Project: 1 CS 205. Introduction to Artificial Intelligence

Dr. Eamonn Koegh

Prince Choudhary
SID: 862254827

The Eight Puzzle

In completing this assignment I consulted:

- The lecture videos and slides.
- The Project 1 pdf given to us.
- Comparator implementation https://stackoverflow.com/questions/4805606/how-to-sort-by-two-fields-in-java
- Wikipedia page for the 15-puzzle: https://en.wikipedia.org/wiki/15_puzzle
- Wikipedia page for sliding puzzle: https://en.wikipedia.org/wiki/Sliding_puzzle

All important code is original. Unimportant subroutines that are not completely original are:

- Used java.util.PriorityQueue to implement the queue functionality.
- Used java.util.ArrayList to store visited nodes.
- Used java.util.Comparator to sort nodes as per heuristic.

Outline of this report:

Introduction

8-puzzle is a combination puzzle that challenges a player to slide pieces along certain routes to establish a certain end-configuration (goal-state). The 8-puzzle is a smaller version of 15-puzzle. It consists of 3x3 area with tiles numbered 1 to 8 and one blank space where tiles can be slid to solve the puzzle or attain the goal state. Similarly in 15-puzzle there is 4x4 tile area with 15 numbered tiles and one blank. Below is a figure of 15-puzzle.

This assignment is the first project in Dr. Eamonn Keogh's Artificial Intelligence course (CS205) at the University of California, Riverside during the quarter of Spring 2021. The following write up is to detail my findings through the course of project completion. It explores uniform cost search, A∗ search with the misplaced tile heuristic, and A∗ search with the Manhattan distance heuristic. My language of choice is Java (version 11), and the full code for the project is included.

Problem Formulation:

- Goal: Goal State is generated by code depending on dimension of the puzzle.
- States: It is the location of different tiles in the puzzle.
- Actions: Move the blank tile in left, up, down and right positions .
- Performance: Number of nodes explored, time taken and maximum queue size.

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

Comparison of Algorithms:

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

- Uniform cost search : Uniform cost search is a form of blind search, relying only on the expansion cost to choose which nodes to traverse. It does not have any heuristic guiding the search, hence h(n) = 0, we define the evaluation of as f(n) = g(n). It is essentially breadth first search algorithm.
- A∗ with Misplaced Tiles: Here h(n) is calculated by comparing the number of misplaced tiles between the current state and goal state.
- A∗ with Manhattan Distance: Here h(n) is calculated by the distance between the tiles in current and goal state. It is the sum of absolute values of differences between goal state (x, y) coordinates and current state (i, j) coordinates respectively, i.e. |i - x|+ |j - y|

Implementation details:

I have implemented the project in Java 11. I have created a custom data structure “Node” to store the states of the puzzle. Inbuilt priority queue library is used to for queuing the nodes. Custom comparator is created so as to sort the nodes as per heuristic values. I have implemented various methods to serve different functionalities and retain modularity.

- “main” takes the input choices.
- “generalSearch” this is where the most suitable puzzle state is chosen and compared to attain goal state.
- "printPuzzleState" prints the current state of the puzzle.
- "addToQueue" creates a node and add the puzzle state to the queue.
- "comparePuzzle" compares the two puzzles states and return true if they are same otherwise false.
- "exploredCheck" checks if the state of the puzzle is already explored.
- "getChildren" returns a list of children after moving left, right, up or down.
- "generateGoalState" generates the goal state of n-puzzle.

Results:

Using several initial puzzle states of differing solution depths, we can compare and contrast the time taken, number of nodes expanded (represents time complexity) and maximum number of nodes in the queue (represents space complexity) at a given time for each algorithm and come to a conclusion on the best algorithm for the puzzle configuration. Below are the results in tabular form which I got on running sample inputs.

Uniform cost search:

	Nodes expapanded	Maximum queue size	Time taken(ms)
Trivial	0	1	15
Very Easy	2	5	15
Easy	5	6	16
Doable	27	20	20
OhBoy	Keeps running	Keeps running	Keeps running

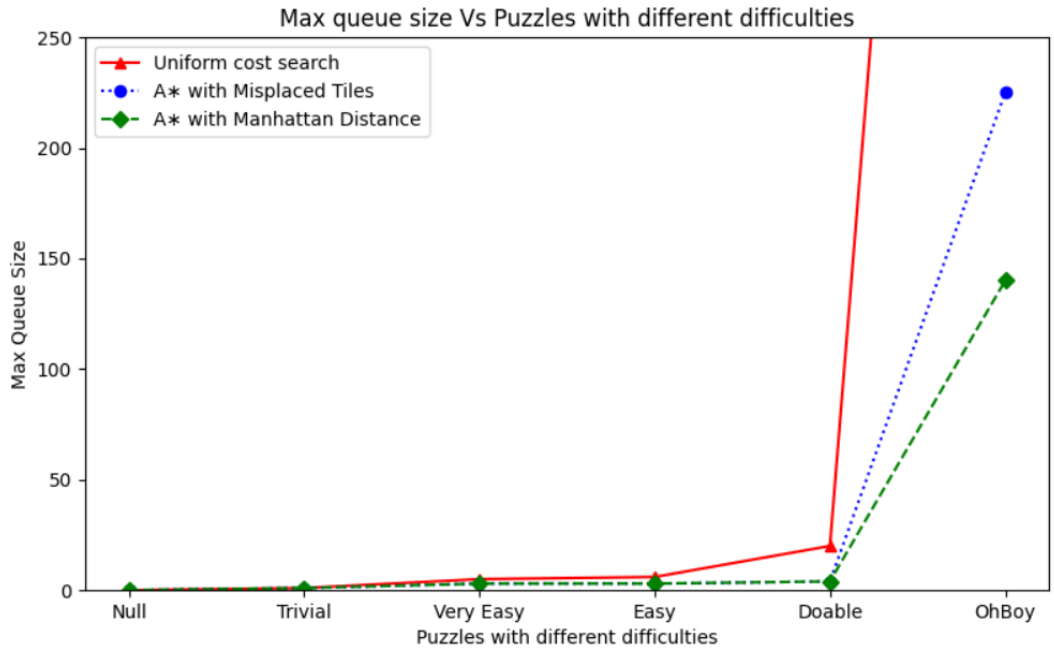
A∗ with Misplaced Tiles:

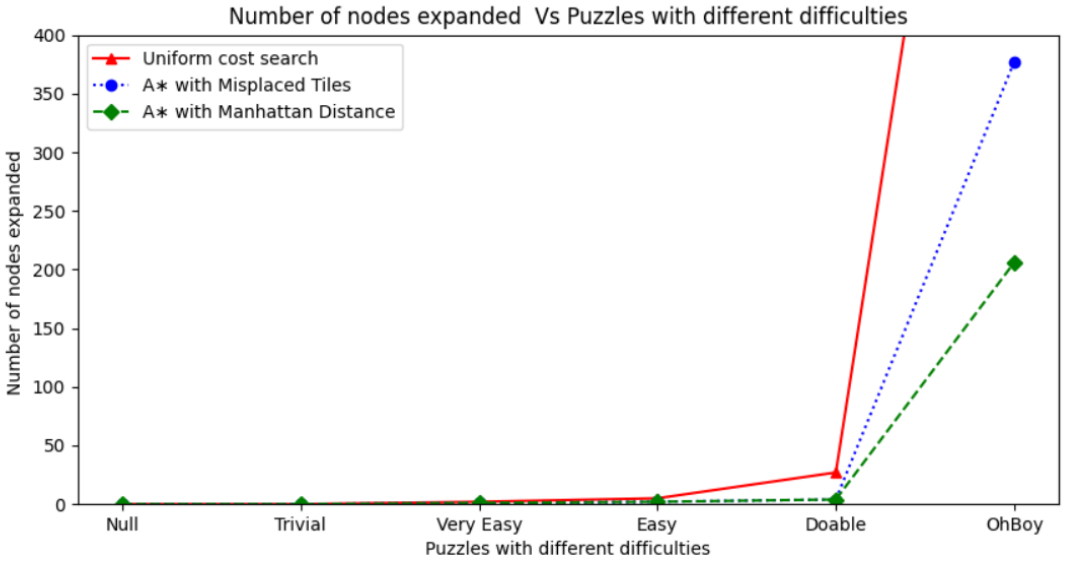
	Nodes expapanded	Maximum queue size	Time taken(ms)
Trivial	0	1	14
Very Easy	1	3	14
Easy	2	3	14
Doable	4	4	15
OhBoy	377	225	44

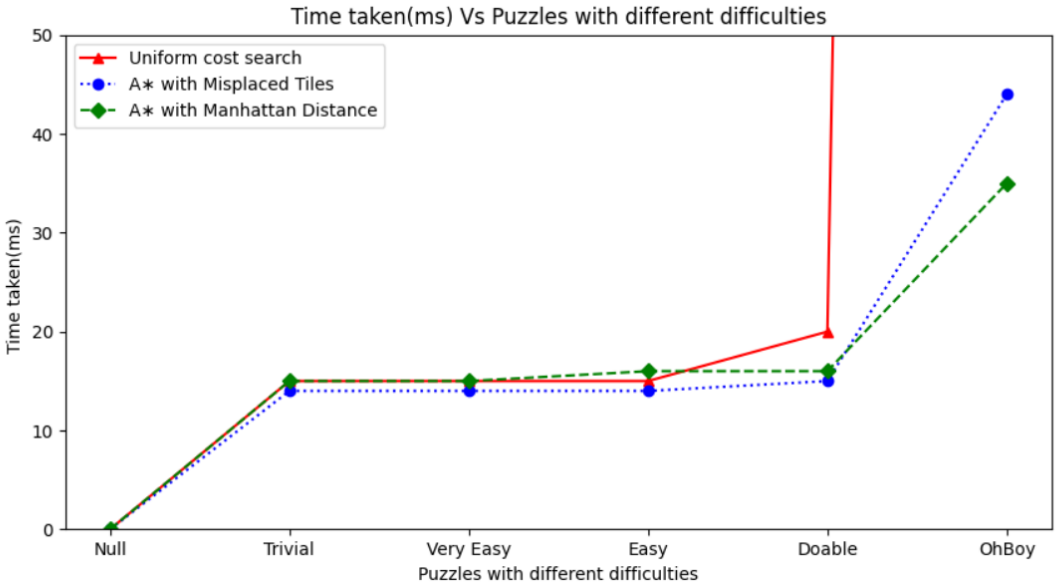
A∗ with Manhattan Distance:

	Nodes expapanded	Maximum queue size	Time taken(ms)
Trivial	0	1	15
Very Easy	1	3	15
Easy	2	3	16
Doable	4	4	16
OhBoy	206	140	35

From these results, we can see that the heuristic searches perform orders of magnitude better than the blind uniform cost search. Furthermore, the Manhattan Distance heuristic significantly outperforms the Misplaced Tiles heuristic as expected. Below are some graphical representations of our results.







Conclusion:

This project compared the usefulness of three search algorithms in solving the 8-puzzle. The results clearly showed that admissible heuristics drive faster convergence to the optimal solution, while also requiring less space than Uniform cost search. Between the two heuristics used for the A∗ algorithm, the Manhattan Distance significantly outperform the Misplaced Tile heuristic.

Sample Trace on 8 Puzzle:

***** Welcome to 8-Puzzle Game Solver *****

Enter '1' to use a default puzzle, or '2' to create your own.

1

Enter your choice for algorithm:

1) Uniform Cost Search

2) A* with the Misplaced Tile heuristic

3) A* with the Manhattan Distance heuristic.

2

Please difficulty level:

1) Trivial

2) Very Easy

3) Easy

4) Doable

5) Oh Boy

3

Input state:

1 2 0

4 5 3

7 8 6

Fetching the node from queue g(n)=0 and h(n)=3

1 2 0

4 5 3

7 8 6

Fetching the node from queue g(n)=1 and h(n)=2

1 2 3

4 5 0

7 8 6

Fetching the node from queue g(n)=2 and h(n)=0

1 2 3

4 5 6

7 8 0

The Puzzle is Solved!

Number of nodes expanded: 2

Max queue size: 3

Time taken: 20ms

Sample trace on 15-puzzle:

***** Welcome to 8-Puzzle Game Solver *****

Enter '1' to use a default puzzle, or '2' to create your own.

2

Enter your choice for algorithm:

1) Uniform Cost Search

2) A* with the Misplaced Tile heuristic

3) A* with the Manhattan Distance heuristic.

3

Enter the dimension of you puzzle eg: 3 for 8-puzzle, 4 for 15-puzzle etc.

4

Enter your custom puzzle with white space between the numbers and hit enter for next row

1 2 3 4

5 6 7 8

9 10 11 12

13 14 0 15

Input state:

1 2 3 4

5 6 7 8

9 10 11 12

13 14 0 15

Fetching the node from queue g(n)=0 and h(n)=2

1 2 3 4

5 6 7 8

9 10 11 12

13 14 0 15

Fetching the node from queue g(n)=1 and h(n)=0

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 0

The Puzzle is Solved!

Number of nodes expanded: 1

Max queue size: 3

Time taken: 5ms

My Implementation:

Also available here: https://github.com/princerokn/CS205

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;
import static java.lang.System.exit;

public class EightPuzzle {
    static int puzzleDimension = 3; // This is to determine if its a 8-
puzzle or 15-puzzle etc.
    static PriorityQueue<Node> queue = new PriorityQueue<>(new
PQComparator()); //Priority queue will sort on the minimum heuristic value
    static ArrayList<Node> explored = new ArrayList<>(); // This will store
the explored list
    static int returnValue = -1;
    //Sample harcoded input
    static int[][] trivial = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};
    static int[][] veryEasy = {{1, 2, 3}, {4, 5, 6}, {7, 0, 8}};
    static int[][] easy = {{1, 2, 0}, {4, 5, 3}, {7, 8, 6}};
    static int[][] doable = {{0, 1, 2}, {4, 5, 3}, {7, 8, 6}};
```

```
    static int[][] ohBoy = {{8, 7, 1}, {6, 0, 2}, {5, 4, 3}};
    static int[][] goalState;
    private static int nodesExpanded = 0;
    private static int maxQueueSize = 1;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int choice, difficultyLevel = 0, algorithmChoice;
        long startTime = 0, endTime = 0;
        int[][] customPuzzle;

        System.out.println("***** Welcome to 8-Puzzle Game Solver *****");
        System.out.println("Enter '1' to use a default puzzle, or '2' to
create your own.");
        choice = sc.nextInt(); //select default or custom
        System.out.println("Enter your choice for algorithm:\n" +
                "1) Uniform Cost Search\n" +
                "2) A* with the Misplaced Tile heuristic\n" +
                "3) A* with the Manhattan Distance heuristic.");
        algorithmChoice = sc.nextInt();
        if (algorithmChoice < 1 || algorithmChoice > 3) {
            System.out.println("Enter correct choice 1,2 or 3");
            exit(-1);
        }
        if (choice == 1) {  //default puzzle
            System.out.println("Please difficulty level:\n1) Trivial\n2)
Very Easy\n3) Easy\n4) Doable\n5) Oh Boy");
            difficultyLevel = sc.nextInt();
            if (difficultyLevel > 5 || difficultyLevel < 1) {
                System.out.println("Enter correct choice:\n1) Trivial\n2)
Very Easy\n3) Easy\n4) Doable\n5) Oh Boy");
                exit(-1);
            }
            generateGoalState();
            startTime = System.currentTimeMillis();
            switch (difficultyLevel) {
                case 1:
                    System.out.println("Input state:");
                    printPuzzleState(trivial);
                    returnValue = generalSearch(trivial, algorithmChoice);
                    break;
                case 2:
                    System.out.println("Input state:");
                    printPuzzleState(veryEasy);
                    returnValue = generalSearch(veryEasy, algorithmChoice);
                    break;
                case 3:
                    System.out.println("Input state:");
                    printPuzzleState(easy);
                    returnValue = generalSearch(easy, algorithmChoice);
                    break;
                case 4:
                    System.out.println("Input state:");
                    printPuzzleState(doable);
                    returnValue = generalSearch(doable, algorithmChoice);
                    break;
                case 5:
                    System.out.println("Input state:");
                    printPuzzleState(ohBoy);
                    returnValue = generalSearch(ohBoy, algorithmChoice);
                    break;
```

```
            }
            endTime = System.currentTimeMillis();
        } else if (choice == 2) { //custom puzzle
            System.out.println("Enter the dimension of you puzzle eg: 3 for
8-puzzle,  4 for 15-puzzle etc.");
            puzzleDimension = sc.nextInt(); //Enter the dimension of your
custom puzzle
            customPuzzle = new int[puzzleDimension][puzzleDimension];
            System.out.println("Enter your custom puzzle with white space
between the numbers and hit enter for next row");
            sc.nextLine();
            for (int i = 0; i < puzzleDimension; i++) {
                String[] row;
                String line = sc.nextLine();
                if (line == null) {
                    System.err.println("Invalid Entry");
                    exit(1);
                }
                row = line.split(" ");
                for (int j = 0; j < puzzleDimension; j++)
                    customPuzzle[i][j] = Integer.parseInt(row[j]);
            }
            System.out.println("Input state:");
            printPuzzleState(customPuzzle);
            generateGoalState();
            startTime = System.currentTimeMillis();
            returnValue = generalSearch(customPuzzle, algorithmChoice);
            endTime = System.currentTimeMillis();
        } else {
            System.out.println("Enter correct choice 1 or 2");
            exit(-1);
        }

        if (returnValue == 0) {
            System.out.println("The Puzzle is Solved!");
            System.out.println("Number of nodes expanded: " +
nodesExpanded);
            System.out.println("Max queue size: " + maxQueueSize);
            System.out.println("Time taken: " + (endTime - startTime) +
"ms");
        } else {
            System.out.println("Error: Given input has no solution!");
        }
    }

    //This is where the most suitable puzzle state is chosen and compared
to attain goalState
    static int generalSearch(int[][] currPuzzle, int algorithm) {
        addToQueue(currPuzzle, 0, algorithm); //initial state
        while (true) {
            if (queue.isEmpty())
                return -1;
            ArrayList<int[][]> children;
            Node tempState = queue.poll();
            int[][] tempNode = tempState.getPuzzle();
            int gN = tempState.getgN();
            int hN = tempState.gethN();
            System.out.println("Fetching the node from queue g(n)=" + gN +
" and h(n)=" + hN);
            printPuzzleState(tempState.getPuzzle());
            if (comparePuzzle(tempState.getPuzzle(), goalState)) { //Goal
```

```
                return 0;
            } else {  // Goal state not found, add eligible child nodes
                children = getChildren(tempState);
                if (children.size() == 0) {
                    continue;  //No eligible child
                }
                nodesExpanded++;
                for (int[][] child : children) {
                    if (!exploredCheck(child)) {  // check if already
explored
                        addToQueue(child, gN + 1, algorithm);
                        if (queue.size() > maxQueueSize)
                            maxQueueSize = queue.size();
                    }
                }
            }
        }
    }

    //This will print the current state of the puzzle
    static void printPuzzleState(int[][] puzzle) {
        for (int i = 0; i < puzzleDimension; i++) {
            for (int j = 0; j < puzzleDimension; j++) {
                System.out.print(puzzle[i][j] + " ");
            }
            System.out.println();
        }
    }
    //This will create a node add the puzzle state to the queue
    static void addToQueue(int[][] puzzle, int gN, int algorithm) {
        int hN = 0;
        switch (algorithm) {
            case 1: //Uniform Cost
                hN = 0;
                break;
            case 2: //A* misplaced tile
                for (int i = 0; i < puzzleDimension; i++)
                    for (int j = 0; j < puzzleDimension; j++) {
                        if (puzzle[i][j] != goalState[i][j])
                            hN++;
                    }
                break;
            case 3: //A* manhattan distance
                for(int i = 0; i < puzzleDimension; i++)
                    for(int j = 0; j < puzzleDimension; j++) {
                        int target = puzzle[i][j];
                        for(int k = 0; k < puzzleDimension; k++)
                            for (int l = 0; l < puzzleDimension; l++) {
                                if (goalState[k][l] == target) {
                                    hN += Math.abs(k - i) + Math.abs(l -
j);
                                }
                            }
                    }
                break;
            default:
                System.err.println("Wrong algorithm choice");
                break;
        }
        Node currNode = new Node(puzzle, gN, hN);
        queue.add(currNode);
```

```
        explored.add(currNode);
    }

    //This will compare the two puzzles states and return true if they are
same otherwise false
    static boolean comparePuzzle(int[][] puzzle1, int[][] puzzle2) {
        for (int i = 0; i < puzzleDimension; i++)
            for (int j = 0; j < puzzleDimension; j++)
                if (puzzle1[i][j] != puzzle2[i][j])
                    return false;
        return true;
    }

    //checks if the state of the puzzle is already explored
    static boolean exploredCheck(int[][] child) {
        boolean identical = false;
        for (Node n : explored) {
            int[][] temp = n.getPuzzle();
            identical = comparePuzzle(child, temp);
            if (identical)
                return identical;
        }
        return identical;
    }


    //returns a list children after moving left, right, up or down
    static ArrayList<int[][]> getChildren(Node currState) {
        ArrayList<int[][]> list = new ArrayList<>(); //this will store list
of eligble child
        int[][] currPuzzle = currState.getPuzzle();
        int blankX = -1,blankY = -1; //coordinates of blank or zero
        for (int i = 0; i < puzzleDimension; i++) {
            for (int j = 0; j < puzzleDimension; j++) {
                if (currPuzzle[i][j] == 0) {
                    blankX = i;
                    blankY = j;
                }
            }
        }
        if (blankX - 1 >= 0) {  //move up
            int[][] tempState = new int[puzzleDimension][];
            for (int i = 0; i < puzzleDimension; i++)
                tempState[i] = currPuzzle[i].clone();
            int temp = tempState[blankX][blankY];
            tempState[blankX][blankY] = tempState[blankX - 1][blankY];
            tempState[blankX - 1][blankY] = temp;
            list.add(tempState);
        }
        if (blankX + 1 < puzzleDimension) {  //move down
            int[][] tempState = new int[puzzleDimension][];
            for (int i = 0; i < puzzleDimension; i++)
                tempState[i] = currPuzzle[i].clone();

            int temp = tempState[blankX][blankY];
            tempState[blankX][blankY] = tempState[blankX + 1][blankY];
            tempState[blankX + 1][blankY] = temp;
            list.add(tempState);
        }
        if (blankY - 1 >= 0) {  //move left
            int[][] tempState = new int[puzzleDimension][];
```

```
            for (int i = 0; i < puzzleDimension; i++)
                tempState[i] = currPuzzle[i].clone();

            int temp = tempState[blankX][blankY];
            tempState[blankX][blankY] = tempState[blankX][blankY - 1];
            tempState[blankX][blankY - 1] = temp;
            list.add(tempState);
        }
        if (blankY + 1 < puzzleDimension) {  //move right
            int[][] tempState = new int[puzzleDimension][];
            for (int i = 0; i < puzzleDimension; i++)
                tempState[i] = currPuzzle[i].clone();
            int temp = tempState[blankX][blankY];
            tempState[blankX][blankY] = tempState[blankX][blankY + 1];
            tempState[blankX][blankY + 1] = temp;
            list.add(tempState);
        }
        return list;
    }
    //This will generate the goal state of n-puzzle
    static void generateGoalState() {
        goalState = new int[puzzleDimension][puzzleDimension];
        int counter = 1;
        for (int i = 0; i < puzzleDimension; i++) {
            for (int j = 0; j < puzzleDimension; j++) {
                goalState[i][j] = counter++;
            }
        }
        goalState[puzzleDimension - 1][puzzleDimension - 1] = 0;
    }
}

//This data structure represent the puzzle and its state
class Node {
    private final int[][] puzzle; //puzzle board
    private final int gN; // cost so far to reach goal state
    private final int hN; // heuristic cost or estimated cost to reach goal
state

    //constructor and getters
    public Node(int[][] puzzle, int gN, int hN) {
        this.puzzle = puzzle;
        this.gN = gN;
        this.hN = hN;
    }

    public int[][] getPuzzle() {
        return puzzle;
    }

    public int getgN() {
        return gN;
    }

    public int gethN() {
        return hN;
    }
}
//Comparator class to sort the nodes
class PQComparator implements Comparator<Node> {
    public int compare(Node n1, Node n2) {
```

```
        int Hn1 = n1.gethN();
        int Hn2 = n2.gethN();
        if ((Hn1 - Hn2) == 0)
            return n1.getgN() - n2.getgN();
        return Hn1 - Hn2;
    }
}
```