**程序报告**

学号：2013750 姓名：管昀玫

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）

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

**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。**

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**）

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

**使用蒙特卡洛树搜索算法来实现。**

**1.选择子节点的策略：选择指算法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到达具有还未被扩展过的子节点的节点L。这个向下递归选择过程可由UCB算法来实现，在递归选择过程中记录下每个节点被选择次数和每个节点得到的奖励均值。**

**2.顺序扩展得到未扩展的结点：如果节点 L 不是一个终止节点（或对抗搜索的终局节点），则随机扩展它的一个未被扩展过的后继边缘节点M。**

**3.模拟：若子节点都扩展完毕，则求UCB值最大的子节点。从节点M出发，模拟扩展搜索树，直到找到一个终止节点。模拟过程使用的策略和采用UCB算法实现的选择过程并不相同，前者通常会使用比较简单的策略，例如使用随机策略。**

**4.反向传播：用模拟所得结果（终止节点的代价或游戏终局分数）回溯更新模拟路径中M以上（含M）节点的奖励均值和被访问次数。需要返回双方棋子的数量差，并交换棋手。**

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）

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

class MCTS:

def \_\_init\_\_(self, color, board, count=50):

self.color = color

self.root = Node(color, board)

self.count = count

def tree\_policy(self):#选择子节点的策略

node=self.root

while node.is\_terminal()==False:

if node.is\_all\_expand():

if len(node.children)==0:

break

node=self.best\_child(node,True)

else:

sub\_node = self.expand(node)

return sub\_node

return node

def expand(self,node):#顺序扩展得到未扩展的子节点

board = copy.deepcopy(node.board)

actions = list(board.get\_legal\_actions(node.color))

board.\_move(actions[len(node.children)], node.color)

node.add\_child(board)

return node.children[-1]

def best\_child(self,node,is\_exploration):#若子节点都扩展完了，求UCB值最大的子节点

best\_score=-sys.maxsize

best\_sub\_node = []

for sub\_node in node.children:

if is\_exploration:

C=1/math.sqrt(2.0)

else:

C=0.0

left=sub\_node.quality\_value/sub\_node.visit\_times

right=2.0\*math.log(node.visit\_times)/sub\_node.visit\_times

score=left+C\*math.sqrt(right)

if score>best\_score:

best\_sub\_node = sub\_node

return best\_sub\_node

def default\_policy(self,node):#返回双方棋子的数量差

current = copy.deepcopy(node)

while current.is\_terminal()==False:

actions = list(current.board.get\_legal\_actions(current.color))

if len(actions) != 0:

current.board.\_move(random.choice(actions), current.color)

current.color = reverse\_color[current.color] #交换棋手

return current.board.count(self.color)-current.board.count(reverse\_color[self.color])

def backup(self,node,reward):

while node != None:

node.visit\_times\_add\_one()

if(node.color==self.color):

node.quality\_value\_add\_n(reward)

else:

node.quality\_value\_add\_n(-reward)

node = node.parent

def monte\_carlo\_tree\_search(self):#蒙特卡洛树搜索总函数

#computation\_budget=1000

for \_ in range(self.count):

expand\_node = self.tree\_policy()

reward = self.default\_policy(expand\_node)

self.backup(expand\_node,reward)

best\_next\_node = self.best\_child(self.root,False)

return best\_next\_node

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**）

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

由实验结果来看，初级没问题，中级有点困难，高级能过，但是没打赢。

遇到的困难：我感觉最大的困难在于理解蒙特卡洛树的机制。写代码的过程实际上就是理解这一机制的过程，特别是在求UCB值最大的结点和反向传播的过程中，我费了一番心思，总体来说，就算有了伪代码，这次实验也不好写。通过这次实验，我再次感受到理解人工智能算法内涵的重要性，今后的实验中我会加强这一方面的学习。

看到了一个github上大佬的算法，我用来跑了一下，虽然算法的正确率没有提升，但是效率有一定的提升。

主要是在扩展函数上提升：

def expand(self, node):

"""

选择扩展的节点

:param node: 根节点，Node 类

:return: leave:Node 类

"""

# 随机选择动作

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

# 防止尾盘时出现卡死，没有动作可以选择

if len(action\_list) == 0:

return node.parent

action = random.choice(action\_list)

tried\_action = [c.action for c in node.children]

while action in tried\_action:

action = random.choice(action\_list)

# 复制状态并根据动作更新到新状态

new\_state = copy.deepcopy(node.state)

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

# 确定子节点颜色

if node.color == 'X':

new\_color = 'O'

else:

new\_color = 'X'

# 新建节点

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

return node.children[-1]