**黑白棋程序报告**

学号：2212452 姓名：孟启轩

1. **问题重述**

使用 python语言的蒙特卡洛树搜索算法实现 miniAlphaGo for Reversi（黑白棋）。算法部分需要自己实现，不要使用现成的包、工具或者接口。

1. **设计思想**

代码定义了Node和AIPlayer类。

在Node类中，定义了蒙特卡洛树结构，包含访问次数、期望值、棋盘状态、子节点、父节点、从父节点转移到本节点采取的动作动作、玩家持子信息，定义了增加子节点和判断完全展开的函数。

在AIPlayer中，定义了玩家持子和UCB超参数。select\_expand\_node函数实现了扩展结点的选择。expand实现了结点的扩展。ucb函数实现了结点分值的计算并返回最佳结点。uct函数根据当前棋盘状态获取最佳落子位置并返回。random\_stimulate\_chess函数实现了模拟对弈，定义期望值时以是否胜利加胜利子数进行评分。backup函数实现了模拟结果的反向传播。game\_overed函数实现了判断对局是否结束。

本方法在模拟和扩展时有一定的随机性，故可以设计一定的算法（MiniMax或Alpha-Beta剪枝算法）进行有效的扩展和模拟，以使得落子后胜率更大。此外，我们也可以寻找一个更合适的UCB超参数值，实现探索和利用平衡的最优化。

1. **代码内容**

import datetime

import numpy as np

from board import Board

from copy import deepcopy

import math

import random

from game import Game

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

class HumanPlayer:

"""

人类玩家

"""

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

"""

玩家初始化

:param color: 下棋方，'X' - 黑棋，'O' - 白棋

"""

self.color = color

def get\_move(self, board):

"""

根据当前棋盘输入人类合法落子位置

:param board: 棋盘

:return: 人类下棋落子位置

"""

# 如果 self.color 是黑棋 "X",则 player 是 "黑棋"，否则是 "白棋"

if self.color == "X":

player = "黑棋"

else:

player = "白棋"

# 人类玩家输入落子位置，如果输入 'Q', 则返回 'Q'并结束比赛。

# 如果人类玩家输入棋盘位置，e.g. 'A1'，

# 首先判断输入是否正确，然后再判断是否符合黑白棋规则的落子位置

while True:

action = input(

"请'{}-{}'方输入一个合法的坐标(e.g. 'D3'，若不想进行，请务必输入'Q'结束游戏。): ".format(player,

self.color))

# 如果人类玩家输入 Q 则表示想结束比赛

if action == "Q" or action == 'q':

return "Q"

else:

row, col = action[1].upper(), action[0].upper()

# 检查人类输入是否正确

if row in '12345678' and col in 'ABCDEFGH':

# 检查人类输入是否为符合规则的可落子位置

if action in board.get\_legal\_actions(self.color):

return action

else:

print("你的输入不合法，请重新输入!")

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

class RandomPlayer:

"""

随机玩家, 随机返回一个合法落子位置

"""

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

"""

玩家初始化

:param color: 下棋方，'X' - 黑棋，'O' - 白棋

"""

self.color = color

def random\_choice(self, board):

"""

从合法落子位置中随机选一个落子位置

:param board: 棋盘

:return: 随机合法落子位置, e.g. 'A1'

"""

# 用 list() 方法获取所有合法落子位置坐标列表

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

# 如果 action\_list 为空，则返回 None,否则从中选取一个随机元素，即合法的落子坐标

if len(action\_list) == 0:

return None

else:

return random.choice(action\_list)

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))

action = self.random\_choice(board)

return action

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

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

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=50

#玩家颜色

self.color = color

#UCB超参数

self.SCALAR=1.2

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

"""

选择扩展的节点

:param node:根节点

:return: 扩展节点

"""

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

if len(node.children)==0:

new\_node=self.expand(node)

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: leaf，Node类

"""

#随机选择动作

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

#防止卡死，无动作

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")#无穷

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):

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):

"""

模拟随即对弈

:param node:节点

:return: 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

:param node:

:param reward:

:return:

"""

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

:param state:

:return:

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

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))

board\_state=deepcopy(board)

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

action=self.uct(50,root)

return action

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

# 导入黑白棋文件

from game import Game

# 人类玩家黑棋初始化

black\_player = HumanPlayer("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**

本代码利用蒙特卡洛树算法实现了miniAlphaGo for Reversi的AI棋手，可以实现人类玩家、随机玩家与AIPlayer算法对战，AIPlayer与AIPlayer。



1. **总结**

本次实验完成了利用蒙特卡洛树算法实现miniAlphaGo for Reversi的AI棋手，达到基本目标，并且对阵初级难度有一定的胜率，但当面对中级难度，高级难度胜率下降较多，可能是因为扩展及模拟过程中随机选择有关。可以利用MiniMax算法或Alpah-Beta剪枝算法进行优化。除此之外，还可以通过获取更完美的UCB超参数来使探索和利用更加平衡。实验中，对于蒙特卡洛树以及相关代码的理解有一定的难度，通过这次试验，对相关定义以及相关代码的编写有了更深层次的理解。