**Assignment 1**

**Problem Statement:** Implement DFS, BFS for 8-puzzle problem

**Library:**

For Java implementation:

1. java.util.Queue: Used for implementing BFS, as it requires first-in, first-out (FIFO) behavior.
2. java.util.Stack: Used for DFS, where depth-first search explores a path by pushing and popping nodes in a last-in, first-out (LIFO) order.
3. java.util.HashSet: To store visited states, preventing the exploration of already-visited states.
4. java.util.Arrays: For comparing puzzle states.
5. java.util.List / ArrayList: For managing the states of the puzzle during exploration.

**Theory:**

The 8-puzzle problem can be modeled as a state-space search problem where each state is a specific arrangement of tiles. The state space consists of all possible arrangements (configurations) of tiles, and each move (sliding a tile) transforms one state into another. The problem can be solved using search algorithms like DFS and BFS.

1. Depth-First Search (DFS): DFS explores one path from the starting configuration to the goal configuration as deep as possible before backtracking. It's implemented using a stack (LIFO). DFS can sometimes find solutions faster, but it may also get stuck in deep or infinite paths, making it less optimal for large state spaces.
2. Breadth-First Search (BFS): BFS explores the search tree level by level, checking all neighbors before moving deeper. It uses a queue (FIFO) and guarantees finding the shortest solution (in terms of moves), but it requires more memory than DFS.

The 8-puzzle problem has a branching factor of around 2-4 (depending on the position of the empty tile), and both BFS and DFS will explore all possible moves to reach the solution.

**Methodology:**

1. State Representation:
   * Each state of the puzzle can be represented as a 2D array or a 1D array (flattened grid).
   * The position of the empty tile (often denoted as 0) is tracked.
2. DFS Algorithm:
   * Initialize a stack with the starting state.
   * While the stack is not empty, pop the top element (current state), check if it's the goal state, and expand it (move the tiles) to generate child states.
   * Push unvisited child states onto the stack.
   * Backtrack if necessary and explore other paths.
3. BFS Algorithm:
   * Initialize a queue with the starting state.
   * While the queue is not empty, dequeue the front element (current state), check if it’s the goal state, and expand it to generate child states.
   * Enqueue unvisited child states.
   * BFS continues exploring level by level until the solution is found.
4. Goal Test:
   * Check if the current configuration of the puzzle matches the desired goal configuration.
5. Move Generation:
   * Valid moves involve sliding a tile into the empty space (up, down, left, right), and the result of each move creates a new puzzle state.
6. Handling Cycles:
   * Use a set of visited states to avoid revisiting the same puzzle configuration multiple times.

**Advantages:**

* DFS:
  1. Memory efficient as it requires storing only the current path.
  2. Can be faster for deep solutions (if the solution happens to be in a deep path).
* BFS:
  1. Guarantees finding the shortest solution.
  2. Effective for problems where solutions are located near the root of the tree.

**Disadvantages:**

* DFS:
  1. May get stuck in deep or infinite paths, leading to longer runtimes.
  2. Doesn't guarantee the shortest solution.
* BFS:
  1. Memory intensive, as it stores all explored states.
  2. Slower for deep solutions due to exploring all nodes level by level.

**Conclusion:**

The 8-puzzle problem is a classic example of a combinatorial search problem that can be solved using DFS and BFS. Both algorithms have their strengths and weaknesses. DFS is more memory efficient, while BFS is guaranteed to find the shortest solution but can be more resource-heavy. For practical implementations, using additional strategies such as heuristic-based searches (like A\*) can lead to more efficient solutions. However, DFS and BFS remain foundational for understanding basic search techniques in problem-solving.