

# Programming Project 1:

## Solving the 8-puzzle using A\* algorithm

Pedram Bashiri

800874993

The program applies A\* algorithm to solve any N-puzzle problem. Dimensions of the boarding game are specified in [Main.java Ln: 12](#) and can be easily changed to any other N-puzzle problem. The program accept any initial and goal state and determines if it is solvable. If yes it prints out the states the algorithm takes to go from initial state to goal state, alongside number of total nodes generated in search tree and number of expanded nodes.

Number of expanded nodes is actually the same as number of states that should be visited during the algorithm and is saved in a variable called [numberOfMoves](#) in the code.

Number of total generated nodes is equal to size of the PriorityQueue that stores all of the [SreachNodes](#) created. After finding the goal state since every [SearchNode](#) keeps track of its previous node, we can chase the nodes all the way back to initial state. That's how the program outputs the path.

### [What is 8 puzzle?](#)

Given a 3x3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles in an arbitrary order given as goal state. We can slide four adjacent (left, right, above and below) tiles into the empty space.

**Goal Formulation:** Goal is described as an order of numbers that should be displayed on the board.

**Search Algorithm:** A\*

**Problem Formulation:**

*Initial State:* From Input

*Goal State:* From Input

*Possible Actions:* Left, Right, Above, Below

*State Space:* 3\*3 board

*Path:* Different states of the board during the execution time

*Path cost:* Number of moves prior to this state used as  $g(n)$  in algorithm to set the priority of `searchNodes` in the PriorityQueue.

Heuristic Function:

$h1(n)$  = number of misplaced tiles

$h2(n)$  = total Manhattan distance

Since  $h1$  dominates  $h2$ , it is better for search and has a smaller branching factor. This function is used in this implementation of A\*.

The program starts from `Main.java` and after creating the initial State by calling its class constructor, it calls the `solve(state)` method of `AStar` class which returns the number of nodes generated and stores the path from initial to goal state.

In `AStar` class the program uses `java.util.Stack` class to store all the states (expanded nodes) and all the `SearchNodes` (generated nodes) are stored in a priorityQueue called queue.

Here I used the fact that “exactly one of a board and its twin are solvable” to determine whether a puzzle is solvable or not, where “A twin is obtained by swapping two adjacent blocks (the blank does not count) in the same row”.

<http://coursera.cs.princeton.edu/>

State class represents the board in each step of the algorithm. It keeps the position of all 8 numbers in a 3\*3 array called Tiles. Following methods are implemented in this class:

- `manhatan()` *sum of Manhattan distances between tiles and goal State*
- `twin()` *Returns a new Board result of swapping to adjacent tiles*
- `neighbors()` *returns the neighbors of each state/node*

Execution Instruction:

Run the program from command line. First, compile `AStar.java` `State.java` and `Main.java` using command “`javac Astar.java`”. Next, execute `Main` and pass the initial and goal states as parameters, separate tiles by space (also use space to separate initial from goal states)

Example:

Initial state: 5 4 0

6 1 8

7 3 2

Goal State: 1 2 3

4 0 5

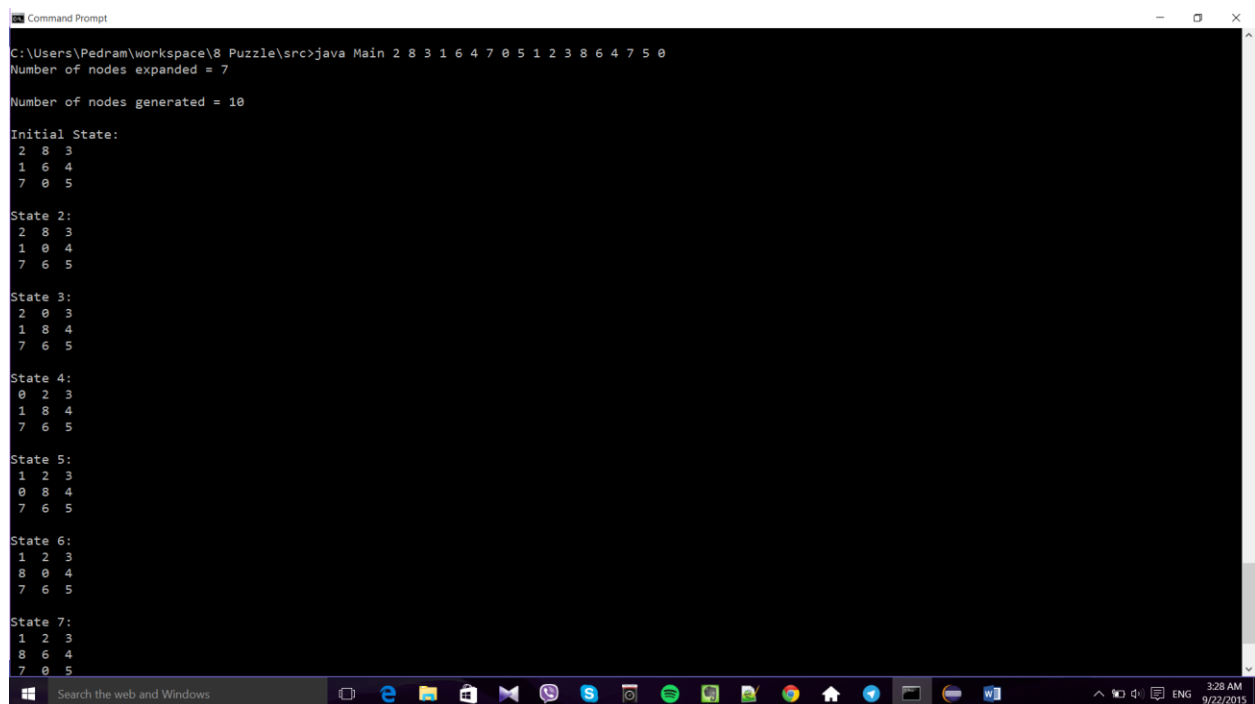
6 7 8

Command should be like:

`"java Main 5 4 0 6 1 8 7 3 2 1 2 3 4 0 5 6 7 8"`

The execution results for some example pairs:

1.



```
Command Prompt
C:\Users\Pedram\workspace\8 Puzzle\src>java Main 2 8 3 1 6 4 7 0 5 1 2 3 8 6 4 7 5 0
Number of nodes expanded = 7
Number of nodes generated = 10

Initial State:
2 8 3
1 6 4
7 0 5

State 2:
2 8 3
1 0 4
7 6 5

State 3:
2 0 3
1 8 4
7 6 5

State 4:
0 2 3
1 8 4
7 6 5

State 5:
1 2 3
0 8 4
7 6 5

State 6:
1 2 3
8 0 4
7 6 5

State 7:
1 2 3
8 6 4
7 0 5
```

2.

```
Command Prompt
C:\Users\Pedram\workspace\8 Puzzle\src>java Main 5 4 0 6 1 8 7 3 2 1 2 3 4 0 5 6 7 8
Number of nodes expanded = 22
Number of nodes generated = 342

Initial State:
5 4 0
6 1 8
7 3 2

State 2:
5 0 4
6 1 8
7 3 2

State 3:
5 1 4
6 0 8
7 3 2

State 4:
5 1 4
6 3 8
7 0 2

State 5:
5 1 4
6 3 8
7 2 0

State 6:
5 1 4
6 3 0
7 2 8

State 7:
5 1 4
6 0 3
7 2 8

State 8:
5 1 4
6 2 3
7 0 8

State 9:
5 1 4
6 2 3
0 7 8

State 10:
5 1 4
0 2 3
6 7 8

State 11:
5 1 4
2 0 3
6 7 8

State 12:
5 0 4
2 1 3
6 7 8

State 13:
0 5 4
2 1 3
6 7 8

State 14:
2 5 4
0 1 3
6 7 8

Synchronization complete
```

```
Command Prompt

6 7 8
State 14:
2 5 4
0 1 3
6 7 8

State 15:
2 5 4
1 0 3
6 7 8

State 16:
2 0 4
1 5 3
6 7 8

State 17:
2 4 0
1 5 3
6 7 8

State 18:
2 4 3
1 5 0
6 7 8

State 19:
2 4 3
1 0 5
6 7 8

State 20:
2 0 3
1 4 5
6 7 8

State 21:
0 2 3
1 4 5

C:\Users\Pedram\workspace\8 Puzzle\src>
```

```
Command Prompt

6 7 8
State 17:
2 4 0
1 5 3
6 7 8

State 18:
2 4 3
1 5 0
6 7 8

State 19:
2 4 3
1 0 5
6 7 8

State 20:
2 0 3
1 4 5
6 7 8

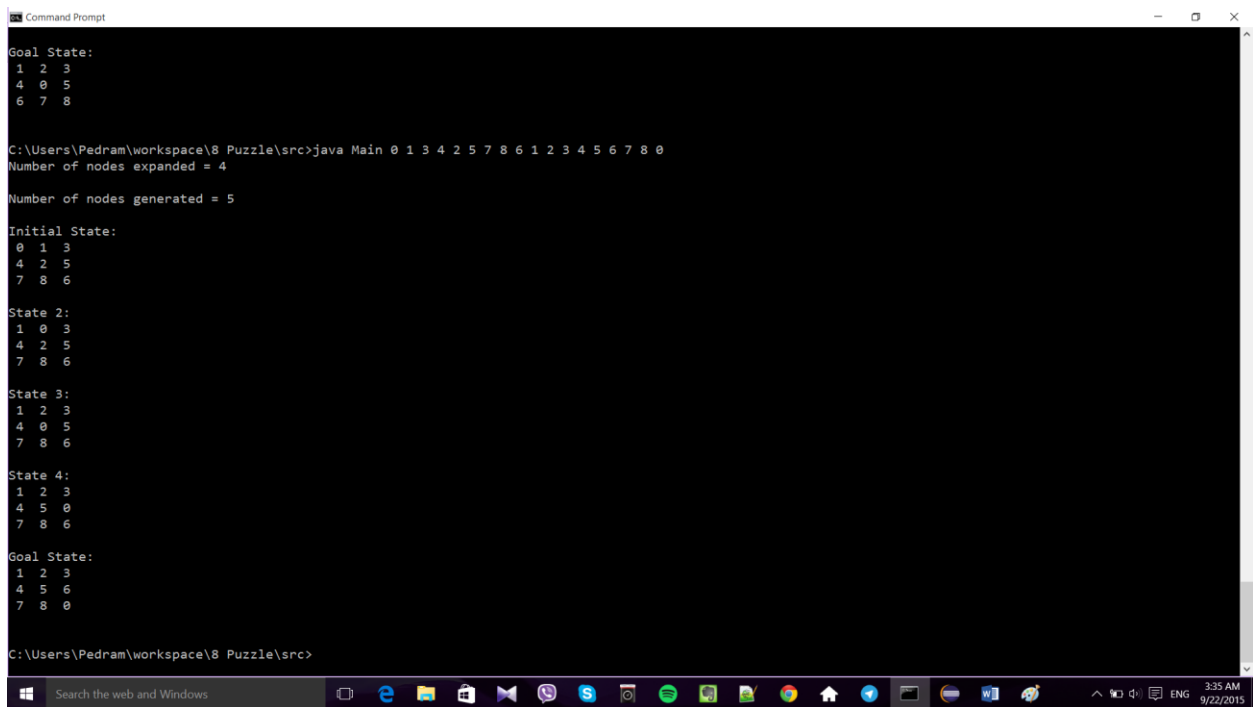
State 21:
0 2 3
1 4 5
6 7 8

State 22:
1 2 3
0 4 5
6 7 8

Goal State:
1 2 3
4 0 5
6 7 8

C:\Users\Pedram\workspace\8 Puzzle\src>
```

3.



```
Command Prompt

Goal State:
1 2 3
4 0 5
6 7 8

C:\Users\Pedram\workspace\8 Puzzle\src>java Main 0 1 3 4 2 5 7 8 6 1 2 3 4 5 6 7 8 0
Number of nodes expanded = 4

Number of nodes generated = 5

Initial State:
0 1 3
4 2 5
7 8 6

State 2:
1 0 3
4 2 5
7 8 6

State 3:
1 2 3
4 0 5
7 8 6

State 4:
1 2 3
4 5 0
7 8 6

Goal State:
1 2 3
4 5 6
7 8 0

C:\Users\Pedram\workspace\8 Puzzle\src>
```

## Source Code

### Main.java:

```
import java.util.Collections;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        AStar Solver = new AStar();

        int count = 0;
        int n = 3;

        // Get Initial State form input
        int[][] initialState = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                initialState[i][j] = Integer.parseInt(args[count]);
                count++;
            }
        }
    }
}
```

```

// Initialize the board
State initial = new State(initialState);

// Get Final State from input
Solver.finalState = new int[n][n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        Solver.finalState[i][j] = Integer.parseInt(args[count]);
        count++;
    }
}

// solve the puzzle
int nodesGenerated = Solver.solve(initial);

// print solution to standard output
if (!Solver.isSolvable())
    System.out.println("Unsolvable");
else {
    System.out.println("Number of nodes expanded = "
        + Solver.numberOfMoves() + "\n");

    System.out.println("Number of nodes generated = " + nodesGenerated
        + "\n");

    Iterable<State> result = Solver.solution();
    List<State> resultList = (List<State>) result;

    if (result != null) {
        Collections.reverse(resultList);

        int counter = 0;
        System.out.println("Initial State:");

        for (State board : resultList) {
            if (counter == 0)
                counter++;
            else if (counter == resultList.size() - 1)
                System.out.println("Goal State" + ":");
            else
                System.out.println("State " + ++counter + ":");
            System.out.println(board);
        }
    }
}
}

```

### AStar.java:

```
import java.util.PriorityQueue;
import java.util.Stack;

public class AStar {

    private boolean isSolvable;
    public int[][] finalState;
    private int numberOfMoves;
    private final Stack<State> states;

    private class SearchNode implements Comparable<SearchNode> {
        private State state;
        private int moves;
        private SearchNode previous;
        private int cachedPriority = -1;

        SearchNode(State state, int moves, SearchNode previous) {
            this.state = state;
            this.moves = moves;
            this.previous = previous;
        }

        private int priority() {
            if (cachedPriority == -1) {
                cachedPriority = moves + state.manhattan(finalState);
            }
            return cachedPriority;
        }

        @Override
        public int compareTo(SearchNode that) {
            if (this.priority() < that.priority()) {
                return -1;
            }
            if (this.priority() > that.priority()) {
                return +1;
            }
            return 0;
        }
    }

    public boolean isSolvable() {
```



```

        return isSolvable;
    }

    public int numberOfMoves() {
        if (isSolvable) {
            return states.size() - 1;
        } else {
            return -1;
        }
    }

    // Sequence of states from initial to goal state
    public Iterable<State> solution() {
        if (isSolvable) {
            return states;
        } else {
            return null;
        }
    }

    // Apply A* algorithm
    public int solve(State initial) {

        if (initial.isGoalState(finalState)) {
            isSolvable = true;
            this.states.push(initial);
            return 0;
        }
        if (initial.twin().isGoalState(finalState)) {
            isSolvable = false;
            return 0;
        }

        PriorityQueue<SearchNode> queue = new PriorityQueue<SearchNode>();
        PriorityQueue<SearchNode> twinQueue = new PriorityQueue<SearchNode>();

        numberOfMoves = 0;

        State board = initial;
        State boardTwin = initial.twin();

        SearchNode node = new SearchNode(board, 0, null);
        SearchNode nodeTwin = new SearchNode(boardTwin, 0, null);

```

```

queue.add(node);
twinQueue.add(nodeTwin);

while (numberOfMoves < 100) {
    node = queue.remove();
    nodeTwin = twinQueue.remove();

    board = node.state;
    boardTwin = nodeTwin.state;

    // Just one of a board and its twin is solvable
    if (boardTwin.isGoalState(finalState)) {
        isSolvable = false;
        return queue.size();
    }
    if (board.isGoalState(finalState)) {
        isSolvable = true;
        this.states.push(board);
        while (node.previous != null) {
            node = node.previous;
            this.states.push(node.state);
        }
        return queue.size();
    }

    node.moves++;
    nodeTwin.moves++;

    Iterable<State> neighbors = board.neighbors();
    for (State neighbor : neighbors) {
        if (node.previous != null
            && neighbor.equals(node.previous.state)) {
            continue;
        }
        SearchNode newNode = new SearchNode(neighbor, node.moves,
node);

        queue.add(newNode);
    }
    Iterable<State> neighborsTwin = boardTwin.neighbors();
    for (State neighbor : neighborsTwin) {
        if (nodeTwin.previous != null
            && neighbor.equals(nodeTwin.previous.state)) {
            continue;
        }
    }
}

```

```

        SearchNode newNode = new SearchNode(neighbor, nodeTwin.moves,
                                              nodeTwin);
        twinQueue.add(newNode);
    }
}
return queue.size();
}

public AStar() {
    // TODO Auto-generated constructor stub
    states = new Stack<State>();
}
}

```

#### **State.java:**

```

import java.util.Stack;

public class State {
    private final int N;
    private final int[][] Tiles;

    public State(int[][] positions) {
        N = positions.length;
        Tiles = new int[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                Tiles[i][j] = positions[i][j];
            }
        }
    }

    // is this board the goal board?
    public boolean isGoalState(int[][] goalState) {

        return compare(this.Tiles, goalState);
    }

    private boolean compare(int[][] first, int[][] second) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (first[i][j] != second[i][j]) {

```

```

        return false;
    }
}
return true;
}

// sum of Manhattan distances between tiles and goal State
public int manhattan(int[][] finalState) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int value = Tiles[i][j];
            if (value != 0 && value != finalState[i][j]) {
                int distance = Math.abs(i
                    - getFinalPosition(value, finalState)[0])
                    + Math.abs(j
                        - getFinalPosition(value,
finalState)[1]);
                sum += distance;
            }
        }
    }
    return sum;
}

public int[] getFinalPosition(int valueToSearch, int[][] finalState) {

    int[] result = new int[2];

    for (int i = 0; i < finalState.length; i++)
        for (int j = 0; j < finalState[0].length; j++) {
            if (finalState[i][j] == valueToSearch) {
                result[0] = i;
                result[1] = j;
                return result;
            }
        }
    return result;
}

// Returns a new Board result of swapping to adjacent tiles
public State twin() {

```

```

        State board = new State(Tiles);

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N - 1; j++) {
                if (Tiles[i][j] != 0 && Tiles[i][j + 1] != 0) {
                    board.swap(i, j, i, j + 1);
                    return board;
                }
            }
        }

        return board;
    }

```

```

private boolean swap(int i, int j, int it, int jt) {
    if (it < 0 || it >= N || jt < 0 || jt >= N) {
        return false;
    }
    int temp = Tiles[i][j];
    Tiles[i][j] = Tiles[it][jt];
    Tiles[it][jt] = temp;
    return true;
}

```

```

// returns neighboring states
public Iterable<State> neighbors() {
    int i0 = 0, j0 = 0;
    boolean found = false;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (Tiles[i][j] == 0) {
                i0 = i;
                j0 = j;
                found = true;
                break;
            }
        }
        if (found) {
            break;
        }
    }
}

```

```

Stack<State> boards = new Stack<State>();
State board = new State(Tiles);

```

```

        boolean isNeighbor = board.swap(i0, j0, i0 - 1, j0);
        if (isNeighbor) {
            boards.push(board);
        }
        board = new State(Tiles);
        isNeighbor = board.swap(i0, j0, i0, j0 - 1);
        if (isNeighbor) {
            boards.push(board);
        }
        board = new State(Tiles);
        isNeighbor = board.swap(i0, j0, i0 + 1, j0);
        if (isNeighbor) {
            boards.push(board);
        }
        board = new State(Tiles);
        isNeighbor = board.swap(i0, j0, i0, j0 + 1);
        if (isNeighbor) {
            boards.push(board);
        }

        return boards;
    }

```

```

@Override
public String toString() {
    StringBuilder s = new StringBuilder();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            s.append(String.format("%2d ", Tiles[i][j]));
        }
        s.append("\n");
    }
    return s.toString();
}

```

```

@Override
public boolean equals(Object x) {
    if (x == this)
        return true;
    if (x == null)
        return false;
    if (x.getClass() != this.getClass())
        return false;
}

```

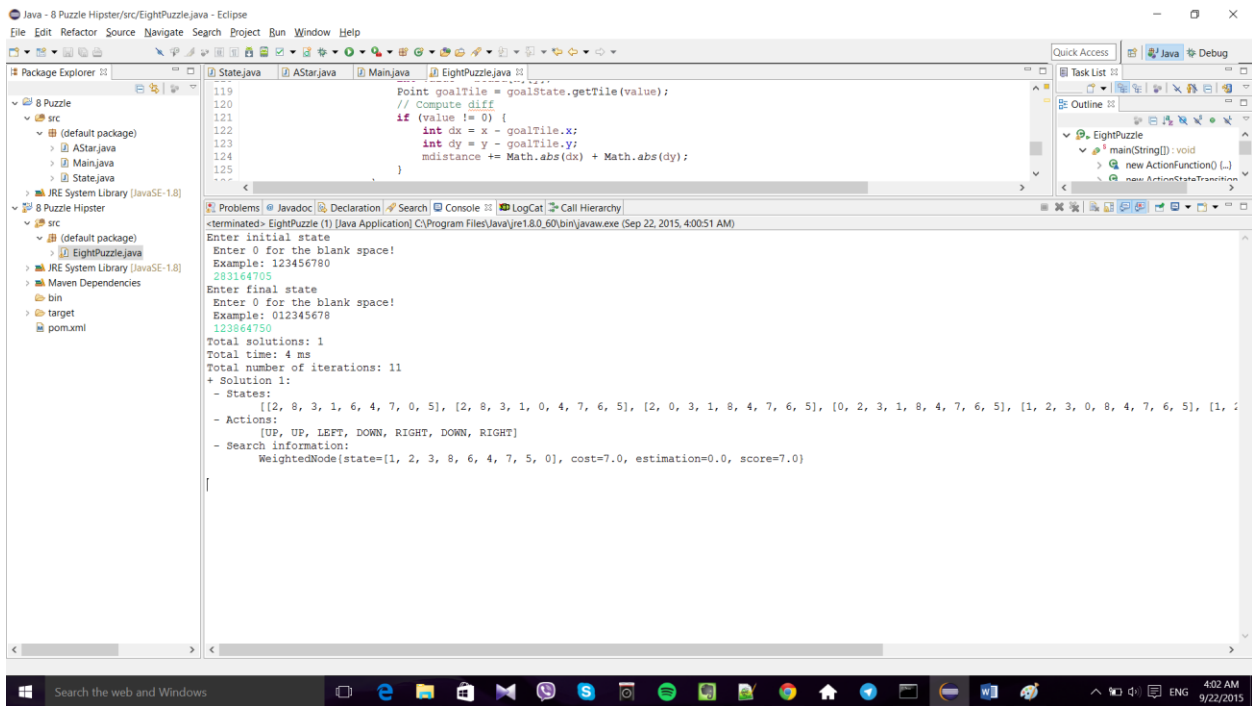
$$\left. \begin{array}{l} \{ \\ \} \end{array} \right\}$$

## Second Implementation

As I was researching about the subject I came across this interesting library called hipster -An Open Source Java Library for Heuristic Search- which is an easy to use yet powerful library for heuristic search, written in pure Java. <http://www.hipster4j.org/>

Using this library I could easily define the 8 puzzle problem, search strategy and introduce the desired heuristic function. The library solves the problem using A\* with Manhattan distance as the heuristic and returns the path as well as detailed information of the actions, time, and number of iterations.

Below is a result of running the code:



To execute this program the best way is to open the code with eclipse because I have used maven to add the libraries. Compiling maven projects from windows command prompt is not really convenient.

Source Code:

### **EightPuzzle.java:**

```
import java.awt.Point;
import java.util.LinkedList;
import java.util.Scanner;

import es.usc.citius.hipster.algorithm.Hipster;
import es.usc.citius.hipster.examples.problem.NPuzzle.Puzzle;
import es.usc.citius.hipster.examples.problem.NPuzzle.PuzzleMove;
import es.usc.citius.hipster.model.Transition;
import es.usc.citius.hipster.model.function.ActionFunction;
import es.usc.citius.hipster.model.function.ActionStateTransitionFunction;
import es.usc.citius.hipster.model.function.CostFunction;
import es.usc.citius.hipster.model.function.HeuristicFunction;
import es.usc.citius.hipster.model.impl.WeightedNode;
import es.usc.citius.hipster.model.problem.ProblemBuilder;
import es.usc.citius.hipster.model.problem.SearchProblem;

public class EightPuzzle {

    public static void main(String[] args) {

        System.out
            .print("Enter initial state \n Enter 0 for the blank space! \n Example:
123456780 \n ");
        Scanner scan = new Scanner(System.in);
        String input = scan.nextLine();

        final Puzzle initialState = new Puzzle(new int[] {
            input.charAt(0) - 48, input.charAt(1) - 48,
            input.charAt(2) - 48, input.charAt(3) - 48,
            input.charAt(4) - 48, input.charAt(5) - 48,
            input.charAt(6) - 48, input.charAt(7) - 48,
            input.charAt(8) - 48 });

        System.out
            .print("Enter final state \n Enter 0 for the blank space! \n Example:
012345678 \n ");
        input = scan.nextLine();

        final Puzzle goalState = new Puzzle(new int[] { input.charAt(0) - 48,
            input.charAt(1) - 48, input.charAt(2) - 48,
            input.charAt(3) - 48, input.charAt(4) - 48,
            input.charAt(5) - 48, input.charAt(6) - 48,
            input.charAt(7) - 48, input.charAt(8) - 48 });
```



```

// ActionFunction: Taking an state as input, determine the actions which
// can be applied to reach other state.
ActionFunction<PuzzleMove, Puzzle> af = new ActionFunction<PuzzleMove, Puzzle>() {
    @Override
    public Iterable<PuzzleMove> actionsFor(Puzzle state) {
        LinkedList<PuzzleMove> movements = new LinkedList<PuzzleMove>();
        // Get which place the gap tile is in
        Point gap = state.getTile(0);
        // side size of the board
        int boardSize = state.getMatrixBoard().length;

        // Check valid movements. There are always maximum
        // 4 tiles around the gap (left, right, top, down)
        if (gap.getX() > 0 && gap.getX() < boardSize) {
            movements.add(PuzzleMove.UP);
        }
        if (gap.getX() >= 0 && gap.getX() < boardSize - 1) {
            movements.add(PuzzleMove.DOWN);
        }
        if (gap.getY() >= 0 && gap.getY() < boardSize - 1) {
            movements.add(PuzzleMove.RIGHT);
        }
        if (gap.getY() > 0 && gap.getY() < boardSize) {
            movements.add(PuzzleMove.LEFT);
        }
        return movements;
    }
};

```

```

// ActionStateTransitionFunction: Taking as input the current state and
// current action. It generates following state after applying the
// action.

```

```

ActionStateTransitionFunction<PuzzleMove, Puzzle> atf = new
ActionStateTransitionFunction<PuzzleMove, Puzzle>() {
    @Override
    public Puzzle apply(PuzzleMove action, Puzzle state) {
        // Generate the next board
        Point gap = state.getTile(0);
        int[][] board = state.copyBoard();
        // System.out.println("Applying " + action + " to " + state +
        // " gap: " + gap.toString());
        // x=row, y=column
        switch (action) {

```

```

        case UP:
            board[gap.x][gap.y] = state.getMatrixBoard()[gap.x - 1][gap.y];
            board[gap.x - 1][gap.y] = 0;
            break;
        case DOWN:
            board[gap.x][gap.y] = state.getMatrixBoard()[gap.x + 1][gap.y];
            board[gap.x + 1][gap.y] = 0;
            break;
        case LEFT:
            board[gap.x][gap.y] = state.getMatrixBoard()[gap.x][gap.y - 1];
            board[gap.x][gap.y - 1] = 0;
            break;
        case RIGHT:
            board[gap.x][gap.y] = state.getMatrixBoard()[gap.x][gap.y + 1];
            board[gap.x][gap.y + 1] = 0;
            break;
    }
    Puzzle successor = new Puzzle(board);
    return successor;
}

};

// definition of an uniform cost function g(n)
CostFunction<PuzzleMove, Puzzle, Double> cf = new CostFunction<PuzzleMove, Puzzle,
Double>() {
    @Override
    public Double evaluate(Transition<PuzzleMove, Puzzle> transition) {
        return 1d;
    }
};

// definition of an heuristic, the Manhattan distance between states
// h(n)
HeuristicFunction<Puzzle, Double> hf = new HeuristicFunction<Puzzle, Double>() {
    @Override
    public Double estimate(Puzzle state) {
        // Compute the manhattan distance
        int mdistance = 0;
        int[][] board = state.getMatrixBoard();
        int size = board.length;
        for (int x = 0; x < size; x++)
            for (int y = 0; y < size; y++) {
                int value = board[x][y];
                Point goalTile = goalState.getTile(value);

```

```

        // Compute diff
        if (value != 0) {
            int dx = x - goalTile.x;
            int dy = y - goalTile.y;
            mdistance += Math.abs(dx) + Math.abs(dy);
        }
    }
    return (double) mdistance;
}
};

/*
 * Search problem is instantiated defining all the components to be used
 * in the search: the initial state, the action and transition function,
 * and the cost and heuristic SO let's formulate the problem.
 */

SearchProblem<PuzzleMove, Puzzle, WeightedNode<PuzzleMove, Puzzle, Double>> p =
ProblemBuilder

    .create().initialState(initialState)
    .defineProblemWithExplicitActions().useActionFunction(af)
    .useTransitionFunction(atf).useCostFunction(cf)
    .useHeuristicFunction(hf).build();

// Here we assign A* as the search algorithm
// search() method starts the search process until the goal state is
// reached
// and print out time, number of iterations, and cost of the search .
System.out.println(Hipster.createAStar(p).search(goalState));

}
}

```