CS 205: Artificial Intelligence
The Eight Puzzle

Prudhvi Manukonda
SID:862325916

In completing this report I consulted:
1) Blind search and Heuristic search Lecture Slides of CS 205.
2)Java documentation Of Priority Queue and Comparator.
url:https://www.geeksforgeeks.org/implement-priorityqueue-comparator-java/
3)Java documentation for converting an array to an arraylist.
url:https://www.geeksforgeeks.org/array-to-arraylist-conversion-in-java/

All important ocde is original and some unimportant subroutines that are not completely original are:

- Used Java.util.Collections package for sorting purposes in Priority Queue.
- Used Jav.util.* package for list,stack and queue implementations.

Outline of this report

CS 205 PROJECT 1: The Eight Puzzle.

Prudhvi Manukonda

Introduction:

The sliding puzzle is a famous problem solving game. Generally we will be having 16 puzzles which are 4 x 4 grid and with an empty grid which we will slide to arrange all the elements in the grid sequentially. In Fig i) we can see the screenshot of the puzzle. This project in which we are solving an 8 puzzle game which consists of 3x3 grid and we have to arrange the elements sequentially 1 to 8 from an arbitrary state.This assignment given by professor Emman Keogh during spring 2022 . **I solved this project in Java(1.8) by using algorithms uniform search ,A* with manhattan heuristic and A* with misplaced heuristic.**

Fig i)A picture on a 15 puzzle and an empty element .

Uniform Search:

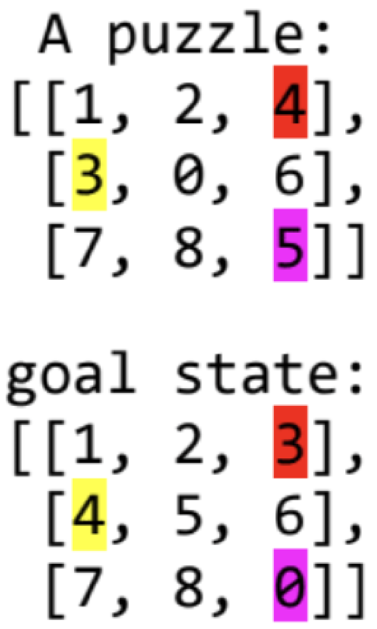
Uniform cost zero algorithm is exploring the nodes which have least cost first and then exploring next highest cost nodes it is nothing but an A* algorithm with h(n)=0. Because of this search there will be a lot of nodes to expand and time taken to search for a solution will be higher compared with A* algorithms with better heuristics.

A* star Algorithm:

A* star algorithm is an optimized version of uniform search algorithm; It expands the node which has a least g(n) + h(n) value . A* will be a better algorithm if it is used with better heuristic.

Misplaced heuristic:

Misplaced heuristic function calculates and counts how many misplaced blocks in the grid by iterating through the whole grid.In the below Fig ii) 4,3,5 are misplaced so we take misplaced heuristic count as 3 in below case.



```
A puzzle:
[[1, 2, 4],
 [3, 0, 6],
 [7, 8, 5]]

goal state:
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 0]]
```

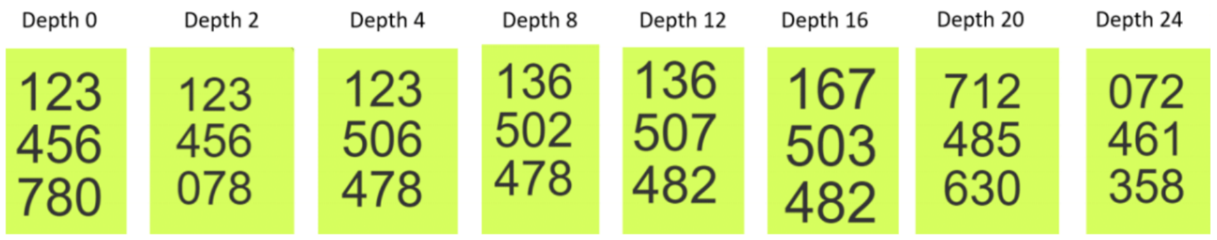
Fig ii): A worked example on Misplaced Tiles

Manhattan Heuristic:

Manhattan heuristic function calculates the Manhattan distance between the value present in the grid to the original location of the node and sums up for all elements in the grid.In the figure ii) manhattan heuristic will be 8 and we don't count zeros.

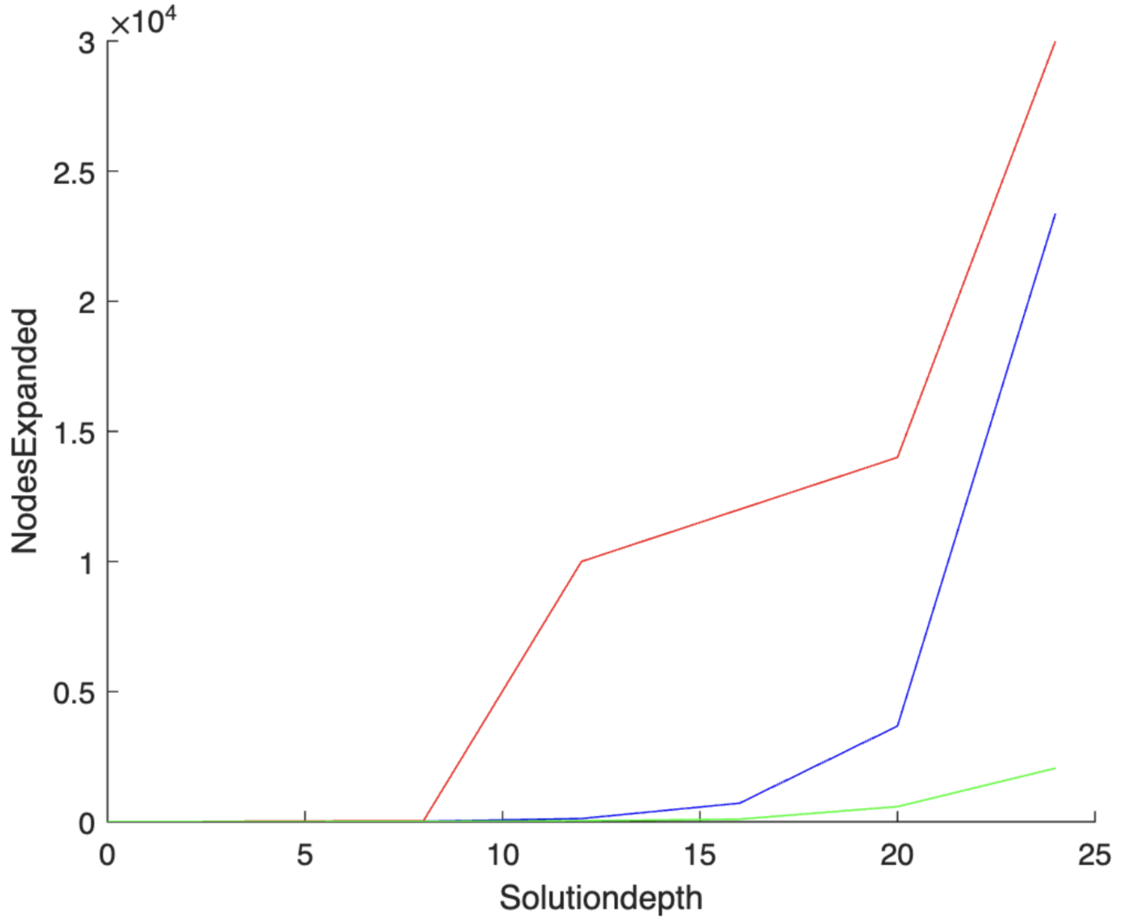
Comparison Of Different Algorithms

Given below in fig iii) are the different test cases provided by Professor Eamonn Keogh and are sorted by depth by using these test cases, I got to know that uniform cost search algorithm expands more nodes compared to A* and requires more time. A* star algorithms perform well with manhattan heuristic compared to misplaced tiles heuristic.



Depth 0	Depth 2	Depth 4	Depth 8	Depth 12	Depth 16	Depth 20	Depth 24
123	123	123	136	136	167	712	072
456	456	506	502	507	503	485	461
780	078	478	478	482	482	630	358

Fig iii) : Above are the test cases of 8 puzzle which are sorted by depth.



Red->uniform cost search
Blue->Misplaced Heuristic
Green->Manhattan Heuristic
Fig iv) Graph Drawn Above nodes vs depth

In the above figure iv) in graph we can clearly see that at smaller depths the difference between number of nodes expanded at three approaches is negligible but at higher depth around level 8 we can clearly see there is massive increase in number of nodes expanded by uniform cost search compared with misplaced heuristic. Although Misplaced manhattan heuristics expanded each node equally up to depth 15, we can clearly see that after depth 16 there are a lot more nodes expanded by misplaced heuristic compared with manhattan heuristic. For this problem manhattan heuristic worked very well. Then we can clearly state that although A* star algorithm is optimal when heuristic is bad it leads to a bad solution. We need to find better heuristic for A* star algorithm .In this case Manhattan distance is the best heuristic.

Conclusion:

By performing these three algorithms Uniform Cost search, Manhattan Heuristic and Misplaced Tile Heuristic on Eight puzzle problem we can conclude that

- It can be seen from the three algorithms Manhattan heuristic A* algorithm performed best followed by Misplaced tile heuristic A* algorithm followed by Uniform Cost search Algorithm.
- The Misplaced tile and Manhattan distance heuristic algorithms improve the uniform cost search algorithm which has h(n)=0 which is nothing but uses similar type of Breadth First Search Algorithm.Breadth First Search Algorithm has space complexity o(bpowerd) and time complexity o(bpowerd) where b is the branching factor and d is depth of the solution.
- While both manhattan and misplaced tile improved the breadth first search algorithm but examples above shows that when depth of the solution is more manhattan heuristic is more helpful.

Results:

i)**Result for depth 4**

The following is walk through of the output for depth 4

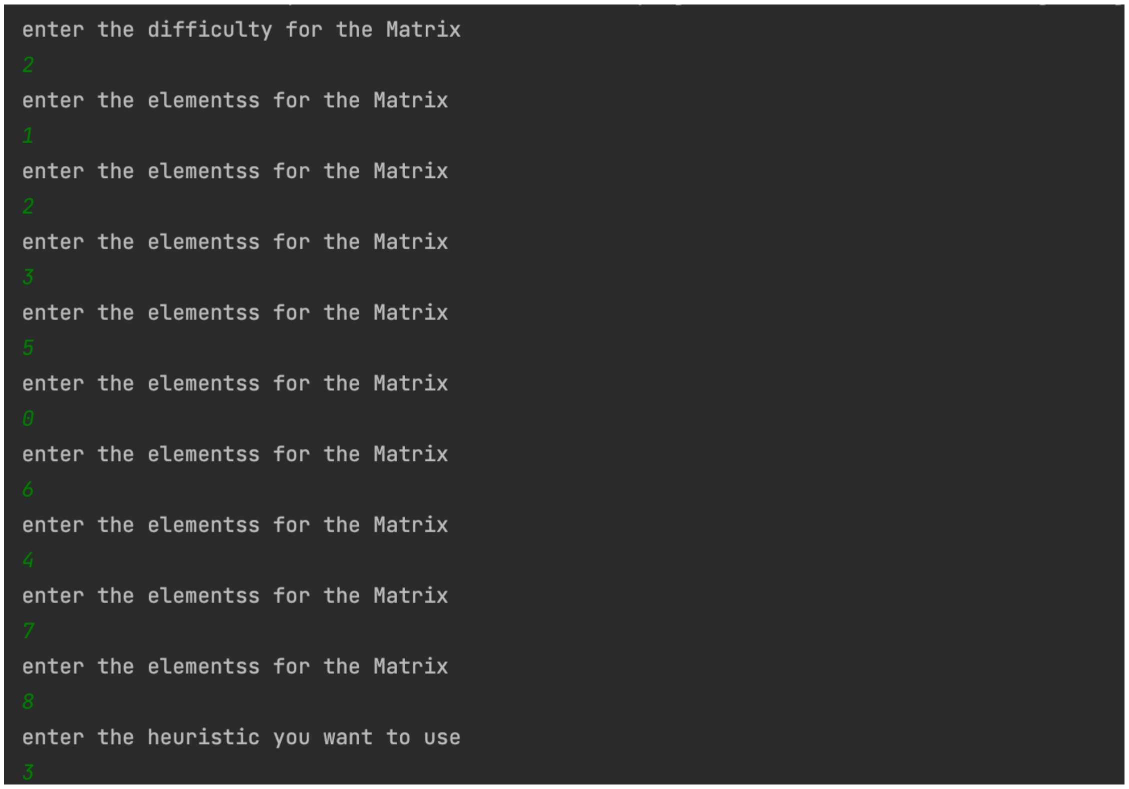
1)Enter the difficulty variable to select the hardcoded array or user array.

- If difficulty is 1 algorithm will execute already defined input or else it will take input from the user

2)Two-Dimensional Array is given from user element by element.

3)User will enter what heuristic to perform if heuristic is

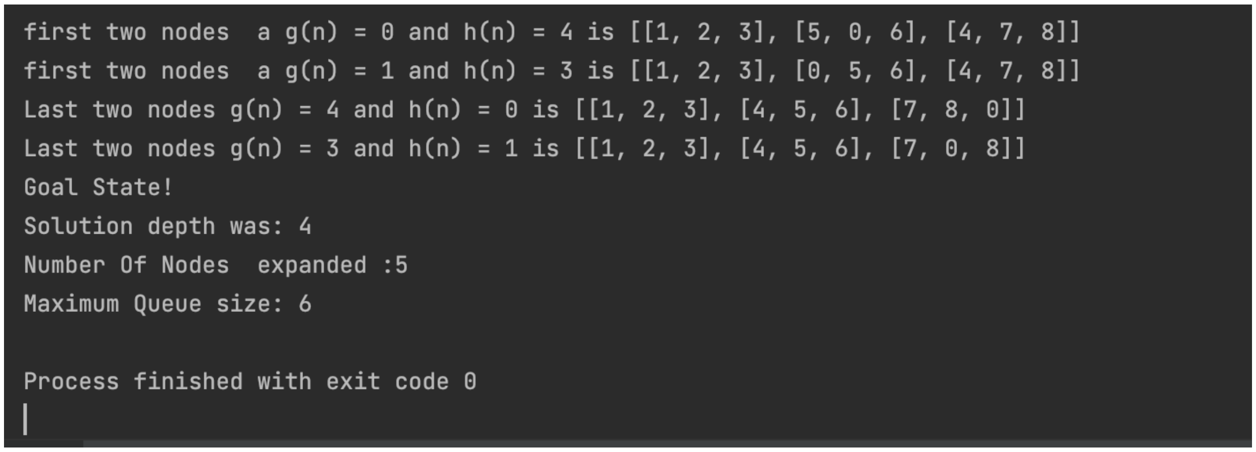
- One it will execute uniform cost search
- Two it will execute Misplaced Tile a* heuristic algorithm.
- Three it will execute Manhattan heuristic.



```
enter the difficulty for the Matrix
2
enter the elementss for the Matrix
1
enter the elementss for the Matrix
2
enter the elementss for the Matrix
3
enter the elementss for the Matrix
5
enter the elementss for the Matrix
0
enter the elementss for the Matrix
6
enter the elementss for the Matrix
4
enter the elementss for the Matrix
7
enter the elementss for the Matrix
8
enter the heuristic you want to use
3
```

Fig v)Fig contains input array taken for depth 4 element wise.

4)Since, It is not possible to print every node of the tree so ,I'm printing first two and last two nodes of the tree so output contains the first two best nodes and the last two nodes before goal state which is shown in .Fig vi).We can see from Fig vi) solution found at depth 4 and 5 nodes expanded and maximum queue size is 6.



```
first two nodes  a g(n) = 0 and h(n) = 4 is [[1, 2, 3], [5, 0, 6], [4, 7, 8]]
first two nodes  a g(n) = 1 and h(n) = 3 is [[1, 2, 3], [0, 5, 6], [4, 7, 8]]
Last two nodes g(n) = 4 and h(n) = 0 is [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Last two nodes g(n) = 3 and h(n) = 1 is [[1, 2, 3], [4, 5, 6], [7, 0, 8]]
Goal State!
Solution depth was: 4
Number Of Nodes  expanded :5
Maximum Queue size: 6

Process finished with exit code 0
```

Fig vi)Output Nodes for depth 4

ii)**Result for depth 24**

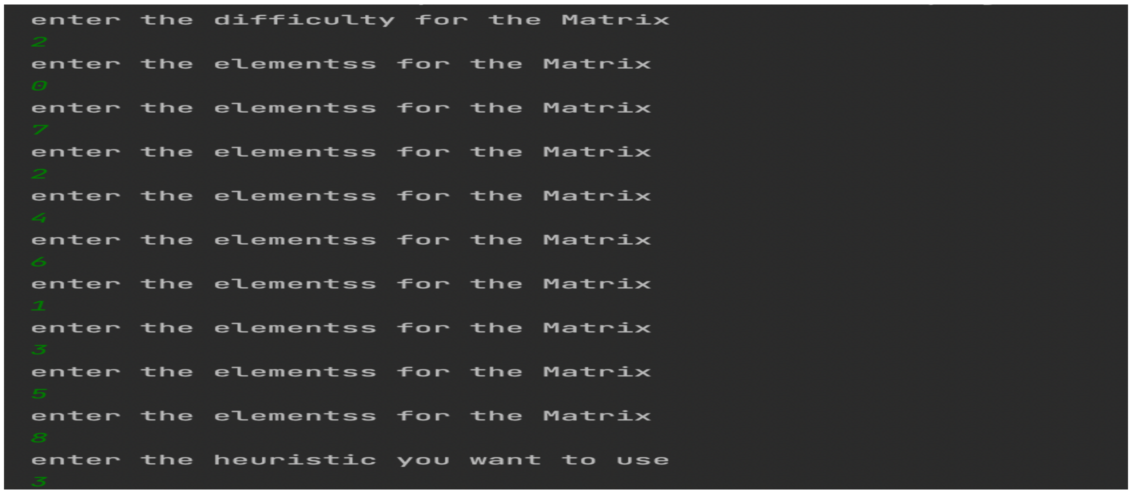
The following is walk through of the output for depth 24

1)Enter the difficulty variable to select the hardcoded array or user array.

- If difficulty is 1 algorithm will execute already defined input or else it will take input from the user.

2)Two-Dimensional Array is given from user element by element.
3)User will enter what heuristic to perform if heuristic is

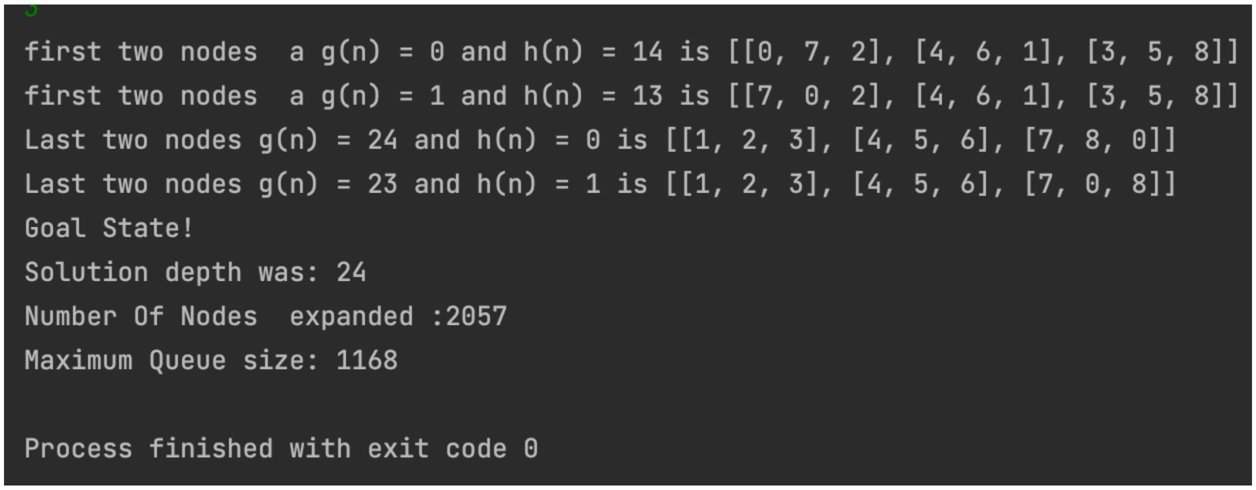
- One it will execute uniform cost search
- Two it will execute Misplaced Tile a* heuristic algorithm.
- Three it will execute Manhattan heuristic.



```
enter the difficulty for the Matrix
2
enter the elementss for the Matrix
0
enter the elementss for the Matrix
7
enter the elementss for the Matrix
2
enter the elementss for the Matrix
4
enter the elementss for the Matrix
6
enter the elementss for the Matrix
1
enter the elementss for the Matrix
3
enter the elementss for the Matrix
5
enter the elementss for the Matrix
8
enter the heuristic you want to use
3
```

Fig vii) Fig contains input array taken for depth 24 element wise.

4)Since, It is not possible to print every node of the tree so ,I'm printing first two and last two nodes of the tree so output contains the first two best nodes and the last two nodes before goal state which is shown in .Fig viii).We can see from Fig viii) solution found at depth 24 and 2057 nodes expanded and maximum queue size is 1168.



```
first two nodes  a g(n) = 0 and h(n) = 14 is [[0, 7, 2], [4, 6, 1], [3, 5, 8]]
first two nodes  a g(n) = 1 and h(n) = 13 is [[7, 0, 2], [4, 6, 1], [3, 5, 8]]
Last two nodes g(n) = 24 and h(n) = 0 is [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Last two nodes g(n) = 23 and h(n) = 1 is [[1, 2, 3], [4, 5, 6], [7, 0, 8]]
Goal State!
Solution depth was: 24
Number Of Nodes  expanded :2057
Maximum Queue size: 1168

Process finished with exit code 0
```

Fig viii)Goal State Nodes for depth 24

PROGRAM CODE:

Below written code for the program, also comments are highlighted and github link for the code is:https://github.com/PMANU005/CS-205-Project/blob/main/MyClass.java

```
import java.util.*;
import java.util.stream.Collectors;
import java.util.Scanner;
//defining our own comparator to sort according to the astar values in a node.
class ourcomparator implements Comparator<Node> {
  // we are overriding compare method from comparator class to sort the queue
according to astar values.
    public int compare(Node n1, Node n2) {
        //we are returning -1 because we want order to be in ascending order
        if (n1.getAstar() < n2.getAstar()) {
            return -1;
        } else if (n1.getAstar() > n2.getAstar()) {
            return 1;
        }
      //if both nodes astar values are equal then we are sorting according to
smallest g(n).
        else {
//sorting the nodes in ascending order comparing with nodes g(n) when nodes
astar values are equal.
            if (n1.g < n2.g) {
                return -1;
            } else if (n1.g > n2.g) {
                return 1;
            } else {

                return 0;
            }
        }
    }
}


//defining node class
class Node{
    //defining instance variables array a, g(n),h(n)
    int[][] a;
    int g;
    int h;
    //Initialization node class with varaiables
    Node(int g,int h,int[][] a){
        this.g=g;
        this.h=h;
        this.a=a;
    }
    //Defining a function Astar to return f(n) which is equal to g(n)+h(n)
    int getAstar(){
        return g+h;
    }
}
public class MyClass {
    //Defining function of manhattan heuristic

    public int manhattanheuristic(int[][] a) {
        int r = a.length;
        int c = a[0].length;
        int count = 0;
        int ans = 0;
            for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++) {
                if (a[i][j] != 0) {
                //finding corresponding element in the target 2d array by getting
corresponding row and column.
                    int row = (a[i][j]-1) / c;
     //since all elements are sequentially arranged in target array we can simply
take percentile of 3.
                    int column = (a[i][j]-1) % c;
                    ans = ans + Math.abs(i - row) + Math.abs(j - column);
                }
            }
        }
        return ans;
    }
    //I am defining a function to assign a given arrayvalues to other array.
    public int[][] assignthearray(int[][] given) {
        int[][] newarr = new int[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                newarr[i][j] = given[i][j];
            }
        }
        return newarr;
    }
    public int misplacedheuristic(int[][] a){
        int r=a.length;
        int c=a[0].length;
        int count=0;
        int ans=0;
        //updating the answer variable only when value is not equal to corresponding
count value.
        for(int i=0;i<r;i++){
            for(int j=0;j<c;j++){
                count=count+1;
                if(a[i][j]!=0){
                    if(a[i][j]!=count){
                        ans=ans+1;
                    }
                }
            }
        }
        return ans;
    }
    //I am defining another function to convert array to list.
    public List<List<Integer>> convertarrtoList(int[][] arr) {

        List<List<Integer>> nestedLists =
                Arrays.
                        //Convert the 2d array into a stream.
                                stream(arr).
                        //Map each 1d array (internalArray) in 2d array to a List.
                                map(
                                //Stream all the elements of each 1d array and put them into a
list of Integer.
                                internalArray ->
Arrays.stream(internalArray).boxed().collect(Collectors.toList())
                                )
                                //Put all the lists from the previous step into one big list.
                ).collect(Collectors.toList());
        return nestedLists;
    }
    public static void main(String args[]) {
        int i1 = 0;
        int j1 = 0;
        int[][] b;
        int[][] target = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};
        int r=target.length;
        int c=target[0].length;
        Scanner input = new Scanner(System.in);
        System.out.println("enter the difficulty for the Matrix");
        //hardcoded a random array if difficulty value is 1.
        int difficulty = input.nextInt();
        if (difficulty == 1) {
            b = new int[][]{{1, 2, 3}, {5, 0, 6}, {4, 7, 8}};
        } else {
            b = new int[3][3];
            for (int row = 0; row < r; row++) {
                for (int col = 0; col < c; col++) {
                    System.out.println("enter the elementss for the Matrix");
                    b[row][col] = input.nextInt();
                }
            }
        }
        System.out.println("enter the heuristic you want to use");
        int typeofheuristic=input.nextInt();
        // defining the directions array to add children into the queue.
        int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
        int ichild;
        int jchild;
        int noofnodes = 0;
        int maxqsize = 0;
        Node proot;
        Node child;
        Stack<Node> st = new Stack<Node>();
        Queue<Node> qu = new LinkedList<Node>();
        //Defining hash set to see that parent is not added in the queue again by
children.
        HashSet<List<List<Integer>>> h = new HashSet<List<List<Integer>>>();
        MyClass m1 = new MyClass();
        //creating the Node class with g(n),h(n),b array
        //hardcoded heuristic type with 1,2,3 to define uniform,misplacedheuristic
and manhattanheuristic.
        if(typeofheuristic==1){
            proot = new Node(0,0, b);}
        else if(typeofheuristic==2){
            proot = new Node(0, m1.misplacedheuristic(b), b);}
        else{
            proot = new Node(0, m1.manhattanheuristic(b), b);
        }
        //In java we can implement heap using priority queue.
        //priority queue contains pair class of node and sorted the queue using our
own comparator which sorts the queue using nodes astar value.
        PriorityQueue<Node> q = new PriorityQueue<Node>(new ourcomparator());
        q.add(proot);
        while (!q.isEmpty() && q.size() < 200000) {
            Node temp = q.poll();
            //Incrementing the node when a node is popped.
            noofnodes = noofnodes + 1;
            // Converting the 2d array into list.
            List<List<Integer>> nestedListsfunc = m1.convertarrtoList(temp.a);
            //adding into hashset to avoid repeated parents.
            h.add(nestedListsfunc);
            st.add(temp);
            qu.add(temp);
            int[][] arr = new int[r][c];
            //declaring a new array and making it equal to current popped node array
            arr = m1.assignthearray(temp.a);
            // getting zero location of the popped node thereby we can add its children
into the queue.
            out:
            for (int i = 0; i < r; i++) {
                for (int j = 0; j < c; j++) {
                    if (arr[i][j] == 0) {
                i1 = i;
                j1 = j;
                break out;
                    }
                }
```

```
            }
            //checking whether popped node is goal state
            if (Arrays.deepEquals(arr, target)) {

              int tl=0;
              //to print first two nodes of the tree
              while(tl<2 && !qu.isEmpty()) {
                    //popping the nodes to print the values.
                    Node out = qu.poll();
                    int[][] outarr = m1.assignthearray(out.a);
                    List<List<Integer>> li = m1.convertarrtoList(outarr);
                    System.out.println("first two nodes  a g(n) = " + out.g + " and h(n) = " +
out.h + " is " + li);
                    tl++;
                }
                //to print last two nodes
                int t=0;
                while(t<2 && !st.isEmpty())
                {
                    //popping the nodes to print the values.

                    Node out = st.pop();
                    int[][] outarr = m1.assignthearray(out.a);
                    List<List<Integer>> li = m1.convertarrtoList(outarr);
                    System.out.println("Last two nodes g(n) = " + out.g + " and h(n) = " + out.h
+ " is " + li);
                    t++;
                }
                System.out.println("Goal State!");
                System.out.println("Solution depth was: " + temp.g);
                System.out.println("Number Of Nodes  expanded :" + noofnodes);
                System.out.println("Maximum Queue size: " +  maxqsize);
                break;
            }
            //if poppednode is not equal to goal state then add it's children
            else {
                // to swap zero with 4 directions.
                for (int[] direction : directions) {
                    int[][] arrchild = new int[3][3];
                    arrchild = m1.assignthearray(arr);
                    //updating i and j by exploring 4 directions
                    ichild = i1 + direction[0];
                    jchild = j1 + direction[1];
                    //checking the boundary conditions
                    if (ichild >= 0 && ichild < 3 && jchild >= 0 && jchild < 3) {
                        //swapping the the zero with the updated inew and jnew value in the
array
                        arrchild[i1][j1] = arr[i1][j1] + arr[ichild][jchild];
                        arrchild[ichild][jchild] = 0;
                        //converting this new array into list
                        List<List<Integer>> nestedLists1 = m1.convertarrtoList(arrchild);
                        //creating a node for the children only if the children doesn't exist in
the hashset
                        if (!h.contains(nestedLists1)) {
                            //checking heuristic to call correspoding heuristic function.
                        //create node class with updated g(n) which will be current level +1
and h(n) and newarray
                            if(typeofheuristic==1){
                                child = new Node(temp.g + 1, 0, arrchild);}
                            else if(typeofheuristic==2){
                         child = new Node(temp.g + 1, m1.misplacedheuristic(arrchild), arrchild);}
                            else{
                         child = new Node(temp.g + 1, m1.manhattanheuristic(arrchild), arrchild);
                            }
                            //add the child into the queue
                            q.add(child);
                            maxqsize = Math.max(maxqsize, q.size());
                        }
                    }
                }

            }

        }

    }
}
```