**程序报告**

学号： 2112060 姓名：孙蕗

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）

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

使用蒙特卡洛树搜索算法实现黑白棋游戏 AI 对手部分，即 miniAlphaGo for Reversi。

黑白棋是一种双方交替下棋的策略性游戏，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。每次合法的棋步都需要在空格处落下一枚棋子，同时翻转对手一个或多个棋子；新落下的棋子要在可夹住对手棋子的位置上，对方被夹住的所有棋子都要翻转过来，夹住的位置上必须全部是对手的棋子，不能有空格。如果一方没有合法棋步，即不管下到哪里，都不能至少翻转对手一个棋子，则这一轮必须弃权，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**）

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

实现了一种基于蒙特卡洛树算法的AI玩家，通过模拟对局来评估当前棋面的胜率，从而进行决策。

该AI玩家使用了Node类表示蒙特卡洛树的节点，每个节点存储了当前的棋面状态、下一步落子的颜色、父节点、子节点、落子位置、节点访问次数和胜率等信息。UCT算法是一种基于蒙特卡罗树搜索的博弈AI算法，核心思想是通过遍历游戏树，找到最优策略。UCT算法包含四个步骤：选择、扩展、模拟和回传。在搜索过程中，通过调用select\_policy函数，使用UCB1算法选择最优的节点并进行扩展。游戏树中的节点，通过扩展节点，不断生成新的子节点，从而构建整棵游戏树。在扩展节点时，随机选择一个未扩展的合法落子位置，并在当前节点下进行模拟对局，直到到达游戏结束状态。在随机模拟函数中，代码利用 Python 自带的随机库 random 对候选行动进行随机选择。随机化技术可以保证每个行动都有被探索的机会，避免固化在某一个状态或策略上。在模拟对局结束后，根据胜负情况和模拟对局的步数，计算该局游戏的胜率并进行回传。在回传过程中，更新节点的访问次数和胜率信息。通过不断的搜索和模拟对局，蒙特卡洛树会不断更新节点的胜率信息，从而尽可能地寻找最优的落子位置。最终，AI玩家会返回蒙特卡洛树中访问次数最多的子节点的落子位置作为最优解。

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）

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

class AIPlayer:

"""

AI 玩家

"""

def \_\_init\_\_(self, color):

"""

玩家初始化

param color: 下棋方，'X' - 黑棋，'O' - 白棋

"""

self.color = color#记录当前AI玩家的颜色

self.param = 1#UCT中的探索参数

self.maxtime = 50#搜索时间上限

def is\_ended(self, board):

"""判断游戏是否结束"""

list1 = list(board.get\_legal\_actions('X'))

list2 = list(board.get\_legal\_actions('O'))

return len(list1) == 0 and len(list2) == 0

def UCTSearch(self, board):

"""

UCT搜索函数

param board: 当前棋盘状态

return: UCB1得分最高的子节点的行动

"""

start\_time = time.time()

root\_node = Node(board, self.color)

while time.time() - start\_time < self.maxtime:

next\_node = self.select\_policy(root\_node)

reward = self.simualte\_policy(next\_node)

self.back\_propagate(next\_node, reward)

return self.UCB1(root\_node, 0).action

def select\_policy(self, node):

"""

如果还未全部扩展，用于选择下一步行动对应的子节点；如果全部扩展，根据UCB1公式选择最佳子节点

:param node: 当前节点

:return: 扩展后的新节点或者最佳子节点

"""

while not self.is\_ended(node.board):

if not node.is\_fully\_expanded():

new\_node = self.expand(node)

return new\_node

else:

node = self.UCB1(node)

return node

def switch\_color(self, color):

"""交换颜色"""

if color == 'X':

new\_color = 'O'

else:

new\_color = 'X'

return new\_color

def simualte\_policy(self, node):

"""

随机模拟函数，用于估计策略收益

:param node: 当前节点

:return: 随机模拟得到的收益

"""

board = deepcopy(node.board)

color = node.color

while not self.is\_ended(board):

action\_list = list(board.get\_legal\_actions(color))

if not len(action\_list) == 0:

action = random.choice(action\_list)

board.\_move(action, color)

color = self.switch\_color(color)

else:

color = self.switch\_color(color)

action\_list = list(board.get\_legal\_actions(color))

action = random.choice(action\_list)

board.\_move(action, color)

color = self.switch\_color(color)

winner, count = board.get\_winner()

if winner == 2:

return 0

reward = count + 15

if (winner == 1 and self.color == 'O') or (winner == 0 and self.color == 'X'):

return reward

return -reward

def back\_propagate(self, node, reward):

"""

回传函数，用于更新遍历路径上所有节点的访问次数及对应的收益值

:param node: 当前节点

:param reward: 对应随机模拟得到的收益

"""

while not node == None:

if node.color == self.color:

node.reward -= reward

else:

node.reward += reward

node.visit += 1

node = node.parent

def expand(self, node):

"""

扩展函数，用于向下一层添加一个新的子节点

:param node: 当前节点

:return: 新子节点

"""

new\_color = self.switch\_color(node.color)

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

new\_board = deepcopy(node.board)

# 获取已扩展过的节点列表

expanded\_list = [child.action for child in node.children]

# 随机选择一个未扩展的子节点

action = random.choice(list(set(action\_list)-set(expanded\_list)))

new\_board.\_move(action, node.color)

#检查对手有没有动作

if len(list(new\_board.get\_legal\_actions(new\_color))) == 0:

new\_color = node.color

# 扩展该子节点

node.expand\_children(new\_board, new\_color, action)

return node.children[-1]

def UCB1(self, node, param=1):

"""

根据UCB1公式计算每个子节点的得分，并返回UCB1得分最高的子节点

:param node: 当前节点

:param param: UCT搜索中的探索参数

:return: UCB1得分最高的子节点

"""

max\_score = -INF

for child in node.children:

avg\_reward = child.reward / child.visit

exploration = param \* \

math.sqrt(2.0 \* math.log(node.visit) / float(child.visit))

score = avg\_reward+exploration

if(score > max\_score):

best\_child = child

max\_score = score

return best\_child

def get\_move(self, board):

"""

根据当前棋盘状态，获取AI玩家的落子位置

:param board: 当前棋盘状态

:return: AI玩家的落子位置

"""

if self.color == 'X':

player\_name = '黑棋'

else:

player\_name = '白棋'

print("请等一会，对方 {}-{} 正在思考中...".format(player\_name, self.color))

current\_board = deepcopy(board)

action = self.UCTSearch(current\_board)

return action

1. **实验结果**

（实验结果，必填）

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





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**）

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

该AI玩家能够在规定时间内做出比较合理的落子，但是其性能还有待提升。可能的改进方向包括：

调整参数：UCB1算法中的参数对算法性能有很大影响，需要根据具体情况进行调整。

改进模拟对局策略：当前实现中的模拟对局策略比较简单，可以考虑采用更高级的算法，比如蒙特卡洛树搜索的变种算法。

加入启发式搜索：蒙特卡洛树搜索算法本身是一种无信息搜索方法，可以考虑加入启发式搜索来提高性能。

并行化：可以考虑将搜索过程并行化，以提高搜索速度。