Rahul Ratra

Mohit Varma

Kalidindi Grishma Hari Vallabhi

8 Puzzle using A\* search

Rahul Ratra

Mohit Varma

8 Puzzle using A\* search

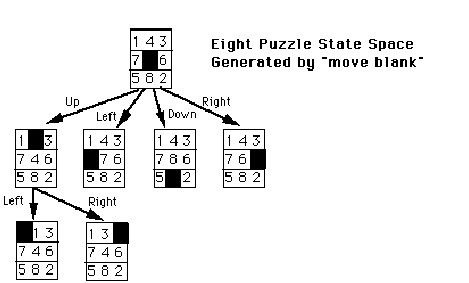
**Purpose**:

The purpose of the project is to solve the 8 puzzle problem using A star (\*) search technique to find the best possible way to solve the puzzle.

**State space representation:** A state space essentially consists of a set of nodes representing each state of the problem, arcs between nodes representing the legal moves from one state to another, an initial state and a goal state. The state space is searched to find a solution to the problem. Here 0 represents the blank position (space) on the board.

• In the state space representation of the problem:

* **Nodes** of a graph correspond to partial problem solution **states.**
* **Arcs** correspond to **steps (application of operators)**  in a problem solving process - The operators can be thought of in terms of the direction that the blank space effectively moves. i.e**. up, down, left, right**
* **The root** of the graph corresponds to the **initial state** of the problem.
* **The goal** node is a leaf node which corresponds to a **goal state**



**A\* Search Algorithm:**

It is an informed search algorithm or a best first search meaning that it is formulated in terms of weighted graphs : starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied. At each iteration of its main loop, A\* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A\* selects the path that minimizes f(n)=g(n)+h(n)

where n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal. Two different examples of admissible heuristics :

Hamming distance: The number of Misplaced Tiles

Manhattan distance: The distance between two points measured along axes at right angles.

**Problem Implementation Details**:

* + We have used Java Programming Language to implement it.

**Classes Used**:

1. **EightPuzzle**

Description:

* + This class is responsible for triggering the 8 puzzle application problem
  + It takes in user input for start puzzle states and goal puzzle states  Prints out the following

◦ the g(n) i.e. the distances the puzzle has traveled from its start state for every expansion

◦ f(n) i.e. A star distance calculated which is the sum of g(n) and the heuristic function calculation.

**Global variables:**

* + int[][] goalState
  + PriorityQueue<EightPuzzleBeanH1> priorityQueueH1
  + PriorityQueue<EightPuzzleBeanH2> priorityQueueH2
  + ArrayList<EightPuzzleBeanH1> expandedNodesH1
  + ArrayList<EightPuzzleBeanH2> expandedNodesH2

**Functions**:

* + calculateBasedOnMisplacedTiles(**int**[][] startState)
  + calculateBasedOnManhattanDistance(**int**[][] startState)
  + processPuzzlePlayH1(EightPuzzleBeanH1 move)
  + processPuzzlePlayH2(EightPuzzleBeanH2 move)

1. **EightPuzzleUtil**

Description:

* + This class can be described as the utilities class for the Eight Puzzle game.

**Global Variables:**

* + THRESHOLD

**Functions:**

* + printMoveStats(T EightPuzzleBean)
  + generate2dArrayFromUserInput(Scanner input)
  + movePiece(int[][] currentPuzzlePlay, int row1, column1, int row2, int column2)

1. **EightPuzzleBean**

Description:

* + This is the parent class to both EightPuzzleBeanH1 (Missing Tiles Calculation) and EightPuzzleBeanH2 (Manhattan Distance Calculation)

**Global variables:**

* + int aStarDistance
  + int[][] stateOfPuzzle
  + int level

1. **EightPuzzleBeanH1** Description:
   * The class performance the movements of the 8 puzzle game while at the same time generating the child nodes for each move.
   * The best move for each level is calculated on the basis of the Misplaced Tiles concept where the number of misplaced tiles of the current puzzle state and the goal state are compared.

**Functions**:

* + calculateMisplacedTilesDistance()
  + generatePossibleStates(EightPuzzleBeanH1 parentNode)
  + addChildNodesToList(EightPuzzleBeanH1 parentNode, ArrayList<EightPuzzleBeanH1> childNodes, int[][] a)
  + compareTo(EightPuzzleBeanH1 EightPuzzleBean)

1. **EightPuzzleBeanH2** Description:
   * The class performance the movements of the 8 puzzle game while at the same time generating the child nodes for each move.
   * The best move for each level is calculated on the basis of the Manhattan Distance concept where the distance between the current tile on the puzzle is compared with the location on the goal state using Heuristic 2 (H2) function.

**Functions**:

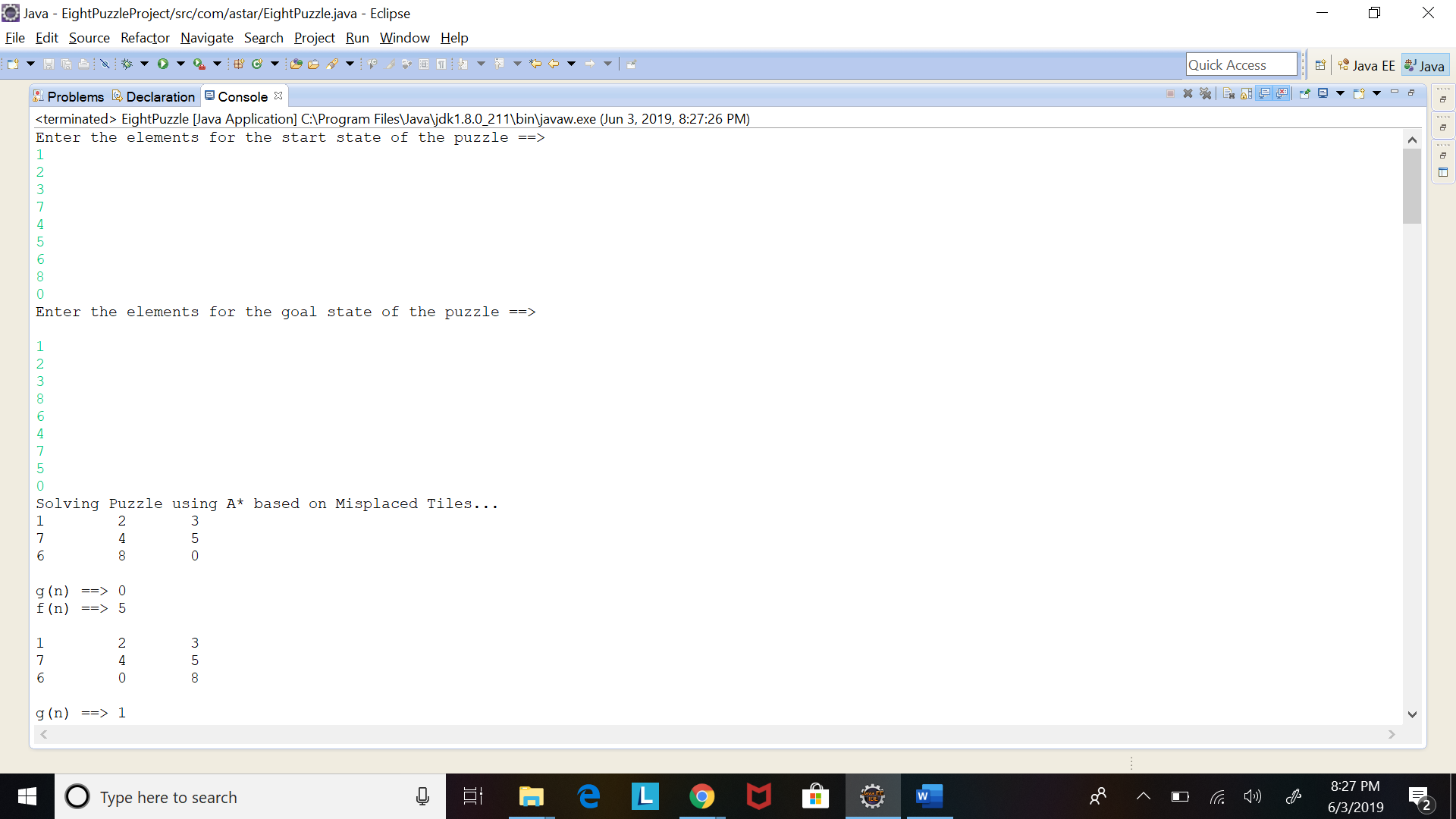
* + calculateManhattanDistance()
  + checkPosition(int[][] currentPuzzleState, int element)
  + generatePossibleStates(EightPuzzleBeanH2 parentNode)
  + addChildNodesToList(EightPuzzleBeanH2 parentNode, ArrayList<EightPuzzleBeanH2> childNodes, int[][] a)
  + compareTo(EightPuzzleBeanH2 EightPuzzleBean)

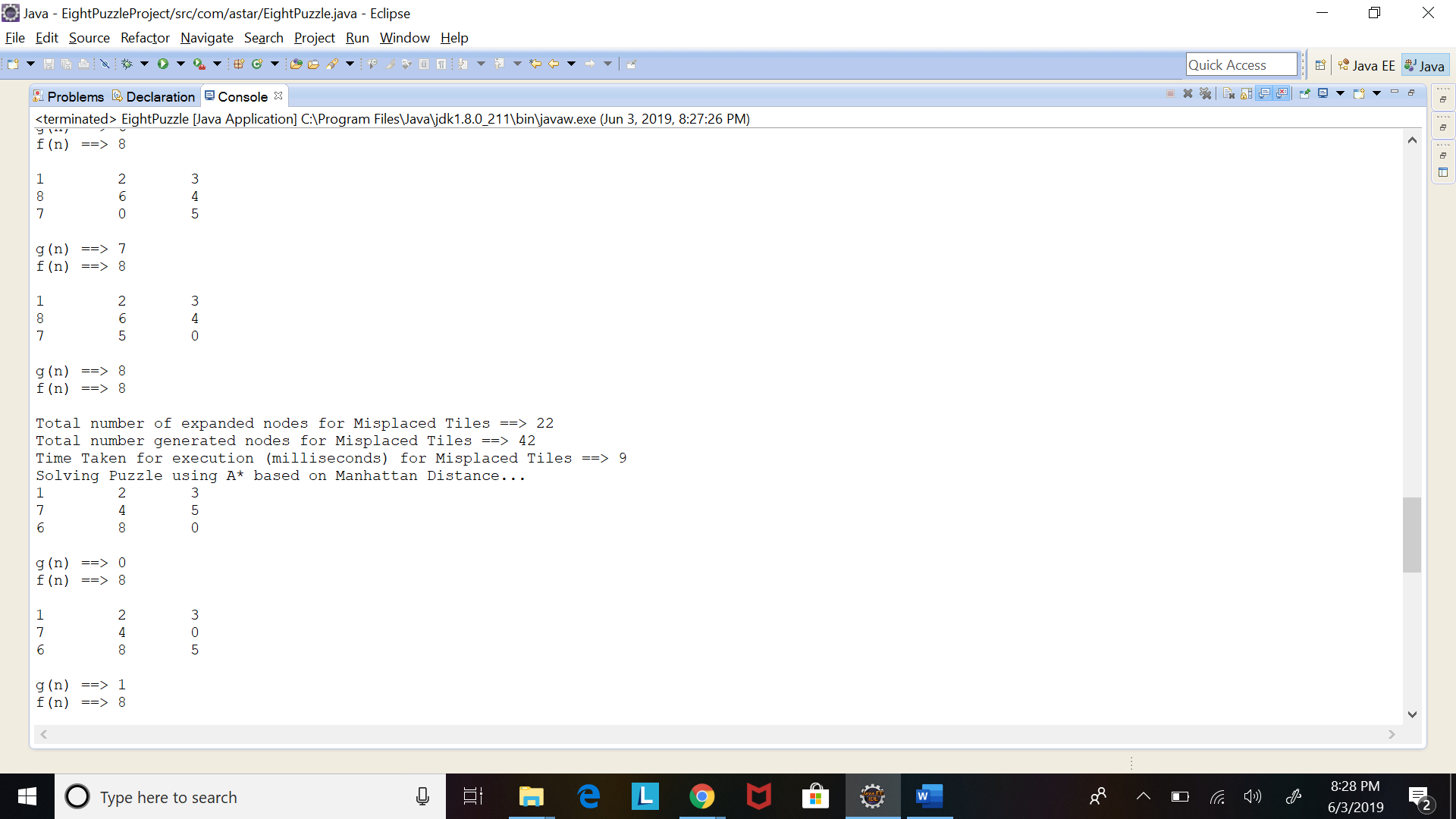
**Test Cases:**

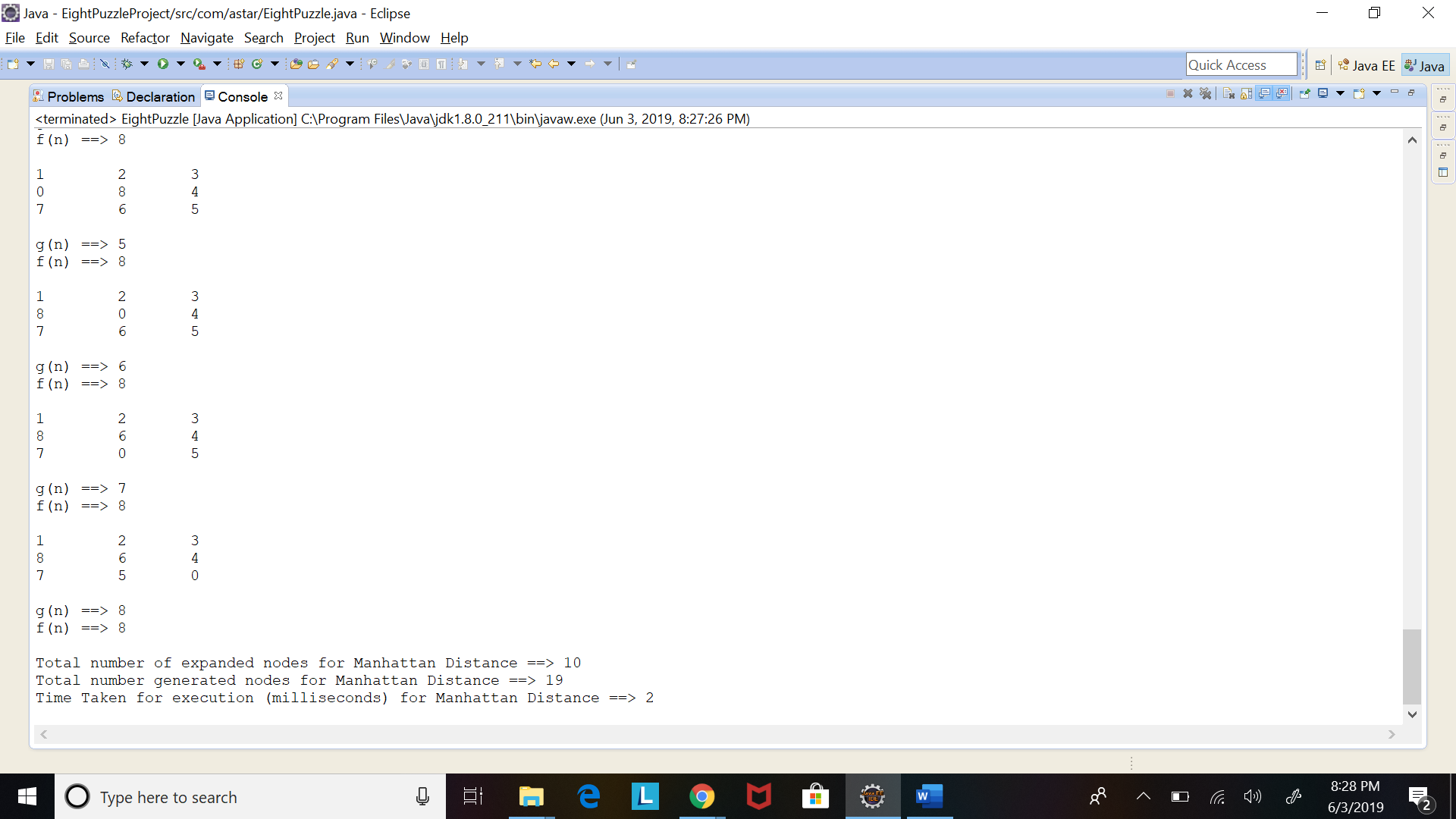
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S.No** | **Initial State** | **Goal State** | **Manhattan Distance**  **Heuristic** | | **Misplaced Tiles Heuristic** | |
|  |  |  | **Nodes Generated** | **Nodes Expanded** | **Nodes Generated** | **Nodes Expanded** |
| 1 | 1 2 3  7 4 5  6 8 0 | 1 2 3  8 6 4  7 5 0 | 19 | 10 | 42 | 22 |
| 2 | 1. 8 1 2. 4 6   7 5 0 | 3 2 1  8 0 4  7 5 6 | 13 | 7 | 15 | 8 |
| 3 | 1 2 3  0 4 6  7 5 8 | 1 2 3  4 5 6  7 8 0 | 9 | 4 | 9 | 4 |
| 4 | 1 2 3  8 0 4  7 6 5 | 2 8 1  0 4 3  7 6 5 | 44 | 25 | 67 | 38 |

**Solution Paths:**

**Test Case 1:**







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

Solving Puzzle using A\* based on Misplaced Tiles...

1 2 3

7 4 5

6 8 0

g(n) ==> 0

f(n) ==> 5

1 2 3

7 4 5

6 0 8

g(n) ==> 1

f(n) ==> 6

1 2 3

7 4 0

6 8 5

g(n) ==> 1

f(n) ==> 6

1 2 3

7 0 4

6 8 5

g(n) ==> 2

f(n) ==> 6

1 2 3

7 4 5

0 6 8

g(n) ==> 2

f(n) ==> 7

1 2 3

7 8 4

6 0 5

g(n) ==> 3

f(n) ==> 7

1 2 3

0 4 5

7 6 8

g(n) ==> 3

f(n) ==> 7

1 2 3

7 8 4

6 5 0

g(n) ==> 4

f(n) ==> 7

1 2 3

0 7 4

6 8 5

g(n) ==> 3

f(n) ==> 7

1 2 3

7 0 5

6 4 8

g(n) ==> 2

f(n) ==> 7

1 2 0

7 4 3

6 8 5

g(n) ==> 2

f(n) ==> 8

1 2 3

6 7 4

0 8 5

g(n) ==> 4

f(n) ==> 8

1 2 3

0 7 5

6 4 8

g(n) ==> 3

f(n) ==> 8

1 0 3

7 2 4

6 8 5

g(n) ==> 3

f(n) ==> 8

1 2 3

7 5 0

6 4 8

g(n) ==> 3

f(n) ==> 8

1 2 3

7 8 4

0 6 5

g(n) ==> 4

f(n) ==> 8

1 2 3

4 0 5

7 6 8

g(n) ==> 4

f(n) ==> 8

1 2 3

0 8 4

7 6 5

g(n) ==> 5

f(n) ==> 8

1 2 3

4 6 5

7 0 8

g(n) ==> 5

f(n) ==> 8

1 2 3

8 0 4

7 6 5

g(n) ==> 6

f(n) ==> 8

1 2 3

8 6 4

7 0 5

g(n) ==> 7

f(n) ==> 8

1 2 3

8 6 4

7 5 0

g(n) ==> 8

f(n) ==> 8

Total number of expanded nodes for Misplaced Tiles ==> 22

Total number generated nodes for Misplaced Tiles ==> 42

Time Taken for execution (milliseconds) for Misplaced Tiles ==> 7

Solving Puzzle using A\* based on Manhattan Distance...

1 2 3

7 4 5

6 8 0

g(n) ==> 0

f(n) ==> 8

1 2 3

7 4 0

6 8 5

g(n) ==> 1

f(n) ==> 8

1 2 3

7 0 4

6 8 5

g(n) ==> 2

f(n) ==> 8

1 2 3

7 8 4

6 0 5

g(n) ==> 3

f(n) ==> 8

1 2 3

7 8 4

0 6 5

g(n) ==> 4

f(n) ==> 8

1 2 3

7 8 4

6 5 0

g(n) ==> 4

f(n) ==> 8

1 2 3

0 8 4

7 6 5

g(n) ==> 5

f(n) ==> 8

1 2 3

8 0 4

7 6 5

g(n) ==> 6

f(n) ==> 8

1 2 3

8 6 4

7 0 5

g(n) ==> 7

f(n) ==> 8

1 2 3

8 6 4

7 5 0

g(n) ==> 8

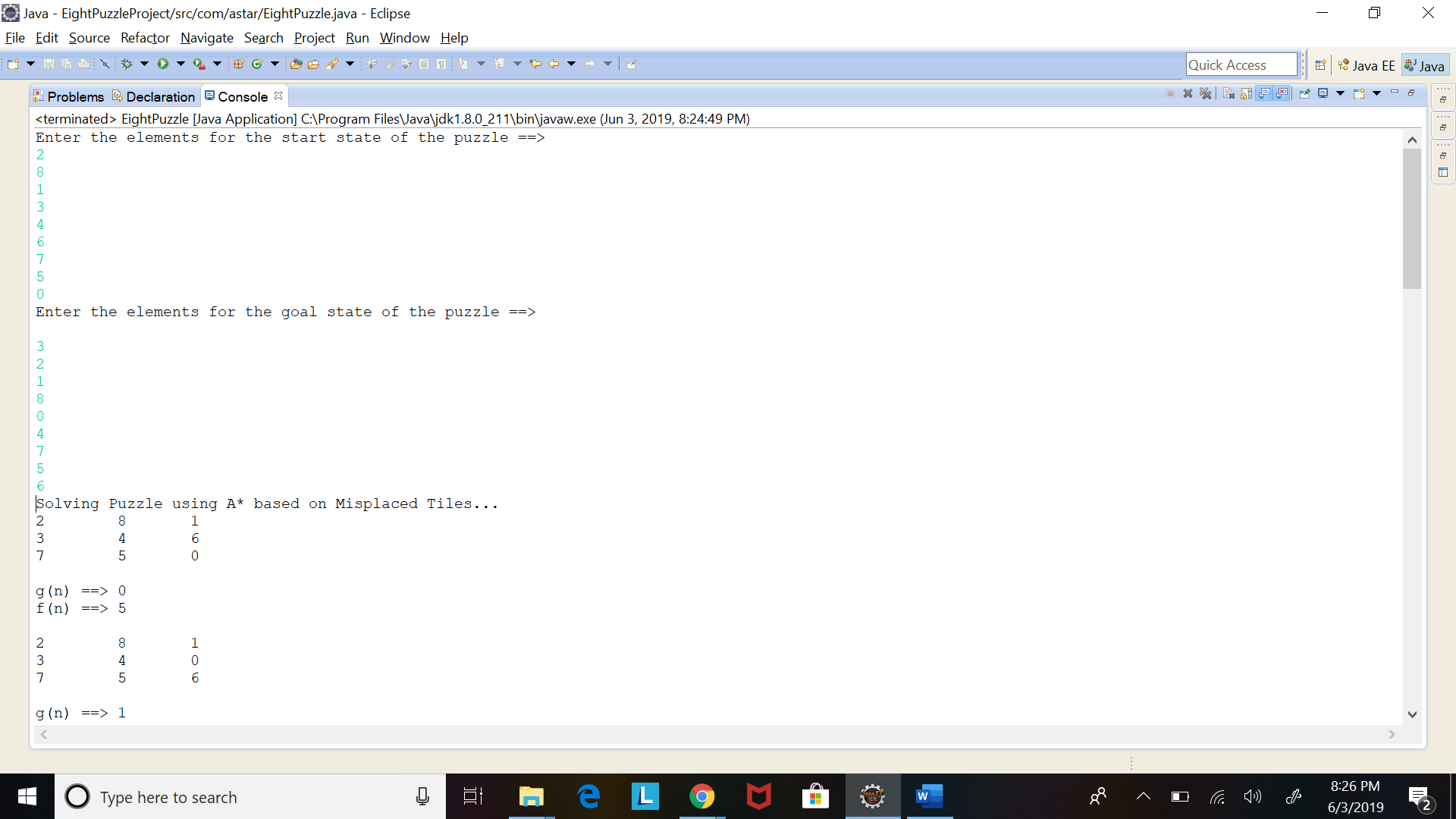
f(n) ==> 8

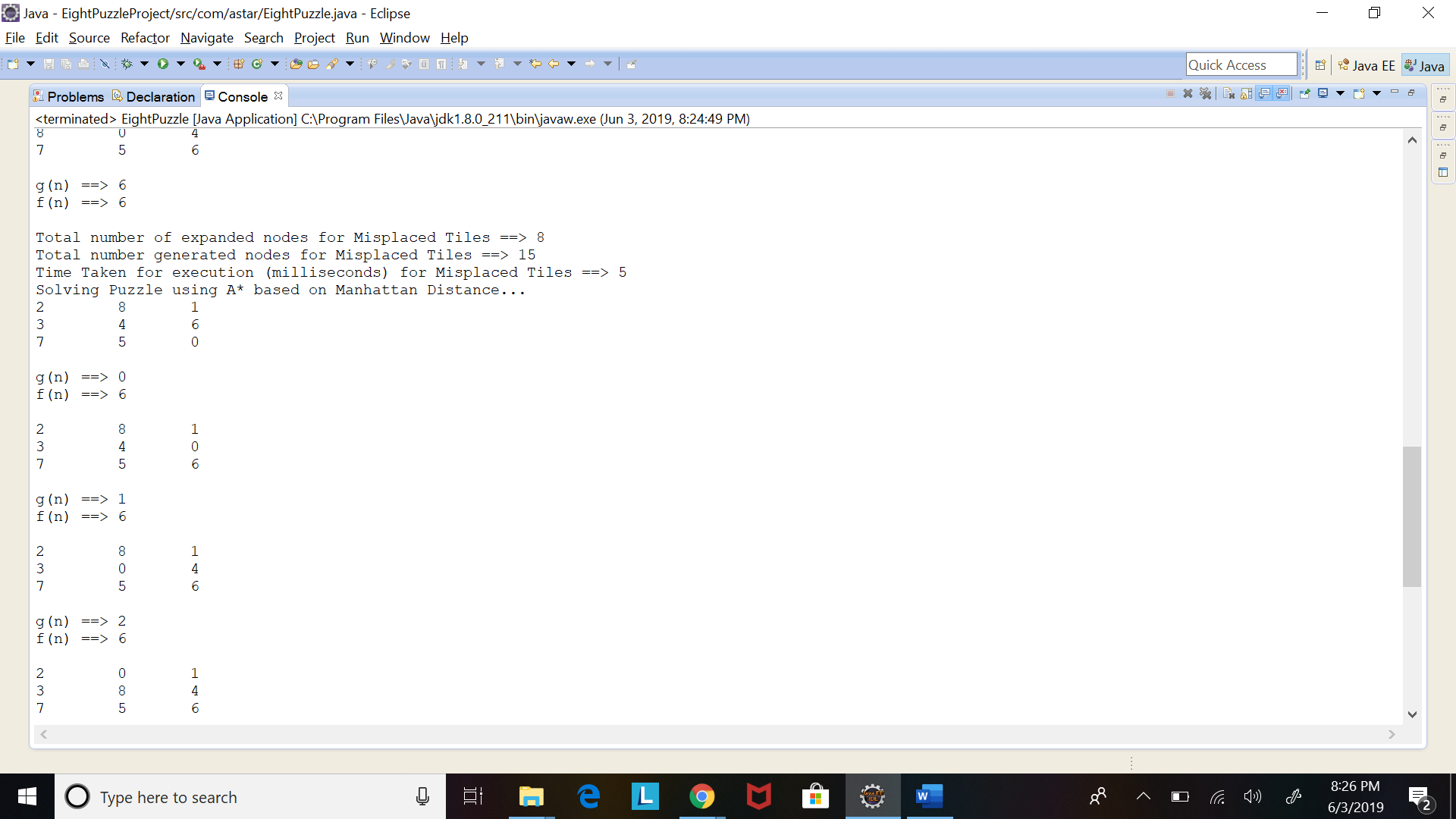
Total number of expanded nodes for Manhattan Distance ==> 10

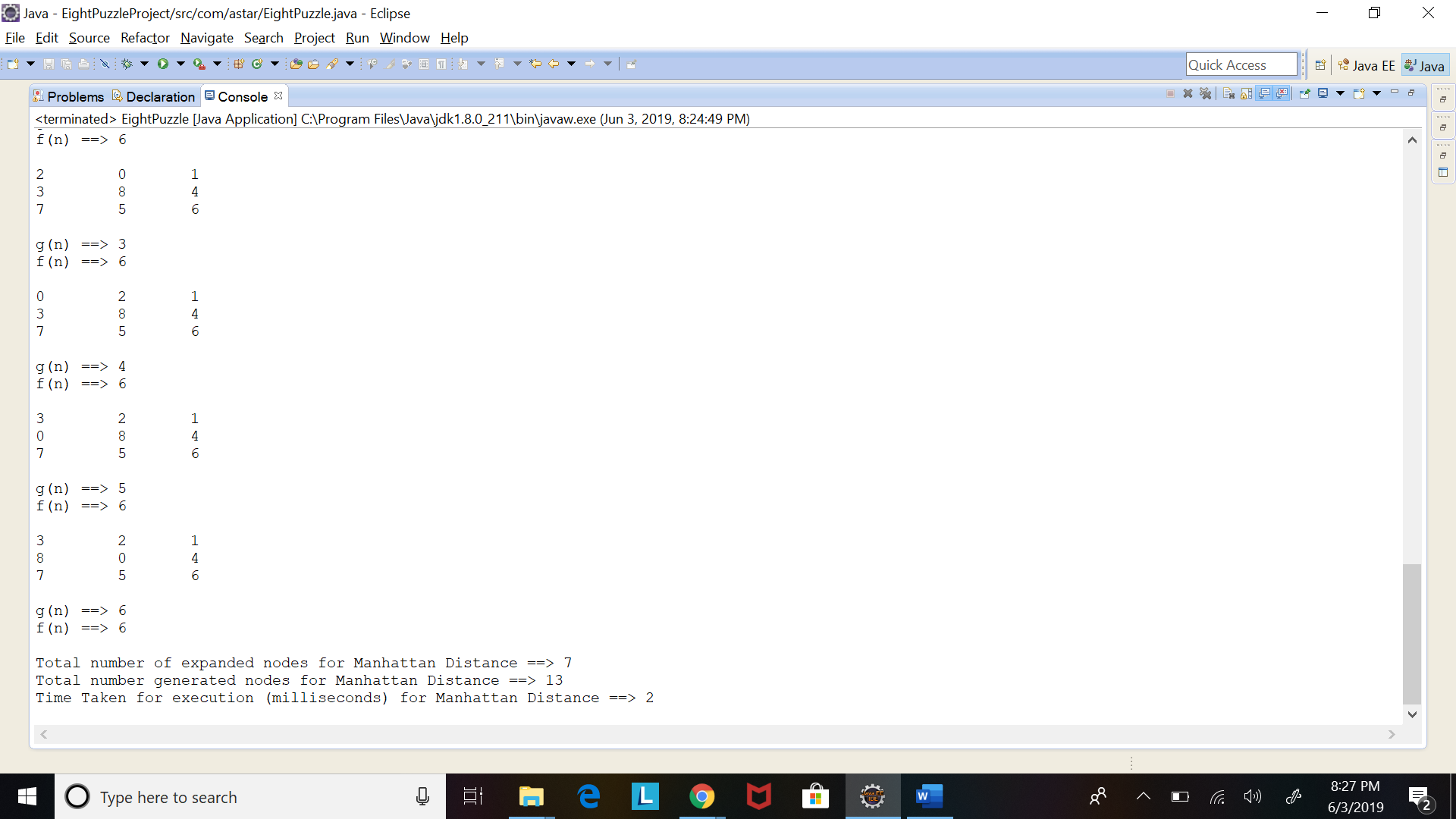
Total number generated nodes for Manhattan Distance ==> 19

Time Taken for execution (milliseconds) for Manhattan Distance ==> 2

# **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

Solving Puzzle using A\* based on Misplaced Tiles...

2 8 1

3 4 6

7 5 0

g(n) ==> 0

f(n) ==> 5

2 8 1

3 4 0

7 5 6

g(n) ==> 1

f(n) ==> 5

2 8 1

3 0 4

7 5 6

g(n) ==> 2

f(n) ==> 5

2 8 1

0 3 4

7 5 6

g(n) ==> 3

f(n) ==> 6

2 0 1

3 8 4

7 5 6

g(n) ==> 3

f(n) ==> 6

0 2 1

3 8 4

7 5 6

g(n) ==> 4

f(n) ==> 6

3 2 1

0 8 4

7 5 6

g(n) ==> 5

f(n) ==> 6

3 2 1

8 0 4

7 5 6

g(n) ==> 6

f(n) ==> 6

Total number of expanded nodes for Misplaced Tiles ==> 8

Total number generated nodes for Misplaced Tiles ==> 15

Time Taken for execution (milliseconds) for Misplaced Tiles ==> 4

Solving Puzzle using A\* based on Manhattan Distance...

2 8 1

3 4 6

7 5 0

g(n) ==> 0

f(n) ==> 6

2 8 1

3 4 0

7 5 6

g(n) ==> 1

f(n) ==> 6

2 8 1

3 0 4

7 5 6

g(n) ==> 2

f(n) ==> 6

2 0 1

3 8 4

7 5 6

g(n) ==> 3

f(n) ==> 6

0 2 1

3 8 4

7 5 6

g(n) ==> 4

f(n) ==> 6

3 2 1

0 8 4

7 5 6

g(n) ==> 5

f(n) ==> 6

3 2 1

8 0 4

7 5 6

g(n) ==> 6

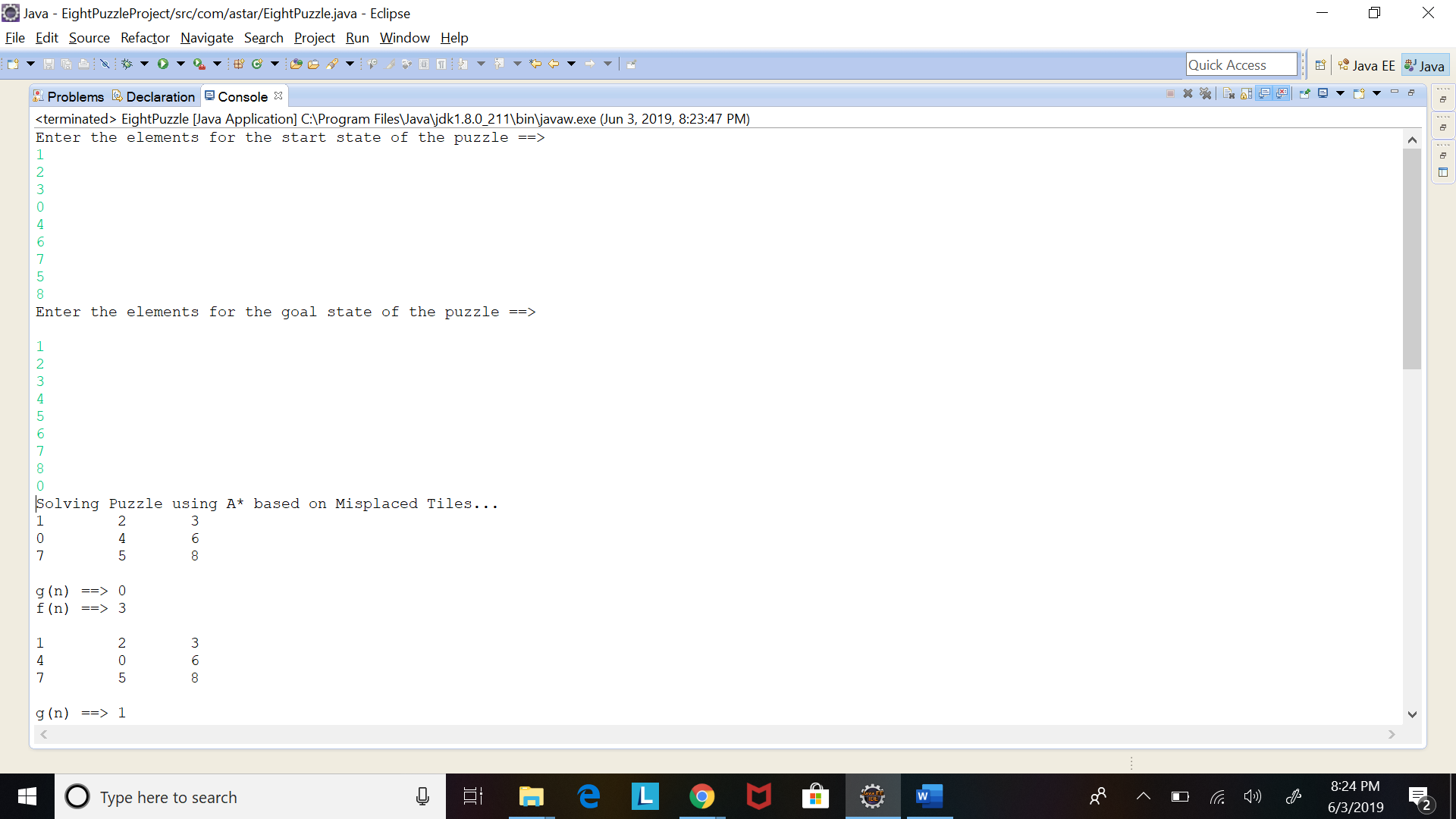
f(n) ==> 6

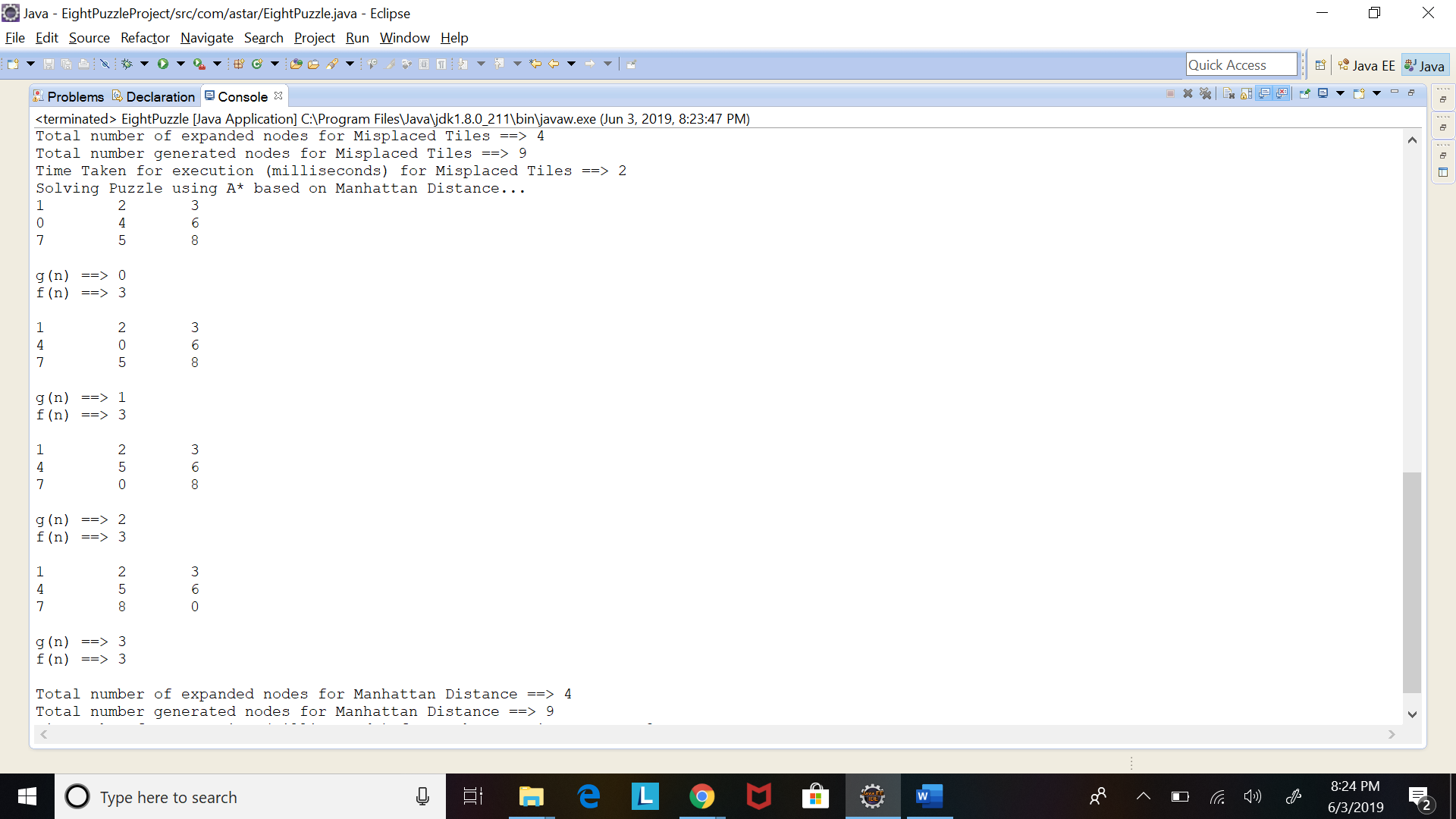
Total number of expanded nodes for Manhattan Distance ==> 7

Total number generated nodes for Manhattan Distance ==> 13

Time Taken for execution (milliseconds) for Manhattan Distance ==> 1

**Test Case 3:**





Enter the elements for the start state of the puzzle ==> 1 2 3 0 4 6 7 5 8

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

1 2 3 4 5 6 7 8 0

Solving Puzzle using A\* based on Misplaced Tiles...

1 2 3

0 4 6

7 5 8

g(n) ==> 0

f(n) ==> 3

1 2 3

4 0 6

7 5 8

g(n) ==> 1

f(n) ==> 3

1 2 3

4 5 6

7 0 8

g(n) ==> 2

f(n) ==> 3

1 2 3

4 5 6

7 8 0

g(n) ==> 3

f(n) ==> 3

Total number of expanded nodes for Misplaced Tiles ==> 4

Total number generated nodes for Misplaced Tiles ==> 9

Time Taken for execution (milliseconds) for Misplaced Tiles ==> 2

Solving Puzzle using A\* based on Manhattan Distance...

1 2 3

0 4 6

7 5 8

g(n) ==> 0

f(n) ==> 3

1 2 3

4 0 6

7 5 8

g(n) ==> 1

f(n) ==> 3

1 2 3

4 5 6

7 0 8

g(n) ==> 2

f(n) ==> 3

1 2 3

4 5 6

7 8 0

g(n) ==> 3

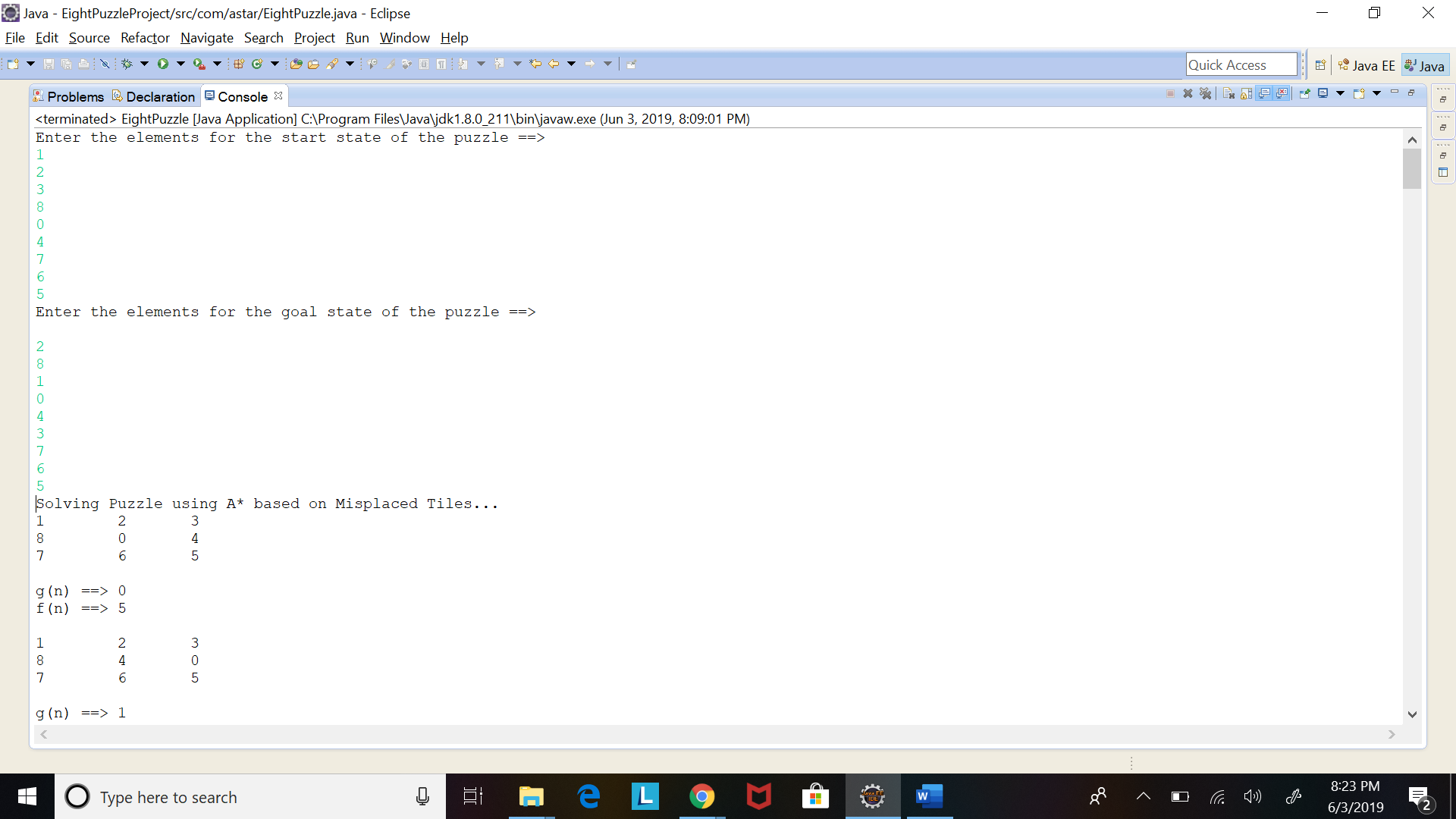
f(n) ==> 3

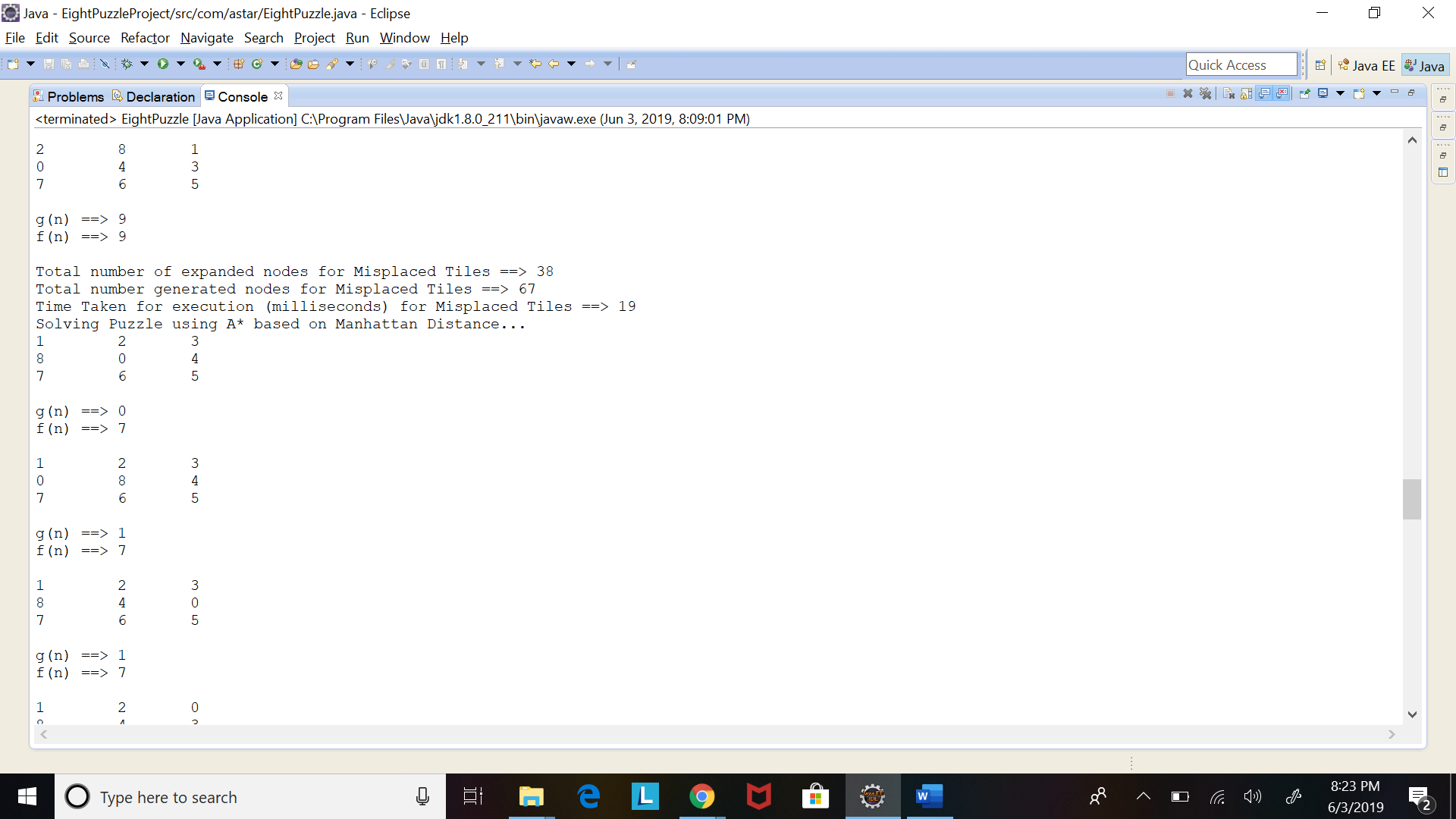
Total number of expanded nodes for Manhattan Distance ==> 4

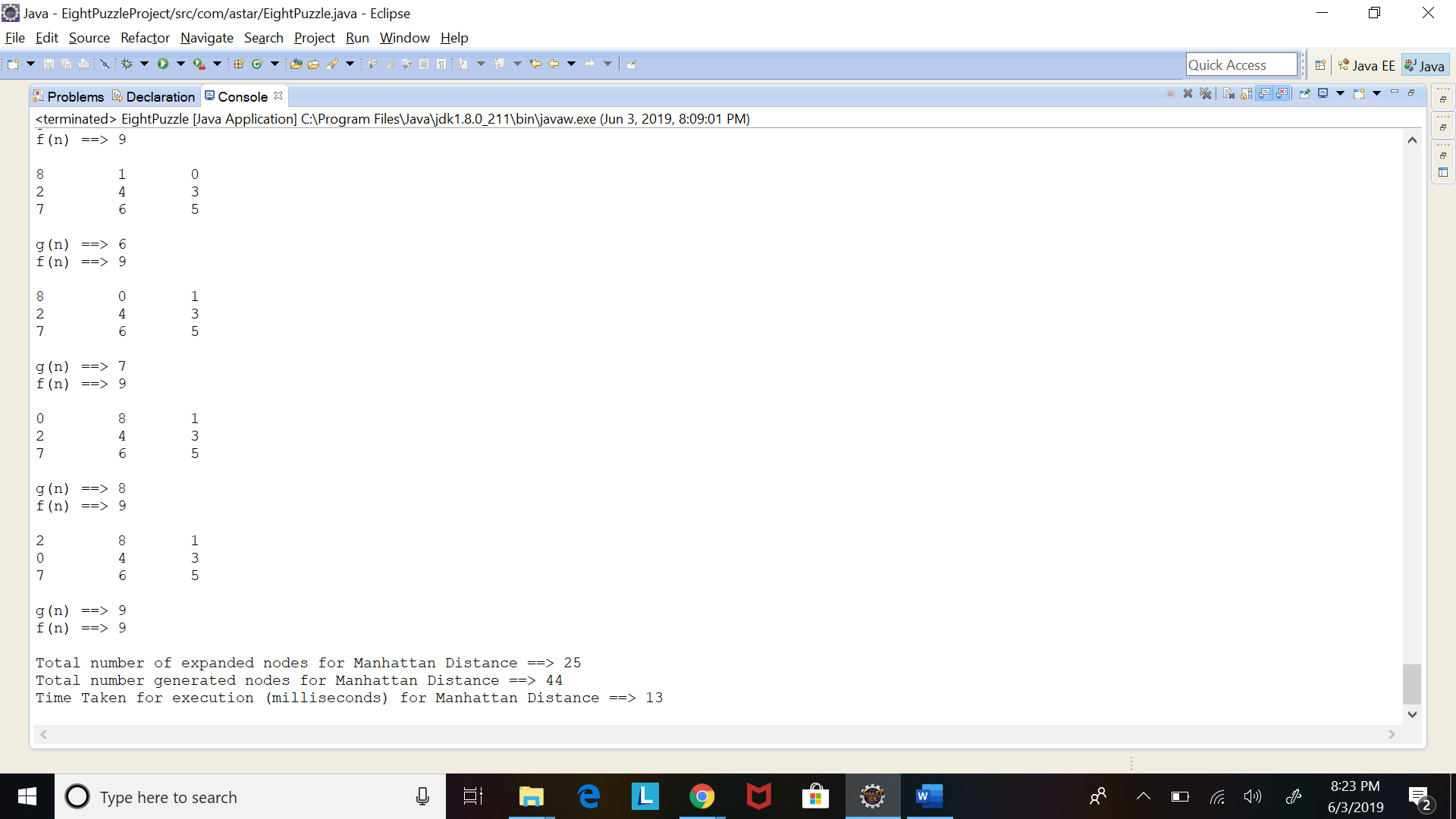
Total number generated nodes for Manhattan Distance ==> 9

Time Taken for execution (milliseconds) for Manhattan Distance ==> 3

**Test Case 4:**







Enter the elements for the start state of the puzzle ==> 1 2 3 8 0 4 7 6 5

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

2 8 1 0 4 3 7 6 5

Solving Puzzle using A\* based on Misplaced Tiles...

1 2 3

8 0 4

7 6 5

g(n) ==> 0

f(n) ==> 5

1 2 3

8 4 0

7 6 5

g(n) ==> 1

f(n) ==> 5

1 2 0

8 4 3

7 6 5

g(n) ==> 2

f(n) ==> 5

1 2 3

0 8 4

7 6 5

g(n) ==> 1

f(n) ==> 6

1 0 2

8 4 3

7 6 5

g(n) ==> 3

f(n) ==> 6

1 0 3

8 2 4

7 6 5

g(n) ==> 1

f(n) ==> 6

1 2 3

8 4 5

7 6 0

g(n) ==> 2

f(n) ==> 7

1 2 3

8 6 4

7 0 5

g(n) ==> 1

f(n) ==> 7

1 3 0

8 2 4

7 6 5

g(n) ==> 2

f(n) ==> 7

0 2 3

1 8 4

7 6 5

g(n) ==> 2

f(n) ==> 7

0 1 2

8 4 3

7 6 5

g(n) ==> 4

f(n) ==> 7

2 0 3

1 8 4

7 6 5

g(n) ==> 3

f(n) ==> 7

0 1 3

8 2 4

7 6 5

g(n) ==> 2

f(n) ==> 7

2 8 3

1 0 4

7 6 5

g(n) ==> 4

f(n) ==> 7

2 8 3

1 4 0

7 6 5

g(n) ==> 5

f(n) ==> 7

2 8 0

1 4 3

7 6 5

g(n) ==> 6

f(n) ==> 7

2 3 0

1 8 4

7 6 5

g(n) ==> 4

f(n) ==> 8

1 4 2

8 0 3

7 6 5

g(n) ==> 4

f(n) ==> 8

2 8 3

0 1 4

7 6 5

g(n) ==> 5

f(n) ==> 8

1 3 4

8 2 0

7 6 5

g(n) ==> 3

f(n) ==> 8

8 1 3

0 2 4

7 6 5

g(n) ==> 3

f(n) ==> 8

8 1 2

0 4 3

7 6 5

g(n) ==> 5

f(n) ==> 8

1 2 3

7 8 4

0 6 5

g(n) ==> 2

f(n) ==> 8

2 8 3

1 6 4

7 0 5

g(n) ==> 5

f(n) ==> 9

1 2 3

8 6 4

0 7 5

g(n) ==> 2

f(n) ==> 9

1 3 4

8 0 2

7 6 5

g(n) ==> 4

f(n) ==> 9

8 1 3

2 0 4

7 6 5

g(n) ==> 4

f(n) ==> 9

1 2 3

8 4 5

7 0 6

g(n) ==> 3

f(n) ==> 9

2 3 4

1 8 0

7 6 5

g(n) ==> 5

f(n) ==> 9

2 8 3

1 4 5

7 6 0

g(n) ==> 6

f(n) ==> 9

2 0 8

1 4 3

7 6 5

g(n) ==> 7

f(n) ==> 9

8 1 3

2 4 0

7 6 5

g(n) ==> 5

f(n) ==> 9

1 4 2

0 8 3

7 6 5

g(n) ==> 5

f(n) ==> 9

1 2 3

8 6 4

7 5 0

g(n) ==> 2

f(n) ==> 9

8 1 0

2 4 3

7 6 5

g(n) ==> 6

f(n) ==> 9

8 0 1

2 4 3

7 6 5

g(n) ==> 7

f(n) ==> 9

0 8 1

2 4 3

7 6 5

g(n) ==> 8

f(n) ==> 9

2 8 1

0 4 3

7 6 5

g(n) ==> 9

f(n) ==> 9

Total number of expanded nodes for Misplaced Tiles ==> 38

Total number generated nodes for Misplaced Tiles ==> 67

Time Taken for execution (milliseconds) for Misplaced Tiles ==> 9

Solving Puzzle using A\* based on Manhattan Distance...

1 2 3

8 0 4

7 6 5

g(n) ==> 0

f(n) ==> 7

1 2 3

0 8 4

7 6 5

g(n) ==> 1

f(n) ==> 7

1 2 3

8 4 0

7 6 5

g(n) ==> 1

f(n) ==> 7

1 2 0

8 4 3

7 6 5

g(n) ==> 2

f(n) ==> 7

1 2 3

8 4 5

7 6 0

g(n) ==> 2

f(n) ==> 9

1 0 2

8 4 3

7 6 5

g(n) ==> 3

f(n) ==> 9

1 2 3

7 8 4

0 6 5

g(n) ==> 2

f(n) ==> 9

0 2 3

1 8 4

7 6 5

g(n) ==> 2

f(n) ==> 9

1 2 3

8 6 4

7 0 5

g(n) ==> 1

f(n) ==> 9

2 0 3

1 8 4

7 6 5

g(n) ==> 3

f(n) ==> 9

1 0 3

8 2 4

7 6 5

g(n) ==> 1

f(n) ==> 9

2 8 3

1 0 4

7 6 5

g(n) ==> 4

f(n) ==> 9

0 1 3

8 2 4

7 6 5

g(n) ==> 2

f(n) ==> 9

2 8 3

0 1 4

7 6 5

g(n) ==> 5

f(n) ==> 9

2 8 3

1 4 0

7 6 5

g(n) ==> 5

f(n) ==> 9

0 1 2

8 4 3

7 6 5

g(n) ==> 4

f(n) ==> 9

8 1 3

0 2 4

7 6 5

g(n) ==> 3

f(n) ==> 9

2 8 0

1 4 3

7 6 5

g(n) ==> 6

f(n) ==> 9

8 1 2

0 4 3

7 6 5

g(n) ==> 5

f(n) ==> 9

8 1 3

2 0 4

7 6 5

g(n) ==> 4

f(n) ==> 9

8 1 3

2 4 0

7 6 5

g(n) ==> 5

f(n) ==> 9

8 1 0

2 4 3

7 6 5

g(n) ==> 6

f(n) ==> 9

8 0 1

2 4 3

7 6 5

g(n) ==> 7

f(n) ==> 9

0 8 1

2 4 3

7 6 5

g(n) ==> 8

f(n) ==> 9

2 8 1

0 4 3

7 6 5

g(n) ==> 9

f(n) ==> 9

Total number of expanded nodes for Manhattan Distance ==> 25

Total number generated nodes for Manhattan Distance ==> 44

Time Taken for execution (milliseconds) for Manhattan Distance ==> 4

Source Code:

**1.EightPuzzle.Java**

package com.Astar.Puzzle;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.PriorityQueue;

import java.util.Scanner;

/\*\*

\* This is the class that triggers that 8 Puzzle problem and solves it using the A\* algorithm.

\*/

public class EightPuzzle {

//Goal State will be constant hence it will be loaded only one at class level

public static int[][] goalState;

public static PriorityQueue<EightPuzzleBeanH1> priorityQueueH1 = new PriorityQueue<EightPuzzleBeanH1>();

public static PriorityQueue<EightPuzzleBeanH2> priorityQueueH2 = new PriorityQueue<EightPuzzleBeanH2>();

public static ArrayList<EightPuzzleBeanH1> expandedNodesH1 = new ArrayList<EightPuzzleBeanH1>();

public static ArrayList<EightPuzzleBeanH2> expandedNodesH2 = new ArrayList<EightPuzzleBeanH2>();

public static void main(String args[]) {

Scanner in = new Scanner(System.in);

int startState[][] = new int[3][3];

goalState = new int[3][3];

//Start state and goal state

System.out.println("Enter the elements for the start state of the puzzle ==>");

startState = EightPuzzleUtil.generate2dArrayFromUserInput(in);

System.out.println("Enter the elements for the goal state of the puzzle ==>");

goalState = EightPuzzleUtil.generate2dArrayFromUserInput(in);

calculateBasedOnMisplacedTiles(startState);

calculateBasedOnManhattanDistance(startState);

}

/\*\*

\* The method calculated the best possible moves in a 8 puzzle game based on the Misplaced Tiles

\*/

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

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

long startTime = System.currentTimeMillis();

//Level is 0 since we have only one node for now which is yet to be expanded

EightPuzzleBeanH1 state = new EightPuzzleBeanH1(startState, 0);

processPuzzlePlayH1(state);

for (EightPuzzleBeanH1 gameStatesBean : expandedNodesH1) {

EightPuzzleUtil.printMoveStats(gameStatesBean);

}

//This prevents the application from running for a very long duration for a difficult puzzle

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

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

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

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

} else {

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

System.out.println("Total number generated nodes for Misplaced Tiles ==> " + (expandedNodesH1.size() + priorityQueueH1.size()));

}

long endTime = System.currentTimeMillis();

// Calculate Time Taken for Total Execution

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

}

/\*\*

\* The method calculated the best possible moves in a 8 puzzle game based on the Manhattan Distance

\* @param startState The current State of the puzzle

\*/

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

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

long startTime = System.currentTimeMillis();

//Level is 0 since we have only one node for now which is yet to be expanded

EightPuzzleBeanH2 state = new EightPuzzleBeanH2(startState, 0);

processPuzzlePlayH2(state);

for (EightPuzzleBeanH2 gameStatesBean : expandedNodesH2) {

EightPuzzleUtil.printMoveStats(gameStatesBean);

}

//This prevents the application from running for a very long duration for a difficult puzzle

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

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

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

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

} else {

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

System.out.println("Total number generated nodes for Manhattan Distance ==> " + (expandedNodesH2.size() + priorityQueueH2.size()));

}

long endTime = System.currentTimeMillis();

// Calculate Time Taken for Total Execution

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

}

/\*\*

\* Processes the current state of the Puzzle. Takes the current move as the argument and adds it to the PriorityQueue

\*/

public static void processPuzzlePlayH1(EightPuzzleBeanH1 move) {

priorityQueueH1.add(move);

ArrayList<EightPuzzleBeanH1> childNodesList = new ArrayList<EightPuzzleBeanH1>();

do {

boolean isNodeVisited;

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

EightPuzzleBeanH1 currentPuzzlePlay = priorityQueueH1.poll();

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

expandedNodesH1.add(currentPuzzlePlay);

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

if (Arrays.deepEquals(currentPuzzlePlay.stateOfPuzzle, goalState)) {

break;

}

childNodesList = currentPuzzlePlay.generatePossibleStates(currentPuzzlePlay);

//Check if expanded node is already visited

for (EightPuzzleBeanH1 childNode : childNodesList) {

isNodeVisited = false;

for (EightPuzzleBeanH1 expandedNode : expandedNodesH1) {

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

isNodeVisited = true;

}

}

if (isNodeVisited) {

continue;

}

priorityQueueH1.add(childNode);

}

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

}

/\*\*

\* Processes the current state of the Puzzle. Takes the current move as the argument and adds it to the PriorityQueue

\*/

public static void processPuzzlePlayH2(EightPuzzleBeanH2 move) {

priorityQueueH2.add(move);

ArrayList<EightPuzzleBeanH2> childNodesList = new ArrayList<EightPuzzleBeanH2>();

do {

boolean isNodeVisited;

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

EightPuzzleBeanH2 currentPuzzlePlay = priorityQueueH2.poll();

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

expandedNodesH2.add(currentPuzzlePlay);

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

if (Arrays.deepEquals(currentPuzzlePlay.stateOfPuzzle, goalState)) {

break;

}

childNodesList = currentPuzzlePlay.generatePossibleStates(currentPuzzlePlay);

//Check if expanded node is already visited

for (EightPuzzleBeanH2 childNode : childNodesList) {

isNodeVisited = false;

for (EightPuzzleBeanH2 expandedNode : expandedNodesH2) {

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

isNodeVisited = true;

}

}

if (isNodeVisited) {

continue;

}

priorityQueueH2.add(childNode);

}

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

}

}

**2.EightPuzzleBean.java**

package com.Astar.Puzzle;

/\*\*

\* Common parent class containing variables declarations that can be used by both Heuristics

\*/

public class EightPuzzleBean {

public int aStarDistance;

public int[][] stateOfPuzzle;

public int level;

}

**3.EightPuzzleBeanH1.java**

package com.Astar.Puzzle;

import java.util.ArrayList;

/\*\*

\* The class performance the movements of the 8 puzzle game while at the same time generating the child nodes for each move.

\* The best move for each level is calculated on the basis of the Misplaced Tiles concept where the number of misplaced tiles

\* of the current puzzle state and the goal state are compared.

\*/

public class EightPuzzleBeanH1 extends EightPuzzleBean implements Comparable<EightPuzzleBeanH1> {

public EightPuzzleBeanH1(int[][] array, int level) {

int lengthOfArray = array.length;

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

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

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

this.stateOfPuzzle[i][j] = array[i][j];

}

}

this.level = level;

this.aStarDistance = calculateMisplacedTilesDistance() + level;

}

/\*\*

\* Below function calculates the Missing Tiles distance(heuristic value) for each

\* state or node. I.e the sum of the distances of the tiles from their goal

\* positions

\*/

private int calculateMisplacedTilesDistance() {

int count = 0;

int lengthOfArray = EightPuzzle.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] != EightPuzzle.goalState[i][j]) {

count++;

}

}

}

return count;

}

/\*\*

\* Generates child nodes by searching for '0' which is the free space and moving it in all possible directions.

\*/

public ArrayList<EightPuzzleBeanH1> generatePossibleStates(EightPuzzleBeanH1 parentNode) {

ArrayList<EightPuzzleBeanH1> childNodes = new ArrayList<EightPuzzleBeanH1>();

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++) {

for (int column1 = 0; column1 < 3; column1++) {

a[row1][column1] = parentNode.stateOfPuzzle[row1][column1];

}

}

a = EightPuzzleUtil.movePiece(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++) {

for (int column2 = 0; column2 < 3; column2++) {

a[row2][column2] = parentNode.stateOfPuzzle[row2][column2];

}

}

a = EightPuzzleUtil.movePiece(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++) {

for (int column3 = 0; column3 < 3; column3++) {

a[row3][column3] = parentNode.stateOfPuzzle[row3][column3];

}

}

a = EightPuzzleUtil.movePiece(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++) {

for (int column4 = 0; column4 < 3; column4++) {

a[row4][column4] = parentNode.stateOfPuzzle[row4][column4];

}

}

a = EightPuzzleUtil.movePiece(a, row, column, row + 1, column);

addChildNodesToList(parentNode, childNodes, a);

}

}

}

}

return childNodes;

}

/\*\*

\* For every move of the tiles, g(n) i.e. the current level of the puzzle is incremented by one for every expansion

\*/

public void addChildNodesToList(EightPuzzleBeanH1 parentNode, ArrayList<EightPuzzleBeanH1> childNodes, int[][] a) {

EightPuzzleBeanH1 childNode = new EightPuzzleBeanH1(a, parentNode.level + 1);

childNodes.add(childNode);

}

/\*\*

\* The comparator determines the order in which the elements are accessed in the PriorotyQueue

\* which in this case is based on the Misplaced Tiles

\*/

@Override

public int compareTo(EightPuzzleBeanH1 EightPuzzleBean) {

return this.aStarDistance > EightPuzzleBean.aStarDistance ? 1 : this.aStarDistance < EightPuzzleBean.aStarDistance ? -1 : 0;

}

}

**4.EightPuzzleBeanH2.java**

package com.Astar.Puzzle;

import java.util.ArrayList;

/\*\*

\* The class performance the movements of the 8 puzzle game while at the same time generating the child nodes for each move.

\* The best move for each level is calculated on the basis of the Manhattan Distance concept

\*/

public class EightPuzzleBeanH2 extends EightPuzzleBean implements Comparable<EightPuzzleBeanH2> {

public EightPuzzleBeanH2(int[][] array, int level) {

int lengthOfArray = array.length;

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

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

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

this.stateOfPuzzle[i][j] = array[i][j];

}

}

this.level = level;

this.aStarDistance = calculateManhattanDistance() + level;

}

/\*\*

\* 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() {

int manhattanDistance = 0;

int[] index = new int[2];

int lengthOfArray = EightPuzzle.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(EightPuzzle.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;

}

/\*\*

\* Generates child nodes by searching for '0' which is the free space and moving it in all possible directions.

\*/

public ArrayList<EightPuzzleBeanH2> generatePossibleStates(EightPuzzleBeanH2 parentNode) {

ArrayList<EightPuzzleBeanH2> childNodes = new ArrayList<EightPuzzleBeanH2>();

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++) {

for (int column1 = 0; column1 < 3; column1++) {

a[row1][column1] = parentNode.stateOfPuzzle[row1][column1];

}

}

a = EightPuzzleUtil.movePiece(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++) {

for (int column2 = 0; column2 < 3; column2++) {

a[row2][column2] = parentNode.stateOfPuzzle[row2][column2];

}

}

a = EightPuzzleUtil.movePiece(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++) {

for (int column3 = 0; column3 < 3; column3++) {

a[row3][column3] = parentNode.stateOfPuzzle[row3][column3];

}

}

a = EightPuzzleUtil.movePiece(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++) {

for (int column4 = 0; column4 < 3; column4++) {

a[row4][column4] = parentNode.stateOfPuzzle[row4][column4];

}

}

a = EightPuzzleUtil.movePiece(a, row, column, row + 1, column);

addChildNodesToList(parentNode, childNodes, a);

}

}

}

}

return childNodes;

}

/\*\*

\* For every move of the tiles, g(n) i.e. the current level of the puzzle is incremented by one for every expansion

\*/

public void addChildNodesToList(EightPuzzleBeanH2 parentNode, ArrayList<EightPuzzleBeanH2> childNodes, int[][] a) {

EightPuzzleBeanH2 childNode = new EightPuzzleBeanH2(a, parentNode.level + 1);

childNodes.add(childNode);

}

/\*\*

\* The comparator determines the order in which the elements are accessed in the PriorotyQueue

\* which in this case is based on the Manhattan Distance

\*/

@Override

public int compareTo(EightPuzzleBeanH2 EightPuzzleBean) {

return this.aStarDistance > EightPuzzleBean.aStarDistance ? 1 : this.aStarDistance < EightPuzzleBean.aStarDistance ? -1 : 0;

}

}

**5.EightPuzzleUtil**

package com.Astar.Puzzle;

import java.util.Arrays;

import java.util.Scanner;

/\*\*

\* This class contains common utilities that are used by the app

\*/

public class EightPuzzleUtil {

/\*\*

\* This constant determines how long each heuristic technique will run based

\* on the size of the Priority Queue

\*/

public static final int THRESHOLD = 20000;

/\*\*

\* Method prints the following variables per expanded node

\*/

public static <T extends EightPuzzleBean> void printMoveStats(

T EightPuzzleBean) {

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

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

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

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

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

}

//Common method that takes input from user. Contains sanity check for input.

public static int[][] generate2dArrayFromUserInput(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. Try Again...");

System.exit(0);

}

}

}

return puzzleInput;

}

// Moves the 0 element by swapping

public static int[][] movePiece(int[][] currentPuzzlePlay, int row1,

int column1, int row2, int column2) {

int tmp = currentPuzzlePlay[row1][column1];

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

currentPuzzlePlay[row2][column2] = tmp;

return currentPuzzlePlay;

}

}