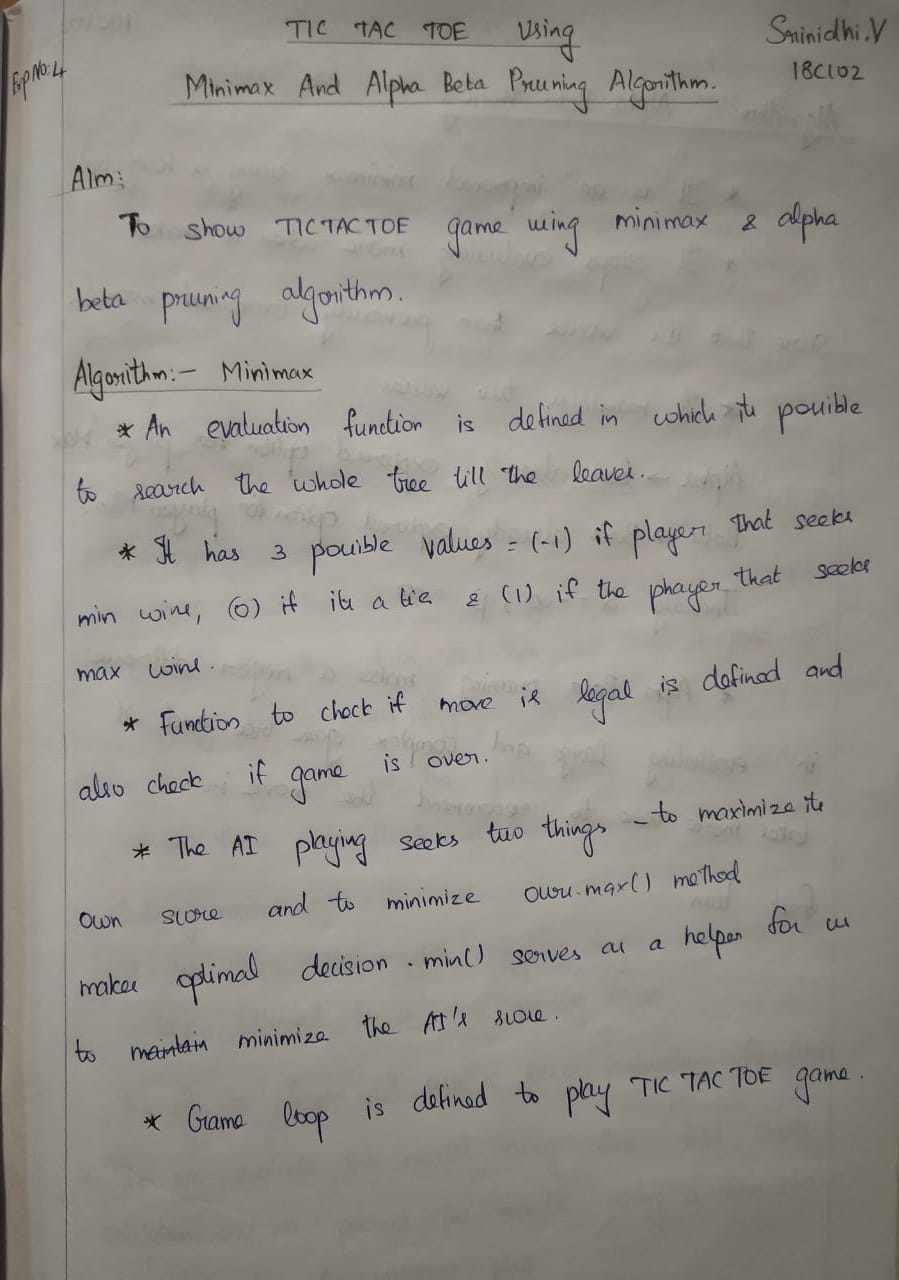
****

**Program:**

Minimax:

import time

class Game:

def \_\_init\_\_(self):

self.initialize\_game()

def initialize\_game(self):

self.current\_state = [['.','.','.'],

['.','.','.'],

['.','.','.']]

# Player X always plays first

self.player\_turn = 'X'

def draw\_board(self):

for i in range(0, 3):

for j in range(0, 3):

print('{}|'.format(self.current\_state[i][j]), end=" ")

print()

print()

# Determines if the made move is a legal move

def is\_valid(self, px, py):

if px < 0 or px > 2 or py < 0 or py > 2:

return False

elif self.current\_state[px][py] != '.':

return False

else:

return True

# Checks if the game has ended and returns the winner in each case

def is\_end(self):

# Vertical win

for i in range(0, 3):

if (self.current\_state[0][i] != '.' and

self.current\_state[0][i] == self.current\_state[1][i] and

self.current\_state[1][i] == self.current\_state[2][i]):

return self.current\_state[0][i]

# Horizontal win

for i in range(0, 3):

if (self.current\_state[i] == ['X', 'X', 'X']):

return 'X'

elif (self.current\_state[i] == ['O', 'O', 'O']):

return 'O'

# Main diagonal win

if (self.current\_state[0][0] != '.' and

self.current\_state[0][0] == self.current\_state[1][1] and

self.current\_state[0][0] == self.current\_state[2][2]):

return self.current\_state[0][0]

# Second diagonal win

if (self.current\_state[0][2] != '.' and

self.current\_state[0][2] == self.current\_state[1][1] and

self.current\_state[0][2] == self.current\_state[2][0]):

return self.current\_state[0][2]

# Is whole board full?

for i in range(0, 3):

for j in range(0, 3):

# There's an empty field, we continue the game

if (self.current\_state[i][j] == '.'):

return None

# It's a tie!

return '.'

# Player 'O' is max, in this case AI

def max(self):

# Possible values for maxv are:

# -1 - loss

# 0 - a tie

# 1 - win

# We're initially setting it to -2 as worse than the worst case:

maxv = -2

px = None

py = None

result = self.is\_end()

# If the game came to an end, the function needs to return

# the evaluation function of the end. That can be:

# -1 - loss

# 0 - a tie

# 1 - win

if result == 'X':

return (-1, 0, 0)

elif result == 'O':

return (1, 0, 0)

elif result == '.':

return (0, 0, 0)

for i in range(0, 3):

for j in range(0, 3):

if self.current\_state[i][j] == '.':

# On the empty field player 'O' makes a move and calls Min

# That's one branch of the game tree.

self.current\_state[i][j] = 'O'

(m, min\_i, min\_j) = self.min()

# Fixing the maxv value if needed

if m > maxv:

maxv = m

px = i

py = j

# Setting back the field to empty

self.current\_state[i][j] = '.'

return (maxv, px, py)

# Player 'X' is min, in this case human

def min(self):

# Possible values for minv are:

# -1 - win

# 0 - a tie

# 1 - loss

# We're initially setting it to 2 as worse than the worst case:

minv = 2

qx = None

qy = None

result = self.is\_end()

if result == 'X':

return (-1, 0, 0)

elif result == 'O':

return (1, 0, 0)

elif result == '.':

return (0, 0, 0)

for i in range(0, 3):

for j in range(0, 3):

if self.current\_state[i][j] == '.':

self.current\_state[i][j] = 'X'

(m, max\_i, max\_j) = self.max()

if m < minv:

minv = m

qx = i

qy = j

self.current\_state[i][j] = '.'

return (minv, qx, qy)

def play(self):

while True:

self.draw\_board()

self.result = self.is\_end()

# Printing the appropriate message if the game has ended

if self.result != None:

if self.result == 'X':

print('The winner is X!')

elif self.result == 'O':

print('The winner is O!')

elif self.result == '.':

print("It's a tie!")

self.initialize\_game()

return

# If it's player's turn

if self.player\_turn == 'X':

while True:

start = time.time()

(m, qx, qy) = self.min()

end = time.time()

print('Evaluation time: {}s'.format(round(end - start, 7)))

print('Recommended move: X = {}, Y = {}'.format(qx, qy))

px = int(input('Insert the X coordinate: '))

py = int(input('Insert the Y coordinate: '))

(qx, qy) = (px, py)

if self.is\_valid(px, py):

self.current\_state[px][py] = 'X'

self.player\_turn = 'O'

break

else:

print('The move is not valid! Try again.')

# If it's AI's turn

else:

(m, px, py) = self.max()

self.current\_state[px][py] = 'O'

self.player\_turn = 'X'

def main():

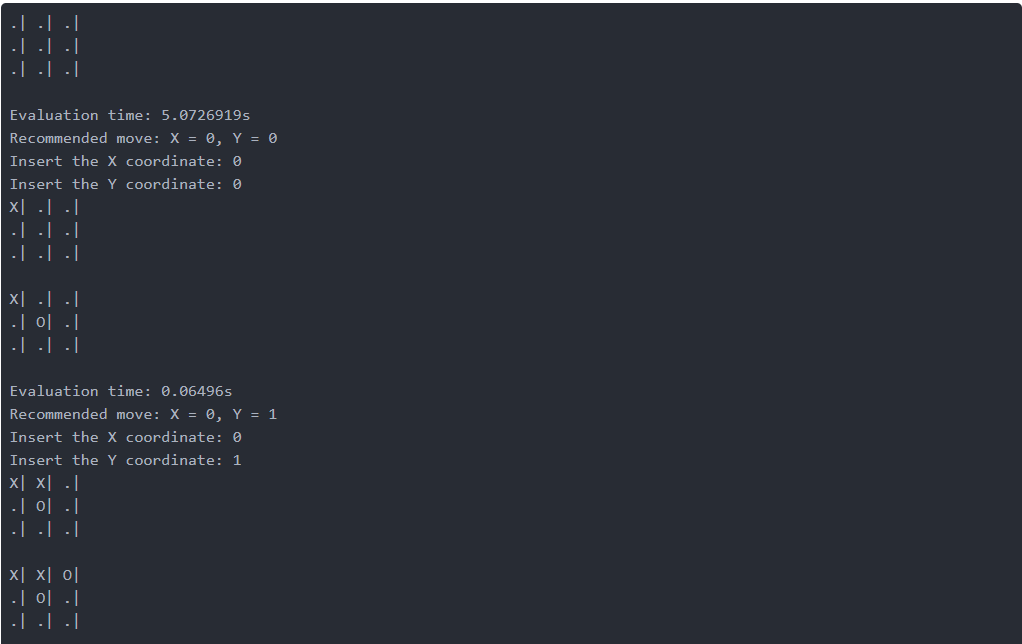
g = Game()

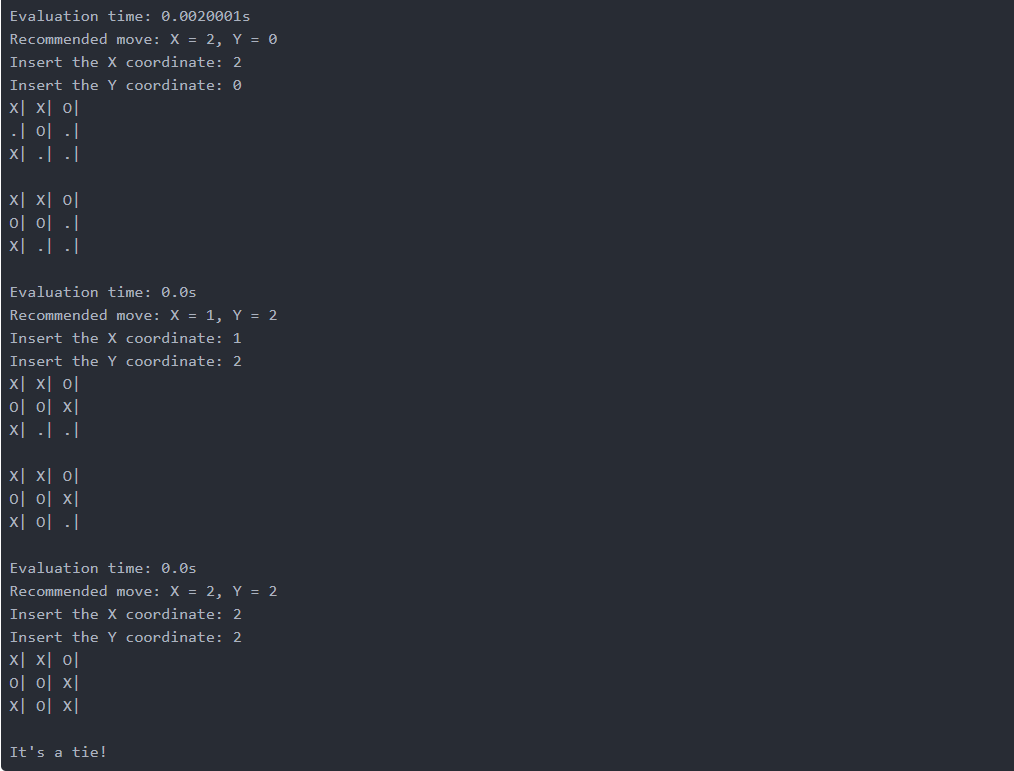
g.play()

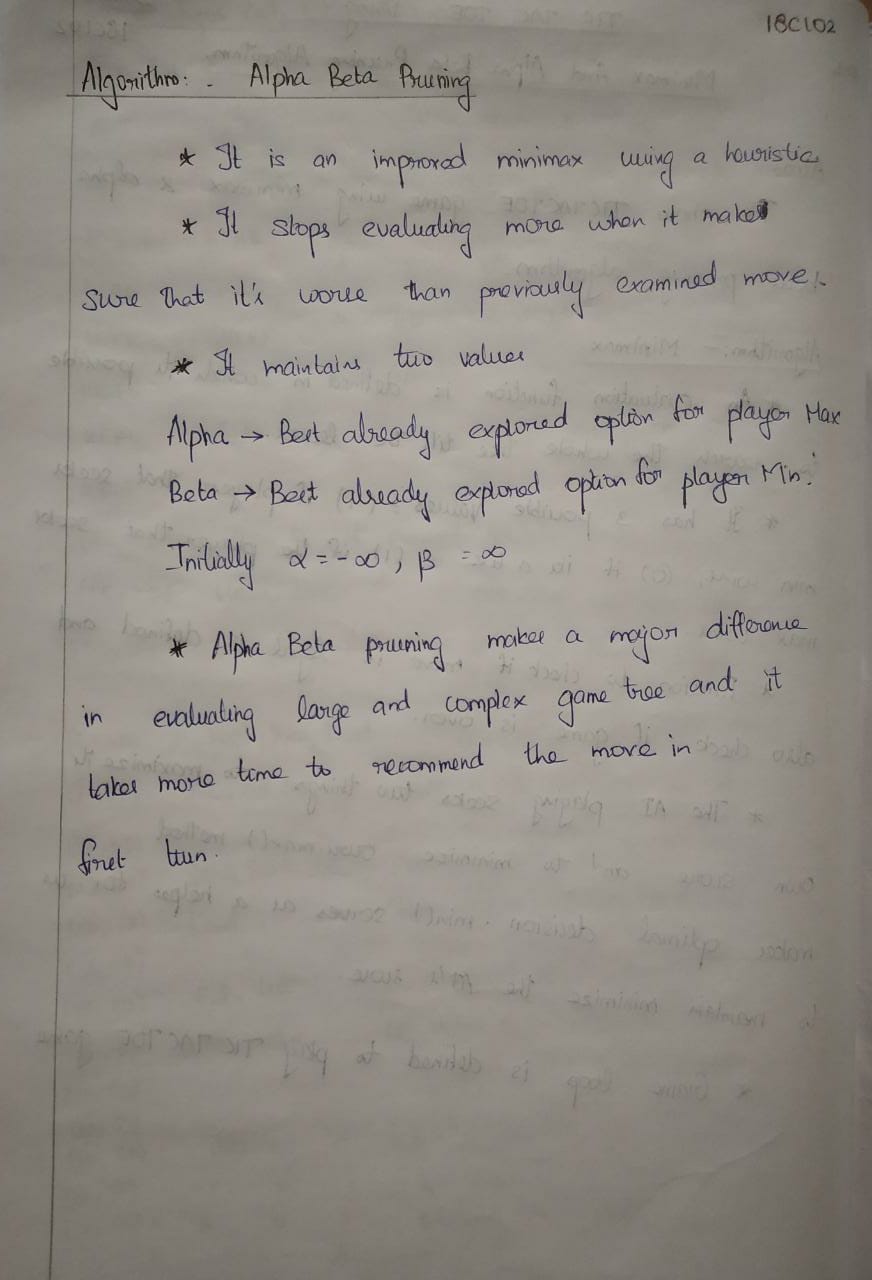
if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**







Alpha-Beta Pruning:

import time

class Game:

def \_\_init\_\_(self):

self.initialize\_game()

def initialize\_game(self):

self.current\_state = [['.','.','.'],

['.','.','.'],

['.','.','.']]

# Player X always plays first

self.player\_turn = 'X'

def draw\_board(self):

for i in range(0, 3):

for j in range(0, 3):

print('{}|'.format(self.current\_state[i][j]), end=" ")

print()

print()

# Determines if the made move is a legal move

def is\_valid(self, px, py):

if px < 0 or px > 2 or py < 0 or py > 2:

return False

elif self.current\_state[px][py] != '.':

return False

else:

return True

# Checks if the game has ended and returns the winner in each case

def is\_end(self):

# Vertical win

for i in range(0, 3):

if (self.current\_state[0][i] != '.' and

self.current\_state[0][i] == self.current\_state[1][i] and

self.current\_state[1][i] == self.current\_state[2][i]):

return self.current\_state[0][i]

# Horizontal win

for i in range(0, 3):

if (self.current\_state[i] == ['X', 'X', 'X']):

return 'X'

elif (self.current\_state[i] == ['O', 'O', 'O']):

return 'O'

# Main diagonal win

if (self.current\_state[0][0] != '.' and

self.current\_state[0][0] == self.current\_state[1][1] and

self.current\_state[0][0] == self.current\_state[2][2]):

return self.current\_state[0][0]

# Second diagonal win

if (self.current\_state[0][2] != '.' and

self.current\_state[0][2] == self.current\_state[1][1] and

self.current\_state[0][2] == self.current\_state[2][0]):

return self.current\_state[0][2]

# Is whole board full?

for i in range(0, 3):

for j in range(0, 3):

# There's an empty field, we continue the game

if (self.current\_state[i][j] == '.'):

return None

# It's a tie!

return '.'

def max\_alpha\_beta(self, alpha, beta):

maxv = -2

px = None

py = None

result = self.is\_end()

if result == 'X':

return (-1, 0, 0)

elif result == 'O':

return (1, 0, 0)

elif result == '.':

return (0, 0, 0)

for i in range(0, 3):

for j in range(0, 3):

if self.current\_state[i][j] == '.':

self.current\_state[i][j] = 'O'

(m, min\_i, in\_j) = self.min\_alpha\_beta(alpha, beta)

if m > maxv:

maxv = m

px = i

py = j

self.current\_state[i][j] = '.'

# Next two ifs in Max and Min are the only difference between regular algorithm and minimax

if maxv >= beta:

return (maxv, px, py)

if maxv > alpha:

alpha = maxv

return (maxv, px, py)

def min\_alpha\_beta(self, alpha, beta):

minv = 2

qx = None

qy = None

result = self.is\_end()

if result == 'X':

return (-1, 0, 0)

elif result == 'O':

return (1, 0, 0)

elif result == '.':

return (0, 0, 0)

for i in range(0, 3):

for j in range(0, 3):

if self.current\_state[i][j] == '.':

self.current\_state[i][j] = 'X'

(m, max\_i, max\_j) = self.max\_alpha\_beta(alpha, beta)

if m < minv:

minv = m

qx = i

qy = j

self.current\_state[i][j] = '.'

if minv <= alpha:

return (minv, qx, qy)

if minv < beta:

beta = minv

return (minv, qx, qy)

def play\_alpha\_beta(self):

while True:

self.draw\_board()

self.result = self.is\_end()

if self.result != None:

if self.result == 'X':

print('The winner is X!')

elif self.result == 'O':

print('The winner is O!')

elif self.result == '.':

print("It's a tie!")

self.initialize\_game()

return

if self.player\_turn == 'X':

while True:

start = time.time()

(m, qx, qy) = self.min\_alpha\_beta(-2, 2)

end = time.time()

print('Evaluation time: {}s'.format(round(end - start, 7)))

print('Recommended move: X = {}, Y = {}'.format(qx, qy))

px = int(input('Insert the X coordinate: '))

py = int(input('Insert the Y coordinate: '))

qx = px

qy = py

if self.is\_valid(px, py):

self.current\_state[px][py] = 'X'

self.player\_turn = 'O'

break

else:

print('The move is not valid! Try again.')

else:

(m, px, py) = self.max\_alpha\_beta(-2, 2)

self.current\_state[px][py] = 'O'

self.player\_turn = 'X'

def main():

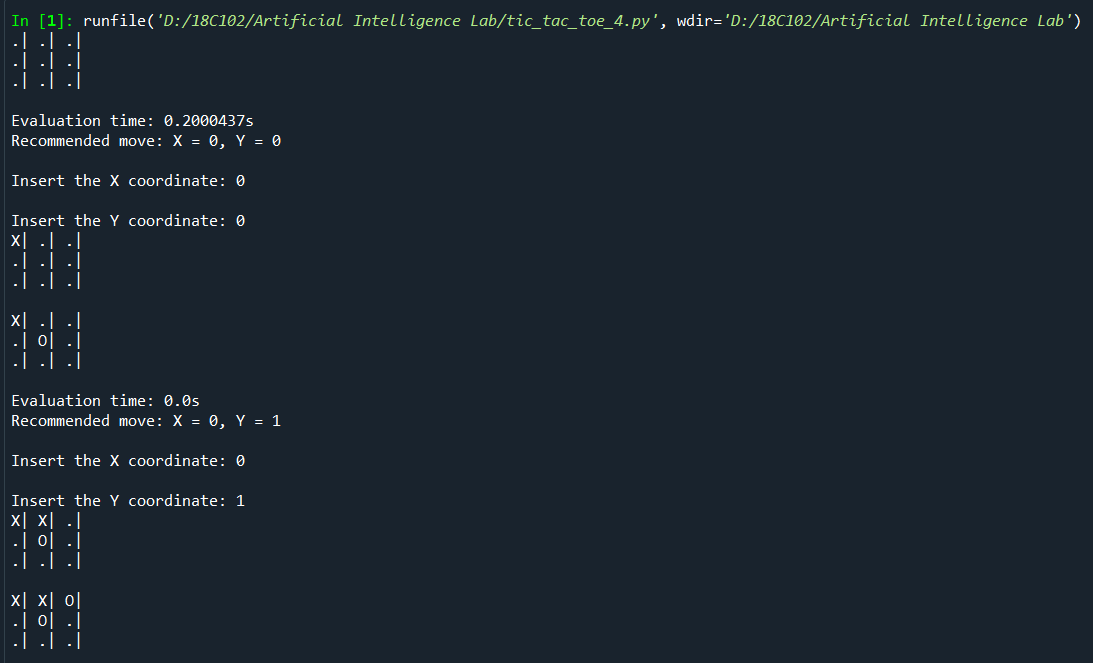
g = Game()

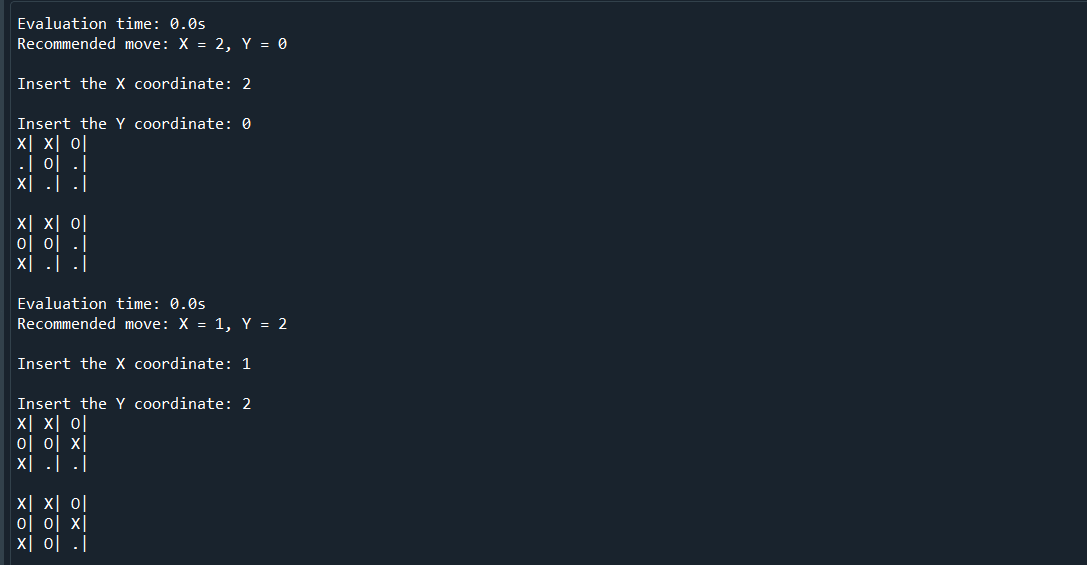
g.play\_alpha\_beta()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**







**Result:**

Thus, Tic Tac Toe is implemented using minimax algorithm and Alpha beta pruning.