Assignment No: 6

**Title: -**

Solve 8 puzzle Problem using A\* Algorithm. Assume any initial configuration and define goal configuration clearly.

**Theory:**

[**N-Puzzle**](https://en.wikipedia.org/wiki/15_puzzle) or **sliding puzzle** is a popular puzzle that consists of N tiles where N can be 8, 15, 24, and so on. In our example N = 8. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns. Eg. 15-Puzzle will have 4 rows and 4 columns and an 8-Puzzle will have 3 rows and 3 columns. The puzzle consists of N tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal configuration.

Rules for solving the puzzle.

Instead of moving the tiles in the empty space, we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions viz.,

1.Up  
2.Down  
3.Right   
4. Left

The empty space cannot move diagonally and can take **only one step at a time**(i.e. move the empty space one position at a time).

## A\* Algorithm

A\* is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. Noted for its performance and accuracy, it enjoys widespread use.  
The key feature of the A\* algorithm is that it keeps a track of each visited node which helps in ignoring the nodes that are already visited, saving a huge amount of time. It also has a list that holds all the nodes that are left to be explored and it chooses the most optimal node from this list, thus saving time not exploring unnecessary or less optimal nodes.

**Program: -**

import java.util.ArrayList;

import java.util.List;

import java.util.PriorityQueue;

import java.util.\*;

public class Puzzle {

public int dimension = 3;

// Moves

int[] row = { 1, 0, -1, 0 };

int[] col = { 0, -1, 0, 1 };

public int calculateCost(int[][] initial, int[][] goal) {

int count = 0;

int n = initial.length;

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

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

if (initial[i][j] != 0 && initial[i][j] != goal[i][j]) {

count++;

}

}

}

return count;

}

public void printMatrix(int[][] matrix) {

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

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

System.out.print(matrix[i][j] + " ");

}

System.out.println();

}

}

public boolean isSafe(int x, int y) {

return (x >= 0 && x < dimension && y >= 0 && y < dimension);

}

public void printPath(Node root) {

if (root == null) {

return;

}

printPath(root.parent);

System.out.println(" |");

System.out.println(" V");

System.out.println();

printMatrix(root.matrix);

System.out.println();

}

public boolean isSolvable(int[][] matrix) {

int count = 0;

List<Integer> array = new ArrayList<Integer>();

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

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

array.add(matrix[i][j]);

}

}

Integer[] anotherArray = new Integer[array.size()];

array.toArray(anotherArray);

for (int i = 0; i < anotherArray.length - 1; i++) {

for (int j = i + 1; j < anotherArray.length; j++) {

if (anotherArray[i] != 0 && anotherArray[j] != 0 && anotherArray[i] > anotherArray[j]) {

count++;

}

}

}

return count % 2 == 0;

}

public void solvePuzzle(int[][] initial, int[][] goal, int x, int y) {

PriorityQueue<Node> pq = new PriorityQueue<Node>(1000, (a, b) -> (a.cost + a.level) - (b.cost + b.level));

Node root = new Node(initial, x, y, x, y, 0, null);

root.cost = calculateCost(initial, goal);

pq.add(root);

while (!pq.isEmpty()) {

Node min = pq.poll();

if (min.cost == 0) {

printPath(min);

return;

}

for (int i = 0; i < 4; i++) {

if (isSafe(min.x + row[i], min.y + col[i])) {

Node child = new Node(min.matrix, min.x, min.y, min.x + row[i], min.y + col[i], min.level + 1, min);

child.cost = calculateCost(child.matrix, goal);

pq.add(child);

}

}

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int initial[][] = new int[3][3];

int goal[][] = new int[3][3];

System.out.println("Enter Initial Matrix: ");

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

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

initial[i][j] = scanner.nextInt();

}

}

System.out.println("Enter Goal Matrix: ");

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

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

goal[i][j] = scanner.nextInt();

}

}

System.out.println();

// White tile coordinate

int x = 1, y = 0;

Puzzle puzzle = new Puzzle();

if (puzzle.isSolvable(initial)) {

puzzle.solvePuzzle(initial, goal, x, y);

}

else {

System.out.println("Puzzle is not solvable");

}

}

}

class Node {

public Node parent;

public int[][] matrix;

// Blank tile cordinates

public int x, y;

// Number of misplaced tiles

public int cost;

// The number of moves so far

public int level;

public Node(int[][] matrix, int x, int y, int newX, int newY, int level, Node parent) {

this.parent = parent;

this.matrix = new int[matrix.length][];

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

this.matrix[i] = matrix[i].clone();

}

// Swap value

this.matrix[x][y] = this.matrix[x][y] + this.matrix[newX][newY];

this.matrix[newX][newY] = this.matrix[x][y] - this.matrix[newX][newY];

this.matrix[x][y] = this.matrix[x][y] - this.matrix[newX][newY];

this.cost = Integer.MAX\_VALUE;

this.level = level;

this.x = newX;

this.y = newY;

}

}

**Output:**

