Lab10

Code for tic tac toe

import math

from copy import deepcopy

# Define the Tic-Tac-Toe board size and players

EMPTY = "-"

PLAYER\_X = "X"  # Maximizing player (Computer)

PLAYER\_O = "O"  # Minimizing player (User)

# Helper functions

def is\_terminal(board):

    """Checks if the game has ended."""

    winner = get\_winner(board)

    if winner or not any(EMPTY in row for row in board):

        return True

    return False

def get\_winner(board):

    """Checks for a winner on the board."""

    # Check rows and columns

    for i in range(3):

        if board[i][0] == board[i][1] == board[i][2] != EMPTY:

            return board[i][0]

        if board[0][i] == board[1][i] == board[2][i] != EMPTY:

            return board[0][i]

    # Check diagonals

    if board[0][0] == board[1][1] == board[2][2] != EMPTY:

        return board[0][0]

    if board[0][2] == board[1][1] == board[2][0] != EMPTY:

        return board[0][2]

    return None

def utility(board):

    """Returns the utility of a terminal state."""

    winner = get\_winner(board)

    if winner == PLAYER\_X:

        return 1

    elif winner == PLAYER\_O:

        return -1

    return 0

def get\_actions(board):

    """Returns a list of possible moves."""

    actions = []

    for i in range(3):

        for j in range(3):

            if board[i][j] == EMPTY:

                actions.append((i, j))

    return actions

def result(board, action, player):

    """Returns the board resulting from applying an action."""

    new\_board = deepcopy(board)

    new\_board[action[0]][action[1]] = player

    return new\_board

# Alpha-Beta Search

def alpha\_beta\_search(board):

    """Performs Alpha-Beta Pruning to find the best action."""

    alpha = -math.inf

    beta = math.inf

    best\_action = None

    def max\_value(state, alpha, beta):

        if is\_terminal(state):

            return utility(state)

        v = -math.inf

        for action in get\_actions(state):

            v = max(v, min\_value(result(state, action, PLAYER\_X), alpha, beta))

            if v >= beta:

                return v

            alpha = max(alpha, v)

        return v

    def min\_value(state, alpha, beta):

        if is\_terminal(state):

            return utility(state)

        v = math.inf

        for action in get\_actions(state):

            v = min(v, max\_value(result(state, action, PLAYER\_O), alpha, beta))

            if v <= alpha:

                return v

            beta = min(beta, v)

        return v

    for action in get\_actions(board):

        value = min\_value(result(board, action, PLAYER\_X), alpha, beta)

        if value > alpha:

            alpha = value

            best\_action = action

    return best\_action

# Game loop

def print\_board(board):

    """Displays the board."""

    for row in board:

        print(" | ".join(row))

    print()

def play\_game():

    """Runs the Tic-Tac-Toe game with user input."""

    board = [[EMPTY for \_ in range(3)] for \_ in range(3)]

    print("Welcome to Tic-Tac-Toe!")

    print("You are 'O', and the computer is 'X'.")

    print\_board(board)

    while not is\_terminal(board):

        # User's turn

        user\_move = None

        while user\_move not in get\_actions(board):

            try:

                print("Your turn! Enter your move as 'row col' (e.g., '1 2'):")

                row, col = map(int, input().split())

                user\_move = (row - 1, col - 1)  # Convert to 0-based index

                if user\_move not in get\_actions(board):

                    print("Invalid move! Try again.")

            except ValueError:

                print("Invalid input! Please enter two numbers separated by a space.")

        board = result(board, user\_move, PLAYER\_O)

        print("You played:")

        print\_board(board)

        if is\_terminal(board):

            break

        # Computer's turn

        print("Computer's turn...")

        computer\_move = alpha\_beta\_search(board)

        board = result(board, computer\_move, PLAYER\_X)

        print("Computer played:")

        print\_board(board)

    # Game over

    winner = get\_winner(board)

    if winner == PLAYER\_X:

        print("Computer wins!")

    elif winner == PLAYER\_O:

        print("Congratulations! You win!")

    else:

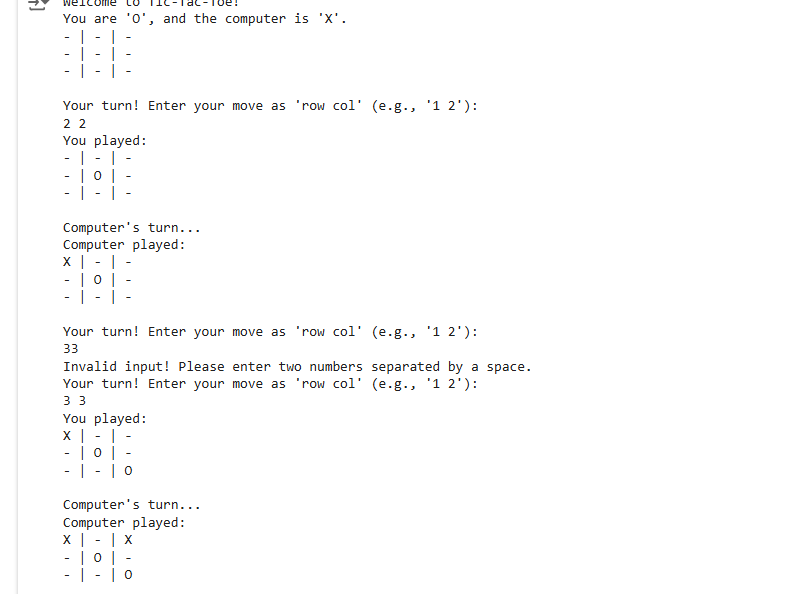
        print("It's a draw!")

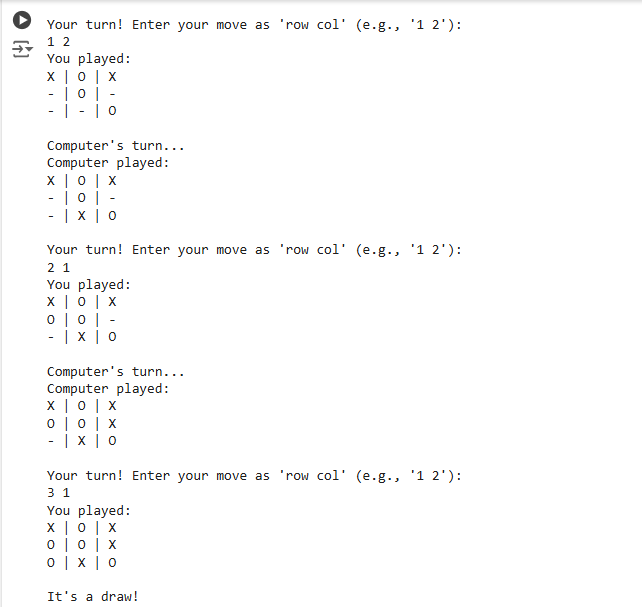
# Run the game

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

    play\_game()

ouput:-





code for 8 queens

import math

def is\_terminal(state, n):

    """Check if the board is a valid solution (no conflicts, all queens placed)."""

    return len(state) == n

def count\_conflicts(state):

    """Calculate the number of conflicts between queens."""

    conflicts = 0

    for i in range(len(state)):

        for j in range(i + 1, len(state)):

            # Same column or diagonal conflicts

            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):

                conflicts += 1

    return conflicts

def utility(state):

    """Return a utility score based on the number of conflicts (higher is better)."""

    return -count\_conflicts(state)

def actions(state, n):

    """Generate possible next moves."""

    next\_row = len(state)

    if next\_row >= n:

        return []

    return [state + [col] for col in range(n)]

def max\_value(state, alpha, beta, n):

    """Maximizing function."""

    if is\_terminal(state, n):

        return utility(state)

    v = -math.inf

    for action in actions(state, n):

        v = max(v, min\_value(action, alpha, beta, n))

        if v >= beta:

            return v

        alpha = max(alpha, v)

    return v

def min\_value(state, alpha, beta, n):

    """Minimizing function."""

    if is\_terminal(state, n):

        return utility(state)

    v = math.inf

    for action in actions(state, n):

        v = min(v, max\_value(action, alpha, beta, n))

        if v <= alpha:

            return v

        beta = min(beta, v)

    return v

def alpha\_beta\_search(n):

    """Perform Alpha-Beta pruning to solve the N-Queens problem."""

    alpha = -math.inf

    beta = math.inf

    best\_action = None

    initial\_state = []

    for action in actions(initial\_state, n):

        value = min\_value(action, alpha, beta, n)

        if value > alpha:

            alpha = value

            best\_action = action

    return best\_action

# Example usage

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

    n = 8  # Number of queens

    solution = alpha\_beta\_search(n)

    if solution:

        print("Solution:", solution)

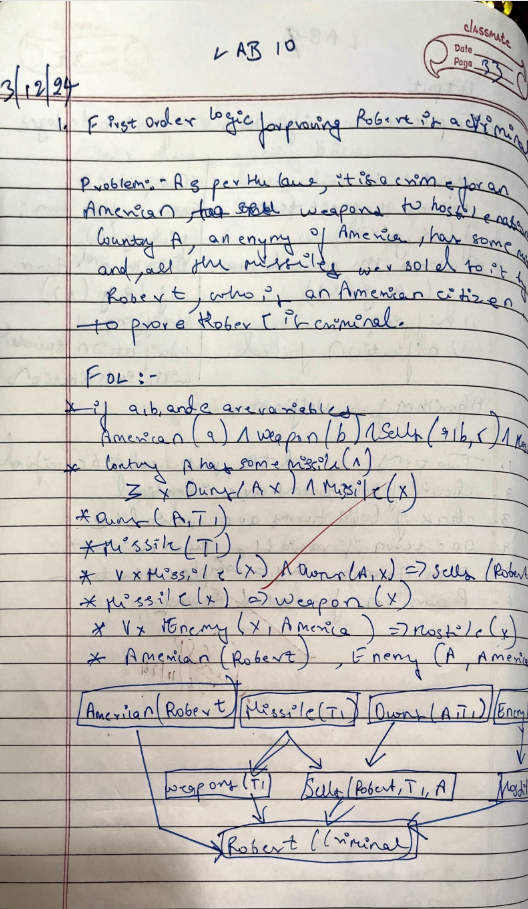
    else:

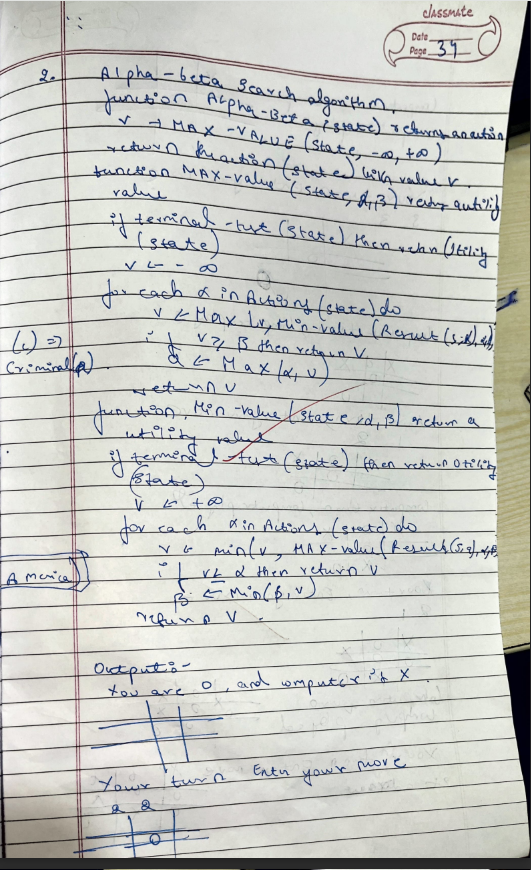
        print("No solution found.")

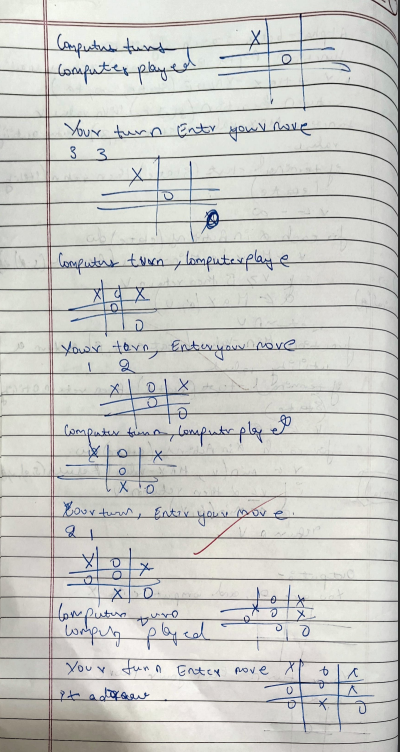
output:-

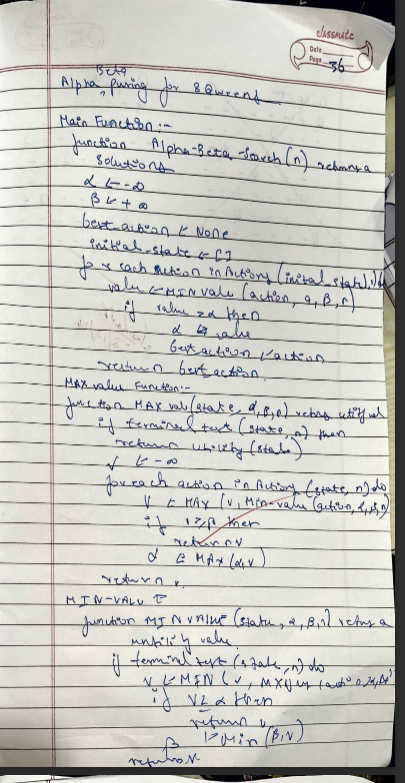
Solution: [0, 4, 7, 5, 2, 6, 1, 3]

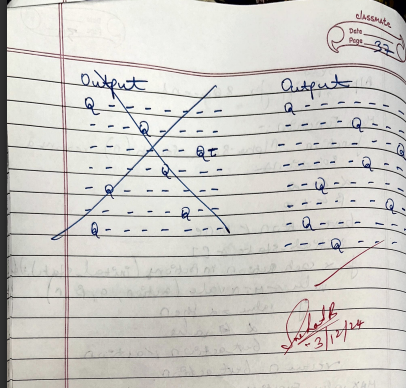
Observation book :-





****

****

****