```python
import copy
from heapq import heappop, heappush

n = 3  # Size of the puzzle (3x3 for the 8-puzzle)

# Directions for moving the empty tile: Down, Left, Up, Right
directions = [(1, 0), (0, -1), (-1, 0), (0, 1)]

def calculate_cost(matrix, goal):
    """Calculate the number of misplaced tiles."""
    return sum(1 for i in range(n) for j in range(n) if matrix[i][j] != goal[i][j] and matrix[i][j] != 0)

def is_safe(x, y):
    """Check if the new position is within the puzzle boundaries."""
    return 0 <= x < n and 0 <= y < n

def print_matrix(matrix):
    """Print the current state of the matrix."""
    for row in matrix:
        print(" ".join(map(str, row)))
    print()

def solve(initial, empty_pos, goal):
    """Solve the puzzle using a priority queue."""
    pq = []
    heappush(pq, (calculate_cost(initial, goal), initial, empty_pos, []))  # Include path

    while pq:
        cost, current_matrix, (empty_x, empty_y), path = heappop(pq)

        if cost == 0:  # Goal state reached
            print("Solution path:")
            for state in path + [current_matrix]:  # Include the final state
                print_matrix(state)
            print("Goal achieved!")
            return

        # Generate possible moves
        for dx, dy in directions:
            new_x, new_y = empty_x + dx, empty_y + dy
            if is_safe(new_x, new_y):
                new_matrix = copy.deepcopy(current_matrix)
                # Swap the empty tile with the adjacent tile
                new_matrix[empty_x][empty_y], new_matrix[new_x][new_y] = new_matrix[new_x][new_y], new_matrix[empty_x][empty_y]
                new_cost = calculate_cost(new_matrix, goal)
                heappush(pq, (new_cost, new_matrix, (new_x, new_y), path + [current_matrix]))  # Add current state to path

# Initial configuration
initial = [
    [1, 2, 3],
    [0, 4, 6],
    [7, 5, 8]
]

# Final configuration
goal = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

# Starting position of the empty tile
empty_pos = (1, 0)  # This matches the position of 0 in the initial matrix

# Solve the puzzle
solve(initial, empty_pos, goal)
```

```
Solution path:
1 2 3
0 4 6
7 5 8

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

Goal achieved!
```

```python
import copy

n = 3  # Size of the puzzle (3x3 for the 8-puzzle)

# Directions for moving the empty tile: Down, Left, Up, Right
directions = [(1, 0), (0, -1), (-1, 0), (0, 1)]

def print_matrix(matrix):
    """Print the current state of the matrix."""
    for row in matrix:
        print(" ".join(map(str, row)))
    print()

def dfs(matrix, empty_pos, goal, visited, path, depth, max_depth):
    """Depth-First Search to solve the puzzle."""
    if matrix == goal:
        print("Solution path:")
        for state in path + [matrix]:  # Include the final state
            print_matrix(state)
        print(f"Number of moves: {len(path)}")  # Display the number of moves
        return True

    if depth >= max_depth:
        return False  # Stop if we reach the maximum depth

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

    for dx, dy in directions:
        new_x, new_y = empty_pos[0] + dx, empty_pos[1] + dy
        if 0 <= new_x < n and 0 <= new_y < n:  # Check boundaries
            new_matrix = copy.deepcopy(matrix)
            # Swap the empty tile with the adjacent tile
            new_matrix[empty_pos[0]][empty_pos[1]], new_matrix[new_x][new_y] = new_matrix[new_x][new_y], new_matrix[empty_pos[0]][empty_pos[1]]

            if tuple(map(tuple, new_matrix)) not in visited:
                if dfs(new_matrix, (new_x, new_y), goal, visited, path + [matrix], depth + 1, max_depth):
                    return True

    visited.remove(tuple(map(tuple, matrix)))  # Unmark the state for other paths
    return False

# Initial configuration
initial = [
    [1, 2, 3],
    [0, 4, 6],
    [7, 5, 8]
]

# Final configuration
goal = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

# Starting position of the empty tile
empty_pos = (1, 0)

# Solve the puzzle using DFS
visited = set()  # To keep track of visited states
path = []  # To store the solution path
max_depth = 20  # Set a reasonable maximum depth

if not dfs(initial, empty_pos, goal, visited, path, 0, max_depth):
    print("No solution found.")
```

```
Solution path:
1 2 3
0 4 6
7 5 8

1 2 3
7 4 6
0 5 8

1 2 3
7 4 6
5 0 8

1 2 3
7 0 6
5 4 8

1 2 3
0 7 6
5 4 8

1 2 3
5 7 6
0 4 8

1 2 3
5 7 6
4 0 8

1 2 3
5 0 6
4 7 8

1 2 3
0 5 6
4 7 8

1 2 3
4 5 6
0 7 8

1 2 3
4 5 6
7 0 8

1 2 3
```