**程序报告**

学号：2211999 姓名：邢清画

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)，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。

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。

设计AI 玩家，使用 『蒙特卡洛树搜索算法』 来实现 miniAlphaGo for Reversi。1、 不要修改get\_move方法的输入和输出。2、 可以添加 AIPlayer 的属性和方法。3、 完善算法时请注意落子时间：落子需要在 60s 之内！4、 落子 3 次不在合法范围内即判断该方失败, 故落子前请检查棋子的合法性。

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**）

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

1. **方法**：

本代码采用了蒙特卡洛树搜索（Monte Carlo Tree Search，MCTS）算法，结合Roxanne策略进行黑白棋游戏的AI决策。蒙特卡洛树搜索是一种自适应的搜索方法，可以在有限时间内找到近似最优解。启发式搜索策略则是通过对落子优先级进行排序，提高了搜索过程中的效率。

1. **改进**：

**模拟扩展**阶段：

代码将随机策略替换为**启发式搜索**策略，使得搜索过程更加高效。

**权重矩阵**的适应性：调整了权重矩阵以反映不同游戏阶段的战略重要性。从**单一矩阵**到在**开局、中盘和终盘**阶段分别根据棋局阶段性落子特征设置三个阶段的权重矩阵，这些权重更具体地强调了棋盘上关键位置的战略价值。**稳定性**评估（calculate\_stability）：增加了一种新的方法来评估棋子的稳定性，不仅考虑棋子**是否在边缘**，而且还考虑到棋子的**连通性**和**是否可能被对手翻转**。这种更复杂的稳定性计算方法可以帮助AI更准确地评估棋局，从而做出更好的战略决策。**行动力**评估（calculate\_mobility）：使用行动力的概念，评估对手的行动自由度。行动力指的是一个玩家可以进行的合法走步的数量。通常，限制对手的行动力被认为是一种有效的策略。通过计算对手可能的合法移动数量，AI可以策略性地限制对手的选择，增强自己的竞争优势。

**综合评估函数**（evaluate\_board）：  
将上述所有因素综合到评估函数中，使AI更能理解不同棋局阶段的策略变化、棋子的稳定性以及通过限制对手行动力的战术优势。

1. **方法局限性**：

MCTS受时间限制影响较大，因此可能无法找到全局最优解，MCTS\_search 中的 maxt 值

maxt 值在 MCTS\_search 函数中代表蒙特卡洛树搜索的迭代次数，其大小直接影响搜索深度和广度，较小的 maxt 值：会导致搜索较浅，可能无法充分探索潜在的最佳走法，但会增快搜索速度，适用于对响应时间有要求的场景；较大的 maxt 值：能够更深入地探索可能的走法，理论上能够找到更优的走法，但需要更长的计算时间，可能不适合时间敏感的应用。

判断稳定性的函数内容不够丰富，目前只考虑了边缘稳定、部分稳定两种形式，可以进一步根据棋局进行特点增加稳定性判定。

1. **优化方向**：

参数调整：尝试调整**开局、中盘和终盘**三个阶段的**权重表**以改进搜索效果；

增强启发式评估函数：添加棋局阶段的适应性权重、稳定性评估和对手行动力的考量。

优化MCTS中的模拟策略：引入更智能的棋局模拟。

稳定性考虑：增强稳定性计算的准确性和复杂度，不仅仅考虑边角，还要分析棋子的整体配置。动态调整UCB探索参数：为不同棋局阶段调整探索参数以更好地平衡探索与利用。

参数调优：实验不同的参数，如模拟的深度（在 stimulate\_policy 函数中的 cnt），或是搜索的宽度和深度（在 MCTS\_search 函数中的 maxt）。增加模拟次数：增加每个动作模拟的次数可以提供更准确的评估。动态调整模拟深度：动态调整 C 值（即探索常数），根据游戏进程和剩余时间动态调整模拟的深度。

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）

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

1. **游戏阶段权重获取**函数 get\_game\_phase\_weights 根据棋盘上的空位数量确定当前游戏的阶段（开局、中盘、终局），并为每个阶段返回不同的权重矩阵。这些权重用于评估棋盘上各个位置的战略价值。

#棋局阶段的适应性权重，开局阶段可能更重视控制中心区域，而终局阶段可能更重视角落和边缘的稳定性。def get\_game\_phase\_weights(board): # 开局阶段权重矩阵 early\_game\_weights = [ [90, -10, 15, 10, 10, 15, -10, 90],# 增强角落和中心附近的位置，减轻边缘的负面影响 [-10, -30, 5, 2, 2, 5, -30, -10],# 增加靠近中心的负位置的权重，稍微降低边缘的负面影响 [10, -2, 2, 1, 1, 2, -2, 10], [5, -2, 1, 2, 2, 1, -2, 5], [5, -2, 1, 2, 2, 1, -2, 5], [10, -2, 2, 1, 1, 2, -2, 10], [-10, -30, 5, 2, 2, 5, -30, -10], [90, -10, 15, 10, 10, 15, -10, 90] ] # 中盘阶段权重矩阵,在游戏的中盘，尤其是边缘棋子（非角落），可能因为易于被对方翻转而变得不那么有价值。 mid\_game\_weights = [ [110, -20, 20, 15, 15, 20, -20, 110],# 加强角落的吸引力，同时提高中间位置的吸引力 [-20, -40, 10, 5, 5, 10, -40, -20],# 减轻边缘位置的惩罚，提高中心接近位置的吸引力 [20, 0, 5, 2, 2, 5, 0, 20], [10, 0, 2, 5, 5, 2, 0, 10], [10, 0, 2, 5, 5, 2, 0, 10], [20, 0, 5, 2, 2, 5, 0, 20], [-20, -40, 10, 5, 5, 10, -40, -20], [110, -20, 20, 15, 15, 20, -20, 110] ] # 终局阶段权重矩阵 end\_game\_weights = [ [130, -10, 25, 20, 20, 25, -10, 130], # 极大地加强角落和边缘的吸引力，以确保稳定性 [-10, -20, 10, 5, 5, 10, -20, -10],# 减少边缘的负影响，同时提升接近中心位置的积极性 [15, -1, 5, 2, 2, 5, -1, 15], [10, -1, 2, 2, 2, 2, -1, 10], [10, -1, 2, 2, 2, 2, -1, 10], [15, -1, 5, 2, 2, 5, -1, 15], [-10, -20, 10, 5, 5, 10, -20, -10], [130, -10, 25, 20, 20, 25, -10, 130] ] empty\_count = sum(row.count('.') for row in board) # 计算空位数量 if empty\_count > 48: return early\_game\_weights elif empty\_count > 16: return mid\_game\_weights else: return end\_game\_weights

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

**2. 稳定性判定**is\_stable: 判断给定的棋子是否稳定。稳定的棋子是指在当前游戏进程中不可能被对方翻转的棋子。calculate\_stability: 遍历棋盘，使用 is\_stable 函数计算稳定棋子的数量，用于评估函数中增加稳定性的得分。#稳定的棋子是不能被对方翻转的棋子。稳定性的计算可以从四个角开始，逐步向外扩展。def is\_stable(x, y, board, color): if board[x][y] != color: return False # 方向向量定义所有可能的移动方向 directions = [(0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1)] stable = True for dx, dy in directions: nx, ny = x + dx, y + dy # 检查每个方向 while 0 <= nx < 8 and 0 <= ny < 8: if board[nx][ny] == color: nx += dx ny += dy else: # 如果在任何方向上找到非同色棋子或边界，则检查该方向相反方向是否也是非同色或边界 if board[nx][ny] != color: back\_x, back\_y = x - dx, y - dy while 0 <= back\_x < 8 and 0 <= back\_y < 8: if board[back\_x][back\_y] != color: stable = False break back\_x -= dx back\_y -= dy break if not stable: break return stabledef calculate\_stability(board, color): stable\_count = 0 for r in range(8): for c in range(8): if board[r][c] == color and is\_stable(r, c, board, color): stable\_count += 1 return stable\_count

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

1. **行动力计算**

函数 opponent\_mobility 计算对手可能的合法移动数，这是AI策略中减少对方行动力的一部分，影响总体评估得分。

#行动力是指对手可行的走步数量。减少对手行动力的核心是限制对方下一步的合法位置。def opponent\_mobility(board, color): opponent\_color = 'O' if color == 'X' else 'X' # 正确地使用 board 实例调用 get\_legal\_actions 方法 return len(list(board.get\_legal\_actions(opponent\_color)))

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

1. **棋盘评估**evaluate\_board 根据当前棋盘状态、棋子颜色、位置权重、稳定性和对手行动力综合评估棋盘，得出一个分数。这个分数用于MCTS中选择最优策略。

#将上述因素综合到评估函数中：def evaluate\_board(board, color): weights = get\_game\_phase\_weights(board) score = 0 for r in range(8): for c in range(8): if board[r][c] == color: score += weights[r][c] if is\_stable(r, c, board, color): score += 20 # 稳定棋子增加额外分数 elif board[r][c] != '.': score -= weights[r][c] # 增加稳定性评分 score += calculate\_stability(board, color) \* 20 # 减少对方的行动力 score -= opponent\_mobility(board, color) return score

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

**5. MCTS节点定义**Node 类是MCTS的基础，包含：状态管理：棋盘状态、行动方颜色、父节点、子节点、所采取的行动。访问和胜利统计：用于在MCTS中计算访问次数和胜利次数，支撑UCB计算。fully\_expanded：检查节点是否已完全扩展。class Node:# Node 类：代表搜索树中的节点 def \_\_init\_\_(self,state,color,parent = None,action = None): # 初始化节点访问次数，黑白棋胜利次数，节点奖励，棋盘状态，子节点列表，父节点，节点动作，行动方颜色 self.visit = 0 self.blackwin = 0 self.whitewin = 0 self.reward = 0.0 self.state = state self.children = [] self.parent = parent self.action = action self.color = color def add\_child\_nodes(self,new\_state,action,color): #添加子节点 child\_node = Node(new\_state,parent=self,action = action,color=color) self.children.append(child\_node) def fully\_expanded(self): # 检查是否所有合法动作都已扩展为子节点 cnt\_max = len(list(self.state.get\_legal\_actions(self.color))) cnt\_now = len(self.children) if(cnt\_max <= cnt\_now): return True else: return False====================================================================

**6.AI玩家**AIPlayer 类实现了MCTS搜索和棋盘评估的集成，主要方法包括：

get\_move：根据当前棋盘状态，使用MCTS搜索算法找到**最佳移动**。  
# -----------------请实现你的算法代码--------------------------------------action = Noneroot\_board = copy.deepcopy(board)root = Node(state=root\_board,color=self.color)action = self.MCTS\_search(root)----------------------------------------------------------------------------------------------------------------------

MCTS\_search：执行MCTS算法的核心循环，包括**选择、扩展、模拟和后向传播**。

# 蒙特卡洛树搜索，实现MCTS搜索，返回最佳动作。def MCTS\_search(self,root,maxt = 200): for t in range(maxt): leave = self.select\_policy(root) blackwin,whitewin = self.stimulation\_strategy(leave) self.back\_propagation(leave,blackw=blackwin,whitew=whitewin) return self.dynamic\_UCB(root).action----------------------------------------------------------------------------------------------------------------------select\_policy 和 expand：用于在MCTS中根据UCB值**选择和扩展**节点。# 扩展节点，当节点未完全扩展时，随机选择一个未探索的动作创建新的子节点。def expand(self,node): actions\_available = list(node.state.get\_legal\_actions(node.color)) actions\_already = [c.action for c in node.children] if(len(actions\_available)==0): return node.parent action = random.choice(actions\_available) while action in actions\_already: action=random.choice(actions\_available) new\_state = copy.deepcopy(node.state) new\_state.\_move(action,node.color) new\_state.display() new\_color = self.color\_reversal(node.color) node.add\_child\_nodes(new\_state,action = action,color= new\_color) return node.children[-1]# 选择策略，选择过程，从根节点开始，根据是否已完全扩展和UCB值选择或扩展节点。def select\_policy(self,node): def select\_policy(self,node): while(not self.if\_the\_end(node.state)): if(len(list(node.state.get\_legal\_actions(node.color)))==0): return node; elif(not node.fully\_expanded()): print("need to expand") new\_node = self.expand(node) print("the end of expand") return new\_node else: print("fully expaned") node.state.display() print(len(node.children)) print(list(node.state.get\_legal\_actions(node.color))) node = self.dynamic\_UCB(node) return node----------------------------------------------------------------------------------------------------------------------

stimulate\_policy：从当前节点出发**模拟**随机游戏至终局，用于评估动作结果。

# 模拟策略，从给定节点开始，随机模拟直到游戏结束，返回模拟结束时的分数（黑棋和白棋的棋子数）。def stimulation\_strategy(self, node): board = copy.deepcopy(node.state) # 获取当前棋盘状态的深拷贝 color = copy.deepcopy(node.color) # 当前落子颜色 cnt = 0 while not self.if\_the\_end(board): actions = list(board.get\_legal\_actions(color)) if not actions: color = self.color\_reversal(color) # 没有合法动作时切换颜色 else: action = random.choice(actions) board.\_move(action, color) # 执行落子动作 color = self.color\_reversal(color) # 切换颜色 cnt += 1 if cnt > 19: break # 正确计算并返回黑棋和白棋的数量 black\_count = board.count('X') white\_count = board.count('O') return black\_count, white\_count----------------------------------------------------------------------------------------------------------------------

back\_propagate：从模拟结束的节点向根节点**回传游戏结果**，更新节点统计。# 后向传播，从给定节点向上回溯至根节点，更新路径上每个节点的访问次数和胜利次数。def back\_propagation(self,node,blackw,whitew): while(node is not None): node.visit+=1 node.blackwin+=blackw node.whitewin+=whitew node = node.parent return 0----------------------------------------------------------------------------------------------------------------------

calculate\_dynamic\_C：**动态计算探索参数C**，用于调整探索和利用的平衡。def calculate\_dynamic\_C(self, node, base\_C=1.414): # 基于节点访问次数动态调整C值 return base\_C / (1 + 0.1 \* math.log1p(node.visit)) # 基于搜索深度动态调整C值（如果有存储深度信息） # return base\_C / (1 + 0.01 \* node.depth)----------------------------------------------------------------------------------------------------------------------

ucb：计算UCB值，选择最佳子节点进行进一步探索。

# UCB计算，使用UCB公式选择具有最大期望奖励的子节点。def dynamic\_UCB(self, node): max\_score = -float('inf') best\_children = [] for child in node.children: exploit = child.blackwin / child.visit if child.color == 'O' else child.whitewin / child.visit dynamic\_C = self.calculate\_dynamic\_C(node) explore = math.sqrt(2.0 \* math.log(node.visit) / child.visit) \* dynamic\_C uct\_score = exploit + explore if uct\_score > max\_score: best\_children = [child] max\_score = uct\_score elif uct\_score == max\_score: best\_children.append(child) return random.choice(best\_children) if best\_children else None

1. **实验结果**

（实验结果，必填）

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





关键参数：cnt19,maxt200, base\_C=1.414，C 值的减少速度0.1.（或提高至1.65，缩小至0.05）

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**）

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

1. 达到目标预期

在实现基于蒙特卡洛树搜索（MCTS）的AI玩家过程中，整体目标是建立一个能够进行有效决策的系统，使其在不同的游戏阶段能根据棋局状态作出合理的走棋选择。已经具备了以下功能：动态权重评估，针对不同的游戏阶段调整棋盘位置的价值。稳定性评估，识别并优化那些在游戏中无法被对手翻转的棋子。动态调整探索参数 C 值，以适应不确定性的环境。实现了完整的MCTS流程，包括节点选择、扩展、模拟和后向传播。2. 改进的方向稳定性评估优化：当前稳定性的评估过于简化，实际应用中可以考虑更复杂的稳定性算法，如基于模式识别的稳定性评价方法，可能会提高AI的棋局分析精度。参数调整与优化：动态调整 C 值的策略还可以根据不同游戏的特定情况进行进一步的微调，例如考虑对手的行为模式、当前的棋局平衡状态等因素。3. 实现过程中遇到的困难模拟策略的选择：选择一个合理的模拟策略，以快速有效地评估节点的潜在价值，当前的启发式模拟可能不足以捕捉复杂的游戏动态。参数调整：C 值的动态调整需要平衡探索与利用，找到最佳的平衡点，尤其是在高不确定性环境下。4. 提升性能

更精确的启发式评估函数：确保评估函数能够准确反映不同棋局配置的优劣，包括对棋局的动态特性的深入分析，如稳定性、控制中心等因素的加权。策略的多样性：在模拟过程中使用多种策略，例如在不同情况下选择不同的启发式规则，探索更多元的棋局可能性。学习机制的引入：考虑实现一些机器学习算法，通过对历史数据的学习来优化启发式规则，或者动态调整策略参数。