```
using System.Collections;
using System.Collections.Generic;


/*
This class should only provide API functionality so therefore no instance of Algorithm
class ever need to be created. All methods within the class are static
*/
public class Algorithm
{
    /*
    The algorithm takes a graph and a starting point as input and return a record of corresponding Node with both
    its shortest distane and ancestor
    */
    public static Record dijkstra(Graph graph, Node source) {
        // a constant for infinity value
        const int INF = int.MaxValue;
        // create a new Record
        Record record = new Record();
        // The first step is to initialise the record
        foreach(Node v in graph.getVertices()) {
            // initalise all distances of nodes to 0;
            record.shortestDistanceEstimate.Add(v, INF);
            record.parent.Add(v,null);
        }
        // set source's distance to 0
        record.shortestDistanceEstimate[source] = 0;
        // create a new queue
        PriorityQueue q = new PriorityQueue();
        // Add all vertices to the queue
        foreach(Node v in graph.getVertices()) {
            q.enqueue(v, record.shortestDistanceEstimate[v]);
        }
        while(!q.isEmpty()) {
            // current is the currently processing node
            Node current = q.dequeue().Item1;
            // loop through each of current's neighbor
            foreach(var v in current.getNeighbor()) {
                // assign the key to a variable neighbor
                Node neighbor = v.Key;
                int currentEdgeWeight = v.Value;
                // perform relax operation on the each edge outgoing from current
                if(record.shortestDistanceEstimate[current] + currentEdgeWeight < record.shortestDistanceEstimate[neighbor]) {
                    // update shortestDistanceEstimate
                    record.shortestDistanceEstimate[neighbor] = record.shortestDistanceEstimate[current] + currentEdgeWeight;
                    // update the parent
                    record.parent[neighbor] = current;
                    // update the priority of neighbor in the priority queue
                    q.updateKey(neighbor, record.shortestDistanceEstimate[current] + currentEdgeWeight);
                }
            }
        }
        return record;
    }

```

```
    public static List<Node> findShortestPath(Graph graph, Node source, Node destination) {
        // call dijkstra's algorithm and save the result in a variable
        Record result = dijkstra(graph, source);
        // list to return the final path
        List<Node> shortestPath = new List<Node>();
        // starting from destination node and work backward
        Node current = destination;
        while(current != null) {
            // add to the list backward from end to beginning
            shortestPath.Insert(0,current);
            // traverse from destination to source iteratively
            current = result.parent[current];
        }
        return shortestPath;
    }



}

```