ITCS 6150 – Intelligent Systems Project Report
Title: 8 Puzzle Problem Solution using A* Search
Name: Sourav Roy Choudhury
ID- 801100959
Date: 02-11-2019

Source Code

```
package firstIS;

import java.util.Arrays;
import java.util.Scanner;

public class MispacedTilesUsingA {

	static int[] goalStateArray;
	static int heuristicChoice;
	public static void main(String args[]) {
		Calculate8Puzzle calculate8Puzzle=new Calculate8Puzzle();

		System.out.println(" Enter Heuristic function");
		System.out.println("****************************");
		System.out.println("1) Manhattan 2) Misplaced tiles");

		Scanner reader = new Scanner(System.in);

		heuristicChoice = Integer.parseInt(reader.nextLine());

		System.out
				.println("Enter Initial State - Use space after every number
(example : 0 1 2 3 4 5 6 7 8): \n");
		reader = new Scanner(System.in);
		int[] initialStateArray = getArrayFromInputOutput(reader.nextLine()
				.split(" "));
		System.out
				.println("Enter goal State - Use space after every number
(example : 0 1 2 3 4 5 6 7 8) : \n");
		reader = new Scanner(System.in);
```

```
			goalStateArray = getArrayFromInputOutput(reader.nextLine().split(
					" "));
			if(Arrays.equals(initialStateArray, goalStateArray)){
				System.out.println("Goal State Reached...");
				System.exit(0);

			}
			calculate8Puzzle.calculateData(goalStateArray,initialStateArray);



		}



		private static int[] getArrayFromInputOutput(String[] a) {
			int[] initState = new int[9];
			for (int i = 0; i < a.length; i++) {
				initState[i] = Integer.parseInt(a[i]);
			}
			return initState;
		}

}




package firstIS;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Stack;

public class Calculate8Puzzle {
		MovePosition position = new MovePosition();
		NodeDetail GoalNode = new NodeDetail();
		int Nodes_generated = 0;
		public void calculateData(int[] goalStateArray,int[] initialStateArray)
```

```
	{
		ArrayList<Integer> initialState = new ArrayList<>();
		NodeDetail[] stateNode = new NodeDetail[4];
		NodeDetail goalNodeTrack = new NodeDetail();
		goalNodeTrack = null;
		Stack stack = new Stack();
		NodeDetail current = new NodeDetail();
		LinkedList<ArrayList<?>> visitElement = new LinkedList<ArrayList<?>>();
		int nodeCount = 0;

		// Creating start node
		NodeDetail startNode = new NodeDetail();
		for (int i : initialStateArray) {
			initialState.add(i);
		}
		startNode.move = null;
		startNode.priority = 0;
		startNode.nodeState = initialState;
		startNode.dist = 0;
		startNode.parent = null;

		// Creating goal node
		NodeDetail goalNode = new NodeDetail();
		ArrayList<Integer> goalState = new ArrayList<>();

		for (int i : goalStateArray) {
			goalState.add(i);
		}
		// Creating goal node
		goalNode.nodeState = goalState;
		goalNode.parent = null;
		goalNode.dist = 0;
		goalNode.move = null;

		Comparator<NodeDetail> priorityCompare = new CompareNode();
		PriorityQueue<NodeDetail> pQ = new PriorityQueue<NodeDetail>(100,
priorityCompare);
		pQ.add(startNode);
		Nodes_generated++;
		visitElement.add(startNode.nodeState);

		int depth=0;
		while (!pQ.isEmpty()) {
			nodeCount++;
```

```
current = pQ.remove();
visitElement.add(current.nodeState);

// goal test when you remove
if (current.nodeState.equals(goalNode.nodeState)) {
        goalNodeTrack = current;
        GoalNode = current;
        break;
}
// if the current node is not goal
stateNode = findMove(current);


for (int i = 0; i <= 3; i++) {

                        if(stateNode[i] == null)
                                continue;
                        // check in the explored nodes
                        if (!visitElement.contains(stateNode[i].nodeState))
{

                                stateNode[i].dist = current.dist + 1;
                                depth=stateNode[i].dist;
                                if (MispacedTilesUsingA.heuristicChoice ==
1)
                                        stateNode[i].priority =
calculateManhattanDistance(stateNode[i].nodeState, MispacedTilesUsingA.goalStateArray);
                                else if(MispacedTilesUsingA.heuristicChoice
== 2)
                                        stateNode[i].priority =
misplacedTiles(stateNode[i], goalNode);

                                // check in the frontier. add only if the state
is not in the frontier
                                if (!pQ.contains(stateNode[i]))
                                {
//                              visitElement.add(stateNode[i].nodeState);
                                //stateNode[i].priority =
misplacedTiles(stateNode[i], goalNode);

        System.out.println(Arrays.toString(stateNode[i].nodeState.toArray()));
                                pQ.add(stateNode[i]);
                                Nodes_generated++;
                                }
```

```
                                                    else if(pQ.contains(stateNode[i])){

                                                            // if the frontier queue has the
current node. check if the current node has the least value and if it has least value replace it
with the frontier element.
                                                            pQ.add(stateNode[i]);
                                                            Nodes_generated++;
                                                    }


                                            }
                                    }


                    if (goalNodeTrack != null)
                            break;


            }

            while (goalNodeTrack.parent != null) {
                    if (goalNodeTrack.move != null) {
                            stack.push(goalNodeTrack.move);
                    }
                    goalNodeTrack = goalNodeTrack.parent;
            }
            while(!stack.isEmpty()){
                    System.out.println(stack.pop());
            }
            System.out.println("Final count f(n)"+depth);

            // backtrack the goal node to the initial node
            List<ArrayList<Integer>> goalsequences = new ArrayList<ArrayList<Integer>>();
            NodeDetail currentnode = GoalNode;
            do{

                    goalsequences.add(currentnode.nodeState);
                    currentnode = currentnode.parent;
            }while(currentnode != null);

            // print the solution as matrix

            int sizeofallsequence = goalsequences.size() - 1;
            for(int x = sizeofallsequence; x >= 0 ; x--){
```

```
                ArrayList<Integer> sequence1 = goalsequences.remove(x);

                int sequence1size = sequence1.size();
                for(int i = 0 ; i < sequence1size ; i++){

                        if((i%3)==0){
                                System.out.println("");
                        }

                        System.out.print(sequence1.get(i) + "\t");
                }
                System.out.println("");
        }
        //System.out.println("Total nodes traversed"+nodeCount);

        System.out.println("Total nodes explored :"+nodeCount);
        System.out.println("Total nodes generated: " +Nodes_generated);
    }

    private static int misplacedTiles(NodeDetail node, NodeDetail goal) {
        //method stub

        int priority;
        int count = 0;

        //Heuristic Function Calculation

        for (int i = 0; i < 9; i++) {

                if (node.nodeState.get(i) != goal.nodeState.get(i)) {
                        if(node.nodeState.get(i) == 0)
                                continue;
                        count++;
                }
        }

        priority = node.dist + count;
        return priority;
    }

    //Manhattan
    //Below function calculates the Manhattan distance(heuristic value) for each
//state or node. I.e the sum of the distances of the tiles from their goal
```

```
    //positions

    private int calculateManhattanDistance(ArrayList<Integer> state_array, int[] array2){
            int n = state_array.size();
            int sum = 0;
            for (int j = 0; j < n ; j++) {
                    int x = j/3;
                            int y = j%3;
                            int[] location = checkPosition(state_array.get(j));
                            sum += (Math.abs(x - location[0]) +  Math.abs(y - location[1]));
                    }
            return sum;
    }

    public static int[] checkPosition(int element) {
       int[] location = new int[2];
       for (int i = 0; i < MispacedTilesUsingA.goalStateArray.length; ++i) {
             if (MispacedTilesUsingA.goalStateArray[i] == element) {
                location[0] = i/3;
                location[1] = i%3;
           }
       }
       return location;
    }

            public NodeDetail[] findMove(NodeDetail state) {

                    NodeDetail state1, state2, state3, state4;

                    state1 = position.Up(state);
                    state2 = position.Down(state);
                    state3 = position.Left(state);
                    state4 = position.Right(state);

                    NodeDetail[] states = { state1, state2, state3, state4 };

                    return states;
            }


}
```

```
package firstIS;

import java.util.ArrayList;

public class MovePosition {
	public NodeDetail Right(NodeDetail nodeTrack) {
		// method stub
		int space = nodeTrack.nodeState.indexOf(0);
		ArrayList<Integer> childState;
		int temp;
		NodeDetail childNode = new NodeDetail();

		if (space != 2 && space != 5 && space != 8) {
			childState = (ArrayList<Integer>) nodeTrack.nodeState.clone();
			temp = childState.get(space + 1);
			childState.set(space + 1, 0);
			childState.set(space, temp);
			childNode.nodeState = childState;
			childNode.parent = nodeTrack;
			childNode.dist = nodeTrack.dist + 1;
			childNode.move = "RIGHT";
			return childNode;
		} else {
			return null;
		}
	}

	public NodeDetail Left(NodeDetail nodeTrack) {
		//method stub
		int space = nodeTrack.nodeState.indexOf(0);
		ArrayList<Integer> childState;
		int temp;
		NodeDetail childNode = new NodeDetail();

		if (space != 0 && space != 3 && space != 6) {
			childState = (ArrayList<Integer>) nodeTrack.nodeState.clone();
			temp = childState.get(space - 1);
			childState.set(space - 1, 0);
			childState.set(space, temp);
			childNode.nodeState = childState;
```

```
                childNode.parent = nodeTrack;
                childNode.dist = nodeTrack.dist + 1;
                childNode.move = "LEFT";
                return childNode;
        } else {
                return null;
        }
}

public NodeDetail Down(NodeDetail nodeTrack) {
        //method stub
        int space = nodeTrack.nodeState.indexOf(0);
        ArrayList<Integer> childState;
        int temp;
        NodeDetail childNode = new NodeDetail();

        if (space <= 5) {
                childState = (ArrayList<Integer>) nodeTrack.nodeState.clone();
                temp = childState.get(space + 3);
                childState.set(space + 3, 0);
                childState.set(space, temp);
                childNode.nodeState = childState;
                childNode.parent = nodeTrack;
                childNode.dist = nodeTrack.dist + 1;
                childNode.move = "DOWN";
                return childNode;
        } else {
                return null;
        }
}

public NodeDetail Up(NodeDetail node) {
        //method stub
        int space = node.nodeState.indexOf(0);
        ArrayList<Integer> childState;
        int temp;
        NodeDetail childNode = new NodeDetail();

        if (space > 2) {
                childState = (ArrayList<Integer>) node.nodeState.clone();
```

```
                    temp = childState.get(space - 3);
                    childState.set(space - 3, 0);
                    childState.set(space, temp);
                    childNode.nodeState = childState;
                    childNode.parent = node;
                    childNode.dist = node.dist + 1;
                    childNode.move = "UP";
                    return childNode;
                } else {
                    return null;
                }
            }

}
```

```
package firstIS;
import java.util.ArrayList;


public class NodeDetail {
        String name;
        ArrayList<Integer> nodeState;
        NodeDetail parent;
        int dist;
        String move;
        public int priority;
        public NodeDetail(String name){
                this.name = name;
        }

        public NodeDetail(){

        }


        public String getName(){
                return this.name;
```

```
	}
}
```

```
package firstIS;

import java.util.Comparator;


public class CompareNode implements Comparator<NodeDetail> {

	@Override
	public int compare(NodeDetail node1, NodeDetail node2) {
		// method stub
		if (node1.priority > node2.priority){
			return 1;
		}
		if (node1.priority < node2.priority){
			return -1;
		}
		return 0;
	}
}
```