EXPERIMENT NO :-02

AIM- Implementation of Depth First Search for Water Jug problem.

The Water Jug problem is a classic example of a state-space search problem. Depth First Search (DFS) can be used to find a solution.

Problem Statement:-

You are given two jugs of capacities x and *y*. Your task is to measure exactly z liters using the two jugs. You can perform the following operations:

Fill a jug completely.

Empty a jug.

Pour water from one jug to another until one is either full or the other is empty.

Explanation:

State Representation: Each state is represented as a tuple (jug1, jug2) indicating the amount of water in the two jugs.

DFS: The stack is used to implement DFS. Each state is paired with the path of steps leading to it.

Transitions: All possible moves (fill, empty, pour) are considered to generate the next states.

Visited States: To avoid cycles, states that have already been visited are skipped.

Goal Check: If the target amount of water is achieved in either jug, the solution path is returned.

This implementation provides a systematic way to explore the problem space using DFS and determine if a solution exists for the given jug capacities and target.

SOURCE CODE-

```
def water_jug_dfs(x, y, z):
    """
    Solve the Water Jug problem using Depth First Search (DFS).
    :param x: Capacity of the first jug
    :param y: Capacity of the second jug
    :param z: Target amount of water
```

```
    :return: Path to the solution or a message if no solution exists
    """
    # Check if the problem is solvable
    if z > max(x, y) or z % gcd(x, y) != 0:
        return "No solution exists."
# Stack for DFS
    stack = []
    # Visited states to avoid revisiting
    visited = set()
    # Solution path
    path = []
# Push the initial state (0, 0) to the stack
    stack.append((0, 0))
 while stack:
        # Get the current state
        current = stack.pop()
if current in visited:
            continue
 # Mark the current state as visited
        visited.add(current)
        path.append(current)
  # Check if the target is reached
        a, b = current
        if a == z or b == z:
            return path
  # Generate all possible next states
        next_states = [
            (x, b),  # Fill the first jug
            (a, y),  # Fill the second jug
            (0, b),  # Empty the first jug
            (a, 0),  # Empty the second jug
            (max(0, a - (y - b)), min(y, b + a)),  # Pour from first to second
```

```
            (min(x, a + b), max(0, b - (x - a)))   # Pour from second to first
        ]
# Add valid next states to the stack
        for state in next_states:
            if state not in visited:
                stack.append(state)
 # If no solution found, backtrack
        if len(stack) == 0:
            path.pop()
return "No solution exists."
def gcd(a, b):
    """Compute the greatest common divisor of a and b."""
    while b:
        a, b = b, a % b
    return a
# Example usage
if __name__ == "__main__":
    x, y, z = 4, 3, 2  # Example capacities and target
    result = water_jug_dfs(x, y, z)
    print(result)
```

OUTPUT-

For x=4, y=3, and z=2, the output could be a sequence of states leading to the solution:

[(0, 0), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3)]

EXPERIMENT NO :-03

AIM- Implementation of Breadth First Search for Tic-Tac-Toe problem.

The Breadth First Search (BFS) can be used to explore all possible states in a Tic-Tac-Toe game. BFS ensures that the states are explored level by level, making it particularly suitable to evaluate the shortest path to a win, draw, or loss.

Problem Statement:

You want to determine the outcome of a Tic-Tac-Toe game using BFS. The state space consists of all possible board configurations, and the game alternates between two players (X and O).

Key Points:

State Representation: The board is represented as a list of 9 elements, where each element can be 'X', 'O', or '.' (empty).

Win Check: The function is_winner determines if a player has won based on predefined winning positions.

Draw Check: The function is_draw determines if the board is full with no winner.

BFS Exploration: The BFS explores all possible states from the initial empty board, alternating between players.

SOURCE CODE-

```
from collections import deque

def is_winner(board, player):
    """Check if the player has won."""
    winning_positions = [
        [0, 1, 2],  # Top row
        [3, 4, 5],  # Middle row
        [6, 7, 8],  # Bottom row
        [0, 3, 6],  # Left column
        [1, 4, 7],  # Middle column
        [2, 5, 8],  # Right column
```

```
        [0, 4, 8],  # Diagonal
        [2, 4, 6],  # Anti-diagonal
    ]
    for positions in winning_positions:
        if all(board[pos] == player for pos in positions):
            return True
    return False
def is_draw(board):
    """Check if the game is a draw (no empty spaces left)."""
    return all(cell != '.' for cell in board)
def bfs_tic_tac_toe():
    """Perform BFS to explore all possible game states."""
    initial_board = ['.'] * 9  # Empty board
    queue = deque([(initial_board, 'X')])  # Initial state with 'X' to move
    visited = set()
    visited.add(tuple(initial_board))
     while queue:
        board, current_player = queue.popleft()
 # Check for win or draw
        if is_winner(board, 'X'):
            print("Found a win for X:")
            print_board(board)
            return
        if is_winner(board, 'O'):
            print("Found a win for O:")
            print_board(board)
            return
        if is_draw(board):
            print("Found a draw:")
            print_board(board)
            return
# Generate all possible next states
```

```
        for i in range(9):
            if board[i] == '.':  # If the cell is empty
                new_board = board[:]
                new_board[i] = current_player
                if tuple(new_board) not in visited:
                    visited.add(tuple(new_board))
                    next_player = 'O' if current_player == 'X' else 'X'
                    queue.append((new_board, next_player))
def print_board(board):
    """Print the Tic-Tac-Toe board."""
    for i in range(0, 9, 3):
        print(" ".join(board[i:i + 3]))
    print()
# Run BFS for Tic-Tac-Toe
if __name__ == "__main__":
    bfs_tic_tac_toe()
```

OUTPUT-

The program will terminate once it finds a win for X, a win for O, or a draw, and print the corresponding board state.

Found a win for X:

X X X

. O O

. . .