```python
# 8 puzzle problem using BFS technique
# prompt: solve 8-puzzle problem using BFS

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."""
        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 = []
        if row > 0:
            new_state = [row[:] for row 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:
            new_state = [row[:] for row 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:
            new_state = [row[:] for row 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:
            new_state = [row[:] for row 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]]
    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        current_state, path = queue.popleft()
        if current_state == goal_state:
            return path + [current_state]

        visited.add(tuple(map(tuple, current_state)))
        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("Solution found:")
    for state in solution:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

```
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]
```

```
from collections import deque

def solve_8puzzle_dfs(initial_state):
    """
    Solves the 8-puzzle using Depth-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."""
        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 = []
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = [r[:] for r in state]
                new_state[row][col], new_state[new_row][new_col] = new_state[new_row][new_col], new_state[row]
                neighbors.append(new_state)
        return neighbors

    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    stack = [(initial_state, [])]
    visited = set()

    while stack:
        current_state, path = stack.pop()
        state_tuple = tuple(map(tuple, current_state))  # Convert to tuple for set
        if state_tuple in visited:
            continue
            continue
        visited.add(state_tuple)

        if current_state == goal_state:
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            stack.append((neighbor, path + [current_state]))

    return None  # No solution found

# Example usage:
initial_state = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
solution = solve_8puzzle_dfs(initial_state)

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

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

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

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

Start coding or generate with AI.