**Monte Carlo Tree Search on Connect-4**

from math import sqrt, log  
from random import choice  
  
  
class ConnectFour:  
 def \_\_init\_\_(self):  
 self.player1 = 'x'  
 self.player2 = 'o'  
 self.height = 6  
 self.width = 7  
 self.connectNumber = 4  
 self.win = 1  
 self.lose = -1  
 self.tie = 0  
  
  
 def resultingState(self, state, action, player):  
 returnState = []  
 for i, col in enumerate(state):  
 if i == action:  
 returnState.append(col + (player,))  
 else:  
 returnState.append(col)  
 return tuple(returnState)  
  
 def isTerminalState(self, state):  
 if all([len(col) == self.height for col in state]):  
 return True   
 if self.gameOutcome(state, self.player1) == self.win or self.gameOutcome(state, self.player1) == self.lose:  
 return True   
 return False  
  
 def actions(self, state):  
 possibleActions = [i for i in range(self.width) if len(state[i]) < self.height]  
 return tuple(possibleActions)  
  
 def nextPlayer(self, player):  
 if self.player1 == player:  
 return self.player2  
 return self.player1  
  
 def streakHandler(self, playerToCompare, p1\_count, p2\_count):  
 if playerToCompare == self.player1:  
 p2\_count = 0   
 p1\_count += 1  
 else:  
 p1\_count = 0   
 p2\_count += 1  
 return p1\_count, p2\_count  
  
 def isGameOverUpDown(self, state):  
 for colIdx in range(self.width):   
 p1\_count, p2\_count = 0, 0  
 for i in range(self.height):   
 try:  
 playerAtCurPos = state[colIdx][i]  
 except IndexError:  
 break   
 p1\_count, p2\_count = self.streakHandler(playerAtCurPos, p1\_count, p2\_count)  
 if p1\_count == self.connectNumber:  
 return True, self.player1  
 if p2\_count == self.connectNumber:  
 return True, self.player2  
 return False, None # no winner found  
  
   
 def isGameOverLeftRight(self, state):  
 for rowIdx in range(self.height):   
 p1\_count, p2\_count = 0, 0  
 for i in range(self.width):   
 try:  
 valueAtCurPos = state[i][rowIdx]  
 except IndexError:  
 p1\_count, p2\_count = 0, 0   
 continue   
 p1\_count, p2\_count = self.streakHandler(valueAtCurPos, p1\_count, p2\_count)   
 if p1\_count == self.connectNumber:  
 return True, self.player1  
 if p2\_count == self.connectNumber:  
 return True, self.player2  
 return False, None   
  
  
 def isGameOverDiag(self, state):  
  
 for col in range(7):  
 for row in range(6):  
 try:  
 temp = state[col][row]  
 temp = state[col + 1][row - 1]  
 temp = state[col + 2][row - 2]  
 temp = state[col + 3][row - 3]  
 except IndexError:  
 continue   
 if row - 3 < 0:  
 continue  
 if state[col][row] == state[col + 1][row - 1] == state[col + 2][row - 2] == \  
 state[col + 3][row - 3] == self.player1:  
 return True, self.player1  
 if state[col][row] == state[col + 1][row - 1] == state[col + 2][row - 2] == \  
 state[col + 3][row - 3] == self.player2:  
 return True, self.player2  
  
 for col in range(7):  
 for row in range(6):  
 try:  
 temp = state[col][row]  
 temp = state[col + 1][row + 1]  
 temp = state[col + 2][row + 2]  
 temp = state[col + 3][row + 3]  
 except IndexError:  
 continue   
 if state[col][row] == state[col + 1][row + 1] == state[col + 2][row + 2] == \  
 state[col + 3][row + 3] == self.player1:  
 return True, self.player1  
 if state[col][row] == state[col + 1][row + 1] == state[col + 2][row + 2] == \  
 state[col + 3][row + 3] == self.player2:  
 return True, self.player2  
 return False, None  
  
 def isGameOver(self, state):  
 upDown = self.isGameOverUpDown(state)  
 upDownBool, upDownPlayer = upDown[0], upDown[1]  
 leftRight = self.isGameOverLeftRight(state)  
 leftRightBool, leftRightPlayer = leftRight[0], leftRight[1]  
 diag = self.isGameOverDiag(state)  
 diagBool, diagPlayer = diag[0], diag[1]  
  
 if upDownBool:  
 return upDownPlayer  
 if leftRightBool:  
 return leftRightPlayer  
 if diagBool:  
 return diagPlayer  
 return None  
  
  
 def gameOutcome(self, state, player):  
 gameOver = self.isGameOver(state)  
 if gameOver == player:  
 return self.win  
 if gameOver is not None:   
 return self.lose  
 return self.tie   
  
  
class Node(ConnectFour):  
 def \_\_init\_\_(self, daddyNode, action, state, player, game=None):  
 super().\_\_init\_\_()   
 self.game = game  
 self.parentNode = daddyNode  
 self.childNodes = dict.fromkeys(self.actions(state)) # creates dict keys that are made of actions.  
 self.action = action  
 self.state = state  
 self.player = player  
 self.visits = 0   
 self.value = 0.0   
  
 def nodeWeightForVisits(self):  
 return self.value / self.visits if self.visits > 0 else 0  
  
  
 def mctsWeightFormula(self, c):  
 return self.nodeWeightForVisits() + c \* sqrt(2 \* log(self.parentNode.visits) / self.visits)  
  
  
 def allChildrenExpanded(self):  
 return None not in self.childNodes.values()  
  
  
 def expandNode(self):  
 try:  
 indexOfNoneNode = list(self.childNodes.values()).index(None)   
 listOfChildNodeKeys = list(self.childNodes.keys())  
 action = listOfChildNodeKeys[indexOfNoneNode]  
 except ValueError:  
 pass   
 newState = self.resultingState(self.state, action, self.player)  
 nextPlayer = self.nextPlayer(self.player)  
 childNode = Node(self, action, newState, nextPlayer)  
 self.childNodes[action] = childNode   
 return childNode   
  
 def optimalChildNode(self, cVal=1 / sqrt(2)):  
 returnValue = None  
 if self.allChildrenExpanded():  
 returnValue = max(self.childNodes.values(), key=lambda node: node.mctsWeightFormula(cVal))  
 return returnValue  
  
 def optimalAction(self, cVal=1 / sqrt(2)):  
 return self.optimalChildNode(cVal).action  
  
  
 def simulate(self):  
 player = self.player  
 state = self.state  
 while not self.isTerminalState(state):   
 nextAction = choice(self.actions(state))   
 state = self.resultingState(state, nextAction, player)  
 player = self.nextPlayer(player)  
 return self.gameOutcome(state, player)  
  
  
def monteCarloTreeSearch(connect4Game, state, player, numOfIterations=4500):  
 rootNode = Node(None, None, state, player, connect4Game)  
 for \_ in range(numOfIterations):  
 curNode = rootNode  
 while not curNode.isTerminalState(curNode.state):   
 if not curNode.allChildrenExpanded():  
 curNode = curNode.expandNode()   
 break  
 curNode = curNode.optimalChildNode()   
   
 deltaValue = curNode.simulate()   
   
 while curNode is not None:  
 curNode.visits += 1  
 curNode.value += deltaValue  
 curNode = curNode.parentNode  
 return rootNode.optimalAction(0)