**程序报告**

学号：3190100923 姓名：陈志博

1. **问题重述**

利用蒙特卡洛树搜索的算法来进行黑白棋的博弈。黑白棋的格子虽小，但每个格子都有2种不同的可能性，若考虑所有的黑白组合的话将会是爆炸性的一个数据，必然无法用树将其每一种可能都列出来。因此这次实验采用的是蒙特卡洛树的方法更高效地扩展搜索树，通过采样为主打方式来估计每一个动作的优劣,以在总体可能性无法枚举的情况下尽可能的求得一个近似最优解。

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

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. **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）

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

class node:

def \_\_init\_\_(self, pos, parent):

self.pos = pos

self.children = []

self.parent= parent

self.depth =1

self.win=0

self.times=0

def add(self, node):

self.children.append(node)

node.depth=self.depth+1

def select(self,parent,board1):

maxucb=float('-inf')

for children in parent.children:

#print(children.pos,end='')

if children.times==0:

ucb=float('inf')

else:

c=math.sqrt(2)/3

ucb=children.win/children.times+c\*math.sqrt(2\*math.log(parent.times)/children.times)

if ucb>maxucb:

nextnode=children

maxucb=ucb

if parent.depth%2==1:

nextcolor=self.color

else:

if self.color=='X':

nextcolor='O'

else:

nextcolor='X'

board1.\_move(nextnode.pos,nextcolor)

#board1.display()

l=nextnode.children

#print(nextnode.depth,nextcolor)

#board1.display()

if len(l)==0:

#print(nextnode.pos)

return nextnode

else:

return self.select(nextnode,board1)

def expand(self,vselect,board1):

if vselect.depth%2==1:

color=self.color

else:

if self.color=='X':

color='O'

else:

color='X'

v0=board1.get\_legal\_actions(color)

for i in v0:

vselect.add(node(i,vselect))

if len(list(v0)) ==0:

return vselect

choice = random.choice(vselect.children)

board1.\_move(choice.pos,color)

return choice

def simulate(self,vexpand,board1):

d=vexpand.depth%2

if d==1:

color=self.color

else:

if self.color=='X':

color='O'

else:

color='X'

nextmove=list(board1.get\_legal\_actions(color))

while len(nextmove)!=0:

board1.\_move(random.choice(nextmove),color)

if color=='X':

color='O'

else:

color='X'

nextmove=list(board1.get\_legal\_actions(color))

#print(1)

winner=board1.get\_winner()[0]

if self.color=='X':

b=0

else:

b=1

if winner==b:

#print(-1,end='')

sresult= 1

elif winner==2:

#print(2,end='')

sresult= 0

else:

#print(1,end='')

sresult= -1

#print('###',winner,sresult,self.color)

return sresult

def backpropagate(self,vexpand,sresult):

l=['A1','A8','H1','H8']

v=vexpand

while v!=None:

v.times=v.times+1

if v.pos in l:

if v.depth%2==0:

sresult=sresult+2

else:

sresult=sresult-2

if v.depth%2==0:

#print(v.win,sresult)

v.win=v.win+sresult

else:

v.win=v.win-sresult

v=v.parent

def uctsearch(self,board):

root=node('0',None)

c=math.sqrt(2)/5

v0=board.get\_legal\_actions(self.color)

for i in v0:

root.add(node(i,root))

t0=t1=time.time()

board1=Board()

while t1-t0<40:

board1.\_board=copy.deepcopy(board.\_board)

#board1.display()

vselect=self.select(root,board1)

#print(type(vselect))

#print(1)

vexpand=self.expand(vselect,board1)

#print(2)

if vexpand==vselect:

sresult=self.simulate(vexpand,board1)

else:

winner=board1.get\_winner()[0]

if self.color=='X':

b=0

else:

b=1

if winner==b:

sresult=1

elif winner==2:

sresult=0

else:

sresult=-1

#print('###',winner,sresult,self.color)

#print(3)

#print()

self.backpropagate(vexpand,sresult)

#print(4)

t1=time.time()

#print(root.children[0].pos,root.children[0].win,root.win)

maxucb=float('-inf')

for item in root.children:

if item.times==0:

ucb=float('inf')

else:

c=math.sqrt(2)/3

ucb=item.win/item.times+c\*math.sqrt(2\*math.log(root.times)/item.times)

if ucb>maxucb:

a=item.pos

maxucb=ucb

#print(item.win,item.times,ucb,item.pos)

#print(a)

return a

1. **实验结果**

（实验结果，必填）

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

当每步运行时间设在2秒左右时可以以较大比分战胜随机程序和初级电脑

但中级和高级电脑比较难，每步时间40s时测试能赢1/3场左右，基本棋子数目差距并不太大。

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**）

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

一开始没有关注get\_win函数但返回数据是一个tuple导致浪费来很多时间。

在超参数但设置方面我发现若设置太小会使得随机的部分过少，若第一次扩展该节点的simulate结果是负，则后续的循环将几乎不再会选择该节点，最后的选择几乎集中在一两个节点上，使得样本的偏差巨大。若超参数设置太大则会使得每一步最后采样的次数较为平均，因为ucb值因为较大的容差而差距极小，难以区分，也没有达到较好的优化近似效果。

为了能有更好的效果、更高的效率，我觉得可以在back propagation的过程中的权重做一定优化。黑白棋的基本直觉告诉我们若能在落子过程中占领棋盘的四个角，则能获得更大的胜率。因此我在back propagation函数中加上了对于四个角的检测，若在path的回溯过程中发现有四个角的落子，若是我方执子下的则权重+2，若是对方执子下的则权重/-2.