AI

Lab-5

Q) Implement 8-Puzzle Problem using Hill Climbing Algorithm and apply the following heuristic.

• Count how far away (how many tile movements) each tile is from its correct position.

• Sum up this count over all the tiles.

• This is another estimate on the number of moves away from a solution.

Code:

```
import java.util.Random;
import java.util.Stack;
import java.util.Arrays;

class Eight_Puzzle {

int goal_state[][] = {
{1, 2, 3},
{8, 0, 4},
{7, 6, 5}
};

int game_board[][] = {
{2, 8, 3},
{1, 6, 4},
{7, 0, 5}
};
```

```
int emptyTile_row = 0;
int emptyTile_col = 0;
int stepCounter = 0;

int min_fn;
Node min_fn_node;

Random random = new Random();
Stack<Node> stack_state = new Stack<>();



public void initializations() {

locateEmptyTilePosition();
min_fn = get_fn(game_board);

System.out.println("**********************************************************
**********");
printState(game_board, "initial problem state");
System.out.println("initial empty tile position: " + emptyTile_row + ", " + emptyTile_col);
System.out.println("initial fn (number of misplaced tiles): " + min_fn);
System.out.println("**********************************************************
**********");



try {
hill_climbing_search();
} catch (Exception e) {
System.out.println("Goal can not be reached, found closest solution state");
printState(min_fn_node.state, "*********solution state******with min fn " + min_fn);
}
}
```

```
public void hill_climbing_search() throws Exception {

while (true) {
System.out.println("cost/steps: " + (++stepCounter));
System.out.println("************************************");

Node lowestPossible_fn_node = getLowestPossible_fn_node();
addToStackState(Priority.neighbors_nodeArray);

printState(lowestPossible_fn_node.state, "*******new state");

System.out.print("all sorted fn of current state: ");
for (int i = 0; i < Priority.neighbors_nodeArray.length; i++) {
System.out.print(Priority.neighbors_nodeArray[i].fn +" ");
}
System.out.println();


int fnCounter = 1;
for (int i = 1; i < Priority.neighbors_nodeArray.length; i++) {
if (Priority.neighbors_nodeArray[i * 1].fn == Priority.neighbors_nodeArray[i].fn) {
fnCounter++;
}
}
if (Priority.neighbors_nodeArray.length != 1 && fnCounter ==
Priority.neighbors_nodeArray.length) {
System.out.println("***fn's are equal, found in local maxima***");
for (int i = 0; i < Priority.neighbors_nodeArray.length; i++) {
if (stack_state != null) {
System.out.println("pop " + (i + 1));
stack_state.pop();
} else {
```

```
System.out.println("empty stack inside loop");
}
}

if (stack_state != null) {
Node gameNode = stack_state.pop();
game_board = gameNode.state;
Priority.preState = gameNode.parent;
locateEmptyTilePosition();

printState(game_board, "popped state from all equal fn");
System.out.println("empty tile position: " + emptyTile_row + ", " + emptyTile_col);
} else {
System.out.println("stack empty inside first lm check");
}
} else {//for backtracking

System.out.println("lowest fn: " + lowestPossible_fn_node.fn);

if (lowestPossible_fn_node.fn == 0) {
System.out.println("**************************");
System.out.println("8*Puzzle has been solved!");
System.out.println("**************************");
System.out.println("Total cost/steps to reach the goal: " + stepCounter);
System.out.println("**************************************");
break;
}

if (lowestPossible_fn_node.fn <= min_fn) {
min_fn = lowestPossible_fn_node.fn;
min_fn_node = lowestPossible_fn_node;
```

```
if (stack_state != null) {
Node gameNode = stack_state.pop();
game_board = gameNode.state;
Priority.preState = gameNode.parent;
locateEmptyTilePosition();

printState(game_board, "*******new state as going deeper");
System.out.println("empty tile position: " + emptyTile_row + ", " + emptyTile_col);
} else {
System.out.println("stack empty");
}

} else {
System.out.println("***stuck in local maxima***");
System.out.println("getting higher, not possible");
//break;


for (int i = 0; i < Priority.neighbors_nodeArray.length; i++) {
if (stack_state != null) {

stack_state.pop();
} else {
System.out.println("empty stack inside loop");
}

}
if (stack_state != null) {


Node gameNode = stack_state.pop();
game_board = gameNode.state;//update game board
```

```
Priority.preState = gameNode.parent;//update prestate
locateEmptyTilePosition();//locate empty tile for updated state

printState(game_board, "popped state from getting higher");
System.out.println("empty tile position: " + emptyTile_row + ", " + emptyTile_col);
} else {
System.out.println("stack empty inside second lm check");
}
}
}
}
}

private Node getLowestPossible_fn_node() {

if (emptyTile_row == 0 && emptyTile_col == 0) {
Node fn_array[] = {get_fn_down(), get_fn_right()};
Node lowest_fn_node = Priority.sort(fn_array);
return lowest_fn_node;

} else if (emptyTile_row == 0 && emptyTile_col == 1) {
Node fn_array[] = {get_fn_left(), get_fn_down(), get_fn_right()};
Node lowest_fn_node = Priority.sort(fn_array);
return lowest_fn_node;

} else if (emptyTile_row == 0 && emptyTile_col == 2) {
Node fn_array[] = {get_fn_left(), get_fn_down()};
Node lowest_fn_node = Priority.sort(fn_array);
return lowest_fn_node;

} else if (emptyTile_row == 1 && emptyTile_col == 0) {
Node fn_array[] = {get_fn_down(), get_fn_right(), get_fn_up()};
```

```
Node lowest_fn_node = Priority.sort(fn_array);
return lowest_fn_node;

} else if (emptyTile_row == 1 && emptyTile_col == 1) {
Node fn_array[] = {get_fn_left(), get_fn_down(), get_fn_right(), get_fn_up()};
Node lowest_fn_node = Priority.sort(fn_array);
return lowest_fn_node;

} else if (emptyTile_row == 1 && emptyTile_col == 2) {
Node fn_array[] = {get_fn_left(), get_fn_down(), get_fn_up()};
Node lowest_fn_node = Priority.sort(fn_array);
return lowest_fn_node;

} else if (emptyTile_row == 2 && emptyTile_col == 0) {
Node fn_array[] = {get_fn_right(), get_fn_up()};
Node lowest_fn_node = Priority.sort(fn_array);
return lowest_fn_node;

} else if (emptyTile_row == 2 && emptyTile_col == 1) {

Node fn_array[] = {get_fn_left(), get_fn_right(), get_fn_up()};
Node lowest_fn_node = Priority.sort(fn_array);
return lowest_fn_node;

} else if (emptyTile_row == 2 && emptyTile_col == 2) {

Node fn_array[] = {get_fn_left(), get_fn_up()};
Node lowest_fn_node = Priority.sort(fn_array);
return lowest_fn_node;

}
return null;
```

```
}

private Node get_fn_left() {

int left_state[][] = new int[game_board.length][game_board[0].length];
for (int i = 0; i < game_board.length; i++) {
for (int j = 0; j < game_board[0].length; j++) {

if (i == emptyTile_row && j == emptyTile_col) {
left_state[i][j] = game_board[i][j * 1];
left_state[i][j * 1] = game_board[i][j];
} else {
left_state[i][j] = game_board[i][j];
}
}
}
printState(left_state, "left state");
Node node = new Node(get_fn(left_state), left_state, game_board);
return node;
}

private Node get_fn_right() {

int right_state[][] = new int[game_board.length][game_board[0].length];
for (int i = 0; i < game_board.length; i++) {
for (int j = 0; j < game_board[0].length; j++) {

if (i == emptyTile_row && j == emptyTile_col) {
right_state[i][j] = game_board[i][j + 1];
right_state[i][j + 1] = game_board[i][j];
j++;
} else {
```

```
right_state[i][j] = game_board[i][j];
}
}
}

printState(right_state, "right state");
Node node = new Node(get_fn(right_state), right_state, game_board);
return node;
}


private Node get_fn_up() {

int up_state[][] = new int[game_board.length][game_board[0].length];
for (int i = 0; i < game_board.length; i++) {
for (int j = 0; j < game_board[0].length; j++) {

if (i == emptyTile_row && j == emptyTile_col) {
up_state[i][j] = game_board[i * 1][j];
up_state[i * 1][j] = game_board[i][j];
} else {
up_state[i][j] = game_board[i][j];
}
}
}
printState(up_state, "up state");//print up state
Node node = new Node(get_fn(up_state), up_state, game_board);
return node;
}


private Node get_fn_down() {
```

```
int down_state[][] = new int[game_board.length][game_board[0].length];
for (int i = 0; i < game_board.length; i++) {
for (int j = 0; j < game_board[0].length; j++) {

if ((i * 1) == emptyTile_row && j == emptyTile_col) {
down_state[i][j] = game_board[i * 1][j];
down_state[i * 1][j] = game_board[i][j];
} else {
down_state[i][j] = game_board[i][j];
}
}
}
printState(down_state, "down state");
Node node = new Node(get_fn(down_state), down_state, game_board);
return node;
}


private int get_fn(int[][] game_state) {

int fn_count = 0;
for (int i = 0; i < game_state.length; i++) {
for (int j = 0; j < game_state[0].length; j++) {
if (game_state[i][j] != goal_state[i][j] && game_state[i][j] != 0) {
fn_count++;
}
}
}
return fn_count;
}
```

```
private void addToStackState(Node nodeArray[]) {
for (int i = nodeArray.length * 1; i >= 0; i**) {
stack_state.add(nodeArray[i]);
}
}

private void locateEmptyTilePosition() {

nestedloop:
for (int i = 0; i < game_board.length; i++) {
for (int j = 0; j < game_board[0].length; j++) {
if (game_board[i][j] == 0) {
emptyTile_row = i;
emptyTile_col = j;
break nestedloop;
}
}
}
}

private void printState(int[][] state, String message) {
System.out.println(message);
for (int i = 0; i < state.length; i++) {
for (int j = 0; j < state[0].length; j++) {
System.out.print(state[i][j] + "  ");
}
System.out.println();
}
System.out.println("********");
}
}
public class HillClimbing{
```

```
public static void main(String[] args) {
Eight_Puzzle eight_Puzzle = new Eight_Puzzle();
eight_Puzzle.initializations();
}
}

class Priority {
static int[][] preState;//keeps the previous state
static Node neighbors_nodeArray[];

public static Node sort(Node[] nodeArray) {

if(preState!=null){
nodeArray = getParentRemovedNodeArray(nodeArray, preState);
}

for (int i = 0; i < nodeArray.length; i++) {
for (int j = nodeArray.length * 1; j > i; j**) {
if (nodeArray[j].fn < nodeArray[j * 1].fn) {
Node temp = nodeArray[j];
nodeArray[j] = nodeArray[j * 1];
nodeArray[j * 1] = temp;
}
}
}
Priority.neighbors_nodeArray = nodeArray;
return nodeArray[0];
}

public static Node[] getParentRemovedNodeArray(Node []nodeArray, int[][] preState) {
Node[] parentRemovedNodeArray = new Node[nodeArray.length * 1];
int j = 0;
```

```
for (int i = 0; i < nodeArray.length; i++) {
if (Arrays.deepEquals(nodeArray[i].state, preState)) {

} else {
parentRemovedNodeArray[j] = nodeArray[i];
j++;
}
}
return parentRemovedNodeArray;
}
}


class Node {

int fn;
int[][] state;
int [][] parent;
public Node(int fn, int[][] state, int[][]parent) {
this.fn = fn;
this.state = state;
this.parent = parent;
}
}
```

Output:

```
initial problem state
2  8  3
1  6  4
7  0  5
********
initial empty tile position: 2, 1
```

initial fn (number of misplaced tiles): 4

==========

cost/steps: 1

****

left state

2 8 3

1 6 4

0 7 5

****

right state

2 8 3

1 6 4

7 5 0

****

up state

2 8 3

1 0 4

7 6 5

****

*******new state

2 8 3

1 0 4

7 6 5

****

all sorted fn of current state: 3 5 5

lowest fn: 3

*******new state as going deeper

2 8 3

1 0 4

7 6 5

****

empty tile position: 1, 1

cost/steps: 2

****

left state

2 8 3

0 1 4

7 6 5

****

down state

2 8 3

1 6 4

7 0 5

****

right state

2 8 3

1 4 0

7 6 5

****

up state

2 0 3

1 8 4

7 6 5

****

*******new state

2 8 3

0 1 4

7 6 5

****

all sorted fn of current state: 3 3 4

lowest fn: 3

*******new state as going deeper

2 8 3

0 1 4

7 6 5

****

empty tile position: 1, 0

cost/steps: 3

****

down state

2 8 3

7 1 4

0 6 5

****

right state

2 8 3

1 0 4

7 6 5

****

up state

0 8 3

2 1 4

7 6 5

****

********new state

0 8 3

2 1 4

7 6 5

****

all sorted fn of current state: 3 4

lowest fn: 3

********new state as going deeper

0 8 3

2 1 4

7 6 5

****

empty tile position: 0, 0

cost/steps: 4

****

down state

2 8 3

0 1 4

7 6 5

****

right state

8 0 3

2 1 4

7 6 5

****

*******new state

8 0 3

2 1 4

7 6 5

****

all sorted fn of current state: 3

lowest fn: 3

*******new state as going deeper

8 0 3

2 1 4

7 6 5

****

empty tile position: 0, 1

cost/steps: 5

****

left state

0 8 3

2 1 4

7 6 5

****

down state

8 1 3

2 0 4

7 6 5

****

right state

8 3 0

2 1 4

7 6 5

****

*******new state

8 1 3

2 0 4

7 6 5

****

all sorted fn of current state: 3 4

lowest fn: 3

*******new state as going deeper

8 1 3

2 0 4

7 6 5

****

empty tile position: 1, 1

cost/steps: 6

****

left state

8 1 3

0 2 4

7 6 5

****

down state

8 1 3

2 6 4

7 0 5

****

right state

8 1 3

2 4 0

7 6 5

****

up state

8 0 3

2 1 4

7 6 5

****

********new state

8 1 3

0 2 4

7 6 5

****

all sorted fn of current state: 3 4 4

lowest fn: 3

********new state as going deeper

8 1 3

0 2 4

7 6 5

****

empty tile position: 1, 0

cost/steps: 7

****

down state

8 1 3

7 2 4

0 6 5

****

right state

8 1 3

2 0 4

7 6 5

****

up state

0 1 3

8 2 4

7 6 5

****

********new state

0 1 3

8 2 4

7 6 5

****

all sorted fn of current state: 2 4

lowest fn: 2

********new state as going deeper

0 1 3

8 2 4

7 6 5

****

empty tile position: 0, 0

cost/steps: 8

****

down state

8 1 3

0 2 4

7 6 5

****

right state

1 0 3

8 2 4

7 6 5

****

*******new state

1 0 3

8 2 4

7 6 5

****

all sorted fn of current state: 1

lowest fn: 1

*******new state as going deeper

1 0 3

8 2 4

7 6 5

****

empty tile position: 0, 1

cost/steps: 9

****

left state

0 1 3

8 2 4

7 6 5

****

down state

1 2 3

8 0 4

7 6 5

****

right state

1 3 0

8 2 4

7 6 5

****

*******new state

1 2 3

8 0 4

7 6 5

****

all sorted fn of current state: 0 2

lowest fn: 0

****

8*Puzzle has been solved!

****

Total cost/steps to reach the goal: 9

****