Ankit Pandita

Jinraj Jain

implement 8 puzzle using A\*

**Problem Implementation Details**:

We used Java Programming Language to implement this program.

**Test Cases:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S.No** | **Initial State** | **Goal State** | **Manhattan Distance**  **Heuristic** | | **Misplaced Tiles Heuristic** | |
|  |  |  | **Nodes Generated** | **Nodes Expanded** | **Nodes Generated** | **Nodes Expanded** |
| 1 | 1 2 3  4 8 0  7 6 5 | 1 2 3  4 5 6  7 8 0 | 12 | 6 | 18 | 9 |
| 2 | 1. 8 1 2. 4 6   7 5 0 | 3 2 1  8 0 4  7 5 6 | 13 | 7 | 15 | 8 |
| 3 | 4 1 3  0 2 6  7 5 8 | 1 2 3  4 5 6  7 8 0 | 12 | 6 | 12 | 6 |
| 4 | 1 2 3  7 4 5  6 8 0 | 1 2 3  8 6 4  7 5 0 | 19 | 10 | 42 | 22 |

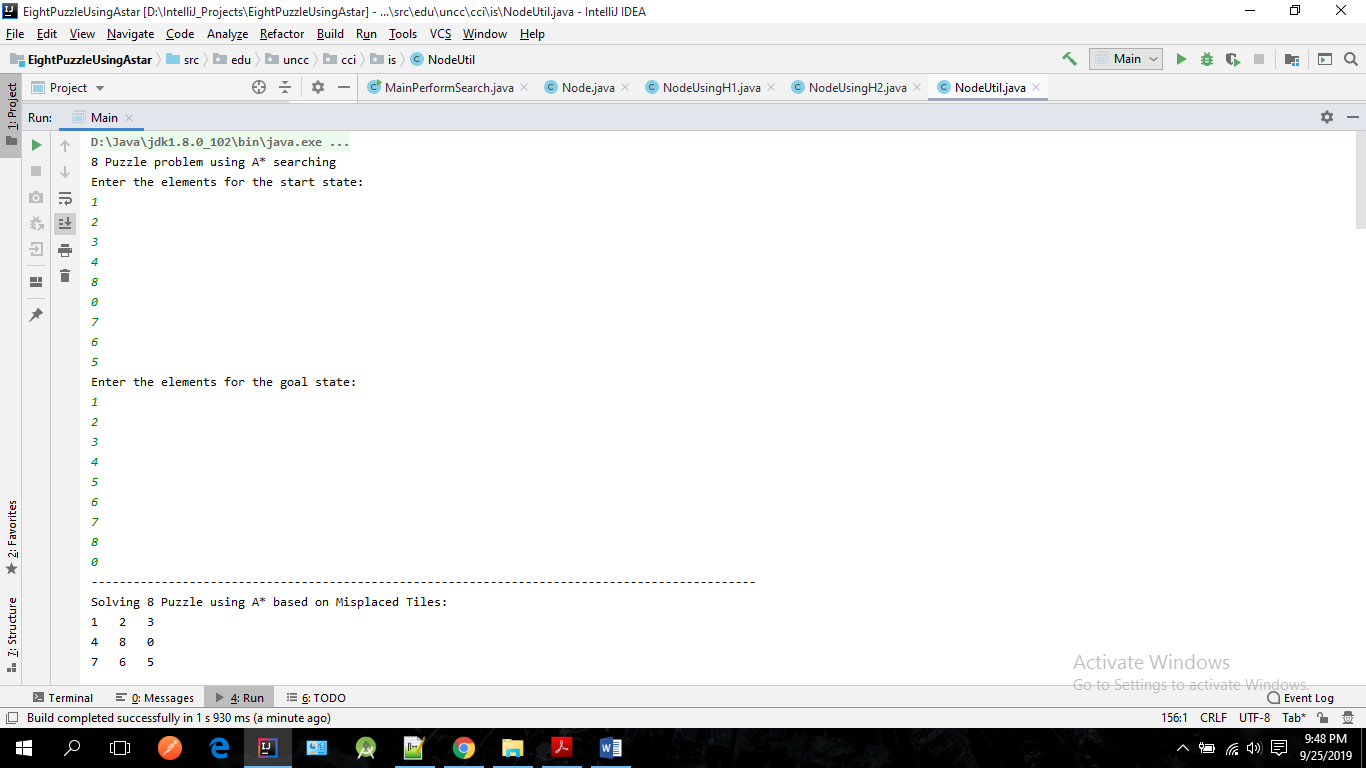
# **Test Case 1:**

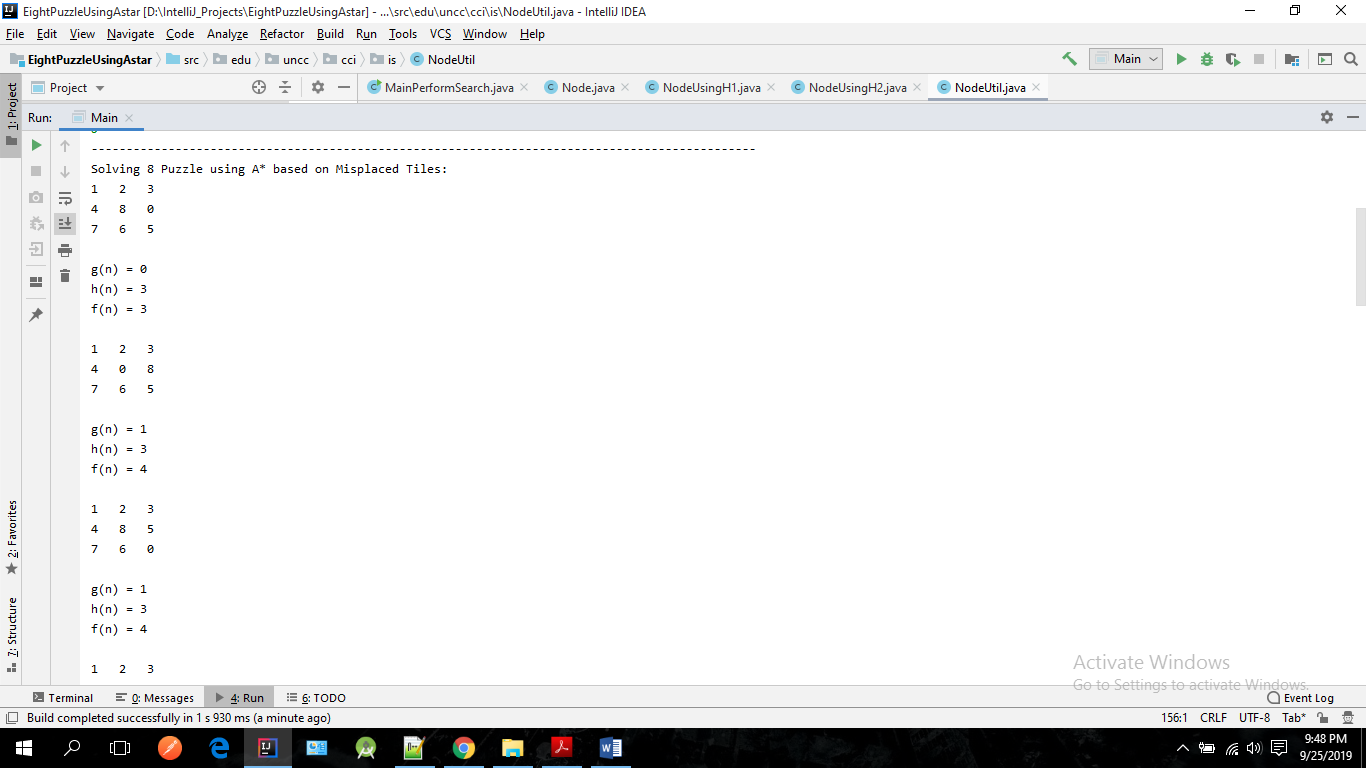
Enter the elements for the start state of the puzzle ==>

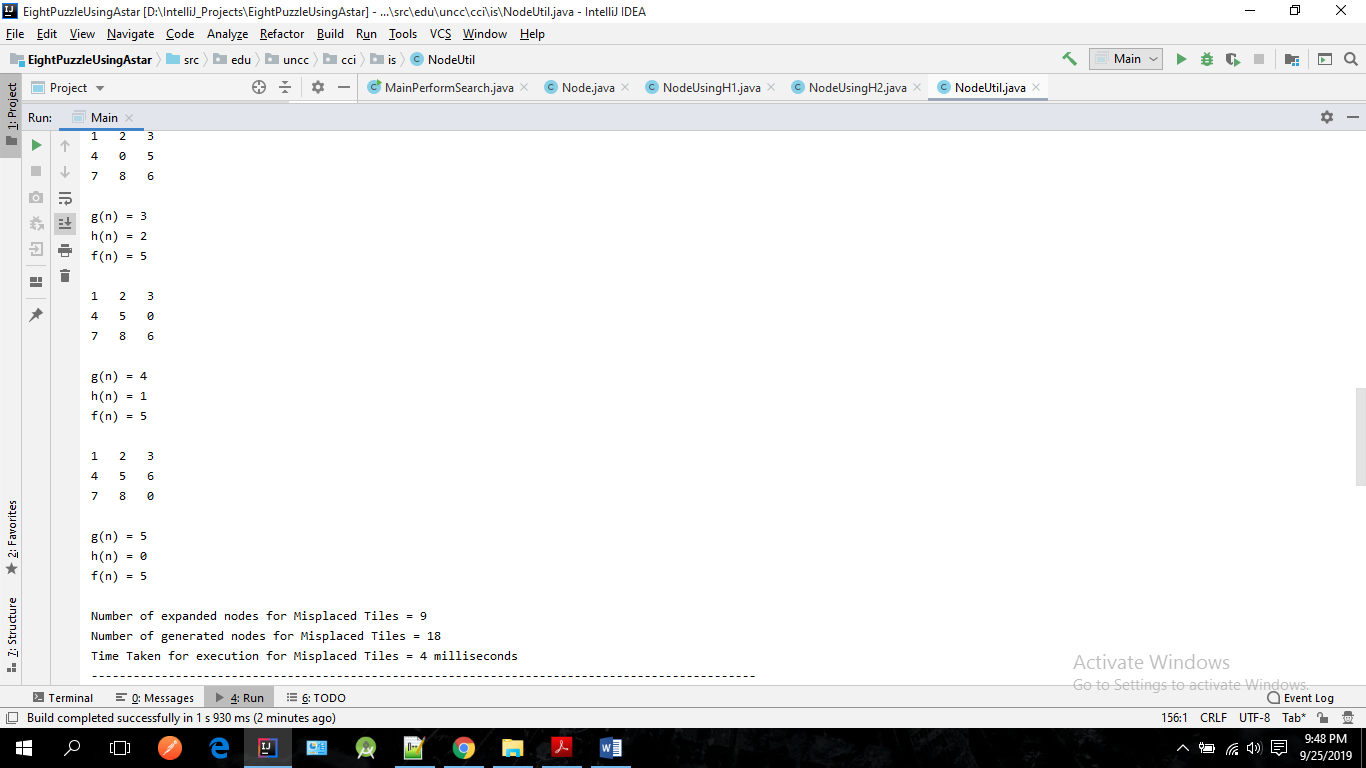
1 2 3 4 8 0 7 6 5

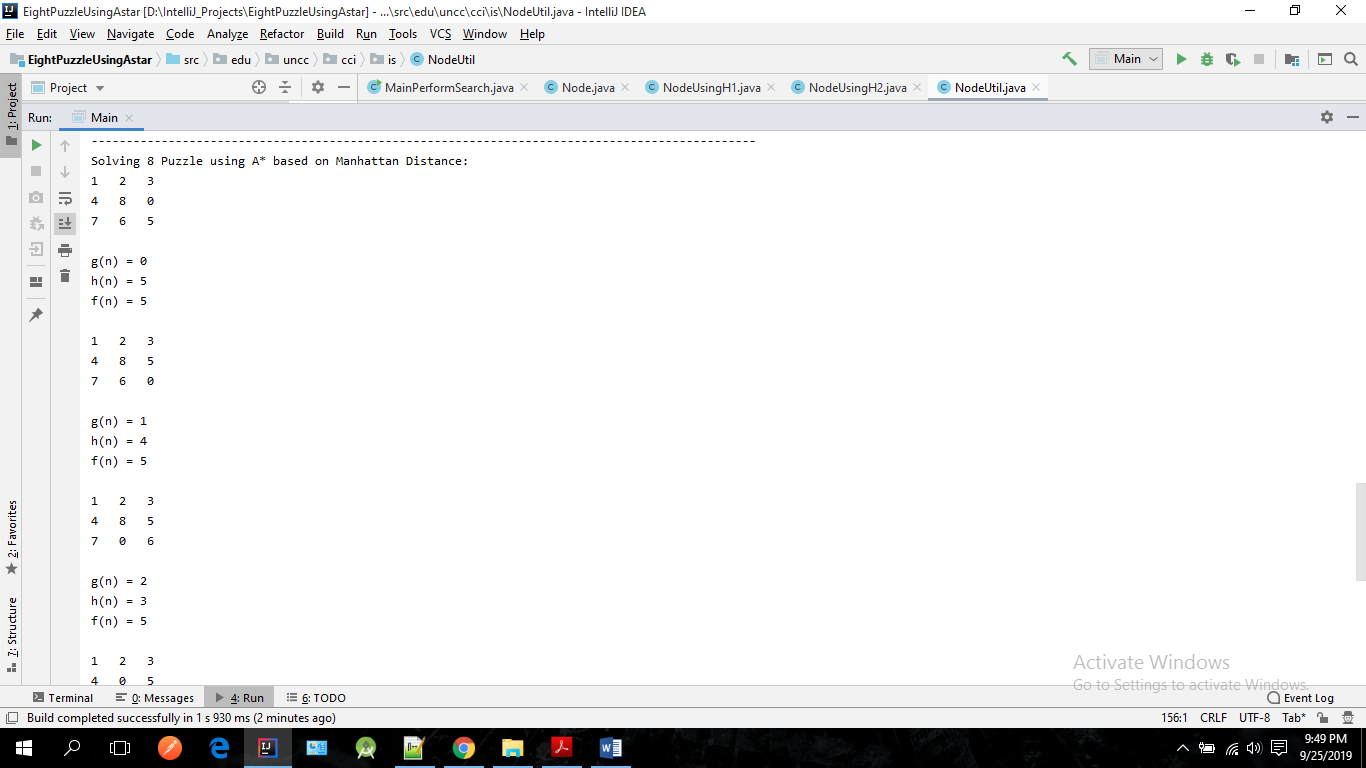
Enter the elements for the goal state of the puzzle ==>

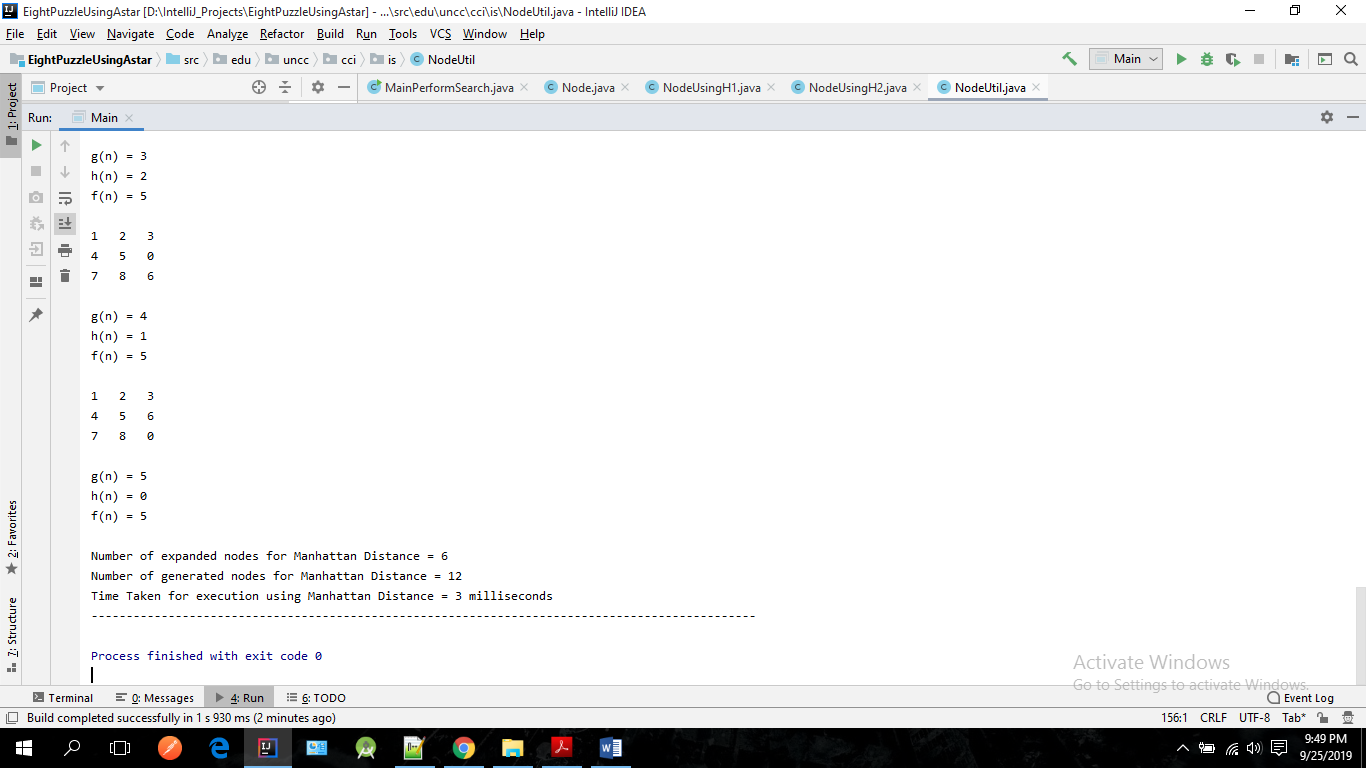
1 2 3 4 5 6 7 8 0







****

****

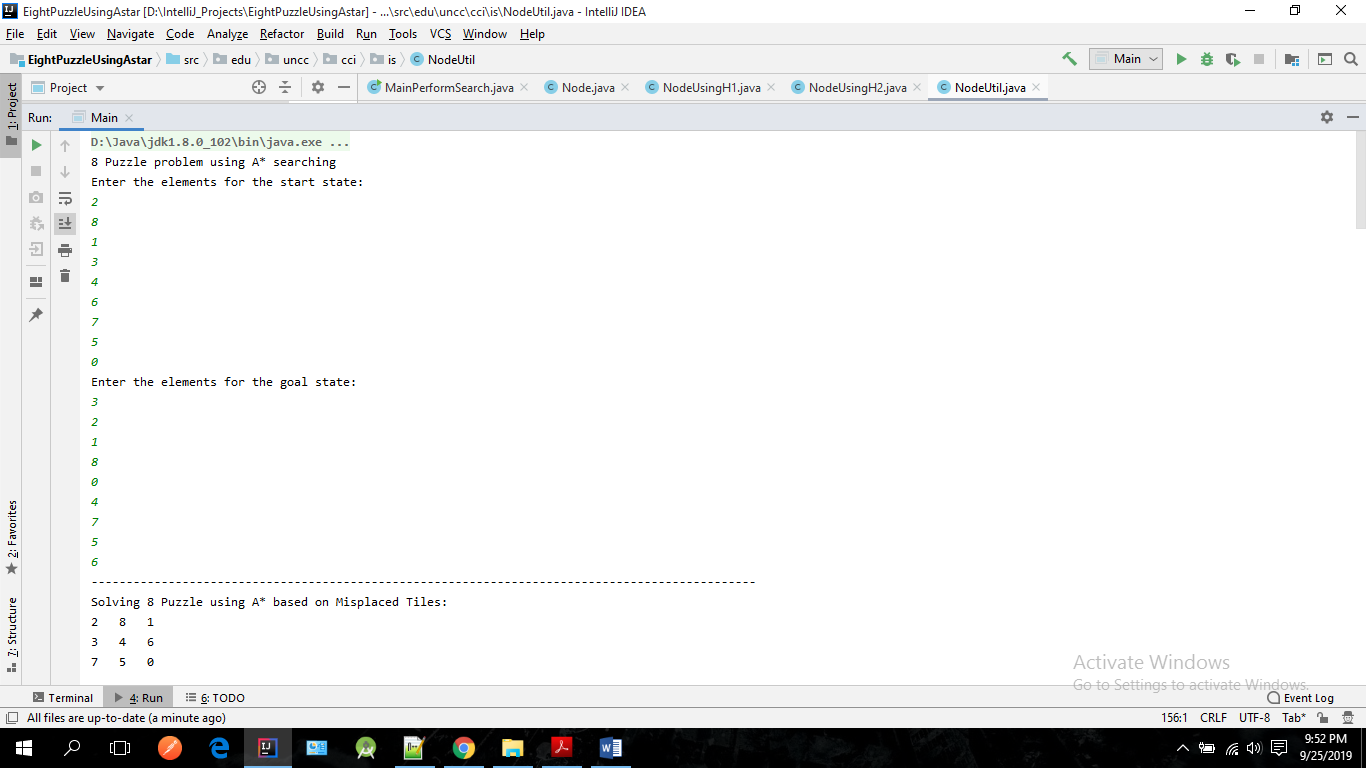
# **Test Case 2:**

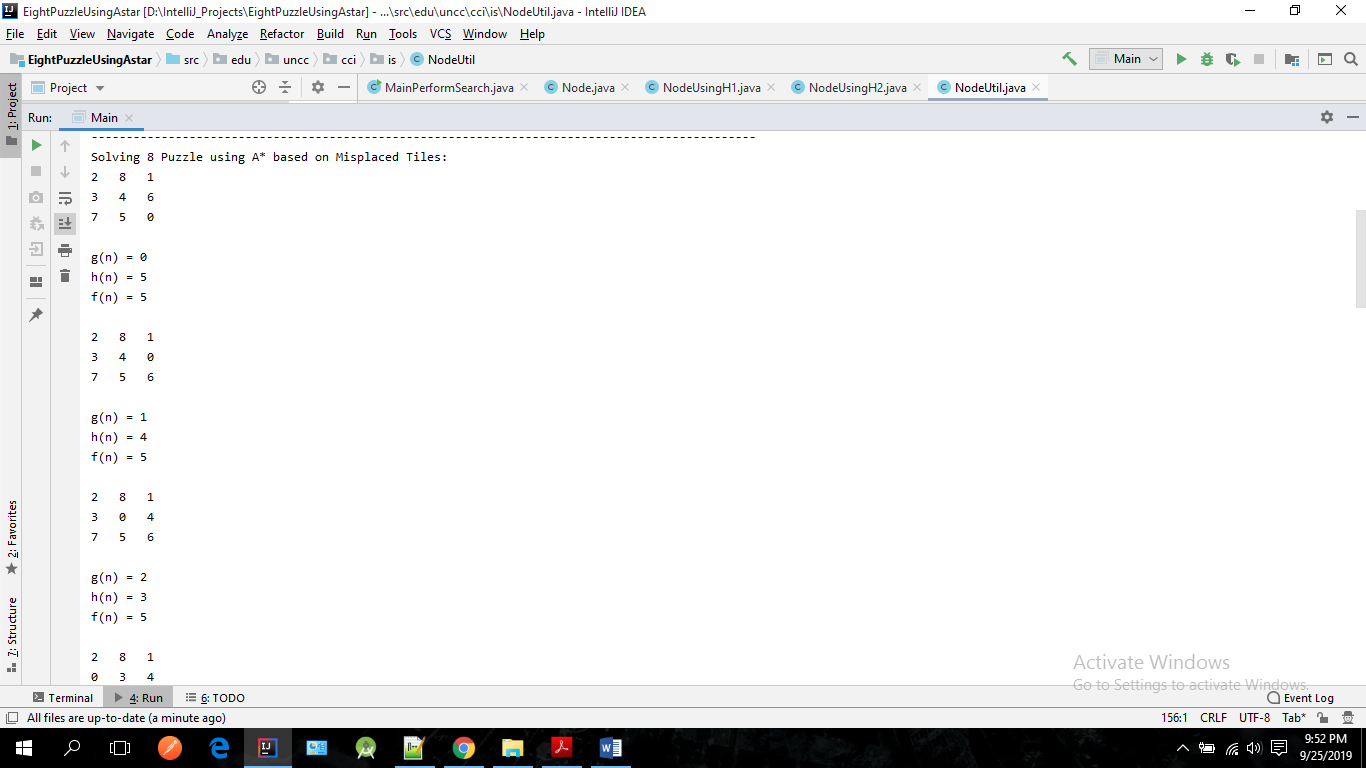
Enter the elements for the start state of the puzzle ==>

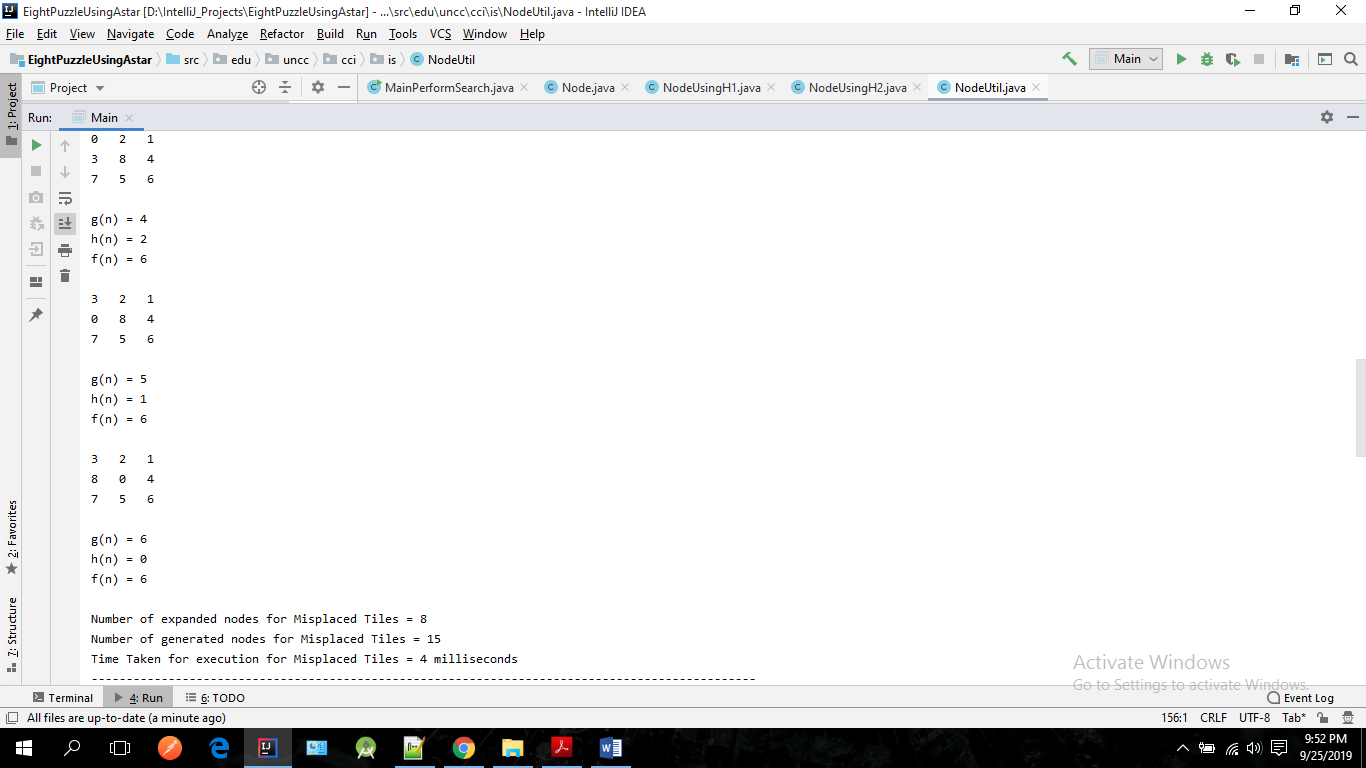
2 8 1 3 4 6 7 5 0

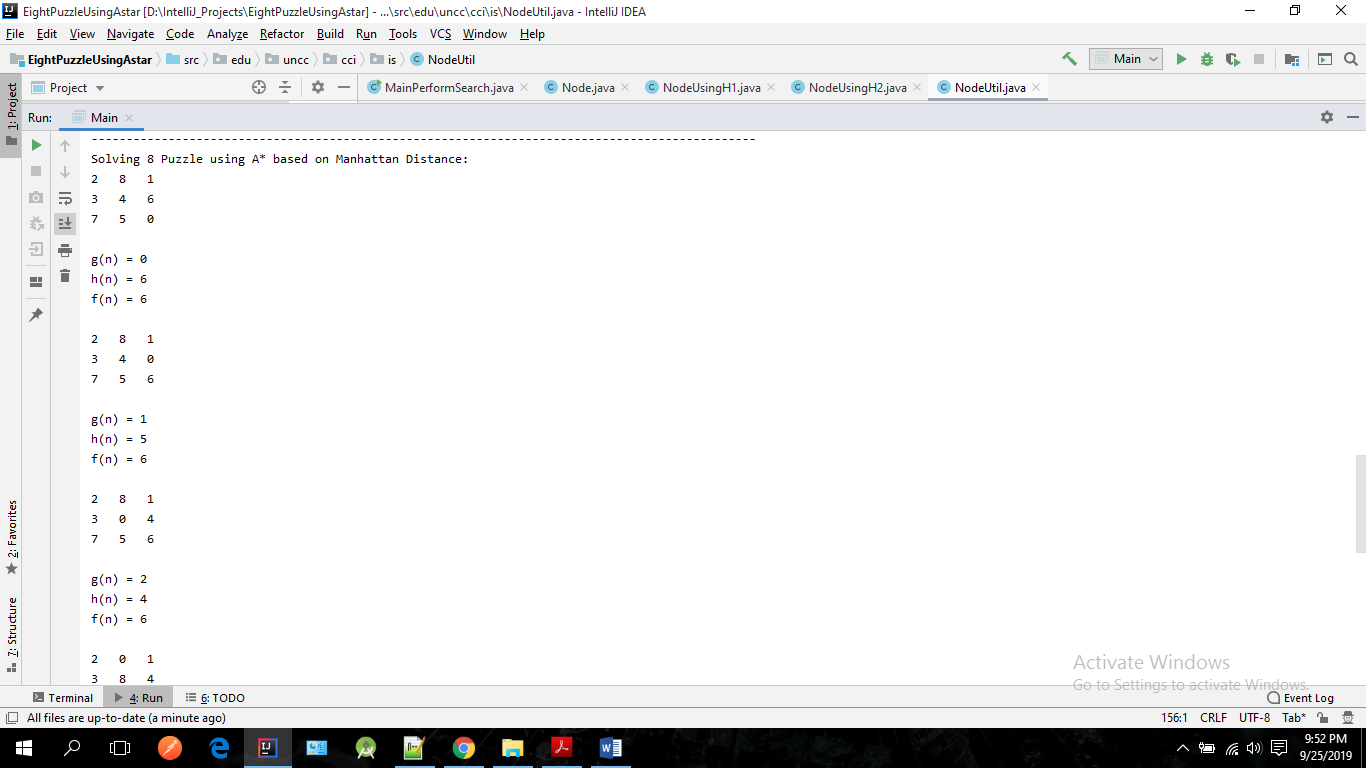
Enter the elements for the goal state of the puzzle ==>

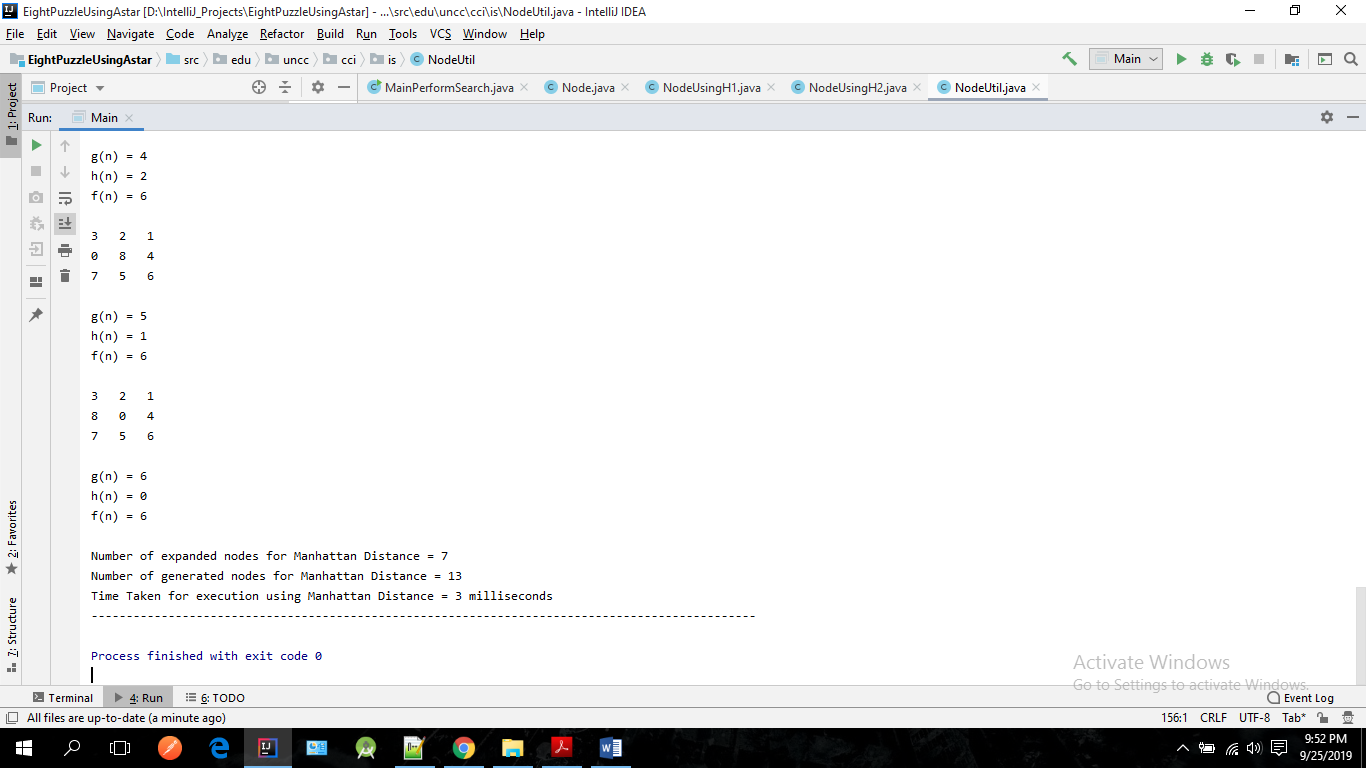
3 2 1 8 0 4 7 5 6











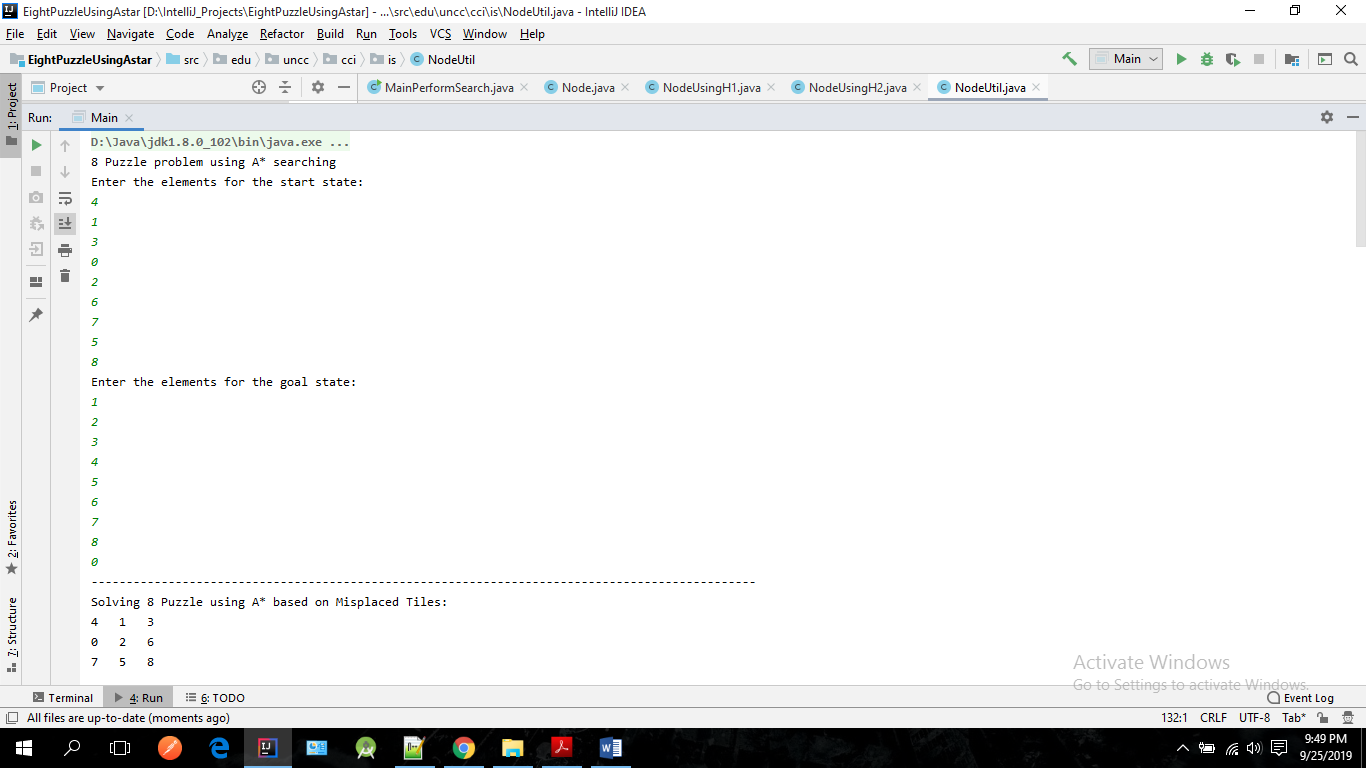
**Test Case 3:**

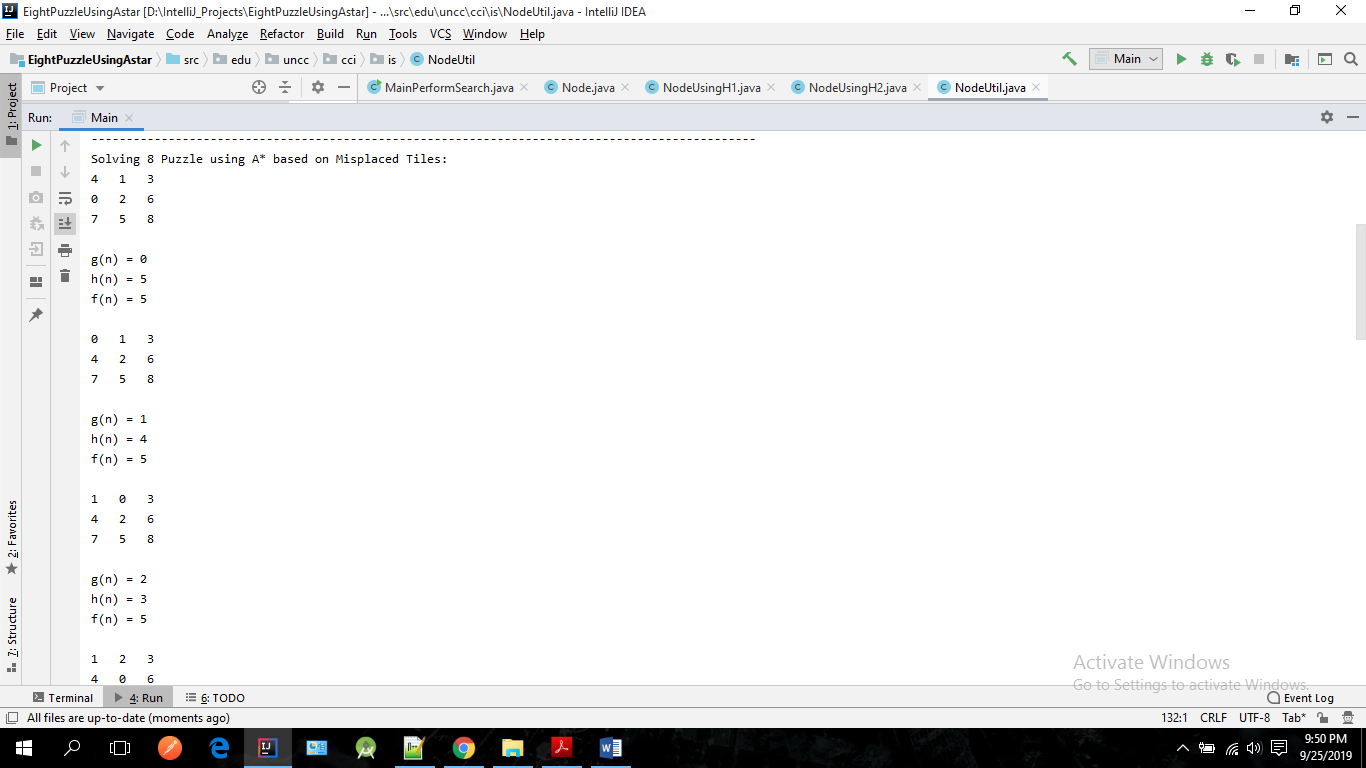
Enter the elements for the start state of the puzzle ==>

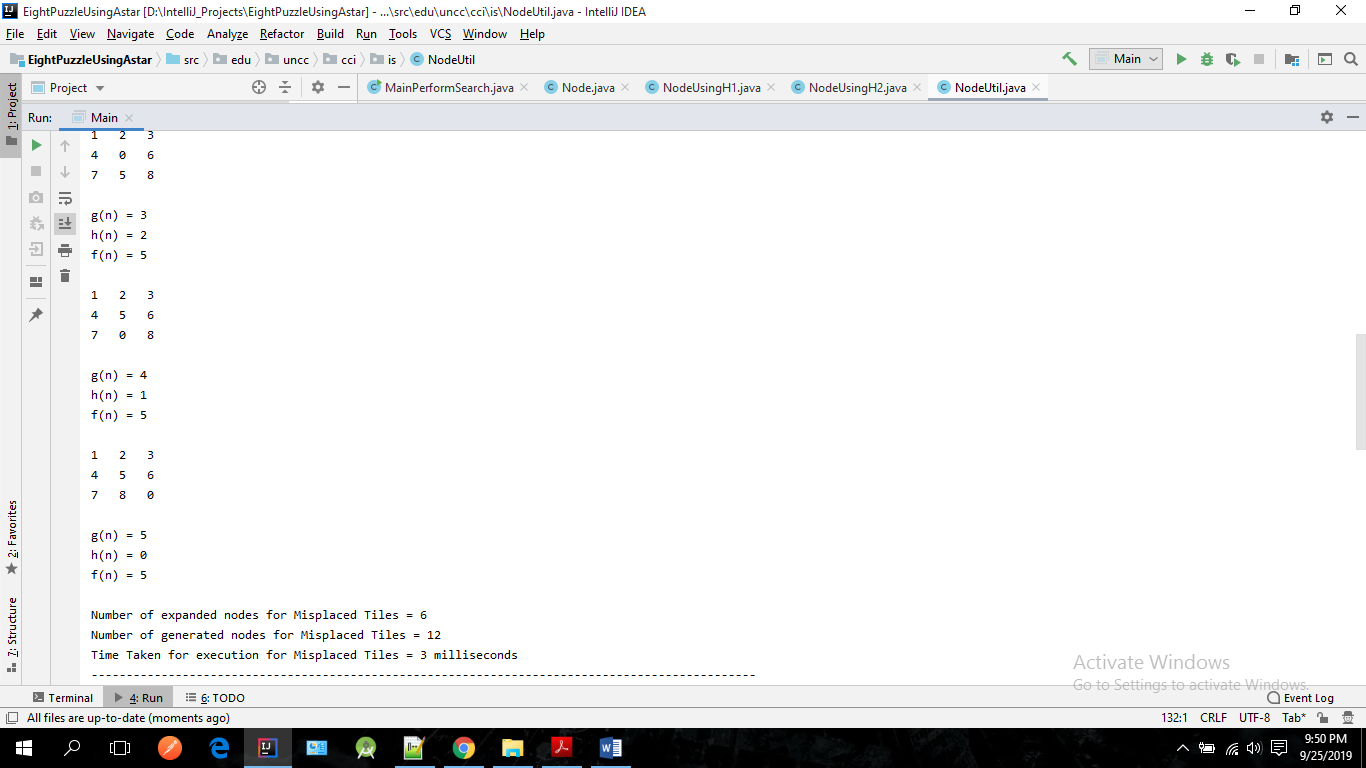
4 1 3 0 2 6 7 5 8

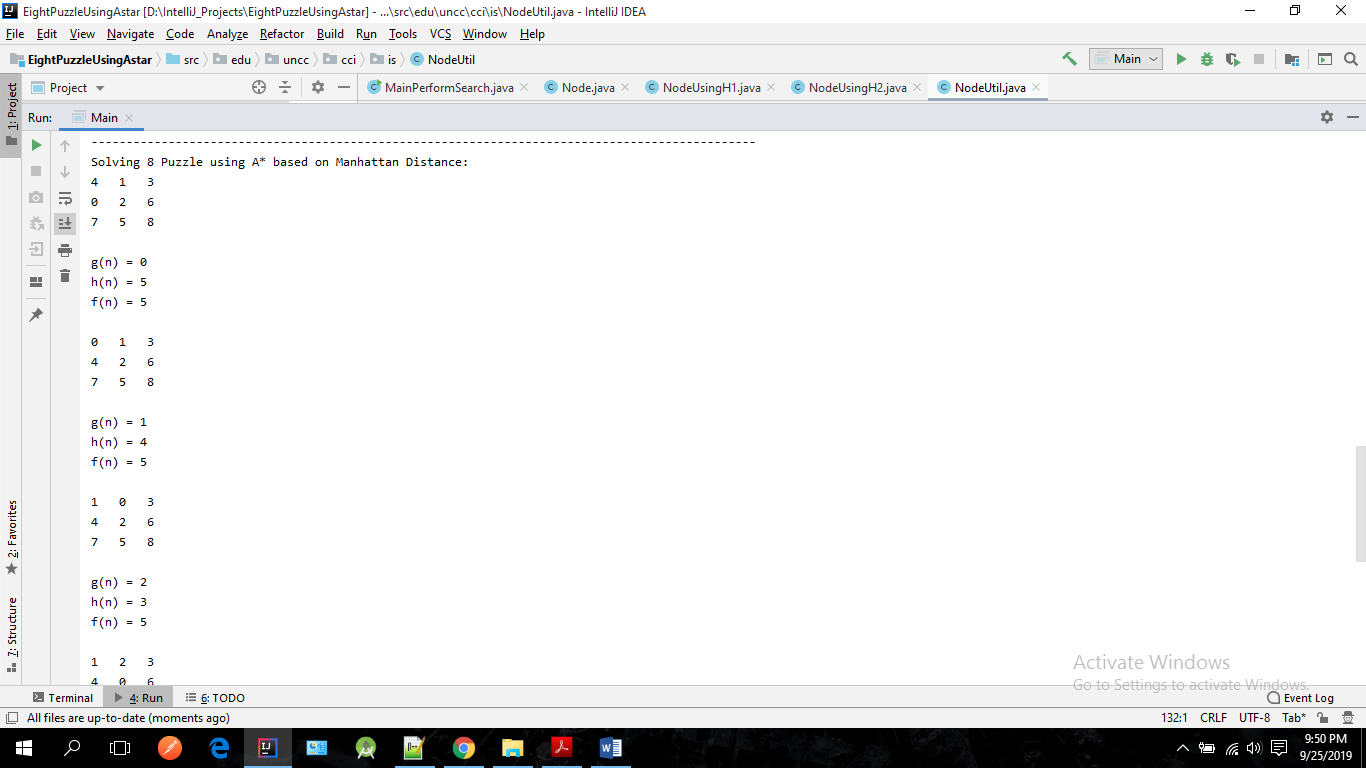
Enter the elements for the goal state of the puzzle ==>

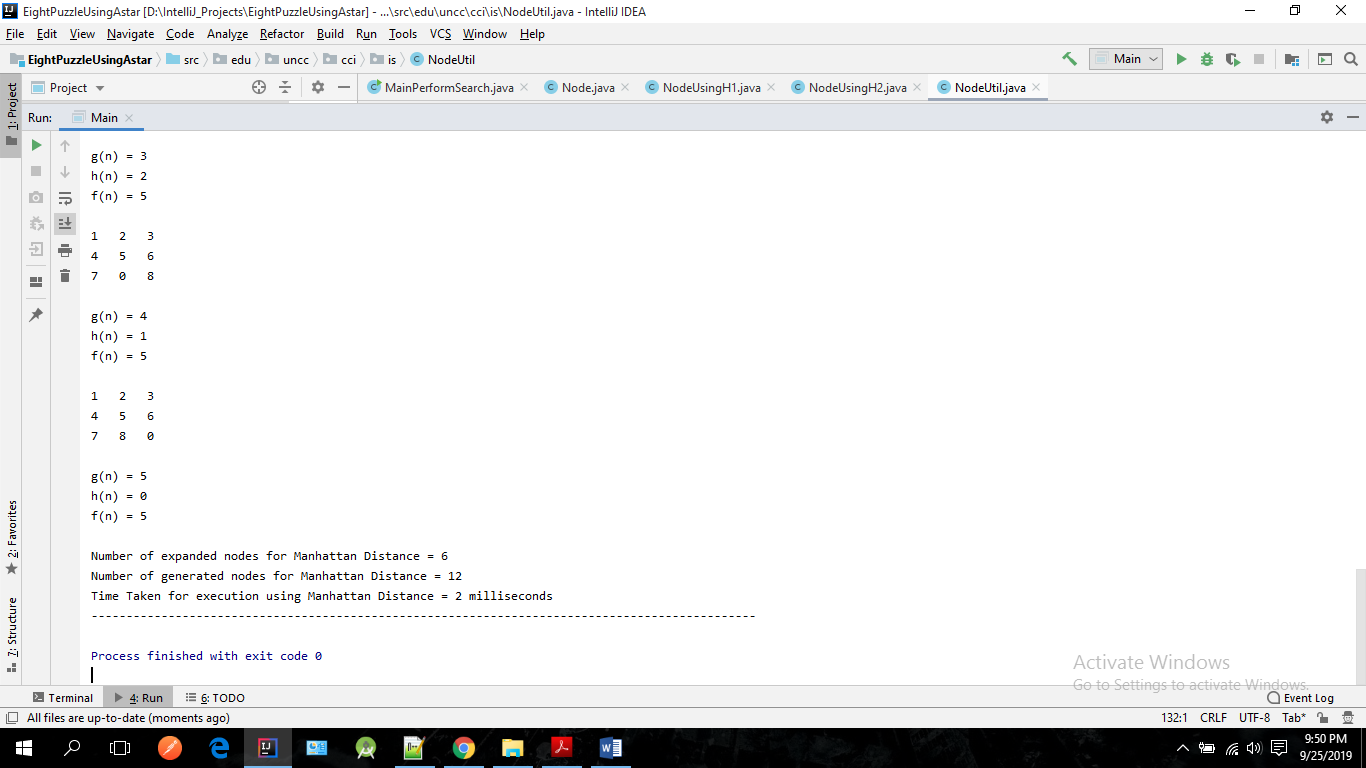
1 2 3 4 5 6 7 8 0











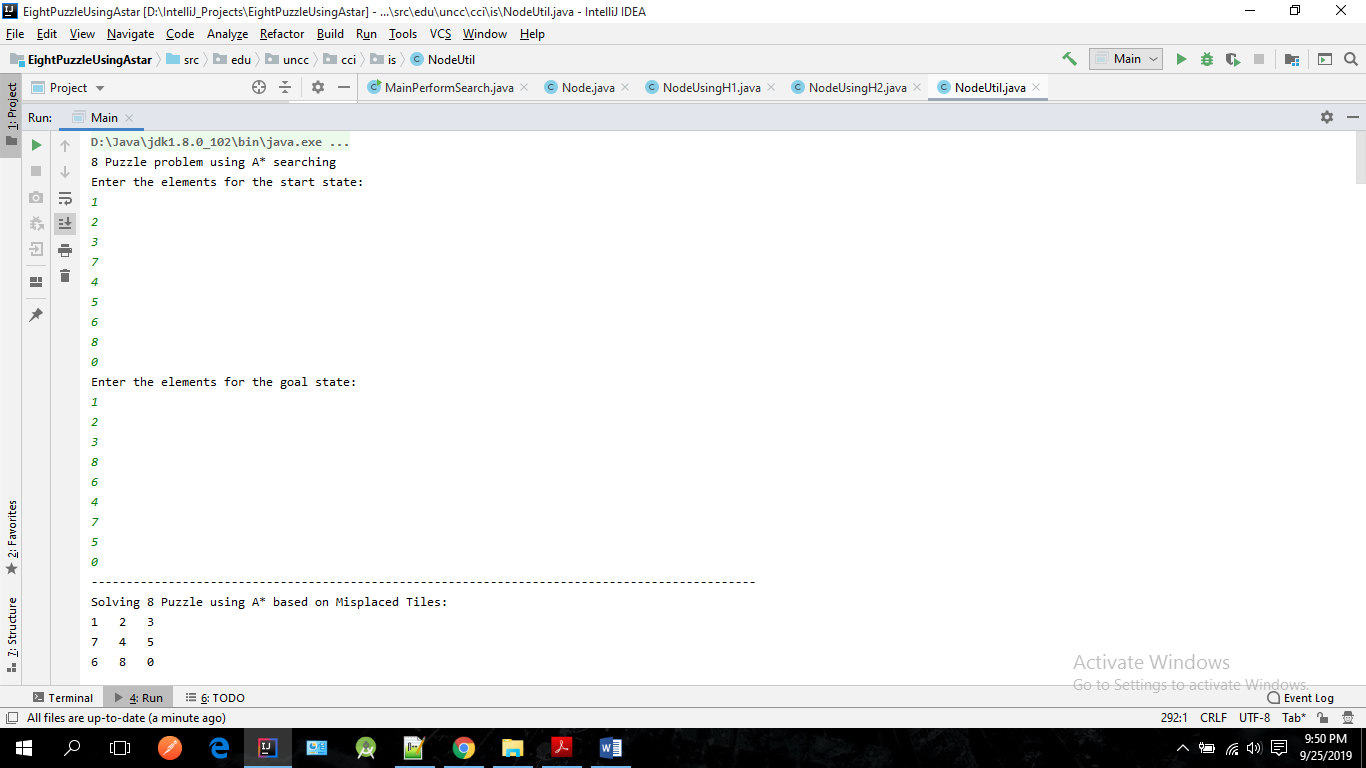
**Test Case 4:**

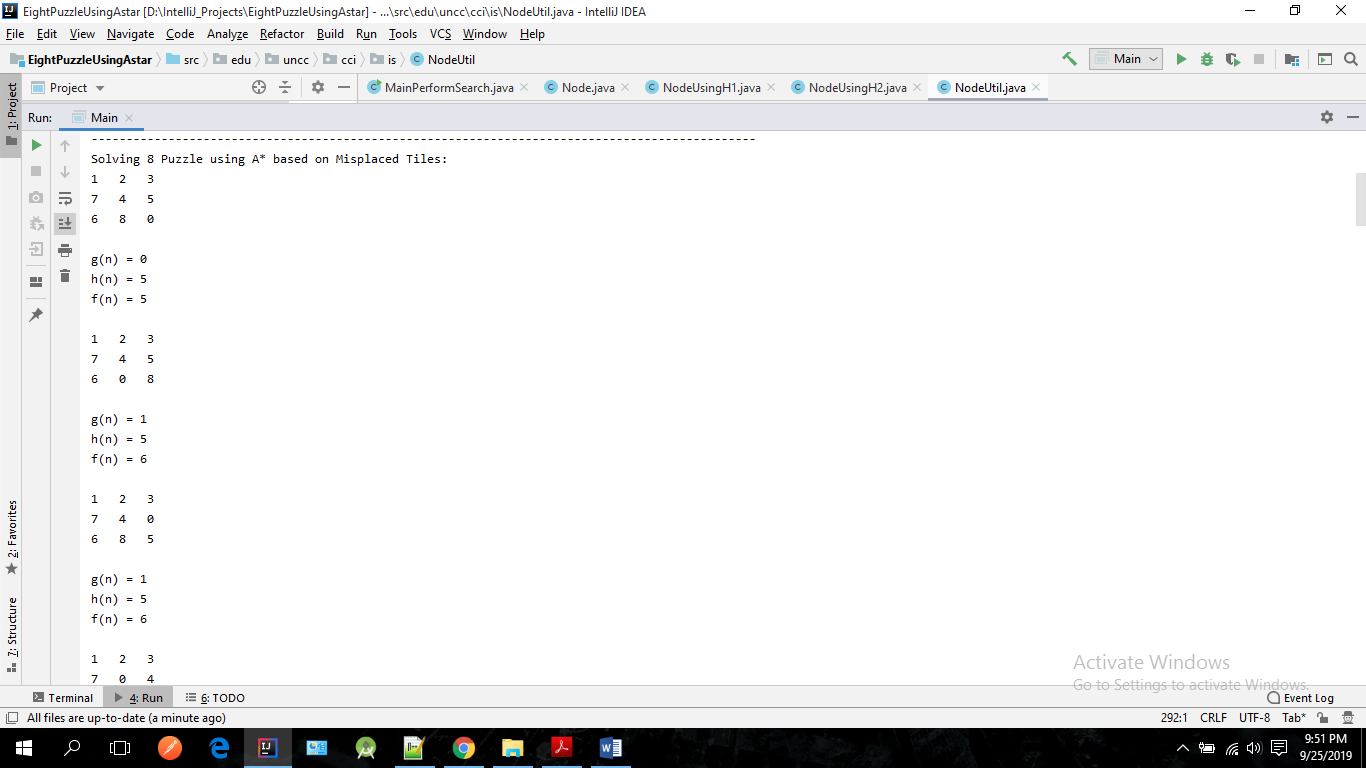
Enter the elements for the start state of the puzzle ==>

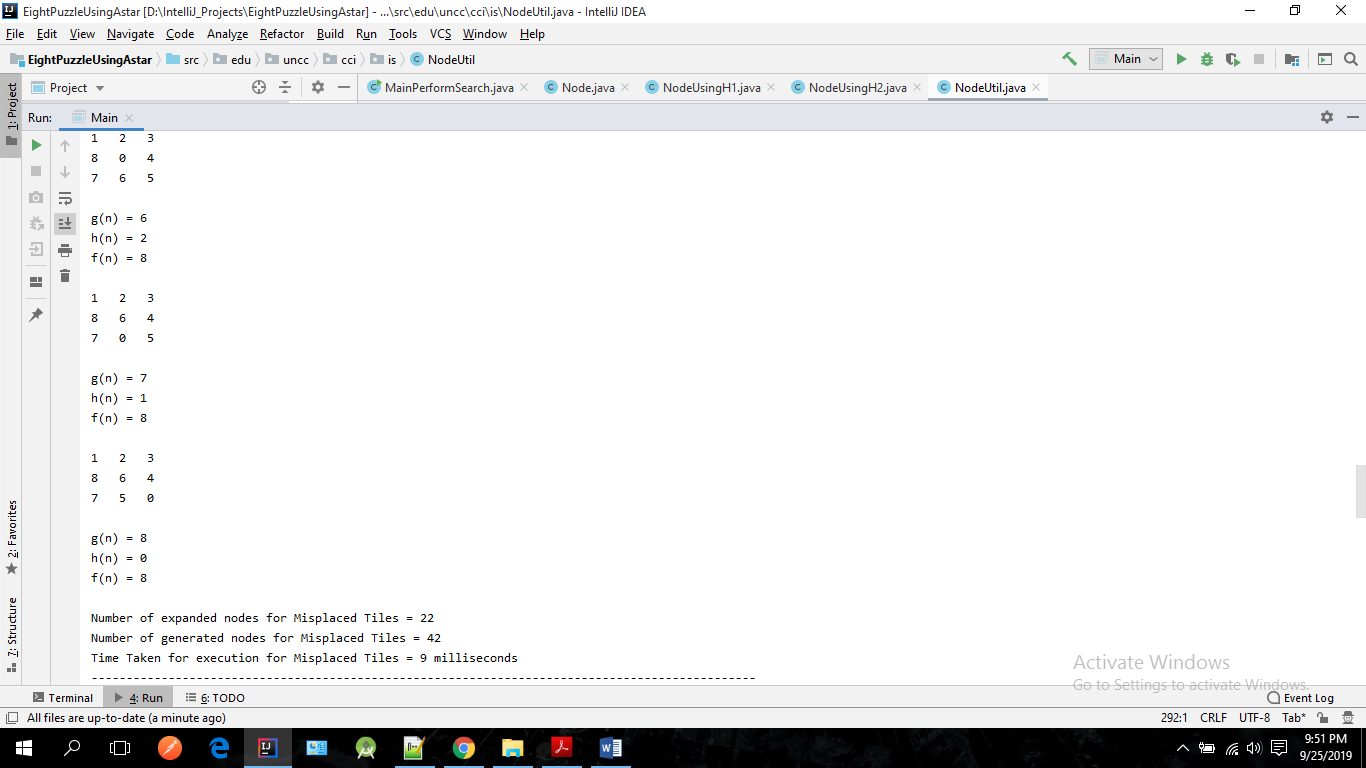
1 2 3 7 4 5 6 8 0

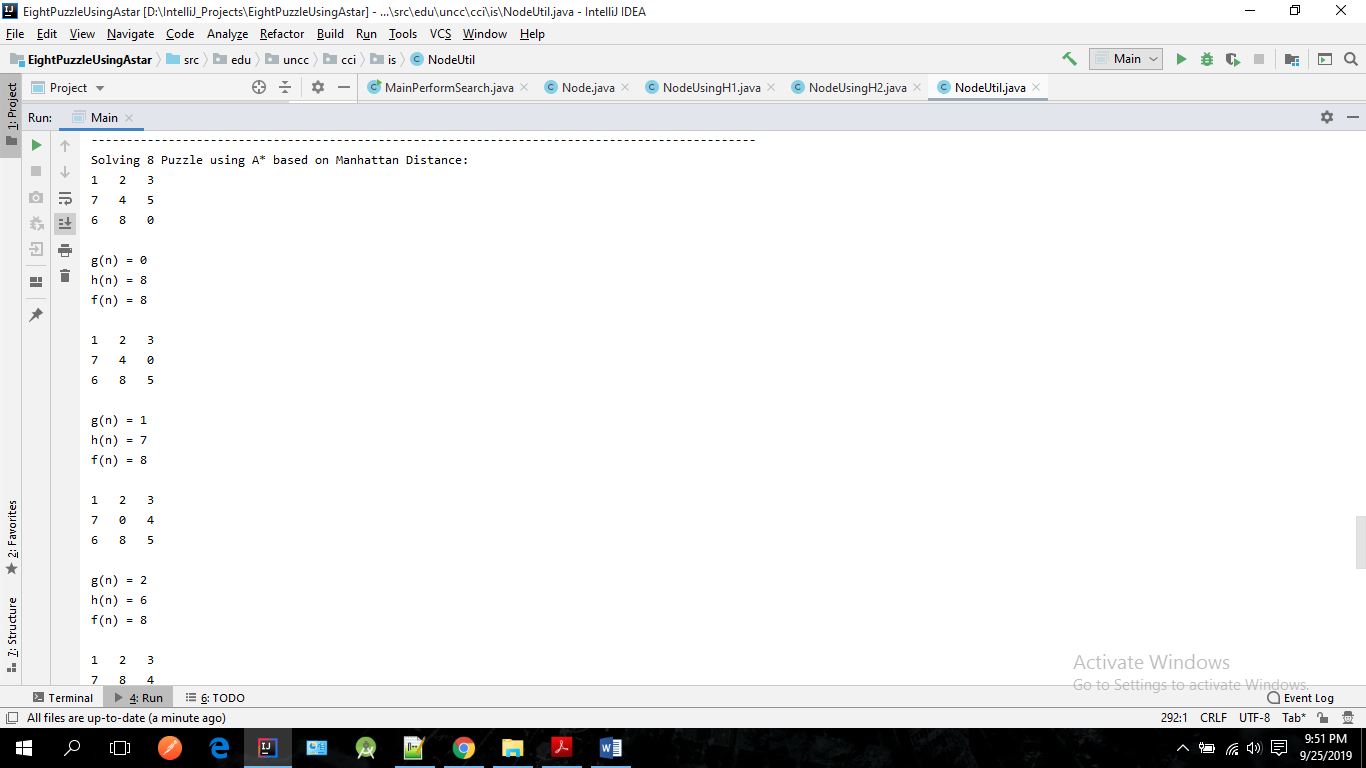
Enter the elements for the goal state of the puzzle ==>

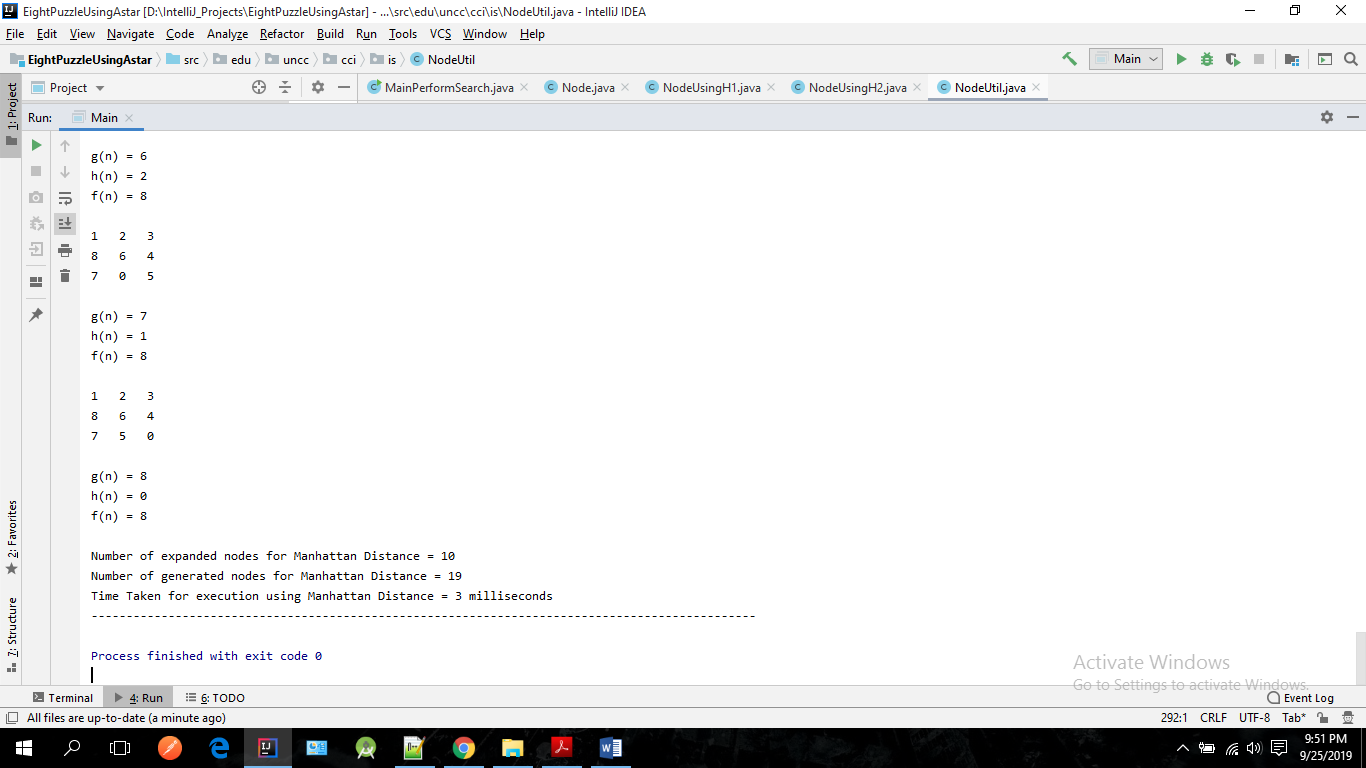
1 2 3 8 6 4 7 5 0



****

****

****

****

**Source Code:**

1. **MainPerformSearch**

package edu.uncc.cci.is;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.PriorityQueue;

import java.util.Scanner;

/\*\*

\* Project 01

\* Authors: Ankit Pandita, Jinraj Jain

\*/

//Main Class

public class MainPerformSearch {

public static int[][] goalState;

public static PriorityQueue<NodeUsingH1> priorityQueueH1;

public static PriorityQueue<NodeUsingH2> priorityQueueH2;

public static ArrayList<NodeUsingH1> expandedNodesH1;

public static ArrayList<NodeUsingH2> expandedNodesH2;

public static void main(String[] args) {

priorityQueueH1 = new PriorityQueue<NodeUsingH1>();

priorityQueueH2 = new PriorityQueue<NodeUsingH2>();

expandedNodesH1 = new ArrayList<NodeUsingH1>();

expandedNodesH2 = new ArrayList<NodeUsingH2>();

Scanner in = new Scanner(System.in);

int[][] startState;

goalState = new int[3][3];

System.out.println("8 Puzzle problem using A\* searching");

//Input start state and goal state

System.out.println("Enter the elements for the start state:");

startState = NodeUtil.generateUserInputMatrix(in);

System.out.println("Enter the elements for the goal state:");

goalState = NodeUtil.generateUserInputMatrix(in);

calcUsingMisplacedTiles(startState);

calcUsingManhattanDistance(startState);

}

//Mehod using Heuristic One (Misplaced tiles)

private static void calcUsingMisplacedTiles(int[][] startState) {

System.out.println("-----------------------------------------------------------------------------------------------");

System.out.println("Solving 8 Puzzle using A\* based on Misplaced Tiles:");

long startTime = System.currentTimeMillis();

//Level is 0

NodeUsingH1 state = new NodeUsingH1(startState, 0);

processPuzzlePlayH1(state);

for (NodeUsingH1 nodeUsingH1 : expandedNodesH1) {

NodeUtil.printMoveDetails(nodeUsingH1);

}

//Check for infinite loop

if (priorityQueueH1.size() >= NodeUtil.THRESHOLD) {

System.out.println("Application is performing beyond set threshold value and will be exiting.");

System.out.println("To increase threshold value, modify the NodeUtil.THRESHOLD variable.");

System.out.println("NOTE: This will increase the time needed for solution calculation.");

} else {

System.out.println("Number of expanded nodes for Misplaced Tiles = " + expandedNodesH1.size());

System.out.println("Number of generated nodes for Misplaced Tiles = " + (expandedNodesH1.size() + priorityQueueH1.size()));

}

long endTime = System.currentTimeMillis();

// Print Time Taken for Total Execution

System.out.println("Time Taken for execution for Misplaced Tiles = " + (endTime - startTime) + " milliseconds");

System.out.println("-----------------------------------------------------------------------------------------------");

}

//Method using Heuristic Two (Manhattan Distance)

private static void calcUsingManhattanDistance(int[][] startState) {

System.out.println("Solving 8 Puzzle using A\* based on Manhattan Distance:");

long startTime = System.currentTimeMillis();

//Level is 0

NodeUsingH2 state = new NodeUsingH2(startState, 0);

processPuzzlePlayH2(state);

for (NodeUsingH2 nodeUsingH2 : expandedNodesH2) {

NodeUtil.printMoveDetails(nodeUsingH2);

}

//Check for infinite loop

if (priorityQueueH2.size() >= NodeUtil.THRESHOLD) {

System.out.println("Application is performing beyond set threshold value and will be exiting");

System.out.println("To increase threshold value, modify the NodeUtil.THRESHOLD variable");

System.out.println("NOTE: This will increase the time needed for solution calculation");

} else {

System.out.println("Number of expanded nodes for Manhattan Distance = " + expandedNodesH2.size());

System.out.println("Number of generated nodes for Manhattan Distance = " + (expandedNodesH2.size() + priorityQueueH2.size()));

}

long endTime = System.currentTimeMillis();

// Print Time Taken for Total Execution

System.out.println("Time Taken for execution using Manhattan Distance = " + (endTime - startTime) + " milliseconds");

System.out.println("-----------------------------------------------------------------------------------------------");

}

// Processes the current state of the Puzzle and PriorityQueue for Heuristic One

public static void processPuzzlePlayH1(NodeUsingH1 move) {

priorityQueueH1.add(move);

ArrayList<NodeUsingH1> childNodesList = new ArrayList<NodeUsingH1>();

do {

boolean isNodeVisited;

//Poll function retrieves and removes the head of this queue

NodeUsingH1 currentPuzzleState = priorityQueueH1.poll();

//Once removed it is added to the expandedNodes queue to avoid duplicate processing

expandedNodesH1.add(currentPuzzleState);

//Keep checking if goal state has been reached by comparing every element position with the goal state

if (currentPuzzleState != null && Arrays.deepEquals(currentPuzzleState.stateOfPuzzle, goalState)) {

break;

}

if (currentPuzzleState != null) {

childNodesList = currentPuzzleState.generateChildNodesH1(currentPuzzleState);

}

//Check if expanded node is already visited

for (NodeUsingH1 childNode : childNodesList) {

isNodeVisited = false;

for (NodeUsingH1 expandedNode : expandedNodesH1) {

if (Arrays.deepEquals(childNode.stateOfPuzzle, expandedNode.stateOfPuzzle)) {

isNodeVisited = true;

}

}

if (isNodeVisited) {

continue;

}

priorityQueueH1.add(childNode);

}

} while (!priorityQueueH1.isEmpty() && priorityQueueH1.size() <= NodeUtil.THRESHOLD);

}

// Processes the current state of the Puzzle and PriorityQueue for Heuristic Two

public static void processPuzzlePlayH2(NodeUsingH2 move) {

priorityQueueH2.add(move);

ArrayList<NodeUsingH2> childNodesList = new ArrayList<NodeUsingH2>();

do {

boolean isNodeVisited;

//Poll function retrieves and removes the head of this queue

NodeUsingH2 currentPuzzleState = priorityQueueH2.poll();

//Once removed it is added to the expandedNodes queue to avoid duplicate processing

expandedNodesH2.add(currentPuzzleState);

//Keep checking if goal state has been reached by comparing every element position with the goal state

if (currentPuzzleState != null && Arrays.deepEquals(currentPuzzleState.stateOfPuzzle, goalState)) {

break;

}

if (currentPuzzleState != null) {

childNodesList = currentPuzzleState.generateChildNodesH2(currentPuzzleState);

}

//Check if expanded node is already visited

for (NodeUsingH2 childNode : childNodesList) {

isNodeVisited = false;

for (NodeUsingH2 expandedNode : expandedNodesH2) {

if (Arrays.deepEquals(childNode.stateOfPuzzle, expandedNode.stateOfPuzzle)) {

isNodeVisited = true;

}

}

if (isNodeVisited) {

continue;

}

priorityQueueH2.add(childNode);

}

} while (!priorityQueueH2.isEmpty() && priorityQueueH2.size() <= NodeUtil.THRESHOLD);

}

}

1. **Node**

package edu.uncc.cci.is;

import java.util.ArrayList;

/\*\*

\* Project 01

\* Authors: Ankit Pandita, Jinraj Jain

\*/

public class Node {

public int aStarDistance;

public int level;

public int heuristicDistance;

public int[][] stateOfPuzzle;

}

1. **NodeUsingH1**

package edu.uncc.cci.is;

import java.util.ArrayList;

/\*\*

\* Project 01

\* Authors: Ankit Pandita, Jinraj Jain

\*/

//By counting the number of misplaced tiles

public class NodeUsingH1 extends Node implements Comparable<NodeUsingH1> {

public NodeUsingH1(int[][] array, int level) {

int lengthOfArray = array.length;

this.stateOfPuzzle = new int[lengthOfArray][lengthOfArray];

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

System.arraycopy(array[i], 0, this.stateOfPuzzle[i], 0, lengthOfArray);

}

this.level = level;

this.heuristicDistance = calcMisplacedTilesDistance();

this.aStarDistance = this.level + this.heuristicDistance;

}

//method to calculate sum of the distances of the tiles from their goal positions

private int calcMisplacedTilesDistance() {

int count = 0;

int lengthOfArray = MainPerformSearch.goalState.length;

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

for(int j = 0 ; j < lengthOfArray; j++) {

if (this.stateOfPuzzle[i][j] == 0) {

continue;

}

if(this.stateOfPuzzle[i][j] != MainPerformSearch.goalState[i][j]) {

count++;

}

}

}

return count;

}

//Method to generate child nodes

public ArrayList<NodeUsingH1> generateChildNodesH1(NodeUsingH1 parentNode) {

ArrayList<NodeUsingH1> childNodes = new ArrayList<NodeUsingH1>();

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

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

if (parentNode.stateOfPuzzle[row][column] == 0) {

//Up

if (column - 1 >= 0) {

int[][] a = new int[3][3];

for (int row1 = 0; row1 < 3; row1++) {

System.arraycopy(parentNode.stateOfPuzzle[row1], 0, a[row1], 0, 3);

}

NodeUtil.moveElement(a, row, column, row, column - 1);

addChildNodesToList(parentNode, childNodes, a);

}

//Down

if (column + 1 < 3) {

int[][] a = new int[3][3];

for (int row2 = 0; row2 < 3; row2++) {

System.arraycopy(parentNode.stateOfPuzzle[row2], 0, a[row2], 0, 3);

}

NodeUtil.moveElement(a, row, column, row, column + 1);

addChildNodesToList(parentNode, childNodes, a);

}

//Left

if (row - 1 >= 0) {

int[][] a = new int[3][3];

for (int row3 = 0; row3 < 3; row3++) {

System.arraycopy(parentNode.stateOfPuzzle[row3], 0, a[row3], 0, 3);

}

NodeUtil.moveElement(a, row, column, row - 1, column);

addChildNodesToList(parentNode, childNodes, a);

}

//Right

if (row + 1 < 3) {

int[][] a = new int[3][3];

for (int row4 = 0; row4 < 3; row4++) {

System.arraycopy(parentNode.stateOfPuzzle[row4], 0, a[row4], 0, 3);

}

NodeUtil.moveElement(a, row, column, row + 1, column);

addChildNodesToList(parentNode, childNodes, a);

}

}

}

}

return childNodes;

}

//method increments the current level by one for every expansion

public void addChildNodesToList(NodeUsingH1 parentNode, ArrayList<NodeUsingH1> childNodes, int[][] a) {

NodeUsingH1 childNode = new NodeUsingH1(a, parentNode.level + 1);

childNodes.add(childNode);

}

//comparator that determines the order in which the elements are accessed in the Priority Queue

@Override

public int compareTo(NodeUsingH1 nodeUsingH1) {

return Integer.compare(this.aStarDistance, nodeUsingH1.aStarDistance);

}

}

1. **NodeUsingH2**

package edu.uncc.cci.is;

import java.util.ArrayList;

/\*\*

\* Project 01

\* Authors: Ankit Pandita, Jinraj Jain

\*/

// By using the Manhattan distance

public class NodeUsingH2 extends Node implements Comparable<NodeUsingH2> {

public NodeUsingH2(int[][] array, int level) {

int lengthOfArray = array.length;

this.stateOfPuzzle = new int[lengthOfArray][lengthOfArray];

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

System.arraycopy(array[i], 0, this.stateOfPuzzle[i], 0, lengthOfArray);

}

this.level = level;

this.heuristicDistance = calcManhattanDistance();

this.aStarDistance = this.level + this.heuristicDistance;

}

//method to calculate Manhattan Distance

private int calcManhattanDistance() {

int manhattanDistance = 0;

int[] index = new int[2];

int lengthOfArray = MainPerformSearch.goalState.length;

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

for (int j = 0; j < lengthOfArray; j++) {

if (this.stateOfPuzzle[i][j] == 0) {

continue;

}

index = checkPosition(MainPerformSearch.goalState, this.stateOfPuzzle[i][j]);

manhattanDistance += (Math.abs(i - index[0]) + Math.abs(j - index[1]));

}

}

return manhattanDistance;

}

// Returns the current position of the element in the puzzle state provided as argument

public static int[] checkPosition(int[][] currentPuzzleState, int element) {

int[] location = new int[2];

for (int i = 0; i < currentPuzzleState.length; ++i) {

for (int j = 0; j < currentPuzzleState.length; ++j) {

if (currentPuzzleState[i][j] == element) {

location[0] = i;

location[1] = j;

}

}

}

return location;

}

//Method to generate child nodes

public ArrayList<NodeUsingH2> generateChildNodesH2(NodeUsingH2 parentNode) {

ArrayList<NodeUsingH2> childNodes = new ArrayList<NodeUsingH2>();

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

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

if (parentNode.stateOfPuzzle[row][column] == 0) {

//Up

if (column - 1 >= 0) {

int[][] a = new int[3][3];

for (int row1 = 0; row1 < 3; row1++) {

System.arraycopy(parentNode.stateOfPuzzle[row1], 0, a[row1], 0, 3);

}

NodeUtil.moveElement(a, row, column, row, column - 1);

addChildNodesToList(parentNode, childNodes, a);

}

//Down

if (column + 1 < 3) {

int[][] a = new int[3][3];

for (int row2 = 0; row2 < 3; row2++) {

System.arraycopy(parentNode.stateOfPuzzle[row2], 0, a[row2], 0, 3);

}

NodeUtil.moveElement(a, row, column, row, column + 1);

addChildNodesToList(parentNode, childNodes, a);

}

//Left

if (row - 1 >= 0) {

int[][] a = new int[3][3];

for (int row3 = 0; row3 < 3; row3++) {

System.arraycopy(parentNode.stateOfPuzzle[row3], 0, a[row3], 0, 3);

}

NodeUtil.moveElement(a, row, column, row - 1, column);

addChildNodesToList(parentNode, childNodes, a);

}

//Right

if (row + 1 < 3) {

int[][] a = new int[3][3];

for (int row4 = 0; row4 < 3; row4++) {

System.arraycopy(parentNode.stateOfPuzzle[row4], 0, a[row4], 0, 3);

}

NodeUtil.moveElement(a, row, column, row + 1, column);

addChildNodesToList(parentNode, childNodes, a);

}

}

}

}

return childNodes;

}

//method increments the current level by one for every expansion

public void addChildNodesToList(NodeUsingH2 parentNode, ArrayList<NodeUsingH2> childNodes, int[][] a) {

NodeUsingH2 childNode = new NodeUsingH2(a, parentNode.level + 1);

childNodes.add(childNode);

}

//comparator that determines the order in which the elements are accessed in the Priority Queue

@Override

public int compareTo(NodeUsingH2 nodeUsingH2) {

return Integer.compare(this.aStarDistance, nodeUsingH2.aStarDistance);

}

}

1. **NodeUtil**

package edu.uncc.cci.is;

import java.util.Arrays;

import java.util.Scanner;

/\*\*

\* Project 01

\* Authors: Ankit Pandita, Jinraj Jain

\*/

public class NodeUtil {

//constant to determines how long each heuristic technique will run

public static final int THRESHOLD = 1000;

//prints the expanded node

public static <T extends Node> void printMoveDetails(T Node) {

System.out.println(Arrays.deepToString(Node.stateOfPuzzle)

.replace("[", "").replace("], ", "\n").replace("[[", "")

.replace("]]", "").replace(", ", "\t"));

System.out.println("\ng(n) = " + Node.level);

System.out.println("h(n) = " + Node.heuristicDistance);

System.out.println("f(n) = " + Node.aStarDistance + "\n");

}

//method to takes input from user

public static int[][] generateUserInputMatrix(Scanner input) {

int[][] puzzleInput = new int[3][3];

for (int i = 0; i < puzzleInput.length; i++) {

for (int j = 0; j < puzzleInput[i].length; j++) {

puzzleInput[i][j] = input.nextInt();

if (puzzleInput[i][j] < 0 || puzzleInput[i][j] / 9 > 0) {

System.out.println("Invalid Entry. Please Try Again.");

System.exit(0);

}

}

}

return puzzleInput;

}

// method to move element by swapping

public static void moveElement(int[][] currentPuzzleState, int row1, int column1, int row2, int column2) {

int tmp = currentPuzzleState[row1][column1];

currentPuzzleState[row1][column1] = currentPuzzleState[row2][column2];

currentPuzzleState[row2][column2] = tmp;

}

}