8 PUZZLE GAME

IMPLEMENTATION

```
print("1BM22CS324")
print("V DHANUSH REDDY")
from collections import deque

def solve_8puzzle_bfs(initial_state):
    """
    Solves the 8-puzzle using Breadth-First Search.

    Args:
        initial_state: A list of lists representing the initial state of the puzzle.

    Returns:
        A list of lists representing the solution path, or None if no solution is found.
    """

    def find_blank(state):
        """Finds the row and column of the blank tile (0)."""
        for row in range(3):
            for col in range(3):
                if state[row][col] == 0:
                    return row, col

    def get_neighbors(state):
        """Generates possible neighbor states by moving the blank tile."""
        row, col = find_blank(state)
        neighbors = []
```

```
        # Possible moves: Up, Down, Left, Right
        if row > 0:  # Up
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col],
new_state[row][col]
            neighbors.append(new_state)
        if row < 2:  # Down
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col],
new_state[row][col]
            neighbors.append(new_state)
        if col > 0:  # Left
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1],
new_state[row][col]
            neighbors.append(new_state)
        if col < 2:  # Right
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1],
new_state[row][col]
            neighbors.append(new_state)

        return neighbors

    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    # Print initial and goal states
    print("Initial State:")
    for row in initial_state:
        print(row)
    print("\nGoal State:")
```

```
    for row in goal_state:
        print(row)
    print("\nStarting BFS...\n")

    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        current_state, path = queue.popleft()

        # Check if the goal state is reached
        if current_state == goal_state:
            return path + [current_state]

        # Mark the current state as visited
        visited.add(tuple(map(tuple, current_state)))

        # Explore neighbors
        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) not in visited:
                queue.append((neighbor, path + [current_state]))

    return None  # No solution found

# Example usage:
initial_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
solution = solve_8puzzle_bfs(initial_state)
if solution:
    print("\nSolution found:")
    for state in solution:
```

```
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

OUTPUT:

```
1BM22CS324
V DHANUSH REDDY
Initial State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Starting BFS...


Solution found:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

OBSERVATION:

Q Implement 8-puzzle problem using BFS & DFS.

→ BFS Algorithm

Loop

if fringe is empty return failure
Node ← remove_first (fringe)
if Node is a goal
then return the path from initial state to Node.
else generate all successors of Node and add generated nodes to back of fringe

End Loop.

→ DFS Algorithm

Loop.

If fringe is empty return failure.
Node ← remove_first (fringe)
if Node is a goal
then return the path from initial state to Node.

else
generate all successors of Node and add generated nodes to the front of fringe.

End Loop.