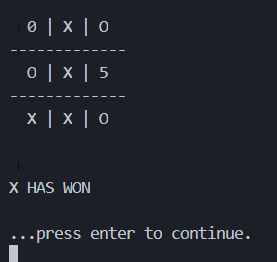
Thomas Mulvey

Project 1.A : TTT WITH MINIMAX

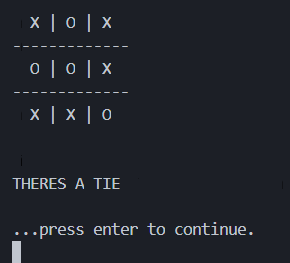
4/1/19

Analysis / Sample Outputs: “X” is COMPUTER and goes first

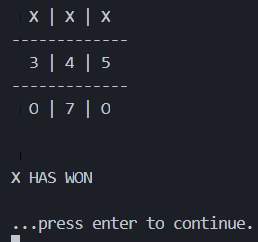
Game 1: HUMAN LOSS



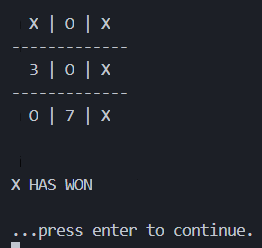
Game 2: TIE



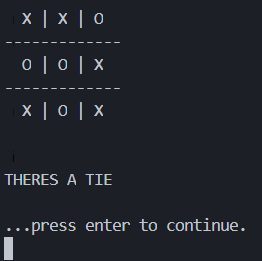
Game 3: HUMAN LOSS



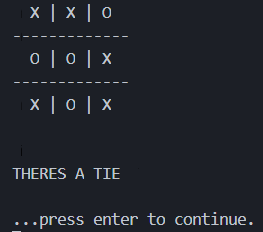
Game 4: HUMAN LOSS



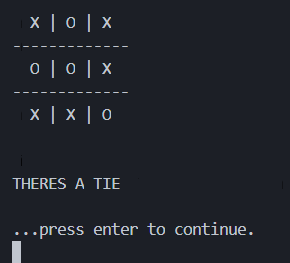
Game 5: TIE



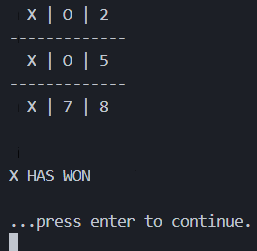
Game 6: TIE



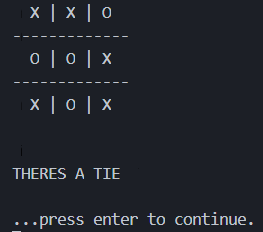
Game 7: TIE



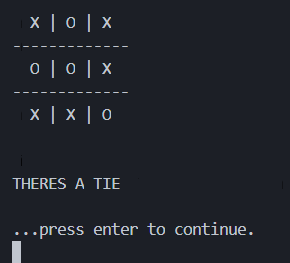
Game 8: HUMAN LOSS



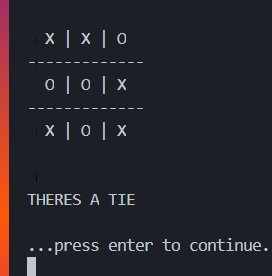
Game 9: TIE



Game 10: TIE



MINIMAX vs MINIMAX: TIE



Results:

Humans vs AI (Human Wins, AI Win, TIE) -- (0, 4, 6)

* Game 3 and 8: AI obviously sees optimal path
* Game 4 and 1: Starting a corner like the AI results in loss
* Starting in middle after corner move is only move that can result in a tie

AI vs AI: always a tie

CODE

TTT GAME

from os import system, name

import time

def clear():

# for windows

if name == 'nt':

\_ = system('cls')

# for mac and linux(here, os.name is 'posix')

else:

\_ = system('clear')

'''

CLASS that defines the board and its behaviour over a tic tac toe game

'''

class TicTacToe:

'''

CONSTRUCTOR: defines board to use in tic tac toe games

constructor makes empty list of 9 \_.

0 | 1 | 2

3 | 4 | 5

6 | 7 | 8

'''

def \_\_init\_\_(self, x, o):

self.board = [ '█' for \_ in range(9) ] # inits board of 9 '\_',

self.turns\_played = 0 #9 max.

self.winning\_moves = ( [0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],[0,4,8],[2,4,6] )

self.winner = None

self.x = x #whoever is x player, real or AI

# note : X ALWAYS GOES FIRST

self.o = o #whoever is x player, real or AI

self.turn = self.x #current turn

'''

returns TRUE if a player has won, or if there is a tie, FALSE otherwise

self.winner is updated with either 'X' 'O' or 'TIE'

'''

def is\_game\_done(self):

if self.num\_turns()<5: #impossible to get win without 5 min moves. save some cpu cycles

return False

else:

for l in self.winning\_moves:

# print("checking " + self.player\_at(l[0]) + " AND " + self.player\_at(l[1]) + " AND " + self.player\_at(l[2]) )

if (self.player\_at(l[0])) == (self.player\_at(l[1])) == (self.player\_at(l[2])) and (self.player\_at(l[0])) !='█' :

self.winner = (self.player\_at(l[0]))

return True

if self.num\_turns() == 9: #no one won and 9 turns made

self.winner = 'TIE'

return True

return False

'''

returns how many valid moves have been played

'''

def num\_turns(self):

return int(self.turns\_played)

'''

given position in board, return 'x', 'o', or '\_'

'''

def player\_at(self, position):

if position > 9 or position < 0:

return "INVALID"

return self.board[position]

'''

print the contents of da board

'''

def display\_board(self):

s=['█' for \_ in range(9)]

for i in range(0,9):

if self.board[i] == '█':

s[i] = i

else:

s[i] = self.board[i]

line1 = " " + str(s[0]) + " | " + str(s[1]) + " | " + str(s[2])

filler = "-------------"

line2 = " " + str(s[3]) + " | " + str(s[4]) + " | " + str(s[5])

line3 = " " + str(s[6]) + " | " + str(s[7]) + " | " + str(s[8])

print(line1)

print(filler)

print(line2)

print(filler)

print(line3)

print("\n")

'''

actually play the game!

'''

def play\_ttt(self):

while True:

clear()

self.turns\_played += 1

if self.turn is self.x:

current\_player = self.x

char = 'X'

else:

current\_player = self.o

char = 'O'

if current\_player.kind == 'human':

self.display\_board()

move = current\_player.move(self.board) #get move from player, validation is done within the current\_players class

self.board[move] = char # place move

if self.is\_game\_done() is True:

clear()

self.display\_board()

if self.winner == "TIE":

print("THERES A TIE")

else:

clear()

self.display\_board()

print(str(self.winner) + " HAS WON")

return True

if self.x.kind == 'MiniMax' == self.o.kind :

self.display\_board()

time.sleep(1.5)

if self.turn == self.x:

self.turn = self.o

else:

self.turn = self.x

AI CLASS:

import random

from Player import \*

# X IS MAX (computer) O IS MINS (human)

# we will return score based off of X's position

class MiniMax(Player):

def \_\_init\_\_(self, char='X'):

self.char = char

self.kind = 'MiniMax'

if self.char == 'X':

self.opponent = 'O'

else :

self.opponent = 'X'

'''

is game done given board state?

returns (TRUE, value of state [10 win, -10 lost] ) or FALSE

'''

def is\_terminal\_state(self, board):

winning\_states = ( [0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],[0,4,8],[2,4,6] )

for a,b,c in winning\_states:

if board[a]==board[b]==board[c]==self.char:

return (True, 10) #minimax won!

elif board[a]==board[b]==board[c]==self.opponent:

return (True, -10) #other player won

space\_counter = 0

for spot in board:

if spot=='█':

space\_counter+=1

if space\_counter==0: #TIE

return (True, 0)

return (False, 0) # aint over yet, chiefton

def move(self, board): #acutal MINIMAX IMPLEMENTATION

# in order to cut down brnaching factor a bit, IF ai

# is going first, just choose a corner.

if len( self.available\_positions(board) ) == 9:

return random.choice( [0,2,6,8] )

# ON THE MINIMAX TURN, YOU WANT THE BEST (MAX) OF THE OTHER PLAYERS TURNS(MIN)

moves=[-10 for \_ in range(9)] #move values

for move in self.available\_positions(board) : # for every child, is it a winner? is a successor a winner? else play random

board[int(move)] = str(self.char)

r = (self.is\_terminal\_state(board))[0]

if (self.is\_terminal\_state(board))[0] is True:

return move

board\_val = self.min\_value(board)

board[move] = '█'

moves[move] = board\_val

# try all moves where its currently a tie, if enemy places at this move, then game over, so place a blocker

c=0

for i in moves:

if i == 0 and board[c] == '█':

board[c] = self.opponent

res = (self.is\_terminal\_state(board))[1]

if int(res) == int(-10):

return c

board[c] = '█'

c+=1

# otherwise play random move

return moves.index(max(moves))

# if cant find a move there, just take a tie from here.

# return random.choice(self.available\_positions(board))

def max\_value(self, board):

board\_done, return\_value = self.is\_terminal\_state(board)

if board\_done: # if current board is done, return -10, 0 , 10

return return\_value

value = -100

for moves in self.available\_positions(board):

board[moves] = self.char

new\_value = self.min\_value(board)

if new\_value > value:

value = new\_value

board[moves] = '█'

return value

def min\_value(self, board):

board\_done, return\_value = self.is\_terminal\_state(board)

if board\_done:

return return\_value

value = 100

for moves in self.available\_positions(board):

board[moves] = self.opponent

new\_value = self.max\_value(board)

if new\_value < value:

value = new\_value

board[moves] = '█'

return value

PLAYER CLASS

class Player:

def \_\_init\_\_(self, char='X'):

self.kind = 'human'

self.char = char

def move(self, board):

while True: #valid move

move = int(input('Your move? '))

if board[move] != "X" and board[move] != "O" and move >= 0 and move <= 9:

return move

def available\_positions(self, board):

return [i for i in range(0, 9) if board[i] == '█']