**ASSIGNMENT NO.2**

importjava.util.PriorityQueue;

importjava.util.HashSet;

importjava.util.Set;

importjava.util.List;

importjava.util.Comparator;

importjava.util.ArrayList;

importjava.util.Collections;

publicclassAstarSearchAlgo{

        //h scores is the stright-line distance from the current city to Bucharest

        publicstaticvoidmain(String[] args){

                //initialize the graph base on the Romania map

                Noden1 = newNode("Arad",366);

                Noden2 = newNode("Zerind",374);

                Noden3 = newNode("Oradea",380);

                Noden4 = newNode("Sibiu",253);

                Noden5 = newNode("Fagaras",178);

                Noden6 = newNode("Rimnicu Vilcea",193);

                Noden7 = newNode("Pitesti",98);

                Noden8 = newNode("Timisoara",329);

                Noden9 = newNode("Lugoj",244);

                Noden10 = newNode("Mehadia",241);

                Noden11 = newNode("Drobeta",242);

                Noden12 = newNode("Craiova",160);

                Noden13 = newNode("Bucharest",0);

                        Noden14 = newNode("Giurgiu",77);

                //initialize the edges

                //Arad

                n1.adjacencies = newEdge[]{

                        newEdge(n2,75),

                        newEdge(n4,140),

                        newEdge(n8,118)

                };

                 //Zerind

                n2.adjacencies = newEdge[]{

                        newEdge(n1,75),

                        newEdge(n3,71)

                };

                 //Oradea

                n3.adjacencies = newEdge[]{

                        newEdge(n2,71),

                        newEdge(n4,151)

                };

                 //Sibiu

                n4.adjacencies = newEdge[]{

                        newEdge(n1,140),

                        newEdge(n5,99),

                        newEdge(n3,151),

                        newEdge(n6,80),

                };

                 //Fagaras

                n5.adjacencies = newEdge[]{

                        newEdge(n4,99),

                        //178

                        newEdge(n13,211)

                };

                 //RimnicuVilcea

                n6.adjacencies = newEdge[]{

                        newEdge(n4,80),

                        newEdge(n7,97),

                        newEdge(n12,146)

                };

                 //Pitesti

                n7.adjacencies = newEdge[]{

                        newEdge(n6,97),

                        newEdge(n13,101),

                        newEdge(n12,138)

                };

                 //Timisoara

                n8.adjacencies = newEdge[]{

                        newEdge(n1,118),

                        newEdge(n9,111)

                };

                 //Lugoj

                n9.adjacencies = newEdge[]{

                        newEdge(n8,111),

                        newEdge(n10,70)

                };

                 //Mehadia

                n10.adjacencies = newEdge[]{

                        newEdge(n9,70),

                        newEdge(n11,75)

                };

                 //Drobeta

                n11.adjacencies = newEdge[]{

                        newEdge(n10,75),

                        newEdge(n12,120)

                };

                 //Craiova

                n12.adjacencies = newEdge[]{

                        newEdge(n11,120),

                        newEdge(n6,146),

                        newEdge(n7,138)

                };

                //Bucharest

                n13.adjacencies = newEdge[]{

                        newEdge(n7,101),

                        newEdge(n14,90),

                        newEdge(n5,211)

                };

                 //Giurgiu

                n14.adjacencies = newEdge[]{

                        newEdge(n13,90)

                };

                AstarSearch(n1,n13);

                List<Node>path = printPath(n13);

                        System.out.println("Path: " + path);

        }

        publicstaticList<Node>printPath(Nodetarget){

                List<Node>path = newArrayList<Node>();

        for(Nodenode = target; node!=null; node = node.parent){

            path.add(node);

        }

        Collections.reverse(path);

        return path;

        }

        publicstaticvoidAstarSearch(Nodesource, Nodegoal){

                Set<Node>explored = newHashSet<Node>();

                PriorityQueue<Node>queue = newPriorityQueue<Node>(20,

                        newComparator<Node>(){

                                 //override compare method

                 publicintcompare(Nodei, Nodej){

                    if(i.f\_scores>j.f\_scores){

                        return1;

                    }

                    elseif (i.f\_scores<j.f\_scores){

                        return -1;

                    }

                    else{

                        return0;

                    }

                 }

                        }

                        );

                //cost from start

                source.g\_scores = 0;

                queue.add(source);

                booleanfound = false;

                while((!queue.isEmpty())&&(!found)){

                        //the node in having the lowest f\_score value

                        Nodecurrent = queue.poll();

                        explored.add(current);

                        //goal found

                        if(current.value.equals(goal.value)){

                                found = true;

                        }

                        //check every child of current node

                        for(Edgee:current.adjacencies){

                                Nodechild = e.target;

                                doublecost = e.cost;

                                doubletemp\_g\_scores = current.g\_scores + cost;

                                doubletemp\_f\_scores = temp\_g\_scores + child.h\_scores;

                                /\*if child node has been evaluated and

                                the newer f\_score is higher, skip\*/

                                if((explored.contains(child)) &&

                                        (temp\_f\_scores>= child.f\_scores)){

                                        continue;

                                }

                                /\*else if child node is not in queue or

                                newer f\_score is lower\*/

                                elseif((!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);

                                }

                        }

                }

        }

}

classNode{

        publicfinalStringvalue;

        publicdoubleg\_scores;

        publicfinaldoubleh\_scores;

        publicdoublef\_scores = 0;

        publicEdge[] adjacencies;

        publicNodeparent;

        publicNode(Stringval, doublehVal){

                value = val;

                h\_scores = hVal;

        }

        publicStringtoString(){

                return value;

        }

}

classEdge{

        publicfinaldoublecost;

        publicfinalNodetarget;

        publicEdge(NodetargetNode, doublecostVal){

                target = targetNode;

                cost = costVal;

        }

}