Fr. Conceicao Rodrigues College of Engineering

Department of Computer Engineering (CE)

EXPERIMENT 5

Practical No:	5
Title:	Game Playing and Adversarial Search
Date of Performance:	19/03/2025
Date of Submission:	27/04/2025
Roll No:	9913
Name of the Student:	Mark Lopes

Rubrics for Evaluation:

Sr. No	Performance Indicator	Excellent	Good	Below Average	Total Score
1	On time Completion & Submission (01)	01 (On Time)	NA	00 (Not on Time)	
2	Logic/Theory understanding(02)	02(Correct)	NA	01 (Tried)	
3	Coding Standards (03): Comments/indention/Naming conventions Output/Test Cases	03(All used)	02 (Partial)	01 (rarely followed)	
4	Post Lab Assignment (04)	04(done well)	3 (Partially Correct)	2(submitted)	

Academic Year	**2024-25**	**Estimated Time**	**Experiment No. 5 – 02 Hours**
Course & Semester	**T.E. (CE) – Sem. VI**	**Subject Name**	**CSC604: Artificial Intelligence Lab**
Chapter No.	**03**	**Chapter Title**	**Game Playing and Adversarial Search**
Experiment Type	**Software/Finding Solution**	**Software**	**Prolog/Python**

AIM: Write a program to solve a Tic- Tac- Toe game problem in Prolog/ Python.

1. OBJECTIVES

• To gain insights of adversarial search through basic game playing algorithms.

• To be able to implement search space improvement techniques for game playing.

2. DEMONSTRATION OF USEFUL RESOURCES

Game playing assumes multiple-agent environment, and thus offers ideal example for adversarial search. As the agents' goals are in conflict and they always plan against each other, the search space becomes complicated. Moreover, real games involve huge state spaces.

3. Finding Optimal Game Strategies using MINIMAX Algorithm

Two-player board game as a search problem:

✓ Players are usually named MAX & MIN. Anyone can start, and they make moves alternating one another.

✓ Search problem with 4 components: Initial state, Successor function, Terminal test, Utility function.

✓ Strategies of Players: MAX searches for the sequence of moves that leads to a terminal with maximum possible utility value, even if MIN plays in the best way; MIN searches for the opposite, that is, terminal with minimum possible utility.

✓ Major steps of the MINIMAX algorithm, from opener's point of view:

1. Generate the whole game tree.
2. Find the utility of the terminal nodes.

3. Determine the MINIMAX values of the non-terminal nodes, from lower nodes up to the root. If a level represents MAX's turn, then the highest values of the successors are taken, and in case of MIN's – lowest values.

4. Choose the best opening move.

✓ An imaginary game of small depth may be used for explanation. The game of Tic-Tac-Toe is suggested for implementation in Python or Prolog.

- A 3x3 grid is provided with the information of opener, and his/her symbol.

- All nodes up to the terminals are generated, and utilities (-1, 0, +1) are assigned to them.
- The MINIMAX values of non-terminals are computed up to the root, and the winning strategy is returned.

4. Attach the screenshot of the code.

```
import streamlit as st
import math
import time

# Set page config
st.set_page_config(page_title="Tic Tac Toe AI", layout="centered")

# Custom CSS
st.markdown("""
<style>
    .board-cell {
        height: 100px;
        font-size: 2.5rem !important;
        font-weight: bold !important;
    }
    .game-status {
        font-size: 1.5rem;
        padding: 10px;
        border-radius: 5px;
        margin: 10px 0;
        text-align: center;
    }
    .scoreboard {
        font-size: 1.2rem;
        padding: 10px;
        border-radius: 5px;
        margin: 10px 0;
    }
    .footer {
```

```
        text-align: center;
        margin-top: 20px;
        font-size: 0.8rem;
    }
    .move-indicator {
        text-align: center;
        font-size: 1.2rem;
        margin: 10px 0;
    }
    .win-cell {
        background-color: #a8f0a8 !important;
    }
    .stButton button:hover {
        border-color: #ff4b4b;
    }
    /* Custom button styling */
    .stButton button {
        height: 100px;
        font-size: 2.5rem;
        font-weight: bold;
    }
</style>
""", unsafe_allow_html=True)

# Initialize session state
if 'board' not in st.session_state:
       st.session_state.board = [[' ' for _ in range(3)] for _ in
range(3)]
if 'game_over' not in st.session_state:
    st.session_state.game_over = False
if 'winner' not in st.session_state:
    st.session_state.winner = None
if 'winning_cells' not in st.session_state:
    st.session_state.winning_cells = []
if 'player_score' not in st.session_state:
    st.session_state.player_score = 0
if 'ai_score' not in st.session_state:
    st.session_state.ai_score = 0
if 'draws' not in st.session_state:
    st.session_state.draws = 0
if 'show_ai_thinking' not in st.session_state:
    st.session_state.show_ai_thinking = False
if 'difficulty' not in st.session_state:
```

```
    st.session_state.difficulty = "Hard"
if 'ai_goes_first' not in st.session_state:
    st.session_state.ai_goes_first = False
if 'current_turn' not in st.session_state:
                    st.session_state.current_turn      =      'AI'      if
st.session_state.ai_goes_first else 'Player'
if 'ai_scores' not in st.session_state:
     st.session_state.ai_scores = [[None for _ in range(3)] for _ in
range(3)] # Store AI scores


# Check for winning combinations and return winning cells
def check_winner(board):
    # Check rows
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return board[i][0], [(i, 0), (i, 1), (i, 2)]

    # Check columns
    for i in range(3):
        if board[0][i] == board[1][i] == board[2][i] != ' ':
            return board[0][i], [(0, i), (1, i), (2, i)]

    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0], [(0, 0), (1, 1), (2, 2)]

    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2], [(0, 2), (1, 1), (2, 0)]

    return None, []

# Evaluate board for minimax algorithm
def evaluate(board):
    winner, _ = check_winner(board)
    if winner == 'X':
        return 10
    elif winner == 'O':
        return -10
    return 0

# Check if there are moves left
def is_moves_left(board):
```

```
    for row in board:
        if ' ' in row:
            return True
    return False

#  Minimax  with  Alpha-Beta  Pruning  and  limited  depth  for  easier
difficulties
def minimax(board, depth, max_depth, is_maximizing, alpha, beta):
    score = evaluate(board)

    # Terminal conditions
    if score == 10:
        return score - depth  # Prefer quicker wins
    if score == -10:
        return score + depth  # Avoid quick losses
    if not is_moves_left(board) or depth == max_depth:
        return 0

    if is_maximizing:
        best = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'
                          best  =  max(best,  minimax(board,  depth  +  1,
max_depth, False, alpha, beta))
                    board[i][j] = ' '
                    alpha = max(alpha, best)
                    if beta <= alpha:
                        break
        return best
    else:
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'
                          best  =  min(best,  minimax(board,  depth  +  1,
max_depth, True, alpha, beta))
                    board[i][j] = ' '
                    beta = min(beta, best)
                    if beta <= alpha:
                        break
```

```
        return best

# Find the best move with different difficulty levels
def find_best_move():
    # Set max depth based on difficulty
    if st.session_state.difficulty == "Easy":
        max_depth = 1  # Very limited lookahead
    elif st.session_state.difficulty == "Medium":
        max_depth = 3  # Moderate lookahead
    else:  # Hard
        max_depth = 9  # Full lookahead

    best_val = -math.inf
    best_move = (-1, -1)
    scores = [[None for _ in range(3)] for _ in range(3)]

    # Introduce randomness for easier difficulties
    import random
    moves = []
    for i in range(3):
        for j in range(3):
            if st.session_state.board[i][j] == ' ':
                moves.append((i, j))

    # For Easy difficulty, sometimes make a random move
    if st.session_state.difficulty == "Easy" and random.random() < 0.4:
        return random.choice(moves), scores

    # For Medium difficulty, sometimes make a slightly suboptimal move
    make_suboptimal = st.session_state.difficulty == "Medium" and
random.random() < 0.3

    for i in range(3):
        for j in range(3):
            if st.session_state.board[i][j] == ' ':
                st.session_state.board[i][j] = 'X'
                         move_val = minimax(st.session_state.board, 0,
max_depth, False, -math.inf, math.inf)
                st.session_state.board[i][j] = ' '

                scores[i][j] = move_val

                if move_val > best_val:
```

```
                    best_val = move_val
                    best_move = (i, j)

    # For Medium difficulty, sometimes choose a suboptimal move
    if make_suboptimal and len(moves) > 1:
        suboptimal_moves = [m for m in moves if m != best_move]
        if suboptimal_moves:
            return random.choice(suboptimal_moves), scores

    return best_move, scores

# Handle the player click
def handle_click(row, col):
    if st.session_state.game_over:
        return

          if    st.session_state.board[row][col]    ==    '    '    and
st.session_state.current_turn == 'Player':
        st.session_state.board[row][col] = 'O'
        st.session_state.current_turn = 'AI'
         st.session_state.ai_scores = [[None for _ in range(3)] for _ in
range(3)] # Clear AI scores on player move

        # Check if player won
        winner, winning_cells = check_winner(st.session_state.board)
        if winner == 'O':
            st.session_state.winner = 'Player'
            st.session_state.winning_cells = winning_cells
            st.session_state.game_over = True
            st.session_state.player_score += 1
            return

        if not is_moves_left(st.session_state.board):
            st.session_state.game_over = True
            st.session_state.winner = 'Draw'
            st.session_state.draws += 1
            return

# AI's turn
def ai_move():
     if  st.session_state.game_over  or  st.session_state.current_turn  !=
'AI':
        return
```

```
    if st.session_state.show_ai_thinking:
        with st.spinner("AI is thinking..."):
            time.sleep(1) # Introduce 1-second delay

    best_move, scores = find_best_move()
    st.session_state.ai_scores = scores # Store AI scores
    x, y = best_move

    if x != -1 and y != -1:
        st.session_state.board[x][y] = 'X'
        st.session_state.current_turn = 'Player'

        # Check if AI won
        winner, winning_cells = check_winner(st.session_state.board)
        if winner == 'X':
            st.session_state.winner = 'AI'
            st.session_state.winning_cells = winning_cells
            st.session_state.game_over = True
            st.session_state.ai_score += 1
            return

        if not is_moves_left(st.session_state.board):
            st.session_state.game_over = True
            st.session_state.winner = 'Draw'
            st.session_state.draws += 1

# Reset the game
def reset_game():
       st.session_state.board = [[' ' for _ in range(3)] for _ in
range(3)]
    st.session_state.game_over = False
    st.session_state.winner = None
    st.session_state.winning_cells = []
      st.session_state.ai_scores = [[None for _ in range(3)] for _ in
range(3)] # Clear AI scores on reset

    # Change who goes first based on toggle
                    st.session_state.current_turn = 'AI' if
st.session_state.ai_goes_first else 'Player'

# Change difficulty
def change_difficulty(difficulty):
```

```
    st.session_state.difficulty = difficulty
    reset_game()

# Toggle AI first
def toggle_ai_first():
    st.session_state.ai_goes_first = not st.session_state.ai_goes_first
    reset_game()

# Toggle AI thinking indicator
def toggle_ai_thinking():
                       st.session_state.show_ai_thinking      =      not
st.session_state.show_ai_thinking

# Reset scores
def reset_scores():
    st.session_state.player_score = 0
    st.session_state.ai_score = 0
    st.session_state.draws = 0

# UI Layout
st.title("🎮 Tic Tac Toe - AI vs Human")

# Game settings in sidebar
with st.sidebar:
    st.header("Game Settings")

    # Difficulty selector
    st.subheader("Difficulty")
    col1, col2, col3 = st.columns(3)
    with col1:
        if st.button("Easy", use_container_width=True,
                       type="primary" if st.session_state.difficulty ==
"Easy" else "secondary"):
            change_difficulty("Easy")
    with col2:
        if st.button("Medium", use_container_width=True,
                       type="primary" if st.session_state.difficulty ==
"Medium" else "secondary"):
            change_difficulty("Medium")
    with col3:
        if st.button("Hard", use_container_width=True,
                       type="primary" if st.session_state.difficulty ==
"Hard" else "secondary"):
```

```
            change_difficulty("Hard")

    # Who goes first
    st.subheader("Who goes first?")
    col1, col2 = st.columns(2)
    with col1:
        if st.button("Player", use_container_width=True,
                                                type="primary"  if  not
st.session_state.ai_goes_first else "secondary"):
            if st.session_state.ai_goes_first:
                toggle_ai_first()
    with col2:
        if st.button("AI", use_container_width=True,
                        type="primary" if st.session_state.ai_goes_first
else "secondary"):
            if not st.session_state.ai_goes_first:
                toggle_ai_first()

    # AI thinking toggle
    st.subheader("AI Behavior")
                              st.toggle("Show       AI       thinking",
value=st.session_state.show_ai_thinking,
              on_change=toggle_ai_thinking)

    # Score display
    st.subheader("Scoreboard")
    st.markdown(f"""
    <div class="scoreboard">
        <p>👤 Player: {st.session_state.player_score}</p>
        <p>🤖 AI: {st.session_state.ai_score}</p>
        <p>🤝 Draws: {st.session_state.draws}</p>
    </div>
    """, unsafe_allow_html=True)

    if st.button("Reset Scores", use_container_width=True):
        reset_scores()

# Display whose turn it is
if not st.session_state.game_over:
    st.markdown(f"""
    <div class="move-indicator">
           {"🤖 AI's Turn..." if st.session_state.current_turn == 'AI'
else "👤 Your Turn..."}
```

```
    </div>
    """, unsafe_allow_html=True)

# Game status display
if st.session_state.game_over:
    if st.session_state.winner == 'Player':
        st.markdown("""
        <div class="game-status" style="background-color: #a8f0a8;">
            🎉 You Win! 🎉
        </div>
        """, unsafe_allow_html=True)
    elif st.session_state.winner == 'AI':
        st.markdown("""
        <div class="game-status" style="background-color: #f0a8a8;">
            🤖 AI Wins! Better luck next time.
        </div>
        """, unsafe_allow_html=True)
    else:
        st.markdown("""
        <div class="game-status" style="background-color: #f0f0a8;">
            🤝 It's a Draw!
        </div>
        """, unsafe_allow_html=True)

# Create Tic Tac Toe board grid with improved styling
st.markdown("### Game Board")
for i in range(3):
    cols = st.columns(3)
    for j in range(3):
        with cols[j]:
            cell_content = st.session_state.board[i][j]

            # Check if this cell is part of the winning combination
            is_winning_cell = (i, j) in st.session_state.winning_cells

            # Style based on content and winning status
            if cell_content == 'X':
                if is_winning_cell:
                    cell_style = "background-color: #a8f0a8; color:
#ff4b4b;"
                else:
                    cell_style = "background-color: #ffebeb; color:
#ff4b4b;"
```

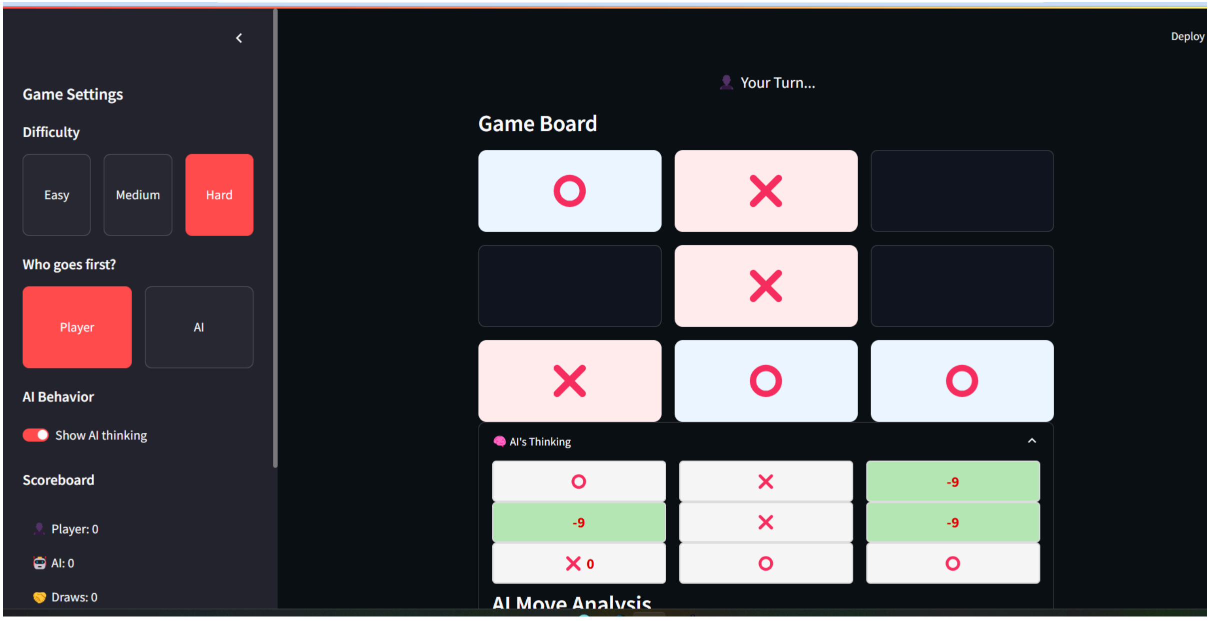
```
                cell_display = "❌"
            elif cell_content == 'O':
                if is_winning_cell:
                        cell_style = "background-color: #a8f0a8; color:
#4b7bff;"
                else:
                        cell_style = "background-color: #ebf5ff; color:
#4b7bff;"
                cell_display = "⭕"
            else:
                cell_style = "background-color: #f5f5f5;"
                cell_display = " "

            # Create the button with appropriate styling
            if cell_content == ' ' and not st.session_state.game_over:
                  # For empty cells, we use a standard button (without
the problematic css parameter)
                                        st.button("   ", key=f"{i}-{j}",
use_container_width=True,
                            on_click=handle_click, args=(i, j))
            else:
                # For filled cells or game over, display a styled div
                st.markdown(
                    f"""
                            <div style="{cell_style} display: flex;
align-items: center; justify-content: center; height: 100px;
border-radius: 10px; font-size: 2.5rem; font-weight: bold;">
                    {cell_display}
                    </div>
                    """,
                    unsafe_allow_html=True
                )

# AI info section
if st.session_state.show_ai_thinking:
        with st.expander("🧠 AI's Thinking", expanded=True): # Keep
expander open by default when thinking is shown

          if st.session_state.current_turn == 'AI': # Only recalculate
scores when it's AI's turn
            _, scores = find_best_move()
            st.session_state.ai_scores = scores # Update stored scores
```

```
        # Display the scores in a grid using stored scores
        for i in range(3):
            score_cols = st.columns(3)
            for j in range(3):
                with score_cols[j]:
                    cell_content = st.session_state.board[i][j]
                            score = st.session_state.ai_scores[i][j] if
st.session_state.ai_scores[i][j] is not None else ""

                    # Style based on score
                    if score == "":
                        bg_color = "#f9f9f9"
                        text_color = "#333"
                    elif score > 0:
                        intensity = min(abs(score) / 10, 1)
                        bg_color = f"rgba(255, 200, 200, {intensity})"
                             text_color = "#d00" # Red for positive (AI
favor)
                    elif score < 0:
                        intensity = min(abs(score) / 10, 1)
                        bg_color = f"rgba(200, 255, 200, {intensity})"
                          text_color = "#d00" # Red for negative (Player
favor) - changed to red as per request
                    else:
                        bg_color = "#f9f9f9"
                            text_color = "#d00" # Red for zero (Draw) -
changed to red as per request

                    # Show the cell with score
                      symbol = "❌" if cell_content == 'X' else "⭕" if
cell_content == 'O' else ""
                    st.markdown(
                        f"""
                        <div style="
                            text-align: center;
                            font-size: 18px;
                            font-weight: bold;
                            padding: 10px;
                            border: 2px solid #ddd;
                            background-color: {bg_color};
                            color: {text_color};
                            border-radius: 5px;
                            height: 50px;
```

```
                            display: flex;
                            align-items: center;
                            justify-content: center;
                        ">
                        {symbol} {score}
                        </div>
                        """,
                        unsafe_allow_html=True
                    )

           st.markdown("### AI Move Analysis") # Moved markdown here,
below the score grid
           st.write("The AI evaluates each possible move and assigns a
score:")
        st.write("- Positive scores favor the AI")
        st.write("- Negative scores favor the player")
        st.write("- Zero indicates a likely draw")


# Game controls
col1, col2 = st.columns(2)
with col1:
    if st.button("🔄 New Game", use_container_width=True):
        reset_game()
with col2:
    # Help button that expands info section
    with st.expander("ℹ️ How to Play"):
        st.markdown("""
        ### Game Rules
        - You play as ⭕, the AI plays as ❌
        - Take turns placing your symbol on the board
         - Get three of your symbols in a row (horizontal, vertical, or
diagonal) to win
        - If all spaces are filled with no winner, the game is a draw

        ### Difficulty Levels
        - **Easy**: AI makes some mistakes and doesn't look ahead much
            - **Medium**: AI plays better but still makes occasional
mistakes
        - **Hard**: AI plays optimally using the minimax algorithm
        """)

# Footer
```

```
st.markdown("""
<div class="footer">
    Tic Tac Toe - AI Powered Game | Built with Streamlit
</div>
""", unsafe_allow_html=True)

# Make AI move if it's the AI's turn
if st.session_state.current_turn == 'AI' and not
st.session_state.game_over:
    ai_move()
    st.rerun()
```

5. Attach the screenshot of the output.



Game can be played at https://mark-ttt.streamlit.app/

6. Conclusion

We implemented Tic-Tac-Toe using the Minimax algorithm with Alpha-Beta pruning. This helped us understand adversarial search and optimal move selection in competitive games. The experiment demonstrated how AI can efficiently make decisions by evaluating possible game states.

7. Postlab Questions:

1. Why is it called min-max algorithm
2. How is the min-max algorithm used in solving real life problems? Explain
3. What is min-max v/s max-min
4. What is max - min example
5. What are the properties of the min-max algorithm?

19/3/25

AI Exp- 05 Postlab

Q.1 why is it called minmax algorithm?
→ The minmax algorithm is named after its strategy of minimizing the possible loss for a worst-case scenario (minimizing maximum loss). It is used in decision making and game theory where one player tries to minimize their maximum possible loss & while opponent tries to maximize their gain.

Q.2 How Min-Max algorithm is used in solving real-life problems?
→ a) Game playing - chess, Tic-tac-toe, and other two-player ~~f~~ games to decide the best possible move.
b) AI Decision making - To simulate opponent strategies and plan accordingly
c) Economics - In competitive market analysis where businesses try to minimize losses while maximizing profit
d) Cybersecurity - For anticipating and countering hacking attempts.

what is minmax vs maxmin?
a) Minmax - The player minimizes the possible maximum loss by assuming the opponent will play optimally.
b) Max min - The player maximizes the minimum gain, assuming the player's score.

Q.4 What is maxmin example?
→ 1. In business strategy:
A company choosing a product price that guarantees the highest minimum profit assuming the worst case competitor pricing strategy

2. In games:
A player in a card game choosing the safest play that guarantees the highest ~~for~~ minimum points.

Q.5 What are the properties of minmax algorithm?
- Complete - If search tree is finite, minmax will find a solution.
- Optimal - Provides the best possible strategy if both players play optimally.
- Time complexity - $O(b^d)$ where b is the ~~the~~ branching factor and d is search depth.
- Space complexity - $O(bd)$ for depth - first search
- Deterministic - Assumes perfect information.