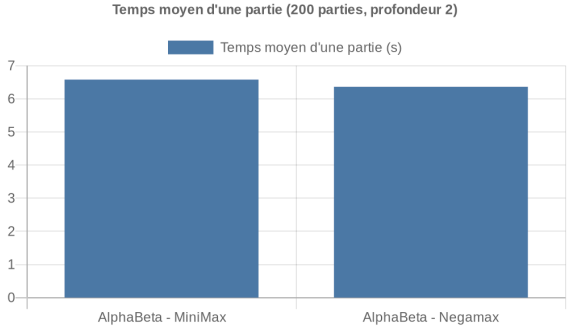

Fig. 13. – Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 2, heuristique mixte)

Nous pouvons constaté avec plus de tests que l'algorithme Alpha-Beta (NegaMax) est plus performant que l'algorithme Alpha-Beta (MiniMax) lorsque la profondeur de recherche est plus importante.

3.2.2. Comparaison des heuristiques

Nous avons effectué 50 parties pour chaque heuristique contre l'IA aléatoire en profondeur 2. L'algorithme de recherche utilisé est l'Alpha-Beta (NegaMax).

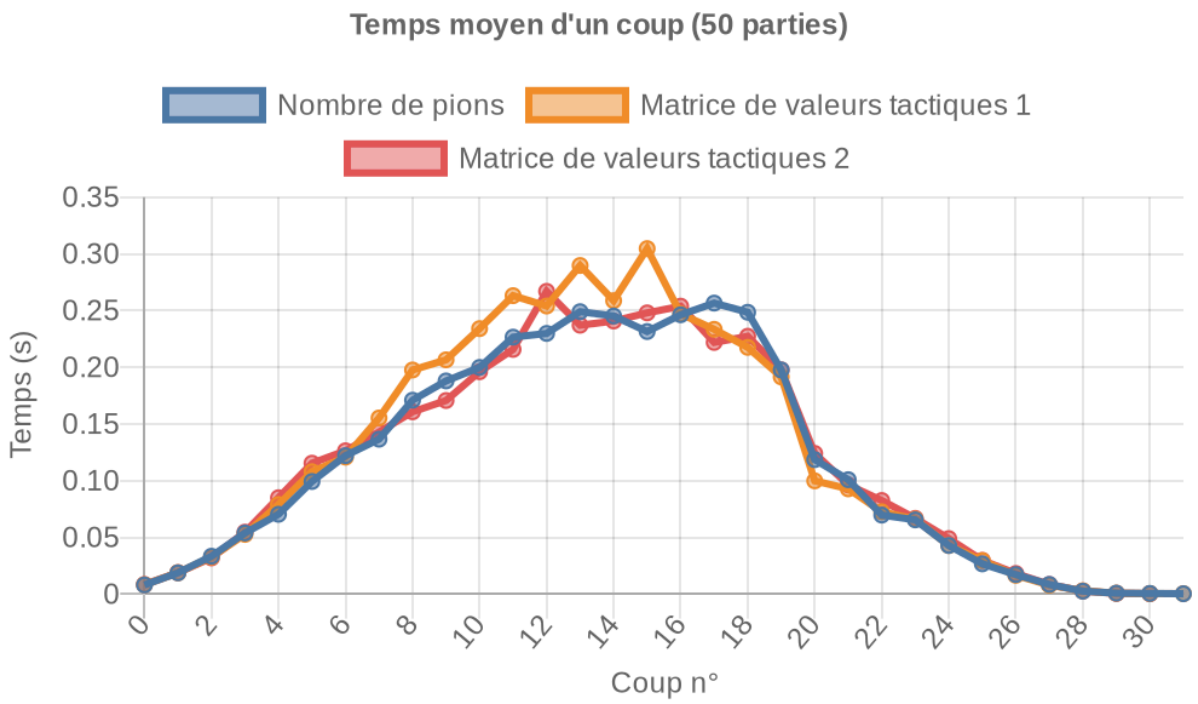

Fig. 14. – Temps moyen de recherche à un numéro de coup donné (profondeur 2, Alpha-Beta (NegaMax))

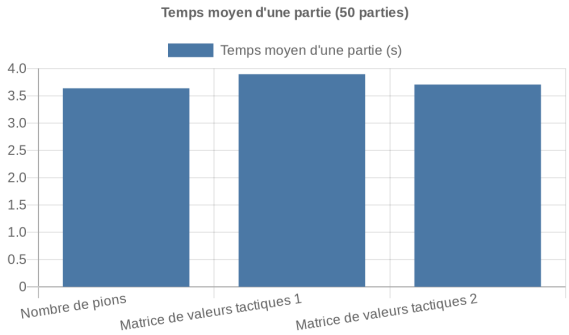

Fig. 15. – Temps moyen d'une partie en fonction de l'heuristique utilisée (profondeur 2, Alpha-Beta (NegaMax))

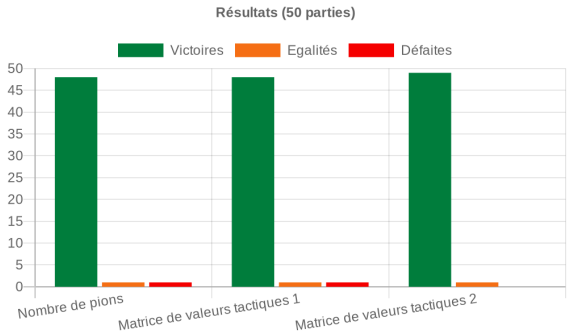

Fig. 16. – Nombre de victoire en fonction de l'heuristique utilisée (profondeur 2, Alpha-Beta (NegaMax))

Nous pouvons voir que ces heuristiques ont des résultats très similaire avec tout de même une légère avance pour la deuxième matrice d'évaluation. La différence est cependant très faible et ne justifie pas un changement d'heuristique. Nous avons donc décidé de continuer à utiliser la première matrice d'évaluation pour la suite de nos tests.

3.2.3. Comparaison des profondeurs de recherche

Nous avons effectué 50 parties pour chaque profondeur de recherche contre l'IA aléatoire. L'algorithme de recherche utilisé est l'Alpha-Beta (NegaMax) avec l'heuristique mixte.

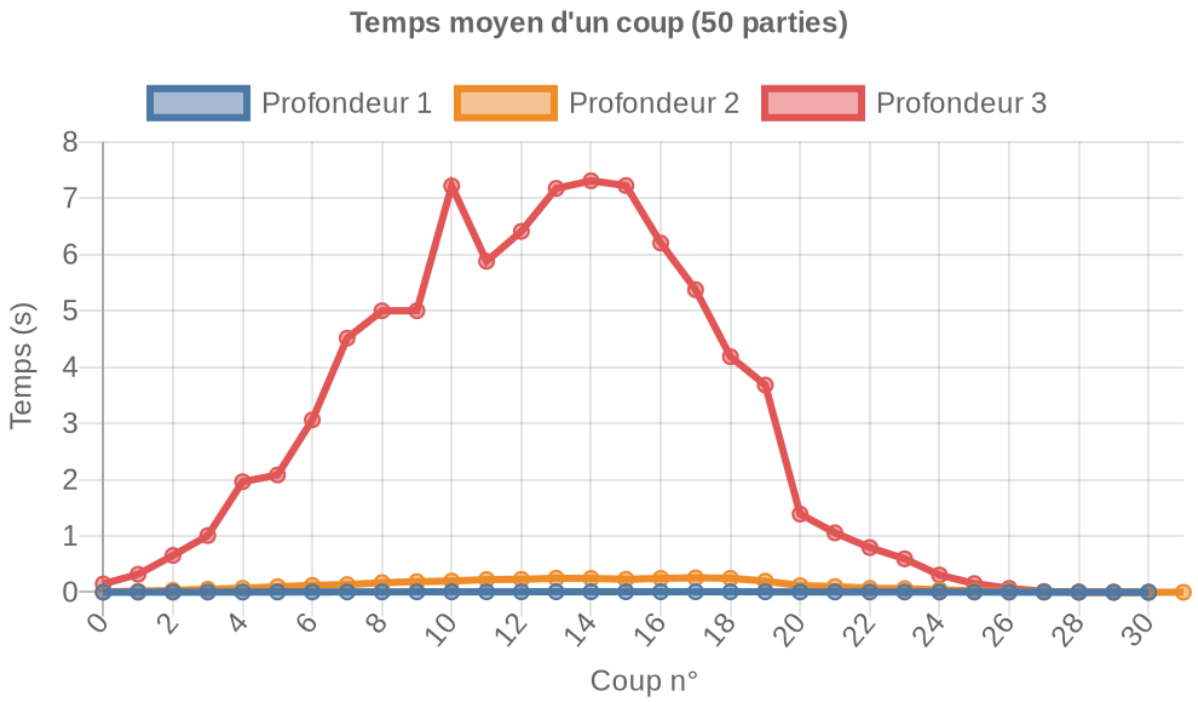

Fig. 17. – Temps moyen de recherche à un numéro de coup donné en fonction de la profondeur de recherche (Alpha-Beta (NegaMax), heuristique mixte)

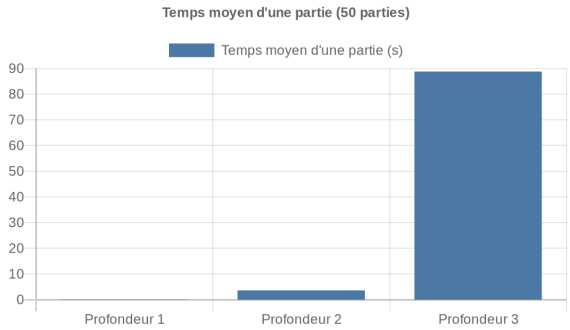

Fig. 18. – Temps moyen d'une partie en fonction de la profondeur de recherche (Alpha-Beta (NegaMax), heuristique mixte)

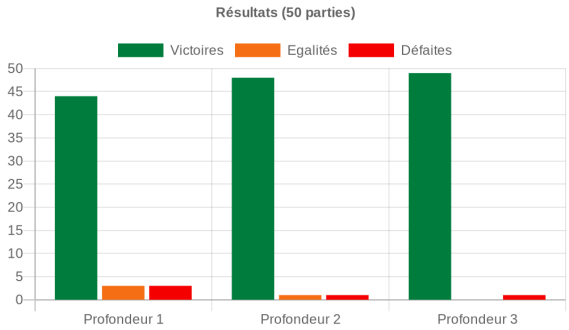

Fig. 19. – Nombre de victoire en fonction de la profondeur de recherche (Alpha-Beta (NegaMax), heuristique mixte)

Sans surprise, nous pouvons voir que plus la profondeur de recherche est grande, plus le temps de calcul est long et plus le pourcentage de victoire est élevé. La profondeur 2 est suffisante si nous voulons un jeu rapide. Si nous faisons face à un adversaire plus coriace, il est préférable de choisir une profondeur de 3 : même si le temps de calcul est plus long il n'est pas plus lent qu'un être humain.

3.2.4. Comparaison heuristique mixte et heuristique non mixte

Nous avons effectué 50 parties pour chaque heuristique contre l'IA aléatoire en profondeur 2. L'algorithme de recherche utilisé est l'Alpha-Beta (NegaMax). Les heuristiques comparées sont la mixte (matrice d'évaluation 1 puis nombre de pions) et la matrice d'évaluation 1.

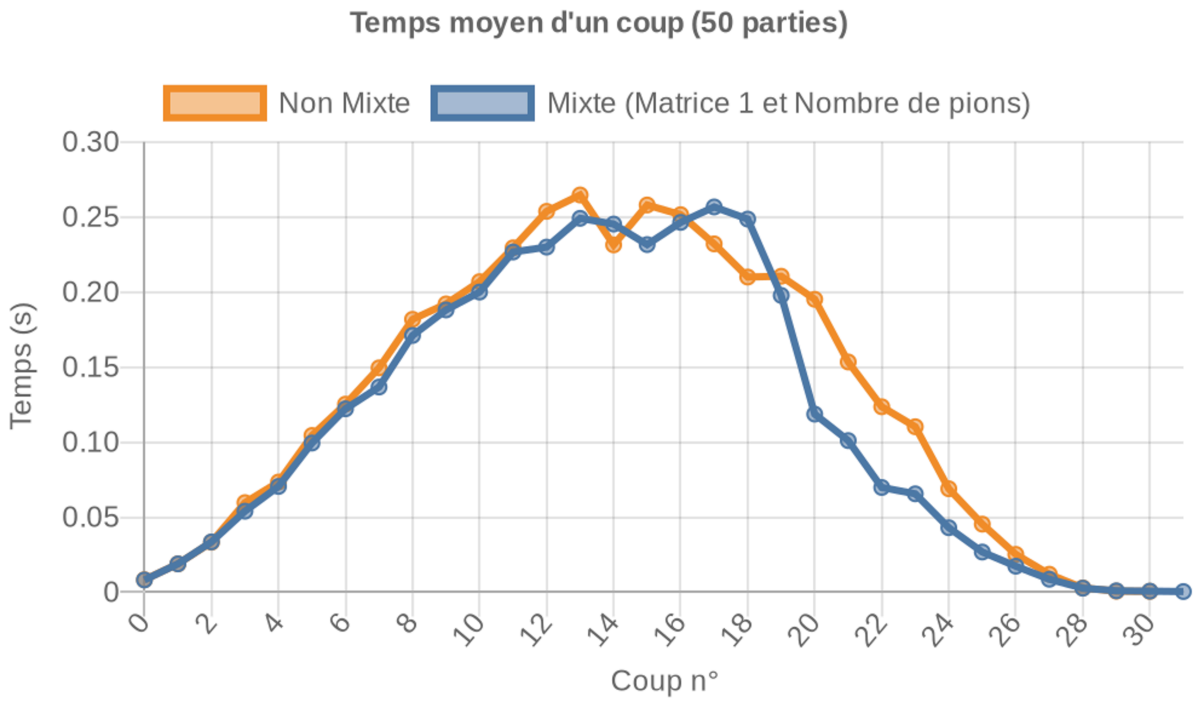

Fig. 20. – Temps moyen de recherche à un numéro de coup donné (profondeur 2, Alpha-Beta (NegaMax))

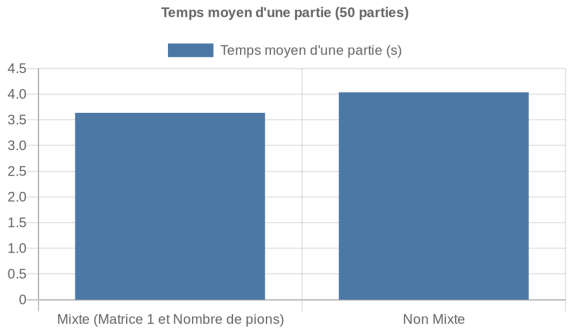

Fig. 21. – Temps moyen d'une partie en fonction de l'heuristique utilisée (profondeur 2, Alpha-Beta (NegaMax))

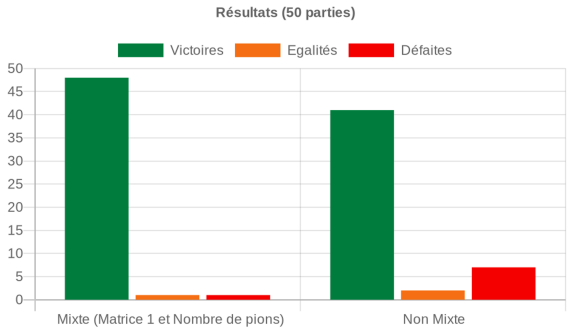

Fig. 22. – Nombre de victoire en fonction de l'heuristique utilisée (profondeur 2, Alpha-Beta (NegaMax))

L'heuristique mixte permet d'avoir un temps de calcul plus faible et un pourcentage de victoire plus élevé que l'heuristique non mixte. En effet, les matrices d'évaluation sont adaptées au début et au milieu de partie tandis que compter le nombre de pions est plus adapté à la fin de partie. Cela permet à l'IA de mieux « s'adapter » à la situation de jeu et de prendre des décisions plus pertinentes.

3.2.5. Comparaison avec mémoire et sans mémoire

Nous avons effectué 50 parties pour chaque algorithme de recherche contre l'IA aléatoire en profondeur 3. L'algorithme de recherche utilisé est l'Alpha-Beta (NegaMax). Nous n'avons pas réussi à mettre en place la mémorisation des coups joués, voici les résultats de notre tentative qui s'est soldée par un échec.

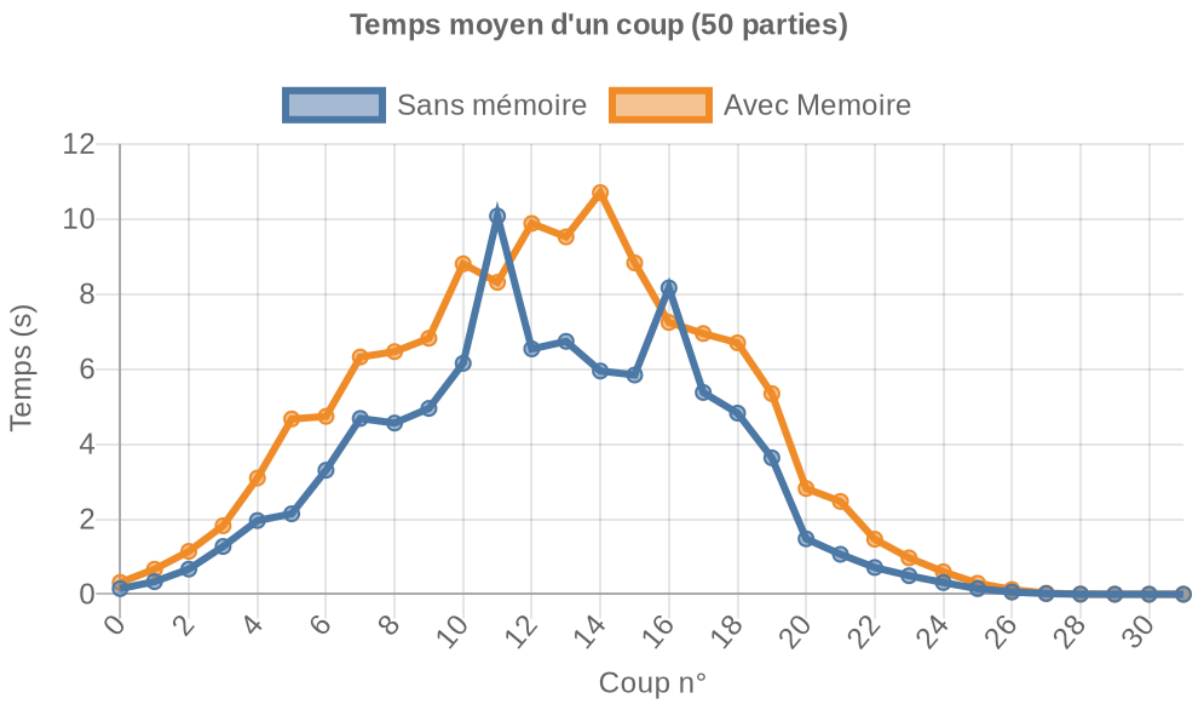

Fig. 23. – Temps moyen de recherche à un numéro de coup donné (profondeur 3, Alpha-Beta (NegaMax))

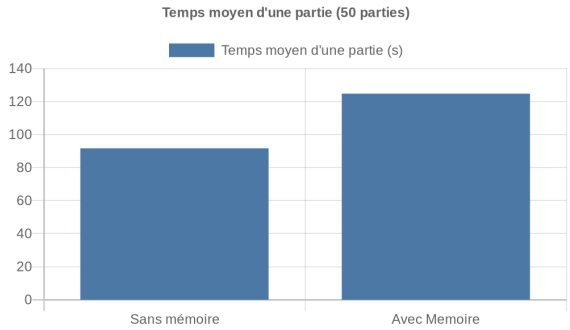

Fig. 24. – Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 3, Alpha-Beta (NegaMax))

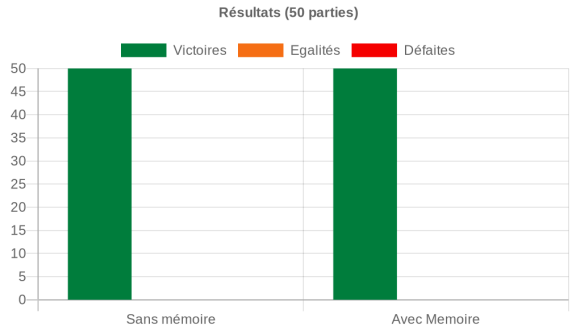

Fig. 25. – Nombre de victoire en fonction de l'algorithme de recherche (profondeur 3, Alpha-Beta (NegaMax))

L'IA est moins performante avec notre tentative de mise en place de mémoire.

3.3. Bilan

Notre analyse statistique des résultats montre que l'algorithme Alpha-Beta (NegaMax) est le plus performant des quatre algorithmes testés lorsque la profondeur de recherche est importante. De plus, nous pouvons déduire que l'heuristique la plus pertinente est la mixte qui utilise la deuxième matrice d'évaluation en début de partie et la comparaison du nombre de pions en fin de partie. Enfin, nous avons constaté que la profondeur de recherche a un impact significatif sur les performances de l'IA : plus la profondeur est grande, plus le temps de calcul est long et plus le pourcentage de victoire est élevé.

4. Discussion

4.1. Mémorisation des coups

Pour améliorer les performances de nos intelligences artificielles, nous avons tenté la mise en place un système de mémorisation des coups joués. En effet, pour chaque plateau de jeu, nous avons stocké les coups possibles pour le joueur courant. Cela devait nous permettre de réduire le temps de calcul en évitant de recalculer les coups possibles à chaque itération de l'algorithme de recherche. Cependant après maintes tentatives pour changer la structure de notre code, ou modifier la méthode de construction des algorithmes, nous n'avons pas réussi à obtenir de résultats concluants. Voici une des tentatives de mise en place de cette mémorisation des coups qu'il est possible de retrouver parmi les algorithmes de tests transmis dans le dossier du projet:

```
class Memorization:
        def __init__(self):
            self.table = {}

        def lookup(self, key):
            return self.table.get(str(key))

        def store(self, key, value, alpha, beta):
            self.table[key] = (value, alpha, beta)

def __init__(self):
        self.memorization = Memorization()

def generate_key(self):
        board_tuple = tuple(tuple(row) for row in self.board)
        return (board_tuple, self.current_player)

    # alpha-beta negamax algorithm
    def alphabeta_negamax2(self, depth, time_left, color, heuristic, alpha, beta):
        if depth == 0 or time_left < 0 or self.go:
            return self.heuristic_choice(heuristic) * color, []

        key = self.generate_key()
        stored_value = self.memorization.lookup(key)
        if stored_value is not None:
            value, stored_alpha, stored_beta = stored_value
            if stored_alpha <= alpha and stored_beta >= beta:
                return value, None  # Return stored value

        best_value = -math.inf
        time_begin = time.perf_counter()
        best_move = []

        for x, y in self.shadow_pawn:
            sim_board = self.copy()
            sim_board.make_move(x, y)
                child_value,_ = sim_board.alphabeta_negamax2(depth - 1, time_left -
(time.perf_counter() - time_begin), -color, heuristic, -beta, -alpha)
```

```
            if -child_value > best_value:
                best_value = -child_value
                best_move = [x, y]
                alpha = max(alpha, best_value)
                if alpha >= beta:
                    break
        # print("Coups possibles : ", self.shadow_pawn, "Meilleur coup : ", best_move)

        return best_value, best_move
```

4.2. Heuristiques

Pendant ce projet, nous n'avons testé que des heuristiques simples, vues en cours. Il serait intéressant de tester des heuristiques plus complexes pour voir si elles permettent d'améliorer les performances des intelligences artificielles. Par exemple, on pourrait imaginer une heuristique qui prend en compte le nombre de coups possible pour l'adversaire et qui permettrait à l'IA de bloquer l'adversaire.

4.3. Profondeur

Nous avons essayer d'optimiser notre code au maximum mais nous n'avons pas réussi à dépasser une profondeur de 3 avec l'algorithme Alpha-Beta (NegaMax) en un temps raisonnable. Une profondeur plus grande rend le jeu très lent et peu agréable à jouer.

4.4. Interface graphique

Nous avons donc mis en place une interface graphique pour jouer à Othello. Pour lancer celle-ci, il suffit de lancer le fichier `application.py` puis de se rendre sur l'adresse `http://localhost:5000/` dans un navigateur web. L'interface graphique permet de jouer contre une IA ou de simuler une partie entre une IA et une IA aléatoire. Voici à quoi ressemble l'interface graphique :

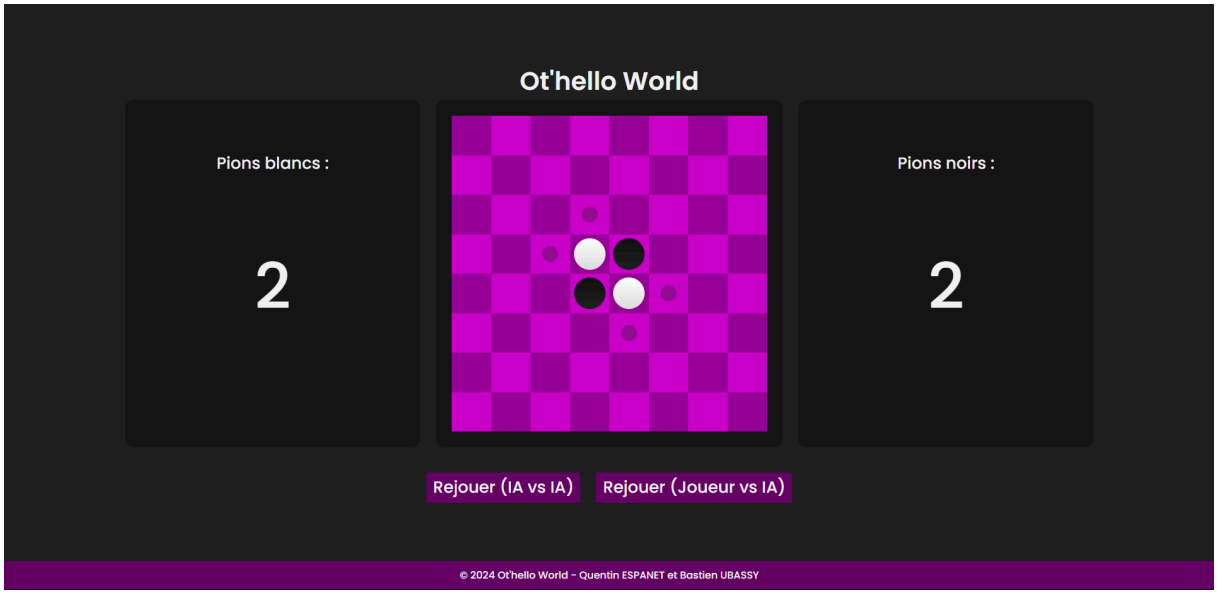

Fig. 26. – Interface graphique du jeu Ot'hello World

5. Conclusion

Nous avons pu constater que l'algorithme Alpha-Beta (NegaMax) est le plus performant des quatre algorithmes testés. En effet, il a le temps de calcul moyen le plus faible et le pourcentage de victoire le plus élevé. Cela s'explique par le fait que l'algorithme Alpha-Beta (NegaMax) permet de réduire le nombre de nœuds explorés en élaguant les branches inutiles de l'arbre de recherche.

Nous avons également constaté que l'heuristique mixte qui utilise une matrice d'évaluation en début de partie et la comparaison du nombre de pions en fin de partie est la plus performante des trois heuristiques testées.

En conclusion, ce projet nous a permis de mieux comprendre le fonctionnement des intelligences artificielles et des algorithmes de recherche. Nous avons pu mettre en pratique nos connaissances en intelligence artificielle et en programmation pour développer des intelligences artificielles capables de jouer à Othello. Nous avons également pu tester différentes heuristiques et différents algorithmes de recherche pour trouver le meilleur coup à jouer. Ce projet nous a permis d'acquérir de nouvelles compétences en intelligence artificielle et en programmation.

6. Annexes

6.1. Bibliothèques Python externes utilisées

- numpy : pour manipuler des tableaux et des matrices
- quickchart.io : pour réaliser des graphiques
- flask : pour l'interface graphique

Table des figures

Fig. 1: Ot'hello World (Capture d'écran) 1
Fig. 2: Première matrice d'évaluation 2
Fig. 3: Seconde matrice d'évaluation 3
Fig. 4: Temps moyen de recherche à un numéro de coup donné (profondeur 1, heuristique mixte) .. 10
Fig. 5: Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 1, heuristique mixte) 10
Fig. 6: Nombre de victoire en fonction de l'algorithme de recherche (profondeur 1, heuristique mixte) 10
Fig. 7: Temps moyen de recherche à un numéro de coup donné (profondeur 2, heuristique mixte) .. 11
Fig. 8: Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 2, heuristique mixte) 11
Fig. 9: Nombre de victoire en fonction de l'algorithme de recherche (profondeur 2, heuristique mixte) 11
Fig. 10: Temps moyen de recherche à un numéro de coup donné (profondeur 1, heuristique mixte) .12
Fig. 11: Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 1, heuristique mixte) 12
Fig. 12: Temps moyen de recherche à un numéro de coup donné (profondeur 2, heuristique mixte) .12
Fig. 13: Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 2, heuristique mixte) 12
Fig. 14: Temps moyen de recherche à un numéro de coup donné (profondeur 2, Alpha-Beta (NegaMax)) 13
Fig. 15: Temps moyen d'une partie en fonction de l'heuristique utilisée (profondeur 2, Alpha-Beta (NegaMax)) 13
Fig. 16: Nombre de victoire en fonction de l'heuristique utilisée (profondeur 2, Alpha-Beta (NegaMax)) 13
Fig. 17: Temps moyen de recherche à un numéro de coup donné en fonction de la profondeur de recherche (Alpha-Beta (NegaMax), heuristique mixte) 14
Fig. 18: Temps moyen d'une partie en fonction de la profondeur de recherche (Alpha-Beta (NegaMax), heuristique mixte) 14
Fig. 19: Nombre de victoire en fonction de la profondeur de recherche (Alpha-Beta (NegaMax), heuristique mixte) 14
Fig. 20: Temps moyen de recherche à un numéro de coup donné (profondeur 2, Alpha-Beta (NegaMax)) 15
Fig. 21: Temps moyen d'une partie en fonction de l'heuristique utilisée (profondeur 2, Alpha-Beta (NegaMax)) 15
Fig. 22: Nombre de victoire en fonction de l'heuristique utilisée (profondeur 2, Alpha-Beta (NegaMax)) 15
Fig. 23: Temps moyen de recherche à un numéro de coup donné (profondeur 3, Alpha-Beta (NegaMax)) 16
Fig. 24: Temps moyen d'une partie en fonction de l'algorithme de recherche (profondeur 3, Alpha-Beta (NegaMax)) 16
Fig. 25: Nombre de victoire en fonction de l'algorithme de recherche (profondeur 3, Alpha-Beta (NegaMax)) 16
Fig. 26: Interface graphique du jeu Ot'hello World 18