NAME: Soham Phadke  
CLASS: TY\_CS\_D

BATCH: 2

ROLL NO: 37

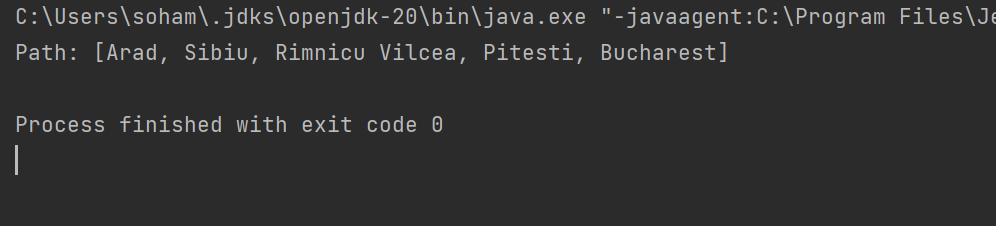
PRN NO: 12111222

**ASSIGNMENT 3**

**Problem Statement:** Informed search: A star algorithm on Romania map problem

**Code:**

import java.util.PriorityQueue;  
import java.util.HashSet;  
import java.util.Set;  
import java.util.List;  
import java.util.Comparator;  
import java.util.ArrayList;  
import java.util.Collections;  
  
public class AStar{  
  
  
 public static void main(String[] args){  
  
 Node n1 = new Node("Arad",366);  
 Node n2 = new Node("Zerind",374);  
 Node n3 = new Node("Oradea",380);  
 Node n4 = new Node("Sibiu",253);  
 Node n5 = new Node("Fagaras",178);  
 Node n6 = new Node("Rimnicu Vilcea",193);  
 Node n7 = new Node("Pitesti",98);  
 Node n8 = new Node("Timisoara",329);  
 Node n9 = new Node("Lugoj",244);  
 Node n10 = new Node("Mehadia",241);  
 Node n11 = new Node("Drobeta",242);  
 Node n12 = new Node("Craiova",160);  
 Node n13 = new Node("Bucharest",0);  
 Node n14 = new Node("Giurgiu",77);  
  
  
 //Arad  
 n1.adjacencies = new Edge[]{  
 new Edge(n2,75),  
 new Edge(n4,140),  
 new Edge(n8,118)  
 };  
  
 //Zerind  
 n2.adjacencies = new Edge[]{  
 new Edge(n1,75),  
 new Edge(n3,71)  
 };  
  
  
 //Oradea  
 n3.adjacencies = new Edge[]{  
 new Edge(n2,71),  
 new Edge(n4,151)  
 };  
  
 //Sibiu  
 n4.adjacencies = new Edge[]{  
 new Edge(n1,140),  
 new Edge(n5,99),  
 new Edge(n3,151),  
 new Edge(n6,80),  
 };  
  
  
 //Fagaras  
 n5.adjacencies = new Edge[]{  
 new Edge(n4,99),  
  
 //178  
 new Edge(n13,211)  
 };  
  
 //Rimnicu Vilcea  
 n6.adjacencies = new Edge[]{  
 new Edge(n4,80),  
 new Edge(n7,97),  
 new Edge(n12,146)  
 };  
  
 //Pitesti  
 n7.adjacencies = new Edge[]{  
 new Edge(n6,97),  
 new Edge(n13,101),  
 new Edge(n12,138)  
 };  
  
 //Timisoara  
 n8.adjacencies = new Edge[]{  
 new Edge(n1,118),  
 new Edge(n9,111)  
 };  
  
 //Lugoj  
 n9.adjacencies = new Edge[]{  
 new Edge(n8,111),  
 new Edge(n10,70)  
 };  
  
 //Mehadia  
 n10.adjacencies = new Edge[]{  
 new Edge(n9,70),  
 new Edge(n11,75)  
 };  
  
 //Drobeta  
 n11.adjacencies = new Edge[]{  
 new Edge(n10,75),  
 new Edge(n12,120)  
 };  
  
 //Craiova  
 n12.adjacencies = new Edge[]{  
 new Edge(n11,120),  
 new Edge(n6,146),  
 new Edge(n7,138)  
 };  
  
 //Bucharest  
 n13.adjacencies = new Edge[]{  
 new Edge(n7,101),  
 new Edge(n14,90),  
 new Edge(n5,211)  
 };  
  
 //Giurgiu  
 n14.adjacencies = new Edge[]{  
 new Edge(n13,90)  
 };  
  
 *AstarSearch*(n1,n13);  
  
 List<Node> path = *printPath*(n13);  
  
 System.*out*.println("Path: " + path);  
  
  
 }  
  
 public static List<Node> printPath(Node goal){  
 List<Node> path = new ArrayList<Node>();  
  
 for(Node node = goal; node!=null; node = node.parent){  
 path.add(node);  
 }  
  
 Collections.*reverse*(path);  
  
 return path;  
 }  
  
 public static void AstarSearch(Node source, Node goal){  
  
 Set<Node> explored = new HashSet<Node>();  
  
 PriorityQueue<Node> queue = new PriorityQueue<Node>(20,  
 new Comparator<Node>(){  
 @Override  
 public int compare(Node i, Node j){  
 if(i.f\_scores > j.f\_scores){  
 return 1;  
 }  
  
 else if (i.f\_scores < j.f\_scores){  
 return -1;  
 }  
  
 else{  
 return 0;  
 }  
 }  
  
 }  
 );  
  
 //cost from start  
 source.g\_scores = 0;  
  
 queue.add(source);  
  
 boolean found = false;  
  
 while((!queue.isEmpty())&&(!found)){  
  
 //the node in having the lowest f\_score value  
 Node current = queue.poll();  
  
 explored.add(current);  
  
 //goal found  
 if(current.value.equals(goal.value)){  
 found = true;  
 }  
  
 //check every child of current node  
 for(Edge e : current.adjacencies){  
 Node child = e.target;  
 double cost = e.cost;  
 double temp\_g\_scores = current.g\_scores + cost;  
 double temp\_f\_scores = temp\_g\_scores + child.h\_scores;  
  
 if((explored.contains(child)) &&  
 (temp\_f\_scores >= child.f\_scores)){  
 continue;  
 }  
  
 else if((!queue.contains(child)) ||  
 (temp\_f\_scores < child.f\_scores)){  
  
 child.parent = current;  
 child.g\_scores = temp\_g\_scores;  
 child.f\_scores = temp\_f\_scores;  
  
 if(queue.contains(child)){  
 queue.remove(child);  
 }  
  
 queue.add(child);  
  
 }  
  
 }  
  
 }  
  
 }  
  
}  
  
class Node{  
  
 public final String value;  
 public double g\_scores;  
 public final double h\_scores;  
 public double f\_scores = 0;  
 public Edge[] adjacencies;  
 public Node parent;  
  
 public Node(String val, double hVal){  
 value = val;  
 h\_scores = hVal;  
 }  
  
 public String toString(){  
 return value;  
 }  
  
}  
  
class Edge{  
 public final double cost;  
 public final Node target;  
  
 public Edge(Node targetNode, double costVal){  
 target = targetNode;  
 cost = costVal;  
 }  
}

**Output:  
**