**程序报告**

学号：2213410 姓名：徐俊智

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）

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

使用蒙特卡洛树搜索（MCTS）算法来实现黑白棋的 AIPlayer ，在有限的时间内探索多种可能的落子路径，并选择一个基于模拟结果最有可能获胜的落子位置。

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**）

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

创建一个Node 类来记录蒙特卡洛树中的节点，AIPlayer 类负责执行 MCTS 算法，包括计算UCB1 值，选择、扩展、模拟、反向传播四个过程，再使用get\_move获取最佳落子位置，最后判断游戏是否结束。

执行 MCTS 算法时，首先从根节点开始进行 max\_time 次迭代，通过选择、模拟、扩展、反向传播四个过程构造出一棵蒙特卡洛树，再从根节点开始向下寻找最佳子节点，即最佳落子位置。

选择步骤，从根节点开始，每次选择 UCB1 值最大的子节点，直到叶子节点。

扩展步骤，如果节点还没有被访问过，不扩展，直接模拟。否则给节点加上一个颜色反转的子节点，进入下一步模拟。

模拟步骤，在模拟游戏结束前，获取当前棋盘状态下所有合法的落子动作，随机执行其中一个落子动作，落子后切换颜色。模拟游戏结束后，返回AIPlayer的得分（获胜返回1，失败返回0，平局返回）。

反向传播步骤，从叶子节点开始，向上回溯至父节点，沿途更新各个父节点的统计信息，包括访问次数加1，根据模拟结果更新节点得分。

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）

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

import math

import random

import copy

# 切换玩家颜色

def SwapColor(color):

if color == 'X':

return 'O'

return 'X'

class Node:

def \_\_init\_\_(self, parent, color, board, action, C=1):

self.parent = parent

self.children = []

self.count = 0 # 节点模拟次数

self.value = 0 # 节点得分

self.board = board # 节点对应的棋盘状态

self.color = color # 当前颜色

self.action = action # 从父节点到当前节点的落子动作

self.C = C # 参数

class AIPlayer:

"""

AI 玩家

"""

def \_\_init\_\_(self, color, max\_time = 60):

"""

玩家初始化

:param color: 下棋方，'X' - 黑棋，'O' - 白棋

"""

self.color = color

self.max\_time = max\_time

# 计算 ucb1 值

def get\_ucb1(self, node, t):

if node.count == 0:

return float('INF')

return node.value / node.count + node.C \* ((2 \* math.log(t) / node.count) \*\* 0.5)

# MCTS 算法

def MCTS(self, root):

root.board.display() # 显示当前的棋盘状态。

for i in range(1, self.max\_time):

select\_node = self.select(root, i) # 选择

leaf\_node = self.expand(select\_node) # 扩展

value = self.stimulate(leaf\_node) # 模拟

self.backup(leaf\_node, value) # 反向传播

max\_ucb = -float('INF')

max\_child = None

# 寻找最佳落子位置

for child in root.children:

ucb1 = self.get\_ucb1(child, self.max\_time)

if ucb1 > max\_ucb:

max\_ucb = ucb1

max\_child = child

return max\_child.action

# 选择

def select(self, node, t):

# 如果当前节点是叶子节点

if len(node.children) == 0:

return node

else:

max\_ucb = -float('INF')

max\_child = None

# 更新 UCB1 值最大的子节点

for child in node.children:

ucb1 = self.get\_ucb1(child, t)

if ucb1 > max\_ucb:

max\_ucb = ucb1

max\_child = child

return self.select(max\_child, t)# 递归选择 UCB1 值最大的子节点

# 扩展

def expand(self, node):

# 如果当前节点没有被访问过

if node.count == 0:

return node

else:

# 遍历所有合法落子动作并创建新的子节点

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

board = copy.deepcopy(node.board)

board.\_move(action, node.color)

node.children.append(Node(node, SwapColor(node.color), board, action))

# 如果没有合法的落子位置

if len(node.children) == 0:

return node

# 随机返回一个子节点

return random.choice(node.children)

# 模拟

def stimulate(self, node):

board = copy.deepcopy(node.board)

while (not self.game\_over(board)):

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

# 如果存在合法落子动作，就随机选择一个动作

if len(legal\_actions) > 0:

action = random.choice(legal\_actions)

board.\_move(action, node.color)

# 切换玩家颜色

node.color = SwapColor(node.color)

# winner: [0,1,2] 代表黑棋获胜、白棋获胜、平局

# diff: 棋子差

winner, diff = board.get\_winner()

if winner == 2:# 平局

return 0

elif (winner == 0 and self.color == 'X') or (winner == 1 and self.color == 'O'):# AIPlayer 获胜

return 1

else:# AIPlayer 失败

return 0

# 反向传播

def backup(self, node, value):

while node is not None:

node.count += 1 # 增加节点的访问次数

node.value += value # 根据模拟结果更新节点的得分

node = node.parent # 向上回溯到父节点

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))

root = Node(None, self.color, board, None)

action = self.MCTS (root)

return action

def game\_over(self, board):

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 # 返回值 True/False

return is\_over

1. **实验结果**

（实验结果，必填）

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

白棋获胜, 领先棋子数：26

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**）

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

我的代码实现了 MCTS 算法的所有关键步骤：选择（Selection）、扩展（Expansion）、模拟（Simulation）和反向传播（Backpropagation），AIPlayer 类能够根据当前棋盘状态，通过 MCTS 算法找到并返回最佳的落子位置。

可能改进的方向：采用Roxanne策略代替随机策略搜索，优化选择（Selection）和扩展（Expansion）阶段，以提高搜索效率和落子质量。

提升性能：根据具体情况调整 max\_time 和 C 参数，以提高搜索效率和落子质量。

max\_time 和 C 参数对算法性能有很大影响。在当前实现中，这些参数是硬编码的，但它们可以根具体情况进行调整。框架搜索的合理性取决于棋盘游戏的复杂性和搜索空间的大小。对于当前的黑白棋游戏，超参数和框架搜索较为合理。