《黑白棋》程序报告

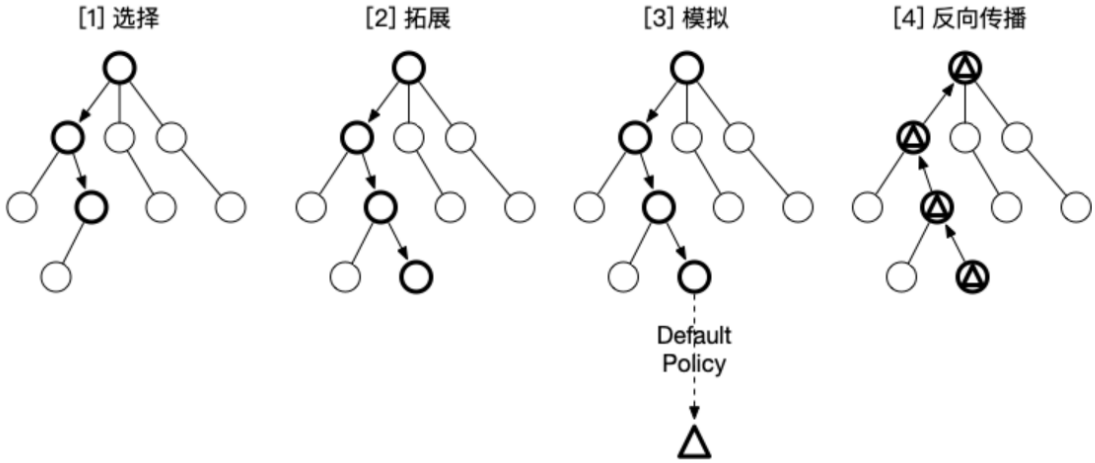
姓名： 聂志强 学号：2012307 班级：信息安全

一. 问题重述

黑白棋问题：使用『蒙特卡洛树搜索算法』实现 miniAlphaGo for Reversi。

二. 五大核心设计思想

1. **蒙特卡洛树框架**



a. 选择：

从搜索树的根节点开始，向下递归选择子节点，直至到达叶子节点或者到达具有还未被扩展过的子节点的节点L。选择过程中：

- 如果所有可行动作都已经被拓展过了，那么我们将使用UCB公式计算该节点所有子节点的UCB值，并找到值最大的一个子节点继续检查。反复向下迭代。

$$\textbf{UCB:} \quad l_t = \operatorname{argmax}_i \bar{x}_{i,T_{(i,t-1)}} + C\sqrt{\frac{2\ln t}{T_{(i,t-1)}}}$$

- 如果被检查的局面依然存在没有被拓展的子节点，那么认为这个节点就是本次迭代的的目标节点N，并找出N还未被拓展的动作A。执行步骤[2]
- 如果被检查到的节点是一个游戏已经结束的节点。那么从该节点直接执行步骤[4]

每一个被检查的节点的被访问次数在这个阶段都会自增。在反复的迭代之后， 我们将在搜索树的底端找到一个节点，来继续后面的步骤。

b. 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）。

c. 模拟：

我们基于目前的这个状态，根据某一种策略（例如random policy）进行模拟，直到游戏结束为止，以棋差作为评分。

d. 反向传播：

模拟结束之后，根据模拟的结果，我们要自底向上，反向更新所有节点的信息。如果在[1]的选择中直接发现了一个游戏结局的话，根据该结局来更新评分。每一次迭代都会拓展搜索树，随着迭代次数的增加，搜索树的规模也不断增加。当到了一定的迭代次数或者时间之后结束，选择根节点下最好的子节点作为本次决策的结果。

e. 代码框架

采用蒙特卡洛搜索树算法，定义了TreeNode树节点类和MCTS算法类。分别在两个类中定义相关方法

- Node搜索树节点类：
  判断节点是否完全扩展：
  `def full_expand(self)`
  判断当前是否为终结状态
  `def isfinish(self):`
  添加子节点
  `def add_child(self, child_state, action, color)`

  AIplayer算法类：
  选择：所有子节点都扩展完了,计算所有节点的UCB值
  `def get_bestchild(self,node,is_exploration)`
  拓展节点：
  `def expand(self,node)`
  选择节点策略
  `def SelectPolicy(self)`
  模拟过程：
  `def SimulatePolicy(self,node)`
  反向传播：
  `def BackPropogate(self,node,rewrd)`

2. **剪枝搜索**

当黑白棋黑子先手,在前面12步中,也就是抛开开局系统给出的四颗棋子外，最好不要把棋子放在中心两圈方框之外。这个部分的宗旨是先占满中心两圈方框,把对方逼出方框。因此设计剪枝搜索函数并且设置每次搜索深度为6，每搜索一次计算一次当前棋盘得分并根据得分计算出最佳走法

3. **超参数搜索**

第一组以5递增对C进行超参数搜索，根据第一组结果确定C区间40—65，第二组以1递增对C进行超参数搜索，为减小不确定性和增加鲁棒性，对每组C重复3次搜索并调用matplotlib库绘制曲线图

4. **棋盘加权**

   根据黑白棋下棋技巧，当不同合法落子位置计算出的UCB值相近时，应该优先选择最边上的合法落子位置，尤其是四个角的落子点，因此给棋盘赋权重，达到以上效果

5. **蒙特卡洛搜索次数**

   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

三. 重点代码解析

```
class treeNode(object):
    def __init__(self, color, board, parent=None): #初始化函数
        self.parent = parent #当前节点的父节点，以备反向传播使用
        self.children=[] #当前节点的子节点集合
        self.numVisits=0 #访问次数
        self.totalReward = 0 #获取的奖励
        self.color=color #当前的执棋方颜色
        self.board=board #当前board状态
```

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

```
def full_expand(self):
    action = list(self.state.get_legal_actions(self.color))
    if len(self.children) == len(action):
        return True
    return False
```

添加子节点

```
def add_child(self, child_state, action, color):
    child_node = Node(child_state, parent=self, action=action, color=color)
    self.children.append(child_node)
```

UCT search的核心框架

- 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55s时退出循环，根据UCB的值计算出下一步最佳落子位置
- 复盘数次蒙特卡洛树搜索落子过程，发现蒙特卡洛搜索算法在前期表现并不是很佳，根据黑白棋技巧：当黑白棋黑子先手,在前面12步中(抛开开局系统给出的四颗棋子外)，最好不要把棋子放在中心两圈方框之外。这个部分的宗旨是先占满中心两圈方框,把对方逼出方框。根据如上技巧，设计剪枝搜索算法，当棋盘总子数小于28时，调用剪枝搜索函数给被选的节点权重，优先选择剪枝搜索的结果，后面过程仅利用蒙特卡洛搜索算法

```
def uct(self, max_times, root):
    board_a=deepcopy(root.state)
    action_a, _= self.max_value(board_a, -65, 65, 0)
    for t in range(0, max_times):
        if time.time() - self.start >= 58:
            break
        leave_node = self.select_expand_node(root)
        reward = self.random_stimulate_chess(leave_node)
        if leave_node.action==action_a and
((leave_node.state.count('X')+leave_node.state.count('O'))<=28):
            reward+=1000
        self.backup(leave_node, reward)

     n = len(root.children)
     for i in range(0, n):
            if 'A1' == root.children[i]:
                return 'A1'
            if 'H8' == root.children[i]:
                return 'H8'
            if 'A8' == root.children[i]:
                return 'A8'
            if 'H1' == root.children[i]:
                return 'H1'
     best_child = self.ucb(root, self.SCALAR)
     return best_child.action
```

选择+拓展过程：

1. 如果当前节点所有合法落子结果已经扩展，则根据当前节点各个孩子UCB的值选择节点（选择过程）
2. 拓展子节点的策略：在这里对随机选择算法进行了改进，优先考虑目前期望值较大的节点，有0.5的概率在当前节点存在可扩展节点时选择不扩展。

```
def select_expand_node(self, node):
    if not self.game_overed(node.state):
        l = list(node.state.get_legal_actions(node.color))
    if len(l) == 0:
        return node.parent
    if len(node.children) < len(l):
        r = random.uniform(0, 1)
        if r < self.balance or len(node.children) == 0:
            new_action = l[len(node.children)]
            new_state = deepcopy(node.state)
            new_state._move(new_action, node.color)
            if node.color == 'X':
                new_color = 'O'
            else:
                new_color = 'X'
                node.add_child(new_state, new_action, new_color)
                return node.children[-1]
        else:
```

```
            return random.choice(node.children)
    else:
        new_node = self.ucb(node, self.SCALAR)
        return self.select_expand_node(new_node)
    return node
```

UCB函数计算

```
def ucb(self, node, scalar):
    if node.color == self.color:
        best_score = -1000
        best_children = []
        for child in node.children:
            exploit = child.reward / child.visits
            if child.visits == 0:
                best_children = [child]
                break
                explore = math.sqrt(2.0 * math.log(node.visits) /
float(child.visits))
                now_score = exploit + scalar * explore
                if now_score == best_score:
                    best_children.append(child)
                if now_score > best_score:
                    best_children = [child]
                    best_score = now_score
                return random.choice(best_children
     else:
         best_score = 1000
         best_children = []
         for child in node.children:
              exploit = child.reward / child.visits
              if child.visits == 0:
                  best_children = [child]
                  break
              explore = math.sqrt(2.0 * math.log(node.visits) / float(child.visits))
              now_score = exploit + scalar * explore
              if now_score == best_score:
                  best_children.append(child)
              if now_score < best_score:
                  best_children = [child]
                  best_score = now_score
        return random.choice(best_children)
```

模拟过程：

1. 经过奖励机制的超参数搜索，发现以 50+|棋差| 作为奖励机制效果最好，如下模拟过程代码，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reward的值作为奖励
2. 在上述奖励机制的前提下又做了优化，因为四个角是黑白棋中的关键，所以在合法落子中如果存在四个角，则reward+=10增加该落点的reward权重

```
def SimulatePolicy(self,node):
      board = deepcopy(node.state)
        color = node.color
        count = 0
        while not self.game_overed(board):
            action_list = list(node.state.get_legal_actions(color))
            if not len(action_list) == 0:
                action = random.choice(action_list)
                board._move(action, color)
                if color == 'X':
                    color = 'O'
                else:
                    color = 'X'
            else:
                if color == 'X':
                    color = 'O'
                else:
                    color = 'X'
                action_list = list(node.state.get_legal_actions(color))
                action = random.choice(action_list)
                board._move(action, color)
                if color == 'X':
                    color = 'O'
                else:
                    color = 'X'
            count = count + 1
            if count >= 61:
                break
        winner, difference = board.get_winner()
        if winner == 2:
            reward = 0
        elif winner == 1:
            reward = 50+difference
        else:
            reward = -(50+difference)

        if node.state[0][0] == self.color:
            reward += 10
        if node.state[0][7] == self.color:
            reward += 10
        if node.state[7][0] == self.color:
            reward += 10
        if node.state[7][7] == self.color:
            reward += 10

        if self.color == 'X':
            reward = - reward
        return reward
```

反向传播过程：从拓展节点向上递归至根节点，每个节点访问次数+1，reward值相应增加模拟结果的返回值

```
def backup(self, node, reward):
    while node is not None:
    node.changevisits(1)
    node.changereward(reward)
    node = node.parent
    return 0
```

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。

```
def game_overed(self, state):
    now_loc = list(state.get_legal_actions('X'))
    next_loc = list(state.get_legal_actions('O'))
    over = len(now_loc) == 0 and len(next_loc) == 0
    return over
```

为棋盘赋权：

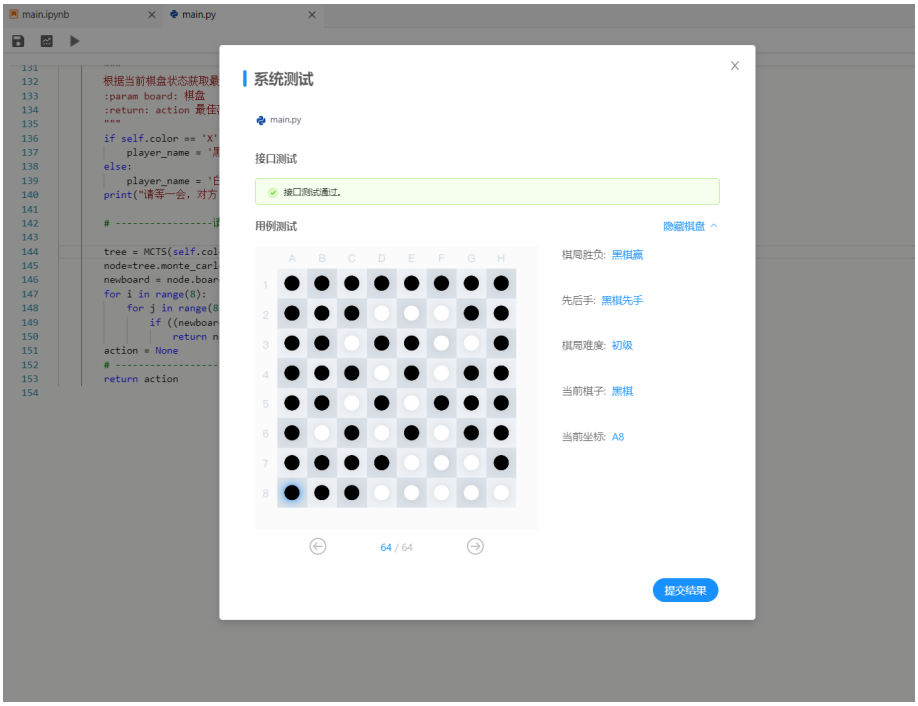
1. 使一开始黑棋先占满中心两圈方框,把对方逼出方框
2. 当多个合法落子点reeward值相近时优先选择靠边，尤其是四个角

```
self.weight = [[35, 2, 9, 9, 9, 9, 2, 35],
               [2, 2, 3, 3, 3, 3, 2, 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, 2, 3, 3, 3, 3, 2, 2],
               [35, 2, 9, 9, 9, 9, 2, 35]]
```

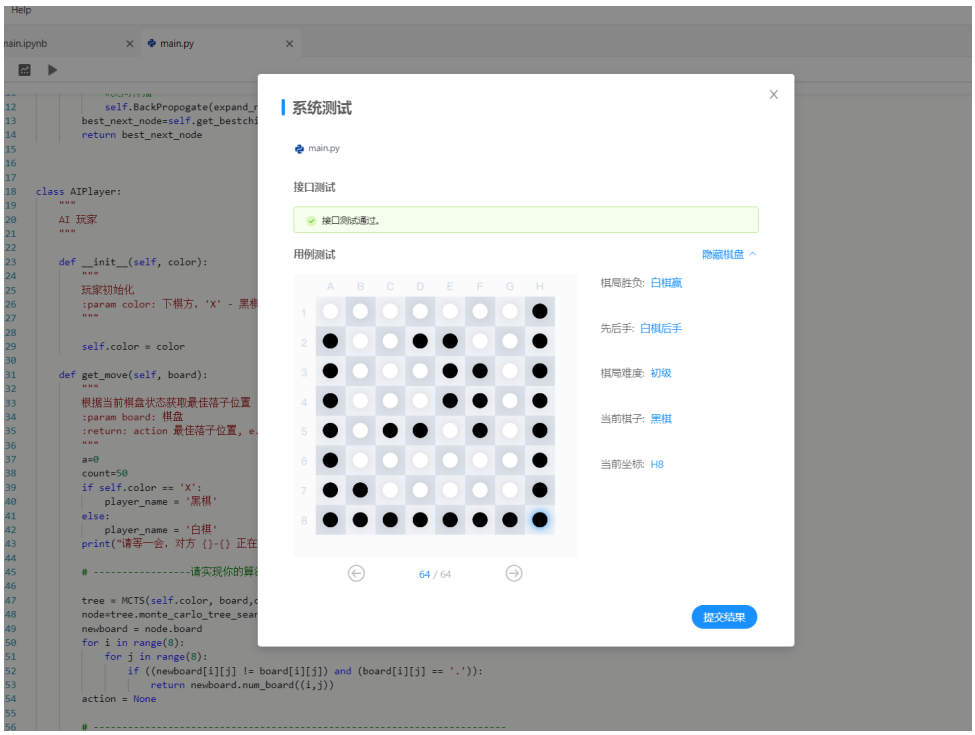
四. 实验结果

1. 初级

1）黑棋先手（黑棋赢）

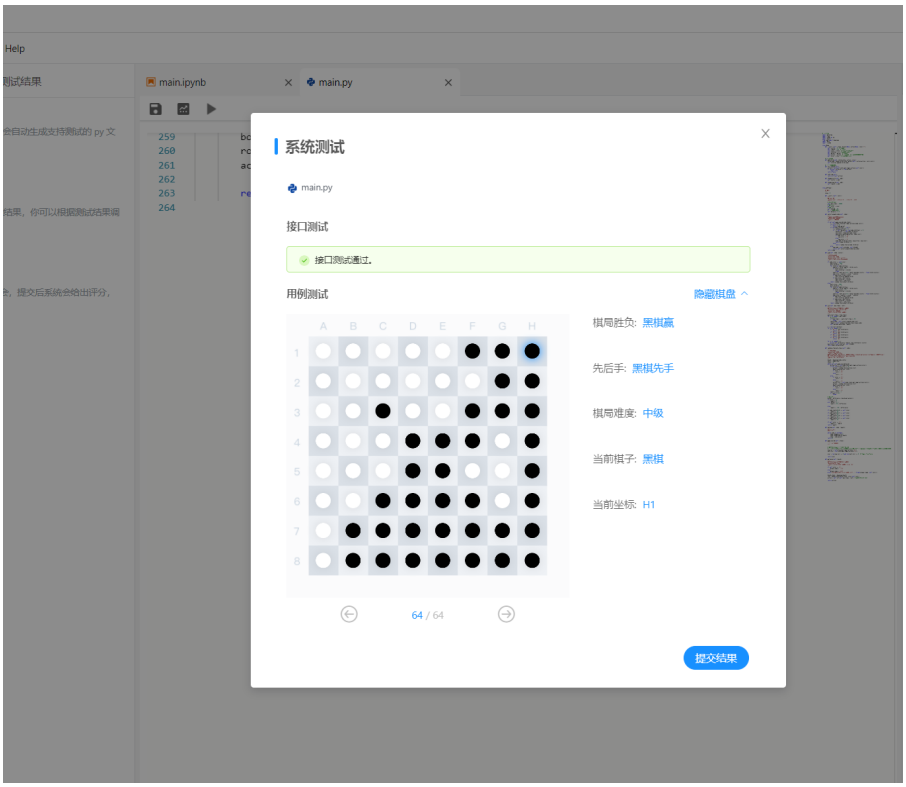


2）白棋后手（白棋赢）

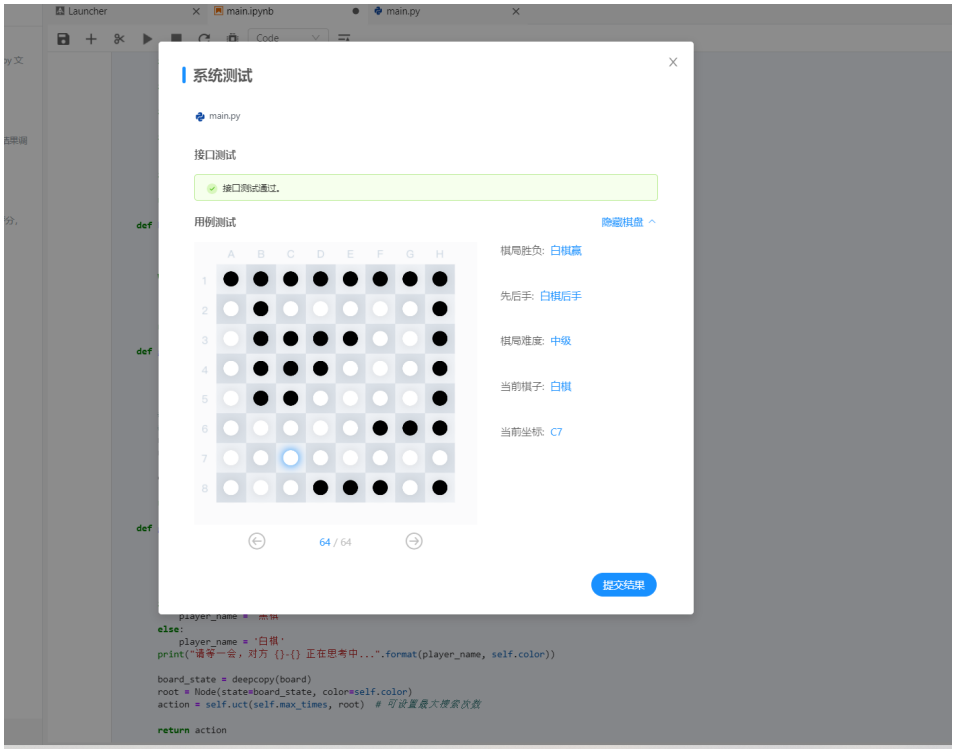


2. 中级

1） 黑棋先手（黑棋赢）

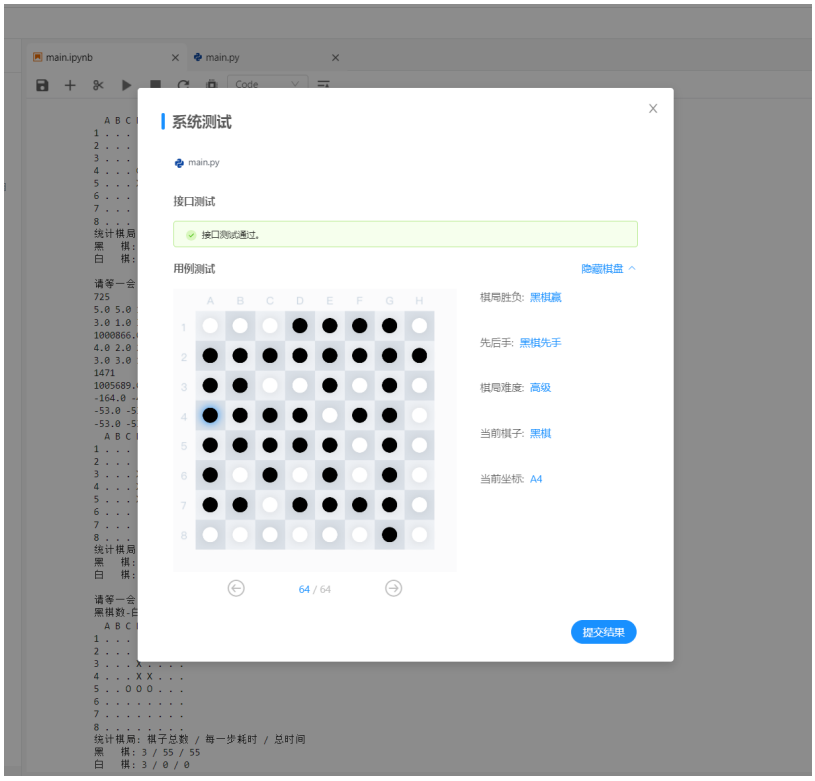


2） 白棋后手（白棋赢）

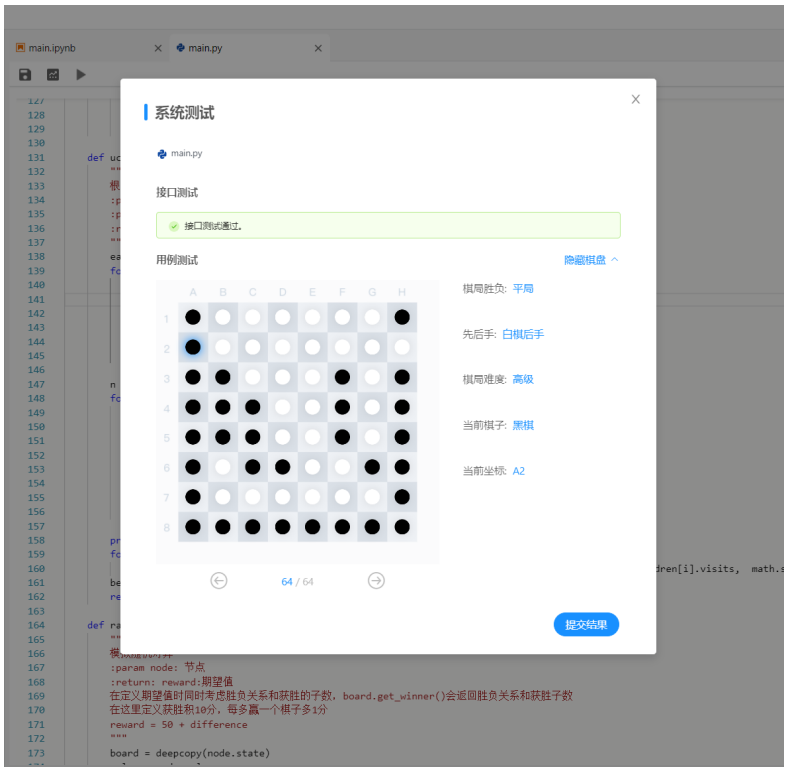


3. **高级**

1）黑棋先手（黑棋赢）



2）白棋后手（白棋赢）

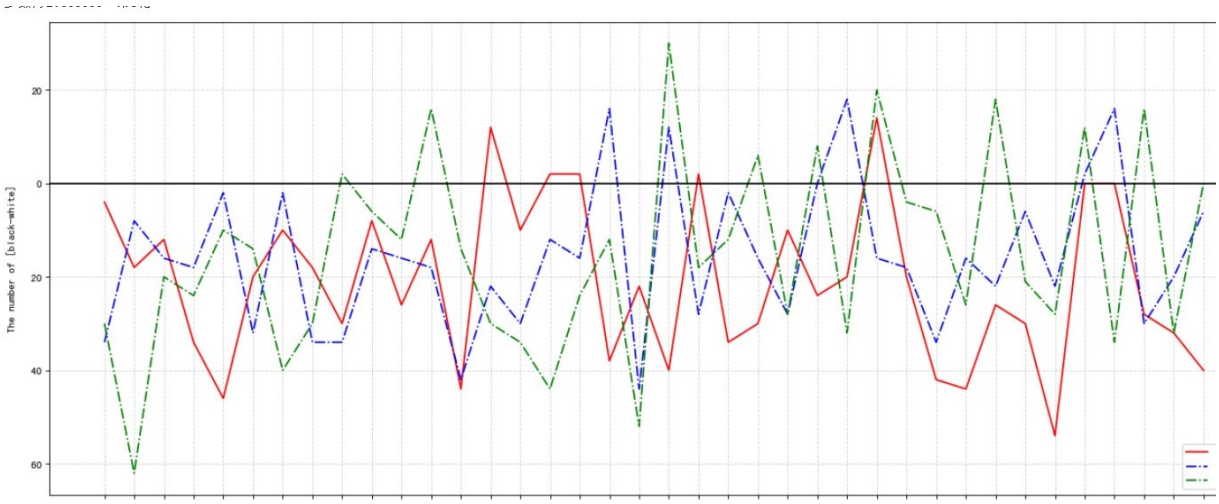


五. 总结

1. 共改进13版代码，提交mo平台测试100次

2. 初级/中级/高级 * 黑棋先手/白棋后手 共六次测试均获胜
3. 超参数搜索全过程：UCB起到探索和利用之间的平衡，所以C的参数设定很关键
    - 首先输出每次的reward大小及访问次数，观察后预先判断C的一个范围，使UCB等式加号两端的计算值量级匹配，否则会使选择过程仅根据搜索或利用单一维度在进行超参数搜索
    - 超参数搜索过程分为两组，第一组的目的是判断一个大致区间，因此使C以5递增，通过第一次的搜索结果，估计出最佳C的范围后，以1递增在所找范围内进行第二次超参数搜索，找到精确值

    如下为第二次超参数搜索时（Random[黑方]—AI[白方]）绘制的图像，每个参数黑白棋均重复对抗3次以增加稳定性（每种颜色代表一次），函数值在0轴以下代表我方（AI）赢，具体数值为赢子的个数



4. 要注意对列表为空进行判断，例如如下实验中某次错误，模拟阶段中，children列表为空但未进行判断，仍调用函数random.choice随机选取 children列表中的元素产生报错

```
/tmp/ipykernel_69/172449176.py in select_expand_node(self, node)
    107             else:
    108                 new_node = self.ucb(node, self.SCALAR)
--> 109                 return self.select_expand_node(new_node)
    110         return node
    111

/tmp/ipykernel_69/172449176.py in select_expand_node(self, node)
    107             else:
    108                 new_node = self.ucb(node, self.SCALAR)
--> 109                 return self.select_expand_node(new_node)
    110         return node
    111

/tmp/ipykernel_69/172449176.py in select_expand_node(self, node)
    107             else:
    108                 new_node = self.ucb(node, self.SCALAR)
--> 109                 return self.select_expand_node(new_node)
    110         return node
    111

/tmp/ipykernel_69/172449176.py in select_expand_node(self, node)
    106                     return random.choice(node.children)
    107             else:
--> 108                 new_node = self.ucb(node, self.SCALAR)
    109                 return self.select_expand_node(new_node)
    110         return node

/tmp/ipykernel_69/172449176.py in ucb(self, node, scalar)
    148                     best_children = [child]
    149                     best_score = now_score
--> 150             return random.choice(best_children)
    151
    152     def uct(self, max_times, root):

/usr/lib/python3.7/random.py in choice(self, seq)
    259             i = self._randbelow(len(seq))
    260         except ValueError:
--> 261             raise IndexError('Cannot choose from an empty sequence') from None
    262         return seq[i]
    263

IndexError: Cannot choose from an empty sequence
```

```
[30]: # from game import Game
```