Calvin Pugmire

CS 470, section 001

Programming Assignment (Adversarial Search II)

1.

Depth limit of 1: 0.1 seconds.

Depth limit of 2: 0.5 seconds.

Depth limit of 3: 1.5 seconds.

Depth limit of 4: 10 seconds.

Depth limit of 5: times out.

2.

Depth limit of 2.

3.

EM is much better about blocking potential losses earlier on.

EM toys with you more often.

4.

Depth limit of 1: 1st: win. 2nd: win.

Depth limit of 2: 1st: win. 2nd: win.

Depth limit of 3: 1st: win. 2nd: win.

Depth limit of 3: 1st: win. 2nd: win.

Depth limit of 4: 1st: win. 2nd: win.

Not in this case.

5.

Game 1: EM=1 vs AB=3: EM win.

Game 2: EM=2 vs AB=3: EM win.

Game 3: EM=3 vs AB=3: EM loss.

Game 5: EM=4 vs AB=3: EM loss.

Game 4: EM=3 vs AB=4: EM win.

Game 6: AB=1 vs EM=3: EM loss.

Game 7: AB=2 vs EM=3: EM win.

Game 8: AB=3 vs EM=3: EM loss.

Game 9: AB=4 vs EM=3: EM loss.

Game 0: AB=5 vs EM=3: EM loss.

Higher depth limit positively impacts AB vs EM.

Higher depth limit negatively impacts EM vs AB.

6.

6 hours.

Revising AB code.

I would revise AB code during the previous PA.

*# Modified 10.3.2023 by Chris Archibald to*  
*# - incorporate MCTS with other code*  
*# - pass command line param string to each AI*  
  
import numpy as np  
  
  
class AIPlayer:  
 def \_\_init\_\_(self, player\_number, name, ptype, param):  
 self.player\_number = player\_number  
 self.name = name  
 self.type = ptype  
 self.player\_string = 'Player {}: '.format(player\_number) + self.name  
 self.other\_player\_number = 1 if player\_number == 2 else 2  
  
 *# Parameters for the different agents*  
  
self.depth\_limit = 3 *# default depth-limit - change if you desire*  
 *# Alpha-beta*  
 *# Example of using command line param to overwrite depth limit*  
if self.type == 'ab' and param:  
 self.depth\_limit = int(param)  
  
 *# Expectimax*  
 *# Example of using command line param to overwrite depth limit*  
if self.type == 'expmax' and param:  
 self.depth\_limit = int(param)  
  
 *# MCTS*  
self.max\_iterations = 1000 *# Default max-iterations for MCTS - change if you desire*  
 *# Example of using command line param to overwrite max-iterations for MCTS*  
if self.type == 'mcts' and param:  
 self.max\_iterations = int(param)  
  
 def get\_alpha\_beta\_move(self, board):  
 *"""*  
 *Given the current state of the board, return the next move based on*  
 *the alpha-beta pruning algorithm*  
  
 *This will play against either itself or a human player*  
  
 *INPUTS:*  
 *board - a numpy array containing the state of the board using the*  
 *following encoding:*  
 *- the board maintains its same two dimensions*  
 *- row 0 is the top of the board and so is*  
 *the last row filled*  
 *- spaces that are unoccupied are marked as 0*  
 *- spaces that are occupied by player 1 have a 1 in them*  
 *- spaces that are occupied by player 2 have a 2 in them*  
  
 *RETURNS:*  
 *The 0 based index of the column that represents the next move*  
 *"""*  
 *# moves = get\_valid\_moves(board)*  
 *# best\_move = np.random.choice(moves)*  
  
 *# YOUR ALPHA-BETA CODE GOES HERE*  
  
my\_board = np.copy(board)  
  
 value, best\_move = self.max\_value(my\_board, self.depth\_limit, float('-inf'), float('inf'))  
  
 return best\_move  
  
 def max\_value(self, board, limit, a, b):  
 v = float('-inf')  
 m = -1  
  
 if is\_winning\_state(board, self.player\_number) or is\_winning\_state(board, self.other\_player\_number) or (limit == 0):  
 return self.evaluation\_function(board), m  
  
 limit -= 1  
  
 for act in get\_valid\_moves(board):  
 my\_board = np.copy(board)  
 make\_move(my\_board, act, self.player\_number)  
 new\_v, new\_m = self.min\_value(my\_board, limit, a, b)  
 if v <= new\_v:  
 v = new\_v  
 m = act  
 if v >= b:  
 return v, m  
 a = max(a, v)  
  
 return v, m  
  
 def min\_value(self, board, limit, a, b):  
 v = float('inf')  
 m = -1  
  
 if is\_winning\_state(board, self.player\_number) or is\_winning\_state(board, self.other\_player\_number) or (limit == 0):  
 return self.evaluation\_function(board), m  
  
 limit -= 1  
  
 for act in get\_valid\_moves(board):  
 my\_board = np.copy(board)  
 make\_move(my\_board, act, self.other\_player\_number)  
 new\_v, new\_m = self.max\_value(my\_board, limit, a, b)  
 if v >= new\_v:  
 v = new\_v  
 m = act  
 if v <= a:  
 return v, m  
 b = min(b, v)  
  
 return v, m  
  
 def get\_mcts\_move(self, board):  
 *"""*  
 *Use MCTS to get the next move*  
 *"""*  
  
 *# How many iterations of MCTS will we do?*  
max\_iterations = 1000 *# Modify to work for you*  
  
 *# Make the MCTS root node from the current board state*  
root = MCTSNode(board, self.player\_number, None)  
  
 *# Run our MCTS iterations*  
for i in range(max\_iterations):  
 *# Select + Expand*  
cur\_node = root.select()  
  
 *# Simulate + backpropate*  
cur\_node.simulate()  
  
 *# Print out the info from the root node*  
root.print\_node()  
 print('MCTS chooses action', root.max\_child())  
 return root.max\_child()  
  
 def get\_expectimax\_move(self, board):  
 *"""*  
 *Given the current state of the board, return the next move based on*  
 *the expectimax algorithm.*  
  
 *This will play against the random player, who chooses any valid move*  
 *with equal probability*  
  
 *INPUTS:*  
 *board - a numpy array containing the state of the board using the*  
 *following encoding:*  
 *- the board maintains its same two dimensions*  
 *- row 0 is the top of the board and so is*  
 *the last row filled*  
 *- spaces that are unoccupied are marked as 0*  
 *- spaces that are occupied by player 1 have a 1 in them*  
 *- spaces that are occupied by player 2 have a 2 in them*  
  
 *RETURNS:*  
 *The 0 based index of the column that represents the next move*  
 *"""*  
 *# moves = get\_valid\_moves(board)*  
 *# best\_move = np.random.choice(moves)*  
  
 *# YOUR EXPECTIMAX CODE GOES HERE*  
  
my\_board = np.copy(board)  
  
 value, best\_move = self.player\_value(my\_board, self.depth\_limit)  
  
 if best\_move == -1:  
 best\_move = get\_valid\_moves(board)[0]  
  
 return best\_move  
  
 def player\_value(self, board, limit):  
 v = float('-inf')  
 m = -1  
  
 if is\_winning\_state(board, self.player\_number) or is\_winning\_state(board, self.other\_player\_number) or (limit == 0):  
 return self.evaluation\_function(board), m  
  
 limit -= 1  
  
 for act in get\_valid\_moves(board):  
 my\_board = np.copy(board)  
 make\_move(my\_board, act, self.player\_number)  
 new\_v, new\_m = self.chance\_value(my\_board, limit)  
 if v <= new\_v:  
 v = new\_v  
 m = act  
  
 return v, m  
  
 def chance\_value(self, board, limit):  
 v = 0  
 m = -1  
  
 if is\_winning\_state(board, self.player\_number) or is\_winning\_state(board, self.other\_player\_number) or (limit == 0):  
 return self.evaluation\_function(board), m  
  
 limit -= 1  
  
 chance = 1 / len(get\_valid\_moves(board))  
 for act in get\_valid\_moves(board):  
 my\_board = np.copy(board)  
 make\_move(my\_board, act, self.other\_player\_number)  
 new\_v, new\_m = self.player\_value(my\_board, limit)  
 v += new\_v\*chance  
  
 return v, m  
  
 def evaluation\_function(self, board):  
 *"""*  
 *Given the current stat of the board, return the scalar value that*  
 *represents the evaluation function for the current player*  
  
 *INPUTS:*  
 *board - a numpy array containing the state of the board using the*  
 *following encoding:*  
 *- the board maintains its same two dimensions*  
 *- row 0 is the top of the board and so is*  
 *the last row filled*  
 *- spaces that are unoccupied are marked as 0*  
 *- spaces that are occupied by player 1 have a 1 in them*  
 *- spaces that are occupied by player 2 have a 2 in them*  
  
 *RETURNS:*  
 *The utility value for the current board*  
 *"""*  
  
 *# YOUR EVALUATION FUNCTION GOES HERE*  
  
 *#player\_board = np.copy(board)*  
 *#player\_board[player\_board == 0] = self.player\_number*  
 *#num\_4s = num\_in\_state(player\_board, '{0}{0}{0}{0}'.format(self.player\_number))*  
num\_4s = num\_in\_state(board, '{0}{0}{0}{0}'.format(self.player\_number))  
 if num\_4s > 0:  
 return float('inf')  
 num\_4b = num\_in\_state(board, '{0}{0}{0}{1}'.format(self.other\_player\_number, self.player\_number))  
 num\_4b += num\_in\_state(board, '{1}{0}{0}{0}'.format(self.other\_player\_number, self.player\_number))  
 num\_4b += num\_in\_state(board, '{0}{0}{1}{0}'.format(self.other\_player\_number, self.player\_number))  
 num\_4b += num\_in\_state(board, '{0}{1}{0}{0}'.format(self.other\_player\_number, self.player\_number))  
 num\_3s = num\_in\_state(board, '{0}{0}{0}'.format(self.player\_number))  
 num\_3b = num\_in\_state(board, '{0}{0}{1}'.format(self.other\_player\_number, self.player\_number))  
 num\_3b += num\_in\_state(board, '{1}{0}{0}'.format(self.other\_player\_number, self.player\_number))  
 num\_2s = num\_in\_state(board, '{0}{0}'.format(self.player\_number))  
  
 *#enemy\_board = np.copy(board)*  
 *#enemy\_board[enemy\_board == 0] = self.other\_player\_number*  
 *#op\_num\_4s = num\_in\_state(enemy\_board, '{0}{0}{0}{0}'.format(self.other\_player\_number))*  
op\_num\_4s = num\_in\_state(board, '{0}{0}{0}{0}'.format(self.other\_player\_number))  
 if op\_num\_4s > 0:  
 return float('-inf')  
 op\_num\_4b = num\_in\_state(board, '{0}{0}{0}{1}'.format(self.player\_number, self.other\_player\_number or 0))  
 op\_num\_4b += num\_in\_state(board, '{1}{0}{0}{0}'.format(self.player\_number, self.other\_player\_number or 0))  
 op\_num\_4b += num\_in\_state(board, '{0}{0}{1}{0}'.format(self.player\_number, self.other\_player\_number or 0))  
 op\_num\_4b += num\_in\_state(board, '{0}{1}{0}{0}'.format(self.player\_number, self.other\_player\_number or 0))  
 op\_num\_3s = num\_in\_state(board, '{0}{0}{0}'.format(self.other\_player\_number))  
 op\_num\_3b = num\_in\_state(board, '{0}{0}{1}'.format(self.player\_number, self.other\_player\_number or 0))  
 op\_num\_3b += num\_in\_state(board, '{1}{0}{0}'.format(self.player\_number, self.other\_player\_number or 0))  
 op\_num\_2s = num\_in\_state(board, '{0}{0}'.format(self.other\_player\_number))  
  
 *#return num\_4s - op\_num\_4s*  
return (10000\*num\_4b + 500\*num\_3s + 20\*num\_3b + 1\*num\_2s) - (10000\*op\_num\_4b + 500\*op\_num\_3s + 20\*op\_num\_3b + 1\*op\_num\_2s)  
  
def num\_in\_state(board, player\_win\_str):  
 to\_str = lambda a: ''.join(a.astype(str))  
  
 def check\_horizontal(b):  
 num = 0  
 for row in b:  
 num += to\_str(row).count(player\_win\_str)  
 return num  
  
 def check\_verticle(b):  
 return check\_horizontal(b.T)  
  
 def check\_diagonal(b):  
 num = 0  
  
 for op in [None, np.fliplr]:  
 op\_board = op(b) if op else b  
  
 root\_diag = np.diagonal(op\_board, offset=0).astype(int)  
 num += to\_str(root\_diag).count(player\_win\_str)  
  
 for i in range(1, b.shape[1] - 3):  
 for offset in [i, -i]:  
 diag = np.diagonal(op\_board, offset=offset)  
 diag = to\_str(diag.astype(int))  
 num += diag.count(player\_win\_str)  
  
 return num  
  
 return (check\_horizontal(board) +  
 check\_verticle(board) +  
 check\_diagonal(board))  
  
  
class RandomPlayer:  
 def \_\_init\_\_(self, player\_number):  
 self.player\_number = player\_number  
 self.type = 'random'  
 self.name = 'random'  
 self.player\_string = 'Player {}: random'.format(player\_number)  
  
 def get\_move(self, board):  
 *"""*  
 *Given the current board state select a random column from the available*  
 *valid moves.*  
  
 *INPUTS:*  
 *board - a numpy array containing the state of the board using the*  
 *following encoding:*  
 *- the board maintains its same two dimensions*  
 *- row 0 is the top of the board and so is*  
 *the last row filled*  
 *- spaces that are unoccupied are marked as 0*  
 *- spaces that are occupied by player 1 have a 1 in them*  
 *- spaces that are occupied by player 2 have a 2 in them*  
  
 *RETURNS:*  
 *The 0 based index of the column that represents the next move*  
 *"""*  
valid\_cols = []  
 for col in range(board.shape[1]):  
 if 0 in board[:, col]:  
 valid\_cols.append(col)  
  
 return np.random.choice(valid\_cols)  
  
  
class HumanPlayer:  
 def \_\_init\_\_(self, player\_number):  
 self.player\_number = player\_number  
 self.type = 'human'  
 self.name = 'human'  
 self.player\_string = 'Player {}: human'.format(player\_number)  
  
 def get\_move(self, board):  
 *"""*  
 *Given the current board state returns the human input for next move*  
  
 *INPUTS:*  
 *board - a numpy array containing the state of the board using the*  
 *following encoding:*  
 *- the board maintains its same two dimensions*  
 *- row 0 is the top of the board and so is*  
 *the last row filled*  
 *- spaces that are unoccupied are marked as 0*  
 *- spaces that are occupied by player 1 have a 1 in them*  
 *- spaces that are occupied by player 2 have a 2 in them*  
  
 *RETURNS:*  
 *The 0 based index of the column that represents the next move*  
 *"""*  
  
valid\_cols = []  
 for i, col in enumerate(board.T):  
 if 0 in col:  
 valid\_cols.append(i)  
  
 move = int(input('Enter your move, Human: '))  
  
 while move not in valid\_cols:  
 print('Column full, choose from:{}'.format(valid\_cols))  
 move = int(input('Enter your move: '))  
  
 return move  
  
  
*# CODE FOR MCTS*  
class MCTSNode:  
 def \_\_init\_\_(self, board, player\_number, parent):  
 self.board = board  
 self.player\_number = player\_number  
 self.other\_player\_number = 1 if player\_number == 2 else 2  
 self.parent = parent  
 self.moves = get\_valid\_moves(board)  
 self.terminal = (len(self.moves) == 0) or is\_winning\_state(board, player\_number) or is\_winning\_state(board,  
 self.other\_player\_number)  
 self.children = dict()  
 for m in self.moves:  
 self.children[m] = None  
  
 *# Set up stats for MCTS*  
 *# Number of visits to this node*  
self.n = 0  
  
 *# Total number of wins from this node (win = +1, loss = -1, tie = +0)*  
 *# Note: these wins are from the perspective of the PARENT node of this node*  
 *# So, if self.player\_number wins, that is -1, while if self.other\_player\_number wins*  
 *# that is a +1. (Since parent will be using our UCB value to make choice)*  
self.w = 0  
  
 *# c value to be used in the UCB calculation*  
self.c = np.sqrt(2)  
  
 def print\_tree(self):  
 *# Debugging utility that will print the whole subtree starting at this node*  
print("\*\*\*\*")  
 self.print\_node(self)  
 for m in self.moves:  
 if self.children[m]:  
 self.children[m].print\_tree()  
 print("\*\*\*\*")  
  
 def print\_node(self):  
 *# Debugging utility that will print this node's information*  
print('Total Node visits and wins: ', self.n, self.w)  
 print('Children: ')  
 for m in self.moves:  
 if self.children[m] is None:  
 print(' ', m, ' is None')  
 else:  
 print(' ', m, ':', self.children[m].n, self.children[m].w, 'UB: ',  
 self.children[m].upper\_bound(self.n))  
  
 def max\_child(self):  
 *# Return the most visited child*  
 *# This is used at the root node to make a final decision*  
max\_n = 0  
 max\_m = None  
  
 for m in self.moves:  
 if self.children[m].n > max\_n:  
 max\_n = self.children[m].n  
 max\_m = m  
 return max\_m  
  
 def upper\_bound(self, N):  
 *# This function returns the UCB for this node*  
 *# N is the number of samples for the parent node, to be used in UCB calculation*  
  
 *# YOUR MCTS TASK 1 CODE GOES HERE*  
  
 *# To do: return the UCB for this node (look in \_\_init\_\_ to see the values you can use)*  
  
return 0  
  
 def select(self):  
 *# This recursive function combines the selection and expansion steps of the MCTS algorithm*  
 *# It will return either:*  
 *# A terminal node, if this is the node selected*  
 *# The new node added to the tree, if a leaf node is selected*  
  
max\_ub = -np.inf *# Track the best upper bound found so far*  
max\_child = None *# Track the best child found so far*  
  
if self.terminal:  
 *# If this is a terminal node, then return it (the game is over)*  
return self  
  
 *# For all of the children of this node*  
for m in self.moves:  
 if self.children[m] is None:  
 *# If this child doesn't exist, then create it and return it*  
new\_board = np.copy(self.board) *# Copy board/state for the new child*  
make\_move(new\_board, m, self.player\_number) *# Make the move in the state*  
  
self.children[m] = MCTSNode(new\_board, self.other\_player\_number, self) *# Create the child node*  
return self.children[m] *# Return it*  
  
 *# Child already exists, get it's UCB value*  
current\_ub = self.children[m].upper\_bound(self.n)  
  
 *# Compare to previous best UCB*  
if current\_ub > max\_ub:  
 max\_ub = current\_ub  
 max\_child = m  
  
 *# Recursively return the select result for the best child*  
return self.children[max\_child].select()  
  
 def simulate(self):  
 *# This function will simulate a random game from this node's state and then call back on its*  
 *# parent with the result*  
  
 *# YOUR MCTS TASK 2 CODE GOES HERE*  
  
 *# Pseudocode in comments:*  
 *#################################*  
 *# If this state is terminal (meaning the game is over) AND it is a winning state for self.other\_player\_number*  
 *# Then we are done and the result is 1 (since this is from parent's perspective)*  
 *#*  
 *# Else-if this state is terminal AND is a winning state for self.player\_number*  
 *# Then we are done and the result is -1 (since this is from parent's perspective)*  
 *#*  
 *# Else-if this is not a terminal state (if it is terminal and a tie (no-one won, then result is 0))*  
 *# Then we need to perform the random rollout*  
 *# 1. Make a copy of the board to modify*  
 *# 2. Keep track of which player's turn it is (first turn is current nodes self.player\_number)*  
 *# 3. Until the game is over:*  
 *# 3.1 Make a random move for the player who's turn it is*  
 *# 3.2 Check to see if someone won or the game ended in a tie*  
 *# (Hint: you can check for a tie if there are no more valid moves)*  
 *# 3.3 If the game is over, store the result*  
 *# 3.4 If game is not over, change the player and continue the loop*  
 *#*  
 *# Update this node's total reward (self.w) and visit count (self.n) values to reflect this visit and result*  
  
 *# Back-propagate this result*  
 *# You do this by calling back on the parent of this node with the result of this simulation*  
 *# This should look like: self.parent.back(result)*  
 *# Tip: you need to negate the result to account for the fact that the other player*  
 *# is the actor in the parent node, and so the scores will be from the opposite perspective*  
pass  
  
 def back(self, score):  
 *# This updates the stats for this node, then backpropagates things*  
 *# to the parent (note the inverted score)*  
self.n += 1  
 self.w += score  
 if self.parent is not None:  
 self.parent.back(-score) *# Score inverted before passing along*  
  
  
*# UTILITY FUNCTIONS*  
  
*# This function will modify the board according to*  
*# player\_number moving into move column*  
def make\_move(board, move, player\_number):  
 row = 0  
 while row < 6 and board[row, move] == 0:  
 row += 1  
 board[row - 1, move] = player\_number  
  
  
*# This function will return a list of valid moves for the given board*  
def get\_valid\_moves(board):  
 valid\_moves = []  
 for c in range(7):  
 if 0 in board[:, c]:  
 valid\_moves.append(c)  
 return valid\_moves  
  
  
*# This function returns true if player\_num is winning on board*  
def is\_winning\_state(board, player\_num):  
 player\_win\_str = '{0}{0}{0}{0}'.format(player\_num)  
 to\_str = lambda a: ''.join(a.astype(str))  
  
 def check\_horizontal(b):  
 for row in b:  
 if player\_win\_str in to\_str(row):  
 return True  
 return False  
  
 def check\_verticle(b):  
 return check\_horizontal(b.T)  
  
 def check\_diagonal(b):  
 for op in [None, np.fliplr]:  
 op\_board = op(b) if op else b  
  
 root\_diag = np.diagonal(op\_board, offset=0).astype(int)  
 if player\_win\_str in to\_str(root\_diag):  
 return True  
  
 for i in range(1, b.shape[1] - 3):  
 for offset in [i, -i]:  
 diag = np.diagonal(op\_board, offset=offset)  
 diag = to\_str(diag.astype(int))  
 if player\_win\_str in diag:  
 return True  
  
 return False  
  
 return (check\_horizontal(board) or  
 check\_verticle(board) or  
 check\_diagonal(board))