1-st Task.

import java.util.HashSet;  
import java.util.LinkedList;  
import java.util.Set;  
  
public class Graph {  
  
 private Set<Node> nodes = new HashSet<>();  
 public void addNode(Node nodeA) {  
 nodes.add(nodeA);  
 Node nodeA = new Node("A");  
 Node nodeB = new Node("B");  
 Node nodeC = new Node("C");  
 Node nodeD = new Node("D");  
 Node nodeE = new Node("E");  
 Node nodeF = new Node("F");  
 Node nodeG = new Node("G");  
 Node nodeH = new Node("H");  
 Node nodeI = new Node("I");  
  
 nodeA.addDestination(nodeB, 10);  
 nodeA.addDestination(nodeD, 2);  
 nodeA.addDestination(nodeE, 1/2);  
  
  
 nodeB.addDestination(nodeC, 3);  
  
 nodeC.addDestination(nodeB, 4);  
  
 nodeD.addDestination(nodeE, 0);  
 nodeD.addDestination(nodeD, 1);  
  
 nodeF.addDestination(nodeC, 8);  
 nodeF.addDestination(nodeB, 2);  
  
  
 nodeG.addDestination(nodeH, 2);  
  
 nodeH.addDestination(nodeG, 5);  
 nodeH.addDestination(nodeD, 9);  
 nodeH.addDestination(nodeF, 2);  
  
 nodeI.addDestination(nodeF, 7);  
 nodeI.addDestination(nodeH, 6);  
  
 Graph graph = new Graph();  
  
 graph.addNode(nodeA);  
 graph.addNode(nodeB);  
 graph.addNode(nodeC);  
 graph.addNode(nodeD);  
 graph.addNode(nodeE);  
 graph.addNode(nodeF);  
 graph.addNode(nodeG);  
 graph.addNode(nodeH);  
 graph.addNode(nodeI);  
  
 graph = Dijkstra.calculateShortestPathFromSource(graph, nodeA);  
 }  
  
  
 public static Graph calculateShortestPathFromSource(Graph graph, Node source) {  
 source.setDistance(0);  
  
 Set<Node> settledNodes = new HashSet<>();  
 Set<Node> unsettledNodes = new HashSet<>();  
  
 unsettledNodes.add(source);  
  
 while (unsettledNodes.size() != 0) {  
 Node currentNode = *getLowestDistanceNode*(unsettledNodes);  
 unsettledNodes.remove(currentNode);  
 for (Entry < Node, Integer> adjacencyPair:  
 currentNode.getAdjacentNodes().entrySet()) {  
 Node adjacentNode = adjacencyPair.getKey();  
 Integer edgeWeight = adjacencyPair.getValue();  
 if (!settledNodes.contains(adjacentNode)) {  
 calculateMinimumDistance(adjacentNode, edgeWeight, currentNode);  
 unsettledNodes.add(adjacentNode);  
 }  
 }  
 settledNodes.add(currentNode);  
 }  
 return graph;  
 }  
 private static Node getLowestDistanceNode(Set < Node > unsettledNodes) {  
 Node lowestDistanceNode = null;  
 int lowestDistance = Integer.*MAX\_VALUE*;  
 for (Node node: unsettledNodes) {  
 int nodeDistance = node.getDistance();  
 if (nodeDistance < lowestDistance) {  
 lowestDistance = nodeDistance;  
 lowestDistanceNode = node;  
 }  
 }  
 return lowestDistanceNode;  
 }  
 private static void CalculateMinimumDistance(Node evaluationNode,  
 Integer edgeWeigh, Node sourceNode) {  
 Integer sourceDistance = sourceNode.getDistance();  
 if (sourceDistance + edgeWeigh < evaluationNode.getDistance()) {  
 evaluationNode.setDistance(sourceDistance + edgeWeigh);  
 LinkedList<Node> shortestPath = new LinkedList<>(sourceNode.getShortestPath());  
 shortestPath.add(sourceNode);  
 evaluationNode.setShortestPath(shortestPath);  
 }  
 }

References: <https://www.youtube.com/watch?v=k1kLCB7AZbM>

1-task:

public class Main {

public static void main(String[] args)  
 {  
 double[][] Graph = new double[][]{  
 {0,10,0,2,0.5,0,0,0,0},  
 {0,0,3,0,0,0,0,0,0},  
 {0,4,0,0,0,0,0,0,0},  
 {0,0,0,4,0.1,0,6,0,0},  
 {0,8,0,0,0,7,0,1,0},  
 {0,2,8,0,0,0,0,1,0},  
 {0,0,0,0,0,0,0,2,0},  
 {0,0,0,9,0,2,5,0,0},  
 {0,0,0,0,0,7,0,6,0}  
 };  
 Main graphdistances = new Main();  
 graphdistances.dijkstra(Graph);  
 }  
 public int minimumDistance(double[] distance, Boolean[] visited){  
 double min = Double.*MAX\_VALUE*;  
 int ind =-1;  
 for(int i=0;i<distance.length;i++){  
 if(!visited[i] && distance[i]<=min) {  
 min = distance[i];  
 ind = i;  
  
 }  
 }  
 return ind;  
 }  
 public void dijkstra(double[][] graph){  
 double[] distance = new double[graph.length];  
  
 Boolean[] visited = new Boolean[distance.length];  
 for(int i=0;i<distance.length;i++){  
 distance[i]=Double.*MAX\_VALUE*;  
 visited[i] = false;  
 }  
  
 distance[0]=0;  
  
 for(int i=0;i<distance.length;i++){  
 int min = minimumDistance(distance,visited);  
 visited[min] = true;  
 for(int j=0;j<distance.length;j++){  
 if(!visited[j] && graph[min][j]!=0 && distance[min]!=Double.*MAX\_VALUE* && distance[min]+graph[min][j]<distance[j]){  
 distance[j] = distance[min] + graph[min][j];  
 }  
  
 }  
 }  
 printGraph(distance,distance.length);  
 }  
  
 private void printGraph(double[] distance, int j) {  
 System.*out*.println("The shortest path from node: ");  
 for(int i=0;i<j;i++){  
 System.*out*.println(i+"\t" + distance[i]);  
 }  
 }  
}

2-nd Task.

1. A vertex a represents **an endpoint of an edge**. An edge joins two vertices a, b and is represented by set of vertices it connects. Two edges of a graph are called **adjacent** if they share a common vertex. Two edges of a directed graph are called consecutive if the head of the first one is the tail of the second one. For example, in the picture our vertex is house and the place we go to
2. Every graph is a set of points referred to as vertices or nodes which are connected using lines called edges. The vertices represent entities in a graph. Edges, on the other hand, **express relationships between entities**. In the picture, it’s the path from our house to another place
3. Undirected, because in real we can go wherever we want, we can go to some place from our house , and come back
4. Weighted, because there a certain time to reach certain places, for example to go to the shopping mall from house takes 20 minutes
5. There are no self-loops in real world, it is purposeless to walk at one place.

Parallel edges is representation of different paths to one place (in real life)

1. First, we find out if there are pathes to that place (using boolean), if there is any path, we save it in array, then we create the reverse path. If there are no pathes, we go back
2. It is O(n). Because in this case, we get away from 5-miles path.