**黑白棋(Mini AlphaGo)程序报告**

学号： 2111033 姓名：艾明旭

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。

规则：

1.游戏规则：

开始时，黑棋位于D5和E4,白棋位于D4和E5。一个合法的落子：

1).在空处落子、并翻转对手一个或多个棋子；

2).新落子位置必须在可以夹住对方的位置上、对方被夹住的棋子翻转。可以是横着夹、竖着夹、对角线夹；

3).任何被夹住的棋子必须被反过来。如果一方没有合法棋步，也就是无论他下在哪里，都无法翻转对方的棋子了，这一轮只能弃权。棋局持续知道棋盘填满或双方都没有合法棋步可下。如果一方落子时间超过1min，或者连续三次落子不合法，则判断该方失败。

2.游戏结束的条件：

（1）整个棋盘满了

（2）一方的棋子已经被对方吃光

（2）两名玩家都没有可以落子的棋盘格

（3）一方落子在非法位置

前3种情况以棋子数目来计算胜负，棋子多的一方获胜;第四种情况判定对方获胜。

3.人机对弈流程：

首先，程序询问用户棋盘的大小。接着，程序询问用户“计算机持黑棋还是白棋”。在本程序中，我们用字母’X’代表黑棋， 用字母’O’代表白棋，并且假设总是黑棋玩家先走。

所以，如果计算机持黑棋，计算机就先走; 否则，程序提示人类玩家先走。每走一步，程序输出棋盘。黑白棋玩家轮流下棋，直到一个玩家无符合规则的落子位置。

此时，程序输出信息“O player has no valid move.”(假设白棋玩家无棋可走)，并且提示黑棋玩家继续下棋。

每走一步，程序除输出棋盘外，还要检测游戏是否结束。如果程序检查出游戏结束，输出输赢信息并中止程序。输赢信息可以是: “O player wins.”, “X player wins.” 或者“Draw!”. 如果用户落子非法，程序应检测到并且输出“Invalid move.”， 结束程序，宣布赢家。

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

蒙特卡罗树搜索大概可以被分成四步。选择，拓展，模拟，反向传播。在开始阶段，搜索树只有一个节点，也就是我们需要决策的局面。搜索树中的每一个节点包含了三个基本信息：代表的局面，被访问的次数，累计评分。

（1）选择

在选择阶段，需要从根节点，也就是要做决策的局面R出发向下选择出一个最急迫需要被拓展的节点N，局面R是是每一次迭代中第一个被检查的节点；对于被检查的局面而言，可能有三种可能：

1.该节点所有可行动作都已经被拓展过

2.该节点有可行动作还未被拓展过

3.这个节点游戏已经结束了

对于这三种可能：

1.如果所有可行动作都已经被拓展过了，那么将使用UCB公式计算该节点所有子节点的UCB值，并找到值最大的一个子节点继续检查。反复向下迭代。

2.如果被检查的局面依然存在没有被拓展的子节点，那么认为这个节点就是本次迭代的的目标节点N，并找出N还未被拓展的动作A。执行拓展

3.如果被检查到的节点是一个游戏已经结束的节点。那么从该节点直接执行反向传播。

每一个被检查的节点的被访问次数在这个阶段都会自增。在反复的迭代之后，我们将在搜索树的底端找到一个节点，来继续后面的步骤。

（2）扩展

在选择阶段结束时候，查找到了一个最迫切被拓展的节点N，以及他一个尚未拓展的动作A。在搜索树中创建一个新的节点Nn作为N的一个新子节点。Nn的局面就是节点N在执行了动作A之后的局面。

（3）模拟

为了让Nn得到一个初始的评分。从Nn开始，让游戏随机进行，直到得到一个游戏结局，这个结局将作为Nn的初始评分。一般使用胜利/失败来作为评分，只有1或者0。

（4）反向传播

在Nn的模拟结束之后，它的父节点N以及从根节点到N的路径上的所有节点都会根据本次模拟的结果来添加自己的累计评分。如果在选择中直接发现了一个游戏结局的话，根据该结局来更新评分。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）

import datetime

import numpy as np

from copy import deepcopy

import math

import random

class Node:

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

self.visits = 0 #访问次数

self.reward = 0.0 #期望值

self.state = state #棋盘状态，Broad类

self.children = [] #子节点

self.parent = parent #父节点

self.action = action #从父节点转移到本节点采取的动作

self.color = color #该节点玩家颜色

# 增加子节点

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

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

self.children.append(child\_node)

# 判断是否完全展开

def full\_expand(self):

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

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

return True

return False

class AIPlayer:

"""

AI 玩家

"""

step = 0

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

"""

玩家初始化

:param color: 下棋方，'X' - 黑棋，'O' - 白棋

"""

# 最大迭代次数

self.max\_times = 60

# 玩家颜色

self.color = color

# UCB超参数

self.SCALAR = 1.1

def select\_expand\_node(self, node):

"""

选择扩展的节点

:param node: 根节点

:return: 拓展节点

"""

while not self.game\_overed(node.state):

if len(node.children) == 0:

new\_node = self.expand(node)

# print(new\_node.action)

return new\_node

elif random.uniform(0, 1) < .5:

node = self.ucb(node, self.SCALAR)

else:

node = self.ucb(node, self.SCALAR)

if not node.full\_expand():

return self.expand(node)

else:

node = self.ucb(node, self.SCALAR)

return node

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 = [child.action for child 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 = 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]

def ucb(self, node, scalar):

"""

选择最佳子节点

:param node: 节点

:param scalar: UCT公式超参数

:return: best\_child:最佳子节点

"""

best\_score = -float('inf')

best\_children = []

for child in node.children:

exploit = child.reward / child.visits

if child.visits == 0:

best\_children = [child]

break

explore = math.sqrt(2.0 \* math.log(node.visits) / float(child.visits))

now\_score = exploit + scalar \* explore

if now\_score == best\_score:

best\_children.append(child)

if now\_score > best\_score:

best\_children = [child]

best\_score = now\_score

if len(best\_children) == 0:

return node.parent

return random.choice(best\_children)

def uct(self, max\_times, root):

"""

根据当前棋盘状态获取最佳落子位置

:param max\_times: 最大搜索次数

:param root: 根节点

:return: action 最佳落子位置

"""

for t in range(max\_times):

# print(t)

leave\_node = self.select\_expand\_node(root)

reward = self.random\_stimulate\_chess(leave\_node)

self.backup(leave\_node, reward)

best\_child = self.ucb(root, 0)

return best\_child.action

def random\_stimulate\_chess(self, node):

"""

模拟随机对弈

:param node: 节点

:return: reward:期望值

在定义期望值时同时考虑胜负关系和获胜的子数，board.get\_winner()会返回胜负关系和获胜子数

在这里定义获胜积10分，每多赢一个棋子多1分

reward = 10 + difference

"""

board = deepcopy(node.state)

color = node.color

count = 0

while not self.game\_overed(board):

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

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

action = random.choice(action\_list)

board.\_move(action, color)

if color == 'X':

color = 'O'

else:

color = 'X'

else:

if color == 'X':

color = 'O'

else:

color = 'X'

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

action = random.choice(action\_list)

board.\_move(action, color)

if color == 'X':

color = 'O'

else:

color = 'X'

count = count + 1

if count >= 50:

break

# 价值函数

winner, difference = board.get\_winner()

if winner == 2:

reward = 0

elif winner == 1:

reward = 10 + difference

else:

reward = -(10 + difference)

if self.color == 'X':

reward = - reward

return reward

def backup(self, node, reward):

"""

反向传播函数

"""

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

def game\_overed(self, state):

"""

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(state.get\_legal\_actions('X'))

next\_loc = list(state.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--------------------------------------

board\_state = deepcopy(board)

root = Node(state = board\_state, color=self.color)

action = self.uct( 60, 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**）

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

该算法只能够赢过初级，一开始以为是迭代次数和超参数的问题，但无论怎样改变都不行，以至于即使参数不变，随着测试次数的提升，胜率也会大大下降。

后来我还采用了多种可能的方法进行优化，始终不能有进一步的突破，算法上面仍然有待改进。