**程序报告**

学号：2110508 姓名：杨冰雪

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(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，变为己方棋子，故又称“翻转棋”。棋子双面为红、绿色的成为“苹果棋”。它使用8\*8的棋盘，由两人执黑子和白子轮流下棋，最后子多方为胜。

游戏规则：

棋局开始时黑棋位于 E4 和 D5 ，白棋位于 D4 和 E5。

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。

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**）

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

设计所用算法：蒙特卡洛树搜索算法

MCTS的基本思想是通过模拟随机探索和统计信息来构建搜索树，并基于这些信息动态调整搜索的方向。它主要由四个阶段组成：选择、扩展、模拟、反向传播。

1. 选择

指算法从搜索树的根结点开始，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，如果子节点已经完全扩展，那么就选择子节点中UCB值最大的节点，继续递归进行检查。。

1. 扩展

对于选择的未完全扩展的节点，根据一定的规则进行扩展，生成该节点的所有可能的子节点。这些子节点表示可能的游戏状态或者决策选项。

1. 模拟

对于扩展的子节点，通过模拟进行随机探索，直到达到游戏结束或者达到模拟的深度限制。模拟过程通过随机策略进行。

1. 反向传播

根据模拟的结果，将收益反向传播到根节点，并更新所经过的节点的统计信息，例如访问次数和累计奖励值。

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。

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）

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

def oppo(color):

"""

玩家取反

:param color: 下棋方，'X' - 黑棋，'O' - 白棋

"""

if color == "O":

return "X"

else:

return "O"

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

class TreeNode:

"""

节点

"""

def \_\_init\_\_(self, board,parent=None, color=''):

"""

节点初始化

:param parent:父节点

:param color:玩家颜色

:param reward:奖励值

:param visits:访问次数

:param children:键为动作，值为子节点

:param board:棋盘状态

"""

self.parent = parent

self.color = color

self.reward = 0.0

self.visits = 0

self.children = dict()

self.board=board

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

"""

计算ucb值

:param ucb\_param:探索参数

"""

if self.visits==0:

return sys.maxsize

ucb=self.reward/self.visits+ucb\_param\*sqrt(log(self.parent.visits)/float(self.visits))

return ucb

def full\_expand(self):

"""

判断节点是否完全展开

"""

if(len(self.children))==0:

return False

for k in self.children.keys():

if self.children[k].visits==0:

return False

return True

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

class AIPlayer:

"""

AI 玩家

"""

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

"""

玩家初始化

:param max\_times:最长迭代时间

:param color:玩家颜色

:param ucb\_param:探索参数

"""

self.max\_times = 10

self.color = color

self.ucb\_param=2.0

def mcts(self, root):

"""

玩家初始化

:param max\_times:最长迭代时间

:param color:玩家颜色

:param ucb\_param:探索参数

"""

start = time.time()

while time.time() - start < self.max\_times - 1:

# print(type(root))

select\_node = self.select(root)

expand\_node=self.expand(select\_node)

reward = self.simulation(expand\_node)

self.backprop(expand\_node, reward)

best\_action = None

max\_ucb = -sys.maxsize

for k in root.children.keys():

child\_ucb=root.children[k].get\_ucb(self.ucb\_param)

if child\_ucb>max\_ucb:

max\_ucb=child\_ucb

best\_action=k

return best\_action

def select(self, node):

# print(type(node))

if len(node.children) == 0: #叶节点

return node

if node.full\_expand():

max\_ucb=-sys.maxsize

best\_node=None

for k in node.children.keys():

child\_ucb=node.children[k].get\_ucb(self.ucb\_param)

# print(child\_ucb)

if child\_ucb>max\_ucb:

max\_ucb=child\_ucb

best\_node=node.children[k]

# print(type(best\_node))

# print(type(best\_node))

return self.select(best\_node)

else:

for k in node.children.keys():

if node.children[k].visits==0:

return node.children[k]

def expand(self, node):

if node.visits==0:

return node

else:

new\_color=oppo(node.color)

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

new\_board=deepcopy(node.board)

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

node.children[move]=TreeNode(board=new\_board,parent=node,color=new\_color)

if len(node.children)==0:

return node

for k in node.children.keys():

return node.children[k]

def simulation(self, node):

sim\_board=deepcopy(node.board)

color = node.color

count=0

while (not self.game\_overed(sim\_board)) and count<=50:

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

if not len(action\_list) == 0:

action = random.choice(action\_list)

sim\_board.\_move(action, color)

color = oppo(color)

else:

color = oppo(color)

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

action = random.choice(action\_list)

sim\_board.\_move(action, color)

color = oppo(color)

count+=1

# 价值函数

winner, difference = sim\_board.get\_winner()

if winner == 2: # 平局

reward = 0

elif winner == 1: # 白棋

reward = 10 + difference

else: # 黑棋

reward = -(10 + difference)

if self.color == 'X':

reward = - reward

return reward

def backprop(self, node, reward):

if node==None:

return 0

node.visits += 1

node.reward += reward

self.backprop(node.parent, -reward)

def game\_overed(self, board):

"""

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。

now\_loc = list(board.get\_legal\_actions('X'))

next\_loc = list(board.get\_legal\_actions('O'))

over = len(now\_loc) == 0 and len(next\_loc) == 0 # 返回值 True/False

return over

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--------------------------------------

root=TreeNode(board=deepcopy(board),parent=None,color=self.color)

action = self.mcts(root) # 可设置最大搜索次数

# action = None

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

return action

1. **实验结果**

（实验结果，必填）

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





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**）

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

通过改变最大迭代时间和ucb常量，多次进行实验，发现当最大迭代时间并不是越大越好，当增大到一定程度上，其不但消耗更多时间，还降低了胜率。

改进方向：本算法在模拟过程中采用的是随机策略，这种方法虽然简单，但通常效果不好，可以采取贪婪策略，在每一步中选择当前看起来最好的动作，即具有最高预期收益的动作，或者其他更好的策略。