import sys

import random

# This class represent a tic tac to game

class TicTacToeGame:

# Create a new game

def \_\_init\_\_(self, rows:int, columns:int, goal:int, max\_depth:int=4):

# Create the game state

self.state = []

self.tiles = {}

self.inverted\_tiles = {}

tile = 0

for y in range(rows):

row = []

for x in range(columns):

row += '.'

tile += 1

self.tiles[tile] = (y, x)

self.inverted\_tiles[(y, x)] = tile

self.state.append(row)

self.goal = goal

# Create vectors

self.vectors = [(1,0), (0,1), (1,1), (-1,1)]

# Set lengths

self.rows = rows

self.columns = columns

self.max\_row\_index = rows - 1

self.max\_columns\_index = columns - 1

self.max\_depth = max\_depth

# Heuristics for cutoff

self.winning\_positions = []

self.get\_winning\_positions()

# Set the starting player at random

#self.player = 'O'

self.player = random.choice(['X', 'O'])

def get\_winning\_positions(self):

# Loop the board

for y in range(self.rows):

for x in range(self.columns):

# Loop vectors

for vector in self.vectors:

# Get the start position

sy, sx = (y, x)

# Get vector deltas

dy, dx = vector

# Create a counter

counter = 0

# Loop until we are outside the board

positions = []

while True:

# Add the position

positions.append(self.inverted\_tiles.get((sy, sx)))

# Check if we have a winning position

if (len(positions) == self.goal):

# Add winning positions

self.winning\_positions.append(positions)

# Break out from the loop

break

# Update the position

sy += dy

sx += dx

# Check if the loop should terminate

if(sy < 0 or abs(sy) > self.max\_row\_index or sx < 0 or abs(sx) > self.max\_columns\_index):

break

# Play the game

def play(self):

# Variables

result = None

# Create an infinite loop

print('Starting board')

while True:

# Draw the state

self.print\_state()

# Get a move from a player

if (self.player == 'X'): # AI

# Print AI move

print('Player X moving (AI) ...')

# Get the best move

max, py, px, depth = self.max(-sys.maxsize, sys.maxsize)

# Get a heuristic move at cutoff

print('Depth: {0}'.format(depth))

if(depth > self.max\_depth):

py, px = self.get\_best\_move()

# Make a move

self.state[py][px] = 'X'

# Check if the game has ended, break out from the loop in that case

result = self.game\_ended()

if(result != None):

break

# Change turn

self.player = 'O'

elif (self.player == 'O'): # Human player

# Print turn

print('Player O moving (Human) ...')

# Get a recommended move

min, py, px, depth = self.min(-sys.maxsize, sys.maxsize)

# Get a heuristic move at cutoff

print('Depth: {0}'.format(depth))

if(depth > self.max\_depth):

py, px = self.get\_best\_move()

# Print a recommendation

print('Recommendation: {0}'.format(self.inverted\_tiles.get((py, px))))

# Get input

number = int(input('Make a move (tile number): '))

tile = self.tiles.get(number)

# Check if the move is legal

if(tile != None):

# Make a move

py, px = tile

self.state[py][px] = 'O'

# Check if the game has ended, break out from the loop in that case

result = self.game\_ended()

if(result != None):

break

# Change turn

self.player = 'X'

else:

print('Move is not legal, try again.')

# Print result

self.print\_state()

print('Winner is player: {0}'.format(result))

# An evaluation function to get the best move based on heuristics

def get\_best\_move(self):

# Create an heuristic dictionary

heuristics = {}

# Get all empty cells

empty\_cells = []

for y in range(self.rows):

for x in range(self.columns):

if (self.state[y][x] == '.'):

empty\_cells.append((y, x))

# Loop empty positions

for empty in empty\_cells:

# Get numbered position

number = self.inverted\_tiles.get(empty)

# Loop winning positions

for win in self.winning\_positions:

# Check if number is in a winning position

if(number in win):

# Calculate the number of X:s and O:s in the winning position

player\_x = 0

player\_o = 0

start\_score = 1

for box in win:

# Get the position

y, x = self.tiles[box]

# Count X:s and O:s

if(self.state[y][x] == 'X'):

player\_x += start\_score if self.player == 'X' else start\_score \* 2

start\_score \*= 10

elif (self.state[y][x] == 'O'):

player\_o += start\_score if self.player == 'O' else start\_score \* 2

start\_score \*= 10

# Save heuristic

if(player\_x == 0 or player\_o == 0):

# Calculate a score

score = max(player\_x, player\_o) + start\_score

# Update the score

if(heuristics.get(number) != None):

heuristics[number] += score

else:

heuristics[number] = score

# Get the best move from the heuristic dictionary

best\_move = random.choice(empty\_cells)

best\_count = -sys.maxsize

for key, value in heuristics.items():

if(value > best\_count):

best\_move = self.tiles.get(key)

best\_count = value

# Return the best move

return best\_move

# Check if the game has ended

def game\_ended(self) -> str:

# Check if a player has won

result = self.player\_has\_won()

if(result != None):

return result

# Check if the board is full

for y in range(self.rows):

for x in range(self.columns):

if (self.state[y][x] == '.'):

return None

# Return a tie

return 'It is a tie!'

# Check if a player has won

def player\_has\_won(self) -> str:

# Loop the board

for y in range(self.rows):

for x in range(self.columns):

# Loop vectors

for vector in self.vectors:

# Get the start position

sy, sx = (y, x)

# Get vector deltas

dy, dx = vector

# Create counters

steps = 0

player\_x = 0

player\_o = 0

# Loop until we are outside the board or have moved the number of steps in the goal

while steps < self.goal:

# Add steps

steps += 1

# Check if a player has a piece in the tile

if(self.state[sy][sx] == 'X'):

player\_x += 1

elif(self.state[sy][sx] == 'O'):

player\_o += 1

# Update the position

sy += dy

sx += dx

# Check if the loop should terminate

if(sy < 0 or abs(sy) > self.max\_row\_index or sx < 0 or abs(sx) > self.max\_columns\_index):

break

# Check if we have a winner

if(player\_x >= self.goal):

return 'X'

elif(player\_o >= self.goal):

return 'O'

# Return None if no winner is found

return None

# Get a min value (O)

def min(self, alpha:int=-sys.maxsize, beta:int=sys.maxsize, depth:int=0):

# Variables

min\_value = sys.maxsize

by = None

bx = None

# Check if the game has ended

result = self.game\_ended()

if(result != None):

if result == 'X':

return 1, 0, 0, depth

elif result == 'O':

return -1, 0, 0, depth

elif result == 'It is a tie!':

return 0, 0, 0, depth

elif(depth > self.max\_depth):

return 0, 0, 0, depth

# Loop the board

for y in range(self.rows):

for x in range(self.columns):

# Check if the tile is empty

if (self.state[y][x] == '.'):

# Make a move

self.state[y][x] = 'O'

# Get max value

max, max\_y, max\_x, depth = self.max(alpha, beta, depth + 1)

# Set min value to max value if it is lower than curren min value

if (max < min\_value):

min\_value = max

by = y

bx = x

# Reset the tile

self.state[y][x] = '.'

# Do an alpha test

if (min\_value <= alpha):

return min\_value, bx, by, depth

# Do a beta test

if (min\_value < beta):

beta = min\_value

# Return min value

return min\_value, by, bx, depth

# Get max value (X)

def max(self, alpha:int=-sys.maxsize, beta:int=sys.maxsize, depth:int=0):

# Variables

max\_value = -sys.maxsize

by = None

bx = None

# Check if the game has ended

result = self.game\_ended()

if(result != None):

if result == 'X':

return 1, 0, 0, depth

elif result == 'O':

return -1, 0, 0, depth

elif result == 'It is a tie!':

return 0, 0, 0, depth

elif(depth > self.max\_depth):

return 0, 0, 0, depth

# Loop the board

for y in range(self.rows):

for x in range(self.columns):

# Check if the current tile is empty

if (self.state[y][x] == '.'):

# Add a piece to the board

self.state[y][x] = 'X'

# Set max value to min value if min value is greater than current max value

min, min\_y, min\_x, depth = self.min(alpha, beta, depth + 1)

# Adjust the max value

if (min > max\_value):

max\_value = min

by = y

bx = x

# Reset the tile

self.state[y][x] = '.'

# Do a beta test

if (max\_value >= beta):

return max\_value, bx, by, depth

# Do an alpha test

if (max\_value > alpha):

alpha = max\_value

# Return max value

return max\_value, by, bx, depth

# Print the current game state

def print\_state(self):

for y in range(self.rows):

print('| ', end='')

for x in range(self.columns):

if (self.state[y][x] != '.'):

print(' {0} | '.format(self.state[y][x]), end='')

else:

digit = str(self.inverted\_tiles.get((y,x))) if len(str(self.inverted\_tiles.get((y,x)))) > 1 else ' ' + str(self.inverted\_tiles.get((y,x)))

print('{0} | '.format(digit), end='')

print()

print()

# The main entry point for this module

def main():

# Create a game

#game = TicTacToeGame(7, 6, 4, 1000)

game = TicTacToeGame(3, 3, 3, 1000)

# Play the game

game.play()

# Tell python to run main method

if \_\_name\_\_ == '\_\_main\_\_':

main()