# Projet IA — Puissance 4
# Rapport technique

Étudiant : ENSISA – 2ème année Informatique et Réseaux

22 janvier 2026

## Résumé

Ce rapport présente l'implémentation d'une intelligence artificielle pour le jeu Puissance 4 basée sur l'algorithme Minimax optimisé par élagage Alpha-Beta et intégrée dans une interface Tkinter. Le document décrit l'architecture du code, le fonctionnement de l'algorithme, la stratégie d'évaluation utilisée, les optimisations de performance incluses et propose des axes d'amélioration. L'approche privilégie la clarté et les décisions techniques reproductibles.

## 1 Introduction

L'objectif du projet est d'implémenter un adversaire jouant au Puissance 4 et paramétrable par niveau, où le niveau correspond à la profondeur de recherche de l'algorithme. L'IA doit s'exécuter sans bloquer l'interface graphique et fournir des décisions raisonnables dans un temps acceptable.

## 2 Architecture du code

Le code est organisé autour de deux classes principales ainsi que d'un ensemble de fonctions globales dédiées à l'algorithme de recherche. La classe `Board` encapsule l'état de la grille, les opérations élémentaires (ajout d'un jeton, copie, réinitialisation, détection de victoire) et la fonction heuristique d'évaluation d'une position. La classe `Connect4` orchestre la partie et l'interface graphique Tkinter : elle gère la boucle de jeu, les interactions utilisateur et lance l'IA dans un thread séparé pour éviter le blocage de l'interface. L'algorithme Minimax avec élagage Alpha-Beta est implémenté par les fonctions `max_value` et `min_value` ; la décision finale de l'IA est prise dans `alpha_beta_decision`.

## 3 Algorithme Minimax / Alpha-Beta

L'algorithme suit l'implémentation classique. Chaque appel teste la condition d'arrêt (profondeur nulle ou position terminale). Si la position n'est pas terminale et que la profondeur est encore positive, les fonctions explorent les coups possibles, simulent le coup sur une copie de la grille et propagent les valeurs via les appels récursifs. L'assemblage des bornes $\alpha$ et $\beta$ permet de couper des branches non pertinentes et d'accélérer la recherche. La profondeur de recherche (`depth`) est le paramètre de difficulté : un niveau plus élevé explore plus profondément l'arbre des coups et,

en principe, devrait prendre des décisions tactiques plus pertinentes. Voici l'extrait clé montrant la structure de `max_value` :

```
def max_value(board, depth, alpha, beta, player):
    if depth == 0 or board.check_victory() != 0:
        return board.eval(player)
    possible_moves = board.get_possible_moves()
    if not possible_moves:
        return 0
    v = -float('inf')
    for move in possible_moves:
        new_board = board.copy()
        new_board.add_disk(move, player, update_display=False)
        v = max(v, min_value(new_board, depth - 1, alpha, beta, 3 -
            player))
        if v >= beta:
            return v
        alpha = max(alpha, v)
    return v
```

La gestion de la profondeur est explicite : chaque niveau décrémente la profondeur et la terminaison renvoie une évaluation heuristique. Pour favoriser les victoires rapides, il est recommandé d'ajouter une légère pondération en fonction de la profondeur restante au moment où l'on détecte une position gagnante, afin que des recherches plus profondes préfèrent des gains rapides.

# 4 Stratégie d'évaluation (cœur du rapport)

La stratégie d'évaluation déployée dans la classe `Board` consiste à parcourir toutes les "fenêtres" de 4 cellules possibles (horizontales, verticales et diagonales) et à sommer une valeur heuristique calculée localement pour chaque fenêtre. Au préalable la fonction teste si la position est terminale (victoire pour un joueur) et renvoie une valeur forte positive pour une victoire du joueur évalué ou une valeur forte négative si l'adversaire a gagné. Si la position n'est pas terminale, la méthode parcourt chaque fenêtre et appelle `evaluate_window`. Le code réel est le suivant :

```
def eval(self, player):
    winner = self.check_victory()
    if winner == player:
        return 1000
    elif winner != 0:
        return -1000
    else:
        score = 0
        for row in range(6):
            for col in range(4):
                window = [self.grid[col + i][row] for i in range(4)]
                score += self.evaluate_window(window, player)
        for col in range(7):
            for row in range(3):
                window = [self.grid[col][row + i] for i in range(4)]
                score += self.evaluate_window(window, player)
        for col in range(4):
            for row in range(3):
                window = [self.grid[col + i][row + i] for i in range(4)]
                score += self.evaluate_window(window, player)
        for col in range(4):
            for row in range(3, 6):
```

```
                window = [self.grid[col + i][row - i] for i in range(4)]
                score += self.evaluate_window(window, player)
        return score
```

La fonction `evaluate_window` compte le nombre de jetons du joueur, le nombre de cases vides et le nombre de jetons adverses dans une fenêtre de 4. Elle attribue ensuite un poids selon ces comptes. Le code actuel est le suivant :

```
def evaluate_window(self, window, player):
    score = 0
    opponent = 3 - player
    player_count = window.count(player)
    empty_count = window.count(0)
    opponent_count = window.count(opponent)
    if player_count == 4:
        score += 100
    elif player_count == 3 and empty_count == 1:
        score += 10
    elif player_count == 2 and empty_count == 2:
        score += 2
    if opponent_count == 3 and empty_count == 1:
        score -= 80
    return score
```

L'interprétation de ces choix est la suivante. Un alignement complet de quatre pions est traité comme une condition de victoire (valeur forte). Les alignements de trois pions avec une case libre reçoivent une valeur positive notable (+10) car ils constituent des menaces offensives proches, tandis que les paires reçoivent une valeur faible (+2) car elles sont moins déterminantes. La présence d'une menace adverse immédiate (trois pions adverses et une case vide) entraîne une pénalité forte (-80) afin de prioriser le blocage. Le code favorise également implicitement le contrôle du centre parce que la génération des coups considère la colonne centrale en priorité lors de l'énumération des coups possibles, ce qui oriente naturellement l'exploration vers le centre (une heuristique connue pour améliorer la force en Puissance 4).

Il est important de noter que l'heuristique actuelle ne distingue pas explicitement une ligne de trois "ouverte des deux côtés" d'une ligne de trois "bloquée d'un côté". La méthode se contente de comptages dans la fenêtre. Pour améliorer la finesse, on peut analyser la position exacte de la fenêtre (positions des extrémités) et augmenter la valeur d'une "trois ouvert" comparée à une "trois bloqué". Une autre amélioration efficace est d'introduire une pondération liée à la profondeur sur les positions terminales pour que les recherches plus profondes privilégient des victoires plus rapides.

## 5 Optimisations et parallélisation

Pour réduire le temps de décision, le code combine deux techniques : exécuter l'appel global à l'IA dans un thread afin de ne pas bloquer l'interface graphique, et paralléliser l'évaluation des coups du niveau racine en utilisant des travailleurs. Concrètement, `alpha_beta_decision` construit pour chaque coup candidat une copie de la grille après le coup puis appelle en parallèle une fonction qui reconstruit un objet `Board` à partir de cette grille et exécute une évaluation Minimax (à profondeur moins une). Voici l'extrait illustrant le principe :

```
def alpha_beta_decision(board, turn, ai_level, queue, max_player):
    possible_moves = board.get_possible_moves()
    if not possible_moves:
```

```
        queue.put(None)
        return

    num_processes = min(len(possible_moves), mp.cpu_count())

    args_list = []
    for move in possible_moves:
        new_board = board.copy()
        new_board.add_disk(move, max_player, update_display=False)
        args_list.append((new_board.grid.copy(), ai_level - 1,
           max_player, move))

    with mp.Pool(processes=num_processes) as pool:
        results = pool.map(evaluate_move_parallel, args_list)

    best_move = None
    best_value = -float('inf')
    for move, value in results:
        if value > best_value:
            best_value = value
            best_move = move

    queue.put(best_move)
```

La fonction worker reconstruit l'objet `Board` et appelle `min_value` pour produire la valeur de chaque coup. L'effet pratique est une réduction du temps de décision proportionnelle au nombre de cœurs disponibles pour l'évaluation des branches racines, surtout pour des profondeurs modestes. L'emploi d'un thread pour lancer `alpha_beta_decision` depuis `Connect4` évite par ailleurs que l'interface graphique ne gèle pendant le calcul. En contrepartie, la parallélisation nécessite des copies de la grille et des coûts de sérialisation (si l'on utilise des processus), ce qui peut réduire les gains pour des profondeurs très importantes.

## 6 Conclusion

Le code fourni met en œuvre une IA Minimax/Alpha-Beta correcte et modulaire, avec une fonction d'évaluation simple mais cohérente. Les choix d'heuristique donnent la priorité au blocage des menaces adverses tout en favorisant la construction d'alignements. Les optimisations présentes (thread pour l'UI et parallélisation des évaluations racines) améliorent la réactivité et réduisent le temps de décision sur des machines multi-cœurs. Les axes d'amélioration les plus efficaces sont l'ajout d'un ordre de parcours des coups (move ordering), la prise en compte de la profondeur dans l'évaluation terminale pour préférer des victoires rapides, et l'introduction d'une table de transposition. Ces modifications permettront d'augmenter sensiblement la force de l'IA sans complexifier excessivement la base de code.