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:-

```
Welcome to Tic-Tac-Toe!
You are 'O', and the computer is 'X'.
- | - | -
- | - | -
- | - | -

Your turn! Enter your move as 'row col' (e.g., '1 2'):
2 2
You played:
- | - | -
- | O | -
- | - | -

Computer's turn...
Computer played:
X | - | -
- | O | -
- | - | -

Your turn! Enter your move as 'row col' (e.g., '1 2'):
33
Invalid input! Please enter two numbers separated by a space.
Your turn! Enter your move as 'row col' (e.g., '1 2'):
3 3
You played:
X | - | -
- | O | -
- | - | O

Computer's turn...
Computer played:
X | - | X
- | O | -
- | - | O
```

```
Your turn! Enter your move as 'row col' (e.g., '1 2'):
1 2
You played:
X | O | X
- | O | -
- | - | O

Computer's turn...
Computer played:
X | O | X
- | O | -
- | X | O

Your turn! Enter your move as 'row col' (e.g., '1 2'):
2 1
You played:
X | O | X
O | O | -
- | X | O

Computer's turn...
Computer played:
X | O | X
O | O | X
- | X | O

Your turn! Enter your move as 'row col' (e.g., '1 2'):
3 1
You played:
X | O | X
O | O | X
O | X | O

It's a draw!
```

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 :-

LAB-10 :

* Solve using Forward chaining :-

Consider the following problem : As per the law, it is a crime for an american to sell weapons to hostile nations country A, an enemy of America, has some missiles and all the missiles were sold to it by Robert, who is an American citizen."
Prove that "Robert is criminal".

→ let's say a, b and c variables

American(a) ∧ weapon(b) ∧ sells(a, b, c) ∧ Hostile(r) → criminal(p)

∃ x (own(A,x) ∧ missile(x))

∀ x (missile(x) ∧ owns(A, x)) ⇒ sells(Robert, x, A)

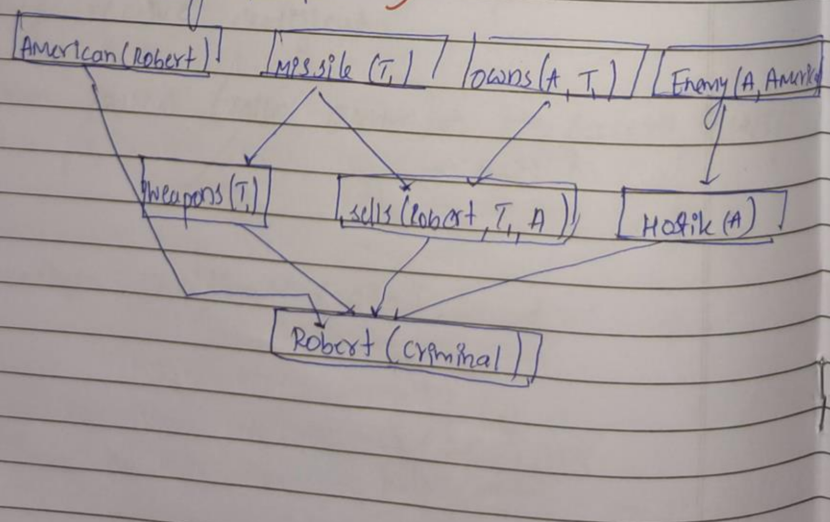
Missile(b) → weapon(b)

∀ x (Enemy(x, America) ⇒ Hostile(x))

American(Robert) (Robert is an American)

Enemy(A, America).

* Forward chaining proof :

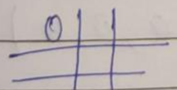


2. Alpha - beta search algorithm :-

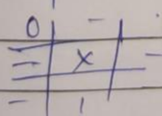
```
function Alpha-beta-search (state) returns an action
    v ← Max.value (state, -∞, +∞)
    return the action in Actions (state) with value v
function Max_value (state, α, β) returns a utility value
  if Terminal-Test (state) then return Utility (state)
  v ← -∞
  for each a in Actions (state) do
      v ← MAX (v, Min_-value (Result (s, a), α, β))
       if v ≥ β then return v
       α ← Max(α, v)
      return v
function Min_value (state, α, β) returns a utility
      value
 if Terminal-Test (state) then return Utility (state)
      v ← + ∞
  For each a in Actions (state) do
     v ← Min (v, MAX_value (Result (s, a), α, β))
       if v ≤ α then return v
       β ← Min (β, v)
      return v
```

* **Output:**

user move (1 1)



computer's move



* users move (2,1)

O		
O	X	

* computer's move (3,3)

O		
O	X	
		X

* user's move (3,1)

O		
O	X	
O		X

computer wins :

* Alpha beta purning for 8 queens :

* main function :—

function Alpha - Beta - search (n) returns a solutions
$\alpha \longleftarrow -\infty$
$\beta \longleftarrow +\infty$
best_action ⟵ none
initial _ state ← [ ]
for each section in Actions (initial_state)
value ← min_value (action, α, β, n)
if value > α then
α ← value
best_action ← action
return best _ action

MAX value Function :—
Function MAX val (state, α, β, n) returns uility

DATE: 35 PAGE:

if terminal test (state, n) then
    return utility (state)
        v ← −∞
for each action in Actions (state, n) do
    v ← MAX (v, Min_value (action, α, β, n)
    if l ≥ β then
        return v
    α ← MAX (α, v)
  return n

MIN - value

function MIN value (state, α, β, n) returns a value.

if terminal test (state, n) do
  v ← Min (v, MXMVS, action (α, β, n)
    if v ≤ α then
        return v
    β > min (β, v)
  return v

* output :

Q _ _ _ _ _ _ _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ Q _
_ Q _ _ _ _ _ _
_ _ _ Q _ _ _ _

3/12/24