**Experiment No. 5**

**D.O.P. : 24/02/2025 D.O.S. :** **03/03/2025**

**Aim:** Implementation of Adversarial search using min max algorithm.

**Theory:**

### Adversarial search is a specialized form of search used to solve decision-making problems where multiple agents (players) are involved, and their goals are in direct conflict. The objective of adversarial search is to make optimal decisions by anticipating and countering the actions of the opposing player, assuming both players are rational and aim to maximize their own outcomes. Adversarial search is often used in two-player games such as chess, checkers, tic-tac-toe, and other competitive environments, where one player's success directly corresponds to the other player's failure.

### Adversarial Search: *Core Concepts*

### In adversarial search, there are two key assumptions:

### **Perfect Rationality**: Each player behaves optimally, trying to maximize their chances of winning or minimizing their losses.

### **Zero-Sum**: The gain of one player is the loss of the other. In this context, the game is often a **zero-sum game**, meaning the total utility for both players adds up to zero.

### Adversarial search can be applied to any competitive scenario where the outcome depends on the interaction between two agents. This search involves creating a model of possible future states and moves, and then evaluating the moves based on the expected outcomes.

The search algorithms like DFS, BFS, and A\* can be well-suited for single-agent environments where there is no direct competition or conflict between multiple agents. These algorithms are suitable for finding an optimal solution in such scenarios. On the

other hand, in zero-sum games where two players compete directly against each other, adversarial search algorithms like Minmax and Alpha-Beta pruning are more appropriate since these algorithms can determine the best course of action for each player in zero-sum games.

**Overview of Min-Max Algorithm**The Min-Max algorithm is a decision-making algorithm used in artificial intelligence, particularly in game theory and adversarial search. It is designed to optimize a player's decision-making by simulating all possible moves in a game, evaluating the outcomes of those moves, and then selecting the optimal one. The main objective is to minimize the possible loss in a worst-case scenario for the maximizer while maximizing the possible gain for the player. It is widely used in two-player zero-sum games where one player’s gain is another player’s loss.

The algorithm assumes that both players are playing optimally. The Maximizing Player (Max) aims to maximize their score, and the Minimizing Player (Min) aims to minimize Max's score.

### **Working of Min-Max Algorithm**

The Min-Max algorithm works by creating a game tree where each node represents a game state, and each edge represents a possible move. The algorithm explores all possible moves and calculates the utility values of each terminal (end) state. Then, it propagates these utility values back up the tree, with each player (Max or Min) selecting the optimal move based on the game state.

#### Steps Involved in the Min-Max Algorithm:

Generate the Game Tree:

* The game tree is created, with each node representing a game state, and edges representing possible moves from that state.

Evaluate Terminal States:

* The terminal states (end states like win, lose, or draw) are assigned utility values:

1. 1: If the maximizing player wins (favorable for Max).
2. 0: If it's a draw (neutral).
3. -1: If the minimizing player wins (favorable for Min).

Propagate Utility Values:

1. Starting from the terminal states, utility values are propagated upward through the tree.
2. If it’s the Maximizing Player’s turn, the algorithm selects the maximum value from its child nodes.
3. If it’s the Minimizing Player’s turn, it selects the minimum value from its child nodes.

Choose the Optimal Move:

* At the root of the tree, the maximizing player selects the move leading to the highest utility value.

#### **Example:**

### Application of Min-Max Algorithm in Chess:

* The **Min-Max algorithm** is widely used in chess engines to make optimal decisions and select the best moves based on a game tree of possible moves and their outcomes. Chess is a perfect application for this algorithm because it’s a two-player, zero-sum game, where one player’s gain is the other player’s loss.

#### Working of Min-Max in Chess:

* Generate the Game Tree:  
  In chess, the game tree starts from the current position of the board. Each node represents a possible configuration of the board after a move, and each edge represents a move made by a player. The tree branches out, representing all possible moves and responses, recursively, until a terminal state (checkmate, stalemate, or draw) is reached.
* Evaluate Terminal States:  
  The evaluation function assigns utility values to the terminal nodes. In chess, these utility values typically reflect the game’s outcome:
  + **+1** for a win by the maximizer (White player).
  + **0** for a draw.
  + **-1** for a win by the minimizer (Black player).
* Propagate Utility Values Upwards:  
  The algorithm works by propagating the utility values up the tree:
  + Maximizing Player’s Turn (White): The algorithm selects the move that leads to the highest possible value (maximizes the score).
  + Minimizing Player’s Turn (Black): The algorithm selects the move that minimizes the maximizing player’s (White’s) score (minimizes the score for White).
* This is done recursively, with the Min-Max algorithm selecting optimal moves at each level of the game tree based on the assumption that both players are playing optimally.
* Select the Optimal Move:  
  At the root of the tree (current game state), the maximizing player (White) selects the move that leads to the highest utility value, considering all possible moves and the responses of the opponent.

#### **Example of Min-Max in Chess**

Let’s say the game tree for a given position looks like this:

Max (White)

/ \

Min Min

/ \ / \

+1 0 -1 +1

* The terminal nodes (leaf nodes) represent the possible outcomes for each move:
  + **+1**: White wins.
  + **0**: Draw.
  + **-1**: Black wins.
* **Black's Turn (Minimizing)**: At Black’s turn, the Min player will choose the minimum value between **+1** (left subtree) and **0** (right subtree), which is **0**.
* **White's Turn (Maximizing)**: Then, White (Max player) will select the maximum value between **0** (left subtree) and **+1** (right subtree), which is **+1**.

Thus, White will select the move that leads to a terminal state where the utility value is **+1**, meaning a win for White.

### **Advantages of Min-Max Algorithm**

1. **Optimal Decision Making**:
   * The Min-Max algorithm guarantees optimal decision-making in two-player zero-sum games by evaluating all possible moves and choosing the best possible one.
2. **Simple and Clear Concept**:
   * The algorithm is easy to understand and implement, especially for games with a finite set of possible states, like chess and tic-tac-toe.
3. **Perfect for Adversarial Environments**:
   * It is especially useful in adversarial environments, where one player’s gain is the other player’s loss, as it ensures optimal play by simulating the opponent’s best responses.
4. **Can be Enhanced**:
   * The Min-Max algorithm can be optimized further using techniques like alpha-beta pruning, which reduces the nu**mber of nodes to be evaluated, speeding up the decision-making process.**

### **Disadvantages of Min-Max Algorithm**

* **High Computational Complexity :** The algorithm evaluates all possible moves at each level of the game tree. As the depth of the tree and the branching factor increase, the number of nodes grows exponentially. This makes it computationally expensive for large and complex games (e.g., chess, Go).
* **Depth Limitations:** To manage computational feasibility, the Min-Max algorithm often limits the depth of the search tree. This can result in suboptimal decisions if crucial moves are missed beyond the limited depth.
* **Assumes Perfect Play**: The Min-Max algorithm assumes both players are playing optimally. In real-world scenarios, this is often not the case, as players may make mistakes or use suboptimal strategies.
* **Doesn't Handle Uncertainty**: The algorithm is designed for deterministic environments, where each move leads to a known outcome. It does not work well in environments with randomness or incomplete information (e.g., poker or any game with hidden information).
* **Inefficient for Large Search Spaces:** In games with a large number of possible moves, the Min-Max algorithm becomes inefficient as the search tree grows too large to explore all possible paths.

**Conclusion:**

The Tic-Tac-Toe implementation uses the Minimax algorithm for Player X (AI) to make optimal moves by evaluating all possible future moves and selecting the best one. Player O (human) inputs moves manually. The algorithm explores the game tree, backpropagates utility values, and ensures both players make optimal choices. This demonstrates how Minimax is effective in adversarial, zero-sum games where two players compete directly.

**Code:**

import math

# Define constants for players

PLAYER\_X = "X"

PLAYER\_O = "O"

EMPTY = " "

# Check if a player has won

def check\_winner(board, player):

# Rows, columns, and diagonals

winning\_combinations = [

[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 combo in winning\_combinations:

if all(board[i] == player for i in combo):

return True

return False

# Evaluate the board and return a score

def evaluate(board):

if check\_winner(board, PLAYER\_X):

return 1

elif check\_winner(board, PLAYER\_O):

return -1

return 0

# Minimax algorithm

def minimax(board, depth, is\_maximizing\_player):

score = evaluate(board)

# If the game is over, return the score

if score == 1 or score == -1:

return score

if " " not in board:

return 0 # Draw

# Maximizing player's move (Player X)

if is\_maximizing\_player:

best = -math.inf

for i in range(9):

if board[i] == " ":

board[i] = PLAYER\_X

best = max(best, minimax(board, depth + 1, not is\_maximizing\_player))

board[i] = " "

return best

# Minimizing player's move (Player O)

else:

best = math.inf

for i in range(9):

if board[i] == " ":

board[i] = PLAYER\_O

best = min(best, minimax(board, depth + 1, not is\_maximizing\_player))

board[i] = " "

return best

# Find the best move for the maximizing player (Player X)

def find\_best\_move(board):

best\_val = -math.inf

best\_move = -1

for i in range(9):

if board[i] == " ":

board[i] = PLAYER\_X

move\_val = minimax(board, 0, False)

board[i] = " "

if move\_val > best\_val:

best\_move = i

best\_val = move\_val

return best\_move

# Function to print the board

def print\_board(board):

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]}")

# Play the game

def play\_game():

board = [EMPTY] \* 9

current\_player = PLAYER\_X

while " " in board:

print\_board(board)

if current\_player == PLAYER\_X:

print("Player X's move:")

move = find\_best\_move(board)

board[move] = PLAYER\_X

else:

print("Player O's move:")

move = int(input("Enter position (1-9): ")) - 1

if board[move] != " ":

print("Invalid move! Try again.")

continue

board[move] = PLAYER\_O

if check\_winner(board, PLAYER\_X):

print\_board(board)

print("Player X wins!")

break

elif check\_winner(board, PLAYER\_O):

print\_board(board)

print("Player O wins!")

break

current\_player = PLAYER\_O if current\_player == PLAYER\_X else PLAYER\_X

if " " not in board:

print\_board(board)

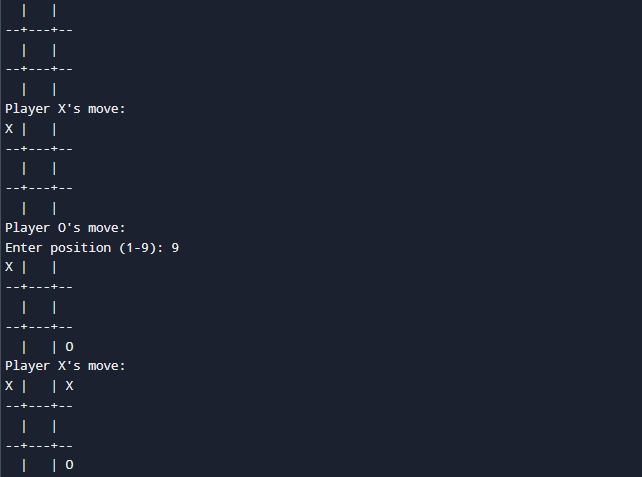
print("It's a draw!")

# Run the game

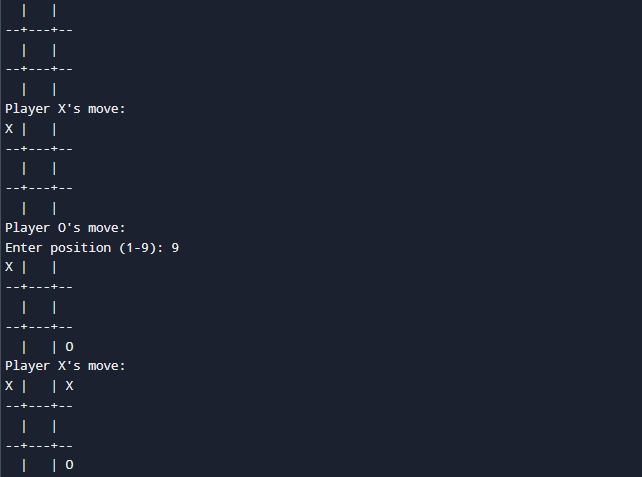
play\_game()

**Output:**

**a.**

****

**b.**

****

**c.**

****