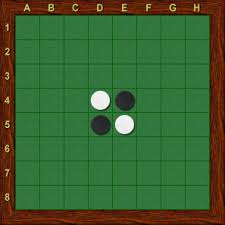
**程序报告**

学号：2113662 姓名：张丛

1. **问题重述**

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

棋局开始时黑棋位于 E4 和 D5 ，白棋位于 D4 和 E5，如图所示。



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。

要求：使用蒙特卡洛树搜索算法实现miniAlphaGo for Reversi.

1. **设计思想**

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

主要是实现AI玩家的蒙塔卡洛树算法部分，即AI的落子位置是经过蒙特卡落树搜索算法后得来的。

在蒙特卡洛树的函数中又分为多个步骤：选择并拓展、模拟、反向传播（回溯）、选择。

将这些步骤又分别编写为函数。

在select\_expend（选择与扩展）中，将判断当前节点是不是叶子节点、节点有没有被扩展过：没有则扩展任意节点，完全扩展则继续选择节点。

在simulate(模拟）中，将模拟到游戏结束。若有合法行动，则任意选择一个合法行动。

模拟结束之后，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。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**

from pyparsing import col

from game import Game

from copy import deepcopy

import math

import random

import time

INF = float('inf') #正无穷

class Node: #蒙特卡洛树节点

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

self.board = board

self.color = color

self.parent = parent

self.children = []

self.action = action

self.visit = 0.0

self.reward = 0.0

def is\_fully\_expanded(self):

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

return len(action\_list) == len(self.children)

def expand\_children(self, next\_board, color, action):

new\_node = Node(next\_board, color, self, action)

self.children.append(new\_node)

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 #下棋方

self.param = 1

self.maxtime = 50 #最大迭代次数

def is\_ended(self, board):

list1 = list(board.get\_legal\_actions('X'))

list2 = list(board.get\_legal\_actions('O'))

return len(list1) == 0 and len(list2) == 0

def UCTSearch(self, board): #蒙塔卡洛树搜索

start\_time = time.time()

root\_node = Node(board, self.color)

while time.time() - start\_time < self.maxtime:

next\_node = self.select\_policy(root\_node) #选择并拓展

reward = self.simualte\_policy(next\_node) #模拟

self.back\_propagate(next\_node, reward) #反向传播

return self.UCB1(root\_node, 0).action #选择回报最大的子节点

def select\_policy(self, node): #在所有选择中，选择UCB值最大的子节点

while not self.is\_ended(node.board):

if not node.is\_fully\_expanded():

new\_node = self.expand(node)

return new\_node

else:

node = self.UCB1(node)

return node

def switch\_color(self, color):

if color == 'X':

new\_color = 'O'

else:

new\_color = 'X'

return new\_color

#模拟并返回奖励

def simualte\_policy(self, node):

board = deepcopy(node.board) #深拷贝棋盘

color = node.color

while not self.is\_ended(board): #模拟直到游戏结束

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

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

action = random.choice(action\_list) #选择任意一个合法行动

board.\_move(action, color)

color = self.switch\_color(color)

else:

color = self.switch\_color(color)

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

action = random.choice(action\_list)

board.\_move(action, color)

color = self.switch\_color(color)

winner, count = board.get\_winner()

if winner == 2:

return 0

reward = count + 15

if (winner == 1 and self.color == 'O') or (winner == 0 and self.color == 'X'):

return reward

return -reward

#反向传播

def back\_propagate(self, node, reward):

while not node == None:

if node.color == self.color:

node.reward -= reward

else:

node.reward += reward

node.visit += 1

node = node.parent

#拓展

def expand(self, node):

new\_color = self.switch\_color(node.color)

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

new\_board = deepcopy(node.board)

expanded\_list = [child.action for child in node.children]

action = random.choice(list(set(action\_list)-set(expanded\_list))) #随机选择一个合法行动

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

if len(list(new\_board.get\_legal\_actions(new\_color))) == 0:

new\_color = node.color

node.expand\_children(new\_board, new\_color, action)

return node.children[-1]

def UCB1(self, node, param=1):

max\_score = -INF

for child in node.children:

avg\_reward = child.reward / child.visit

exploration = param \* \

math.sqrt(2.0 \* math.log(node.visit) / float(child.visit))

score = avg\_reward+exploration

if(score > max\_score):

best\_child = child

max\_score = score

return best\_child

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))

current\_board = deepcopy(board) #深拷贝棋盘，因为算法会改变棋盘状态

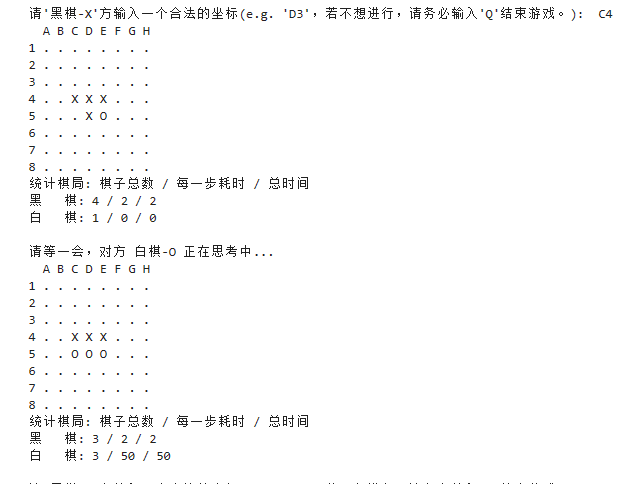
action = self.UCTSearch(current\_board) #蒙特卡洛树搜索

return action

1. **实验结果**

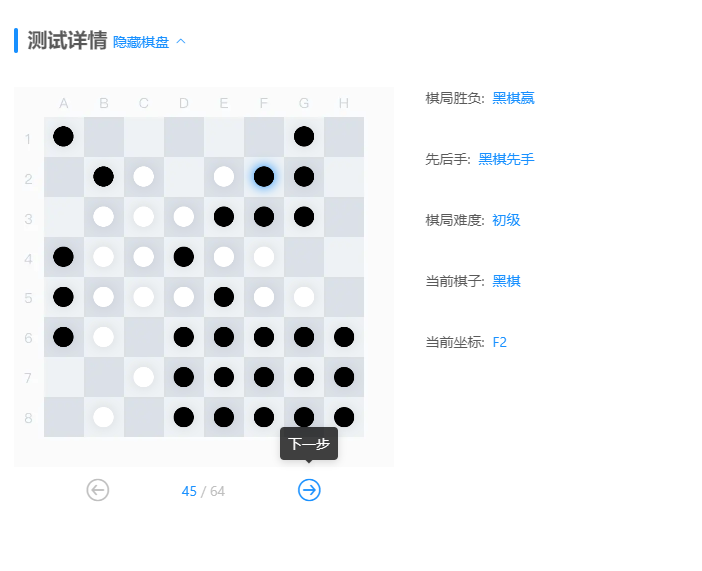
====================================================================

1. 实现人与AI对战：



1. 实现AI与AI对战：





1. **总结**

miniAlphaGo for Reversi的实现其实代码量很大，除了训练AI，还有游戏的搭建也十分繁琐。但此次实验只需要我们熟悉蒙特卡洛树搜索算法，来完成AI的实现。

蒙特卡洛树搜索算法的步骤主要包括选择，拓展，模拟，反向传播。通过编写函数一步步实现此算法，无疑增加了对算法的理解和熟练程度。

而且此次实验的miniAlphaGo的主题，也让人十分感兴趣，激发了对人工智能领域的兴趣，与近几年的热点问题紧密结合。