Calvin Pugmire

CS 470, section 001

Programming Assignment (Monte Carlo Tree Search)

1.

python ConnectFour.py mcts ab -p2 5 -n 5

|  |  |  |  |
| --- | --- | --- | --- |
|  | C = sqrt(2) | C = 5 | C = 20 |
| Game 1 | Children:  0 : 27 -3 UB: 0.5673734801832426  1 : 73 13 UB: 0.5907118508859458  2 : 71 15 UB: 0.6296685892059573  3 : 148 44 UB: 0.5870923809671533  4 : 65 10 UB: 0.5911317707153608  5 : 45 3 UB: 0.5922185712149421  6 : 71 11 UB: 0.5733305610369431  Player 1: mcts wins! | Children:  0 : 57 -5 UB: 1.5632517209808054  1 : 60 -2 UB: 1.5758341660306223  2 : 71 6 UB: 1.5637779059484793  3 : 127 59 UB: 1.5706175838827763  4 : 60 -3 UB: 1.5591674993639557  5 : 75 9 UB: 1.559283166304462  6 : 50 -10 UB: 1.5627546764116373  Player 1: mcts wins! | Children:  0 : 68 -2 UB: 6.016786894452854  1 : 75 17 UB: 5.983799331884514  2 : 69 1 UB: 6.016718475306911  3 : 73 12 UB: 5.9998481630823655  4 : 72 10 UB: 6.014737810261014  5 : 68 -2 UB: 6.016786894452854  6 : 75 19 UB: 6.01046599855118  Player 1: mcts wins! |
| Game 2 | Children:  0 : 26 -8 UB: 0.3837169609769133  1 : 53 -5 UB: 0.3899263302686732  2 : 28 -8 UB: 0.3805443566001897  3 : 193 36 UB: 0.4403002135667294  4 : 119 7 UB: 0.3820064419109645  5 : 53 -5 UB: 0.3899263302686732  6 : 28 -8 UB: 0.3805443566001897  Player 2: mcts wins! | Children:  0 : 63 -9 UB: 1.4275295371585504  1 : 83 5 UB: 1.4284039326436107  2 : 71 -3 UB: 1.4370173425681978  3 : 101 21 UB: 1.448190654364276  4 : 59 -11 UB: 1.436306534720024  5 : 77 1 UB: 1.4334552111826386  6 : 46 -19 UB: 1.4247553141308351  Player 2: mcts wins! | Children:  0 : 70 -2 UB: 5.93062703067611  1 : 73 3 UB: 5.876560491849489  2 : 72 0 UB: 5.875848921372125  3 : 75 13 UB: 5.930465998551181  4 : 75 11 UB: 5.903799331884514  5 : 67 -13 UB: 5.8971225772453355  6 : 68 -8 UB: 5.928551600335207  Player 2: mcts wins! |
| Game 3 | Children:  0 : 63 9 UB: 0.5870295710667931  1 : 38 0 UB: 0.5719131374247864  2 : 71 13 UB: 0.6014995751214502  3 : 161 53 UB: 0.6070416074091809  4 : 91 21 UB: 0.600343030397825  5 : 45 3 UB: 0.5922185712149421  6 : 31 -1 UB: 0.6009421075959149  Player 1: mcts wins! | Children:  0 : 51 -7 UB: 1.6081323097588407  1 : 59 -1 UB: 1.6057980601437527  2 : 64 4 UB: 1.6205697315737084  3 : 116 53 UB: 1.6142016945561708  4 : 102 38 UB: 1.6067241528110705  5 : 63 3 UB: 1.6180057276347408  6 : 45 -11 UB: 1.6136621334135093  Player 1: mcts wins! | Children:  0 : 67 -5 UB: 6.016525562319963  1 : 71 5 UB: 5.987505989991101  2 : 73 14 UB: 6.027245423356338  3 : 76 23 UB: 6.021762953195233  4 : 75 21 UB: 6.037132665217848  5 : 72 11 UB: 6.0286266991499025  6 : 66 -7 UB: 6.031063435833043  Player 1: mcts wins! |
| Game 4 | Children:  0 : 26 -8 UB: 0.3837169609769133  1 : 54 -4 UB: 0.40568698136074655  2 : 77 -1 UB: 0.38878206517457253  3 : 240 38 UB: 0.38590398350638383  4 : 41 -7 UB: 0.3798603508868026  5 : 30 -8 UB: 0.37700033307891573  6 : 32 -8 UB: 0.3732278926294834  Player 2: mcts wins! | Children:  0 : 70 -5 UB: 1.4183710433833132  1 : 88 10 UB: 1.4423626952503967  2 : 74 -2 UB: 1.4219483913222528  3 : 83 5 UB: 1.4284039326436107  4 : 49 -17 UB: 1.4337123462883201  5 : 67 -6 UB: 1.4332358681919308  6 : 69 -5 UB: 1.4280926623049885  Player 2: mcts wins! | Children:  0 : 69 -7 UB: 5.900776446321403  1 : 75 12 UB: 5.9171326652178475  2 : 68 -10 UB: 5.899139835629324  3 : 75 11 UB: 5.903799331884514  4 : 74 8 UB: 5.904009781505227  5 : 70 -4 UB: 5.9020556021046815  6 : 69 -7 UB: 5.900776446321403  Player 2: mcts wins! |
| Game 5 | Children:  0 : 79 11 UB: 0.5358912980754142  1 : 18 -6 UB: 0.4976371901726448  2 : 70 8 UB: 0.5356646783827503  3 : 227 72 UB: 0.5511768910133419  4 : 14 -6 UB: 0.5136605794379874  5 : 49 1 UB: 0.5240523565257739  6 : 43 -1 UB: 0.5143793220462873  Player 1: mcts wins! | Children:  0 : 71 7 UB: 1.577862412990733  1 : 60 -2 UB: 1.5758341660306223  2 : 82 16 UB: 1.5716020967292013  3 : 80 14 UB: 1.5685799333934651  4 : 73 8 UB: 1.568455191455523  5 : 75 9 UB: 1.559283166304462  6 : 59 -4 UB: 1.554950602516634  Player 1: mcts wins! | Children:  0 : 65 -11 UB: 6.014921730840405  1 : 71 7 UB: 6.015675004075608  2 : 76 22 UB: 6.008605058458391  3 : 74 16 UB: 6.012117889613336  4 : 71 7 UB: 6.015675004075608  5 : 73 13 UB: 6.013546793219352  6 : 70 4 UB: 6.016341316390395  Player 1: mcts wins! |
| Game 6 | Children:  0 : 61 -3 UB: 0.402214889840883  1 : 75 -1 UB: 0.3937574214432789  2 : 46 -6 UB: 0.3893732128139533  3 : 162 20 UB: 0.4004469646254495  4 : 75 -1 UB: 0.3937574214432789  5 : 49 -5 UB: 0.4016033769339371  6 : 32 -8 UB: 0.3732278926294834  Player 2: mcts wins! | Children:  0 : 53 -17 UB: 1.3913839790216451  1 : 65 -9 UB: 1.4075765865562553  2 : 99 13 UB: 1.384048330547009  3 : 105 19 UB: 1.397368672713728  4 : 49 -19 UB: 1.3928960197577078  5 : 58 -14 UB: 1.3952973184523654  6 : 71 -7 UB: 1.3806793143991836  Player 2: mcts wins! | Children:  0 : 70 -4 UB: 5.9020556021046815  1 : 66 -12 UB: 5.955305860075468  2 : 72 2 UB: 5.9036266991499025  3 : 76 14 UB: 5.903341900563654  4 : 75 13 UB: 5.930465998551181  5 : 71 -1 UB: 5.902998947737579  6 : 70 -2 UB: 5.93062703067611  Player 2: mcts wins! |
| Game 7 | Children:  0 : 37 -3 UB: 0.49850908625863083  1 : 79 13 UB: 0.5612077537716167  2 : 75 7 UB: 0.5004240881099455  3 : 82 10 UB: 0.5112785975356139  4 : 101 15 UB: 0.49931614353437936  5 : 25 -5 UB: 0.5051018705646548  6 : 101 15 UB: 0.49931614353437936  Player 1: mcts wins! | Children:  0 : 61 3 UB: 1.6451034250575118  1 : 62 4 UB: 1.647516559312368  2 : 77 19 UB: 1.6672214449488725  3 : 93 34 UB: 1.6581058367802912  4 : 84 26 UB: 1.6695185681821043  5 : 53 -3 UB: 1.6555349224178715  6 : 70 12 UB: 1.661228186240456  Player 1: mcts wins! | Children:  0 : 74 16 UB: 6.012117889613336  1 : 69 1 UB: 6.016718475306911  2 : 75 21 UB: 6.037132665217848  3 : 72 10 UB: 6.014737810261014  4 : 66 -10 UB: 5.985608890378498  5 : 73 15 UB: 6.040944053493325  6 : 71 7 UB: 6.015675004075608  Player 1: mcts wins! |
| Game 8 | Children:  0 : 43 -7 UB: 0.37484443832535713  1 : 56 -4 UB: 0.3996874325761365  2 : 88 1 UB: 0.3871841961337997  3 : 102 4 UB: 0.3882931286184349  4 : 53 -5 UB: 0.3899263302686732  5 : 90 2 UB: 0.39384353779381287  6 : 68 -2 UB: 0.39811904252333297  Player 2: mcts wins! | Children:  0 : 53 -10 UB: 1.5234594507197583  1 : 82 12 UB: 1.5228216089243232  2 : 68 2 UB: 1.5409614294955662  3 : 84 14 UB: 1.5266614253249615  4 : 63 -3 UB: 1.5227676323966455  5 : 79 9 UB: 1.5162963736670212  6 : 71 3 UB: 1.5215243848217188  Player 2: mcts wins! | Children:  0 : 67 -13 UB: 5.8971225772453355  1 : 68 -10 UB: 5.899139835629324  2 : 74 6 UB: 5.8769827544782  3 : 83 37 UB: 5.918435007682876  4 : 71 -3 UB: 5.874829933653072  5 : 70 -4 UB: 5.9020556021046815  6 : 67 -15 UB: 5.867271830976678  Player 1: ab5 wins! |
| Game 9 | Children:  0 : 31 -3 UB: 0.5364259785636568  1 : 59 5 UB: 0.5437279860086357  2 : 145 40 UB: 0.5686396848995743  3 : 29 -3 UB: 0.5512223756568083  4 : 75 11 UB: 0.5537574214432789  5 : 114 26 UB: 0.5582643792838804  6 : 47 1 UB: 0.5355249875199956  Player 1: mcts wins! | Children:  0 : 69 7 UB: 1.6020057057832495  1 : 62 0 UB: 1.58300043028011  2 : 74 12 UB: 1.6111375805114418  3 : 95 31 UB: 1.6051524419806857  4 : 90 26 UB: 1.6027686502595775  5 : 68 5 UB: 1.5850790765543898  6 : 42 -14 UB: 1.5899896991175513  Player 1: mcts wins! | Children:  0 : 67 1 UB: 6.106077801125933  1 : 74 24 UB: 6.120225997721444  2 : 70 12 UB: 6.13062703067611  3 : 79 39 UB: 6.103160178212389  4 : 70 12 UB: 6.13062703067611  5 : 68 2 UB: 6.075610423864618  6 : 72 16 UB: 6.098071143594347  Player 2: ab5 wins! |
| Game 10 | Children:  0 : 62 -4 UB: 0.3832240065166573  1 : 106 5 UB: 0.3895975505213102  2 : 10 -8 UB: 0.31486394671477225  3 : 102 4 UB: 0.3882931286184349  4 : 97 9 UB: 0.45074474530782105  5 : 41 -7 UB: 0.3798603508868026  6 : 82 0 UB: 0.38932737802341877  Player 2: mcts wins! | Children:  0 : 49 -17 UB: 1.4337123462883201  1 : 76 0 UB: 1.4297828435619662  2 : 86 8 UB: 1.4371110958133926  3 : 89 11 UB: 1.4448359955141452  4 : 63 -9 UB: 1.4275295371585504  5 : 71 -3 UB: 1.4370173425681978  6 : 66 -7 UB: 1.4282204044128064  Player 2: mcts wins! | Children:  0 : 70 -4 UB: 5.9020556021046815  1 : 77 17 UB: 5.902652013561724  2 : 68 -10 UB: 5.899139835629324  3 : 77 18 UB: 5.915639026548737  4 : 71 0 UB: 5.917083454779833  5 : 71 -3 UB: 5.874829933653072  6 : 66 -18 UB: 5.864396769166377  Player 2: mcts wins! |

Smaller c values work better than larger c values. Sqrt(2) seems to work the best.

2.

python ConnectFour.py mcts ab -p2 \_ -n 2

|  |  |  |  |
| --- | --- | --- | --- |
|  | max\_iterations (MCTS) | depth\_limit (AB) | MCTS wins / AB wins |
| Game 1 | 10 | 1 | 0 / 4 |
| Game 2 | 25 | 2 | 1 / 3 |
| Game 3 | 50 | 3 | 2 / 2 |
| Game 4 | 75 | 4 | 3 / 1 |
| Game 5 | 100 | 5 | 4 / 0 |

AB performs better with less computational resources.

MCTS performs better as computational resources increase.

3.

MCTS hardly loses to me with 500-1000 max\_iterations.

MCTS performs well compared to AB and EM.

MCTS is definitely the strongest of my agents.

4.

5 hours.

No.

No.

*# 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  
import random  
  
  
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)* u\_b = (self.w / self.n) + (self.c \* np.sqrt(np.log(N) / self.n))  
  
 return u\_b  
  
 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:  
 #################################* result = None  
  
 *# If this state is terminal (meaning the game is over) AND it is a winning state for self.other\_player\_number* if is\_winning\_state(self.board, self.other\_player\_number):  
 *# Then we are done and the result is 1 (since this is from parent's perspective)* result = 1  
 *#  
 # Else-if this state is terminal AND is a winning state for self.player\_number* elif is\_winning\_state(self.board, self.player\_number):  
 *# Then we are done and the result is -1 (since this is from parent's perspective)* result = -1  
 *#  
 # Else-if this is not a terminal state (if it is terminal and a tie (no-one won, then result is 0))* elif len(get\_valid\_moves(self.board)) == 0:  
 result = 0  
 else:  
 *# Then we need to perform the random rollout  
 # 1. Make a copy of the board to modify* my\_board = np.copy(self.board)  
 *# 2. Keep track of which player's turn it is (first turn is current node's self.player\_number)* turn = self.player\_number  
 *# 3. Until the game is over:* while result is None:  
 *# 3.1 Make a random move for the player who's turn it is* act = random.choice(get\_valid\_moves(my\_board))  
 make\_move(my\_board, act, turn)  
 *# 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* if is\_winning\_state(my\_board, self.other\_player\_number):  
 result = 1  
 elif is\_winning\_state(my\_board, self.player\_number):  
 result = -1  
 elif len(get\_valid\_moves(my\_board)) == 0:  
 result = 0  
 *# 3.4 If game is not over, change the player and continue the loop* else:  
 if turn == self.player\_number:  
 turn = self.other\_player\_number  
 else:  
 turn = self.player\_number  
 *#  
 # Update this node's total reward (self.w) and visit count (self.n) values to reflect this visit and result* self.w += result  
 self.n += 1  
 *# 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* self.parent.back(-result)  
  
 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))