**程序报告**

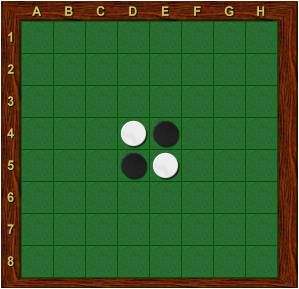
学号： 2212998 姓名：胡博浩

1. **问题重述**

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，变为己方棋子，故又称“翻转棋”。棋子双面为红、绿色的成为“苹果棋”。它使用8\*8的棋盘，由两人执黑子和白子轮流下棋，最后子多方为胜。

游戏规则：

棋局开始时黑棋位于E4和D5，白棋位于D4和E5，如下图所示



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。

1. **设计思想**

1.方法：

a)蒙特卡洛树搜索算法：

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。MCTS主要进行若干次模拟，将结果保存到蒙特卡洛树中，根据模拟结果选择最佳的动作。MCTS的一轮迭代共有四个阶段：选择，扩展，模拟，反向传播。

①选择：从根节点开始，根据UCB的大小选择一个UCB最大的子结点，直到到达叶子节点或具有还未被扩展的子节点。

②扩展：在选择阶段结束时候，如果还没有到达终止状态，那么我们就要对这个节点进行扩展，扩展出一个或多个节点（也就是进行一个可能的action然后进入下一个状态）。

③模拟：从上一环节要被扩展的节点出发，模拟扩展搜索树。在本实验中，采用随机模拟的方法，即从该节点出发，随机走子，直到走子结束。

④反向传播：根据上一环节的模拟结果回溯更新模拟路径上的奖励均值和被访问次数。

b)启发函数：

在模拟对局结束后，通过启发函数计算当前棋盘的分数。启发式函数主要采用3种策略进行组合。第一种策略是静态矩阵赋值，第二种是稳定子，第三种是自由度和棋子数量差。在启发式函数中，首先进行稳定子的赋值，分数保存在变量 value 里面。此后，如果棋盘上的棋子总数少于46个，则value首先加上2倍的自由度，然后再根据棋子数量决定是否要加上棋子数量差。如果总棋子数多于46个，则不再顾及自由度和棋子数量差，直接根据静态矩阵决定每个位置的分数。

c)排序函数：

排序函数是在获得当前棋盘可以走动的合法落子点集合后，对合法点进行一个排序。在进行扩展的时候，通过调用排序函数，可以优先扩展权重较大的节点，从而提高模拟效率。

d)蒙特卡洛搜索次数与时间：

黑白棋规则限制落子时间在60s内，若固定搜索次数则可能会超时或未能完全利用60s，因此采用time.time()函数记录时间并控制整个落子过程小于60s

e)模拟对局总步数限制：

因为模拟直到对局结束太耗时间，所以要在模拟到一定步数的时候停止模拟，保证模拟效率。

2.伪代码：

a)初始化AI玩家,开始对局

b)在一定迭代次数内进行蒙特卡洛树搜索

递归选择ucb最大的子节点

扩展选择的节点，使用排序函数

随机玩家进行模拟，并通过启发函数计算价值

反向传播更新节点信息

c)根据搜索结果结合ucb选择最佳落子位置

3.方法局限性：

MCTS受迭代次数影响较大，在一定次数限制内无法找到全局最优解。并且，启发函数也无法保证找到最优解，甚至误导搜索结果。

4.优化方向：

a)参数调整：通过超参数搜索寻找最优参数组合，以改进搜索效果。

b)框架调整：可以尝试引入其他启发式搜索算法，例如alpha-beta剪枝等。

c)强化学习：结合神经网络模型，对AI进行训练，提升算法的稳定性。

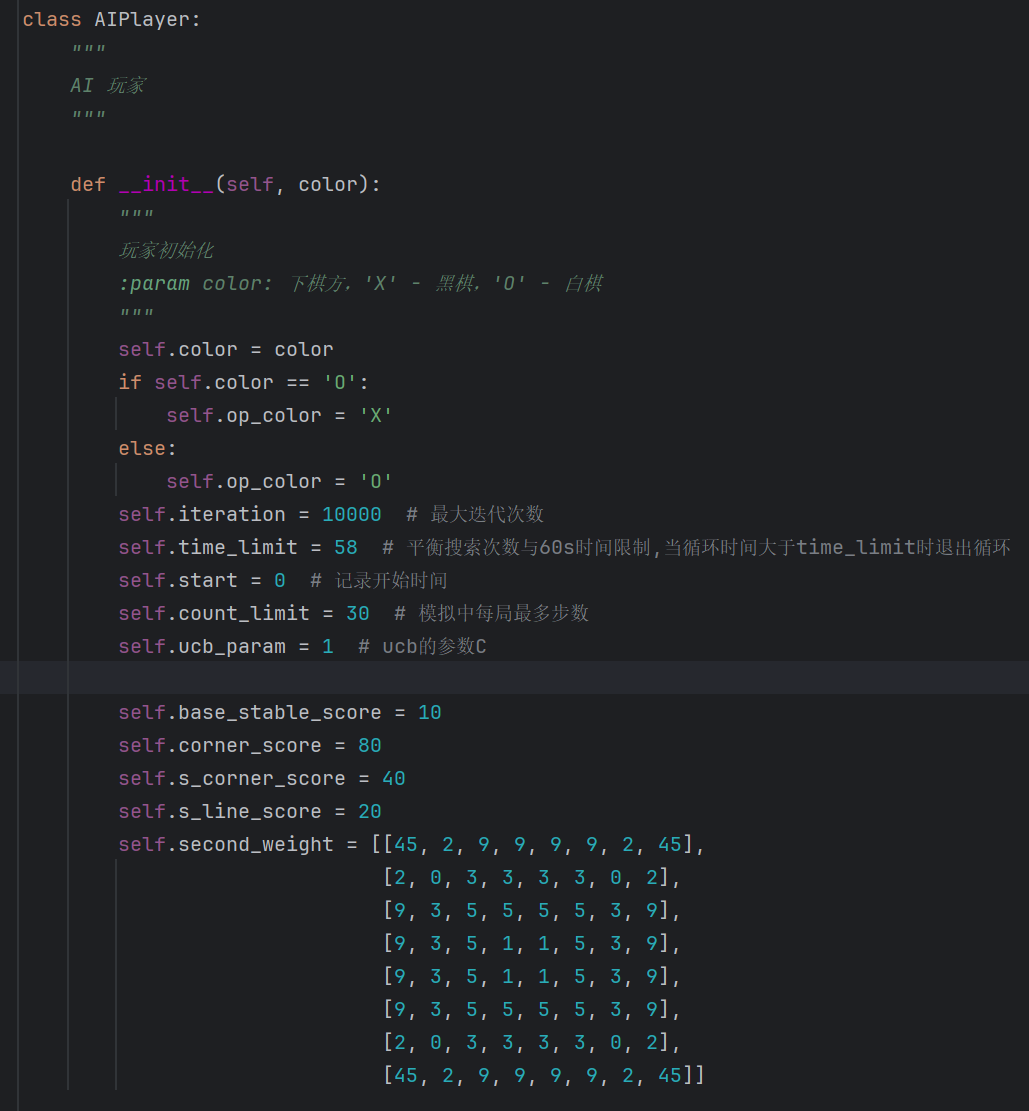
1. **代码解释**

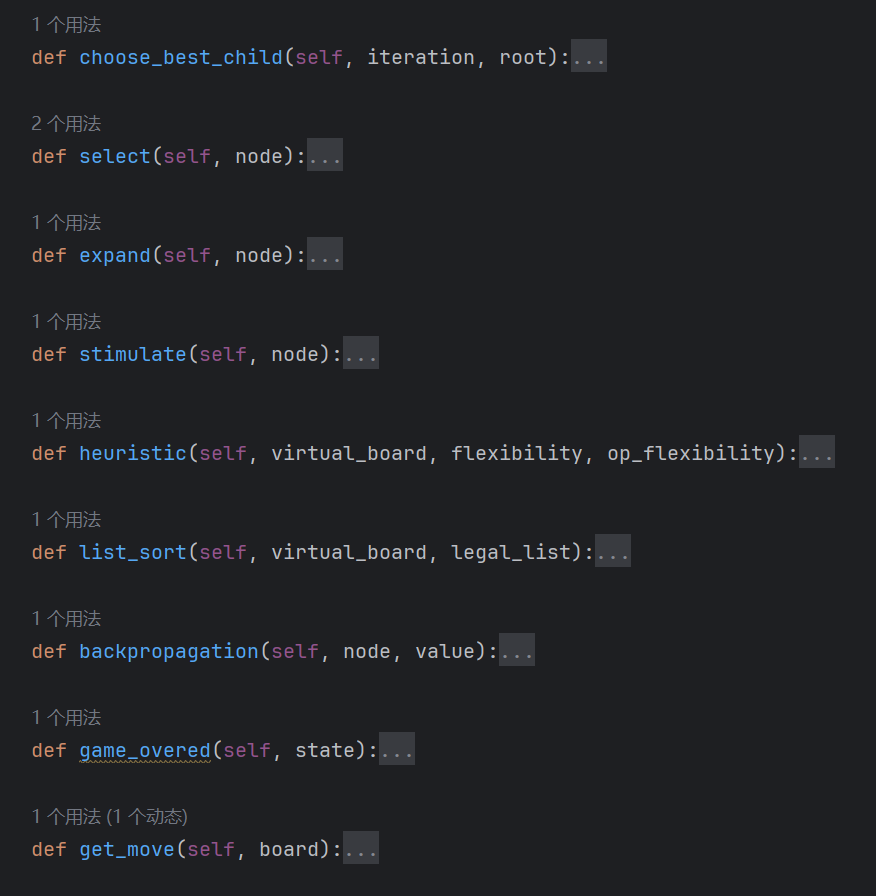
1.整体代码结构：

a)节点类：



b)AI类：





2.重点代码解析

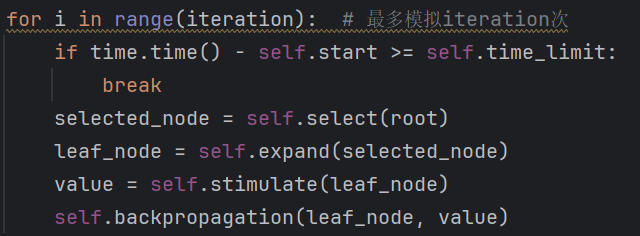
a)full\_expand(self)

判断节点是否完全扩展:如果当前节点的孩子数等于所有合法的可能坐标，则判定这个节点已经被完全扩展

b)搜索核心框架

在每次搜索过程中，依次执行选择、扩展、模拟、反向传播，直至达到最大搜索次数。随着搜索次数的增加，搜索树的规模不断增大。在所有搜索都结束后，返回根节点的最好的子节点作为本次决策的结果。

**优化：**为平衡搜索次数与60s时间限制，使用time.time()函数记录时间，当循环时间大于58s时退出循环，根据UCB的值计算出下一步最佳落子位置。



c)选择：

从node节点开始向下选择，分为以下几种情况：

（1）节点所在棋局没有可以走子的点了（节点为终局节点），选择结束，直接进入反向传播，根据该结局来向上更新；

（2）节点还未完全扩展，则需要扩展该点，执行 expansion 扩展函数；

（3）节点已经完全扩展，继续选择这个它的子节点里ucb最大的点，执行 best\_ucb 函数；

d) 扩展

扩展函数实现对未完全扩展 / 叶节点的扩展。

扩展函数对 node 讨论以下情况：

（1）node 对应的棋局没有可以走子的位置了：返回node的父节点

（2）node 还有未扩展过的走子位置（即还存在可走子位置没被放进children里）：

随机选择一个未扩展过的子节点，并把该子节点的扩展结果放进 node 的 children 里。返回新扩展的子节点。

**优化：**通过调用排序函数，优先访问权重较大的节点，在时间不足的情况下可以首先搜索比较可能的点，得到更优化的点。



e)模拟

模拟过程采取随即策略，通过current\_node.isfinish()判断是否到终结节点，到达终结节点后，返回value的值作为奖励.

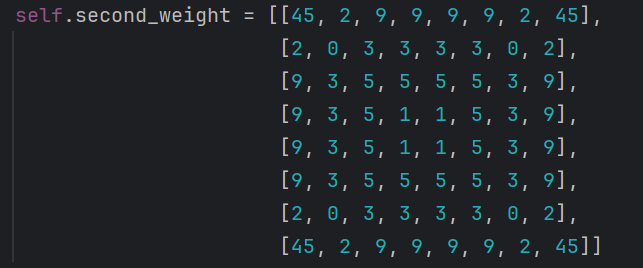
**优化：**通过启发函数计算当前棋盘的分数。

f) 判断游戏是否结束：根据当前棋盘，判断棋局是否终止，如果当前选手没有合法下棋的位子，则切换选手；如果另外一个选手也没有合法的下棋位置，则比赛停止

g) 为棋盘赋权：

(1)使一开始黑棋先占满中心两圈方框,把对方逼出方框

(2)当多个合法落子点reeward值相近时优先选择靠边，尤其是四个角



h)启发式函数策略：

启发式函数主要采用3种策略进行组合。第一种策略是静态矩阵赋值，第二

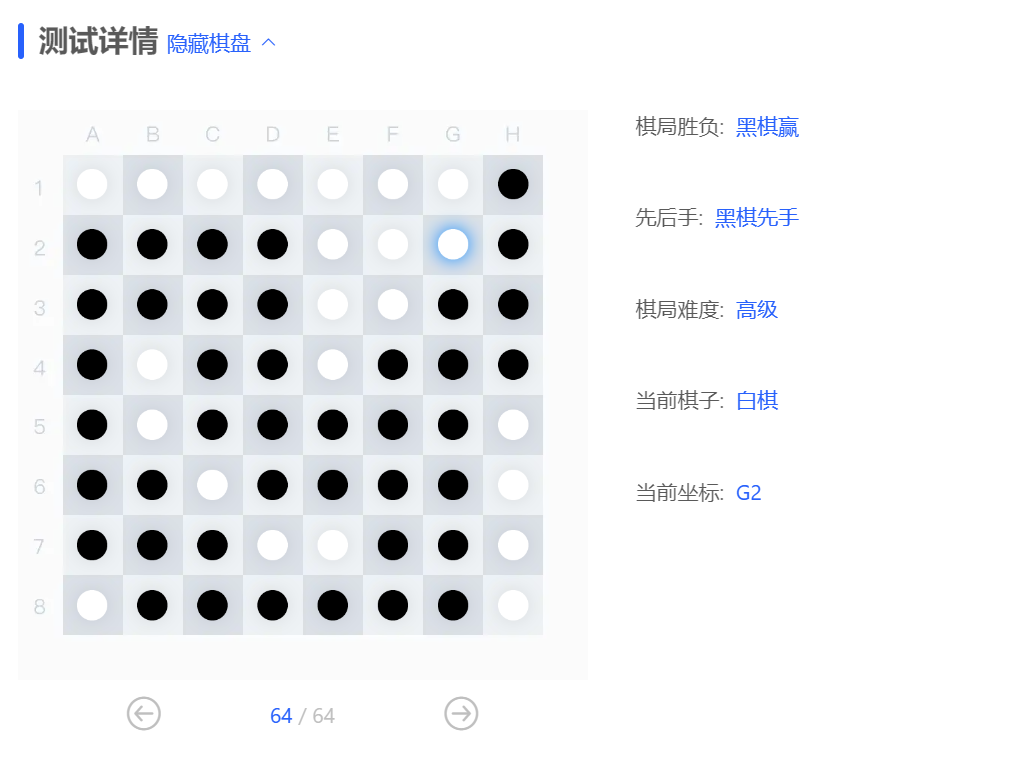
种是稳定子，第三种是自由度。

1. **代码内容**

import math  
import random  
import sys  
import time  
from copy import deepcopy  
  
  
class MCT\_node:  
 def \_\_init\_\_(self, state, parent=None, action=None, color=""):  
 self.color = color # 该节点玩家颜色  
 self.parent = parent # 父节点  
 self.state = state # 棋盘状态  
 self.value = 0.00 # 期望值  
 self.visits\_num = 0 # 访问次数  
 self.action = action # 对应动作  
 self.children = [] # 孩子节点  
  
 def get\_ucb(self, ucb\_param):  
 *"""  
 计算UCB值  
 UCB = Q/N + C \* sqrt(2 \* ln(P) / N)  
 """* if self.visits\_num == 0:  
 return sys.maxsize # 未访问的节点ucb为无穷大  
  
 # UCB公式  
 explore = math.sqrt(2.0 \* math.log(self.parent.visits\_num) / float(self.visits\_num))  
 now\_ucb = self.value / self.visits\_num + ucb\_param \* explore  
 return now\_ucb  
  
 def add\_child(self, child\_state, action, color):  
 child\_node = MCT\_node(child\_state, parent=self, action=action, color=color)  
 self.children.append(child\_node)  
  
 def full\_expand(self):  
 *"""  
 判断是否完全扩展  
 """* # 有孩子并且所有孩子都访问过了就是完全扩展  
 if len(self.children) == 0:  
 return False  
 for kid in self.children:  
 if kid.visits\_num == 0:  
 return False  
 return True  
  
  
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  
 if self.color == 'O':  
 self.op\_color = 'X'  
 else:  
 self.op\_color = 'O'  
 self.iteration = 3000 # 最大迭代次数  
 self.time\_limit = 58 # 平衡搜索次数与60s时间限制,当循环时间大于time\_limit时退出循环  
 self.start = 0 # 记录开始时间  
 self.count\_limit = 50 # 模拟中每局最多步数  
 self.ucb\_param = 1 # ucb的参数C  
  
 self.base\_stable\_score = 10  
 self.corner\_score = 80  
 self.s\_corner\_score = 40  
 self.s\_line\_score = 20  
 self.second\_weight = [[45, 2, 9, 9, 9, 9, 2, 45],  
 [2, 0, 3, 3, 3, 3, 0, 2],  
 [9, 3, 5, 5, 5, 5, 3, 9],  
 [9, 3, 5, 1, 1, 5, 3, 9],  
 [9, 3, 5, 1, 1, 5, 3, 9],  
 [9, 3, 5, 5, 5, 5, 3, 9],  
 [2, 0, 3, 3, 3, 3, 0, 2],  
 [45, 2, 9, 9, 9, 9, 2, 45]]  
  
 def choose\_best\_child(self, iteration, root):  
 *"""  
 根据当前棋盘状态获取最佳落子位置  
 :param iteration: 最大搜索次数  
 :param root: 根节点  
 :return: action 最佳落子位置  
 """* for i in range(iteration): # 最多模拟iteration次  
 if time.time() - self.start >= self.time\_limit:  
 break  
 selected\_node = self.select(root)  
 leaf\_node = self.expand(selected\_node)  
 value = self.stimulate(leaf\_node)  
 self.backpropagation(leaf\_node, value)  
  
 max\_node = None # 搜索完成，然后找出最适合的下一步  
 max\_ucb = -sys.maxsize  
 for child in root.children:  
 child\_ucb = child.get\_ucb(self.ucb\_param)  
 if max\_ucb < child\_ucb:  
 max\_ucb = child\_ucb  
 max\_node = child # max\_node指向ucb最大的孩子  
 return max\_node.action  
  
 def select(self, node):  
 *"""  
 :param node:某个节点  
 :return: ucb值最大的叶子  
 """* if len(node.children) == 0: # 叶子，需要扩展  
 return node  
 elif not node.full\_expand():  
 for kid in node.children: # 从左开始遍历  
 if kid.visits\_num == 0:  
 return kid  
 else:  
 max\_node = None  
 max\_ucb = -sys.maxsize  
 for child in node.children:  
 child\_ucb = child.get\_ucb(self.ucb\_param)  
 if max\_ucb < child\_ucb:  
 max\_ucb = child\_ucb  
 max\_node = child # max\_node指向ucb最大的孩子  
 return self.select(max\_node)  
  
 def expand(self, node):  
 if node.visits\_num == 0:  
 return node # 如果节点未被访问过，则不扩展，直接返回该节点进行模拟  
 else: # 需要扩展,先确定颜色  
 new\_color = "O" if node.color == "X" else "X"  
 l = list(node.state.get\_legal\_actions(node.color))  
 if len(l) == 0:  
 return node  
 self.list\_sort(node.state, l) # 首先搜索比较可能的点，得到更优化的点  
 for action in l: # 把所有可行节点加入孩子列表，并初始化  
 new\_board = deepcopy(node.state)  
 new\_board.\_move(action, node.color)  
 # 新建节点  
 node.add\_child(new\_board, action=action, color=new\_color)  
 return node.children[0] # 返回新的孩子列表的第一个，以供下一步模拟  
  
 def stimulate(self, node):  
 *"""  
 :param node:模拟起始点  
 :return: 模拟结果reward  
 board.get\_winner()会返回胜负关系和获胜子数  
 考虑胜负关系和获胜的子数，定义获胜积win\_reword分，每多赢一个棋子多1分  
 """* board = deepcopy(node.state)  
 color = node.color  
 count = 0  
 while (not self.game\_overed(board)) and count < self.count\_limit: # 游戏没有结束，就模拟下棋  
 action\_list = list(node.state.get\_legal\_actions(color))  
 if not len(action\_list) == 0: # 可以下，就随机下棋  
 action = random.choice(action\_list)  
 board.\_move(action, color)  
 color = "O" if color == "X" else "X"  
 else: # 不能下，就交换选手  
 color = "O" if color == "X" else "X"  
 action\_list = list(node.state.get\_legal\_actions(color))  
 action = random.choice(action\_list)  
 board.\_move(action, color)  
 color = "O" if color == "X" else "X"  
 count = count + 1  
  
 if self.color == 'X':  
 legal\_list = list(board.get\_legal\_actions('X'))  
 op\_legal\_list = list(board.get\_legal\_actions('O'))  
 else:  
 legal\_list = list(board.get\_legal\_actions('O'))  
 op\_legal\_list = list(board.get\_legal\_actions('X'))  
 value = self.heuristic(board, len(legal\_list), len(op\_legal\_list))  
  
 return -value  
  
 def heuristic(self, virtual\_board, flexibility, op\_flexibility):  
 # 通过启发函数计算当前棋盘的分数  
 board = virtual\_board.\_board  
 value = 0  
 s1, s2 = virtual\_board.count(self.op\_color), virtual\_board.count(self.color)  
 s\_sum = s1 + s2  
 s\_diff = s1 - s2  
  
 if board[0][0] == self.color:  
 value += self.corner\_score  
 for i in range(1, 8):  
 if board[0][i] == self.color:  
 value += self.base\_stable\_score  
 else:  
 break  
 for i in range(1, 8):  
 if board[i][0] == self.color:  
 value += self.base\_stable\_score  
 else:  
 break  
 else:  
 if board[1][1] == self.color:  
 value -= self.s\_corner\_score  
 if board[0][1] == self.color:  
 value -= self.s\_line\_score  
 if board[1][0] == self.color:  
 value -= self.s\_line\_score  
  
 if board[7][0] == self.color:  
 value += self.corner\_score  
 for i in range(1, 8):  
 if board[7][i] == self.color:  
 value += self.base\_stable\_score  
 else:  
 break  
 for i in range(6, -1, -1):  
 if board[i][0] == self.color:  
 value += self.base\_stable\_score  
 else:  
 break  
 else:  
 if board[6][1] == self.color:  
 value -= self.s\_corner\_score  
 if board[7][1] == self.color:  
 value -= self.s\_line\_score  
 if board[6][0] == self.color:  
 value -= self.s\_line\_score  
  
 if board[0][7] == self.color:  
 value += self.corner\_score  
 for i in range(6, -1, -1):  
 if board[0][i] == self.color:  
 value += self.base\_stable\_score  
 else:  
 break  
 for i in range(1, 8):  
 if board[i][7] == self.color:  
 value += self.base\_stable\_score  
 else:  
 break  
 else:  
 if board[1][6] == self.color:  
 value -= self.s\_corner\_score  
 if board[0][6] == self.color:  
 value -= self.s\_line\_score  
 if board[1][7] == self.color:  
 value -= self.s\_line\_score  
  
 if board[7][7] == self.color:  
 value += self.corner\_score  
 for i in range(6, -1, -1):  
 if board[7][i] == self.color:  
 value -= self.base\_stable\_score  
 else:  
 break  
 for i in range(6, -1, -1):  
 if board[i][7] == self.color:  
 value += self.base\_stable\_score  
 else:  
 break  
 else:  
 if board[6][6] == self.color:  
 value -= self.s\_corner\_score  
 if board[7][6] == self.color:  
 value -= self.s\_line\_score  
 if board[6][7] == self.color:  
 value -= self.s\_line\_score  
  
 if board[0][0] == self.op\_color:  
 value -= self.corner\_score  
 for i in range(1, 8):  
 if board[0][i] == self.op\_color:  
 value -= self.base\_stable\_score  
 else:  
 break  
 for i in range(1, 8):  
 if board[i][0] == self.op\_color:  
 value -= self.base\_stable\_score  
 else:  
 break  
 else:  
 if board[1][1] == self.op\_color:  
 value += self.s\_corner\_score  
 if board[0][1] == self.op\_color:  
 value += self.s\_line\_score  
 if board[1][0] == self.op\_color:  
 value += self.s\_line\_score  
  
 if board[7][0] == self.op\_color:  
 value -= self.corner\_score  
 for i in range(1, 8):  
 if board[7][i] == self.op\_color:  
 value -= self.base\_stable\_score  
 else:  
 break  
 for i in range(6, -1, -1):  
 if board[i][0] == self.op\_color:  
 value -= self.base\_stable\_score  
 else:  
 break  
 else:  
 if board[6][1] == self.op\_color:  
 value += self.s\_corner\_score  
 if board[7][1] == self.op\_color:  
 value += self.s\_line\_score  
 if board[6][0] == self.op\_color:  
 value += self.s\_line\_score  
  
 if board[0][7] == self.op\_color:  
 value -= self.corner\_score  
 for i in range(6, -1, -1):  
 if board[0][i] == self.op\_color:  
 value -= self.base\_stable\_score  
 else:  
 break  
 for i in range(1, 8):  
 if board[i][7] == self.op\_color:  
 value -= self.base\_stable\_score  
 else:  
 break  
 else:  
 if board[1][6] == self.op\_color:  
 value += self.s\_corner\_score  
 if board[0][6] == self.op\_color:  
 value += self.s\_line\_score  
 if board[1][7] == self.op\_color:  
 value += self.s\_line\_score  
  
 if board[7][7] == self.op\_color:  
 value -= self.corner\_score  
 for i in range(6, -1, -1):  
 if board[7][i] == self.op\_color:  
 value -= self.base\_stable\_score  
 else:  
 break  
 for i in range(6, -1, -1):  
 if board[i][7] == self.op\_color:  
 value -= self.base\_stable\_score  
 else:  
 break  
 else:  
 if board[6][6] == self.op\_color:  
 value += self.s\_corner\_score  
 if board[7][6] == self.op\_color:  
 value += self.s\_line\_score  
 if board[6][7] == self.op\_color:  
 value += self.s\_line\_score  
  
 if flexibility == 0:  
 flexibility = -100  
  
 if s\_sum < 46:  
 value += int(2.3 \* flexibility) - op\_flexibility  
 if s\_sum < 30:  
 value += s\_diff  
 else:  
 for i in range(8):  
 for j in range(8):  
 if board[i][j] == self.color:  
 value += self.second\_weight[i][j]  
 elif board[i][j] == self.op\_color:  
 value -= self.second\_weight[i][j]  
  
 return value / 10  
  
 def list\_sort(self, virtual\_board, legal\_list):  
 # 排序函数是在获得当前棋盘可以走动的合法落子点集合后，对合法点进行一个排序，在时间不足的情况下可以首先搜索比较可能的点，得到更优化的点  
 for i in range(len(legal\_list)):  
 for j in range(len(legal\_list) - i - 1):  
 x1, y1 = virtual\_board.board\_num(legal\_list[j])  
 x2, y2 = virtual\_board.board\_num(legal\_list[j + 1])  
 if self.second\_weight[x1][y1] < self.second\_weight[x2][y2]:  
 tmp = legal\_list[j]  
 legal\_list[j] = legal\_list[j + 1]  
 legal\_list[j + 1] = tmp  
  
 def backpropagation(self, node, value):  
 *"""  
 反向传播  
 自底向上直到根节点进行枚举这一条树链上的所有节点，更新其值  
 """* while node is not None:  
 node.visits\_num += 1  
 if node.color == self.color:  
 node.value += value  
 else:  
 node.value -= value  
 node = node.parent  
 return 0  
  
 def game\_overed(self, state):  
 *"""  
 判断游戏是否结束  
 :return: True/False 游戏结束/游戏没有结束  
 """* # 根据当前棋盘，双方都无处可落子，则终止  
 b\_list = list(state.get\_legal\_actions("X"))  
 w\_list = list(state.get\_legal\_actions("O"))  
 return len(b\_list) == 0 and len(w\_list) == 0  
  
 def get\_move(self, board):  
 *"""  
 根据当前棋盘状态获取最佳落子位置  
 :param board: 棋盘  
 :return: action 最佳落子位置, e.g. 'A1'  
 """* self.start = time.time()  
 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))  
  
 # -----------------请实现你的算法代码--------------------------------------  
 root = MCT\_node(state=deepcopy(board), color=self.color)  
  
 action = self.choose\_best\_child(self.iteration, root)  
 # ------------------------------------------------------------------------  
  
 return action

1. **实验结果**

1.黑棋先手：黑棋赢



2.白棋后手：白棋赢



1. **总结**

1.是否达到目标预期：

本程序考虑了蒙特卡洛树的的不足，通过给棋盘赋值，在选择的时候对合法落子进行排序，解决了选择扩展时的不足；结合启发函数的思想，精心设计了合适的计分策略，对每次模拟对局的结果进行了合理的判断，弥补了模拟时的缺陷。因此，实现了一个相对高效智能的AI。

如图，提交代码170次！



2.优化方向：

a)可以尝试通过机器学习方法,对AI进行训练，以达到更好的效果。

b)考虑使用并行计算或GPU加速来加快搜索速度，提高搜索的深度和广度。

c)引入其他启发式搜索算法，如剪枝搜索，以弥补蒙特卡洛数搜索的不足。

d)通过超参数搜索，寻找最佳的参数组合，达到最好的效果。

3.实现过程中遇到的困难：

a)如何设计评价函数:由于本题中，黑白棋子的个数更能体现最终结果，并且在模拟阶段受走棋次数限制，不一定能够完整走完棋局，所以在模拟对局的时候，关注每局的棋子数量差；另外，黑白棋中角落的位置更具有优势，因此设计棋盘权重，以期获得较好的搜索结果。

b)如何平衡探索和择优：没想到，直接用老师给的c=1

c)如何寻找最佳的参数组合:不会，人工尝试...

4.模型的超参数和框架搜索是否合理：

当前的超参数设置和框架搜索是基于经验的，并非最优解。可以通过超参数搜索、机器学习等方法，尝试寻找最佳的参数组合。也可以结合其他的启发式算法和强化学习的方法，对框架进行调整，以优化AI的性能。