```java
import java.util.Arrays;
import java.util.Set;

public class States {
    int[][] board = new int[3][3];
    int name, depth=0, cost, f, x, y;
    States parent;
```

```java
    //constructor
    public States() {};
    public States(int[][] board, int name, int depth, States parent, int[][] target) {
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++) {
                this.board[i][j] = board[i][j];
                //find the empty cell position
                if (board[i][j] == 0) {
                    x = i;
                    y = j;
                }
            }
        this.board = board;
        this.name = name;
        this.depth = depth;
        this.cost = calculateCost(target);
        this.f = this.cost + this.depth;
        this.parent = parent;
    }
    public States(States Board) {
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                this.board[i][j] = Board.board[i][j];
        this.name = Board.name;
        this.depth = Board.depth;
        this.cost = Board.cost;
        this.f = Board.f;
        this.parent = Board.parent;
    }
```

```java
    //check if the board is goal state
    public boolean isGoal(int[][] target) {
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                if (board[i][j] != target[i][j]) return false;
        return true;
    }
```

```java
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        States state = (States) o;
        return Arrays.deepEquals(board, state.board);
    }
```

```java
    @Override
    public int hashCode() {
        return Arrays.deepHashCode(board);
    }
```

```java
    //print the board
```

```
@Override
public String toString() {
    return "S"+ name ;
}
```

```
//calculate the cost of the board
public int calculateCost(int[][] target) {
    if (board.equals(target))
        return 0;
    cost = 0;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (board[i][j] != target[i][j]) cost++;
    return cost;
}
```

```
//actions
public States moveUp(int[][] target) {
    if (x == 0) return this;
    int[][] newBoard = new int[3][3];
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            newBoard[i][j] = board[i][j];
    newBoard[x][y] = newBoard[x - 1][y];
    newBoard[x - 1][y] = 0;
    return new States(newBoard, name, depth + 1, this, target);
}
public States moveDown(int[][] target) {
    if (x == 2) return this;
    int[][] newBoard = new int[3][3];
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            newBoard[i][j] = board[i][j];
    newBoard[x][y] = newBoard[x + 1][y];
    newBoard[x + 1][y] = 0;
    return new States(newBoard, name, depth + 1, this, target);
}
public States moveLeft(int[][] target) {
    if (y == 0) return this;
    int[][] newBoard = new int[3][3];
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            newBoard[i][j] = board[i][j];
    newBoard[x][y] = newBoard[x][y - 1];
    newBoard[x][y - 1] = 0;
    return new States(newBoard, name, depth + 1, this, target);

}
public States moveRight(int[][] target) {
    if (y == 2) return this;
    int[][] newBoard = new int[3][3];
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            newBoard[i][j] = board[i][j];
    newBoard[x][y] = newBoard[x][y + 1];
    newBoard[x][y + 1] = 0;
    return new States(newBoard, name, depth + 1, this, target);
}
```

```
public void actions(int[][] target, Set<States> adjacent) {
    adjacent.add(moveDown(target));
```

```
        adjacent.add(moveUp(target));
        adjacent.add(moveLeft(target));
        adjacent.add(moveRight(target));
    }
```

```
    //display the board
    public void Display() {
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++)
                System.out.print(board[i][j] + " ");
            System.out.println();
        }
        System.out.println("----------------");
    }
}
```

```
import java.util.*;

public class puzzle {
    List<States> open = new LinkedList<>(); // Create a queue to store the states is going to
be visited
    Queue<States> closed = new LinkedList<>(); // Create a queue to store the visited states
    Set<States> adjacent = new LinkedHashSet<>(); // Create a set to store the adjacent states
of the current state
    int step = 1; // Create a variable to store the step number

    int[][] target = {{1, 2, 3}, {8, 0, 4}, {7, 6, 5}}; // Create a 2D array to store the
target state
    int[][] initial = {{2, 8, 3}, {1, 6, 4}, {7, 0, 5}}; // Create a 2D array to store the
initial state
    States initialState = new States(initial, 0, 0, null, target); // Create a new state to
store the initial state
    int count = 1; // Create a variable to store the name of the state
```

```
    Object[] title = {"STEP", "CURRENT ", "OPENED", "CLOSED"}; // Create a string array to
store the title of the table
```

```
    public void solve8Puzzle() {
        open.add(initialState); // Add the initial state to the open queue
        while (!open.isEmpty()) {
            States current = open.remove(0); // Get the first state from the open queue
            closed.add(current); // Add the current state to the closed queue

            if (current.isGoal(target)) {
                System.out.println("\nGOAL STATE FOUND " + current);
                Stack<States> path = new Stack<>();
                // Add the path to the stack
                while (!current.equals(initialState)) {
                    path.push(new States(current));
                    current = current.parent;
                    // Get the parent of the current state to add to the path from the goal
state to the initial state
                }
                path.push(initialState);
                System.out.print("PATH: \n");
                while (!path.isEmpty()) {
                    System.out.println("S"+path.peek().name);
                    path.pop().Display();
                }
                return;
```

```
            }

            current.actions(target, adjacent); // Get the adjacent states of the current state
            for (States state : adjacent) {
                if (!closed.contains(state) && !open.contains(state)) {
                    if (open.isEmpty()) {
                        for(States adj : adjacent) {
                            if (!closed.contains(adj)){
                                adj.name = count++;
                                open.add(adj); // Add the state to the open queue
                            }
                        }
                    }
                    else{
                        for (int i = 0; i < open.size(); i++) {
                            if (state.f < open.get(i).f) {
                                state.name = count++;
                                open.add(i, state);
                                break;
                            }
                            else if (i == open.size() - 1 ) {
                                state.name = count++;
                                open.add(state); // Add the state to the open queue
                                break;
                            }
                        }
                    }
                }
            }
```

```
            // Display the A* process
            if(step==1) System.out.printf("| %-5s| %-10s | %-70s | %-70s |", title);
            System.out.printf("\n| %-5s| %-10s | %-70s | %-70s |",step++,current,open,closed);
            adjacent.clear();
        }

        System.out.println("\nNO STATES FOUND");

    }
}
```

```
public class App {
    public static void main(String[] args) throws Exception {
        puzzle p = new puzzle();
        p.solve8Puzzle();
    }
}
```