**程序报告**

学号：2111454 姓名：李潇逸

1. **问题重述**

二人于8\*8的棋盘上对弈，要求走过一步后此时走的棋子与另一同色棋子间有且仅有连续的另一颜色的棋子，并将这些棋子全部转换为此时走的棋子的颜色；若无法进行有效的行棋则由对方行棋。填满棋盘后盘上子少者或连续三次无法行棋者负。

1. **设计思想**

使用蒙特卡洛树算法构成AI，从而形成一个小型的Alpha go

1. **代码内容**

# 导入棋盘文件

from board import Board

# 导入黑白棋文件

from game import Game

from copy import deepcopy

import sys

import math

import random

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

# 生个孩子

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

class AIPlayer:

"""

AI 玩家

"""

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

"""

玩家初始化

:param color: 下棋方，'X' - 黑棋，'O' - 白棋

"""

self.color = color

self.max\_times = 50 # 最大迭代次数

self.ucb\_param = 1 # ucb的参数C

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 i in range(max\_times): # 最多模拟max次

selected\_node = self.select(root)

leaf\_node = self.extend(selected\_node)

reward = self.stimulate(leaf\_node)

self.backup(leaf\_node, reward)

max\_node = None # 搜索完成，然后找出最适合的下一步

max\_ucb = -sys.maxsize

for child in root.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 max\_node.action

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

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

def stimulate(self, node):

"""

:param node:模拟起始点

:return: 模拟结果reward

board.get\_winner()会返回胜负关系和获胜子数

考虑胜负关系和获胜的子数，定义获胜积10分，每多赢一个棋子多1分

"""

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，就模拟下棋

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，就交换选手

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:

# 这里逻辑是反的，写出了bug...应该是其他地方逻辑也反了一次，负负得正了...实在不想找bug了对不住

reward = 10 + diff

else:

reward = -(10 + diff)

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\_over(self, board):

"""

判断游戏是否结束

:return: True/False 游戏结束/游戏没有结束

"""

# 根据当前棋盘，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) # 返回值 True/False

return is\_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 = Node(now\_board=deepcopy(board), color=self.color)

action = self.uct(self.max\_times, root)

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

return action

# 初始化棋盘

board = Board()

# 打印初始化棋盘

board.display()

# 人类玩家黑棋初始化

black\_player = AIPlayer("X")

# AI 玩家 白棋初始化

white\_player = AIPlayer("O")

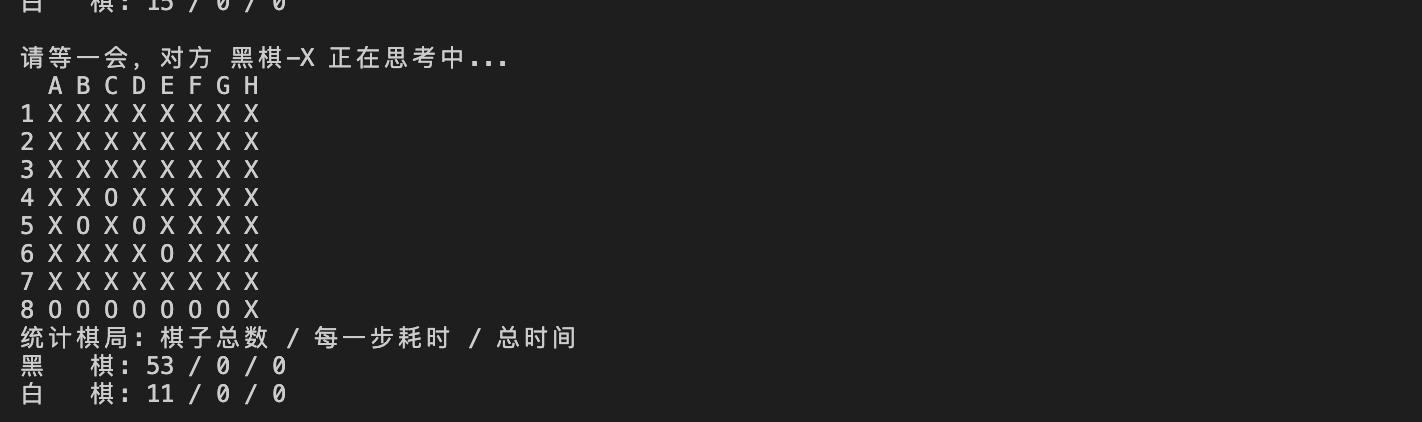
# 游戏初始化，第一个玩家是黑棋，第二个玩家是白棋

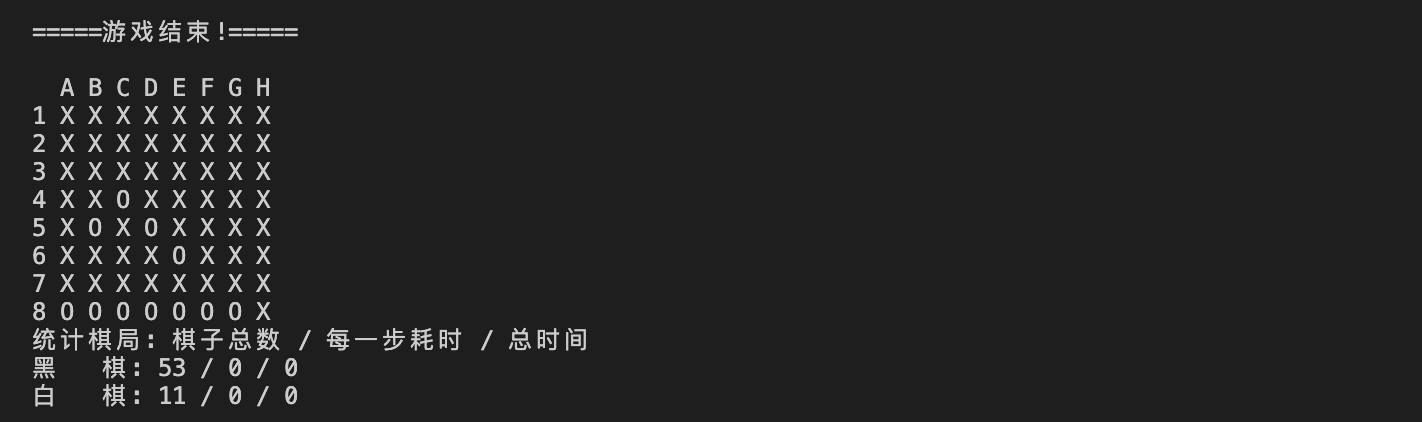
game = Game(black\_player, white\_player)

# 开始下棋

game.run()

1. **实验结果**





1. **总结**

达到了与其设计目的。

改进方向有如下几点：

* 增加人类玩家和随机玩家，使得游戏更加有趣。
* 扩大棋盘，增加游戏思考性。
* 考虑å-ß剪枝，使得AI玩家的思考速度更快。
* 考虑使用诸如QT之类的工具建立图形化界面。