**程序报告**

学号：2311061 姓名：马淏怡

1. **问题重述**

（简单描述对问题的理解，从问题中抓住主干，必填）

====================================================================

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。

2、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。

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。

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。

需要实现AIplayer模块的代码。

1. **设计思想**

（所采用的方法，有无对方法加以改进，该方法有哪些优化方向（参数调整，框架调整，或者指出方法的局限性和常见问题），伪代码，理论结果验证等… **思考题，非必填**）

====================================================================

蒙特卡洛树搜索主要四个重要的步骤：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。

1. **代码内容**

（能体现解题思路的主要代码，有多个文件或模块可用多个"===="隔开，必填）

====================================================================

首先定义蒙特卡洛树结点

class Node:

def \_\_init\_\_(self, now\_board, parent=None, action=None, color=""):

self.visits = 0 # 访问次数

self.reward = 0.0 # 期望值

self.now\_board = now\_board # 棋盘状态

self.children = [] # 孩子节点

self.parent = parent # 父节点

self.action = action # 对应动作

self.color = color # 该节点玩家颜色

def get\_ucb(self, ucb\_param):

if self.visits == 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) / float(self.visits))

now\_ucb = self.reward/self.visits + ucb\_param \* explore

return now\_ucb

def add\_child(self, child\_now\_board, action, color):

child\_node = Node(child\_now\_board, parent=self, action=action, color=color)

self.children.append(child\_node)

# 判断是否完全扩展

def full\_expanded(self):

# 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 == 0:

return False

return True

选择过程

def select(self, node):

"""

:param node:某个节点

:return: ucb值最大的叶子

"""

# print(len(node.children))

if len(node.children) == 0: # 叶子，需要扩展

return node

if node.full\_expanded(): # 完全扩展,递归选择ucb最大的孩子

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)

else: # 没有完全扩展就选访问次数为0的孩子

for kid in node.children: # 从左开始遍历

if kid.visits == 0:

return kid

扩展过程

def extend(self, node):

if node.visits == 0: # 自身还没有被访问过，不扩展，直接模拟

return node

else: # 需要扩展,先确定颜色

if node.color == 'X':

new\_color = 'O'

else:

new\_color = 'X'

for action in list(node.now\_board.get\_legal\_actions(node.color)): # 把所有可行节点加入孩子列表，并初始化

new\_board = deepcopy(node.now\_board)

new\_board.\_move(action, node.color)

# 新建节点

node.add\_child(new\_board, action=action, color=new\_color)

if len(node.children) == 0:

return node

return node.children[0] # 返回新的孩子列表的第一个，以供下一步模拟

模拟过程

def stimulate(self, node):

board = deepcopy(node.now\_board)

color = node.color

count = 0

while (not self.game\_over(board)) and count < 50: # 游戏没有结束，就模拟

action\_list = list(node.now\_board.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)

if color == 'X':

color = 'O'

else:

color = 'X'

else: # 不能下，交换选手

if color == 'X':

color = 'O'

else:

color = 'X'

action\_list = list(node.now\_board.get\_legal\_actions(color))

action = random.choice(action\_list)

board.\_move(action, color)

if color == 'X':

color = 'O'

else:

color = 'X'

count = count + 1

# winner:0-黑棋赢，1-白旗赢，2-表示平局

# diff:赢家领先棋子数

winner, diff = board.get\_winner()

if winner == 2:

reward = 0

elif winner == 0:

reward = 10 + diff

else:

reward = -(10 + diff)

if self.color == 'X':

reward = - reward

return reward

反向传播过程

def backup(self, node, reward):

while node is not None:

node.visits += 1

if node.color == self.color:

node.reward += reward

else:

node.reward -= reward

node = node.parent

return 0

判断游戏是否结束函数

def game\_over(self, board):

# 双方都无处可落子，则终止

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

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

is\_over = (len(b\_list) == 0 and len(w\_list) == 0) # 返回 True即结束

return is\_over

1. **实验结果**

（实验结果，必填）

====================================================================

系统测试通过

初级，黑棋先手



初级，白棋先手



高级，黑棋先手



1. **总结**

（自评分析（是否达到目标预期，可能改进的方向，实现过程中遇到的困难，从哪些方面可以提升性能，模型的超参数和框架搜索是否合理等），**思考题，非必填**）

====================================================================

1. 本次实验重点和难点在于实现蒙特卡洛树的四个过程。
2. 基本达到目标预期。可能改进的方向在于是否可以提高程序执行速度，遇到的困难在于一开始确实无从下手，在查阅资料后有初步思路流程先分析四个过程需要什么变量，记录什么，再设计结点。
3. 第二个难点在于判断终局，要正确讨论这种情况下选择、扩展、模拟、反向传播的实现过程。
4. 将UCB1 进行改进，选择阶段通常使用 UCB1 算法选择子节点，改进它可以加速收敛：使用 PUCT（AlphaGo 用的），结合先验概率和 UCB；动态调整探索/利用平衡参数。这可能可以提高性能。
5. 使用轻量级树节点结构，精简节点数据结构，减少内存占用和复制操作比如使用字典或 numpy 数组存储子节点信息。这可能可以提高性能。