程序报告

学号：2211312　　姓名：贾景顺

一、问题重述

结合对蒙特卡洛树搜索的理解，为黑白棋游戏创造 AI 玩家（miniAlphaGo for Reversi）。通过编写蒙特卡洛树类和 AI 玩家类中成员和函数，完成蒙特卡洛树的建立以及搜索各步骤（选择、扩展、模拟、反向传播）。将自己的 AI 玩家与系统测试所用 AI 玩家进行模拟对弈，并战胜各个等级 AI 玩家。

二、设计思想

代码创建了 MCTSNode 类实现进行蒙特卡洛树搜索的核心数据结构，并封装了实现树节点一系列功能的函数包括判断是否完全展开、是否有子节点、能否继续扩展等方法。expand 方法处理新节点的创建过程，而 backpropagate 方法实现奖励值的反向传播。select_best_child 方法应用 UCT 公式选择最有潜力的子节点。

AIPlayer 类初始化时，首先确定了 AI 玩家和对手的棋子颜色，设置了 60 秒的思考时间限制和 UCT 算法中控制探索程度的参数 c_param 为 1.4。当需要获取走法时，get_move 方法作为主入口，首先检查合法走法列表，若只有一个合法走法则立即返回。

接着创建蒙特卡洛树的根节点 MCTSNode，该节点保存了当前棋盘状态、父节点指针、到达该节点的走法、当前玩家颜色以及所有合法走法。根节点的初始化不直接复制整个棋盘，而是保留引用，在实际需要修改时才进行深拷贝（deepcopy 函数），这种延迟复制策略节省了内存和时间。

主搜索循环在时间限制内持续进行，每次迭代都完整执行 MCTS 的四个阶段。在选择阶段，从根节点出发，沿着树向下选择子节点，直到遇到未完全展开的节点或叶子节点。选择策略采用 UCT 算法，选择具有最大 choices_weights 值的子节点，平衡了利用已知高回报走法和探索较少尝试走法之间的权衡。

当遇到可扩展节点时进入扩展阶段，随机选择一个未尝试的合法走法，创建新的棋盘状态并生成对应的子节点。这里通过_can_fliped 方法验证走法有效性，确保只生成合法的游戏状态。新节点的当前玩家颜色会自动切换，并计算下一玩家的合法走法。

模拟阶段从新扩展的节点开始，使用快速随机策略进行对局直到游戏结束。rollout 方法实现了这一过程，它创建当前棋盘状态的深拷贝，然后双方轮流随机选择合法走法，直到双方都无棋可下。模拟结束后，根据最终棋盘上双方棋子的数量差计算奖励值，从当前玩家视角来看，这个差值越大越好。

回溯阶段将这个奖励值沿着搜索路径向上传播，更新所有经过节点的访问次数和累计奖励。通过这种反向传播机制，树中节点逐渐积累起对其价值的准确估计。

在时间耗尽或满足提前终止条件时，搜索结束。为了减少胜率较高步骤的用时，将提前终止条件设为思考时长大于 5 秒（确保足够的迭代次数）且当前模拟胜率大于 0.99999，此时选择被访问次数最多的子节点对应的走法作为最终决策。

三、代码内容

```
import random
```

```
import time
from math import sqrt, log
from copy import deepcopy

class AIPlayer:
    """
    AI 玩家，使用蒙特卡洛树搜索算法
    """

    def __init__(self, color):
        self.color = color
        self.opponent_color = 'O' if color == 'X' else 'X'
        self.time_limit = 60  # 60 秒思考时间限制
        self.c_param = 1.4  # UCT 算法的探索参数

    def get_move(self, board):
        player_name = '黑棋' if self.color == 'X' else '白棋'
        print(f"请等一会，对方 {player_name}-{self.color} 正在思考中...")

        # 获取所有合法移动（使用棋盘提供的 get_legal_actions 方法）
        legal_actions = list(board.get_legal_actions(self.color))
        if not legal_actions:
            return None

        # 快速返回单一选择
        if len(legal_actions) == 1:
            return legal_actions[0]

        # 创建蒙特卡洛树根节点
        root = MCTSNode(
            board=board,  # 注意这里不 deepcopy，因为我们会使用 board 的方法
            parent=None,
            move=None,
            current_color=self.color,
            legal_actions=legal_actions
        )

        start_time = time.time()
        iterations = 0

        # 主搜索循环
        while time.time() - start_time < self.time_limit:
            if time.time() - start_time > 5 and root.best_child().visits > 100 and
root.best_child().wins / root.best_child().visits > 0.99999:
```

```
                break
            # 选择阶段
            node = root
            while node.is_fully_expanded() and node.has_children():
                node = node.select_best_child(self.c_param)

            # 扩展阶段
            if node.can_expand():
                node = node.expand()

            # 模拟阶段
            reward = self.rollout(node.board, node.current_color)

            # 回溯阶段
            node.backpropagate(reward)

            iterations += 1

        print(f"完成 {iterations} 次模拟")

        # 选择最优动作（访问次数最多）
        best_child = max(root.children, key=lambda c: c.visits)
        return best_child.move

    def rollout(self, board, current_color):
        # 创建模拟用的棋盘对象（使用 deepcopy 确保不影响原棋盘）
        sim_board = deepcopy(board)
        cur_color = current_color

        while True:
            # 获取当前玩家的合法动作
            actions = list(sim_board.get_legal_actions(cur_color))

            # 双方都无合法动作时游戏结束
            if not actions:
                opp_actions = list(sim_board.get_legal_actions(
                    self.opponent_color if cur_color == self.color else self.color
                ))
                if not opp_actions:
                    break
                cur_color = self.opponent_color if cur_color == self.color else self.color
                continue

            # 随机选择动作
```

```
                move = random.choice(actions)
                # 使用_can_fliped 验证移动有效性
                flipped = sim_board._can_fliped(move, cur_color)
                if flipped is not False:  # 如果移动合法
                    sim_board._move(move, cur_color)

                # 切换玩家
                cur_color = self.opponent_color if cur_color == self.color else self.color

            # 使用 count 方法计算最终得分
            ai_count = sim_board.count(self.color)
            opp_count = sim_board.count(self.opponent_color)
            return ai_count - opp_count


class MCTSNode:
    """蒙特卡洛树节点类"""
    def __init__(self, board, parent, move, current_color, legal_actions):
        self.board = board                  # 棋盘对象
        self.parent = parent                # 父节点
        self.move = move                    # 到达本节点的移动
        self.current_color = current_color  # 当前玩家颜色
        self.children = []                  # 子节点列表
        self.untried_actions = legal_actions.copy()  # 未尝试的合法动作
        self.visits = 0                     # 访问次数
        self.total_reward = 0.0             # 累计奖励值

    def is_fully_expanded(self):
        """是否完全展开"""
        return len(self.untried_actions) == 0

    def has_children(self):
        """是否有子节点"""
        return len(self.children) > 0

    def can_expand(self):
        """是否可以扩展"""
        return len(self.untried_actions) > 0

    def select_best_child(self, c_param):
        """根据 UCT 公式选择最佳子节点"""
        choices_weights = [
            (child.total_reward / child.visits)
            + c_param * sqrt(2 * log(self.visits) / child.visits)
```

```
            for child in self.children
        ]
        return self.children[choices_weights.index(max(choices_weights))]

    def expand(self):
        """扩展新节点"""
        # 选择未尝试的动作
        move = random.choice(self.untried_actions)
        new_board = deepcopy(self.board)

        # 使用_can_fliped 验证移动有效性
        flipped = new_board._can_fliped(move, self.current_color)
        if flipped is False:
            self.untried_actions.remove(move)
            return self
        new_board._move(move, self.current_color)
        next_color = 'O' if self.current_color == 'X' else 'X'

        # 获取新状态的合法动作
        next_legal_actions = list(new_board.get_legal_actions(next_color))

        # 创建子节点
        child = MCTSNode(
            board=new_board,
            parent=self,
            move=move,
            current_color=next_color,
            legal_actions=next_legal_actions
        )

        # 更新节点状态
        self.untried_actions.remove(move)
        self.children.append(child)
        return child

    def backpropagate(self, reward):
        self.visits += 1
        self.total_reward += reward
        if self.parent:
            self.parent.backpropagate(reward)
```

四、实验结果

当将提前终止胜率设为 0.9999 以上或取消提前终止条件时，训练 AI 在先后手的情况下均可战胜系统 AI。

五、总结

在本次实验中，通过对蒙特卡洛树搜索算法的复现，较为圆满的完成了题设的要求。这也是我第一次较为大量的编写 python 代码，使得我对 python 的类和函数的语法以及蒙特卡罗树搜索的整体流程和具体实现有了进一步体会。