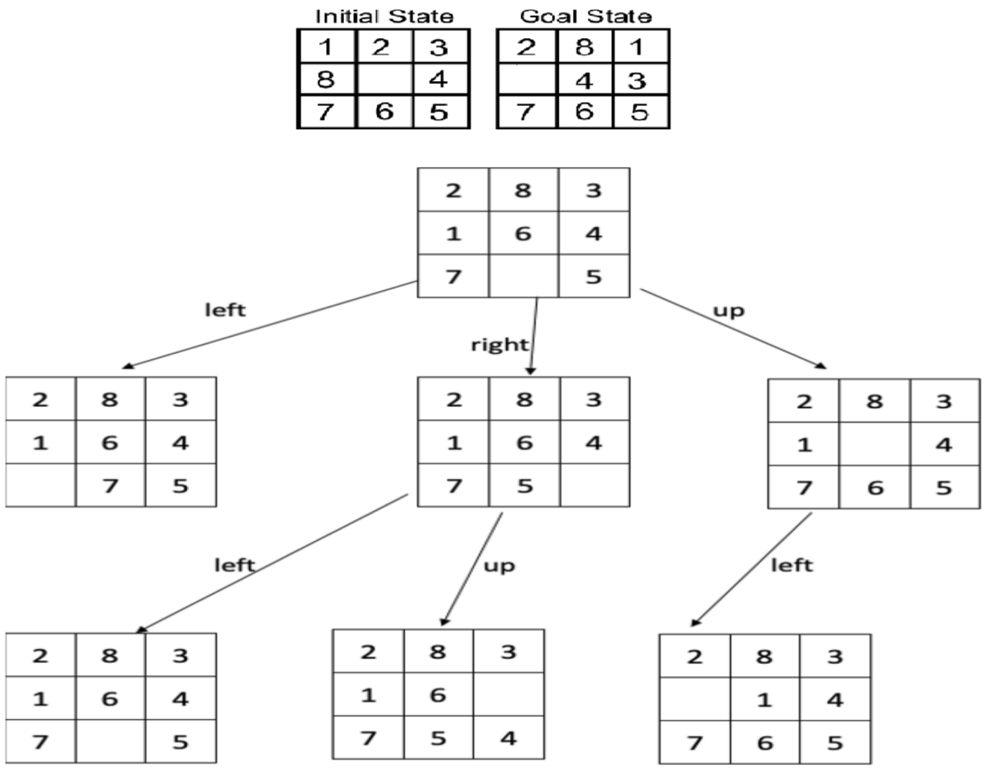
**Program 4**

**Write a Program to Implement 8-Puzzle problem using Python.**



## Description:

The 8-Puzzle is a sliding puzzle that consists of a 3x3 grid with 8 numbered tiles (1 through 8) and one empty space. The objective of the puzzle is to arrange the tiles in a specific goal configuration, typically ordered from 1 to 8 in a row-major order, by sliding the tiles into the empty space. The tiles can only be moved into the empty space (left, right, up, or down).

**Problem Definition:**

- **Initial State:** The initial arrangement of the tiles, where one of the cells is empty (represented by a 0).
- **Goal State:** The goal configuration, which is typically the solved state where the tiles are in order: [1, 2, 3, 4, 5, 6, 7, 8, 0].

- **Moves:** The tiles can be moved into the empty space by sliding them left, right, up, or down, as long as the move stays within the bounds of the grid.
- **Objective:** To find a sequence of moves that transforms the initial state into the goal state.

**Puzzle Representation:**

The puzzle is represented as a 3x3 matrix or list of 9 elements. For example, the following list represents a scrambled 8-puzzle state:

**csharp**

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

**Goal:**

The goal of the 8-puzzle is to rearrange the tiles from any starting configuration into the goal configuration:

**csharp**

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

**Approach to Solve the Problem:**

One way to solve the 8-puzzle is by using **search algorithms**. i.e:

**Breadth-First Search (BFS)** – guarantees the shortest solution.

For simplicity, we will describe how to implement a basic solution using **Breadth-First Search (BFS)** to solve the 8-puzzle problem.

**Key Concepts:**

1. **State Representation:** The board will be represented as a tuple of 9 elements (0-8), where the 0 represents the empty space.
2. **Valid Moves:** The empty space can move in four directions: left, right, up, and down, as long as the move stays within the boundaries of the 3x3 grid.
3. **Goal Check:** At each state, we check if the puzzle matches the goal configuration.

**BFS Algorithm to Solve the 8-Puzzle**

1. **Start** with the initial puzzle configuration.
2. **Generate the next possible configurations** by sliding the tiles into the empty space.
3. **Track the visited states** to avoid revisiting the same state.
4. **Check if the current configuration matches the goal.** If it does, the puzzle is solved.
5. If not, continue exploring the next states using BFS.

**SOURCE CODE :**

```
from collections import deque

def bfs(start_state):
    target = [1, 2, 3, 4, 5, 6, 7, 8 , 0]
    dq = deque([start_state])
    visited = {tuple(start_state): None}

    while dq:
        state = dq.popleft()
        if state == target:
            path = []
            while state:
                path.append(state)
                state = visited[tuple(state)]
            return path[::-1]

        zero = state.index(0)
        row, col = divmod(zero, 3)
        for move in (-3, 3, -1, 1):
            new_row, new_col = divmod(zero + move, 3)
            if 0 <= new_row < 3 and 0 <= new_col < 3 and abs(row -
new_row) + abs(col - new_col) == 1:
                neighbor = state[:]
                neighbor[zero], neighbor[zero + move] = neighbor[zero +
move], neighbor[zero]
                if tuple(neighbor) not in visited:
                    visited[tuple(neighbor)] = state
                    dq.append(neighbor)

def printSolution(path):
    for state in path:
        print("\n".join(' '.join(map(str, state[i:i+3])) for i in range(0,
9, 3)), end="\n-----\n")

# Example Usage
startState = [1, 3, 0 , 6, 8, 4, 7, 5, 2]
solution = bfs(startState)
if solution:
    printSolution(solution)
    print(f"Solved in {len(solution) - 1} moves.")
else:
    print("No solution found.")
```