

Projet IA — Puissance 4

Rapport technique

Léon SCHER - Dorian POELLEN

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 ainsi que des axes d'amélioration.

1 Introduction

L'objectif du projet est d'implémenter un adversaire jouant au Puissance 4 avec un niveau variable correspondant à la profondeur de recherche de l'algorithme. L'IA doit pouvoir fournir des coups pertinents 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 d'évaluation d'une position. La classe **Connect4** gère 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 α et β 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. Ci-dessous le code pour la fonction `max_value` :

```

1 def max_value(board, depth, alpha, beta, player):
2     if depth == 0 or board.check_victory() != 0:
3         return board.eval(player)
4     possible_moves = board.get_possible_moves()
5     if not possible_moves:
6         return 0
7     v = -float('inf')
8     for move in possible_moves:
9         new_board = board.copy()
10        new_board.add_disk(move, player, update_display=False)
11        v = max(v, min_value(new_board, depth - 1, alpha, beta, 3 -
12                             player))
13        if v >= beta:
14            return v
15        alpha = max(alpha, v)
16    return v

```

La gestion de la profondeur est explicite : chaque niveau décrémente la profondeur et la terminaison renvoie une évaluation heuristique.

4 Stratégie d'évaluation (cœur du rapport)

La stratégie d'évaluation décrite dans la classe Board consiste à parcourir toutes les "fenêtres" de 4 cases possibles (horizontales, verticales et diagonales) et à en déduire une valeur heuristique calculée localement pour chaque fenêtre. Avant toute chose, 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`.

```

1 def eval(self, player):
2     winner = self.check_victory()
3     if winner == player:
4         return 1000
5     elif winner != 0:
6         return -1000
7     else:
8         score = 0
9         for row in range(6):
10            for col in range(4):
11                window = [self.grid[col + i][row] for i in range(4)]
12                score += self.evaluate_window(window, player)
13            for col in range(7):
14                for row in range(3):
15                    window = [self.grid[col][row + i] for i in range(4)]
16                    score += self.evaluate_window(window, player)
17            for col in range(4):
18                for row in range(3):
19                    window = [self.grid[col + i][row + i] for i in range(4)]
20                    score += self.evaluate_window(window, player)
21            for col in range(4):
22                for row in range(3, 6):
23                    window = [self.grid[col + i][row - i] for i in range(4)]
24                    score += self.evaluate_window(window, player)
25        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 :

```
1 def evaluate_window(self, window, player):
2     score = 0
3     opponent = 3 - player
4     player_count = window.count(player)
5     empty_count = window.count(0)
6     opponent_count = window.count(opponent)
7     if player_count == 4:
8         score += 100
9     elif player_count == 3 and empty_count == 1:
10        score += 10
11    elif player_count == 2 and empty_count == 2:
12        score += 2
13    if opponent_count == 3 and empty_count == 1:
14        score -= 80
15    return score
```

L'interprétation de ces choix d'heuristiques est la suivante. Un alignement complet de quatre pions est traité comme une condition de victoire (valeur haute). Les alignements de trois pions avec une case libre reçoivent une valeur positive (+10), mais pas non plus trop importante par rapport à un alignement de 4 cases, car ils mettent en difficulté l'adversaire, 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 favorise naturellement l'exploration vers le centre (une stratégie connue et conseillée pour gagner au Puissance 4).

Il est important de noter que l'heuristique actuelle ne distingue pas 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'un alignement "trois ouvert" comparée à un alignement "trois bloqué".

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 "workers". Concrètement, `alpha_beta_decision` construit pour chaque futur coup 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 de code mettant en oeuvre cela :

```
1 def alpha_beta_decision(board, turn, ai_level, queue, max_player):
2     possible_moves = board.get_possible_moves()
3     if not possible_moves:
4         queue.put(None)
5         return
6
7     num_processes = min(len(possible_moves), mp.cpu_count())
8
```

```

9      args_list = []
10     for move in possible_moves:
11         new_board = board.copy()
12         new_board.add_disk(move, max_player, update_display=False)
13         args_list.append((new_board.grid.copy(), ai_level - 1,
14                             max_player, move))
15
16     with mp.Pool(processes=num_processes) as pool:
17         results = pool.map(evaluate_move_parallel, args_list)
18
19     best_move = None
20     best_value = -float('inf')
21     for move, value in results:
22         if value > best_value:
23             best_value = value
24             best_move = move
25
26     queue.put(best_move)

```

La fonction worker reconstruit l'objet `Board` et appelle `min_value` pour générer 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 basses. L'emploi d'un thread pour lancer `alpha_beta_decision` depuis `Connect4` évite par ailleurs que l'interface graphique ne freeze pendant le calcul du meilleur coup. 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 oeuvre une IA Minimax/Alpha-Beta, avec une fonction d'évaluation simple mais cohérente avec le jeu de Puissance 4. Les choix d'heuristique donnent la priorité au blocage des menaces adverses tout en favorisant la construction d'alignements et la prise de contrôle du centre du plateau. Les optimisations présentes (thread pour l'UI et parallélisation des évaluations racines) permettent de . 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.