Assignment 9

Name : Sanket Kele- SE-B-B2-34

---

1. Aim

Understand and implement the basic Minimax algorithm for two-player deterministic, zero-sum games and apply it to a simple game (Tic-Tac-Toe). Evaluate the algorithm's behavior and discuss limitations and improvements.

2. Learning Objectives

By the end of the lab, the student should be able to:

- Explain the minimax decision rule and game trees.
- Implement minimax using recursion to choose optimal moves for perfect-play agents.
- Apply minimax to Tic-Tac-Toe and verify correct play.
- Analyze complexity and discuss pruning (alpha-beta) and depth-limiting.

3. Background / Theory

Two-player, deterministic games with perfect information (e.g., Tic-Tac-Toe, Chess at a conceptual level) can be modeled as a

game tree. Each node represents a game state, and edges represent legal moves. Players alternate turns; one is called

MAX (tries to maximize utility) and the other **MIN** (tries to minimize utility). In a zero-sum game, one player's gain is the other's loss.

The **Minimax** idea is to start from the current state, explore all possible moves down to the terminal states (win, loss, or draw), and evaluate each terminal state with a utility function. For example, for the MAX player: win = +1, draw = 0, loss = -1. These utility values are then propagated upward through the tree: at MAX nodes, the child with the

maximum utility is chosen, and at MIN nodes, the child with the **minimum** utility is chosen. The decision at the root node yields the best possible move, assuming the opponent also plays perfectly.

The time complexity of the algorithm is

O(bd), where b is the branching factor (number of legal moves from each position) and d is the depth of the search. [19]While Tic-Tac-Toe has a small enough game tree to be solved fully, larger games require optimizations like depth-limiting and heuristics.

4. Algorithm

The Minimax algorithm can be described with the following pseudocode. It explores the game tree recursively to find the optimal value for the current player.

```
function minimax(node, depth, is_maximizing_player)

    // Check if we've reached a terminal state (win/loss/draw) or max depth
    if node is a terminal node:
        return utility_value_of(node)

    // MAX player's turn (wants to maximize the score)
    if is_maximizing_player:
        best_value = -infinity
        for each child of node:
            value = minimax(child, depth + 1, false)
            best_value = max(best_value, value)
        return best_value

    // MIN player's turn (wants to minimize the score)
    else:
        best_value = +infinity
        for each child of node:
            value = minimax(child, depth + 1, true)
            best_value = min(best_value, value)
        return best_value
```

---

5. Python Implementation

Here is a complete Python implementation of the Minimax algorithm for Tic-Tac-Toe. The AI (player 'X') uses the minimax function to find the best possible move against the human player ('O').

Python

```python
import math

# The Tic-Tac-Toe board is represented as a list of 9 elements.
# '-' represents an empty cell.
board = ['-'] * 9
human_player = 'O'
ai_player = 'X'

def print_board(current_board):
    """Prints the Tic-Tac-Toe board in a 3x3 grid."""
    print("-------------")
    for i in range(3):
        print(f"| {current_board[i*3]} | {current_board[i*3+1]} | {current_board[i*3+2]} |")
        print("-------------")

def is_winner(b, player):
    """Checks if the given player has won the game."""
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
        [0, 4, 8], [2, 4, 6]              # Diagonals
    ]
    for condition in win_conditions:
        if b[condition[0]] == b[condition[1]] == b[condition[2]] == player:
            return True
    return False

def check_game_over(b):
    """Checks for a terminal state (win, loss, or draw) and returns a utility score."""
    if is_winner(b, ai_player):
        return 1  # AI wins
    elif is_winner(b, human_player):
        return -1 # Human wins
    elif '-' not in b: # Board is full
        return 0  # Draw
    else:
        return None # Game is not over

def minimax(b, is_maximizing):
    """Minimax algorithm implementation to determine the score of a board state."""
```

```
    score = check_game_over(b)
    if score is not None:
        return score

    if is_maximizing:
        best_score = -math.inf
        for i in range(9):
            if b[i] == '-':
                b[i] = ai_player
                score = minimax(b, False)
                b[i] = '-'
                best_score = max(score, best_score)
        return best_score
    else: # Minimizing player
        best_score = math.inf
        for i in range(9):
            if b[i] == '-':
                b[i] = human_player
                score = minimax(b, True)
                b[i] = '-'
                best_score = min(score, best_score)
        return best_score

def find_best_move(b):
    """Finds the best possible move for the AI by iterating through all empty cells."""
    best_score = -math.inf
    move = -1
    for i in range(9):
        if b[i] == '-':
            b[i] = ai_player
            score = minimax(b, False)
            b[i] = '-'
            if score > best_score:
                best_score = score
                move = i
    return move

# Main game loop
if __name__ == "__main__":
    print("Welcome to Tic-Tac-Toe!")
    print("You are 'O'. The AI is 'X'. Cell numbers are 0-8.")
    print_board(board)

    while True:
```

```
        # Human's turn
        try:
            player_move = int(input("Enter your move (0-8): "))
            if 0 <= player_move <= 8 and board[player_move] == '-':
                board[player_move] = human_player
            else:
                print("Invalid move. Try again.")
                continue
        except ValueError:
            print("Invalid input. Please enter a number.")
            continue

        print("\nYour move:")
        print_board(board)

        if check_game_over(board) is not None:
            break

        # AI's turn
        print("\nAI is thinking...")
        ai_move = find_best_move(board)
        board[ai_move] = ai_player

        print(f"AI chose position {ai_move}:")
        print_board(board)

        if check_game_over(board) is not None:
            break

    # Announce the result
    result = check_game_over(board)
    if result == 1:
        print("AI wins! Better luck next time. 😉")
    elif result == -1:
        print("Congratulations! You win! 🏆")
    else:
        print("It's a draw! 🤝")
```

---

6. Sample Output

Below is a sample output of a game where the human plays against the AI. The game results in a draw, which is the expected outcome if both players play optimally.

```
Welcome to Tic-Tac-Toe!
You are 'O'. The AI is 'X'. Cell numbers are 0-8.
-------------
| - | - | - |
-------------
| - | - | - |
-------------
| - | - | - |
-------------
Enter your move (0-8): 4

Your move:
-------------
| - | - | - |
-------------
| - | O | - |
-------------
| - | - | - |
-------------

AI is thinking...
AI chose position 0:
-------------
| X | - | - |
-------------
| - | O | - |
-------------
| - | - | - |
-------------
Enter your move (0-8): 2

Your move:
-------------
| X | - | O |
-------------
| - | O | - |
-------------
| - | - | - |
-------------
```

AI is thinking...
AI chose position 6:

| X | - | O |

| - | O | - |

| X | - | - |

Enter your move (0-8): 3

Your move:

| X | - | O |

| O | O | - |

| X | - | - |

AI is thinking...
AI chose position 5:

| X | - | O |

| O | O | X |

| X | - | - |

Enter your move (0-8): 1

Your move:

| X | O | O |

| O | O | X |

| X | - | - |

AI is thinking...
AI chose position 7:

| X | O | O |

```
-------------
| O | O | X |
-------------
| X | X | - |
-------------
Enter your move (0-8): 8

Your move:
-------------
| X | O | O |
-------------
| O | O | X |
-------------
| X | X | O |
-------------
It's a draw! 🤝
```

---

7. Observations

- **Recursive Structure**: The Python implementation directly mirrors the theoretical recursive nature of the Minimax algorithm. The minimax function calls itself for each possible future move, effectively traversing the game tree.
- **Guaranteed Optimal Play**: The algorithm plays perfectly. It assumes the human opponent will always make the best possible move to minimize the AI's score. By preparing for this "worst-case" scenario at every step, the AI ensures it can never be forced into a losing position.
- **State Evaluation**: The game's outcome is decided only at **terminal nodes** (a win, loss, or draw), where a utility function assigns a score (+1, -1, or 0). These scores are then "backed up" the tree to inform the best decision at the current board state.
- **Computational Cost**: For Tic-Tac-Toe, the total number of possible board states is small (less than 5,478 unique states), so the algorithm runs almost instantaneously. However, this demonstrates the O(bd) complexity; for a game like chess with a massive branching factor and depth, exploring the entire tree this way would be computationally impossible.

---

8. Conclusion

This experiment successfully demonstrated the implementation of the Minimax algorithm for an AI agent in the game of Tic-Tac-Toe. The algorithm enables the AI to play optimally by recursively exploring the game tree, evaluating terminal states, and selecting the move that

maximizes its own utility while assuming the opponent will play to minimize it.

The agent created is "unbeatable" in the sense that you can, at best, force a draw against it. The primary limitation of this basic Minimax algorithm is its

exponential time complexity (O(bd)), which makes it impractical for games with large state spaces like Chess or Go. [23]Future improvements would involve implementing

alpha-beta pruning to significantly reduce the number of nodes evaluated in the game tree without affecting the final result, or using **heuristic evaluation functions** to analyze non-terminal states and limit the search depth.