**程序报告**

学号： 2011937 姓名：姜志凯

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）

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

**基于python，使用蒙特卡洛树搜索算法实现 miniAlphaGo for Reversi。蒙特卡洛树搜索算法，通过不断扩展棋局，得到不同的结果，并记录这个过程每次的得分，平衡探索和利用，找到得分最大的那个路径。通过不断地 选择、扩展、模拟、回溯，得到不同节点的平均的分和访问次数，根据当前的根节点，平衡探索和利用，找到ucb值最大的一个节点，完成动作。这类似于一个学习、实际操作的过程。**

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**）

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

1. **首先创建节点类：表示不同的棋局状态、玩家颜色（即黑棋或白棋）、父节点（即上一步棋）、子节点（下一步动作）、总得分、访问次数。增加子节点的函数add\_child，每扩展一个节点时，就加一个子节点；判断当前节点是否完全扩展的函数**fully\_expanded。
2. 玩家类：

**初始化函数**：确定玩家颜色、最大迭代次数max\_times（即模拟下棋的次数，开辟尽可能多的路径，让AI学习尽可能多的不同的走法，得到尽可能多的不同的终局得分，便于ucb出最优得分，但也要考虑时间限制）、UCB超参数（越大，越倾向于探索）

**get\_move()函数**：根据当前棋局得到最佳落子位置，步骤：

#**加载当前棋盘**

board\_state = copy.deepcopy(board)

#**加载当前根节点**

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

#**蒙特卡洛搜索，搜索出最优的得分**（经过max\_times次的模拟探索，AI已经知道了大部分不同走法的得分，只需根据ucb值选择即可）

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

**Search函数**：蒙特卡洛树搜索具体实现，根据当前盘面获取最佳落子位置，重复搜索max\_time次，扩展尽可能多的节点，最后返回best\_child，步骤（循环max\_times次）：

#**选择扩展**

expand\_node = self.Select(root)

#**模拟**

reward = self.Stimulate(expand\_node)

#**回溯**

self.Back(expand\_node, reward)

#**选择当前盘面的最佳落子位置**

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

**Select函数**：选择并扩展节点，返回被扩展的节点

游戏没结束则继续；如果当前根节点一个都没扩展过，那么直接扩展一个子节点并返回；扩展过，但没扩展干净，则0.8的概率再扩展（expand）一个并返回，0.2的概率在已扩展的节点里选一个ucb值最大的继续选择过程；扩展干净了，则选子节点中ucb最大的继续选择过程。

**expand函数**：扩展的实现

列出所有合法动作，若没有合法落子点，则返回父节点，进入模拟阶段；反之，在合法落子点中随机选一个，若这个点已经被扩展过了（即node.child），则重新在合法落子位置选一个，再判断，直到找到未扩展过的节点；复制状态并根据扩展的节点更新到新状态；确定子节点颜色并新建节点。

**ucb函数**：循环遍历子节点，若未被访问过，直接返回该节点，否则在得分最高的子节点中，随机选一个。

把最高得分初始化为最小的数；分数与最高分一样，则加到可选节点中；出现更高的分数，则更新best\_score和best\_children；在分数一样的节点中随机选一个。

**Stimulate函数**：随机模拟对弈（不用ucb，完全随机）

有合法落子位置，则随机落一个，然后切换庄家，进入下一回合；无合法落子位置，则进入对方回合，随机落子，然后切换庄家，进入下一回合（由于is\_end为false，所以不存在双方都无棋可下的局面）；只要不结束，就一直对弈60次（可以调整，平衡好层数和时间关系）；根据60回合后的局面，判断输赢，得分是领先的棋子数；平局得分0，自己赢则得分为正，否则为负，返回得分。

**Back函数**：回溯。回溯时，自己颜色的节点加这个分数，对方颜色的节点减这个分数。

**is\_end函数**：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，则切换庄家；如果对方也没有合法的下棋位置，则比赛停止。

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 copy

import math

import random

#节点类

class Node:

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

self.color = color #该节点玩家颜色（黑棋或者白棋）

self.parent = parent #父节点

self.children = [] #子节点

self.reward = 0.0 #总得分

self.state = state #棋盘状态

self.visits = 0 #访问次数

self.action = action #从父节点转移到子节点采取的动作

# 增加子节点，每从可以落子的位置中挑选一个扩展，就增加一个子节点

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

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

self.children.append(child\_node)

# 判断是否还有未扩展的节点

def fully\_expanded(self):

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

#当子节点数与合法落子位置数一致时，证明该节点全部扩展

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

return True

return False

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

"""

#玩家颜色

self.color = color

# 最大迭代次数，即模拟下棋的次数，开辟尽可能多的路径，让AI学习尽可能多的不同的走法，

# 得到尽可能多的不同的终局得分，便于ucb出最优得分，但也要考虑时间限制

self.max\_times = 250

# UCB超参数，越大，越倾向于探索

self.SCALAR = 2

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--------------------------------------

#加载当前棋盘

board\_state = copy.deepcopy(board)

#加载当前根节点

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

#蒙特卡洛搜索，搜索出最优的得分（经过max\_times次的模拟探索，AI已经知道了大部分不同走法的得分，只需根据ucb值选择即可）

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

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

return action

#蒙特卡洛树搜索具体实现，根据当前盘面获取最佳落子位置

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

#重复搜索max\_time次，扩展尽可能多的节点

for t in range(max\_times):

#选择扩展

expand\_node = self.Select(root)

#模拟

reward = self.Stimulate(expand\_node)

#回溯

self.Back(expand\_node, reward)

#选择当前盘面的最佳落子位置

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

return best\_child.action

#选择并扩展节点，返回被扩展的节点

def Select(self, node):

#游戏没结束则继续

while not self.is\_end(node.state):

#如果当前根节点一个都没扩展过，那么直接扩展一个子节点并返回

if len(node.children) == 0:

return self.expand(node)

#扩展过，但没扩展干净，则0.8的概率再扩展（expand）一个并返回，0.2的概率在已扩展的节点里选一个ucb值最大的继续选择过程

else:

if not node.fully\_expanded():

if random.uniform(0, 1) < 0.8:

return self.expand(node)

else:

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

#扩展干净了，则选子节点中ucb最大的继续选择过程

else:

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

return node

#扩展的实现

def expand(self, node):

# 列出所有合法动作

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

# 没有合法落子点，则返回父节点，进入模拟阶段

if len(action\_list) == 0:

return node.parent

#在合法落子点中随机选一个，若这个点已经被扩展过了（即node.child），则重新在合法落子位置选一个，再判断，直到找到未扩展过的节点

action = random.choice(action\_list)

tried\_action = [c.action for c in node.children]

while action in tried\_action:

action = random.choice(action\_list)

# 复制状态并根据扩展的节点更新到新状态

new\_state = copy.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'

# 新建子节点

node.add\_child(new\_state, action=action, color=new\_color)

return node.children[-1]

#循环遍历子节点，若未被访问过，直接返回该节点，否则在得分最高的子节点中，随机选一个

def ucb(self, node, scalar):

#把最高得分初始化为最小的数

best\_score = -float('inf')

best\_children = []

for c in node.children:

if c.visits == 0:

best\_children = [c]

break

exploit = c.reward / c.visits

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

score = exploit + scalar \* explore

#分数与最高分一样，则加到可选节点中

if score == best\_score:

best\_children.append(c)

#出现更高的分数，则更新best\_score和best\_children

if score > best\_score:

best\_children = [c]

best\_score = score

if len(best\_children) == 0:

return node.parent

#分数一样的随机选一个

return random.choice(best\_children)

#随机模拟对弈（不用ucb，完全随机）

def Stimulate(self, node):

board = copy.deepcopy(node.state)

color = node.color

#记录对弈回合数

count = 0

#只要不结束，就一直对弈60次

while not self.is\_end(board):

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

#有合法落子位置，则随机落一个，然后切换庄家，进入下一回合

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

action = random.choice(action\_list)

board.\_move(action, color)

if color == 'X':

color = 'O'

else:

color = 'X'

#无合法落子位置，则进入对方回合，随机落子，然后切换庄家，进入下一回合

#由于is\_end为false，所以不存在双方都无棋可下的局面

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

#60回合结束（可以调整，平衡好层数和时间关系）

if count >= 64:

break

# 根据60回合后的局面，判断输赢，得分是领先的棋子数

winner, difference = board.get\_winner()

#平局得分0，自己赢则得分为正，否则为负。

if winner == 2:

reward = 0

elif winner == 1:

reward = difference

else:

reward = -difference

if self.color == 'X':

reward = - reward

return reward

#回溯。回溯时，自己颜色的节点加这个分数，对方颜色的节点减这个分数

def Back(self, node, reward):

while node is not None:

node.visits += 1

if node.parent==None:

node.reward-=reward

break

if node.parent.color == self.color:

node.reward += reward

else:

node.reward -= reward

node = node.parent

return 0

#判断游戏是否结束

def is\_end(self, state):

# 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，则切换庄家；如果对方也没有合法的下棋位置，则比赛停止。

b\_list = list(state.get\_legal\_actions('X'))

w\_list = list(state.get\_legal\_actions('O'))

is\_over = len(b\_list) == 0 and len(w\_list) == 0 # 返回值 True/False

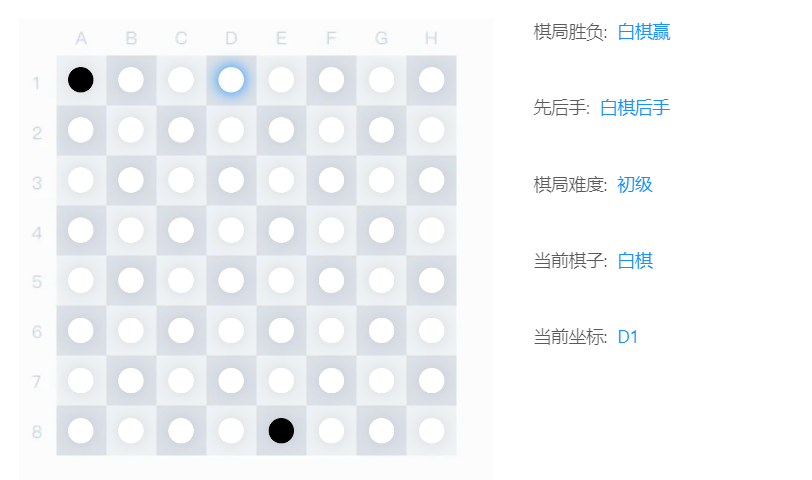
return is\_over

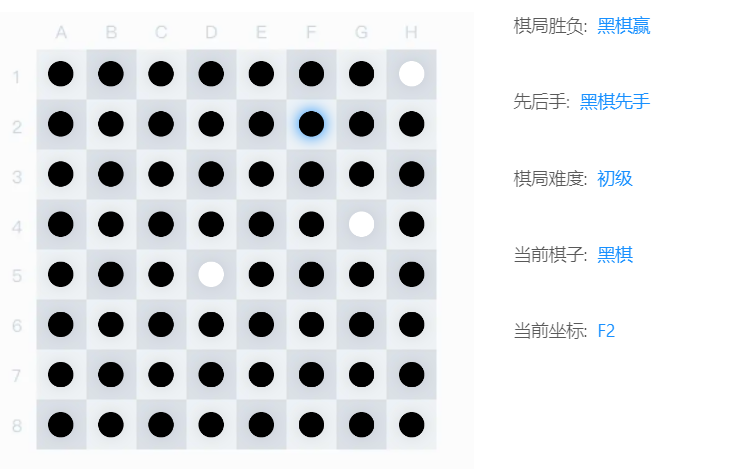
1. **实验结果**

（实验结果，必填）

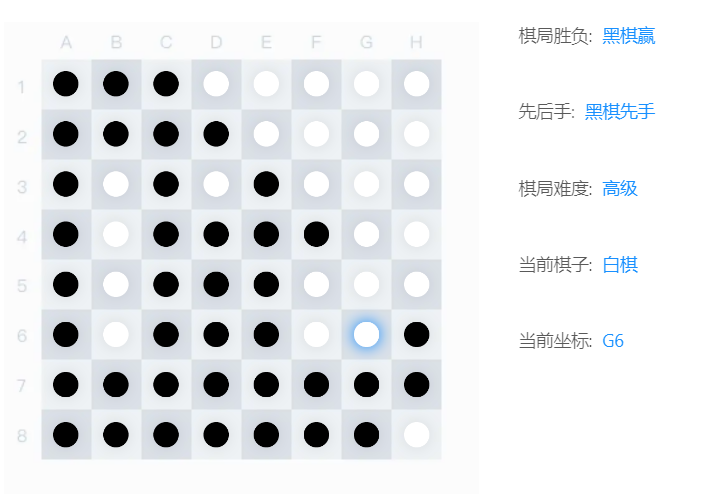
====================================================================

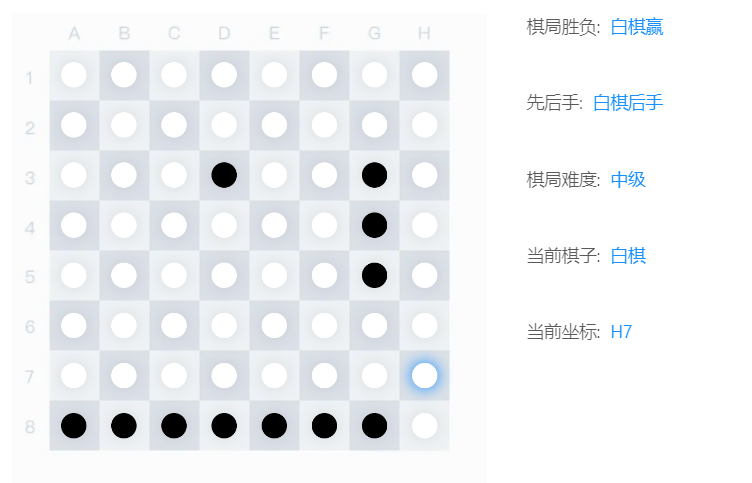
随机测试了初中高级，都获胜了（先后手都有），时间基本在220-260s

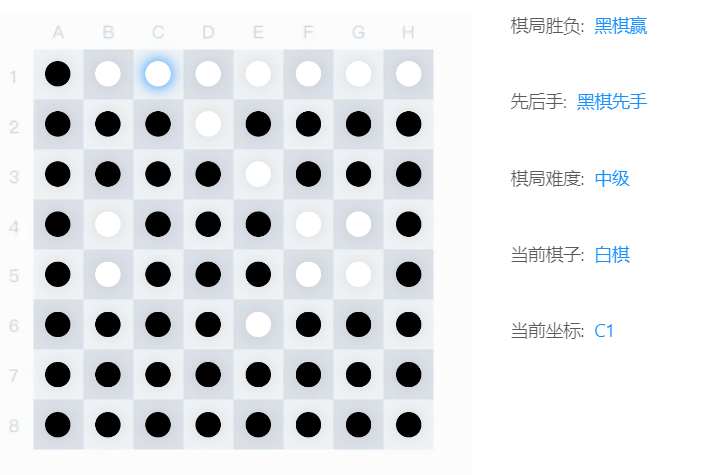
















1. **总结**

（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**）

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

1. 达到了预期目的，碾压初中高三级。

可能的改进（已实现）：

1. 在选择过程中，不只要考虑下一层各节点的**ucb值选择最大的**，还要考虑下一层**未被扩展**的节点，**以一定几率扩展这些节点**，而不是**继续向下选择**，这样的话能保证搜索树的**广度**，而不是反复模拟部分路径，使得这些路径的得分过大，“**一条道跑到黑**”，不能照顾到所有路径。我设定的是0.8的概率扩展未扩展的节点，0.2概率继续向下选择。选择时更倾向于选择同一层未扩展的节点，而不是一直向下（<0.8。
2. reward和超参数C，变换一个就好，reward越大，获胜方奖励越多，越倾向于利用；C越大越倾向于探索，二者是相互制约的，探索与利用一方增减相当于另一方减增。我改变的是超参数C，设为2，reward为获胜方领先的棋子数。
3. 可修改的参数：

**超参数C**、**奖励reward**、**迭代次数max\_times**（即每次选择最佳落子位置时迭代的次数）、**模拟层数**（即每次模拟，对弈的回合数）

（1）超参数越大越倾向于探索，而得分越大则证明奖励越多，越倾向利用；这两个选一个进行调整就行了，没必要都调整；

（2）最大迭代次数，即模拟下棋的次数，开辟尽可能多的路径，让AI学习尽可能多的不同的走法，得到尽可能多的不同的终局得分，便于ucb出最优得分，但也要考虑时间限制；

（3）模拟层数越大，越接近终局，但模拟太多可能超时，所以要做好平衡。