LAB 7

Solve 8-puzzle problem using best first search.

```
package AI;
import java.util.*;

public class Jaswanth {

    final static private String I="134862705";
    final static private String GOAL_STATE ="123804765";
    public static void main(String[] args) {
        String rootState =I;
        long startTime = System.currentTimeMillis();

        SearchTree search = new SearchTree(new NodeM
(rootState),GOAL_STATE);
        search.bestFirstSearch();

        long finishTime = System.currentTimeMillis();
        long totalTime = finishTime - startTime;
        System.out.println("Time  :" + totalTime);
    }
}

 class SearchTree {
    private NodeMroot;
    private String goalSate;

    public NodeMgetRoot() {
```

```
        return root;
    }

    public void setRoot(NodeMroot) {
        this.root = root;
    }
    public String getGoalSate() {
        return goalSate;
    }
    public void setGoalSate(String goalSate) {
        this.goalSate = goalSate;
    }
    public SearchTree(NodeMroot, String goalSate) {
        this.root = root;
        this.goalSate = goalSate;
    }
    private int heuristicOne(String currentState, String goalSate) {
        int difference = 0;
        for (int i = 0; i < currentState.length(); i += 1)
            if (currentState.charAt(i) != goalSate.charAt(i))
                difference += 1;
        return difference;
    }
public void bestFirstSearch() {
        // stateSet is a set that contains node that are already visited
        Set<String> stateSets = new HashSet<String>();
        int totalCost = 0;
```

```
        int time = 0;
        NodeMnode = new NodeM (root.getState());
        node.setCost(0);


        NodePriorityComparator nodePriorityComparator = new
NodePriorityComparator();
        PriorityQueue< NodeM > nodePriorityQueue = new PriorityQueue<
NodeM >(10, nodePriorityComparator);
        NodeMcurrentNode = node;
        while (!currentNode.getState().equals(goalSate)) {
           stateSets.add(currentNode.getState());
           List<String> nodeSuccessors =
NodeUtil.getSuccessors(currentNode.getState());
           for (String n : nodeSuccessors) {
              if (stateSets.contains(n))
                 continue;
              stateSets.add(n);
              NodeMchild = new NodeM (n);
              currentNode.addChild(child);
              child.setParent(currentNode);


              child.setTotalCost(0, heuristicOne(child.getState(), goalSate));
              nodePriorityQueue.add(child);


           }
           currentNode = nodePriorityQueue.poll();
           time += 1;
        }
```

```
        NodeUtil.printSolution(currentNode, stateSets, root, time);


    }
}
class NodeUtil {
    public static List<String> getSuccessors(String state) {
        List<String> successors = new ArrayList<String>();


        switch (state.indexOf("0")) {
            case 0: {
                successors.add(state.replace(state.charAt(0),").replace(state.charAt(1),
state.charAt(0)).replace(", state.charAt(1)));
                successors.add(state.replace(state.charAt(0),").replace(state.charAt(3),
state.charAt(0)).replace(", state.charAt(3)));
                break;
            }
            case 1: {
                successors.add(state.replace(state.charAt(1),
").replace(state.charAt(0), state.charAt(1)).replace(", state.charAt(0)));
                successors.add(state.replace(state.charAt(1),
").replace(state.charAt(2), state.charAt(1)).replace(", state.charAt(2)));
                successors.add(state.replace(state.charAt(1),
").replace(state.charAt(4), state.charAt(1)).replace(", state.charAt(4)));
                break;
            }
            case 2: {


                successors.add(state.replace(state.charAt(2),
").replace(state.charAt(1), state.charAt(2)).replace(", state.charAt(1)));
```

```
            successors.add(state.replace(state.charAt(2),
").replace(state.charAt(5), state.charAt(2)).replace(", state.charAt(5)));
            break;
        }
        case 3: {
            successors.add(state.replace(state.charAt(3),
").replace(state.charAt(0), state.charAt(3)).replace(", state.charAt(0)));
            successors.add(state.replace(state.charAt(3),
").replace(state.charAt(4), state.charAt(3)).replace(", state.charAt(4)));
            successors.add(state.replace(state.charAt(3),
").replace(state.charAt(6), state.charAt(3)).replace(", state.charAt(6)));
            break;
        }
        case 4: {
            successors.add(state.replace(state.charAt(4),
").replace(state.charAt(1), state.charAt(4)).replace(", state.charAt(1)));
            successors.add(state.replace(state.charAt(4),
").replace(state.charAt(3), state.charAt(4)).replace(", state.charAt(3)));
            successors.add(state.replace(state.charAt(4),
").replace(state.charAt(5), state.charAt(4)).replace(", state.charAt(5)));
            successors.add(state.replace(state.charAt(4),
").replace(state.charAt(7), state.charAt(4)).replace(", state.charAt(7)));
            break;
        }
        case 5: {
            successors.add(state.replace(state.charAt(5),
").replace(state.charAt(2), state.charAt(5)).replace(", state.charAt(2)));
            successors.add(state.replace(state.charAt(5),
").replace(state.charAt(4), state.charAt(5)).replace(", state.charAt(4)));
            successors.add(state.replace(state.charAt(5),
").replace(state.charAt(8), state.charAt(5)).replace(", state.charAt(8)));
```

```
            break;
        }
        case 6: {
            successors.add(state.replace(state.charAt(6),
").replace(state.charAt(3), state.charAt(6)).replace(", state.charAt(3)));
            successors.add(state.replace(state.charAt(6),
").replace(state.charAt(7), state.charAt(6)).replace(", state.charAt(7)));
            break;

        }
        case 7: {
            successors.add(state.replace(state.charAt(7),
").replace(state.charAt(4), state.charAt(7)).replace(", state.charAt(4)));
            successors.add(state.replace(state.charAt(7),
").replace(state.charAt(6), state.charAt(7)).replace(", state.charAt(6)));
            successors.add(state.replace(state.charAt(7),
").replace(state.charAt(8), state.charAt(7)).replace(", state.charAt(8)));
            break;
        }
        case 8: {
            successors.add(state.replace(state.charAt(8),").replace(state.charAt(5),
state.charAt(8)).replace(", state.charAt(5)));
            successors.add(state.replace(state.charAt(8),
").replace(state.charAt(7), state.charAt(8)).replace(", state.charAt(7)));
            break;
        }
    }
    return successors;
  }
```

```
   public static void printSolution(NodeMgoalNode, Set<String> visitedNodes,
NodeMroot, int time) {
        int totalCost = 0;
        Stack< NodeM > solutionStack = new Stack< NodeM >();
        solutionStack.push(goalNode);
        while (!goalNode.getState().equals(root.getState())) {
           solutionStack.push(goalNode.getParent());
           goalNode = goalNode.getParent();
        }
        String sourceState = root.getState();
        String destinationState;
        int cost = 0;
        for (int i = solutionStack.size() - 1; i >= 0; i--) {
           System.out.println("----------------------------------------------------------------
----");
           destinationState = solutionStack.get(i).getState();
           if (!sourceState.equals(destinationState)) {
              System.out.println("Move " +
destinationState.charAt(sourceState.indexOf('0')) + " " +
findTransition(sourceState, destinationState));
              cost =
Character.getNumericValue(destinationState.charAt(sourceState.indexOf('0')));
              totalCost += cost;
           }

           sourceState = destinationState;
           System.out.println("Cost of the movement: " + cost);
           System.out.println("*");
```

```
            System.out.println("* " + solutionStack.get(i).getState().substring(0,
3)+" *");
            System.out.println("* " + solutionStack.get(i).getState().substring(3,
6)+" *");
            System.out.println("* " + solutionStack.get(i).getState().substring(6,
9)+" *");
            System.out.println("*");


        }
        System.out.println("----------------------------------------------------------------
-");
        System.out.println("** Number of transitions to get to the goal state from
the initial state:  " + (solutionStack.size() - 1));
        System.out.println("** Number of visited states:  " +
(visitedNodes.size()));
        System.out.println("** Total cost for this solution: " + totalCost);
        System.out.println("** Number of Nodes poped out of the queue: " +
time);
        System.out.println("----------------------------------------------------------------
-");


    }


    public static MovementType findTransition(String source, String destination)
{
        int zero_position_difference = destination.indexOf('0') -
source.indexOf('0');
        switch (zero_position_difference) {
            case -3:
                return MovementType.DOWN;
            case 3:
```

```
                return MovementType.UP;
            case 1:
                return MovementType.LEFT;
            case -1:
                return MovementType.RIGHT;
        }
        return null;
    }
}
class NodeM{
    private boolean visited;

    private String state;
    private ArrayList< NodeM > children;
    private NodeMparent;
    private int cost;
    private int estimatedCostToGoal;
    private int totalCost;
    private int depth;

    public int getDepth() {
        return depth;
    }
    public void setDepth(int depth) {
        this.depth = depth;
    }
    public boolean isVisited() {
```

```
    return visited;
}
public void setVisited(boolean visited) {
    this.visited = visited;
}
public int getTotalCost() {
    return totalCost;
}
public void setTotalCost(int totalCost) {
    this.totalCost = totalCost;
}
public void setTotalCost(int cost, int estimatedCost) {
    this.totalCost = cost + estimatedCost;
}
public int getEstimatedCostToGoal() {
    return estimatedCostToGoal;
}
public void setEstimatedCostToGoal(int estimatedCostToGoal) {
    this.estimatedCostToGoal = estimatedCostToGoal;
}
public int getCost() {
    return cost;
}
public void setCost(int cost) {
    this.cost = cost;
}
public void setState(String state) {
```

```
        this.state = state;
    }
    public NodeMgetParent() {
        return parent;
    }
    public void setParent(NodeMparent) {
        this.parent = parent;
    }
    public NodeM (String state) {
        this.state = state;
        children = new ArrayList<NodeM >();
    }
    public String getState() {
        return state;
    }
    public ArrayList< NodeM > getChildren() {
        return children;
    }
    public void addChild(NodeMchild) {
        children.add(child);
    }
}
enum MovementType {
    UP,
    DOWN,
    LEFT,
    RIGHT;
```

```
}
class NodePriorityComparator implements Comparator< NodeM > {
    public int compare(NodeMx, NodeMy) {
        if (x.getTotalCost() < y.getTotalCost()) {
            return -1;
        }
        if (x.getTotalCost() > y.getTotalCost()) {
            return 1;
        }
        return 0;
    }
}
```