Project Title: Tic-Tac-Toe AI using Minimax Algorithm

Course: MSE Project

Submitted by: Dhairya Goel

Date: 11-March-2025

02

1. INTRODUCTION TIC-TAC-TOE

is a classic two-player game where players take turns marking spaces in a 3x3 grid. The objective of this project is to implement an AI for Tic-Tac-Toe using the Minimax algorithm. The AI will evaluate the best possible move to maximize its chances of winning while minimizing the opponent's chances. This project demonstrates artificial intelligence techniques in decision-making and game theory

03

2. METHODOLOGY

2. Methodology The implementation follows these steps:

1. Game Representation: The Tic-Tac-Toe board is represented as a 3x3 matrix.
2. Winner Check: A function (check_winner) is used to determine if a player has won the game.
3. Full Board Check: A function (is_full) is used to check if the board is full, indicating a draw.
4. Minimax Algorithm: This recursive algorithm evaluates all possible moves, scoring them based on potential game outcomes.
   - If AI wins, it returns a score of +1.
   - If the opponent wins, it returns -1.
   - If the game is a draw, it returns 0.
5. Best Move Selection: The AI selects the move that maximizes its chances of winning.

04

CODE:

```
def check_winner(board):
  """Checks if there is a winner on the board by evaluating
rows, columns, and diagonals."""
  # Check rows for a winner
  for row in board:
    if row[0] == row[1] == row[2] and row[0] != ' ':
      return row[0]

  # Check columns for a winner
  for col in range(3):
    if board[0][col] == board[1][col] == board[2][col] and
board[0][col] != ' ':
      return board[0][col]

  # Check diagonals for a winner
  if board[0][0] == board[1][1] == board[2][2] and board[0]
[0] != ' ':
    return board[0][0]
  if board[0][2] == board[1][1] == board[2][0] and board[0]
[2] != ' ':
    return board[0][2]

  return None # No winner yet

def is_full(board):
  """Checks if the board is full (i.e., no empty spaces left),
indicating a tie if no winner is found."""
  for row in board:
    if ' ' in row:
      return False # Found an empty spot, board is not full
  return True # Board is completely filled

def minimax(board, depth, is_maximizing):
  """Implements the minimax algorithm to evaluate the best
move for 'X' (maximizing) and 'O' (minimizing)."""
  winner = check_winner(board) # Check if there's a winner
  if winner == 'X':
    return 1 # Favorable outcome for 'X'
```

05

```
elif winner == 'O':
 return -1 # Favorable outcome for 'O'
 elif is_full(board):
 return 0 # It's a tie

 if is_maximizing:
 best_score = -float('inf') # Start with the lowest possible score
 for i in range(3):
 for j in range(3):
 if board[i][j] == ' ': # Check for available moves
 board[i][j] = 'X' # Simulate 'X' making a move
 score = minimax(board, depth + 1, False) # Recursively evaluate opponent's move
 board[i][j] = ' ' # Undo the move for backtracking
 best_score = max(score, best_score) # Choose the highest score (best move for 'X')
 return best_score
 else:
 best_score = float('inf') # Start with the highest possible score
 for i in range(3):
 for j in range(3):
 if board[i][j] == ' ': # Check for available moves
 board[i][j] = 'O' # Simulate 'O' making a move
 score = minimax(board, depth + 1, True) # Recursively evaluate opponent's move
 board[i][j] = ' ' # Undo the move for backtracking
 best_score = min(score, best_score) # Choose the lowest score (best move for 'O')
 return best_score

def best_move(board):
 """Finds the best possible move for 'X' by evaluating all potential outcomes using
minimax."""
 best_score = -float('inf') # Initialize the best score to a very low value
 move = None # Variable to store the best move
 for i in range(3):
 for j in range(3):
 if board[i][j] == ' ': # Check for available moves
 board[i][j] = 'X' # Simulate 'X' making a move
 score = minimax(board, 0, False) # Get the minimax score for this move
 board[i][j] = ' ' # Undo the move for backtracking
 if score > best_score: # If this move is better, update best_score and move
 best_score = score
 move = (i, j)
 return move # Return the best move found

# Example usage: Testing the best move function
board = [
 ['X', 'O', 'X'], # Example Tic-Tac-Toe board configuration
 ['O', 'X', ' '],
 [' ', ' ', 'O']
]

print("Best move for 'X':", best_move(board)) # Output the optimal move for 'X'
```

07

OUTPUT:

```
...   Best move for 'X': (2, 0)
```