EXPERIMENT – 04

AIM:

Implement a Program for Game Search.

DESCRIPTION:

Implementing a game search algorithm in AI typically involves creating a program that can make decisions in a two-player game, such as chess or tic-tac-toe. One of the most common algorithms used for this purpose is the **Minimax algorithm**, which can be enhanced with techniques like **Alpha-Beta Pruning** to optimize performance.

The Minimax algorithm is a recursive algorithm used to choose an optimal move for a player, assuming that the opponent is also playing optimally. The game is typically represented as a tree of possible moves, where:

- **Min nodes** represent the opponent's turn, where the opponent tries to minimize the player's score.
- **Max nodes** represent the player's turn, where the player tries to maximize their score.

The Minimax algorithm operates by exploring all possible moves (and the resulting game states) to a certain depth, evaluating each terminal state (end game) and propagating these evaluations back up the tree. The algorithm selects the move that leads to the best outcome for the player.

Key Concepts

1. **Game Tree**: Represents all possible moves for both players. Each node in the tree corresponds to a game state.
2. **Maximizer and Minimizer**:
    - **Maximizer**: The player trying to maximize the score (e.g., X in Tic-Tac-Toe).
    - **Minimizer**: The opponent trying to minimize the score (e.g., O in Tic-Tac-Toe).
3. **Evaluation Function**: Determines the score of a terminal state (e.g., +10 for a win, -10 for a loss, 0 for a draw).
4. **Terminal States**: The possible end states of the game, where a player has won, lost, or the game has ended in a draw.
5. **Recursion**: The algorithm recursively explores all possible moves until it reaches terminal states, and then backtracks to decide the optimal move.

Tic-Tac-Toe Game with Minimax Algorithm

The game is to be played between two people One of the player chooses 'O' and the other 'X' to mark their respective cells. The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X'). If no one wins, then the game is said to be draw.

ALGORITHM:

1. **Board Initialization (initialize_board)**: Creates a 3x3 grid filled with empty spaces to represent the Tic-Tac-Toe board.
2. **Move Availability Check (is_moves_left)**: Checks if there are any available moves left on the board.
3. **Board Evaluation (evaluate)**: Evaluates the current state of the board. It returns +10 if the Maximizer (player X) wins, -10 if the Minimizer (player O) wins, and 0 for a draw.
4. **Minimax Algorithm (minimax)**:
5. Recursively explores all possible moves.
6. If it's the Maximizer's turn, it chooses the move with the highest possible score.
7. If it's the Minimizer's turn, it chooses the move with the lowest possible score.
8. Depth is considered to favor quicker wins or slower losses.
9. **Best Move Finder (find_best_move)**: Iterates through all possible moves and uses the Minimax function to determine the best move for the player.
10. The game alternates between the player X and the player O until someone wins or the board is full (resulting in a tie).

CODE:

```
import itertools

def print_board(board):
    print('\n'.join(' '.join(row) for row in board), end='\n\n')

def check_win(board, player):
    win = [player] * 3
    return (any(all(cell == player for cell in row) for row in board) or          # Check rows
            any(all(board[i][j] == player for i in range(3)) for j in range(3)) or  # Check columns
            all(board[i][i] == player for i in range(3)) or                        # Check main diagonal
            all(board[i][2 - i] == player for i in range(3)))                      # Check anti-diagonal

def is_board_full(board):
    return all(cell != ' ' for row in board for cell in row)

def minimax(board, depth, is_maximizing):
    if check_win(board, 'X'): return 10 - depth
```

```
    if check_win(board, 'O'): return depth - 10
    if is_board_full(board): return 0

    best_score = float('-inf') if is_maximizing else float('inf')
    for i, j in itertools.product(range(3), repeat=2):
        if board[i][j] == ' ':
            board[i][j] = 'X' if is_maximizing else 'O'
            score = minimax(board, depth + 1, not is_maximizing)
            board[i][j] = ' '
            best_score = max(score, best_score) if is_maximizing else min(score, best_score)
    return best_score

def find_best_move(board):
    best_move = None
    best_score = float('-inf')
    for i, j in itertools.product(range(3), repeat=2):
        if board[i][j] == ' ':
            board[i][j] = 'X'
            score = minimax(board, 0, False)
            board[i][j] = ' '
            if score > best_score:
                best_score = score
                best_move = (i, j)
    return best_move

def convert_to_indices(position):
    return (position - 1) // 3, (position - 1) % 3

board = [[' ' for _ in range(3)] for _ in range(3)]
print_board(board)
while True:
    try:
        pos = int(input("Enter position for O (1-9): "))
        row, col = convert_to_indices(pos)
        if board[row][col] == ' ':
            board[row][col] = 'O'
            print("O moves:")
            print_board(board)
            if check_win(board, 'O'):
                print("O wins!")
                break
            if is_board_full(board):
                print("It's a tie!")
                break
        else:
            print("Cell already taken. Try again.")
    except (ValueError, IndexError):
        print("Invalid input. Enter a number between 1 and 9.")
```

```
    move = find_best_move(board)
    if move:
        board[move[0]][move[1]] = 'X'
        print("X moves:")
        print_board(board)
        if check_win(board, 'X'):
            print("X wins!")
            break
        if is_board_full(board):
            print("It's a tie!")
            break
```

OUTPUT:

```
Enter position for O (1-9): 5
O moves:

  O

X moves:
X
  O

Enter position for O (1-9): 7
O moves:
X
  O
O
X moves:
X   X
  O
O

Enter position for O (1-9): 2
O moves:
X O X
  O
O
X moves:
X O X
  O
O X

Enter position for O (1-9): 6
O moves:
X O X
  O O
```

```
O X
X moves:
X O X
X O O
O X

Enter position for O (1-9): 9

O moves:
X O X
X O O
O X O

It's a tie!
```

OUTPUT ANALYSIS:

Branching Factor (b):

- On average, the number of possible moves (children of each node) is around 4 or 5, especially in the mid-game.
- In the worst case, the first player has 9 possible moves, the next player has 8, and so on.

Depth (d):

- The maximum depth of the game tree is equal to the total number of moves in a Tic-Tac-Toe game, which is 9 (since there are 9 cells).

Time Complexity Calculation:

- The time complexity of Minimax is O(b^d).
- For Tic-Tac-Toe:
  - b≈9 (maximum branching factor)
  - d=9 (maximum depth)
- Therefore, in the worst case, the time complexity is O(9^9), which simplifies to O(387,420,489).

Space Complexity Calculation:

- The space complexity is proportional to the maximum depth of the recursion (the maximum number of function calls on the call stack).
- For Tic-Tac-Toe, the depth d is 9, so the space complexity due to the call stack is O(d) which is O(9)

Auxiliary Space:

- Apart from the call stack, the algorithm uses constant space for variables like best_score, score, and the game board. Thus, the auxiliary space is O(1).
- Therefore, the overall space complexity is O(d), which is O(9) or simply O(1) as it is constant for the game of Tic-Tac-Toe.

Summary

- **Time Complexity: O(b^d), specifically O(9^9) for Tic-Tac-Toe.**
- **Space Complexity: O(d), specifically O(9) or O(1) for Tic-Tac-Toe.**