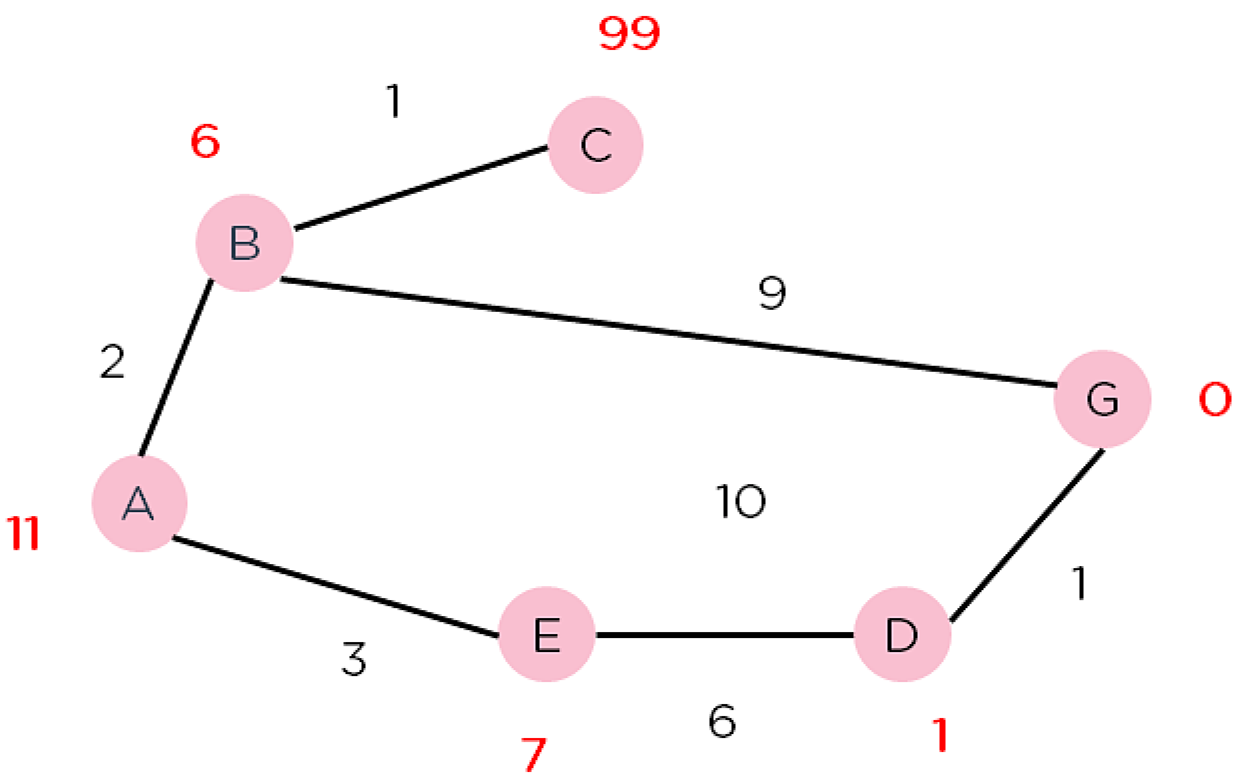
```java
// A* Search Algorithm

// let openList equal empty list of nodes
// let closedList equal empty list of nodes
// put startNode on the openList (leave it's f at zero)
// while openList is not empty
//     let currentNode equal the node with the least f value
//     remove currentNode from the openList
//     add currentNode to the closedList
//     if currentNode is the goal
//         You've found the exit!
//     let children of the currentNode equal the adjacent nodes
//     for each child in the children
//         if child is in the closedList
//             continue to beginning of for loop
//         child.g = currentNode.g + weight b/w child and current
//         child.h = weight from child to end
//         child.f = child.g + child.h
//         if child.position is in the openList's nodes positions
//             if child.g is higher than the openList node's g
//                 continue to beginning of for loop
//         add the child to the openList


import java.io.*;
import java.util.*;


class Graph {

   static class Node {
       String vertex;
       Integer weight;

       public Node(String vertex, Integer weight) {
           this.vertex = vertex;
           this.weight = weight;
       }
   }

   private HashMap<String, ArrayList<Node>> adj;

   private HashMap<String, Integer> H;

   Graph(HashMap<String, ArrayList<Node>> adjac_lis) {
```

```java
        adj = adjac_lis;

        H = new HashMap<String, Integer>();
        H.put("A", 11);
        H.put("B", 6);
        H.put("C", 99);
        H.put("D", 1);
        H.put("E", 7);
        H.put("G", 0);
    }

    ArrayList<Node> get_neighbors(String vertex) {
        return adj.get(vertex);
    }

    // heuristic function with distances from the current node to the goal node
    int h(String v) {
        return H.get(v);
    }

    void a_star_algorithm(String s, String d) {
        // open_list is a list of nodes which have been visited, but who's neighbors
        // haven't all been inspected, starts off with the start node
        // closed_list is a list of nodes which have been visited
        // and who's neighbors have been inspected
        HashSet<String> open_list = new HashSet<String>();
        open_list.add(s);
        HashSet<String> closed_list = new HashSet<String>();

        // g contains current distances from start_node to all other nodes
        // the default value (if it's not found in the map) is +infinity
        HashMap<String, Integer> g = new HashMap<String, Integer>();
        g.put(s, 0);

        // parents contains an adjacency map of all nodes
        HashMap<String, String> parent = new HashMap<String, String>();
        parent.put(s, s);

        while (open_list.size() > 0) {
            String n = null;

            // find a node with the lowest value of f() - evaluation function
            for (String v : open_list) {
                if ( n == null || g.get(v) + h(v) < g.get(n) + h(n))
                    n = v;
```

```
            }

            if (n == null) {
                System.out.println("Path does not exist!");
                return;
            }

            // if the current node is the stop_node
            // then we begin reconstructin the path from it to the start_node
            if (n.equals(d)) {
                ArrayList<String> reconst_path = new ArrayList<String>();

                while (parent.get(n) != n) {
                    reconst_path.add(n);
                    n = parent.get(n);
                }

                reconst_path.add(n);
                Collections.reverse(reconst_path);

                System.out.println("Path found: " + reconst_path);
                return;
            }

            // for all neighbors of the current node do
            for (Node v : get_neighbors(n)) {
                // if the current node isn't in both open_list and closed_list
                // add it to open_list and note n as it's parent
                if (!closed_list.contains(v.vertex) && !open_list.contains(v.vertex))
{
                    open_list.add(v.vertex);
                    parent.put(v.vertex, n);
                    g.put(v.vertex, g.get(n) + v.weight);
                }
                // otherwise, check if it's quicker to first visit n, then m
                // # and if it is, update parent data and g data
                // # and if the node was in the closed_list, move it to open_list
                else {
                    if (g.get(v.vertex) > g.get(n) + v.weight) {
                        g.put(v.vertex, g.get(n) + v.weight);
                        parent.put(v.vertex, n);

                        if (closed_list.contains(v.vertex)) {
                            closed_list.remove(v.vertex);
                            open_list.add(v.vertex);
```

```java
                        }
                    }
                }
            }

            // remove n from the open_list, and add it to closed_list
            // # because all of his neighbors were inspected
            open_list.remove(n);
            closed_list.add(n);
        }

    }


    public static void main(String args[]) {
        HashMap<String, ArrayList<Node>> adjac_lis = new HashMap<String,
ArrayList<Node>>();

        adjac_lis.put(
            "A",
            new ArrayList<Node>(Arrays.asList(
                new Node("B", 2),
                new Node("E", 3)
            ))
        );

        adjac_lis.put(
            "B",
            new ArrayList<Node>(Arrays.asList(
                new Node("C", 1),
                new Node("G", 9)
            ))
        );

        adjac_lis.put(
            "C",
            null
        );

        adjac_lis.put(
            "D",
            new ArrayList<Node>(Arrays.asList(
                new Node("G", 1)
            ))
        );
```

```
        adjac_lis.put(
            "E",
            new ArrayList<Node>(Arrays.asList(
                new Node("D", 6)
            ))
        );

        Graph graph = new Graph(adjac_lis);
        graph.a_star_algorithm("A", "G");
    }
}
```



<u>OUTPUT :-</u>

```
Path found: [A, E, D, G]
```