程序报告

学号：2310422　　姓名：谢小珂

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：

1.棋局开始时黑棋位于 E4 和 D5 ，白棋位于 D4 和 E5 黑方先行，双方交替下棋。

2.一步合法的棋步包括：

• 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。

3.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。

4.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。

5.棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。

6.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：

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

2.使用 Python 语言。

3.算法部分需要自己实现，不要使用现成的包、工具或者接口。

二、设计思想

（1）核心方法

当前的黑白棋 AI 基于蒙特卡洛树搜索（MCTS），这是一种通过模拟对局来评估走法胜率的算法。其核心分为四个阶段：

1. 选择：从根节点出发，按照 UCB 公式（平衡探索与利用）选择子节点，直到叶子节点。
2. 扩展：若当前节点有未尝试的合法走法，则扩展一个新子节点。
3. 模拟：从新节点开始随机落子至终局，生成胜负结果。
4. 反向传播：将模拟结果反向更新路径上所有节点的统计信息（访问次数、累计价值）。

（2）已实现的改进

在基础 MCTS 上，项目已做了以下优化：

1. 动态 UCB 系数 C=1，并引入 30%概率的随机探索，避免陷入局部最优。
2. 奖励设计：胜负结果（±100）叠加棋子数量差（delta），区分“小胜”和“大胜”。
3. 模拟截断：限制单次模拟步数（max_stimulate_times=500），防止无限循环。

（3）参数调整优化方向

1.动态探索系数可以在开局阶段提高 C 值（如 1.5）鼓励多样化尝试，残局阶段降低（如 0.5）专注精确计算。
2.自适应模拟次数可以在简单局面减少模拟次数节省时间，复杂局面增加模拟提升决策质量。
3.随着搜索深度增加，逐步降低随机探索概率（如从 0.3→0.1），增强策略稳定性。

（4）框架级优化建议

1.将大量随机模拟分配到多线程/多进程，速度可提升 5-10 倍，但需注意线程安全和状态隔离。
2.用哈希表缓存常见局面的评估结果，避免重复计算，减少 30%-50%冗余操作。
3.开局使用预存策略库，中局用 MCTS，残局切换为精确搜索（如 Alpha-Beta 剪枝），兼顾速度与精度。

（5）当前局限性及解决方案

1.初始搜索盲目性：前几轮模拟完全随机，效率低 → 可用预训练策略网络引导早期搜索。
2.对称局面重复计算：旋转/镜像局面被当作不同状态处理 → 增加对称性检测模块，合并等效节点。
3.终局不精确：剩余 10 步仍用随机模拟 → 设定阈值，切换为穷举搜索。
4.内存膨胀：搜索树无限增长 → 设置节点数上限，启用 LRU 淘汰机制。

三、代码内容

蒙特卡洛树搜索是一种基于随机模拟的决策树搜索算法，通过重复执行"选择-扩展-模拟-反向传播"四个阶段来逐步构建搜索树。

1. 选择(Selection)

原理：从根节点开始，递归选择 UCB 值最大的子节点，直到到达叶子节点。
UCB 公式：UCB = (节点价值/访问次数) + C * sqrt(ln(父节点访问次数)/子节点访问次数)

```python
# ================ 1. 选择(Selection) ================
    def calu_ucb(self, node, scalar=1):
        max_value = -float('inf')  # 初始化最大值为负无穷
        best_child = []           # 存储具有最大 UCB 值的子节点列表
        for child in node.children:
            # 如果子节点未被访问过，优先选择
            if child.visit_num == 0:
                best_child = [child]
                break
            value = (child.value / child.visit_num) + scalar * math.sqrt(
                2.0 * math.log(node.visit_num) / float(child.visit_num))

            # 更新最大值和最佳子节点列表
            if value > max_value:
                best_child = [child]
                max_value = value
            elif value == max_value:  # 如果有多个相同值的节点，都保留
                best_child.append(child)

        # 如果没有合适的子节点，返回父节点(这种情况理论上不应该发生)
        if len(best_child) == 0:
            return node.parent
        # 从最佳子节点中随机选择一个(如果有多个)
        return random.choice(best_child)

    def select(self, node):
        """选择策略：结合 UCB 策略与随机探索"""
        # 循环直到找到叶子节点
        while any(node.state.get_legal_actions(player) for player in ['X', 'O']):
            # 如果是叶子节点(没有子节点)，准备扩展
            if len(node.children) == 0:
                return self.expand(node)
            # 以 select_probability 的概率进行随机探索
            elif random.uniform(0, 1) < self.select_probability:
                node = self.calu_ucb(node)
            else:  # 否则使用 UCB 策略
                node = self.calu_ucb(node)
                # 如果节点未完全扩展，先扩展
                if not node.full_expand():
                    return self.expand(node)
                else:  # 如果已完全扩展，继续选择
                    node = self.calu_ucb(node)
        return node
```

2. 扩展(Expansion)

原理：当遇到未完全扩展的非终止节点时，添加一个或多个子节点。

```python
# ================ 2. 扩展(Expansion) ================
    def expand(self, node):
        """扩展新的子节点"""
        # 获取当前节点的所有合法动作
        actions = list(node.state.get_legal_actions(node.color))
        # 如果没有合法动作，返回父节点
        if len(actions) == 0:
            return node.parent

        # 获取已经尝试过的动作
        tried_action = [child.action for child in node.children]
        # 找出未尝试的动作
        not_tried_action = [a for a in actions if a not in tried_action]
        # 随机选择一个未尝试的动作
        action = random.choice(not_tried_action)

        new_state = copy.deepcopy(node.state)
        # 执行选定的动作
        new_state._move(action, node.color)
        # 添加新节点到树中，并切换玩家颜色
        node.add(new_state, action, color='O' if node.color == 'X' else 'X')
        return node.children[-1]
```

3. 模拟(Simulation)

原理：从扩展的节点开始随机落子直到终局，获得胜负结果。

```python
# ================ 3. 模拟(Simulation) ================
    def simulate(self, node):
        """从当前节点开始随机模拟直到游戏结束"""
        # 复制当前状态以避免修改原始状态
        state = copy.deepcopy(node.state)
        color = node.color
        # 初始化步数计数器
        count = 0
        # 随机落子直到游戏结束或达到最大模拟次数
        while any(node.state.get_legal_actions(player) for player in ['X', 'O']) and count <
self.max_stimulate_times:
            # 获取当前玩家的合法动作
            actions = list(state.get_legal_actions(color))
            if len(actions) != 0:
                action = random.choice(actions)
                state._move(action, color)
            # 增加步数计数器
            count += 1
```

```
            color = 'X' if color == 'O' else 'O'
        # 计算奖励值
        winner, delta = state.get_winner()
        # 根据胜负结果返回不同的奖励值
        if winner == 0 and self.color == 'X':
            return 100 + delta  # 黑棋(X)胜利
        elif winner == 0 and self.color == 'O':
            return 0
        if winner == 1 and self.color == 'O':
            return 100 + delta  # 白棋(O)胜利
        elif winner == 1 and self.color == 'X':
            return 0
        else:
            return -(100 + delta)  # 平局或失败
```

4. 反向传播(Backpropagation)

原理：将模拟结果反向传播到路径上的所有节点。

```
# ================ 4. 回传(Backpropagation) ================
    def backpropagation(self, node, reward):
        """将模拟结果反向传播到路径上的所有节点"""
        # 从当前节点开始，向上回溯到根节点
        while node is not None:
            # 增加节点的访问次数
            node.visit_num += 1
            # 累加节点的价值
            node.value += reward
            # 移动到父节点
            node = node.parent
        return
```

5. 主要流程代码

在限定时间内，通过模拟大量随机对局，找出当前棋盘状态下胜率最高的落子位置

```
# ================ 主流程 ================
    def choose_best_child(self, node):
        """选择最佳子节点"""
        # 进行多次模拟
        for _ in range(self.max_try_time):
            leaf = self.select(node)
            reward = self.simulate(leaf)
            self.backpropagation(leaf, reward)
        # 选择访问次数最多的节点(探索系数设为0，只考虑利用)
        best_child = self.calu_ucb(node, 0)
        return best_child.action

    def get_move(self, board):
        """获取最佳移动"""
```

```
        if self.color == 'X':
            player_name = '黑棋'
        else:
            player_name = '白棋'
        print(f"请等一会，对方 {player_name}-{self.color} 正在思考中...")
        now_state = copy.deepcopy(board)
        # 创建根节点
        node = MCT_node(now_state, color=self.color)
        # 选择最佳动作并返回
        return self.choose_best_child(node)
```

6. 蒙特卡洛搜索树节点类，表示游戏树中的一个状态节点

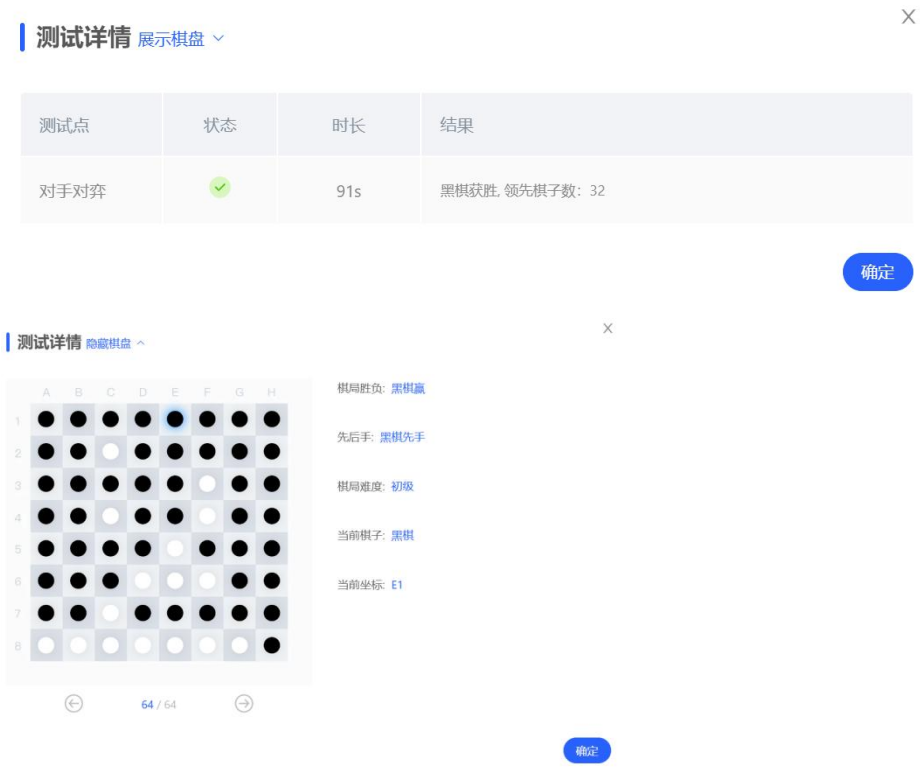
```
class MCT_node:
    def __init__(self, state, parent=None, action=None, color=''):
        # 初始化节点属性
        self.color = color      # 当前玩家颜色，'X'表示黑棋，'O'表示白棋
        self.parent = parent    # 父节点指针，根节点的 parent 为 None
        self.state = state      # 当前游戏状态(棋盘对象)
        self.value = 0.00       # 节点累计价值，用于 UCB 计算
        self.visit_num = 0      # 节点访问次数(visits number)
        self.action = action    # 从父节点到达此节点所采取的动作(如'D3')
        self.children = []      # 子节点列表，存储所有可能的后续状态

    def add(self, child_state, action, color):
        """添加一个新的子节点到当前节点"""
        # 创建新节点：状态为 child_state，父节点为 self，动作为 action，颜色为 color
        child = MCT_node(child_state, self, action, color)
        # 将新节点添加到子节点列表
        self.children.append(child)

    def full_expand(self):
        """检查当前节点是否已经完全扩展(所有合法动作都有对应的子节点)"""
        # 获取当前玩家的所有合法动作
        action = self.state.get_legal_actions(self.color)
        # 比较子节点数量与合法动作数量是否相等
        return len(self.children) == len(list(action))
```

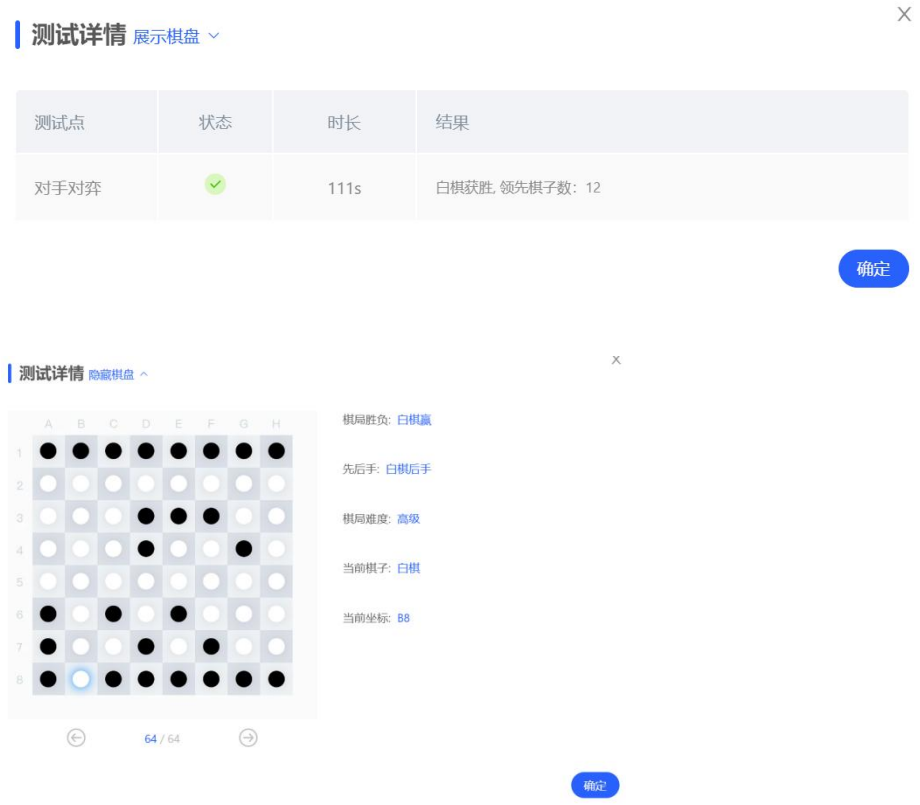
四、实验结果

- 《高级难度 黑棋先手》测试结果：



以 32 个棋子的优势取得胜利！

· 《高级难度 白棋后手》测试结果：



以 12 个棋子的优势取得胜利！

五、总结

1.通过本次项目，我初步理解了蒙特卡洛树搜索（MCTS）的核心机制：
•初步理解了 UCB 公式的数学原理及其在平衡探索与利用中的作用，认识到 C 值的设置对算法性能的关键影响。
• 理解了四阶段（选择-扩展-模拟-反向传播）的协同工作机制，特别是回传过程如何实现信息的逆向传播。
• 认识到 MCTS 与传统的 Minimax 算法的本质区别：前者通过随机采样构建非对称搜索树，后者依赖完整的深度优先搜索。
•在代码实现过程中获得了经验：学会了使用深拷贝保证模拟过程的独立性，同时认识到其性能开销；理解了面向对象的树节点设计方法，掌握父子节点的指针维护技巧。
• 认识到理论算法与工程实现的差距，如理论上的 O(n)复杂度在实际中可能受常数项影响。

2.可以优化改进的部分：
•动态改变探索系数的值：在开局阶段，提高探索系数鼓励多样性；而在终局阶段：降低探索系数专注最优解。
•固定迭代次数无法适应不同复杂度局面。可以采用动态时间分配：在简单局面，快速决策；而在复杂局面进行延长搜索，逐步增加模拟次数，在时间耗尽时返回当前最优解。
• 增强评估函数有待增强，问题在于仅依赖胜负结果（±100）忽略中间优势。 应该考虑多维度评估。

3.策略升级：
• 优先占角：角落位置一旦占据便无法被翻转，价值极高。
• 限制对手行动力：选择让对手合法走法最少的策略。
• 避免危险边线：某些边线位置易被对手利用形成夹击。
• 通过启发式规则引导模拟，单次模拟的质量会大幅提高，减少无效搜索。