

SRM INSTITUTE OF SCIENCE & TECHNOLOGY DEPARTMENT OF NETWORKING & COMMUNICATIONS**18CSC305J-ARTIFICIALINTELLIGENCE**

SEMESTER – 6 BATCH-2

|  |  |
| --- | --- |
| **REGISTRATION NUMBER** | **RA1911003011021** |
| **NAME** | **T V S KRISHNA** |

# B.Tech-CSE / CC, Third Year (Section: H2)

**Faculty In charge: Dr. S. Prabakeran, B.Tech, M.E, PhD**

**Assistant Professor**

**School of Computing - Department of**

**Networking and Communications**

**Year 2021-2022 / Even Semester**

**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ex No** | **DATE** | **Title** | **Page No** | **Marks** |
| 1 | 14/01/2022 | Toy Problem:Tic Tac Toe |  |  |

## Exercise: 1

**Date : 21-01-2021**

**TOY PROBLEM**

Problem Statement :

Two players, named ‘player1’ and ‘player2’, play a tic-tac-toe game on a grid of size ‘3 x 3’. Given an array ‘moves’ of size ‘n’, where each element of the array is a tuple of the form (row, column) representing a position on the grid. Players place their characters alternatively in the sequence of positions given in ‘moves’. Consider that ‘player1’ makes the first move. Your task is to return the winner of the game, i.e., the winning player’s name. If there is no winner and some positions remain unmarked, return ‘uncertain’. Otherwise, the game ends in a draw, i.e., when all positions are marked without any winner, return ‘draw’.

## Algorithm :

 The game is to be played between two people (in this program between HUMAN and COMPUTER).

 One of the player chooses ‘O’ and the other ‘X’ to mark their respective cells.

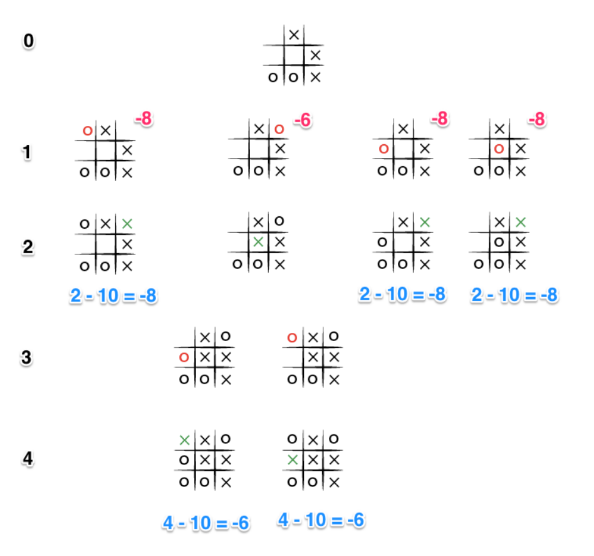
 The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character (‘O’ or ‘X’).

 If no one wins, then the game is said to be draw.

**Optimization technique :** The key is to use Minimax algorithm .A back and forth between the two players, where the player whose "turn it is" desires to pick the move with the maximum score. In turn, the scores for each of the available moves are determined by the opposing player deciding which of its available moves has the minimum score. And the scores for the opposing players moves are again determined by the turn-taking player trying to maximize its score and so on all the way down the move tree to an end state.

A description for the algorithm, assuming X is the "turn taking player,"

* If the game is over, return the score from X's perspective.
* Otherwise get a list of new game states for every possible move
* Create a scores list
* For each of these states add the minimax result of that state to the scores list
* If it's X's turn, return the maximum score from the scores list
* If it's O's turn, return the minimum score from the scores list



**Tool :** VS Code and Python 3.9.0

## Programming code :

## import random

## import copy as cp

## # NOTE use cp.deepcopy() so the temp variable isn't linked with the other

## class Cell:

## def \_\_init\_\_(self, position, location, max\_val, min\_val):

## self.position = position

## self.location = location # NOTE this is a list, [0] is row info and [1] is col info

## self.min\_val = min\_val

## self.max\_val = max\_val

## def generate\_cells(board):

## uboard = cp.deepcopy(board)

## for i in range(len(uboard)):

## for j in range(len(uboard[i])):

## if uboard[i][j] != 'X' and uboard[i][j] != 'O':

## uboard[i][j] = Cell(uboard[i][j], [i,j], 0, 0)

## uboard[i][j].max\_val = max\_val(board, [i, j])

## uboard[i][j].min\_val = min\_val(board, [i, j])

## # NOTE convert uboard[i][j] into list of maxval and minval from objects

## uboard[i][j] = [uboard[i][j].position, uboard[i][j].max\_val, uboard[i][j].min\_val]

## return uboard

## def max\_val(board, location): # NOTE only generates one max\_val by a given location, not entire board

## maxval = 0

## if board[location[0]][location[1]] != 'O' and board[location[0]][location[1]] != 'X':

## maxval += check\_horizontal(board, location[0], 'max') # need row

## maxval += check\_vertical(board, location[1], 'max') # need column

## # NOTE diagonal check is splitted into left and right diagonal for convienence

## maxval += left\_diagonal(board, location[0], location[1], 'max')

## maxval += right\_diagonal(board, location[0], location[1], 'max')

## return maxval

## def min\_val(board, location):

## minval = 0

## if board[location[0]][location[1]] != 'O' and board[location[0]][location[1]] != 'X':

## minval -= check\_horizontal(board, location[0], 'min')

## minval -= check\_vertical(board, location[1], 'min')

## minval -= left\_diagonal(board, location[0], location[1], 'min')

## minval -= right\_diagonal(board, location[0], location[1], 'min')

## return minval

## def check\_horizontal(board, row, u\_type):

## opposed = 'X'

## sign = 'O'

## if u\_type == 'min':

## opposed = 'O'

## sign = 'X'

## v = 0

## unfilled = 0

## for i in range(3): # 3 == len(board)'s row

## if board[row][i] != opposed:

## unfilled += 1

## if board[row][i] == sign:

## v += 1

## if unfilled == 3:

## if v == 2:

## v = 10

## else:

## v += 1

## elif unfilled < 3:

## v = 0

## return v

## def check\_vertical(board, col, u\_type):

## opposed = 'X'

## sign = 'O'

## if u\_type == 'min':

## opposed = 'O'

## sign = 'X'

## v = 0

## unfilled = 0

## for i in range(3): # 3 == len(board)'s column

## if board[i][col] != opposed:

## unfilled += 1

## if board[i][col] == sign:

## v += 1

## if unfilled == 3:

## if v == 2:

## v = 10

## else:

## v += 1

## elif unfilled < 3:

## v = 0

## return v

## def left\_diagonal(board, row, col, u\_type): # NOTE top\_left to bottom\_right diagonal check

## opposed = 'X'

## sign = 'O'

## if u\_type == 'min':

## opposed = 'O'

## sign = 'X'

## v = 0

## unfilled = 0

## if row == col:

## for i in range(3):

## if board[i][i] != opposed:

## unfilled += 1

## if board[i][i] == sign:

## v += 1

## if unfilled == 3:

## if v == 2:

## v = 10

## else:

## v += 1

## elif unfilled < 3:

## v = 0

## return v

## def right\_diagonal(board, row, col, u\_type):

## opposed = 'X'

## sign = 'O'

## if u\_type == 'min':

## opposed = 'O'

## sign = 'X'

## v = 0

## unfilled = 0

## state = False

## for i in range(len(board)):

## if board[i][abs(i-2)] == board[row][col]:

## state = True

## if board[i][abs(i-2)] != opposed:

## unfilled += 1

## if board[i][abs(i-2)] == sign:

## v +=1

## if unfilled == 3 and state == True:

## if v == 2:

## v = 10

## else:

## v += 1

## elif unfilled < 3:

## v = 0

## return v

## def dispUboard(uboard):

## print('\n')

## count = 0

## print("Utility Board:\n")

## for i in range(len(uboard)):

## for j in range(len(uboard[i])):

## count += 1

## if uboard[i][j] == 'O' or uboard[i][j] == 'X':

## print(' ',uboard[i][j],end=' ')

## else:

## print(uboard[i][j],end=' ')

## if count%3 == 0:

## print('\n')

## def checkWin(board, sign):

## if checkHorizontal(board, sign) == True:

## return True

## if checkVertical(board, sign) == True:

## return True

## if checkDiagonal(board, sign) == True:

## return True

## return False

## def checkTie(board):

## filled = 0

## for i in range(len(board)):

## for j in range(len(board[i])):

## if board[i][j] == 'O' or board[i][j] == 'X':

## filled += 1

## if filled == 9:

## return True

## return False

## def checkDiagonal(board, sign):

## for i in range(len(board)):

## filled = 0

## if board[0][0] == sign:

## for j in range(len(board[i])):

## if board[j][j] == sign:

## filled += 1

## elif board[0][2] == sign:

## for j in range(len(board[i])):

## if board[0+j][2-j] == sign:

## filled += 1

## if filled == 3:

## return True

## return False

## def checkHorizontal(board, sign): # NOTE BUGGY so fix it

## for i in range(len(board)):

## if board[i][0] == sign:

## filled = 0

## for j in range(len(board[i])):

## if board[i][j] == sign:

## filled += 1

## if filled == 3:

## return True

## return False

## def checkVertical(board, sign):

## for i in range(len(board)):

## if board[0][i] == sign:

## filled = 0

## for j in range(len(board[i])):

## if board[j][i] == sign:

## filled += 1

## if filled == 3:

## return True

## return False

## def dispboard(board):

## print('\n')

## count = 0

## print('Tictactoe Board:\n')

## for i in range(len(board)):

## for j in range(len(board[i])):

## count += 1

## print(board[i][j],end=' ')

## if count%3 == 0:

## print('\n')

## def checkCompatible(board, move, sign):

## i = 2

## if move <= 2:

## i = 0

## elif move >= 3 and move <= 5:

## i = 1

## loc = [i,(move-(i\*3))]

## if board[loc[0]][loc[1]] == move:

## board[loc[0]][loc[1]] = sign

## return True

## else:

## print("Please select an empty spot and try again.")

## return False

## def computerDecision(board):

## while (checkTie(board) == False) and (checkWin(board, 'X') == False):

## uboard = generate\_cells(board)

## dispUboard(uboard)

## dispboard(board)

## # TODO run minimax algorithm here

## computer\_decision = minimax\_algorithm(uboard)

## computer\_decision = int(computer\_decision)

## if checkCompatible(board, computer\_decision, 'O') == True:

## if checkTie(board) == True:

## dispboard(board)

## play\_again = input("\nThis is a tie game, to play again enter any key, otherwise enter 'q' to quit.\nYour decision: ")

## if play\_again == 'q':

## return

## else:

## board = [[0, 1, 2],[3, 4, 5],[6, 7, 8]]

## GameInitializer(board)

## elif checkWin(board, 'O') == True:

## dispboard(board)

## print("The computer won!")

## return

## else:

## playerDecision(board)

## else:

## computerDecision(board)

## def playerDecision(board):

## while (checkTie(board) == False) and (checkWin(board, 'O') == False):

## dispboard(board)

## player\_decision = input("\n(The player's turn) Enter the empty position you want to place your 'X': ")

## player\_decision = int(player\_decision)

## if checkCompatible(board, player\_decision, 'X') == True:

## if checkTie(board) == True:

## dispboard(board)

## play\_again = input("\nThis is a tie game, if you want to play again enter 'p', to quit enter any key.\nYour decision: ")

## if play\_again == 'q':

## return

## else:

## board = [[0, 1, 2],[3, 4, 5],[6, 7, 8]]

## GameInitializer(board)

## elif checkWin(board, 'X') == True:

## dispboard(board)

## print("The player won!")

## return

## else:

## computerDecision(board)

## else:

## playerDecision(board)

## def GameInitializer(board):

## choice = input("\nDo you want to go first or the computer goes first?\nEnter 'c' for computer first, or 'p' if you would like to go first\nYour Choice: ")

## if choice == 'c':

## computerDecision(board)

## elif choice == 'p':

## playerDecision(board)

## else:

## print("\nPlease enter 'c' or 'p' and try again.")

## GameInitializer(board)

## def minimax\_algorithm(ub): # should return a pos, such as 4, not index[1,1]

## optimal = 0

## options = []

## redundant\_optimal = [] # This adds the random feature for the computer decision.

## for i in range(len(ub)):

## for j in range(len(ub[i])):

## if ub[i][j] != 'X' and ub[i][j] != 'O':

## # NOTE uboard[i][j's 0 is position, 1 is maxval, 2 is minval

## if ub[i][j][1] >= 10:

## return ub[i][j][0]

## elif ub[i][j][2] <= -10:

## return ub[i][j][0]

## else:

## if abs(ub[i][j][1]) == abs(ub[i][j][2]):

## # NOTE if abs of max = abs of min, add 1 to their sum. Why? because we want to win more more than limiting the enemy

## options.append([abs(ub[i][j][1]) + abs(ub[i][j][2])+1, ub[i][j][0]])

## else: # NOTE, [0] is the total val of abs(max + min). [1] is the index

## options.append([abs(ub[i][j][1]) + abs(ub[i][j][2]), ub[i][j][0]])

## optimal = max(options) # NOTE for redundant\_optimal, [0] is index, [1] is val

## for i in range(len(options)):

## if options[i][0] == optimal[0]:

## redundant\_optimal.append(options[i][1])

## redundant\_optimal.append(optimal[1])

## randnum = random.randint(0,len(redundant\_optimal)-1)

## return redundant\_optimal[randnum]

## # NOTE play game here

## init\_board = [[0, 1, 2],

## [3, 4, 5],

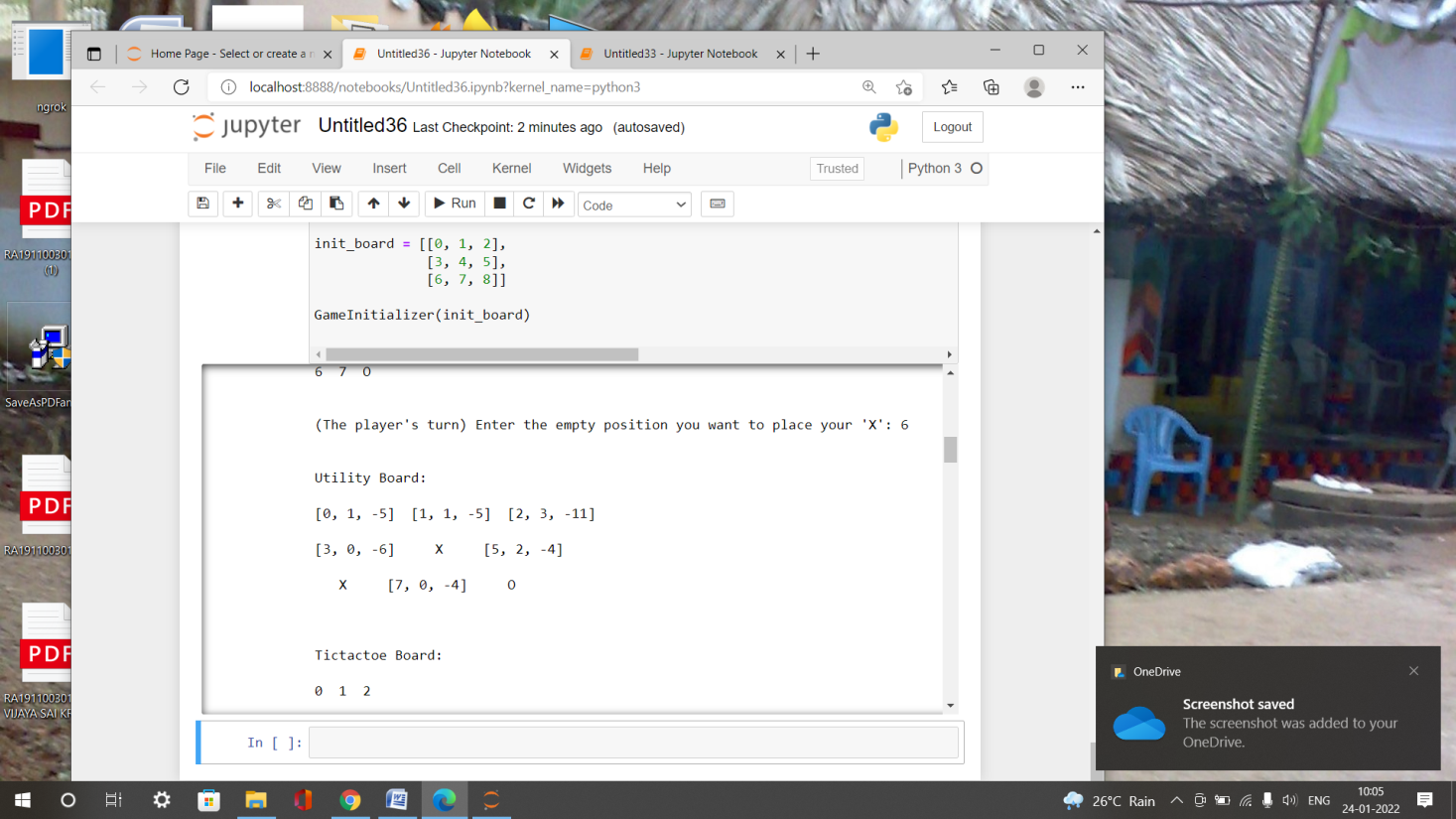
## [6, 7, 8]]

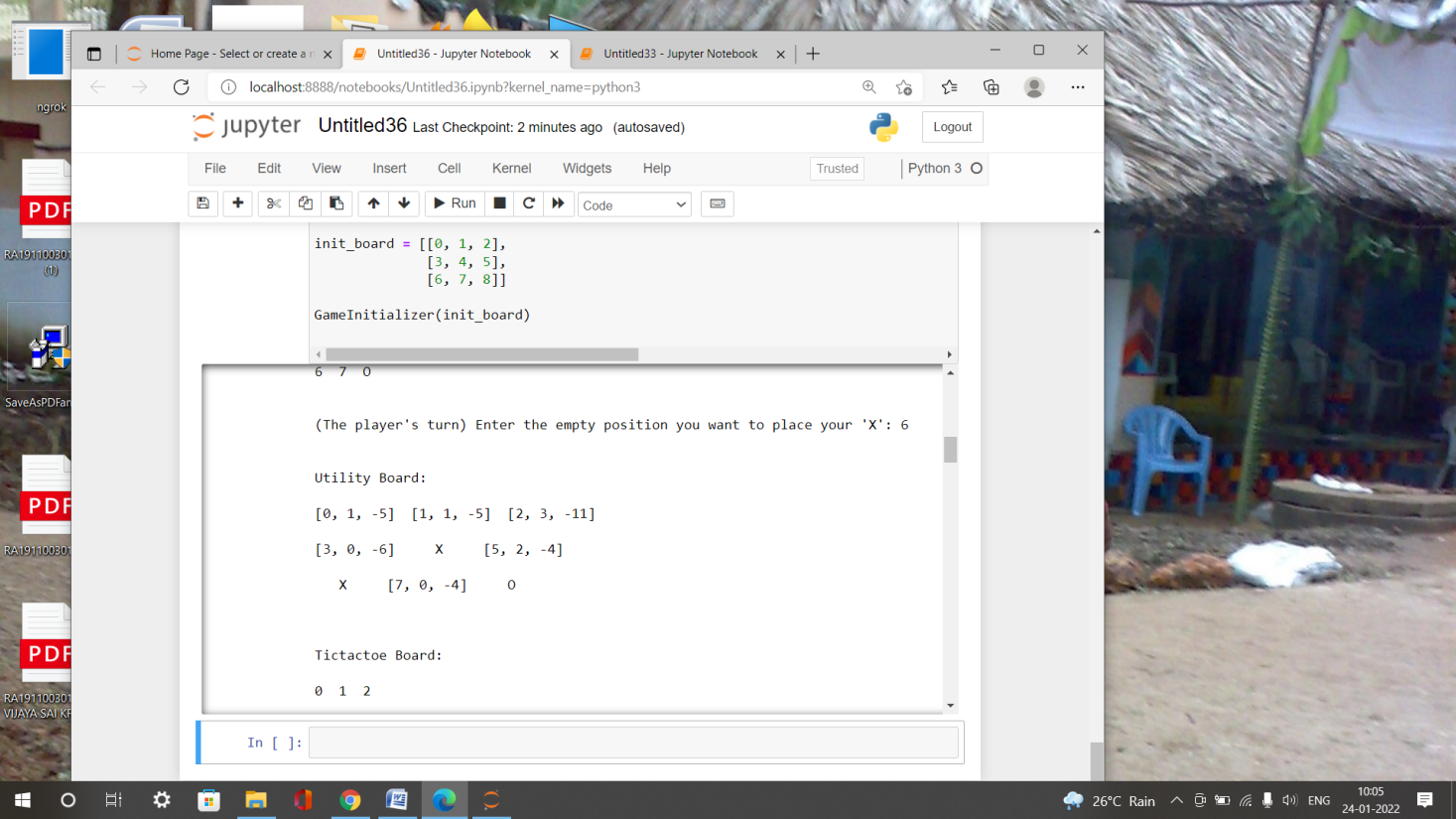
## GameInitializer(init\_board)

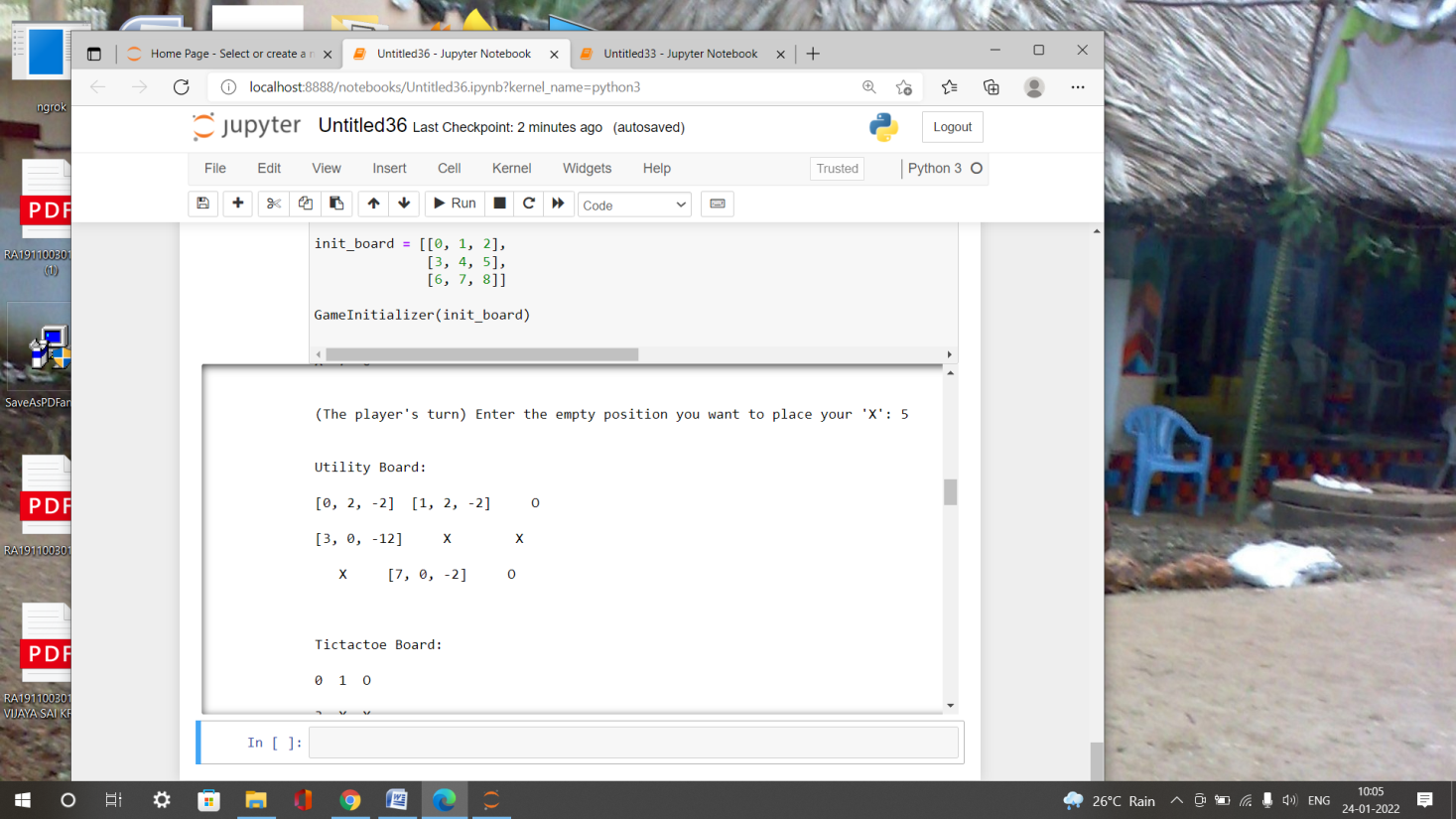
## Output screen shots :

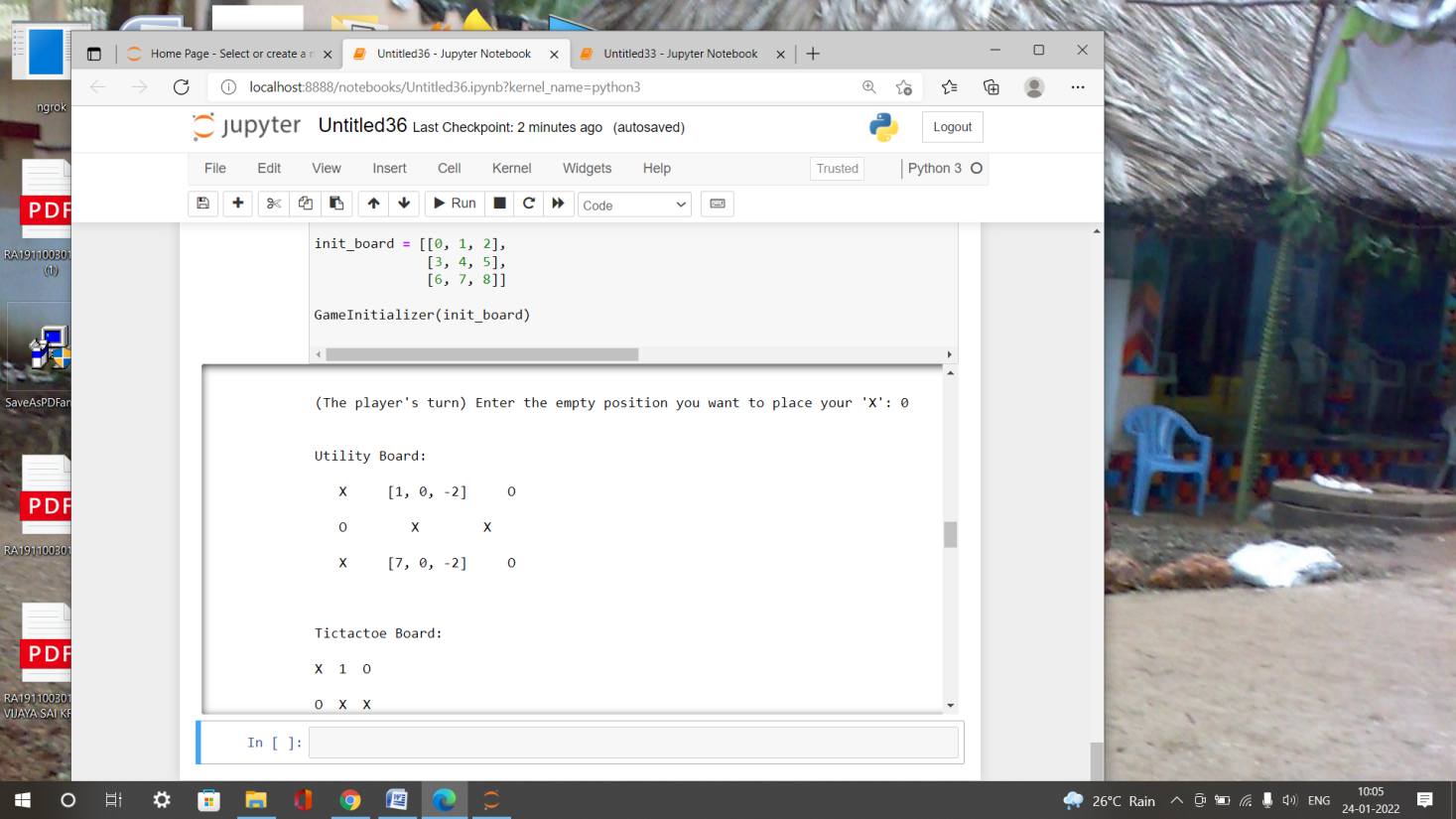
## 2022-01-24 (4).png

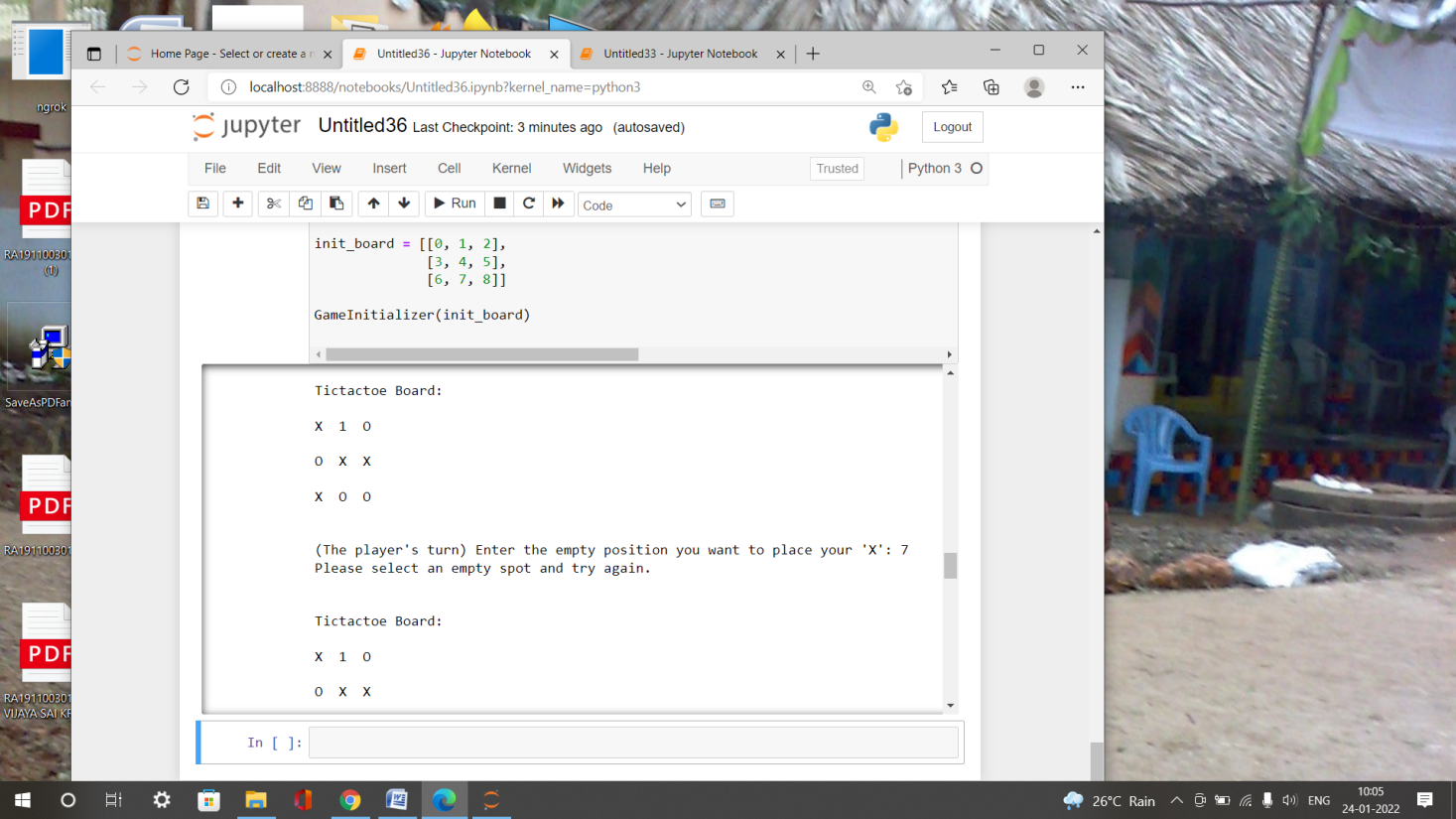
## 2022-01-24 (4).png

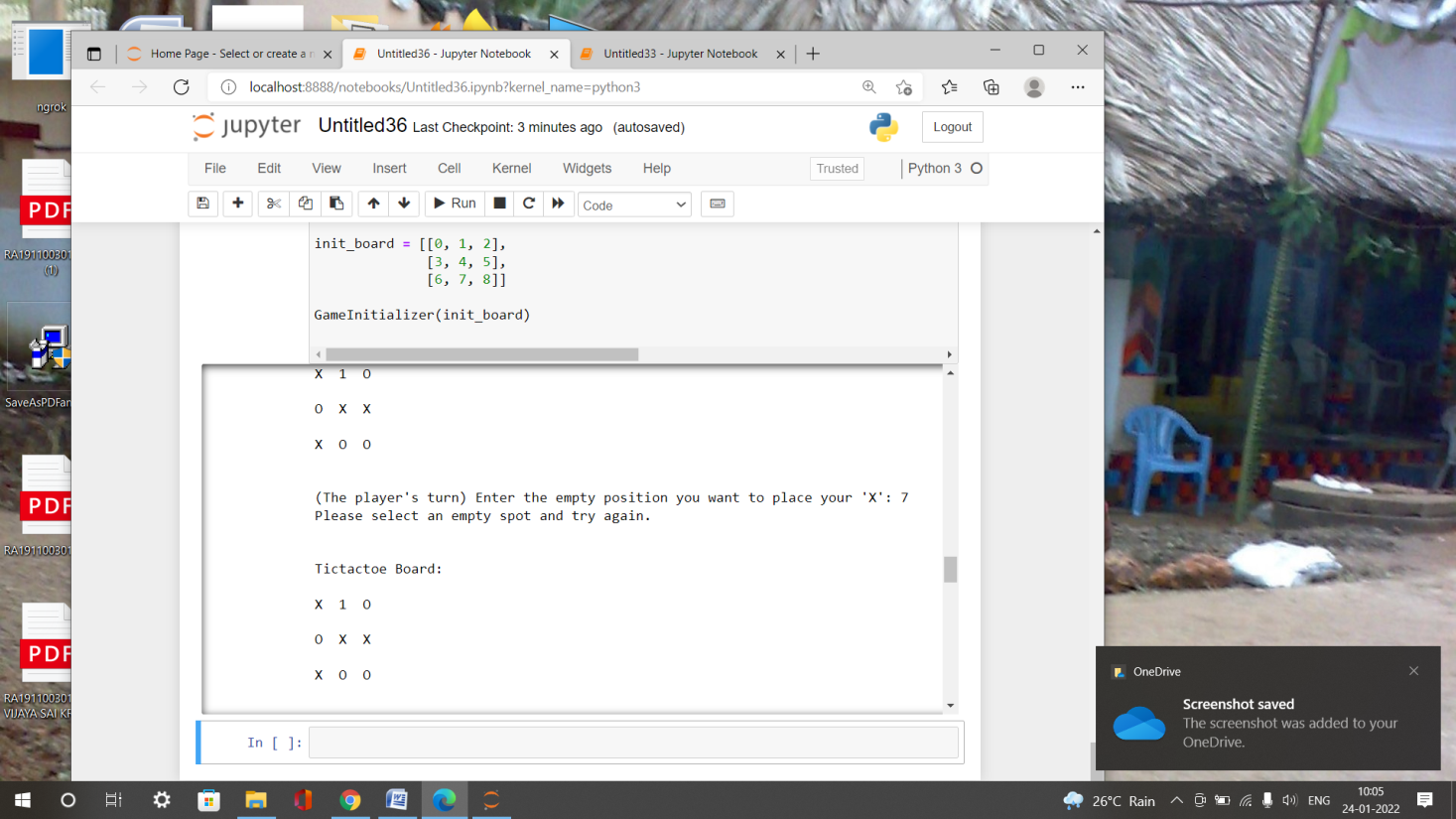
****

****

****

****

****

****

**Result :**  The Tic Tac Toe problem was implemented successfully using minmax algorithm to evaluate the best moves with the highest score.