NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY:

DEPARTMENT OF COMPUTER SCIENCE & IT

ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEMS (CT-361)

ASSIGNMENT

NAME: SYEDA SHINZA WASIF

ROLL NO: CT-22063

SECTION: B

TEACHER: DR. WASEEMULLAH NAZIR

Question:

This assignment focuses on understanding and implementing the Minimax algorithm and its optimization using Alpha-Beta Pruning for the classic game of Tic-Tac-Toe.

Implement the core logic of the Tic-Tac-Toe game.

Implement the Minimax algorithm to create an AI player that plays optimally.

Implement the Alpha-Beta Pruning optimization to improve the efficiency of the Minimax algorithm.
Compare the performance of the standard Minimax and the Alpha-Beta Pruning optimized Minimax.

Code Implementation Of Tic Tac Toe Using Minimax Algorithm

```
# import random

# Create an empty board

def create_board():
      return [[" " for _ in range(3)] for _ in range(3)]

# Print the board in a user-friendly format
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

# Check if the board is full
def is_full(board):
    return all(cell != " " for row in board for cell in row)

# Check for a winner
def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
```

```
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return board[0][2]
    return None

# Let players make a move
def make_move(board, row, col, player):
    if board[row][col] == " ":
        board[row][col] = player
        return True
    return False

# Minimax Algorithm
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner == "X":
        return -10 + depth
    elif winner == "O":
        return 10 - depth
    elif is_full(board):
        return 0

    if is_maximizing:
        max_eval = float('-inf')
        for row in range(3):
            for col in range(3):
                if board[row][col] == " ":
                    board[row][col] = "O"
                    eval = minimax(board, depth + 1, False)
                    board[row][col] = " "
                    max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float('inf')
        for row in range(3):
```

```
            for col in range(3):
                if board[row][col] == " ":
                    board[row][col] = "X"
                    eval = minimax(board, depth + 1, True)
                    board[row][col] = " "
                    min_eval = min(min_eval, eval)
        return min_eval

# AI move using Minimax
def ai_move(board):
    best_score = float('-inf')
    move = None
    for row in range(3):
        for col in range(3):
            if board[row][col] == " ":
                board[row][col] = "O"
                score = minimax(board, 0, False)
                board[row][col] = " "
                if score > best_score:
                    best_score = score
                    move = (row, col)
    return move

# Run the game with AI
def play_tictactoe():
    board = create_board()
    current_player = "X"

    while True:
        print_board(board)

        if current_player == "X":
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter col (0-2): "))
            if not make_move(board, row, col, current_player):
                print("Cell already taken, try again.")
                continue
        else:
            print("AI is making its move...")
```

```
            row, col = ai_move(board)
            make_move(board, row, col, current_player)

        winner = check_winner(board)
        if winner:
            print_board(board)
            print(f"Player {winner} wins!")
            break
        elif is_full(board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = "O" if current_player == "X" else "X"

# Run the game
if __name__ == "__main__":
    play_tictactoe()
```

Output:

```
   |   |
 -----
   |   |
 -----
   |   |
 -----
Enter row (0-2):  1
Enter col (0-2):  1
   |   |
 -----
   | X |
 -----
   |   |
 -----
AI is making its move...
O |   |
 -----
   | X |
 -----
   |   |
 -----
```

```
Enter row (0-2):  2
Enter col (0-2):  1
O |   |
-----
  | X |
-----
  | X |
-----
AI is making its move...
O | O |
-----
  | X |
-----
  | X |
-----
Enter row (0-2):  0
Enter col (0-2):  2
O | O | X
-----
  | X |
-----
  | X |
-----
AI is making its move...
O | O | X
-----
  | X |
-----
O | X |
-----

 Enter row (0-2):  2
 Enter col (0-2):  2
 O | O | X
 -----
   | X |
 -----
 O | X | X
 -----
 AI is making its move...
 O | O | X
 -----
 O | X |
 -----
 O | X | X
 -----
 Player O wins!
```

Code Implementation Of Tic Tac Toe Using Optimized Minimax Algorithm (Alpha Beta Pruning)

```
def create_board():
      return [[" " for _ in range(3)] for _ in range(3)]

def print_board(board):
```

```
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

def is_full(board):
    return all(cell != " " for row in board for cell in row)

def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return board[0][2]
    return None

def make_move(board, row, col, player):
    if board[row][col] == " ":
        board[row][col] = player
        return True
    return False

def minimax(board, depth, is_maximizing, alpha, beta):
    winner = check_winner(board)
    if winner == "X":
        return -10 + depth
    elif winner == "O":
        return 10 - depth
    elif is_full(board):
        return 0

    if is_maximizing:
        max_eval = float('-inf')
        for row in range(3):
            for col in range(3):
                if board[row][col] == " ":
```

```
                    board[row][col] = "O"
                    eval = minimax(board, depth + 1, False,
alpha, beta)  # Recursively call minimax for the AI move
                    board[row][col] = " "
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)  # Update alpha
value
                    if beta <= alpha:  # Alpha-Beta Pruning:
Prune the tree if beta is less than or equal to alpha
                        break
        return max_eval
    else:
        min_eval = float('inf')
        for row in range(3):
            for col in range(3):
                if board[row][col] == " ":
                    board[row][col] = "X"
                    eval = minimax(board, depth + 1, True, alpha,
beta)  # Recursively call minimax for the player's move
                    board[row][col] = " "
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)  # Update beta value
                    if beta <= alpha:  # Alpha-Beta Pruning:
Prune the tree if beta is less than or equal to alpha
                        break
        return min_eval

def best_move(board):
    best_value = float('-inf')
    move = (-1, -1)

    for row in range(3):
        for col in range(3):
            if board[row][col] == " ":
                board[row][col] = "O"
                move_value = minimax(board, 0, False, float('-
inf'), float('inf'))  # Initial call to minimax with alpha and
beta
                board[row][col] = " "
```

```
                if move_value > best_value:
                    best_value = move_value
                    move = (row, col)

    return move

def play_tictactoe():
    board = create_board()
    current_player = "X"

    while True:
        print_board(board)
        if current_player == "X":
            print("Player X's turn.")
            try:
                row = int(input("Enter row (0-2): "))
                col = int(input("Enter col (0-2): "))
            except ValueError:
                print("Please enter valid numbers.")
                continue
            if 0 <= row < 3 and 0 <= col < 3:
                if make_move(board, row, col, current_player):
                    winner = check_winner(board)
                    if winner:
                        print_board(board)
                        print(f"Player {winner} wins!")
                        break
                    elif is_full(board):
                        print_board(board)
                        print("It's a draw!")
                        break
                    current_player = "O"
                else:
                    print("That cell is already taken.")
            else:
                print("Invalid move. Try again.")
        else:
            print("AI's turn.")
            row, col = best_move(board)
```

```
                make_move(board, row, col, current_player)
                winner = check_winner(board)
                if winner:
                    print_board(board)
                    print(f"Player {winner} wins!")
                    break
                elif is_full(board):
                    print_board(board)
                    print("It's a draw!")
                    break
                current_player = "X"

if __name__ == "__main__":
    play_tictactoe()
```

Comparison Of Standard Minimax and Alpha-Beta Pruning optimized Minimax:

When implementing the **Minimax algorithm** for games like Tic-Tac-Toe, the goal is to evaluate all possible moves to choose the optimal one. This process can be computationally expensive, especially as the game tree grows. The **Alpha-Beta Pruning** optimization aims to reduce this computational cost without affecting the correctness of the result.

Minimax:

- The Minimax algorithm explores all possible moves in the game tree. For a game like Tic-Tac-Toe, this can mean examining all 9! (factorial) possible game states, as it looks at every branch at each level of the tree.
- In the worst case, the algorithm needs to explore every possible combination of moves.

Alpha-Beta Pruning:

- Alpha-Beta Pruning reduces the number of nodes the algorithm needs to explore. It prunes branches of the game tree that won't affect the final decision. This is achieved by keeping track of two values, **alpha** and **beta**, which represent the best scores for the maximizing and minimizing players, respectively.
- When a branch's score is worse than the current best option (alpha or beta), further exploration of that branch is stopped (pruned).

The output below compares both algorithms by the nodes they visit of the recursive tree and shows that with Alpha Beta pruning tic tac toe games works faster and in more efficient way.

```
Standard Minimax Game:
Standard Minimax Time: 3.400688 seconds
Nodes Visited by Standard Minimax: 615307

Alpha-Beta Minimax Game:
Alpha-Beta Minimax Time: 0.592413 seconds
Nodes Visited by Alpha-Beta Minimax: 80334

Comparison Results:
Time Improvement: 2.808275 seconds
Node Reduction: 534973 nodes
```