EXPERIMENT-4

Name: Adwait Purao
UID: 2021300101
Div: B
Batch: B2

Problem Definition: Implement a given problem using the Informed searching technique min-max algorithm. Analyse the algorithm with respect to Completeness, Optimality, time and space Complexity.

1. Connect Four

Theory:

The min-max algorithm is an informed search technique that can be used to find the optimal move for a player in a two-player zero-sum game, assuming that the opponent is also playing optimally. It works by recursively evaluating all possible moves that the current player and the opponent player can make, and then returning the move that maximizes the current player's expected score, while minimizing the opponent's expected score.

The min-max algorithm is often used in game theory and artificial intelligence to develop computer programs that can play games against human opponents. It has been used to create programs that can play games such as chess, checkers, and tic-tac-toe at a superhuman level.

Brief about Min-max algorithm:

The min-max algorithm works by traversing a game tree, which is a tree that represents all possible moves that can be made in a game. The root node of the game tree represents the current state of the game, and the child nodes of a node represent the possible moves that can be made from that state.

The min-max algorithm recursively evaluates each node in the game tree, starting at the root node and working its way down to the leaf nodes. At each node, the min-max algorithm evaluates the node's value by assuming that the current player and the opponent player will play optimally.

If the current player is maximizing the value of the node, then the min-max algorithm will return the maximum value of its child nodes. If the current player is minimizing the value of the node, then the min-max algorithm will return the minimum value of its child nodes.

The min-max algorithm terminates when it reaches a leaf node in the game tree. The value of the root node of the game tree is the optimal move for the current player, assuming that the opponent is also playing optimally.

Steps in the program:

- **Step 1:** The program creates a new board using the create_board() function. The board is represented as a 2D numpy array, where each element represents the player who occupies that square (0 for empty, 1 for user, and 2 for computer). The program then prints the board to the console using the print_board() function.
- **Step 2:** While the board is not full and there is no winner, the loop in step 2 is executed.
    - **Step 2.1:** The user enters a column to drop their disc into. The program then checks if the column is valid using the is_valid_move() function. If the column is not valid, the program prompts the user to enter a valid column.
    - **Step 2.2:** If the column is valid, the program drops the user's disc into the column using the drop_disc() function. The drop_disc() function finds the lowest empty row in the column and drops the user's disc into that row.
    - **Step 2.3:** The program then prints the updated board to the console using the print_board() function.
    - **Step 2.4:** The program then checks if the user's move resulted in a win using the check_winner() function. If the user has won, the program prints "You win!" and exits the loop.
    - **Step 2.5:** The program then checks if the board is full using the is_board_full() function. If the board is full, the program prints "It's a draw!" and exits the loop.
    - **Step 2.6:** If the user has not won and the board is not full, the program proceeds to step 3.
- **Step 3:** The computer chooses a column to drop its disc into using the best_move() function. The best_move() function uses the min-max algorithm to find the column that gives the computer the best chance of winning.
- **Step 4:** The program drops the computer's disc into the column using the drop_disc() function.

- **Step 5:** The program then prints the updated board to the console using the print_board() function.
- **Step 6:** The program then checks if the computer's move resulted in a win using the check_winner() function. If the computer has won, the program prints "Computer wins!" and exits the loop.
- **Step 7:** The program then checks if the board is full using the is_board_full() function. If the board is full, the program prints "It's a draw!" and exits the loop.
- **Step 8:** If the computer has not won and the board is not full, the program proceeds back to step 2.1.

The loop in step 2 will continue to execute until there is a winner or the board is full. Once the loop exits, the program will print the final board to the console and exit.

Example:

Consider a game of tic-tac-toe between two players, X and O. The current state of the game is as follows:
X	O	-
-	-	-
Player X is maximizing the value of the current node, so the min-max algorithm will return the maximum value of its child nodes. The child nodes of the current node represent the possible moves that X can make.
X can make three possible moves:

1. Place an X in the top left corner.
2. Place an X in the top right corner.
3. Place an X in the center.

The min-max algorithm will recursively evaluate the values of these child nodes.
For the first child node, the min-max algorithm will assume that O will play optimally. O can block X's move by placing an O in the top left corner. This results in a draw, so the value of the first child node is 0.
The min-max algorithm will perform the same evaluation for the second and third child nodes. In both cases, O can block X's move, resulting in a draw. Therefore, the value of all three child nodes is 0.
The min-max algorithm will then return the maximum value of the child nodes, which is 0. This means that the optimal move for X is to place an X in any of the three empty squares.

Code:

```
import numpy as np

ROWS = 6
COLS = 7
EMPTY = 0
USER = 1
COMPUTER = 2
WINNING_LENGTH = 4

def create_board():
    return np.zeros((ROWS, COLS), dtype=int)

def is_valid_move(board, col):
    return board[ROWS - 1][col] == EMPTY

def drop_disc(board, col, player):
    for row in range(ROWS):
        if board[row][col] == EMPTY:
            board[row][col] = player
            return

def check_winner(board, player):
    # Check horizontal
    for row in range(ROWS):
        for col in range(COLS - WINNING_LENGTH + 1):
            if all(board[row][col + i] == player for i in range(WINNING_LENGTH)):
                return True

    # Check vertical
    for row in range(ROWS - WINNING_LENGTH + 1):
        for col in range(COLS):
            if all(board[row + i][col] == player for i in range(WINNING_LENGTH)):
                return True

    # Check diagonals
    for row in range(ROWS - WINNING_LENGTH + 1):
        for col in range(COLS - WINNING_LENGTH + 1):
            if all(board[row + i][col + i] == player for i in
range(WINNING_LENGTH)) or \
               all(board[row + i][col + WINNING_LENGTH - 1 - i] == player for i
in range(WINNING_LENGTH)):
                return True
```

```
    return False

def is_board_full(board):
    return all(board[0][col] != EMPTY for col in range(COLS))

def print_board(board):
    print(np.flipud(board))

def min_max(board, depth, maximizing_player):
    if depth == 0 or check_winner(board, USER) or check_winner(board, COMPUTER)
or is_board_full(board):
        return evaluate_board(board)

    if maximizing_player:
        max_eval = float('-inf')
        for col in range(COLS):
            if is_valid_move(board, col):
                temp_board = board.copy()
                drop_disc(temp_board, col, COMPUTER)
                eval = min_max(temp_board, depth - 1, False)
                max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float('inf')
        for col in range(COLS):
            if is_valid_move(board, col):
                temp_board = board.copy()
                drop_disc(temp_board, col, USER)
                eval = min_max(temp_board, depth - 1, True)
                min_eval = min(min_eval, eval)
        return min_eval

def best_move(board):
    best_eval = float('-inf')
    best_move = None
    for col in range(COLS):
        if is_valid_move(board, col):
            temp_board = board.copy()
            drop_disc(temp_board, col, COMPUTER)
            eval = min_max(temp_board, 3, False) # You can adjust the depth here
for a stronger/weaker AI
            if eval > best_eval:
                best_eval = eval
```

```
                best_move = col
    return best_move

def evaluate_board(board):
    # Simple evaluation function: +1000 for computer win, -1000 for user win, 0
otherwise
    if check_winner(board, COMPUTER):
        return 1000
    elif check_winner(board, USER):
        return -1000
    else:
        return 0

# Main game loop
board = create_board()
print_board(board)

while True:
    # User's turn
    user_col = int(input("Enter column (0-6): "))
    if is_valid_move(board, user_col):
        drop_disc(board, user_col, USER)
        print_board(board)
        if check_winner(board, USER):
            print("You win!")
            break
        elif is_board_full(board):
            print("It's a draw!")
            break

    # Computer's turn
    computer_col = best_move(board)
    drop_disc(board, computer_col, COMPUTER)
    print("Computer chose column", computer_col)
    print_board(board)
    if check_winner(board, COMPUTER):
        print("Computer wins!")
        break
    elif is_board_full(board):
        print("It's a draw!")
        break
```

Output:

```
PS D:\Rohit College\5th sem\AIML\Exp 4> python -u "d:\Rohit College\5th sem\AIML\Exp 4\Exp_4.py"
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
Enter column (0-6): 1
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0]]
Computer chose column 0
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [2 1 0 0 0 0 0]]
Enter column (0-6): 2
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [2 1 1 0 0 0 0]]
Computer chose column 0
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 1 1 0 0 0 0]]
Enter column (0-6): 0
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 1 1 0 0 0 0]]
```

```
Computer chose column 0
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 1 1 0 0 0 0]]
Enter column (0-6): 3
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 1 1 1 0 0 0]]
Computer chose column 4
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 1 1 1 2 0 0]]
Enter column (0-6): 1
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [2 1 0 0 0 0 0]
 [2 1 1 1 2 0 0]]
Computer chose column 0
[[0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [2 1 0 0 0 0 0]
 [2 1 1 1 2 0 0]]
Enter column (0-6): 1
[[0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [1 1 0 0 0 0 0]
 [2 1 0 0 0 0 0]
 [2 1 1 1 2 0 0]]
```

```
Computer chose column 1
[[0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 2 0 0 0 0 0]
 [1 1 0 0 0 0 0]
 [2 1 0 0 0 0 0]
 [2 1 1 1 2 0 0]]
Enter column (0-6): 2
[[0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 2 0 0 0 0 0]
 [1 1 0 0 0 0 0]
 [2 1 1 0 0 0 0]
 [2 1 1 1 2 0 0]]
Computer chose column 0
[[2 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 2 0 0 0 0 0]
 [1 1 0 0 0 0 0]
 [2 1 1 0 0 0 0]
 [2 1 1 1 2 0 0]]
Enter column (0-6): 2
[[2 0 0 0 0 0 0]
 [2 0 0 0 0 0 0]
 [2 2 0 0 0 0 0]
 [1 1 1 0 0 0 0]
 [2 1 1 0 0 0 0]
 [2 1 1 1 2 0 0]]
Computer chose column 1
[[2 0 0 0 0 0 0]
 [2 2 0 0 0 0 0]
 [2 2 0 0 0 0 0]
 [1 1 1 0 0 0 0]
 [2 1 1 0 0 0 0]
 [2 1 1 1 2 0 0]]
Enter column (0-6): 2
[[2 0 0 0 0 0 0]
 [2 2 0 0 0 0 0]
 [2 2 1 0 0 0 0]
 [1 1 1 0 0 0 0]
 [2 1 1 0 0 0 0]
 [2 1 1 1 2 0 0]]
You win!
PS D:\Rohit College\5th sem\AIML\Exp 4>
```

Analysis of the Code:

- **Completeness:** The min-max algorithm is complete, meaning that it will always find a solution if one exists. This is because the min-max algorithm evaluates all possible moves for both the current player and the opponent.
- **Optimality:** The min-max algorithm is optimal, meaning that it will always find the best move for the current player, assuming that the opponent is also playing optimally.
- **Time complexity:** The time complexity of the min-max algorithm is O(b^m), where b is the branching factor of the game tree and m is the maximum depth of the tree. The branching factor is the number of possible moves that a player can make at any given state of the game. The maximum depth is the number of moves that it takes to reach a leaf node in the game tree.
- **Space complexity:** The space complexity of the min-max algorithm is O(bm). The space complexity is due to the stack that is used to store the nodes in the game tree that have been evaluated but not yet expanded.

Conclusion:

The min-max algorithm is a powerful tool for finding the optimal move in a two-player zero-sum game. However, it can be computationally expensive, especially for games with a large number of possible moves. In practice, a number of techniques are used to improve the performance of the min-max algorithm, such as alpha-beta pruning.

What I learned from the experiment:

- The min-max algorithm is a powerful tool for finding the optimal move in a two-player zero-sum game.
- The min-max algorithm can be computationally expensive, especially for games with a large number of possible moves.
- There are a number of techniques that can be used to improve the performance of the min-max algorithm, such as alpha-beta pruning.