**黑白棋问题程序报告**

学号： 2112213 姓名：冯思程

1. **问题重述**

问题重述：本次实验是对黑白棋 (Reversi)的实现。

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) 。

以下是黑白棋的规则：

1.黑方先行，双方交替下棋。

2.一步合法的棋步包括：

在一个空格新落下一个棋子，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。

问题背景：基于python语言进行的编程实验，将基于python3完成实验。使用 『蒙特卡洛树搜索算法』 实现 miniAlphaGo for Reversi。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**

**方法：**

本题采用的主要算法是蒙特卡洛树算法，蒙特卡洛树算法思路包括四个过程：

1.选择（Selection）：

从根结点R开始，选择连续的子结点向下至叶子结点L。下面的结点有更多选择子结点的方法，使游戏树向最优点扩展移动，这是蒙特卡洛树搜索的本质。

2.扩展 (expansion）：

除非任意一方的输赢导致游戏结束，否则L会创建一个或多个子结点或从结点C中选择。

1. 模拟/仿真（Simulation）：

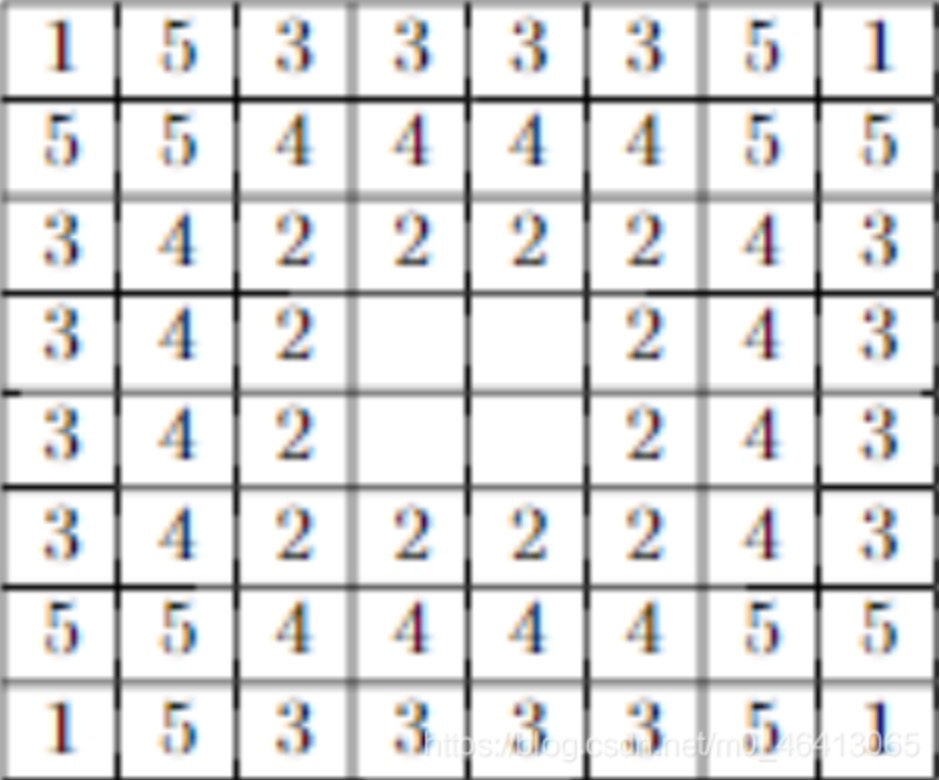
在结点C中进行随机布局。

1. 回溯/反向传播（Backpropagation）：

使用布局结果更新从C到R的路径上的结点信息。

**优化方向：**

将随机确定合法位置策略换成Roxanne策略，这个优先级表（下图）由Roxanne提出，结合了多种策略，同时也结合了Mobility的特性，因为中间子的优先级较高，会提高自己的Mobility而限制对手的可走步数。



**局限性：**

1.时间和计算资源的限制：

MCTS算法需要进行大量的模拟来搜索最优解，每个节点需要进行多次模拟来评估该节点的价值，因此在时间和计算资源受限的情况下，可能无法搜索到足够深的状态，从而影响AI的性能。此外，黑白棋这类博弈类游戏通常有很高的分支因子，因为每个棋子都有多种落子位置可选，这也增加了MCTS算法的搜索难度和计算量。因此，为了获得更好的性能，需要设计高效的数据结构和算法，以及针对具体问题的优化策略。

2.对手策略的影响：

MCTS算法的性能和搜索效果可能会受到对手策略的影响，对手策略的变化可能导致AI的性能发生变化。例如，在黑白棋中，如果对手有明显的优势或劣势，那么AI可能需要调整策略来适应对手的行为。此外，对手的策略可能是未知的，因此AI需要能够适应各种不同的对手策略，而不是针对某一个固定的策略进行优化。

3对状态空间的要求：

MCTS需要将游戏状态表示为一棵搜索树，而某些游戏状态空间可能过于庞大，无法通过搜索树来表示和搜索，从而限制了MCTS算法的应用范围。在黑白棋这类博弈类游戏中，状态空间通常是离散的，因为每个棋子都只有有限的落子位置，但是状态空间的规模仍然可能非常大。

1. **代码内容**

**首先是一个silentgame类，用于重构游戏类**，模拟在游戏中，不实时打印棋盘。代码如下：（其中除了对初始化函数进行了微小实时修改，其他与game类函数一致。

class SilentGame(object):

''' 重构游戏类，模拟下棋过程中，不实时打印棋盘 '''

def \_\_init\_\_(self, black\_player, white\_player, board=Board(), current\_player=None):

self.board = deepcopy(board) # 棋盘

# 定义棋盘上当前下棋棋手，先默认是 None

self.current\_player = current\_player

self.black\_player = black\_player # 黑棋一方

self.white\_player = white\_player # 白棋一方

self.black\_player.color = "X"

self.white\_player.color = "O"

def switch\_player(self, black\_player, white\_player):

"""

游戏过程中切换玩家

:param black\_player: 黑棋

:param white\_player: 白棋

:return: 当前玩家

"""

# 如果当前玩家是 None 或者 白棋一方 white\_player，则返回 黑棋一方 black\_player;

if self.current\_player is None:

return black\_player

else:

# 如果当前玩家是黑棋一方 black\_player 则返回 白棋一方 white\_player

if self.current\_player == self.black\_player:

return white\_player

else:

return black\_player

def print\_winner(self, winner):

"""

打印赢家

:param winner: [0,1,2] 分别代表黑棋获胜、白棋获胜、平局3种可能。

:return:

"""

print(['黑棋获胜!', '白棋获胜!', '平局'][winner])

def force\_loss(self, is\_timeout=False, is\_board=False, is\_legal=False):

"""

落子3个不合符规则和超时则结束游戏,修改棋盘也是输

:param is\_timeout: 时间是否超时，默认不超时

:param is\_board: 是否修改棋盘

:param is\_legal: 落子是否合法

:return: 赢家（0,1）,棋子差 0

"""

if self.current\_player == self.black\_player:

win\_color = '白棋 - O'

loss\_color = '黑棋 - X'

winner = 1

else:

win\_color = '黑棋 - X'

loss\_color = '白棋 - O'

winner = 0

if is\_timeout:

print('\n{} 思考超过 60s, {} 胜'.format(loss\_color, win\_color))

if is\_legal:

print('\n{} 落子 3 次不符合规则,故 {} 胜'.format(loss\_color, win\_color))

if is\_board:

print('\n{} 擅自改动棋盘判输,故 {} 胜'.format(loss\_color, win\_color))

diff = 0

return winner, diff

def run(self):

"""

运行游戏

:return:

"""

# 定义统计双方下棋时间

total\_time = {"X": 0, "O": 0}

# 定义双方每一步下棋时间

step\_time = {"X": 0, "O": 0}

# 初始化胜负结果和棋子差

winner = None

diff = -1

# 游戏开始

while True:

# 切换当前玩家,如果当前玩家是 None 或者白棋 white\_player，则返回黑棋 black\_player;

# 否则返回 white\_player。

self.current\_player = self.switch\_player(self.black\_player, self.white\_player)

start\_time = datetime.datetime.now()

# 当前玩家对棋盘进行思考后，得到落子位置

# 判断当前下棋方

color = "X" if self.current\_player == self.black\_player else "O"

# 获取当前下棋方合法落子位置

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

# print("%s合法落子坐标列表："%color,legal\_actions)

if len(legal\_actions) == 0:

# 判断游戏是否结束

if self.game\_over():

# 游戏结束，双方都没有合法位置

winner, diff = self.board.get\_winner() # 得到赢家 0,1,2

break

else:

# 另一方有合法位置,切换下棋方

continue

action = self.current\_player.get\_move(self.board)

if action is None:

continue

else:

self.board.\_move(action, color)

if self.game\_over():

winner, diff = self.board.get\_winner() # 得到赢家 0,1,2

break

return winner, diff

def game\_over(self):

"""

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，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。

b\_list = list(self.board.get\_legal\_actions('X'))

w\_list = list(self.board.get\_legal\_actions('O'))

is\_over = len(b\_list) == 0 and len(w\_list) == 0 # 返回值 True/False

return is\_over

**根据Roxanne策略写的选落地位置：（优先表是上文中展示的优先级表）**

class RoxannePlayer(object):

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

"""

Roxanne策略初始化

:param roxanne\_table: 从上到下依次按落子优先级排序

:param color: 执棋方

"""

self.roxanne\_table = [

['A1', 'H1', 'A8', 'H8'],

['C3', 'F3', 'C6', 'F6'],

['C4', 'F4', 'C5', 'F5', 'D3', 'E3', 'D6', 'E6'],

['A3', 'H3', 'A6', 'H6', 'C1', 'F1', 'C8', 'F8'],

['A4', 'H4', 'A5', 'H5', 'D1', 'E1', 'D8', 'E8'],

['B3', 'G3', 'B6', 'G6', 'C2', 'F2', 'C7', 'F7'],

['B4', 'G4', 'B5', 'G5', 'D2', 'E2', 'D7', 'E7'],

['B2', 'G2', 'B7', 'G7'],

['A2', 'H2', 'A7', 'H7', 'B1', 'G1', 'B8', 'G8']

]

self.color = color

def roxanne\_select(self, board):

"""

采用Roxanne 策略选择落子策略

:return: 落子策略

"""

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

if len(action\_list) == 0:

return None

else:

for move\_list in self.roxanne\_table:

random.shuffle(move\_list)

for move in move\_list:

if move in action\_list:

return move

def get\_move(self, board):

"""

采用Roxanne 策略进行搜索

:return: 落子

"""

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))

action = self.roxanne\_select(board)

return action

**然后写蒙特卡洛树的节点类：**

class TreeNode():

"""

蒙特卡洛树节点

"""

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

self.parent = parent

self.w = 0

self.n = 0

self.color = color

self.child = dict()

**AI玩家类：**

class AIPlayer:

"""

AI 玩家

"""

def \_\_init\_\_(self, color, time\_limit = 0.5, c\_param = 1.4):

"""

玩家初始化

:param color: 下棋方，'X' - 黑棋，'O' - 白棋

"""

self.c\_param = c\_param

self.time\_limit = time\_limit

self.tick = 0

self.sim\_black = RoxannePlayer('X')

self.sim\_white = RoxannePlayer('O')

self.color = color

def mcts(self, board):

"""

蒙特卡洛树搜索，在时间限制范围内，拓展节点搜索结果

:return: 选择最佳拓展

"""

root = TreeNode(None, self.color)

# 设定一个时间停止计算，限定规模

while time() - self.tick < self.time\_limit:

sim\_board = deepcopy(board)

choice = self.select(root, sim\_board)

self.expand(choice, sim\_board)

winner, diff = self.simulate(choice, sim\_board)

back\_score = [1, 0, 0.5][winner]

if choice.color == 'X':

back\_score = 1 - back\_score

self.back\_prop(choice, back\_score)

best\_n = -1

best\_move = None

for k in root.child.keys():

if root.child[k].n > best\_n:

best\_n = root.child[k].n

best\_move = k

return best\_move

def select(self, node, board):

"""

蒙特卡洛树搜索，节点选择

:return: 搜索树向下递归选择子节点

"""

if len(node.child) == 0:

return node

else:

best\_score = -1

best\_move = None

for k in node.child.keys():

if node.child[k].n == 0:

best\_move = k

break

else:

N = node.n

n = node.child[k].n

w = node.child[k].w

# 随着访问次数的增加，加号后面的值越来越小，因此我们的选择会更加倾向于选择那些还没怎么被统计过的节点

# 避免了蒙特卡洛树搜索会碰到的陷阱——一开始走了歪路。

score = w / n + self.c\_param \* sqrt(log(N) / n)

if score > best\_score:

best\_score = score

best\_move = k

board.\_move(best\_move, node.color)

return self.select(node.child[best\_move], board)

def expand(self, node, board):

"""

蒙特卡洛树搜索，节点扩展

"""

op\_color = 'O' if node.color == 'X' else 'X'

for move in board.get\_legal\_actions(node.color):

node.child[move] = TreeNode(node, op\_color)

def simulate(self, node, board):

"""

蒙特卡洛树搜索，采用Roxanne策略代替随机策略搜索，模拟扩展搜索树

"""

if node.color == 'O':

current\_player = self.sim\_black

else:

current\_player = self.sim\_white

sim\_game = SilentGame(self.sim\_black, self.sim\_white, board, current\_player)

return sim\_game.run()

def back\_prop(self, node, score):

"""

蒙特卡洛树搜索，反向传播，回溯更新模拟路径中的节点奖励

"""

node.n += 1

node.w += score

if node.parent is not None:

self.back\_prop(node.parent, 1 - score)

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.tick = 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--------------------------------------

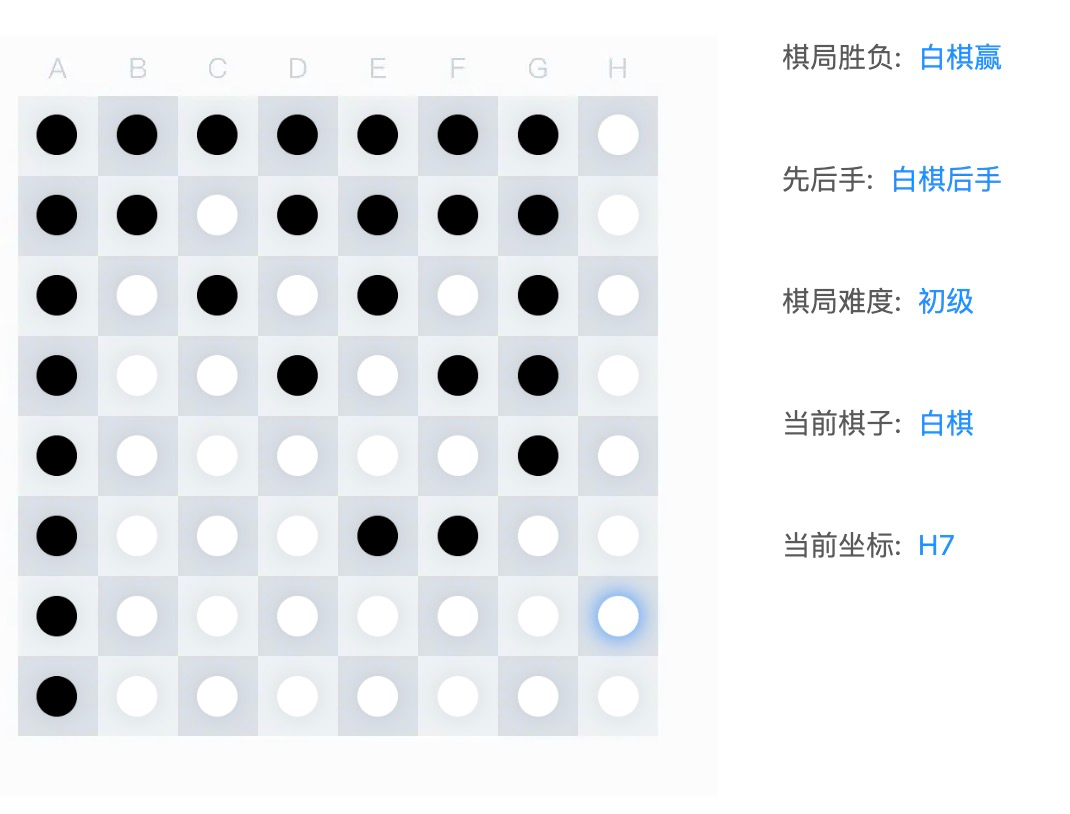
action = self.mcts(deepcopy(board))

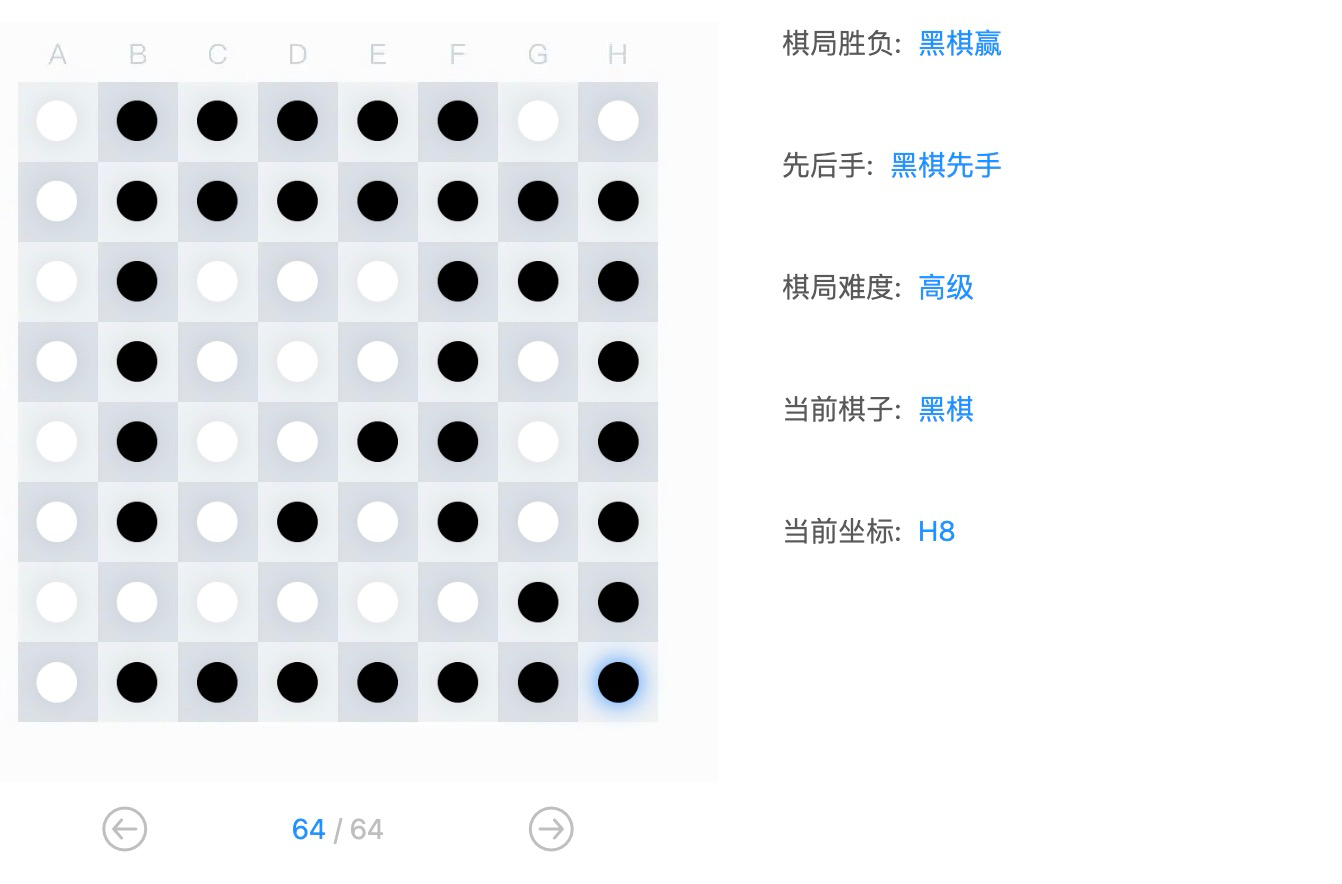
# ------------------------------------------------------------------------

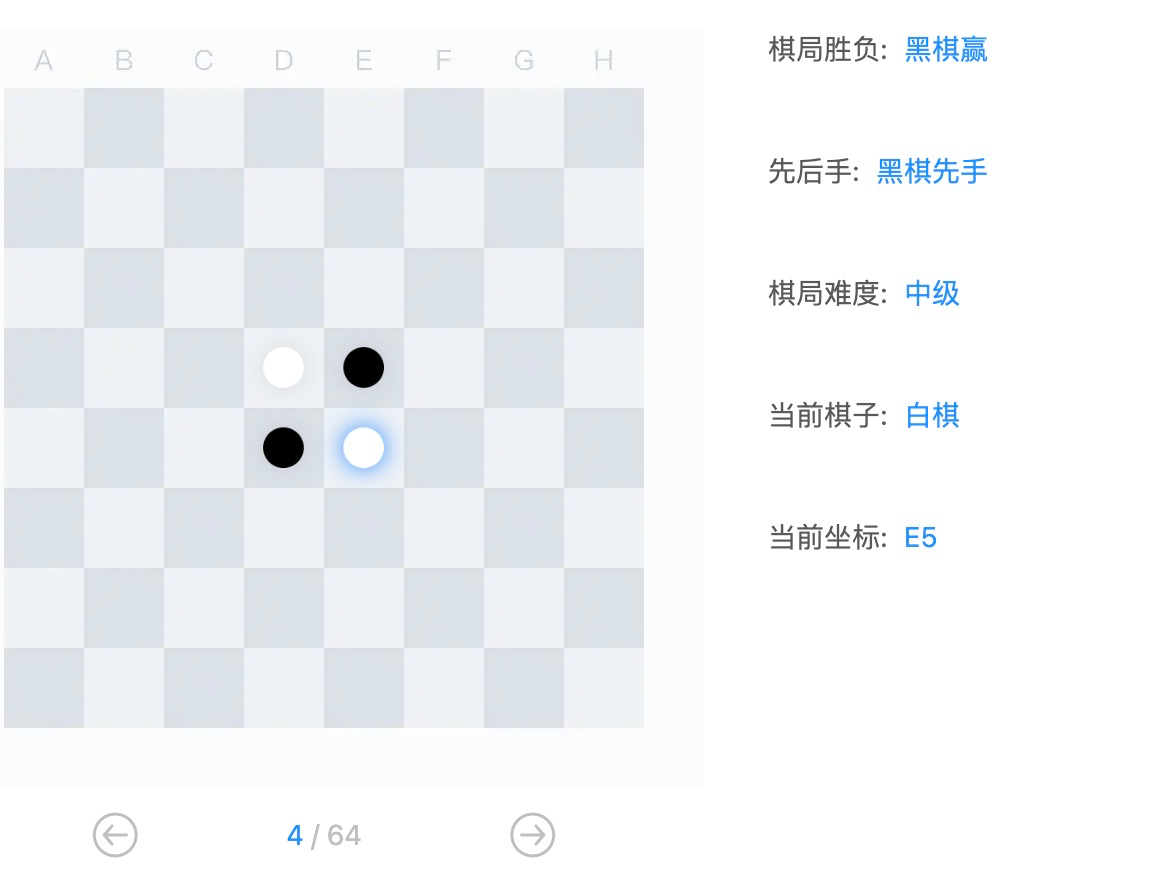
return action

1. **实验结果**

下面是在Mo环境下运行结果的截图：（分别选取了多个难度和先后手进行测试）发现与随机落子策略相比来说，获胜概率大幅度增加，任何难度下，无论是先手还是后手都基本会赢







1. **总结**

**是否达到目标预期：**

我认为在这个实验中，我达到了自己的目标预期。我成功地实现了黑白棋游戏，编写了算法，与系统进行了互动，并且成功地实现了使用Roxanne策略进行AI优化完成基于蒙特卡洛树的游戏，得到了较好的表现。

**可能改进的方向：**

在这个实验中，我认为可以进一步改进的方向是提高系统的鲁棒性和通用性。尽管我已经实现了一个完整的黑白棋游戏，并且能够使用Roxanne策略进行游戏，但是由于黑白棋游戏本身的复杂性，可能会存在一些意料之外的问题。因此，如果能够进一步提高系统的鲁棒性和通用性，将有助于使系统更加完善。利用其他工具或者算法提升AI的智能性，可以考虑加入学习功能。

**实现过程中遇到的困难：**

在实现过程中，我遇到了一些困难。首先，在编写算法的过程中，我需要充分理解黑白棋游戏的规则，以及如何将规则应用到算法中。其次，在使用Roxanne策略进行游戏时，我需要理解该策略的原理，并且在程序中正确地应用它。最后，在调试程序时，我还需要反复检查代码，找出错误并修复它们。

**从哪些方面可以提升性能**：

为了提升系统的性能，我可以考虑从以下几个方面进行改进：

算法优化：对算法进行优化，提高程序的运行效率。

硬件优化：通过使用更快的CPU、GPU等硬件设备，提高程序的运行速度。

超参数调整：调整算法的超参数，使其更适合解决黑白棋游戏问题。

框架优化：使用更加先进的编程框架和工具，提高程序的可维护性和可扩展性。

**模型的超参数和框架搜索是否合理**：

在实验中，我认为我选择的超参数和框架是合理的。我根据自己的理解和经验进行了选择，并且对比了不同参数和框架的性能表现。但是，在进行进一步的优化和改进时，我还需要深入理解各个参数和框架的原理，并且进行适当的比较和评估，以确保所选参数和框架的合理性。