**黑白棋问题求解程序报告**

**学号：2312900 姓名：禹相祐**

1. **问题重述**

**黑白棋问题：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。**

**游戏规则：**

**1. 黑方先行，双方交替下棋。**

1. **一步合法的棋步包括：**

**.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；**

**.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。**

1. **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。**
2. **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。**
3. **棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。**
4. **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。**

**实验要求：**

**使用 『蒙特卡洛树搜索算法』 实现 miniAlphaGo for Reversi**

**使用 Python 语言。**

**算法部分需要自己实现，不要使用现成的包、工具或者接口。**

**自己对问题的理解：**

**题目已经给出了除AI Player部分外的所有代码 所以只需要实现一个AI Player即可（利用蒙特卡洛树来实现）。**

1. **设计思想**

**2.1 蒙特卡洛树介绍**

蒙特卡洛树搜索（简称 MCTS）是 Rémi Coulom 在2006年在它的围棋人机对战引擎 「Crazy Stone」中首次发明并使用的，并且取得了很好的效果。在黑白棋中主要有以下四个重要的步骤，分别是：选择、扩展、模拟、反向传播。

**\*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。

**\*扩展**

若选择的节点不是终止节点，则随机扩展它的一个为扩展过的后继节点。

**\*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。

**\*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。

**2.2 蒙特卡洛树的代码实现**

**2.2.1 MCT的节点Node实现**

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | *# Node可以拿来存储棋盘的情况并在下完棋后更新*  **class** Node:  *# UCB计算时的系数，越大越偏向于探索新节点*      C **=** 1.414  *# Node初始化*  **def** \_\_init\_\_(self, board, color, root\_color, parent**=**None, pre\_action**=**None):          self.board **=** board  *# 当前节点的棋盘的状态*          self.color **=** color.upper()  *# 当前轮到哪方下棋，用upper()将输入统一转换为大写，以防出错*          self.root\_color **=** root\_color  *# 根节点颜色，用于判断胜负归属*          self.parent **=** parent          self.children **=** []  *# 子节点list*          self.best\_child **=** None  *# UCB 值最高的子节点(计算UCB后选择)*          self.get\_best\_child()  *# 初始时候尝试获取最优子节点*          self.preAction **=** pre\_action  *# 到达此节点所执行的落子*          self.actions **=** **list**(self.board.get\_legal\_actions(color**=**color))  *# 当前颜色下合法走法*          self.isOver **=** self.game\_over()  *# 当前节点是否为终局*          self.reward **=** {'X': 0, 'O': 0}  *# 节点的累计奖励值*          self.visit\_count **=** 0  *# 节点被访问的次数*          self.value **=** {'X': 1e5, 'O': 1e5}  *# 节点的评估值（用于 UCB）*          self.isLeaf **=** True  *# 是否为叶子节点*          self.best\_reward\_child **=** None  *# 奖励最高的子节点（用于最终落子）*          self.get\_best\_reward\_child()  *# 初始化时更新一下*    *# 判断当前node是否能end game*  **def** game\_over(self):          black\_list **=** **list**(self.board.get\_legal\_actions('X'))          white\_list **=** **list**(self.board.get\_legal\_actions('O'))  **if** **len**(black\_list) **==** 0 **and** **len**(white\_list) **==** 0:  **return** True  **else**:  **return** False    *# 更新节点UCB值*  **def** get\_value(self):  *# 初始直接跳过*  **if** self.visit\_count **==** 0:  **return**  *# 更新UCB*  **else**:  **for** color **in** ['X', 'O']:  *# 套用公式即可*                  self.value[color] **=** self.reward[color] **/** self.visit\_count **+** Node.C **\*** math.sqrt(math.log(self.parent.visit\_count) **\*** 2 **/** self.visit\_count)    *# 为当前node添加子node*  **def** add\_child(self, child):          self.children.append(child)          self.get\_best\_child()          self.get\_best\_reward\_child()          self.isLeaf **=** False    *# 获取当前子节点中UCB max的*  **def** get\_best\_child(self):  **if** **len**(self.children) **==** 0:              self.best\_child **=** None  **else**:  *# lambda + max 实现获取具有最大UCB的node*              self.best\_child **=** **max**(self.children, key**=lambda** child: child.value[self.color])  **return** self.best\_child    *# 获取子节点中平均奖励最高的（最终落子用）*  **def** get\_best\_reward\_child(self):  **if** **len**(self.children) **==** 0:              best\_reward\_child **=** None  **else**:  *# 依然 lambda + max 不过此处还有个针对visit\_count的判断：不能让没被探索过的子节点被选中*              best\_reward\_child **=** **max**(self.children,key**=lambda** child: child.reward[self.color] **/** child.visit\_count **if** child.visit\_count > 0 **else** **-**99999)          self.best\_reward\_child **=** best\_reward\_child  **return** best\_reward\_child |

**2.2.2 选择**

|  |  |  |
| --- | --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18 | *# 选择阶段（根据 ε-greedy 策略）*  *#  ε-greedy 策略就是让开始偏向于探索，然后随着进行逐渐趋于利用已经有的节点*  **def** select(self):          current\_node **=** self.root  *# 非叶子节点*  **while**  current\_node.isLeaf **==** False:  *# random.random()会返回一个0-1内的随机浮点数，这样也就实现了随机选择是否探索，且概率为30%选新节点，70%选已有的最好的节点*  **if** random.random() > self.init\_epsilon:  *# 70%概率选择最好的节点*                  current\_node **=** current\_node.get\_best\_child()  *# 30%抽奖*  **else**:  *# 随机从可选的内选一个*                  current\_node **=** random.choice(current\_node.children)  *# 让探索率epsilon变成0.995epsilon，逐渐偏向保守*              self.init\_epsilon **=** self.gamma **\*** self.init\_epsilon  *# 返回选择的节点*  **return** current\_node |  |

**2.2.3 扩展**

|  |  |  |
| --- | --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | *# 扩展阶段：将所有可能的走法都变成子节点*  **def** expand(self, node: Node):  *# Case1: 没有可以下的子*  **if** **len**(node.actions) **==** 0:  *# 依然copy board*              temp\_board **=** deepcopy(node.board)  *# Switch 旗子颜色*  **if** node.color **==** 'O':                  curr\_color **=** 'X'  **else**:                  curr\_color **=** 'O'  *# 创建一个新的子节点并跳过该回合:pre\_action=none*              child **=** Node(board**=**temp\_board, color**=**curr\_color, parent**=**node, pre\_action**=**"none", root\_color**=**self.color)  *# 将节点设置为该node的子节点*              node.add\_child(child)  *# 返回最佳child node，方便进行下一步*  **return** node.best\_child  *# Case2: 能下*  **else**:  *# 遍历所有可能*  **for** action **in** node.actions:                  temp\_board **=** deepcopy(node.board)  *# 在新棋盘上走一步*                  temp\_board.\_move(action**=**action, color**=**node.color)  *# Switch 旗子颜色*  **if** node.color **==** 'O':                      next\_color **=** 'X'  **else**:                      next\_color **=** 'O'                  child **=** Node(board**=**temp\_board, color**=**next\_color, parent**=**node, pre\_action**=**action, root\_color**=**self.color)                  node.add\_child(child)  **return** node.best\_child |  |

**2.2.4 模拟**

|  |  |  |
| --- | --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19 | *# 模拟阶段：从当前节点随机走到叶子节点*  **def** simulation(self, node: Node):  *# 拷贝棋盘，不更改棋盘原先模样*         temp\_board **=** deepcopy(node.board)         curr\_color **=** node.color  **while**  self.game\_over(temp\_board) **==** False:             possible\_actions **=** **list**(temp\_board.get\_legal\_actions(color**=**curr\_color))  *# 如果能继续就接着*  **if** possible\_actions !**=**[]:  *# 随机选择落子方法*                 temp\_board.\_move(random.choice(possible\_actions), curr\_color)  *# 换方继续*  **if** curr\_color **==** 'O':                 curr\_color **=** 'X'  **else**:                 curr\_color**=**'O'    *# 返回最终赢家*  **return** temp\_board.get\_winner() |  |

**2.2.5 反向传播**

|  |  |  |
| --- | --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | *# 反向传播*  **def** back\_propagation(self, node: Node, winner, difference):  *# 从下至上直到root停止*  **while** node **is** **not** None:              node.visit\_count **+=** 1  *# 平局,不用传统的+1是为了鼓励AI想办法赢的更大，而不是拘泥于一把是否胜利*  **if** winner **==** 0:                  node.reward['O'] **+=** difference                  node.reward['X'] **+=** difference  *# 黑胜*  **elif** winner **==** 1:                  node.reward['X'] **+=** difference                  node.reward['O'] **-=** 0.5 **\*** difference    *# 白胜*  **elif** winner **==** 2:                  node.reward['O'] **+=** difference                  node.reward['X'] **-=** 0.5 **\*** difference    **if** node **is** **not** self.root:  *# 先手动+1方便子节点更新UCB值*                  node.parent.visit\_count **+=** 1  *# 更新UCB值*  **for** child **in** node.parent.children:                      child.get\_value()  *# 撤回先前的+1*                  node.parent.visit\_count **-=** 1  *# node向上走一步*              node **=** node.parent |  |

1. **全部代码（Node & MCTS）**

|  |  |
| --- | --- |
| 001  002  003  004  005  006  007  008  009  010  011  012  013  014  015  016  017  018  019  020  021  022  023  024  025  026  027  028  029  030  031  032  033  034  035  036  037  038  039  040  041  042  043  044  045  046  047  048  049  050  051  052  053  054  055  056  057  058  059  060  061  062  063  064  065  066  067  068  069  070  071  072  073  074  075  076  077  078  079  080  081  082  083  084  085  086  087  088  089  090  091  092  093  094  095  096  097  098  099  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286 | **from** copy **import** deepcopy  **import** csv  **import** torch  **from** func\_timeout **import** func\_timeout, FunctionTimedOut  **import** math  **import** os.path  **import** random    # Node可以拿来存储棋盘的情况并在下完棋后更新  **class** Node:      # UCB计算时的系数，越大越偏向于探索新节点      exploration\_coefficient **=** 1.414      # Node初始化  **def** \_\_init\_\_(self, board, color, root\_color, parent**=**None, pre\_action**=**None):          # 当前节点的棋盘的状态          self.board **=** board          # 当前轮到哪方下棋，用upper()将输入统一转换为大写，以防出错          self.color **=** color.upper()          # 根节点颜色，用于判断胜负归属          self.root\_color **=** root\_color          self.parent **=** parent          # 子节点list          self.children **=** []          # UCB 值最高的子节点(计算UCB后选择)          self.best\_child **=** None          # 初始时候尝试获取最优子节点          self.get\_best\_child()          # 到达此节点所执行的落子          self.preAction **=** pre\_action          # 当前颜色下合法走法          self.actions **=** list(self.board.get\_legal\_actions(color**=**color))          self.isOver **=** self.game\_over()          self.reward **=** {'X': 0, 'O': 0}          self.visit\_count **=** 0          self.value **=** {'X': 1e5, 'O': 1e5}          self.isLeaf **=** True          self.best\_reward\_child **=** None          # 初始化时更新一下          self.get\_best\_reward\_child()        # 判断当前node是否能end game  **def** game\_over(self):          black\_list **=** list(self.board.get\_legal\_actions('X'))          white\_list **=** list(self.board.get\_legal\_actions('O'))          # 若黑白子可走的都为空 那就代表游戏结束          game\_is\_over **=** (len(black\_list) **==** 0 **and** len(white\_list) **==** 0)  **return** game\_is\_over        # 更新节点UCB值  **def** get\_UCB\_value(self):          # 初始直接跳过  **if** self.visit\_count **==** 0:  **return**  **for** color **in** ['X', 'O']:              self.value[color] **=** self.reward[color] **/** self.visit\_count **+** \                                  Node.exploration\_coefficient **\*** math.sqrt(                  math.log(self.parent.visit\_count) **\*** 2 **/** self.visit\_count)        # 为当前node添加子node  **def** add\_child(self, child):          self.children.append(child)          self.get\_best\_child()          self.get\_best\_reward\_child()          self.isLeaf **=** False        # 获取当前子节点中UCB max的,即最佳child  **def** get\_best\_child(self):  **if** len(self.children) **==** 0:              self.best\_child **=** None  **else**:              # lambda + max 实现获取具有最大UCB的node              sorted\_children **=** sorted(self.children, key**=lambda** child: child.value[self.color], reverse**=**True)              self.best\_child **=** sorted\_children[0]  **return** self.best\_child        # 获取子节点中平均奖励最高的（最终落子用）  **def** get\_best\_reward\_child(self):  **if** len(self.children) **==** 0:              best\_reward\_child **=** None  **else**:              # 依然 lambda + max 不过此处还有个针对visit\_count的判断：不能让没被探索过的子节点被选中              sorted\_children **=** sorted(self.children, key**=lambda** child: child.reward[                                                                            self.color] **/** child.visit\_count **if** child.visit\_count > 0 **else** **-**1e5,                                       reverse**=**True)              best\_reward\_child **=** sorted\_children[0]          self.best\_reward\_child **=** best\_reward\_child  **return** self.best\_reward\_child      # 蒙特卡洛树搜索主体  **class** MonteCarlo\_Search:      # 构造函数两个参数：一个是当前棋盘的状态board 一个是此时下棋代表的颜色  **def** \_\_init\_\_(self, board, color):          self.root **=** Node(board**=**deepcopy(board), color**=**color, root\_color**=**color)          self.color **=** color          self.experience **=** {"state": [], "reward": [], "color": []}          self.max\_experience **=** 1e15          # 三种结果：黑win 白win 平局          self.trans **=** {"X": 1, "O": **-**1, ".": 0}          self.learning\_rate **=** 0.3          # 初始探索率 即30%的概率选择探索新节点          self.epsilon **=** 0.2          # ε衰减因子 后续改变探索率:实现方式即epsilon逐渐乘gamma 越来越小          self.gamma **=** 0.995        # 从整棵树中收集经验（用于强化学习）  **def** get\_experience(self):          queue **=** []  **for** child **in** self.root.children:              queue.append(child)  **while** len(queue) > 0:  **if** len(self.experience) **==** self.max\_experience:  **break**  **if** **not** queue[0].isLeaf:                  self.add\_single\_node\_experiences(queue[0])  **for** child **in** queue[0].children:                      queue.append(child)              queue.pop(0)        # 获取棋盘状态,直接调node的board即可  **def** get\_board\_state(self, node):          new\_board\_state **=** node.board.board  **return** new\_board\_state        # 添加单个节点的经验  **def** add\_single\_node\_experiences(self, node: Node):    **if** len(self.experience["reward"]) **==** self.max\_experience:  **return**            experience **=** self.get\_board\_state(node)          self.experience["state"].append(experience)          reward **=** node.reward["X" **if** node.color **==** "O" **else** "O"] **/** node.visit\_count          self.experience["reward"].append(reward)          self.experience["color"].append(node.color)        # MCTS的搜索步骤  **def** search(self):  **if** len(self.root.actions) **==** 1:  **return** self.root.actions[0]  **return** self.search\_by\_montecarlo\_tree()   # 执行Search，time\_limit设置为3s （因为60s样例运行太久了，3s依然能win）  **def** search\_by\_montecarlo\_tree(self):          # 时间设置为3s，利用func\_timeout（）实现  **try**:              func\_timeout(timeout**=**3, func**=**self.build\_montecarlo\_tree)          # 超出时间则不执行  **except** FunctionTimedOut:  **pass**          # 空值保护逻辑          best\_child **=** self.root.get\_best\_reward\_child()          # 返回最终落子的地方  **if** best\_child **is** None:              # 降级策略：直接选择当前合法动作  **return** random.choice(self.root.actions) **if** self.root.actions **else** None  **return** best\_child.preAction        # 建MCT（不断进行选择 & 扩展 & 模拟 & 反向传播）  **def** build\_montecarlo\_tree(self):  **while** True:              current\_node **=** self.select()  **if** current\_node.isOver **==** True:                  # 返回winner和相差比分                  winner, difference **=** current\_node.board.get\_winner()  **else**:                  # 被访问过才执行拓展  **if** current\_node.visit\_count > 0:                      # 拓展                      current\_node **=** self.expand(current\_node)                  # 无论有没有被访问过都要模拟                  winner, difference **=** self.simulation(current\_node)              # 反向传播同模拟              self.back\_propagation(node**=**current\_node, winner**=**winner, difference**=**difference)        # 选择阶段（根据 ε-greedy 策略）      #  ε-greedy 策略就是让开始偏向于探索，然后随着进行逐渐趋于利用已经有的节点  **def** select(self):          current\_node **=** self.root          # 非叶子节点  **while** **not** current\_node.isLeaf:              # random.random()会返回一个0-1内的随机浮点数，这样也就实现了随机选择是否探索，且概率为30%选新节点，70%选已有的最好的节点  **if** random.random() > self.epsilon:                  # 70%概率选择最好的节点                  current\_node **=** current\_node.get\_best\_child()  **else**:                  # 30%抽奖:随机从可选的内选一个                  current\_node **=** random.choice(current\_node.children)              # 让探索率epsilon变成0.999epsilon，逐渐偏向保守              self.epsilon **\*=** self.gamma    **return** current\_node        # 模拟阶段：从当前节点随机走到叶子节点  **def** simulation(self, node: Node):          # 拷贝棋盘，不更改棋盘原先模样          board **=** deepcopy(node.board)          color **=** node.color  **while** **not** self.game\_over(board**=**board):              actions **=** list(board.get\_legal\_actions(color**=**color))              # 如果能继续就接着  **if** len(actions) !**=** 0:                  # 随机选择落子方法                  board.\_move(random.choice(actions), color)              # 换方继续              color **=** 'X' **if** color **==** 'O' **else** 'O'          winner, difference **=** board.get\_winner()    **return** winner, difference        # 扩展阶段：将所有可能的走法都变成子节点  **def** expand(self, node: Node):          # Case1: 没有可以下的子  **if** len(node.actions) **==** 0:              board **=** deepcopy(node.board)              color **=** 'X' **if** node.color **==** 'O' **else** 'O'              # 创建一个新的子节点并跳过该回合:pre\_action=none              child **=** Node(board**=**board, color**=**color, parent**=**node, pre\_action**=**"none", root\_color**=**self.color)              # 将节点设置为该node的子节点              node.add\_child(child)  **return** node.best\_child          # Case2: 能下          # 遍历所有可能  **for** action **in** node.actions:              board **=** deepcopy(node.board)              # 在新棋盘上走一步              board.\_move(action**=**action, color**=**node.color)              color **=** 'X' **if** node.color **==** 'O' **else** 'O'              child **=** Node(board**=**board, color**=**color, parent**=**node, pre\_action**=**action, root\_color**=**self.color)              node.add\_child(child**=**child)  **return** node.best\_child        # 反向传播  **def** back\_propagation(self, node: Node, winner, difference):          # 奖励逻辑不再采用+1 or +0  **while** node **is** **not** None:              node.visit\_count **+=** 1            # 根据当前节点的颜色考虑  **if** node.root\_color **==** 'X':          # 黑棋视角：  **if** winner **==** 1:  # 黑胜                  reward **=** 1 **+** math.tanh(difference **/** 20)  # 基础奖励+比分差奖励  **elif** winner **==** 2:  # 白胜                  reward **=** **-**1 **-** math.tanh(difference **/** 20)  # 惩罚+比分差惩罚  **else**:  # 平局                  reward **=** 0.5 **\*** math.tanh(difference **/** 20)  # 根据比分差调整  **else**:              # 白棋视角：  **if** winner **==** 2:  # 白胜                  reward **=** 1 **+** math.tanh(**-**difference **/** 20)  **elif** winner **==** 1:  # 黑胜                  reward **=** **-**1 **-** math.tanh(**-**difference **/** 20)  **else**:                  reward **=** 0.5 **\*** math.tanh(**-**difference **/** 20)            # 更新当前节点的奖励值          node.reward[node.root\_color] **+=** reward            # 更新父节点的子节点UCB值  **if** node **is** **not** self.root:              node.parent.visit\_count **+=** 1  **for** child **in** node.parent.children:                  child.get\_UCB\_value()              node.parent.visit\_count **-=** 1            node **=** node.parent        # 判断游戏是否结束  **def** game\_over(self, board):          black\_list **=** list(board.get\_legal\_actions('X'))          white\_list **=** list(board.get\_legal\_actions('O'))          game\_is\_over **=** len(black\_list) **==** 0 **and** len(white\_list) **==** 0  **return** game\_is\_over    **class** AIPlayer:  **def** \_\_init\_\_(self, color: str):          self.color **=** color.upper()          self.comments **=** "请稍后，{}正在思考".format("黑棋(X)" **if** self.color **==** 'X' **else** "白棋(O)")    **def** get\_move(self, board):          print(self.comments)          action **=** MonteCarlo\_Search(board, self.color).search()  **return** action |

1. **实验结果**

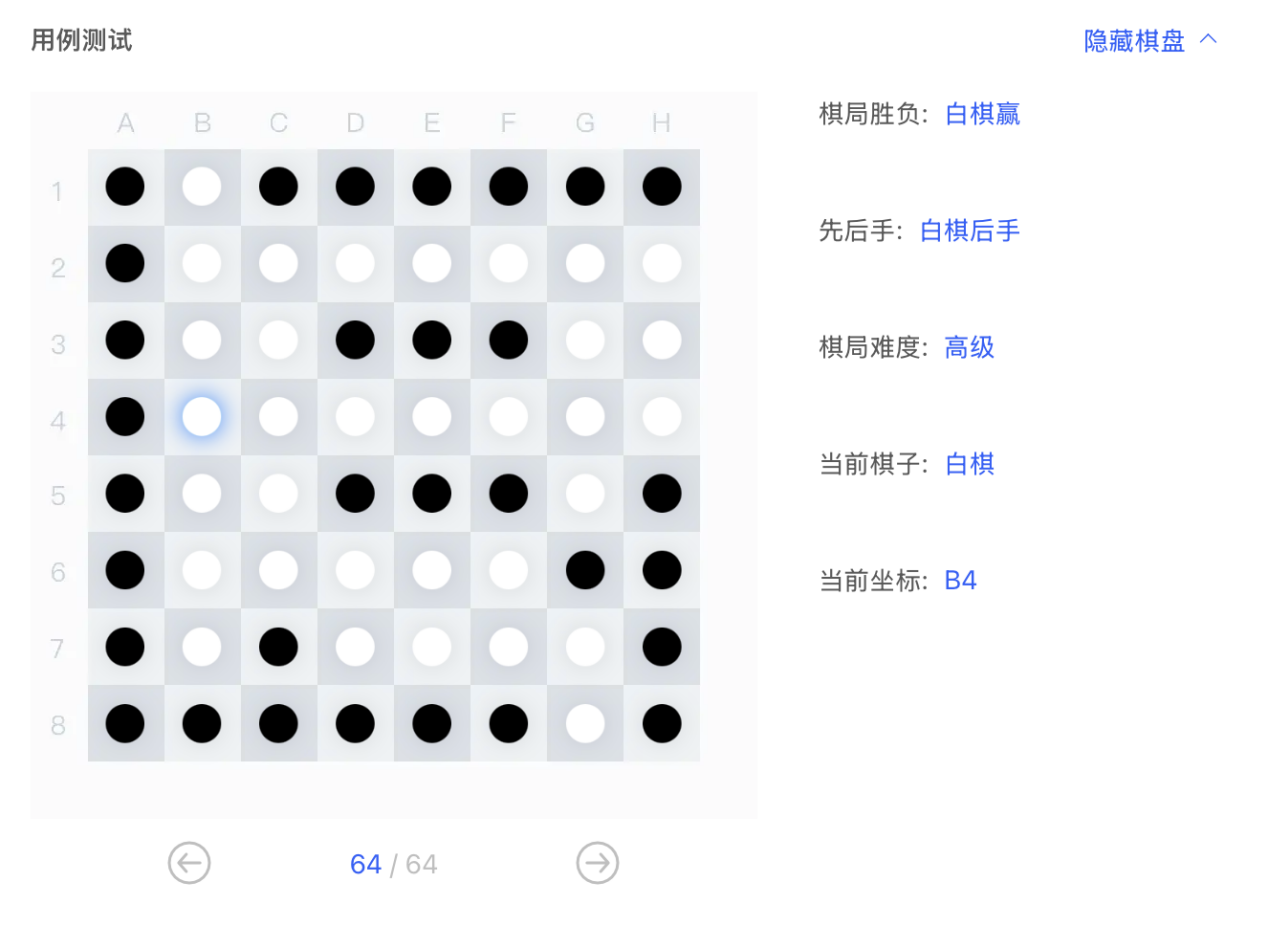
先选择 高级难度 & 黑棋先手：

此时赢了48个子，如图：



再是 高级难度 & 白棋后手：

此时赢了2个子，如图：



1. **总结**
2. 这次实验的关键就是实现蒙特卡洛树搜索的四个步骤：选择、拓展、模拟、反向传播
3. 不仅仅是四个步骤的实现比较关键，还有参数C的选择、运行时间的限制t、初始探索率e以及衰减率gamma大小的确定也一样与结果好坏直接相关。最后还好也是都实现了，不枉我超50次的测试。