NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

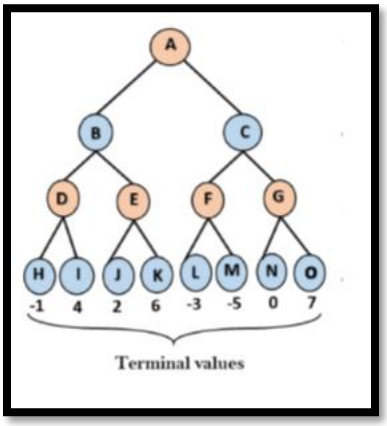
ARTIFICIAL INTELLIGENCE LAB

NAME	Ayesha Imran
Class	CS-A
Lab	07
Course	Artificial Intelligence
Date	30-October-25
Submitted To	Lec. Ijlal Haider

IN LAB TASKS

TASK 01:

Given the graph, where A is the root node and each other node is either an internal node with children or a leaf node with a numerical value. The goal is to use the Minimax algorithm to determine the minimax score for each node, starting from the root and evaluating each intermediate node based on whether it maximizes or minimizes the score. Print the Minimax values for all nodes in the game tree, showing the optimal score that each node can achieve.



CODE:

```
# Minimax calculation for the given tree

# Terminal (leaf) node values
values = {
    'H': -1, 'I': 4,
    'J': 2, 'K': 6,
    'L': -3, 'M': -5,
    'N': 0,  'O': 7
}

# Level 2: Max nodes (D, E, F, G)
D = max(values['H'], values['I'])
```

```
E = max(values['J'], values['K'])
F = max(values['L'], values['M'])
G = max(values['N'], values['O'])

# Level 1: Min nodes (B, C)
B = min(D, E)
C = min(F, G)

# Root: Max node (A)
A = max(B, C)

# Print all minimax values
print("Minimax values for all nodes:")
print(f"H = {values['H']}, I = {values['I']}, J = {values['J']}, K =
{values['K']}, "
      f"L = {values['L']}, M = {values['M']}, N = {values['N']}, O =
{values['O']}")
print(f"D = {D}, E = {E}, F = {F}, G = {G}")
print(f"B = {B}, C = {C}")
print(f"A = {A}")

print("\nFinal Minimax Value at Root (A):", A)
```

OUTPUT:

```
H = -1, I = 4, J = 2, K = 6, L = -3, M = -5, N = 0, O
= 7
D = 4, E = 6, F = -3, G = 7
B = 4, C = -3
A = 4

Final Minimax Value at Root (A): 4
PS D:\Python practice>
```

TASK 02:

Implement Alpha-Beta Pruning on the graph in Task 1.

CODE:

```
# Simple Alpha-Beta Pruning implementation for the given tree
```

```python
# Terminal values (leaves)
values = {
    'H': -1, 'I': 4,
    'J': 2, 'K': 6,
    'L': -3, 'M': -5,
    'N': 0,  'O': 7
}

# Define tree structure as adjacency list
tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H', 'I'],
    'E': ['J', 'K'],
    'F': ['L', 'M'],
    'G': ['N', 'O']
}

# Recursive function for Alpha-Beta Pruning
def alpha_beta(node, alpha, beta, isMax):
    # If node is a leaf, return its value
    if node in values:
        return values[node]

    if isMax:
        maxEval = float('-inf')
        for child in tree[node]:
            eval = alpha_beta(child, alpha, beta, False)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                print(f"Pruning at node {node} (Max) because beta <= alpha")
                break
        return maxEval
    else:
        minEval = float('inf')
        for child in tree[node]:
            eval = alpha_beta(child, alpha, beta, True)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                print(f"Pruning at node {node} (Min) because beta <= alpha")
                break
        return minEval
```

```python
# Run Alpha-Beta Pruning from the root
final_value = alpha_beta('A', float('-inf'), float('inf'), True)

print("\nFinal Minimax Value (with Alpha-Beta Pruning) at Root (A):",
final_value)
```

OUTPUT:

```
Pruning at node E (Max) because beta <= alpha
Pruning at node C (Min) because beta <= alpha

Final Minimax Value (with Alpha-Beta Pruning) at Root
(A): 4
PS D:\Python practice>
```

POST LAB TASKS

TASK 01:

Implement a Tic-Tac-Toe game where the player competes against an AI opponent. The AI will use the Minimax algorithm to determine its optimal moves. Your program should allow players to make their moves and display the current state of the board after each move, including checking for win conditions or draw scenarios.

- ❖ Create a 3x3 Game Board to represent the Tic-Tac-Toe board. Initialize it with empty spaces.
- ❖ Allow the user to select a position on the board for their move.
- ❖ Create a function that allows the AI to evaluate the best move based on the current board state.
- ❖ Check after each move to see if a player has won or if the game has ended in a draw.

❖ Print the board and game messages to indicate the status of the game after each move.

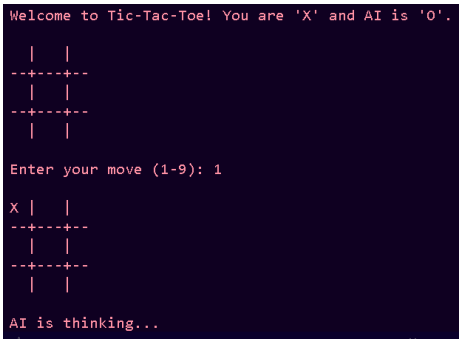
CODE:

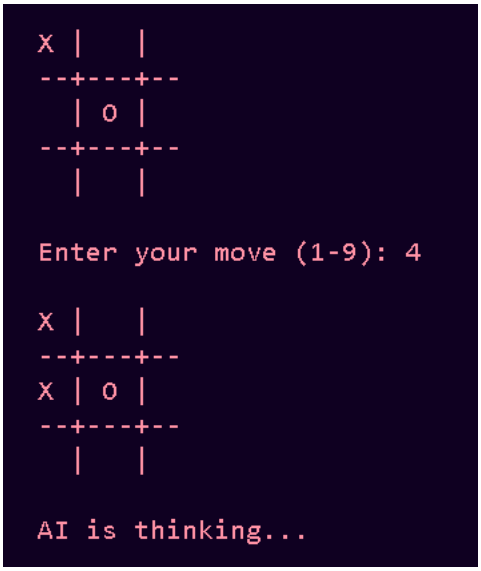
```
import math

# Initialize empty 3x3 board
board = [" " for _ in range(9)]

# Function to print the current state of the board
def print_board(board):
    print()
    print(f"{board[0]} | {board[1]} | {board[2]}")
    print("--+---+--")
    print(f"{board[3]} | {board[4]} | {board[5]}")
    print("--+---+--")
    print(f"{board[6]} | {board[7]} | {board[8]}")
    print()

# Function to check if a player has won
def check_winner(b, player):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # cols
        [0, 4, 8], [2, 4, 6]              # diagonals
    ]
    for combo in win_conditions:
        if b[combo[0]] == b[combo[1]] == b[combo[2]] == player:
            return True
    return False

# Function to check if board is full (draw)
def is_draw(board):
    return " " not in board

# Function to get available moves
def get_available_moves(board):
    return [i for i, spot in enumerate(board) if spot == " "]

# Minimax algorithm
def minimax(board, depth, is_maximizing):
    # Base cases
    if check_winner(board, "O"):  # AI win
```

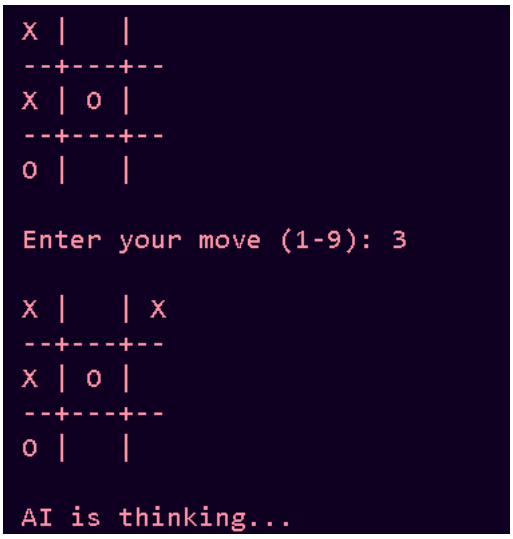
```
        return 1
    if check_winner(board, "X"):  # Player win
        return -1
    if is_draw(board):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for move in get_available_moves(board):
            board[move] = "O"
            score = minimax(board, depth + 1, False)
            board[move] = " "
            best_score = max(best_score, score)
        return best_score
    else:
        best_score = math.inf
        for move in get_available_moves(board):
            board[move] = "X"
            score = minimax(board, depth + 1, True)
            board[move] = " "
            best_score = min(best_score, score)
        return best_score

# Function for AI to find best move
def ai_move(board):
    best_score = -math.inf
    move = None
    for i in get_available_moves(board):
        board[i] = "O"
        score = minimax(board, 0, False)
        board[i] = " "
        if score > best_score:
            best_score = score
            move = i
    return move

# --- Main Game Loop ---
print("Welcome to Tic-Tac-Toe! You are 'X' and AI is 'O'.")
print_board(board)

while True:
    # Player move
    while True:
        try:
            move = int(input("Enter your move (1-9): ")) - 1
```

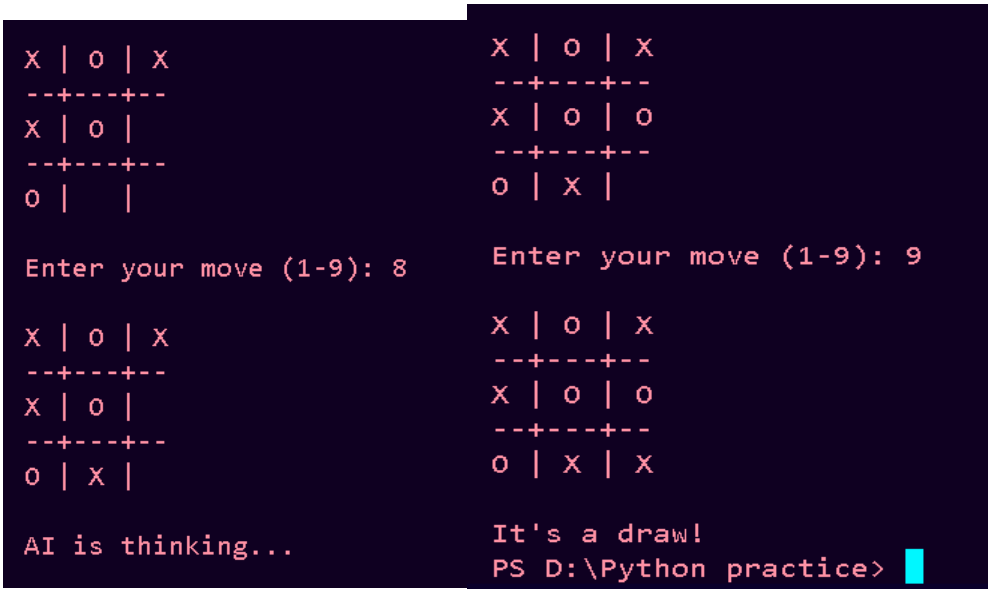
```
            if move in range(9) and board[move] == " ":
                board[move] = "X"
                break
            else:
                print("Invalid move. Try again.")
        except ValueError:
            print("Please enter a number from 1 to 9.")

    print_board(board)

    # Check player win/draw
    if check_winner(board, "X"):
        print("🎉 You win!")
        break
    if is_draw(board):
        print("It's a draw!")
        break

    # AI move
    print("AI is thinking...")
    ai_best = ai_move(board)
    board[ai_best] = "O"
    print_board(board)

    # Check AI win/draw
    if check_winner(board, "O"):
        print("💻 AI wins!")
        break
    if is_draw(board):
        print("It's a draw!")
        break
```

OUTPUT:





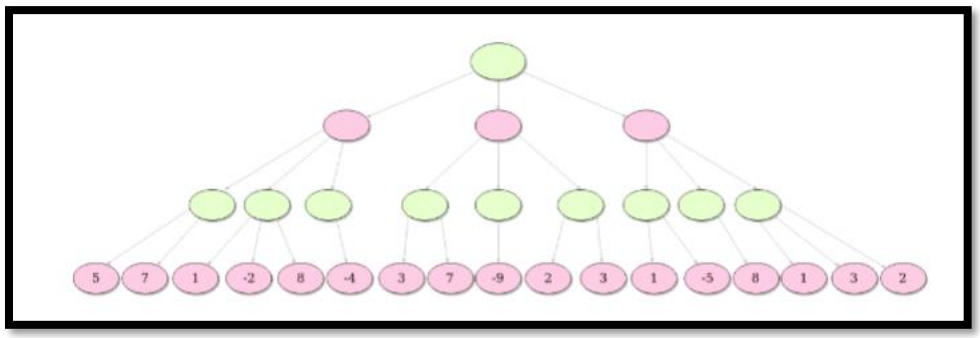




TASK 02:

Write a function that implements the Alpha-Beta Pruning algorithm to evaluate the game tree. The function should take the current node, depth, alpha, and beta values as parameters. While traversing the tree, keep track of the values

that are pruned during the evaluation process and display them in a list. After evaluating the entire tree, return the optimal value at the root node and print it.



CODE:

```
import math

# Leaf nodes of the tree (left to right)
values = [5, 7, 1, -2, 8, -4, 3, 7, -9, 2, 3, 1, -5, 8, 1, 2]

# List to keep track of pruned values
pruned_values = []

# Alpha-Beta Pruning function
def alpha_beta(depth, index, is_max, values, alpha, beta, max_depth):
    # Base case: when we reach a leaf node
    if depth == max_depth:
        return values[index]

    if is_max:
        best = -math.inf
        for i in range(2):  # Two children for each node
            val = alpha_beta(depth + 1, index * 2 + i, False, values, alpha,
beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                # Remaining subtree values are pruned
                remaining_index = index * 2 + i + 1
```

```
                for j in range(remaining_index, 2 ** (max_depth - depth) + index
* 2):
                    if j < len(values):
                        pruned_values.append(values[j])
                break
        return best

    else:
        best = math.inf
        for i in range(2):  # Two children for each node
            val = alpha_beta(depth + 1, index * 2 + i, True, values, alpha, beta,
max_depth)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                # Remaining subtree values are pruned
                remaining_index = index * 2 + i + 1
                for j in range(remaining_index, 2 ** (max_depth - depth) + index
* 2):
                    if j < len(values):
                        pruned_values.append(values[j])
                break
        return best


# Run the Alpha-Beta Pruning algorithm
alpha = -math.inf
beta = math.inf
max_depth = 4  # since we have 16 leaf nodes (2^4)

optimal_value = alpha_beta(0, 0, True, values, alpha, beta, max_depth)

print("Optimal value at root node:", optimal_value)
print("Values that were pruned during evaluation:", pruned_values)
```

OUTPUT:

```
Optimal value at root node: 3
Values that were pruned during evaluation: [-2, 2, 1, -2, 8, -4,
 3, 7, -9, 2]
```

END