程序报告

学号：2313721　　姓名：许洋

一、问题重述

黑白棋问题：

黑白棋 (Reversi)，也叫苹果棋，翻转棋，是一个经典的策略性游戏。一般棋子双面为黑白两色，故称“黑白棋”。因为行棋之时将对方棋子翻转，则变为己方棋子，故又称“翻转棋” (Reversi) 。 棋子双面为红、绿色的称为“苹果棋”。它使用 8x8 的棋盘，由两人执黑子和白子轮流下棋，最后子多方为胜方。

游戏规则：

黑方先行，双方交替下棋。一步合法的棋步包括：在一个空格处落下一个棋子，并且翻转对手一个或多个棋子；新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来，可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格；一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。如果某一方落子时间超过 1 分钟 或者 连续落子 3 次不合法，则判该方失败。

实验要求：

使用 『蒙特卡洛树搜索算法』 实现 miniAlphaGo for Reversi。

使用 Python 语言。

算法部分需要自己实现，不要使用现成的包、工具或者接口。

对问题的理解：

实验平台已经给出了game.py和board.py的源代码，所以我们理论上只需要实现AIplayer模块的代码就可以了。

二、设计思想

(1) 启发式位置评分表

```
POS_WEIGHT = [
    [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]
]
```

- **功能**：定义了标准 8x8 棋盘的位置权重矩阵，用于评估棋盘上不同位置的重要性。
- **作用**：在模拟阶段优先选择评分高的位置，提升决策的合理性。

---

(2) 动作转换工具函数

```
def action_to_index(action):
    row = int(action[1]) - 1
    col = ord(action[0].upper()) - ord('A')
    return row, col
```

- **功能**：将棋盘动作（如 `'D3'`）转换为二维坐标索引（如 `(2, 3)`）。
- **作用**：方便在位置权重矩阵中查找对应评分。

---

(3) Node 类

`Node` 是蒙特卡洛树的核心组件，每个节点代表棋盘的一个状态。

a. 初始化 (`__init__`)

```
def __init__(self, board, color, root_color, parent=None,
action=None):
    self.board = board
    self.color = color
    self.root_color = root_color
    self.parent = parent
    self.action = action
    self.children = []
    self.visit = 0
    self.reward = {'X': 0, 'O': 0}
    self.value = {'X': 1e5, 'O': 1e5}
    self.actions = list(self.board.get_legal_actions(color=color))
    self.is_leaf = True
    self.is_over = self.check_game_over()
```

- **功能**：初始化节点，存储当前棋盘状态、玩家颜色、合法动作列表等信息。
- 关键字段：
    - `board`：当前棋盘状态。
    - `color` 和 `root_color`：当前玩家颜色和根节点玩家颜色。
    - `parent` 和 `children`：父节点和子节点，用于构建树结构。
    - `visit` 和 `reward`：记录节点被访问的次数和奖励值。
    - `value`：节点的价值，用于选择阶段。
    - `actions`：当前状态下所有合法动作。
    - `is_leaf` 和 `is_over`：标记是否为叶节点或游戏结束。

b. 检查游戏是否结束 (`check_game_over`)

```
def check_game_over(self):
    return len(list(self.board.get_legal_actions('X'))) == 0 and \
           len(list(self.board.get_legal_actions('O'))) == 0
```

- **功能**：判断当前棋盘是否没有合法动作，从而确定游戏是否结束。

c. 更新节点价值 (`update_value`)

```
def update_value(self):
    if self.visit == 0 or self.parent is None:
        return
    for c in ['X', 'O']:
        self.value[c] = self.reward[c] / self.visit + \
                        Node.C *
math.sqrt(math.log(self.parent.visit + 1) / self.visit)
```

- 功能：根据 UCB1 公式更新节点的价值，用于选择阶段。
- 公式：UCB1=visitreward+C·visitlog(parent.visit)

d. 扩展节点 (`expand`)

```
def expand(self):
    next_color = 'X' if self.color == 'O' else 'O'
    if not self.actions:
        self.children.append(Node(deepcopy(self.board), next_color,
self.root_color, self, None))
    else:
        for act in self.actions:
            next_board = deepcopy(self.board)
            next_board._move(act, self.color)
            self.children.append(Node(next_board, next_color,
self.root_color, self, act))
    self.is_leaf = False
```

- 功能：为当前节点生成所有可能的子节点，每个子节点对应一个合法动作。

e. 选择子节点 (`select_child`)

```
def select_child(self, epsilon=0.3):
    if random.random() < epsilon:
        return random.choice(self.children)
    return max(self.children, key=lambda c: c.value[self.color])
```

- *功能**：使用 ϵ-贪婪策略，在探索和利用之间找到平衡。

f. 获取最佳动作 (`best_move`)

```
def best_move(self):
    if not self.children:
        return None
    return max(self.children, key=lambda c: c.visit).action
```

- 功能：根据子节点的访问次数，返回最优的动作。

---

(4) MonteCarlo_Search 类

`MonteCarlo_Search` 是实现 MCTS 的主类，负责协调整个搜索过程。

a. 初始化 (`__init__`)

```
def __init__(self, color, board):
    self.color = color.upper()
    self.root = Node(deepcopy(board), self.color, self.color)
    self.epsilon = 0.3
    self.gamma = 0.999
    self.pos_weight = POS_WEIGHT
```

- 功能：初始化搜索器，包含棋盘副本、玩家颜色、探索参数 ϵ 和折扣因子 γ。

b. 搜索入口 (`search`)

```
def search(self):
    if len(self.root.actions) == 1:
        return self.root.actions[0]
    try:
        func_timeout(3, self.mcts_search, args=(self.root,))
    except FunctionTimedOut:
        pass
    return self.root.best_move()
```

- 功能：初始化根节点并调用 `mcts_search` 进行搜索，返回最佳动作。

c. MCTS 主循环 (`mcts_search`)

```
def mcts_search(self, root):
    while True:
        node = self.select(root)
        if node.is_over:
            winner, diff = node.board.get_winner()
        else:
            if node.visit > 0:
                node.expand()
                node = node.select_child(self.epsilon)
                self.epsilon *= self.gamma
            winner, diff = self.simulate(node)
        self.backpropagate(node, winner, diff)
```

- 功能：不断执行选择、扩展、模拟和回溯，直到超时。

d. 选择阶段 (`select`)

```
def select(self, root):
    current = root
    while not current.is_leaf and current.children:
        current = current.select_child(self.epsilon)
        self.epsilon *= self.gamma
    return current
```

- 功能：从根节点开始，递归选择最有潜力的子节点，直到到达叶节点。

e. 模拟阶段 (`simulate`)

```
def simulate(self, node):
    board = deepcopy(node.board)
    color = node.color
    while not self.check_game_over(board):
        actions = list(board.get_legal_actions(color))
        if actions:
            best_action = max(actions, key=lambda a:
self.pos_weight[action_to_index(a)[0]][action_to_index(a)[1]])
            board._move(best_action, color)
        color = 'X' if color == 'O' else 'O'
    return board.get_winner()
```

- 功能：在当前节点的基础上进行随机模拟，直到游戏结束。

f. 回溯阶段 (`backpropagate`)

```
def backpropagate(self, node, winner, diff):
    while node:
        node.visit += 1
        if winner == 0:
            node.reward['X'] += diff
            node.reward['O'] -= diff
        elif winner == 1:
            node.reward['X'] -= diff
        node.update_value()
        node = node.parent
```

- 功能：将模拟结果更新到路径上的所有节点。

---

(5) AIPlayer 类

`AIPlayer` 是一个简单的接口类，用于与外部交互。

a. 初始化 (`__init__`)

```
def __init__(self, color):
    self.color = color.upper()
```

- 功能：设置玩家颜色。

b. 获取下一步动作 (`get_move`)

```
def get_move(self, board):
    print(f"请稍等，对方 {'黑棋' if self.color == 'X' else '白棋'}-
{self.color} 正在思考中...")
    action = MonteCarlo_Search(self.color, board).search()
    return action
```

- 功能：调用 `MonteCarlo_Search` 进行搜索并返回最佳动作。

三、代码内容

```
import math
import random
from copy import deepcopy
from func_timeout import func_timeout, FunctionTimedOut

# 启发式位置评分表（标准 8x8 棋盘）
POS_WEIGHT = [
    [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]
]

def action_to_index(action):
    """将动作如 'D3' 转换为坐标索引 (2, 3)"""
    row = int(action[1]) - 1
    col = ord(action[0].upper()) - ord('A')
    return row, col


class Node:
    C = 2  # UCT 探索常数
```

```
    def __init__(self, board, color, root_color, parent=None,
action=None):
        self.board = board
        self.color = color
        self.root_color = root_color
        self.parent = parent
        self.action = action
        self.children = []
        self.visit = 0
        self.reward = {'X': 0, 'O': 0}
        self.value = {'X': 1e5, 'O': 1e5}
        self.actions =
list(self.board.get_legal_actions(color=color))
        self.is_leaf = True
        self.is_over = self.check_game_over()

    def check_game_over(self):
        return len(list(self.board.get_legal_actions('X'))) == 0
and \
               len(list(self.board.get_legal_actions('O'))) == 0

    def update_value(self):
        if self.visit == 0 or self.parent is None:
            return
        for c in ['X', 'O']:
            self.value[c] = self.reward[c] / self.visit + \
                            Node.C *
math.sqrt(math.log(self.parent.visit + 1) / self.visit)

    def expand(self):
        next_color = 'X' if self.color == 'O' else 'O'
        if not self.actions:
            self.children.append(Node(deepcopy(self.board),
next_color, self.root_color, self, None))
        else:
            for act in self.actions:
                next_board = deepcopy(self.board)
                next_board._move(act, self.color)
                self.children.append(Node(next_board, next_color,
self.root_color, self, act))
        self.is_leaf = False
```

```
    def select_child(self, epsilon=0.3):
        if random.random() < epsilon:
            return random.choice(self.children)
        return max(self.children, key=lambda c:
c.value[self.color])

    def best_move(self):
        if not self.children:
            return None
        return max(self.children, key=lambda c: c.visit).action


class MonteCarlo_Search:
    def __init__(self, color, board):
        self.color = color.upper()
        self.root = Node(deepcopy(board), self.color, self.color)
        self.epsilon = 0.3
        self.gamma = 0.999
        self.pos_weight = POS_WEIGHT

    def search(self):
        if len(self.root.actions) == 1:
            return self.root.actions[0]
        try:
            func_timeout(3, self.mcts_search, args=(self.root,))
        except FunctionTimedOut:
            pass
        return self.root.best_move()

    def mcts_search(self, root):
        while True:
            node = self.select(root)
            if node.is_over:
                winner, diff = node.board.get_winner()
            else:
                if node.visit > 0:
                    node.expand()
                    node = node.select_child(self.epsilon)
                    self.epsilon *= self.gamma
                winner, diff = self.simulate(node)
            self.backpropagate(node, winner, diff)
```

```
    def select(self, root):
        current = root
        while not current.is_leaf and current.children:
            current = current.select_child(self.epsilon)
            self.epsilon *= self.gamma
        return current

    def simulate(self, node):
        board = deepcopy(node.board)
        color = node.color
        while not self.check_game_over(board):
            actions = list(board.get_legal_actions(color))
            if actions:
                # 启发式模拟：优先选择评分高的位置
                best_action = max(actions, key=lambda a:
self.pos_weight[action_to_index(a)[0]][action_to_index(a)[1]])
                board._move(best_action, color)
            color = 'X' if color == 'O' else 'O'
        return board.get_winner()

    def backpropagate(self, node, winner, diff):
        while node:
            node.visit += 1
            if winner == 0:
                node.reward['X'] += diff
                node.reward['O'] -= diff
            elif winner == 1:
                node.reward['X'] -= diff
            node.update_value()
            node = node.parent

    def check_game_over(self, board):
        return len(list(board.get_legal_actions('X'))) == 0 and \
               len(list(board.get_legal_actions('O'))) == 0


class AIPlayer:
    def __init__(self, color):
        self.color = color.upper()

    def get_move(self, board):
```

```
        print(f"请稍等，对方 {'黑棋' if self.color == 'X' else '白棋'}-{self.color} 正在思考中...")
        action = MonteCarlo_Search(self.color, board).search()
        return action
```

四、实验结果

- 平台检测结果：

五、总结

1. 算法的强大性

- 蒙特卡洛树搜索（MCTS）通过随机模拟和统计分析，能够在不依赖复杂评估函数的情况下找到接近最优的策略。
- 适用于黑白棋这种双人零和博弈问题，展现了其在复杂状态空间中的探索能力。

2. 优化策略的有效性

- 引入位置权重矩阵，优先选择重要的棋盘位置，显著提升了模拟的质量。
- 使用贪婪策略，在探索与利用之间找到平衡，避免陷入局部最优解。
- 时间限制机制确保算法在实际应用中具有良好的实时性，适应不同场景的需求。

3. 存在的改进空间

- 当前的模拟策略虽然考虑了位置权重，但未充分挖掘棋盘上的其他关键特征（如棋子稳定性、边缘控制等）。
- 模拟阶段可以引入更复杂的评估函数或结合深度学习模型，进一步提升决策的准确性。
- 并行化技术的应用能够显著提高搜索效率，尤其是在处理大规模棋盘或更深的搜索深度时。

4. 启发与展望

- 未来可以通过引入混合方法（如结合深度强化学习）或改进现有机制，进一步提升 AI 玩家的性能和智能水平。