程序报告

学号：2211290　　姓名：姚知言

一、问题重述

结合对蒙特卡洛树搜索的理解，为黑白棋游戏创造 AI 玩家（miniAlphaGo for Reversi）。通过编写蒙特卡洛树类和 AI 玩家类中成员和函数，完成蒙特卡洛树的建立以及搜索各步骤（选择、扩展、模拟、反向传播）。将自己的 AI 玩家与系统测试所用 AI 玩家进行模拟对弈，并战胜 AI 玩家。

二、设计思想

通过创建 MCT_node 类，编写__init__初始化函数，add 添加子结点函数，full_expand 判断是否扩展完全函数，为搜索过程建树做好准备。

创建 AIplayer 类，编写__init__初始化函数。

calu_ucb 函数计算 ucb 值：设置默认 c 值为 1，在最后返回所求最优解时设置 c 值为 0。由于本题所求蒙特卡洛树为双方交替，后续计算过程中选用记录己方 value 方法（见反向传播步骤，所以计算子结点 ucb 值的时候需要 1-value/visit_num 才能正确得到己方胜率，其余公式不变。

select 函数实现选择步骤：考虑己方无可选点的边界情况，若己方无可选点且无子结点被扩展，则扩展一个不行动的子结点，若该结点已经被扩展，则继续循环遍历该子结点的情况。对于一般情况，若查找到的结点没有被扩展完全，则扩展一个该结点的子结点，若已经扩展完全，则递归查看该结点 ucb 值最大的子结点。若递归到游戏结束（叶子结点）则将其返回，无需扩展。

expand 函数实现扩展步骤：同样分为扩展无可选点的结点和普通情况，普通情况即选择一个没有被扩展的点进行扩展。

simulate 函数实现模拟步骤：将上一步获得的结点以随机选择的方式进行模拟，直至游戏结束（叶子结点），记录胜负情况。

backpropagation 函数实现反向传播步骤：将胜负情况从下往上递归，胜利则 value+1，平局则 value+0.5，失败不增加 value。

choose_best_child 函数通过反复进行蒙特卡洛树搜索确定最优解：设置默认迭代次数 500，可以在此范围内保证 1 分钟内返回结果（实际用时约 8 秒），依次进行以上步骤，结束后设置 c=0 获得答案。

get_move 函数在可选点不足 2 个的时候快速返回，在可选点 2 个及以上的时候进行蒙特卡洛树搜索。

三、代码内容

```
class MCT_node:
    def __init__(self,state,parent=None,action=None,color=''):
        '''
        Args:
            state: 棋盘状态
```

```
                parent: 父节点
                action: 落子位置
                color: 落子颜色
            '''
            self.color=color
            self.parent=parent
            self.state=state
            self.value=0.00
            self.visit_num=0.00
            self.action=action
            self.children=[]
        def add(self,child_state,action,color):
            '''
            添加子节点，我们需要不断扩展整棵树，所以需要添加子节点
            ------------------------------
            Your implementation here
            '''
            child=MCT_node(child_state,self,action,color)
            self.children.append(child)
            return child

        def full_expand(self):
            '''
            判断是否已经扩展完全
            判断的意义在于，如果一个节点的子节点已经全部扩展，那么我们就不需要再次扩展了，直接模拟即可
            ------------------------------
            Your implementation here
            '''
            return len(list(self.state.get_legal_actions(self.color)))==len(self.children)
================================================================================
import math
import copy
import random
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
```

```
    def calu_ucb(self, node, c=1):
        """
        计算 UCB 值
        UCB = Q/N + C * sqrt(2 * ln(P) / N)
        """
        best_ucb=0.00
        best_child=None
        for child in node.children:

child_ucb=1-(child.value/child.visit_num)+c*math.sqrt(2*math.log(node.visit_num)/child.visit_num)
            if best_child==None or child_ucb>best_ucb:
                best_child=child
                best_ucb=child_ucb
        return best_child
    def select(self, node):
        '''
        选择以便于扩展的节点，选择策略为 UCB 策略
        '''
        action_list = list(node.state.get_legal_actions(node.color))
        if node.color=='X':
            new_color='O'
        else:
            new_color='X'
        action_list2 = list(node.state.get_legal_actions(new_color))
        while len(action_list)>0 or len(action_list2)>0:
            if len(action_list)==0:
                if len(node.children)==0:
                    new_node=self.expand(node)
                    return new_node
                else:
                    node=node.children[0]
                    action_list = list(node.state.get_legal_actions(node.color))
                    if node.color=='X':
                        new_color='O'
                    else:
                        new_color='X'
                    action_list2 = list(node.state.get_legal_actions(new_color))
                    continue
            if node.full_expand():
                node=self.calu_ucb(node)
                action_list = list(node.state.get_legal_actions(node.color))
                if node.color=='X':
                    new_color='O'
```

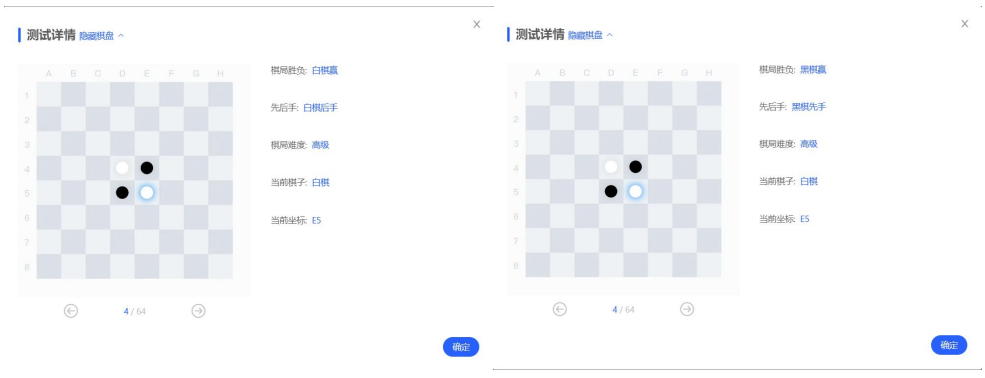
```
                else:
                    new_color='X'
                action_list2 = list(node.state.get_legal_actions(new_color))
                continue
            new_node=self.expand(node)
            return new_node
        return node


    def expand(self, node):
        '''
        扩展节点
        '''
        if node.color=='X':
            new_color='O'
        else:
            new_color='X'
        new_state=copy.deepcopy(node.state)
        action_list = list(node.state.get_legal_actions(node.color))
        if(len(action_list)==0):
            new_node=node.add(new_state,None,new_color)
            return new_node
        for check_action in action_list:
            choice=True
            for child in node.children:
                if check_action==child.action:
                    choice=False
                    break
            if choice==True:
                new_state._move(check_action,node.color)
                new_node=node.add(new_state,check_action,new_color)
                return new_node
        return None
    def simulate(self, node):
        '''
        模拟整个对局，我们可以使用两个 Random Player 来进行模拟
        '''
        board=copy.deepcopy(node.state)
        action_list=list(node.state.get_legal_actions(node.color))
        if node.color=='X':
            new_color='O'
        else:
            new_color='X'
        current_color=node.color
```

```
            action_list2=list(node.state.get_legal_actions(new_color))
            while len(action_list)>0 or len(action_list2)>0:
                if len(action_list) > 0:
                    action=random.choice(action_list)
                    board._move(action,current_color)
                if current_color=='X':
                    current_color='O'
                    new_color='X'
                else:
                    current_color='X'
                    new_color='O'
                action_list=list(board.get_legal_actions(current_color))
                action_list2=list(board.get_legal_actions(new_color))
            winner,difference=board.get_winner()
            if winner==2:
                get_val=0.5
            elif (winner==0 and node.color=='X') or (winner==1 and node.color=='O'):
                get_val=1
            else:
                get_val=0
            return get_val
        def backpropagation(self, node, reward):
            '''
            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 not node==None:
                node.value+=reward
                node.visit_num+=1
                reward=1-reward
                node=node.parent
        def choose_best_child(self, node, lieration = 500):
            '''
            选择最佳子节点，用于做 get_move 中的决策
            lieartion: 迭代次数
            迭代 500 次，选择最佳子节点，通过选择，扩展，模拟，BP 四个步骤，找到最佳子节点
            决策依据？
            UCB 值最大的节点
            '''
            while lieration:
                lieration-=1
                selected_node=self.select(node)
                reward=self.simulate(selected_node)
```

```
            self.backpropagation(selected_node,reward)

        return self.calu_ucb(node,0).action

    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'
        """
        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 = None
        action_list = list(board.get_legal_actions(self.color))

        if len(action_list) == 0:
            return None
        elif len(action_list) == 1:
            return action_list[0]
        else:
            root_state=copy.deepcopy(board)
            root=MCT_node(root_state,None,None,self.color)
            action=self.choose_best_child(root)
            return action
        """
        1. 构建根节点
        2. 通过 choose_best_child 函数获取最佳子节点作为 action
        """
```

四、实验结果

高级难度测试结果：（左图为后手，右图为先手）



五、总结

在本次实验中，通过对蒙特卡洛树搜索算法的复现，较为圆满的完成了题设的要求。这也是我第一次较为大量的编写 python 代码，使得我对 python 的类和函数的语法有了进一步体会。在完成过程中，我也遇到了一些小的报错，但都顺利通过查看中间结果以及针对错误信息上查找原因等方式完成。可能的优化方向有：通过长期对模型的训练结果得到更为合理的 c 值设定，乃至分阶段进行 c 值设定；根据题设所给时间调用时间函数，在规定的时间中增加搜索的次数以获得更好的结果等。