ab剪枝算法

AlphaBeta剪枝方法是对Minimax方法的优化，它们产生的结果是完全相同的，只不过运行效率不一样。

这种方法的前提假设与Minimax也是一样的：

1）双方都按自己认为的最佳着法行棋。

2）对给定的盘面用一个分值来评估，这个评估值永远是从一方（搜索程序）来评价的，红方有利时给一个正数，黑方有利时给一个负数。（如果红方有利时返回正数，当轮到黑方走棋时，评估值又转换到黑方的观点，如果认为黑方有利，也返回正数，这种评估方法都不适合于常规的算法描述）

3）从我们的搜索程序（通常把它称为Max）看来，分值大的数表示对己方有利，而对于对方Min来说，它会选择分值小的着法。

代码：

class TicTacToe(object):

'''

井字棋游戏:

player:当前落子玩家，-1代表AI，1代表人类

board:棋盘

'''

# 传入谁先手，不传入默认AI先手

def \_\_init\_\_(self,depth,first = -1):

self.depth = depth

self.restart(first)

# 重新开始游戏，player先手

def restart(self,player):

self.board = [0]\*9

self.player = player

self.bestmove = -1

# 落子函数

def go(self,pos):

self.board[pos]=self.player

self.player = -self.player

# 撤销落子函数

def cancel\_go(self,pos):

self.board[pos]=0

self.player = -self.player

def print\_board(self):

print("-"\*20)

for i in range(3):

for j in range(3):

if self.board[i\*3+j]==-1:

print("",end="X ")

elif self.board[i\*3+j]==1:

print("",end="O ")

else:

print("",end="\_ ")

print()

print("-"\*20)

# 游戏未结束返回0，AI胜利返回-1,人类获胜返回1，平局返回2

def get\_winner(self):

if any([var==3 or var ==-3 for var in [sum(self.board[i:i+3]) for i in range(0,9,3)]]):

return -self.player

if any([var==3 or var ==-3 for var in [sum(self.board[i:7+i:3]) for i in range(0,3)]]):

return -self.player

if any([var==3 or var ==-3 for var in [sum(self.board[:9:4]),sum(self.board[2:7:2])]]):

return -self.player

return 0 if self.board.count(0)>0 else 2

def evaluate\_minmax(self):

winner = self.get\_winner()

if winner == 2:

return 0

return 1000000\*winner

def evaluate\_nega\_max(self):

winner = self.get\_winner()

if winner == 2:

return 0

return 1000000\*winner\*self.player

# 极大极小算法

def min\_max(self,depth):

# 搜索深度耗尽或者某一方获胜

winner = self.get\_winner()

if winner!=0 or depth==0:

return self.evaluate\_minmax()

# 继续搜索

# 走步生成

moves = [x for x in range(9) if self.board[x]==0]

# 初始化最优值

bestvalue = -1000000\*self.player

for pos in moves:

self.go(pos)

value = self.min\_max(depth-1)

self.cancel\_go(pos)

if self.player==-1:

if value<=bestvalue:

bestvalue = value

if depth==self.depth:

self.bestmove = pos

else:

if value>=bestvalue:

bestvalue = value

if depth==self.depth:

self.bestmove = pos

return bestvalue

# 负极大算法

def nega\_max(self,depth):

# 搜索深度耗尽或者某一方获胜

winner = self.get\_winner()

if winner!=0 or depth==0:

return self.evaluate\_nega\_max()

# 继续搜索

# 走步生成

moves = [x for x in range(9) if self.board[x]==0]

# 初始化最优值

bestvalue = -1000000

for pos in moves:

self.go(pos)

value = -self.nega\_max(depth-1)

self.cancel\_go(pos)

if value>=bestvalue:

bestvalue = value

if depth == self.depth:

self.bestmove = pos

return bestvalue

# alpha\_beta算法

def alpha\_beta(self,depth,alpha,beta):

# 搜索深度耗尽或者某一方获胜

winner = self.get\_winner()

if winner!=0 or depth==0:

return self.evaluate\_nega\_max()

# 继续搜索

# 走步生成

moves = [x for x in range(9) if self.board[x]==0]

for pos in moves:

self.go(pos)

value = -self.alpha\_beta(depth-1,-beta,-alpha)

self.cancel\_go(pos)

if value>=beta:

if self.depth==depth:

self.bestmove = pos

return beta

if value>alpha:

if self.depth==depth:

self.bestmove = pos

alpha = value

return alpha

# 8是深度，因为估值函数的问题，这里深度要大于等于8才能算正常落子

game = TicTacToe(8)

while game.get\_winner()==0:

# game.min\_max(game.depth)

# game.nega\_max(game.depth)

game.alpha\_beta(game.depth,-1000000,1000000)

game.go(game.bestmove)

game.print\_board()

print("result:",game.get\_winner())