期末黑白棋競賽

113321531 游志信

```python
import time

import numpy as np

from othello.OthelloGame import BLACK, WHITE
from othello.OthelloUtil import executeMove, getValidMoves


class AlphaBetaBot:
    def __init__(self, depth=10, time_limit=12):
        self.depth = depth  # 初始深度
        self.time_limit = time_limit  # 時間限制（秒）
        self.start_time = None  # 搜索開始時間

    def evaluateBoard(self, board, color):
        """評估函數總入口，整合多種策略的評估分數"""
        disk_difference_score = self.calculateDiskDifference(board, color)
        positional_score = self.calculatePositionalScore(board, color)
        corner_score = self.calculateCornerScore(board, color)
        corner_adjacent_penalty = self.calculateCornerAdjacentPenalty(board,
color)
        mobility_score = self.calculateMobilityScore(board, color)
        stability_score = self.calculateStabilityScore(board, color)
        edge_score = self.calculateEdgeScore(board, color)

        # 總分加權
        return (
            10 * disk_difference_score
            + positional_score
            + corner_score
            + corner_adjacent_penalty
            + mobility_score
            + stability_score
            + edge_score
        )

    def calculateDiskDifference(self, board, color):
        """計算棋子數量差距得分"""
        return np.sum(board == color) - np.sum(board == -color)
```

```
    def calculatePositionalScore(self, board, color):
        """計算棋盤位置加權分數"""
        WEIGHTS = np.array(
            [
                [120, -40, 20, 20, -40, 120],
                [-40, -60, -5, -5, -60, -40],
                [20, -5, 5, 5, -5, 20],
                [20, -5, 5, 5, -5, 20],
                [-40, -60, -5, -5, -60, -40],
                [120, -40, 20, 20, -40, 120],
            ]
        )
        return np.sum((board == color) * WEIGHTS) - np.sum((board == -color)
* WEIGHTS)

    def calculateCornerScore(self, board, color):
        """計算角落控制得分"""
        corners = [(0, 0), (0, 5), (5, 0), (5, 5)]
        score = 0
        for y, x in corners:
            if board[y, x] == color:
                score += 40
            elif board[y, x] == -color:
                score -= 40
        return score

    def calculateCornerAdjacentPenalty(self, board, color):
        """計算角落鄰接懲罰得分"""
        corner_adjacent = [
            (0, 1),
            (1, 0),
            (1, 1),
            (0, 4),
            (1, 5),
            (1, 4),
            (4, 0),
            (5, 1),
            (4, 1),
            (4, 4),
            (5, 4),
            (4, 5),
        ]
        penalty = 0
        for y, x in corner_adjacent:
            if board[y, x] == color:
```

```
                penalty -= 20
            elif board[y, x] == -color:
                penalty += 20
        return penalty

    def calculateMobilityScore(self, board, color):
        """計算行動力差距得分"""
        my_moves = len(getValidMoves(board, color))
        opponent_moves = len(getValidMoves(board, -color))
        return 10 * (my_moves - opponent_moves)

    def calculateStabilityScore(self, board, color):
        """計算穩定棋子得分"""
        stable_disks = self.countStableDisks(board, color)
        opponent_stable_disks = self.countStableDisks(board, -color)
        return 50 * (stable_disks - opponent_stable_disks)

    def calculateEdgeScore(self, board, color):
        """計算邊界控制得分"""
        edges = list(zip(*np.where(board != 0)))
        score = 0
        for y, x in edges:
            if y == 0 or y == 5 or x == 0 or x == 5:  # 邊界位置
                if board[y, x] == color:
                    score += 10
                elif board[y, x] == -color:
                    score -= 10
        return score

    def countStableDisks(self, board, color):
        """計算穩定棋子的數量"""
        stable_count = 0
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for y in range(board.shape[0]):
            for x in range(board.shape[1]):
                if board[y, x] == color and self.isStable(board, y, x,
directions):
                    stable_count += 1
        return stable_count

    def isStable(self, board, y, x, directions):
        """判斷棋子是否穩定"""
        for dy, dx in directions:
            ny, nx = y + dy, x + dx
            while 0 <= ny < board.shape[0] and 0 <= nx < board.shape[1]:
```

```
                if board[ny, nx] == 0:  # 空位
                    return False
                if board[ny, nx] == -board[y, x]:  # 對手棋子
                    return False
                ny += dy
                nx += dx
        return True

    def getAction(self, board, color):
        """主入口：搜索最佳步驟，限制在指定時間內完成"""
        self.start_time = time.time()  # 記錄搜索開始時間
        valid_moves = getValidMoves(board, color)

        if valid_moves.size == 0:
            return None  # 沒有合法步驟

        best_move = None
        best_score = float("-inf")

        # 迭代加深搜索
        for depth in range(1, self.depth + 1):
            try:
                # 搜索當前深度的最佳步驟
                current_move, current_score = self.searchWithDepth(board,
color, depth)
                if current_score > best_score:
                    best_move = current_move
                    best_score = current_score

                # 檢查是否超時
                if time.time() - self.start_time > self.time_limit:
                    break
            except TimeoutError:
                break

        return best_move

    def searchWithDepth(self, board, color, depth):
        """在指定深度內搜索最佳步驟"""
        valid_moves = getValidMoves(board, color)
        if valid_moves.size == 0:
            return None, float("-inf")  # 無步驟，分數最低

        best_move = None
        best_score = float("-inf")
```

```
        for move in valid_moves:
            new_board = board.copy()
            executeMove(new_board, color, move)
            score = self.alphabeta(
                new_board, depth - 1, float("-inf"), float("inf"), -color
            )

            if score > best_score:
                best_move = move
                best_score = score

            # 檢查是否超時
            if time.time() - self.start_time > self.time_limit:
                raise TimeoutError  # 超時結束搜索

        return best_move, best_score

    def alphabeta(self, board, depth, alpha, beta, color):
        """Alpha-Beta 搜索遞歸，帶時間控制"""
        if time.time() - self.start_time > self.time_limit:
            raise TimeoutError  # 超時立即返回

        valid_moves = getValidMoves(board, color)
        if valid_moves.size == 0 or depth == 0:
            return self.evaluateBoard(board, color)

        if color == BLACK:  # 最大化
            max_eval = float("-inf")
            for move in valid_moves:
                new_board = board.copy()
                executeMove(new_board, color, move)
                eval = self.alphabeta(new_board, depth - 1, alpha, beta,
WHITE)
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
            return max_eval
        else:  # 最小化
            min_eval = float("inf")
            for move in valid_moves:
                new_board = board.copy()
                executeMove(new_board, color, move)
```

```
                eval = self.alphabeta(new_board, depth - 1, alpha, beta,
BLACK)
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break
            return min_eval
```