**黑白棋程序报告**

学号：3210102037 姓名：徐铭

1. **问题重述**

（简单描述对问题的理解，从问题中抓住主干，必填）

====================================================================

黑白棋 (Reversi)，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。

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。

一步合法的棋步包括：

1. 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；
2. 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；
3. 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。

1. **设计思想**

（所采用的方法，有无对方法加以改进，该方法有哪些优化方向（参数调整，框架调整，或者指出方法的局限性和常见问题），伪代码，理论结果验证等… **思考题，非必填**）

====================================================================

首先，实验要求了使用蒙特卡洛搜索算法，因此主要思路就是通过选择-扩展-模拟-回溯的迭代找出胜率最高的一步棋，而其中可以优化的部分在于模拟的过程使用的不是UCB1算法。

一般来说，使用随机算法就可以，但是为了提高胜率，我选择使用位置优先策略，即由Roxanne提出的优先级表，来选择下一个扩展的节点，而不是在可选动作中随机选择一个，最后结果验证，优化后的算法相比随机选择计算量有了大幅降低，在相同计算层数下，能够下赢高级测试AI。

1. **代码内容**

（能体现解题思路的主要代码，有多个文件或模块可用多个"===="隔开，必填）

====================================================================

**def monte\_carlo\_tree\_search(self, board):**

root = MonteCarloTreeNode(color=self.color, action=None, board=board)

for \_ in range(self.time):

node = self.select(root)

node = self.expand(node)

result = self.simulate(node)

self.backpropagation(node, result)

best\_child = max(root.children, key=lambda x: x.visit\_count)

return best\_child.action

**def select(self, node):**

while not node.is\_leaf():

if node.is\_fully\_expanded():

return max(node.children, key=lambda x: x.UCB1())

else:

return self.expand(node)

return node

**def expand(self, node):**

legal\_actions = list(node.board.get\_legal\_actions(node.color))

for child in node.children:

if child.action in legal\_actions:

legal\_actions.remove(child.action)

if len(legal\_actions) == 0:

return node

else:

board = copy.deepcopy(node.board)

action = self.choose\_action(legal\_actions)

board.\_move(action, node.color)

node.add\_child(color=self.change\_color(node.color), action=action, board=board)

return node.children[-1]

**def backpropagation(self, node, result):**

while node is not None:

if node.color == self.color:

node.visit(result)

else:

node.visit(128 - result)

node = node.parent

**def simulate(self, node):**

sim\_board = copy.deepcopy(node.board)

color = node.color

while 1:

legal\_actions = list(sim\_board.get\_legal\_actions(color))

if legal\_actions:

action = self.choose\_action(legal\_actions)

sim\_board.\_move(action, color)

color = self.change\_color(color)

else:

color = self.change\_color(color)

legal\_actions = list(sim\_board.get\_legal\_actions(color))

if legal\_actions:

action = self.choose\_action(legal\_actions)

sim\_board.\_move(action, color)

color = self.change\_color(color)

else:

break

winner, diff = sim\_board.get\_winner()

if winner == 2:

score = 64

elif winner == 0 and node.color == 'X':

score = 64 + diff

elif winner == 1 and node.color == 'O':

score = 64 + diff

else:

score = 64 - diff

return score

上述代码实现了蒙特卡洛树搜索算法（Monte Carlo Tree Search，MCTS）的主要流程，整个算法通过不断地选择、扩展、模拟和回传来构建MCTS树，并利用UCB1公式来指导节点选择和扩展，从而找到最佳的行动策略，其实现逻辑如下：

1. monte\_carlo\_tree\_search(self, board): 这是MCTS的主要方法。它创建了一个MCTS树的根节点，然后在一定次数内（由self.time控制）进行循环，每次循环选择节点、拓展节点、模拟游戏并回传结果。最后，它选择访问次数最多的子节点作为最佳行动。
2. select(self, node): 这个方法用于选择节点，直到遇到叶节点为止。如果当前节点已完全扩展，它将根据UCB1公式选择最有潜力的子节点进行扩展；否则，它会扩展当前节点。
3. expand(self, node): 这个方法用于扩展节点，即添加子节点。它会获取当前节点的所有合法行动，并根据这些行动创建子节点，然后选择其中一个行动并将棋盘状态转移到该子节点。
4. backpropagation(self, node, result): 这个方法用于将游戏结果回传到MCTS树中。它沿着树向根节点回溯，并根据游戏结果更新节点的访问次数和得分。
5. simulate(self, node): 这个方法用于模拟游戏过程。它从当前节点开始，按照随机策略进行游戏模拟，直到游戏结束。然后根据游戏结果计算得分。

======================================================================

**class MonteCarloTreeNode:**

"""

Monte Carlo Tree Node

"""

def \_\_init\_\_(self, color, action, parent=None, board=None):

# State

self.color = color # Current color

self.action = action # Previous action

self.board = board # Current board state

# Node

self.parent = parent

self.children = []

self.visit\_count = 0

self.score = 0

**def UCB1(self, C=2):**

"""

UCB1 formula

"""

if self.visit\_count == 0:

return float('inf')

else:

return self.score / self.visit\_count + C \* (2 \* np.log(self.parent.visit\_count) / self.visit\_count) \*\* 0.5

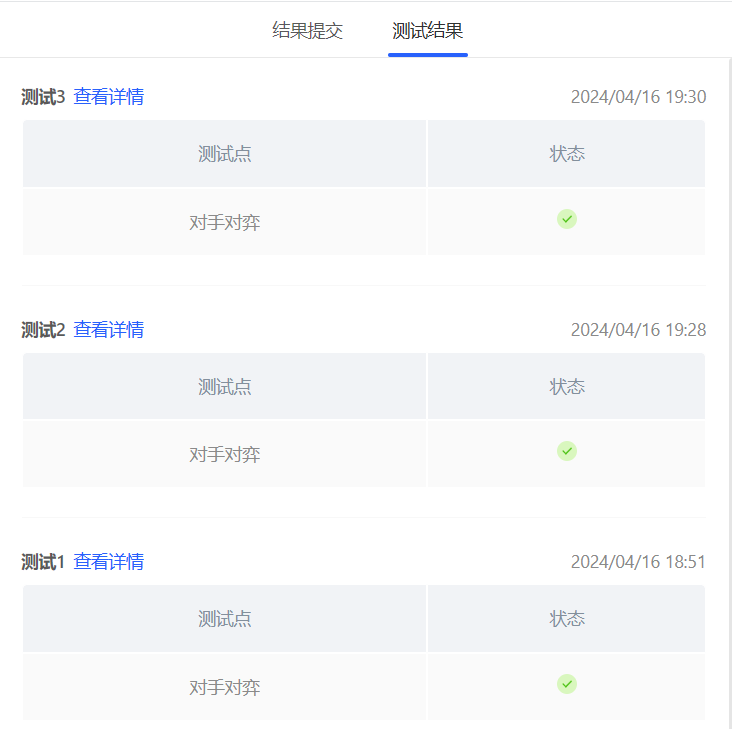
上述代码描述了蒙特卡洛搜索树的节点属性和UCB1算法的实现。

1. **实验结果**

（实验结果，必填）

====================================================================

选择高级对手对弈，结果如下：



其中，测试3的具体情况如下：



1. **总结**

（自评分析（是否达到目标预期，可能改进的方向，实现过程中遇到的困难，从哪些方面可以提升性能，模型的超参数和框架搜索是否合理等），**思考题，非必填**）

====================================================================

主要遇到的困难就是如何优化蒙特卡洛树搜索算法，由于算法本身不难，只是繁琐，而且有众多模板，因此实现没什么困难，优化上则需要测试各种黑白棋的策略，最后发现单一的贪心策略和消失策略效果并不理想，确定子策略的算法比较复杂，程序效率较低，而机动性策略难以用程序体现，奇偶策略的效果也不佳，而位置优先策略在UCT算法中表现优异。

此外可以优化的方向，个人认为可以参考AlphaGo的思路，进一步提高AI棋手的效率，我们可以引入卷积神经网络和自博弈的方式更新模型，并最终保留效果最佳的模型参数。