Project report
on

Solving 8-puzzle using A* algorithm

Project Guidance By

Dr. Dewan Ahmed

Team details

Aarti Nimhan
animhan1@uncc.edu
801098198

Uma Sai Madhuri Jetty
ujetty@uncc.edu
801101049

Sahithi Priya Gutta
sgutta@uncc.edu
801098589

AIM

To solve 8-puzzle problem using A* algorithm

PROBLEM STATEMENT

Implement A* search algorithm and apply it to 8-puzzle problem, provide state space representation, operators, g (cost) and two heuristic functions of the 8-puzzle problem.

8-PUZZLE PROBLEM

The 8-puzzle consists of an area divided into 3x3 (3 by 3) grid. Each grid with in the puzzle is known as tile and each tile contains a number ranged between 1 to 8, so that they can be uniquely identified. Tile adjacent to the empty grid can be moved to the empty space leaving its previous position empty until reaching the goal.

PROBLEM FORMULATION

Goal: Goal State is initially given.
States: Integer locations of tiles.
Actions: Move the blank tile in left, up, down and right positions
Performance: Number of total moves in the solution

A* ALGORITHM

A* is an informed search algorithm used in path findings and graph traversals. It is a combination of uniform cost search and best first search, which avoids expanding expensive paths. A* star uses admissible heuristics which is optimal as it never over-estimates the path to goal. The evaluation function A* star uses for calculating distance is

$$f(n) = g(n) + h(n)$$

g(n) = cost so far to reach n
h(n) = estimated cost from n to goal
f(n) = estimated total cost of path through n to goal

Heuristic Functions

The heuristic function is a way to inform the search regarding the direction to a goal. It provides an information to estimate which neighboring node will lead to the goal. The two heuristic functions that we considered for solving 8-puzzle problem are

Misplaced Tile

The number of misplaced tiles calculated by comparing the current state and goal state.

Manhattan Distance

The distance between two tiles measured along the axes of right angles. It is the sum of absolute values of differences between goal state (i, j) coordinates and current state (l, m) coordinates respectively, i.e. $|i - l| + |j - m|$

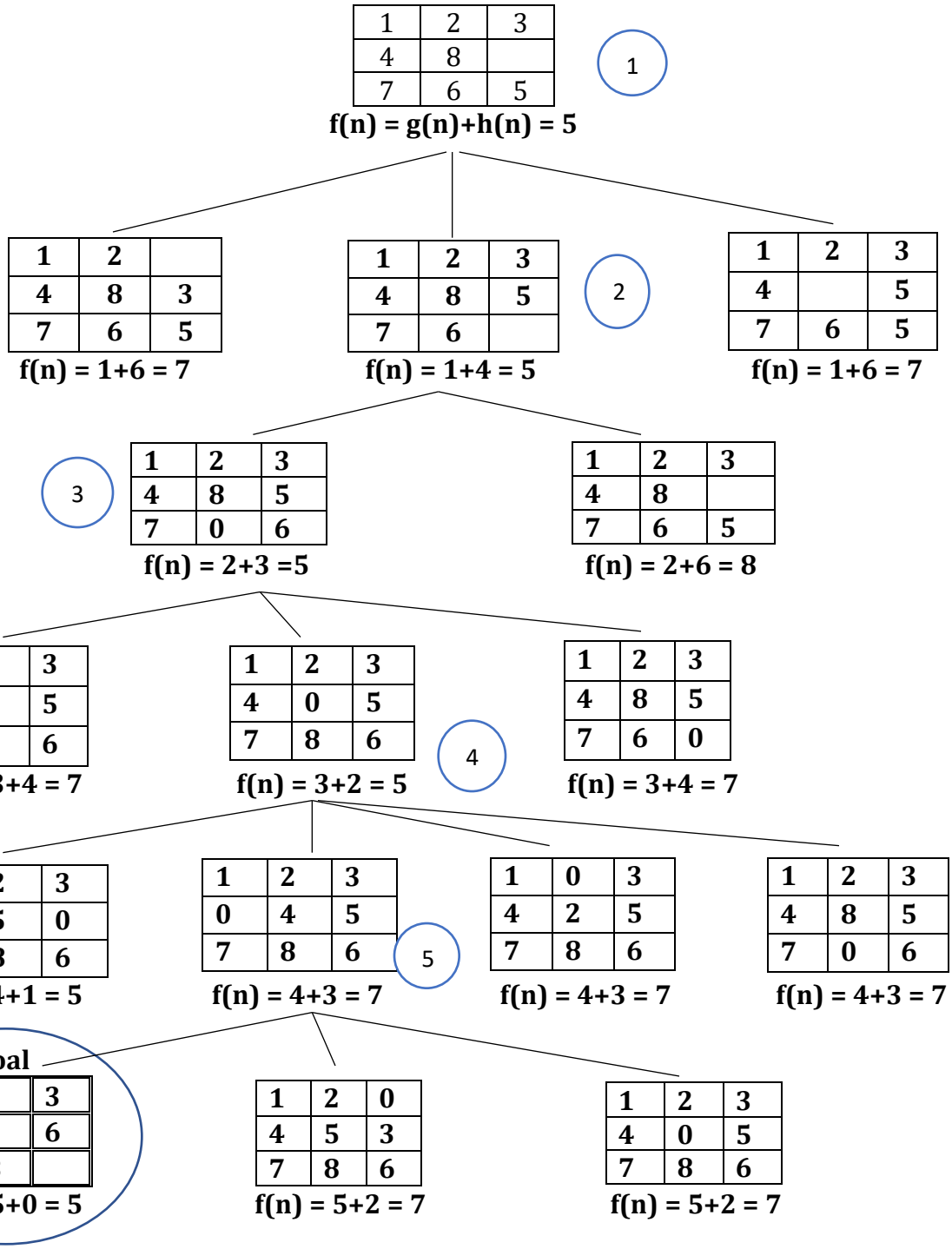
Sample 8-Puzzle solved using A* search Manhattan distance heuristic

Initial State

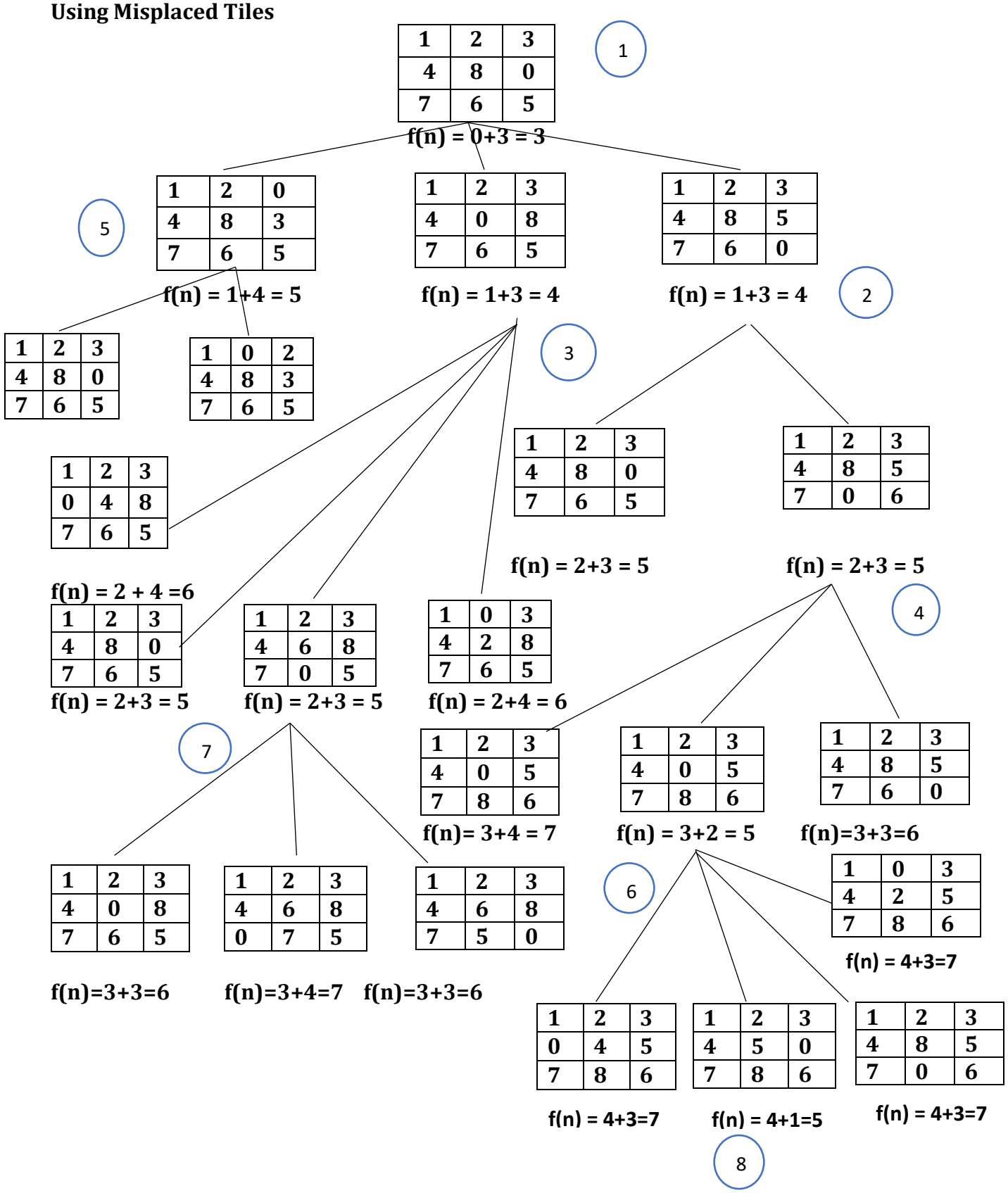
1	2	3
4	8	
7	6	5

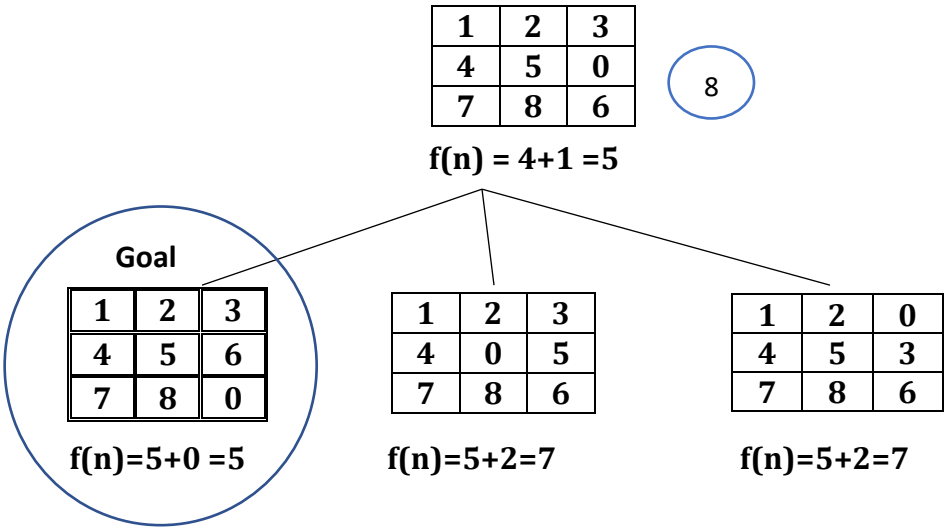
Goal State

1	2	3
4	5	6
7	8	



Using Misplaced Tiles





PROGRAM DESIGN AND EXPLANATION

Global variables

No global variables are used in the implementation**.**

Functions used

- manhattanDistanceHeuristic()
- misplacedTileHeuristic()
- calculateHeuristics()
- findingFreeTileLocation()
- calculatePossiblePositionsForExpansion()
- createChildNode()
- isChildInPriorityQueue()
- isChildInExploredSet()
- compareNodesForEquality()
- printSolutionPathNode()
- main ()

manhattanDistanceHeuristic

For calculating the distance of a node by comparing it with the provided goal state. The returned value is stored as heuristic cost. Identifies same value tile in both the states and calculates the absolute difference of both the tile indexes. Consider tile 1 is in location [0,1] in initial state and [1,2] in goal state then [abs(0-1)+abs(1-2)] = 2 is calculated and same is repeated for all the tiles.

misplacedTileHeuristic

For calculating the total number of misplaced tiles by comparing it with the provided goal state. The returned value is stored as heuristic cost. Compares whether the tiles values match in both initial state and goal state for the same index. If the value doesn't match hcost value is incremented.

calculateHeuristics
Updates the heuristic value for the appropriate type of heuristic. It sets these values to the node which is taken as input.

findingFreeTileLocation
For identifying the location of the free tile/ blank element '0'

calculatePossiblePositionsForExpansion

- For identifying the possible actions (up, down, left, right) that can be taken on the blank element
- Returns list of Strings representing comma separated indexes of valid possible free tile positions.

createChildNode

- Copies the state of the parent node (using copy one state to other method)
- For each child clones the current node and swap with the possible positions.
- Moves the free tile to a new position
- Creates and returns the child node

isChildInPriorityQueue
For checking if the created child node exists in priority queue and in explored set.

isChildInExploredSet
For checking if the created child node exists in priority queue and in explored set.

compareNodesForEquality
For checking the equality of two nodes by comparing the states.

printSolutionPathNode
Prints the solution path from initial state to goal state by backtracking from solution node to parent nodes.

main function

- Taking the Initial state and Final state input from user
- Checks for the valid input
- Creating and initializing a node
- Checking the parent id of the node

SOURCE CODE

AStarFramework.java

This class is a framework class which takes input, validates the input, does the necessary processing of the input so that the input can be initialized as required. That is converting the input of numbers into Nodes.

```java
import java.util.ArrayList;
import java.util.Scanner;

public class AStarFramework {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		AStarFramework aFramework = new AStarFramework();
		AStarSearchAlgo aStar = new AStarSearchAlgo();
		Node initialStateNode, goalStateNode;
		System.out.println(
				" Enter the Initial State (row wise) and seperate each
input by enter for the 8-puzzle problem:  ");
		initialStateNode = aFramework.initializingNode();
		System.out.println(
				" Enter the Goal State (row wise) and seperate each input
by enter for the 8-puzzle problem:  ");
		goalStateNode = aFramework.initializingNode();
		if (validation(initialStateNode.getState()) &&
validation(goalStateNode.getState())) {//If input entered is valid we proceed to
processing.
			aStar.aStarProcess(initialStateNode, goalStateNode);
		} else {
			return;
		}
	}

	/**
	 * Takes a node state as input and returns true if it is a valid input and
false if the input contains repeated numbers.
	 * @param nodeState is a two dimensional integer array which represents the
state of the puzzle board.
	 * @return a boolean true if input is valid and false if it is invalid.
	 */
	private static boolean validation(int[][] nodeState) {
		ArrayList<Integer> validInput = new ArrayList<Integer>();
		for (int x = 0; x < 9; x++) {
			validInput.add(x);
		}
		for (int i = 0; i < nodeState.length; i++) {
			for (int j = 0; j < nodeState.length; j++) {
				if (validInput.contains(nodeState[i][j])) {
					validInput.remove((Integer) nodeState[i][j]);
				} else {
					System.out.println("Entered input does not contain
unique numbers from 0 through 8. ");
					return false;
				}
			}
		}

		return true;
```

```
	}

	/**
	 * Takes input of integers
	 * @return a Node created form the input of numbers
	 */
	private Node initializingNode() {
		int[][] state = new int[3][3]; // As 8 puzzle will always be a 3X3
matrix
		@SuppressWarnings("resource")
		Scanner scanner = new Scanner(System.in);
		for (int i = 0; i < state.length; i++) {
			for (int j = 0; j < state.length; j++) {
				state[i][j] = scanner.nextInt();
			}
		}
		Node createdState = new Node();
		createdState.setState(state);
		return createdState;
	}

}
```

Node.java

This class is a node class which consists of the unique state of the puzzle. It contains State id and parent id of the node and also fcost, gcost and hcost fields of a node.

```
public class Node {
	private int[][] state;
	private int parentId;
	private int id;
	private int fCost;
	private int gCost;
	private int hCost;

	public int[][] getState() {
		return state;
	}

	public void setState(int[][] state) {
		this.state = state;
	}

	public int getParentId() {
		return parentId;
	}

	public void setParentId(int parentId) {
		this.parentId = parentId;
	}

	public int getId() {
		return id;
```

```
	}

	public void setId(int id) {
		this.id = id;
	}

	public int getfCost() {
		return fCost;
	}

	public void setfCost(int fCost) {
		this.fCost = fCost;
	}

	public int getgCost() {
		return gCost;
	}

	public void setgCost(int gCost) {
		this.gCost = gCost;
	}

	public int gethCost() {
		return hCost;
	}

	public void sethCost(int hCost) {
		this.hCost = hCost;
	}

}
```

AStarSearchAlgo.java

This class contains the logic for solving the 8-puzzle problem.

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Stack;

public class AStarSearchAlgo {

	// Constants for type of heuristic
	private static String MANHATTAN_DISTANCE_HEURISTIC = "Manhattan Distance
Heuristic";
	private static String MISPLACED_TILES_HEURISTIC = "Misplaced Tiles Heuristic";

	public String[] heuristicTypes = { MANHATTAN_DISTANCE_HEURISTIC,
MISPLACED_TILES_HEURISTIC };
	public PriorityQueue<Node> priorityQueue;
	public int numberOfNodesExpanded;
	public int numberOfNodesGenerated;
```

```
	public int stateId;
	public HashMap<Integer, Node> exploredNodeSet;

	/**
	 * This method implements the A* algorithm it takes the initial node and goal
node as input and moves one tile at a time to reach the goal state.
	 * It uses Manhattan Distance and Misplaced tiles heuristic to reach the goal
state. It displays the solution path.
	 * @param initialStateNode Node representing the initial state
	 * @param goalStateNode Node representing the goal state
	 */
	public void aStarProcess(Node initialStateNode, Node goalStateNode) {

		// For each heuristic
		for (int heuristic = 0; heuristic < heuristicTypes.length; heuristic++)
{
			System.out.println("--------- A* for " +
heuristicTypes[heuristic]+ "-----------");
			numberOfNodesExpanded = 0;
			numberOfNodesGenerated = 1;
			stateId = 1;
			priorityQueue = new PriorityQueue<Node>(100, new
FCostComparator());
			exploredNodeSet = new HashMap<Integer, Node>();
			initialStateNode.setgCost(0);
			initialStateNode.setId(stateId++);
			initialStateNode.setParentId(0);
			// Update initial state based on heuristic type
			calculateHeuristics(initialStateNode, goalStateNode,
heuristicTypes[heuristic]);

			priorityQueue.add(initialStateNode);
			while (!priorityQueue.isEmpty()) {
				// extract a node to be expanded
				Node dequeueNode = priorityQueue.poll();
				exploredNodeSet.put(dequeueNode.getId(), dequeueNode);
				numberOfNodesExpanded++;
				// Check if goal State is reached. The goal state will
always have a hcost of zero.
				if (dequeueNode.gethCost() == 0) {
					// if the goal is reached we need to backtrack the
goal state to find the solution path. Creating a stack with the solution path nodes.
					Stack<Node> backtrackSolutionStack = new
Stack<Node>();
					backtrackSolutionStack.push(dequeueNode);
					Node currentInPathNode = dequeueNode;
					int parentId = currentInPathNode.getParentId();
					// Finding solution path by backtracking using
parent id.
					while (parentId != 0) {

	backtrackSolutionStack.push(exploredNodeSet.get(parentId));
						currentInPathNode =
exploredNodeSet.get(parentId);
						parentId = currentInPathNode.getParentId();
```

```
                                }
                                // Print Solution path using nodes from the
backtrack solution stack.
                                System.out.println("------------ Printing Solution
Path --------------");
                                int pathCost = backtrackSolutionStack.size() - 1;
                                printSolutionPathNode(backtrackSolutionStack);
                                System.out.println();
                                System.out.println("Nodes generated: " +
numberOfNodesGenerated);
                                System.out.println("Nodes explored: " +
exploredNodeSet.size());
                                System.out.println("Path Cost: " + pathCost);
                                System.out.println();
                                break;

                        }
                        //Finding the free tile location
                        String indexOfFreeTile =
findingFreeTileLocation(dequeueNode);
                        String[] temp = indexOfFreeTile.split(",");
                        int rowOfFreeTile = Integer.parseInt(temp[0]);
                        int columnOfFreeTile = Integer.parseInt(temp[1]);

                        // Calculating possible child states based on the free
tile location
                        List<String> possiblePositionsForExpansion =
calculatePossiblePositionsForExpansion(dequeueNode,
                                        rowOfFreeTile, columnOfFreeTile);

                        // For each Child
                        for (int i = 0; i < possiblePositionsForExpansion.size();
i++) {
                                // Child node created & generated count increased
                                Node childStateNode = createChildNode(dequeueNode,
rowOfFreeTile, columnOfFreeTile,
                                                possiblePositionsForExpansion.get(i));
                                childStateNode.setId(stateId++);
                                numberOfNodesGenerated++;
                                boolean isChildInExploredSet =
isChildInExploredSet(exploredNodeSet, childStateNode);
                                boolean isChildInPriorityQueue =
isChildInPriorityQueue(priorityQueue, childStateNode);
                                // Checking if child is explored set or Priority
Queue
                                if (!isChildInExploredSet &&
!isChildInPriorityQueue) {
                                        // if yes do nothing if no calculate
heuristic function for child
                                        calculateHeuristics(childStateNode,
goalStateNode, heuristicTypes[heuristic]);
                                        // insert child in queue
                                        priorityQueue.add(childStateNode);
                                }
                        }
```

```
                }
            }
        }

        /**
         * This method updates the heuristic value for the appropriate type of
heuristic. It sets these values in the node of the currentStateNode which is taken as
input.
         *
         * @param currentStateNode is the node representing the current state of the
board.
         * @param goalStateNode is the node representing the expected goal state of
the board.
         * @param heuristicType is the type of heuristic that needs to be calculated.
         */
        private void calculateHeuristics(Node currentStateNode, Node goalStateNode,
String heuristicType) {
            if (heuristicType.equalsIgnoreCase(MANHATTAN_DISTANCE_HEURISTIC)) {
                int manhattanDistanceCost =
manhattanDistanceHeuristic(currentStateNode, goalStateNode);
                currentStateNode.sethCost(manhattanDistanceCost);
                currentStateNode.setfCost(currentStateNode.getgCost() +
currentStateNode.gethCost());

            }
            if (heuristicType.equalsIgnoreCase(MISPLACED_TILES_HEURISTIC)) {
                int misplacedTilesCost =
misplacedTilesHeuristic(currentStateNode, goalStateNode);
                currentStateNode.sethCost(misplacedTilesCost);
                currentStateNode.setfCost(currentStateNode.getgCost() +
currentStateNode.gethCost());

            }
        }

        /**
         * This method calculates the Manhattan Distance Heuristic. It counts the
minimum number of moves required for each tile to reach its goal state position.
         * @param currentStateNode Node representing the current state of the board.
         * @param goalStateNode Node representing the expected goal state
         * @return calculated heuristic cost
         */
        private int manhattanDistanceHeuristic(Node currentStateNode, Node
goalStateNode) {
            int[][] currentState = currentStateNode.getState();
            int[][] goalState = goalStateNode.getState();
            int hcost = 0;
            int i = 0, j = 0, l = 0, m = 0;
            for (int a = 1; a <= 8; a++) {
                outerloop: for (i = 0; i < currentState.length; i++)
                    for (j = 0; j < currentState.length; j++)
                        if (currentState[i][j] == a)
                            break outerloop;
                outerloop1: for (l = 0; l < goalState.length; l++)
                    for (m = 0; m < goalState.length; m++)
```

```
                                if (goalState[l][m] == a)
                                        break outerloop1;
                        hcost = hcost + (Math.abs(i - l) + Math.abs(j - m));
                }
                return hcost;
        }

        /**
         * This method calculates the Misplaced Tiles Heuristic. It counts the number
of tiles that are placed differently from their counterparts in the goal state.
         * @param currentStateNode  Node representing the current state of the board.
         * @param goalStateNode Node representing the expected goal state
         * @return calculated heuristic cost
         */
        private int misplacedTilesHeuristic(Node currentStateNode, Node goalStateNode)
{
                int[][] currentState = currentStateNode.getState();
                int[][] goalState = goalStateNode.getState();
                int hcost = 0;
                for (int i = 0; i < currentState.length; i++)
                        for (int j = 0; j < currentState.length; j++) {
                                if (currentState[i][j] != goalState[i][j] &&
currentState[i][j] > 0) {
                                        hcost++;
                                }
                        }
                return hcost;
        }

        /**
         * This method finds the location of the free tile that is zero.
         * @param currentStateNode Node representing the current state of the board.
         * @return a String which is a comma separated concatenation of the indexes of
the free tile.
         */
        private String findingFreeTileLocation(Node currentStateNode) {
                int[][] currentState = currentStateNode.getState();
                int i = 0, j = 0;
                outerloop: for (i = 0; i < currentState.length; i++) {
                        for (j = 0; j < currentState.length; j++) {
                                if (currentState[i][j] == 0)
                                        break outerloop;
                        }
                }
                String indexOfFreeTile = Integer.toString(i) + "," +
Integer.toString(j);
                return indexOfFreeTile;
        }

        /**
         * This method computes the valid possible positions the free tile can move to
next.
         * @param currentStateNode Node representing the current state of the board.
         * @param rowOfFreeTile Row index of the free tile
         * @param columnOfFreeTile Column index of the free tile.
```

```
	 * @return a list of Strings representing comma separated indexes of valid
possible free tile positions.
	 */
	private List<String> calculatePossiblePositionsForExpansion(Node
currentStateNode, int rowOfFreeTile,
			int columnOfFreeTile) {
		int[][] currentState = currentStateNode.getState();
		List<String> possiblePositionsForExpansion = new ArrayList<String>();
		// DOWN
		if ((rowOfFreeTile + 1) < currentState.length)
			possiblePositionsForExpansion
				.add(Integer.toString(rowOfFreeTile + 1) + "," +
Integer.toString(columnOfFreeTile));
		// UP
		if ((rowOfFreeTile - 1) >= 0)
			possiblePositionsForExpansion
				.add(Integer.toString(rowOfFreeTile - 1) + "," +
Integer.toString(columnOfFreeTile));
		// RIGHT
		if ((columnOfFreeTile + 1) < currentState.length)
			possiblePositionsForExpansion
				.add(Integer.toString(rowOfFreeTile) + "," +
Integer.toString(columnOfFreeTile + 1));
		// LEFT
		if ((columnOfFreeTile - 1) >= 0)
			possiblePositionsForExpansion
				.add(Integer.toString(rowOfFreeTile) + "," +
Integer.toString(columnOfFreeTile - 1));
		return possiblePositionsForExpansion;
	}

	/**
	 * This method creates a child Node for a given possible position of free
tile.
	 * @param dequeueNode  Node representing the parent state of the board from
which the child is to be created.
	 * @param rowOfFreeTile Row index of the free tile
	 * @param columnOfFreeTile Column index of the free tile.
	 * @param possiblePositionsForExpansion a String representing comma separated
indexes of the next valid possible free tile positions.
	 * @return Node representing the child state created by moving one tile from
parent state.
	 */
	private Node createChildNode(Node dequeueNode, int rowOfFreeTile, int
columnOfFreeTile,
			String possiblePositionsForExpansion) {
		// copying state of parent
		Node childNode = new Node();
		int[][] childNodeState = new int[3][3];
		// For each Child clone the current state and replace changed tiles
		copyState(dequeueNode.getState(), childNodeState); // Copy state source
to destination
		String[] temp = possiblePositionsForExpansion.split(",");
		int newRow = Integer.parseInt(temp[0]);
		int newColumn = Integer.parseInt(temp[1]);
```

```
			// move empty tile to new position
			int swapNum = childNodeState[rowOfFreeTile][columnOfFreeTile];
			childNodeState[rowOfFreeTile][columnOfFreeTile] =
childNodeState[newRow][newColumn];
			childNodeState[newRow][newColumn] = swapNum;
			childNode.setState(childNodeState);
			childNode.setgCost(dequeueNode.getgCost() + 1);
			childNode.setParentId(dequeueNode.getId());
			return childNode;
		}

		/** This is a utility method to copy one state to another
		 * @param sourceNodeState state to be copied.
		 * @param destinationNodeState copied state.
		 */
		private void copyState(int[][] sourceNodeState, int[][] destinationNodeState)
{
			for (int i = 0; i < sourceNodeState.length; i++)
				for (int j = 0; j < sourceNodeState.length; j++)
					destinationNodeState[i][j] = sourceNodeState[i][j];

		}

		/**
		 * This method checks if the child node created is present in the explored
set.
		 * @param exploredNodeSet Map containing all the nodes that have been explored
already.
		 * @param childStateNode Node representing the child state which is to be
checked in the explored set.
		 * @return true if the child is in explored set or false if the child is not
in the explored set.
		 */
		private boolean isChildInExploredSet(HashMap<Integer, Node> exploredNodeSet,
Node childStateNode) {
			for (Map.Entry<Integer, Node> entry : exploredNodeSet.entrySet()) {
				Node exploredNode = (Node) entry.getValue();
				if (compareNodesForEquality(childStateNode, exploredNode))
					return true;
			}
			return false;
		}

		/** This method compares two nodes for equality.
		 * @param first
		 * @param second
		 * @return true if nodes are equal and false otherwise
		 */
		private boolean compareNodesForEquality(Node first, Node second) {
			int[][] firstState = first.getState();
			int[][] secondState = second.getState();
			for (int i = 0; i < firstState.length; i++)
				for (int j = 0; j < firstState.length; j++)
					if (firstState[i][j] != secondState[i][j])
						return false;
```

```
			return true;

		}

		/** This method checks if the child node created is present in the priority
queue
		 * @param priorityQueue Priority queue
		 * @param childStateNode Node representing the child state which is to be
checked in the priority queue.
		 * @return true if the child is in priority queue or false if the child is not
in the priority queue.
		 */
		private boolean isChildInPriorityQueue(PriorityQueue<Node> priorityQueue, Node
childStateNode) {
			List<Node> pqList = new ArrayList<Node>(priorityQueue);
			for (int i = 0; i < pqList.size(); i++) {
				if (compareNodesForEquality(pqList.get(i), childStateNode))
					return true;
			}
			return false;
		}

		/**This method prints the solution path from initial state to goal state.
		 * @param backtrackSolutionStack contains the list of nodes that are on the
solution path.
		 */
		private void printSolutionPathNode(Stack<Node> backtrackSolutionStack) {
			Node currentPoppedNode;
			int[][] currentPoppedNodeState;
			while (!backtrackSolutionStack.isEmpty()) {
				currentPoppedNode = backtrackSolutionStack.pop();
				currentPoppedNodeState = currentPoppedNode.getState();
				// Print matrix
				System.out.println("g(n)= " + currentPoppedNode.getgCost() + "\t
h(n)= " + currentPoppedNode.gethCost()
						+ "\t f(n)= " + currentPoppedNode.getfCost());
				for (int i = 0; i < currentPoppedNodeState.length; i++) {
					for (int j = 0; j < currentPoppedNodeState.length; j++)
						System.out.print("\t" +
currentPoppedNodeState[i][j]);
					System.out.println();
				}
				if(!backtrackSolutionStack.isEmpty()) {
					System.out.println();
					System.out.println("Next state: ");
				}
			}
		}
}
```

FComparator.java

This is a comparator created which is used in the Priority Queue to decide the ordering based on the fCost. The priority queue is ordered in ascending order of fcost.

```
import java.util.Comparator;

public class FCostComparator implements Comparator<Node> {

	@Override
	public int compare(Node o1, Node o2) {
		if(o1.getfCost()< o2.getfCost())
			return -1;
		if(o1.getfCost() > o2.getfCost())
			return 1;
		return 0;
	}

}
```

SAMPLE INPUT/OUTPUT CASES

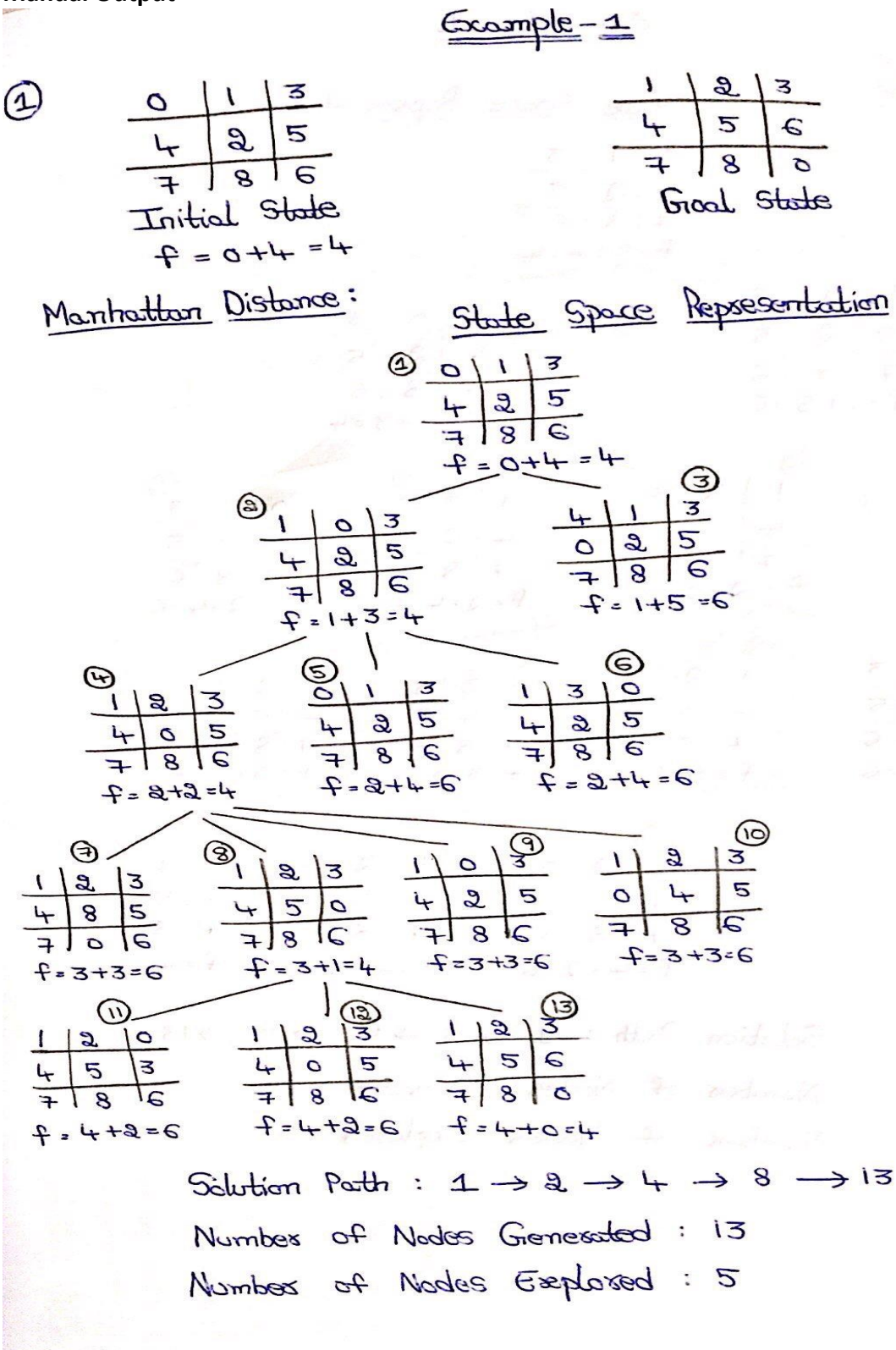
SAMPLE: 1

System Generated Output

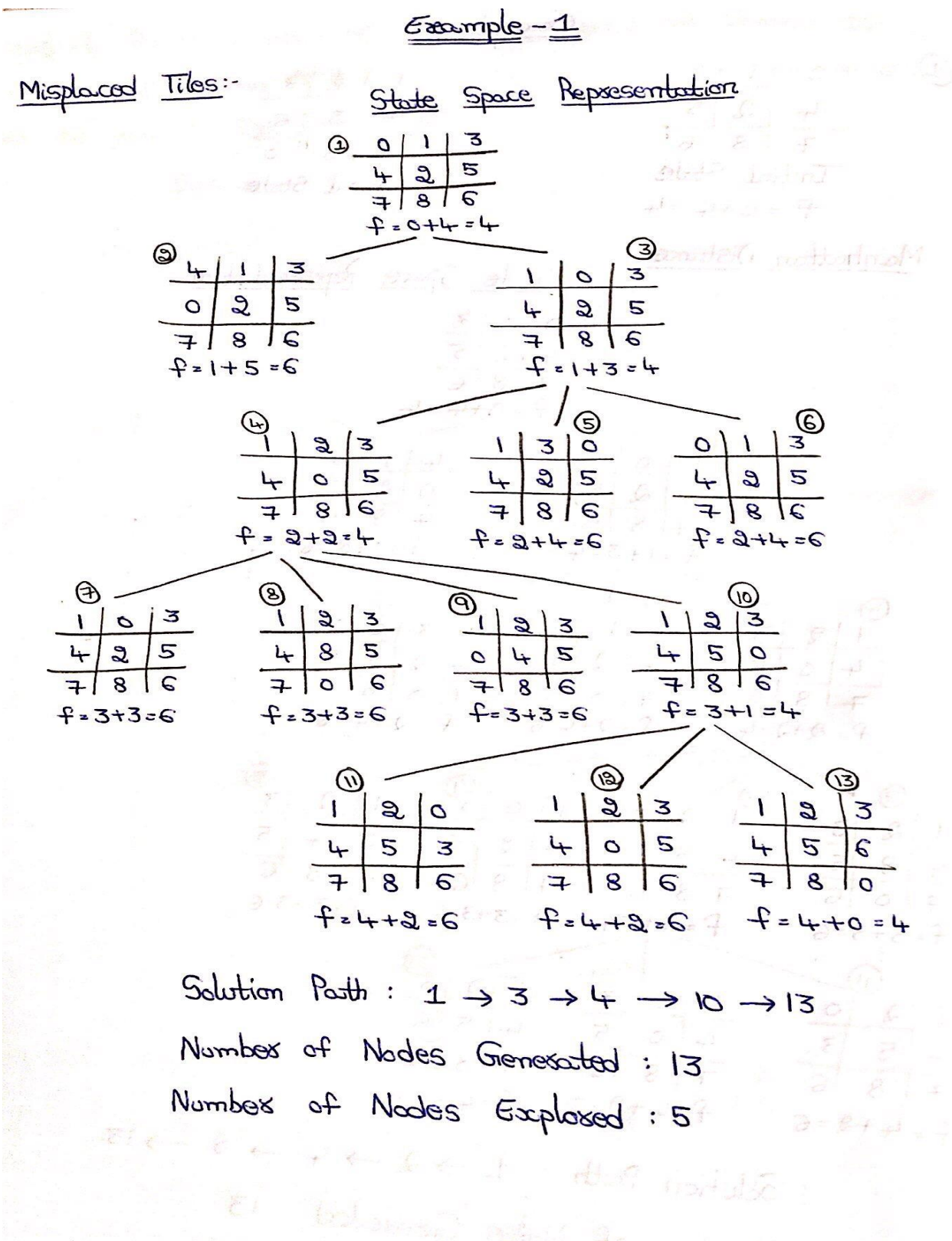
```
 Enter the Initial State (row wise) and seperate each input by enter for the 8-puzzle problem:
0
1
3
4
2
5
7
8
6
 Enter the Goal State (row wise) and seperate each input by enter for the 8-puzzle problem:
1
2
3
4
5
6
7
8
0
```

```
--------- A* for Manhattan Distance Heuristic-----------
------------ Printing Solution Path --------------
g(n)= 0  h(n)= 4          f(n)= 4
        0       1       3
        4       2       5
        7       8       6

Next state:
g(n)= 1  h(n)= 3          f(n)= 4
        1       0       3
        4       2       5
        7       8       6

Next state:
g(n)= 2  h(n)= 2          f(n)= 4
        1       2       3
        4       0       5
        7       8       6

Next state:
g(n)= 3  h(n)= 1          f(n)= 4
        1       2       3
        4       5       0
        7       8       6

Next state:
g(n)= 4  h(n)= 0          f(n)= 4
        1       2       3
        4       5       6
        7       8       0

Nodes generated: 13
Nodes explored: 5
Path Cost: 4


--------- A* for Misplaced Tiles Heuristic-----------
------------ Printing Solution Path --------------
g(n)= 0  h(n)= 4          f(n)= 4
        0       1       3
        4       2       5
        7       8       6

Next state:
g(n)= 1  h(n)= 3          f(n)= 4
        1       0       3
        4       2       5
        7       8       6

Next state:
g(n)= 2  h(n)= 2          f(n)= 4
        1       2       3
        4       0       5
        7       8       6

Next state:
g(n)= 3  h(n)= 1          f(n)= 4
        1       2       3
        4       5       0
        7       8       6

Next state:
g(n)= 4  h(n)= 0          f(n)= 4
        1       2       3
        4       5       6
        7       8       0

Nodes generated: 13
Nodes explored: 5
Path Cost: 4
```

Manual Output



Misplaced Tiles



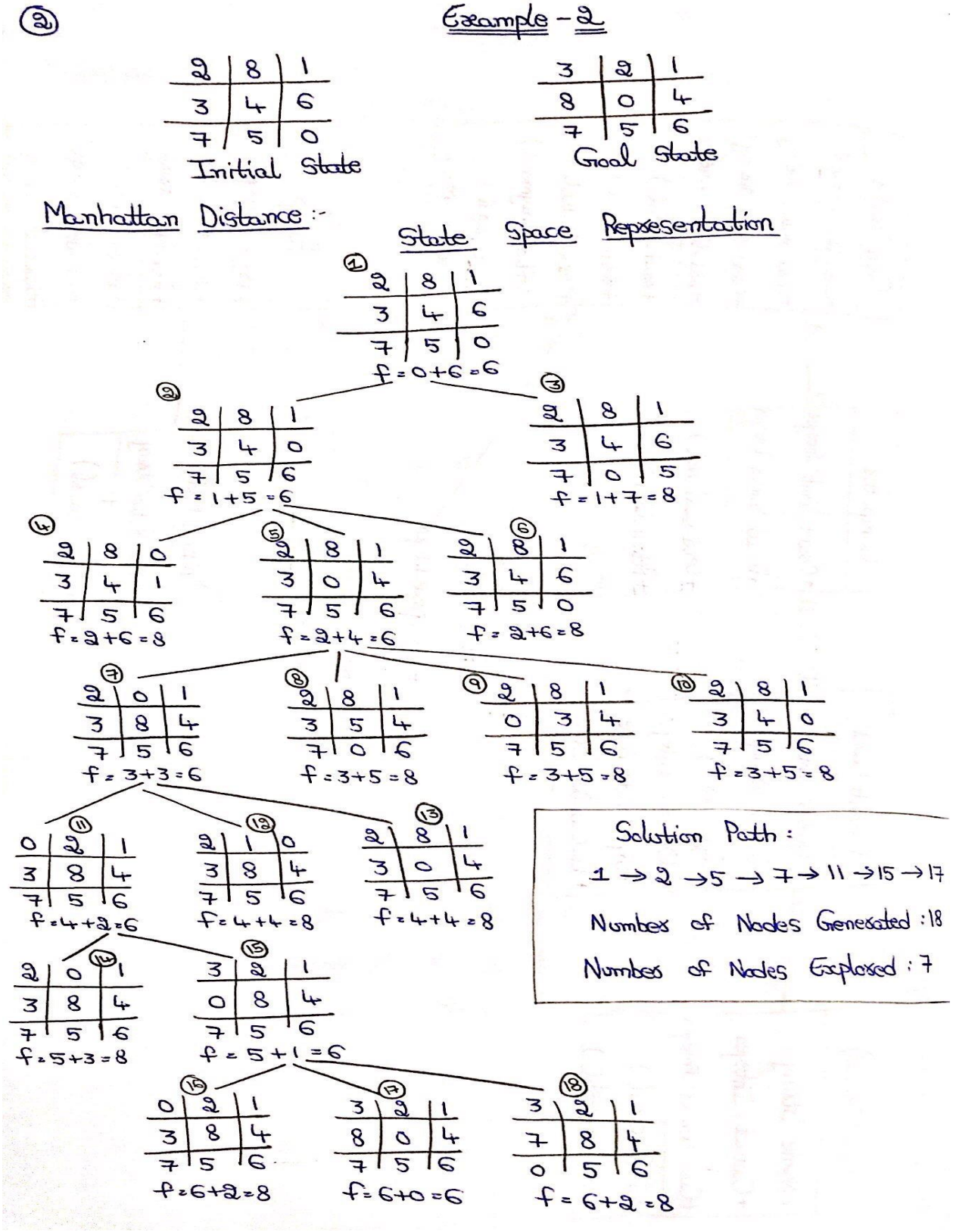
SAMPLE: 2

System Generated Output:

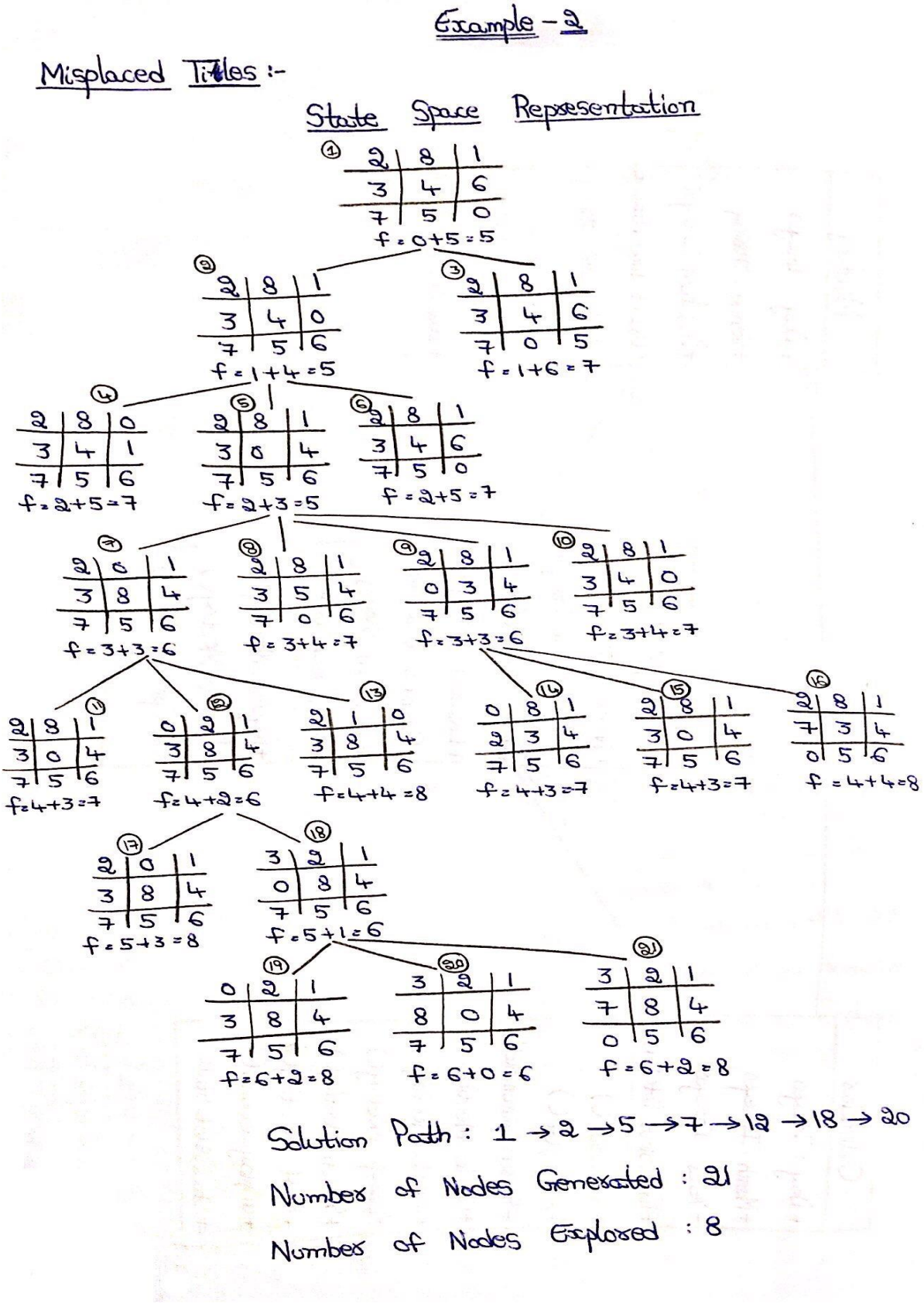
```
 Enter the Initial State (row wise) and seperate each input by enter for the 8-puzzle problem:
2
8
1
3
4
6
7
5
0
 Enter the Goal State (row wise) and seperate each input by enter for the 8-puzzle problem:
3
2
1
8
0
4
7
5
6
--------- A* for Manhattan Distance Heuristic-----------
------------ Printing Solution Path --------------
g(n)= 0  h(n)= 6          f(n)= 6
        2       8       1
        3       4       6
        7       5       0

Next state:
g(n)= 1  h(n)= 5          f(n)= 6
        2       8       1
        3       4       0
        7       5       6

Next state:
g(n)= 2  h(n)= 4          f(n)= 6
        2       8       1
        3       0       4
        7       5       6

Next state:
g(n)= 3  h(n)= 3          f(n)= 6
        2       0       1
        3       8       4
        7       5       6

Next state:
g(n)= 4  h(n)= 2          f(n)= 6
        0       2       1
        3       8       4
        7       5       6


Next state:
g(n)= 5  h(n)= 1          f(n)= 6
        3       2       1
        0       8       4
        7       5       6

Next state:
g(n)= 6  h(n)= 0          f(n)= 6
        3       2       1
        8       0       4
        7       5       6

Nodes generated: 18
Nodes explored: 7
Path Cost: 6
```

```
--------- A* for Misplaced Tiles Heuristic-----------
------------ Printing Solution Path --------------
g(n)= 0  h(n)= 5         f(n)= 5
        2       8       1
        3       4       6
        7       5       0

Next state:
g(n)= 1  h(n)= 4         f(n)= 5
        2       8       1
        3       4       0
        7       5       6

Next state:
g(n)= 2  h(n)= 3         f(n)= 5
        2       8       1
        3       0       4
        7       5       6

Next state:
g(n)= 3  h(n)= 3         f(n)= 6
        2       0       1
        3       8       4
        7       5       6

Next state:
g(n)= 4  h(n)= 2         f(n)= 6
        0       2       1
        3       8       4
        7       5       6

Next state:
g(n)= 5  h(n)= 1         f(n)= 6
        3       2       1
        0       8       4
        7       5       6

Next state:
g(n)= 6  h(n)= 0         f(n)= 6
        3       2       1
        8       0       4
        7       5       6

Nodes generated: 21
Nodes explored: 8
Path Cost: 6
```

Manual Output (Manhattan Distance)



Misplaced Tiles



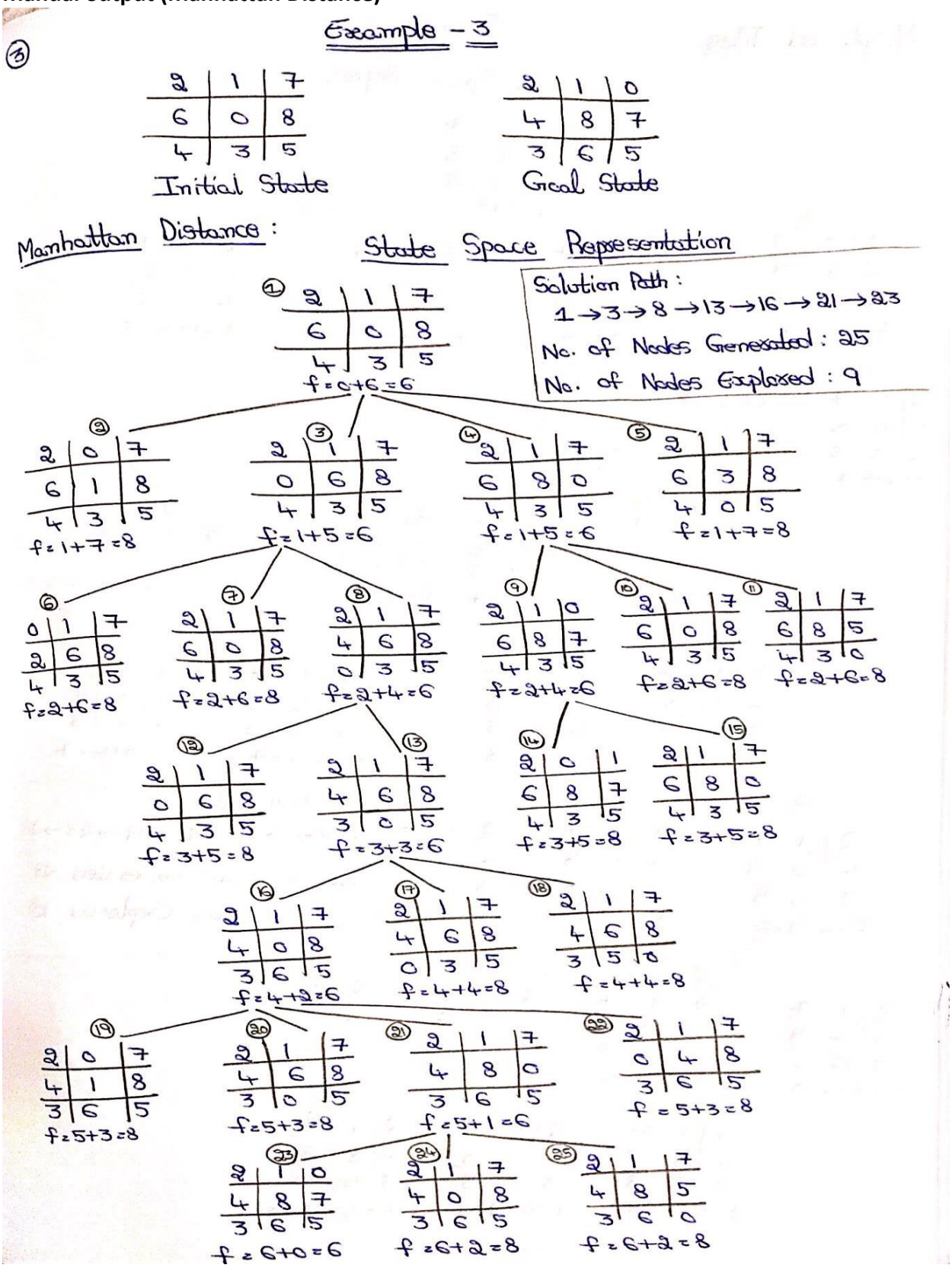
SAMPLE: 3

System Generated Output

```
 Enter the Initial State (row wise) and seperate each input by enter for the 8-puzzle problem:
2
1
7
6
0
8
4
3
5
 Enter the Goal State (row wise) and seperate each input by enter for the 8-puzzle problem:
2
1
0
4
8
7
3
6
5
--------- A* for Manhattan Distance Heuristic-----------
------------ Printing Solution Path --------------
g(n)= 0  h(n)= 6         f(n)= 6
        2       1       7
        6       0       8
        4       3       5

Next state:
g(n)= 1  h(n)= 5         f(n)= 6
        2       1       7
        0       6       8
        4       3       5

Next state:
g(n)= 2  h(n)= 4         f(n)= 6
        2       1       7
        4       6       8
        0       3       5

Next state:
g(n)= 3  h(n)= 3         f(n)= 6
        2       1       7
        4       6       8
        3       0       5

Next state:
g(n)= 4  h(n)= 2         f(n)= 6
        2       1       7
        4       0       8
        3       6       5

Next state:
g(n)= 5  h(n)= 1         f(n)= 6
        2       1       7
        4       8       0
        3       6       5

Next state:
g(n)= 6  h(n)= 0         f(n)= 6
        2       1       0
        4       8       7
        3       6       5

Nodes generated: 25
Nodes explored: 9
Path Cost: 6
```

```
--------- A* for Misplaced Tiles Heuristic-----------
------------ Printing Solution Path --------------
g(n)= 0  h(n)= 5         f(n)= 5
        2       1       7
        6       0       8
        4       3       5

Next state:
g(n)= 1  h(n)= 5         f(n)= 6
        2       1       7
        0       6       8
        4       3       5

Next state:
g(n)= 2  h(n)= 4         f(n)= 6
        2       1       7
        4       6       8
        0       3       5

Next state:
g(n)= 3  h(n)= 3         f(n)= 6
        2       1       7
        4       6       8
        3       0       5

Next state:
g(n)= 4  h(n)= 2         f(n)= 6
        2       1       7
        4       0       8
        3       6       5

Next state:
g(n)= 5  h(n)= 1         f(n)= 6
        2       1       7
        4       8       0
        3       6       5

Next state:
g(n)= 6  h(n)= 0         f(n)= 6
        2       1       0
        4       8       7
        3       6       5

Nodes generated: 28
Nodes explored: 10
Path Cost: 6
```

Manual output (Manhattan Distance)

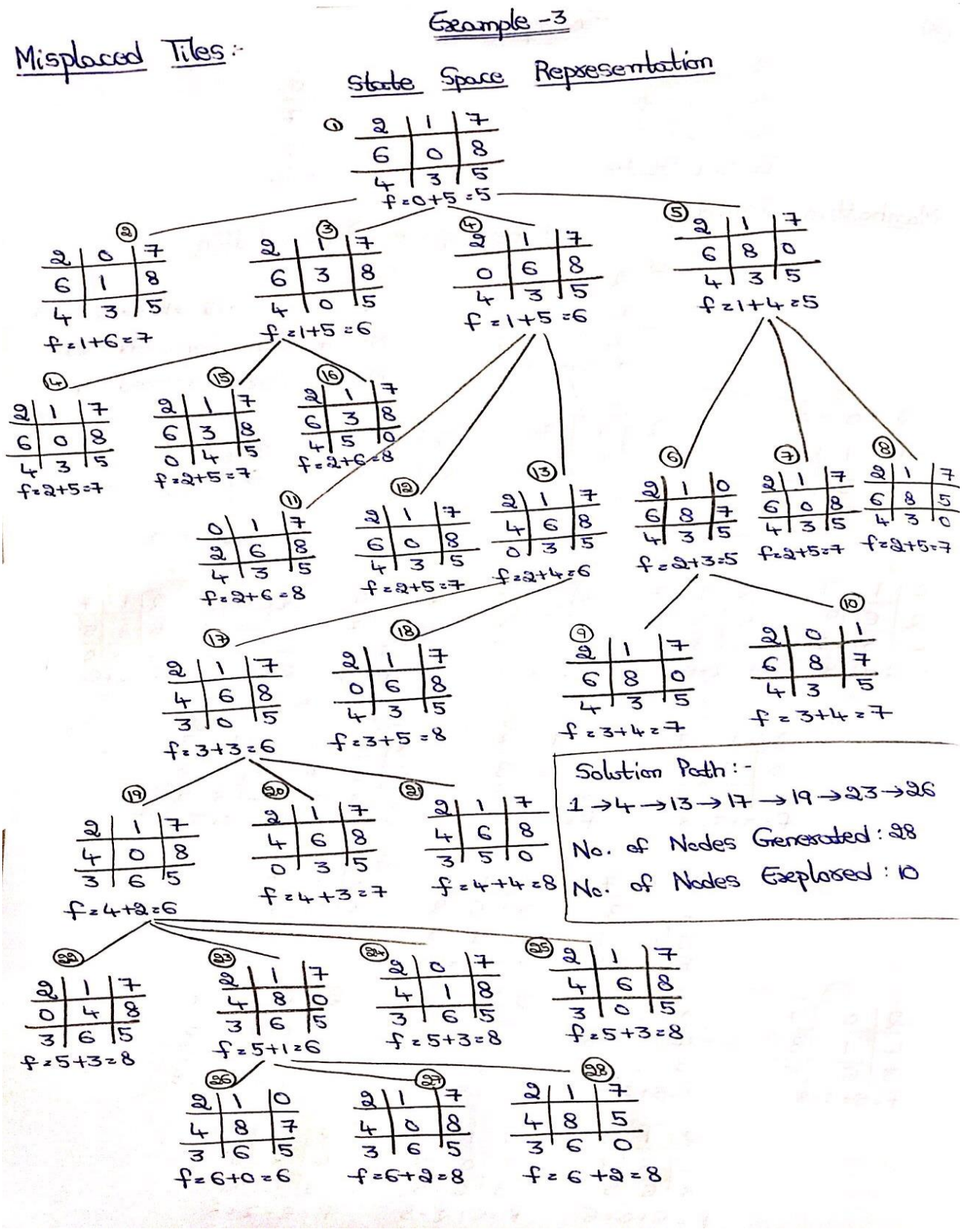


Misplaced Tiles

Example -3

Misplaced Tiles :-

State Space Representation



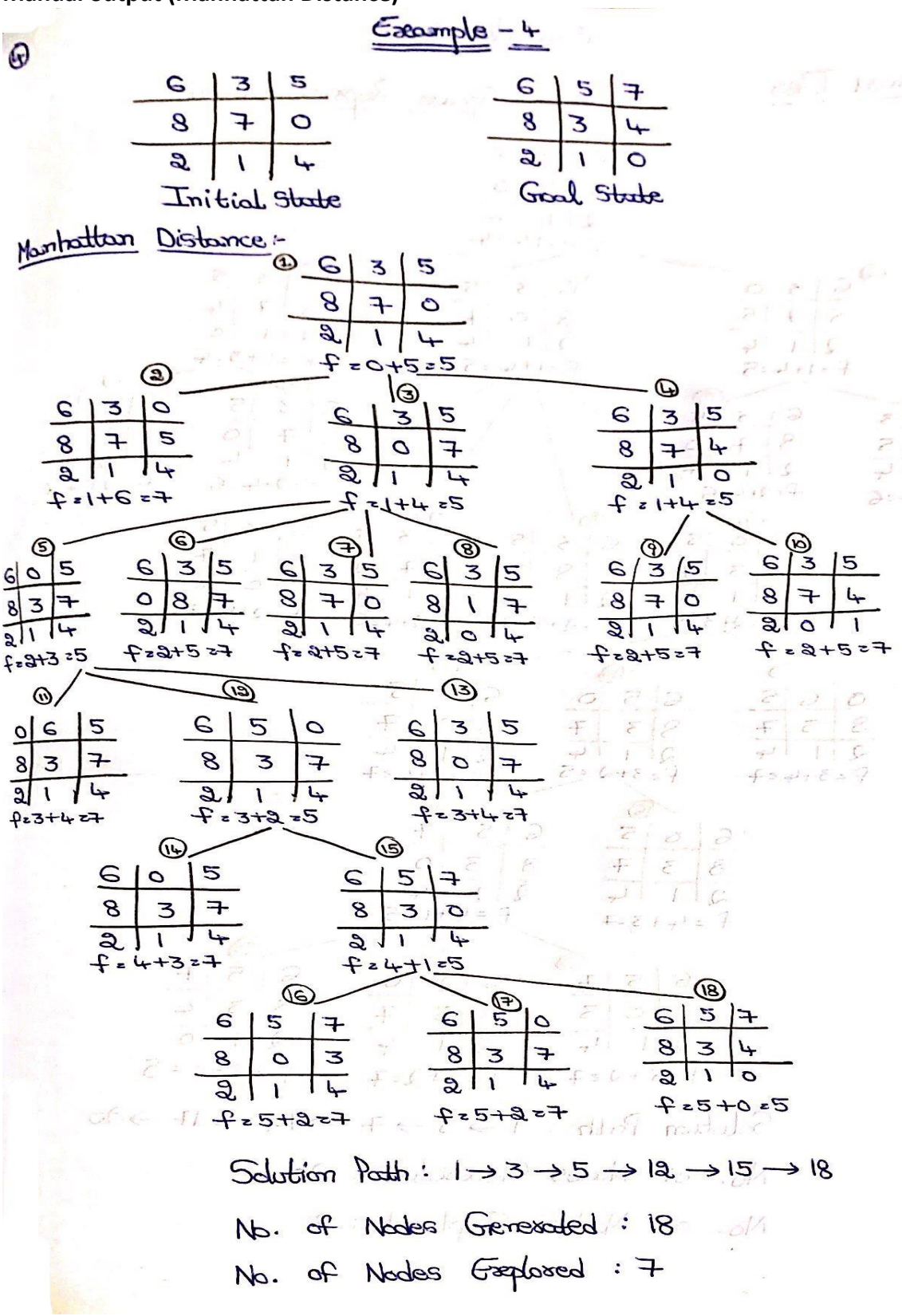
SAMPLE: 4

System Generated Output

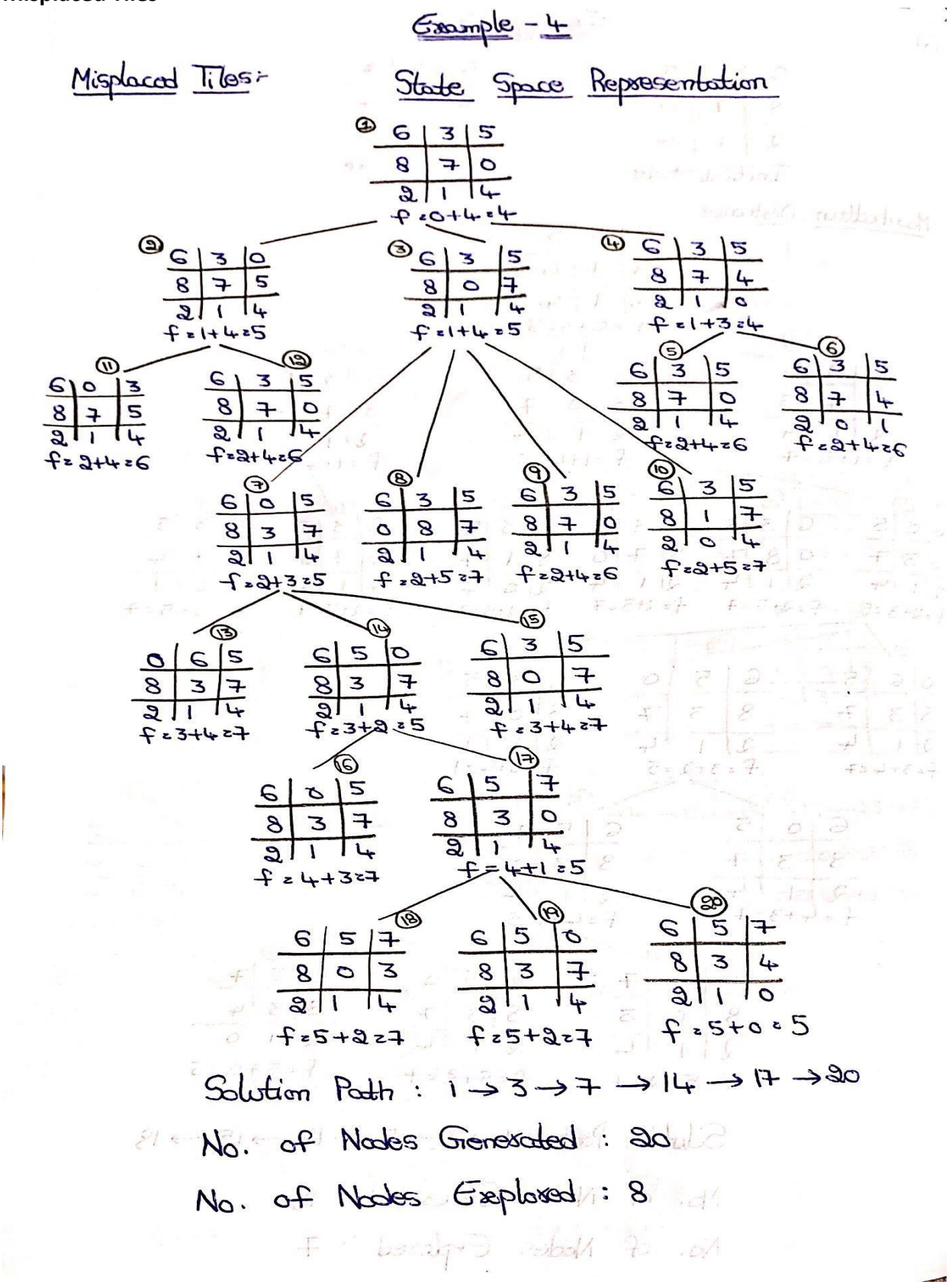
```
 Enter the Initial State (row wise) and seperat<terminated> AStarFramework [Java Application] C:\Program Files\J
6
3
5
8
7
0
2
1
4
 Enter the Goal State (row wise) and seperate each input by enter for the 8-puzzle problem:
6
5
7
8
3
4
2
1
0
--------- A* for Manhattan Distance Heuristic-----------
------------ Printing Solution Path --------------
g(n)= 0  h(n)= 5         f(n)= 5
        6       3       5
        8       7       0
        2       1       4

Next state:
g(n)= 1  h(n)= 4         f(n)= 5
        6       3       5
        8       0       7
        2       1       4

Next state:
g(n)= 2  h(n)= 3         f(n)= 5
        6       0       5
        8       3       7
        2       1       4

Next state:
g(n)= 3  h(n)= 2         f(n)= 5
        6       5       0
        8       3       7
        2       1       4

Next state:
g(n)= 4  h(n)= 1         f(n)= 5
        6       5       7
        8       3       0
        2       1       4


Next state:
g(n)= 5  h(n)= 0         f(n)= 5
        6       5       7
        8       3       4
        2       1       0

Nodes generated: 18
Nodes explored: 7
Path Cost: 5
```

```
--------- A* for Misplaced Tiles Heuristic-----------
------------ Printing Solution Path --------------
g(n)= 0  h(n)= 4         f(n)= 4
        6       3       5
        8       7       0
        2       1       4

Next state:
g(n)= 1  h(n)= 4         f(n)= 5
        6       3       5
        8       0       7
        2       1       4

Next state:
g(n)= 2  h(n)= 3         f(n)= 5
        6       0       5
        8       3       7
        2       1       4

Next state:
g(n)= 3  h(n)= 2         f(n)= 5
        6       5       0
        8       3       7
        2       1       4

Next state:
g(n)= 4  h(n)= 1         f(n)= 5
        6       5       7
        8       3       0
        2       1       4

Next state:
g(n)= 5  h(n)= 0         f(n)= 5
        6       5       7
        8       3       4
        2       1       0

Nodes generated: 20
Nodes explored: 8
Path Cost: 5
```

Manual output (Manhattan Distance)



Misplaced Tiles



RESULTS

Initial and Goal States	Misplaced Tiles Heuristic	Manhattan Distance Heuristic
Initial State 0 1 3 4 2 5 7 8 6 **Goal State** 1 2 3 4 5 6 7 8 0	Number of Nodes Generated: **13** Number of Nodes Expanded: **5**	Number of Nodes Generated: **13** Number of Nodes Expanded: **5**
Initial State 2 8 1 3 4 6 7 5 0 **Goal State** 3 2 1 8 0 4 7 5 6	Number of Nodes Generated: **21** Number of Nodes Expanded: **8**	Number of Nodes Generated: **18** Number of Nodes Expanded: **7**
Initial State 2 1 7 6 0 8 4 3 5 **Goal State** 2 1 0 4 8 7 3 6 5	Number of Nodes Generated: **25** Number of Nodes Expanded: **9**	Number of Nodes Generated: **28** Number of Nodes Expanded: **10**
Initial State 6 3 5 8 7 0 2 1 4 **Goal State** 6 5 7 8 3 4 2 1 0	Number of Nodes Generated: **18** Number of Nodes Expanded: **7**	Number of Nodes Generated: **20** Number of Nodes Expanded: **8**

Observations:

From the above experimental results, we see that the number of nodes generated when Misplaced Tile heuristic is used are more compared to Manhattan Distance heuristic.