**Minimax with AlphaBeta puring**

Main.py

from minimaxAlphaBeta import \*

def sboard(board):  
 saveFile = True if input(YELLOW + 'DO YOU WANT TO SAVE BOARD AND QUIT(y/n)? ' + WHITE).lower() == 'y' else False  
 if saveFile:  
 filename = input(CYAN + 'ENTER FILE NAME: ' + WHITE)   
 if sparser(board,filename):  
 return True  
  
 return False  
  
  
def lparser(filename):  
 def parseBoard(board):  
 for row in range(bHeight):  
 for col in range(bWidth):  
 if board[row][col]=='.':  
 board[row][col]=' '  
 return board  
 if os.name == 'nt':  
 slash = '\\'  
 else:  
 slash = '/'  
 f = open(dir\_path +slash + "saved-games" + slash +filename + '.txt')  
 result = [[c.replace(' ',' ').lower()for c in l.strip('\n').split(' ')] for l in f.readlines()]  
 board = parseBoard(result)  
 return board  
  
def lboard():  
 loadFlag = True if input(YELLOW + 'DO YOU WANT TO LOAD A BOARD(y/n)? ' + WHITE).lower() == 'y' else False  
 if loadFlag:  
 filename = input(CYAN + 'ENTER FILE NAME: ' + WHITE)  
 board = lparser(filename)  
 return board, loadFlag  
 else:  
 return None, loadFlag  
  
  
def playerTurn(board):  
 Col = input(YELLOW + 'Choose a Column between 1 and 7: ' + WHITE)  
 if not(Col.isdigit()):  
 print(MAGENTA + "Input must be integer!" + WHITE)  
 return playerTurn(board)  
  
 playerMove = int(Col) - 1  
  
 if playerMove < 0 or playerMove > 6:  
 print(MAGENTA + "Column must be between 1 and 7!" + WHITE)  
 return playerTurn(board)  
  
 if not(isColumnValid(board, playerMove)):  
 print(MAGENTA + "The Column you select is full!" + WHITE)  
 return playerTurn(board)  
  
  
 board = makeMove(board, playerMove, HUMAN\_PLAYER)[0]  
 playerFourInRow = findFours(board)  
 return board, playerFourInRow  
  
def playerWins(board):  
 printBoard(board)  
 print(' '+BLUE+"HUMAN WINS !!\n" +WHITE)  
 playagain = True if input(YELLOW +'DO YOU WANT TO PLAY AGAIN(y/n)?'+WHITE).lower() == 'y' else False  
 if playagain:  
 mainFucntion()  
 return 0  
  
def aiTurn(board,depth):  
 aiMove = MiniMaxAlphaBeta(board, depth, AI\_PLAYER)  
 board = makeMove(board, aiMove, AI\_PLAYER)[0]  
 aiFourInRow = findFours(board)  
  
 return board, aiFourInRow  
  
def aiWins(board):  
 printBoard(board)  
 print(' '+RED+"AI WINS !!!!\n" +'\033[1;37;40m')  
 playagain = True if input(YELLOW+'DO YOU WANT TO PLAY AGAIN(y/n)?'+WHITE).lower() == 'y' else False  
 if playagain:  
 mainFucntion()  
 return 0  
  
  
def getDepth():  
 depth = input(YELLOW + 'ENTER DIFFICULTY(1-5): ' + WHITE)  
 if not(depth.isdigit()):  
 print(MAGENTA + 'Input must be integer!' + WHITE)  
 return getDepth()  
  
 depth = int(depth,10)   
  
 if depth < 1 or depth > 5:  
 print(MAGENTA + "Difficulty must be between 1 and 5!" + WHITE)  
 return getDepth()  
 return depth  
  
def mainFucntion():  
 os.system('cls' if os.name == 'nt' else 'clear')  
 board, loadFlag = lboard()  
 if board == None:  
 board = initializeBoard()  
 printBoard(board)  
 depth = getDepth()  
 whileCondition = 1  
 if loadFlag == True:  
 whomStart = True  
 else:  
 whomStart = True if input(YELLOW + 'DO YOU WANT TO START(y/n)? ' + WHITE).lower() == 'y' else False  
 if board == None:  
 board = initializeBoard()  
  
 while(whileCondition):  
 if isBoardFilled(board):  
 print("GAME OVER\n")  
 break  
  
 if whomStart:  
  
 board, playerFourInRow = playerTurn(board)  
 if playerFourInRow:  
 whileCondition = playerWins(board)  
 if whileCondition ==0:  
 break  
 board, aiFourInRow = aiTurn(board,depth)  
 if aiFourInRow:  
 whileCondition = aiWins(board)  
 if whileCondition ==0:  
 break  
 printBoard(board)  
   
 if sboard(board):  
 break  
 else:  
  
 board, aiFourInRow = aiTurn(board,depth)  
 if aiFourInRow:  
 whileCondition = aiWins(board)  
 if whileCondition ==0:  
 break  
 printBoard(board)  
  
 if sboard(board):  
 break  
  
 board, playerFourInRow = playerTurn(board)  
 if playerFourInRow:  
 whileCondition = playerWins(board)  
  
 if whileCondition ==0:  
 break  
  
 printBoard(board)  
  
mainFucntion()

MinimaxAlphaBeta.py

from board import \*  
from random import shuffle  
  
def MiniMaxAlphaBeta(board, depth, player):  
 # get array of possible moves   
 validMoves = getValidMoves(board)  
 shuffle(validMoves)  
 bestMove = validMoves[0]  
 bestScore = float("-inf")  
  
 # initial alpha & beta values for alpha-beta pruning  
 alpha = float("-inf")  
 beta = float("inf")  
  
 if player == AI\_PLAYER: opponent = HUMAN\_PLAYER  
 else: opponent = AI\_PLAYER  
   
 # go through all of those boards  
 for move in validMoves:  
 # create new board from move  
 tempBoard = makeMove(board, move, player)[0]  
 # call min on that new board  
 boardScore = minimizeBeta(tempBoard, depth - 1, alpha, beta, player, opponent)  
 if boardScore > bestScore:  
 bestScore = boardScore  
 bestMove = move  
 return bestMove  
  
def minimizeBeta(board, depth, a, b, player, opponent):  
 validMoves = []  
 for col in range(7):  
 # if column col is a legal move...  
 if isValidMove(col, board):  
 # make the move in column col for curr\_player  
 temp = makeMove(board, col, player)[2]  
 validMoves.append(temp)  
  
 # check to see if game over  
 if depth == 0 or len(validMoves) == 0 or gameIsOver(board):  
 return utilityValue(board, player)  
   
 validMoves = getValidMoves(board)   
 beta = b  
   
 # if end of tree evaluate scores  
 for move in validMoves:  
 boardScore = float("inf")  
 # else continue down tree as long as ab conditions met  
 if a < beta:  
 tempBoard = makeMove(board, move, opponent)[0]  
 boardScore = maximizeAlpha(tempBoard, depth - 1, a, beta, player, opponent)  
  
 if boardScore < beta:  
 beta = boardScore  
 return beta  
  
def maximizeAlpha(board, depth, a, b, player, opponent):  
 validMoves = []  
 for col in range(7):  
 # if column col is a legal move...  
 if isValidMove(col, board):  
 # make the move in column col for curr\_player  
 temp = makeMove(board, col, player)[2]  
 validMoves.append(temp)  
 # check to see if game over  
 if depth == 0 or len(validMoves) == 0 or gameIsOver(board):  
 return utilityValue(board, player)  
  
 alpha = a   
 # if end of tree, evaluate scores  
 for move in validMoves:  
 boardScore = float("-inf")  
 if alpha < b:  
 tempBoard = makeMove(board, move, player)[0]  
 boardScore = minimizeBeta(tempBoard, depth - 1, alpha, b, player, opponent)  
  
 if boardScore > alpha:  
 alpha = boardScore  
 return alpha