Lab Assignment 1

Subject: Artificial Intelligence **Guided by:** Dr. Anuradha Yenkikar

Student name: Neha Sawant

Experiment Name: Implement DFS and BFS for the 8-puzzle problem

Objective:

- To understand and implement the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms for solving the 8-puzzle problem.

Problem Statement:

The 8-puzzle problem involves arranging tiles numbered 1 to 8 on a 3x3 grid in such a way that they are in numerical order, with the blank space (denoted by 0) in the bottom-right corner. You need to solve this puzzle using BFS and DFS algorithms.

Algorithm:

1. BFS (Breadth-First Search):
   - Explore all nodes level by level.
   - Ensure all possible configurations are generated and checked until the goal is reached.

2. DFS (Depth-First Search):
   - Explore each branch as deeply as possible.
   - Backtrack when no further moves are possible.

Code:

EightPuzzleBFS.java

AI_1. 8-puzzle problem > EightPuzzleBFS.java > State > State(String, int, int, String)

```java
import java.util.*;

class State {
    String board;
    int zeroIndex;
    int depth;
    String moves;

    State(String board, int zeroIndex, int depth, String moves) {
        this.board = board;
        this.zeroIndex = zeroIndex;
        this.depth = depth;
        this.moves = moves;
    }


    boolean isGoal() {
        return board.equals(anObject:"123456780");
    }


    List<State> getNeighbors() {
        List<State> neighbors = new ArrayList<>();
        int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
        int row = zeroIndex / 3;
        int col = zeroIndex % 3;

        for (int[] dir : directions) {
            int newRow = row + dir[0];
            int newCol = col + dir[1];
```

EightPuzzleBFS.java

AI_1. 8-puzzle problem > EightPuzzleBFS.java > State > State(String, int, int, String)

```java
class State {
    List<State> getNeighbors() {
            if (newRow >= 0 && newRow < 3 && newCol >= 0 && newCol < 3) {
                int newIndex = newRow * 3 + newCol;
                // Swap zero with the target tile
                char[] newBoard = board.toCharArray();
                newBoard[zeroIndex] = newBoard[newIndex];
                newBoard[newIndex] = '0';
                neighbors.add(new State(new String(newBoard), newIndex, depth + 1, moves + getMove(zeroIndex, newIndex)));
            }
        }

        return neighbors;
    }


    private String getMove(int oldIndex, int newIndex) {
        if (newIndex == oldIndex + 3) return "D";
        if (newIndex == oldIndex - 3) return "U";
        if (newIndex == oldIndex + 1) return "R";
        if (newIndex == oldIndex - 1) return "L";
        return "";
    }
}

public class EightPuzzleBFS {
    Run | Debug
    public static void main(String[] args) {
        String initialBoard = "123456078";
        solve(initialBoard);
```

```java
public class EightPuzzleBFS {
    public static void main(String[] args) {
        solve(initialBoard);
    }

    public static void solve(String initialBoard) {
        Queue<State> queue = new LinkedList<>();
        Set<String> visited = new HashSet<>();

        State initialState = new State(initialBoard, initialBoard.indexOf(ch:'0'), depth:0, moves:"");
        queue.add(initialState);
        visited.add(initialState.board);

        while (!queue.isEmpty()) {
            State currentState = queue.poll();

            if (currentState.isGoal()) {
                System.out.println("Solution found with moves: " + currentState.moves + " in depth: " + currentState.depth);
                return;
            }

            for (State neighbor : currentState.getNeighbors()) {
                if (!visited.contains(neighbor.board)) {
                    queue.add(neighbor);
                    visited.add(neighbor.board);
                }
            }
        }
```

Output Example:

```
jdt_ws\AI_Labs_44e150ae\bin' 'EightPuzzleBFS'
Solution found with moves: RR in depth: 2
```

Conclusion:

- BFS explores nodes level by level, ensuring the shortest path is found.

- DFS, when implemented, would explore a path until no further moves are possible before backtracking.