```
public class BreadthFirstPaths {
    private static final int INFINITY = Integer.MAX_VALUE;// /4
    private boolean[] marked;  // marked[v] = is there an s-v path
    private int[] edgeTo;      // edgeTo[v] = previous edge on shortest s-v path
    private int[] distTo;      // distTo[v] = number of edges shortest s-v path

    /**
     * Computes the shortest path between the source vertex {@code s}
     * and every other vertex in the graph {@code G}.
     * @param G the graph
     * @param s the source vertex
     * @throws IllegalArgumentException unless {@code 0 <= s < V}
     */
    public BreadthFirstPaths(Graph G, int s) {
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        edgeTo = new int[G.V()];
        validateVertex(s);
        bfs(G, s);

        assert check(G, s);
    }

    /**
     * Computes the shortest path between any one of the source vertices in {@code
sources}
     * and every other vertex in graph {@code G}.
     * @param G the graph
     * @param sources the source vertices
     * @throws IllegalArgumentException unless {@code 0 <= s < V} for each vertex
     *         {@code s} in {@code sources}
     */
    public BreadthFirstPaths(Graph G, Iterable<Integer> sources) {
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        edgeTo = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
            distTo[v] = INFINITY;
        validateVertices(sources);
        bfs(G, sources);
    }


    // breadth-first search from a single source
    private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        for (int v = 0; v < G.V(); v++)
            distTo[v] = INFINITY;
        distTo[s] = 0;
        marked[s] = true;
        q.enqueue(s);

        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
                }
            }
        }
    }
    // breadth-first search from multiple sources
    private void bfs(Graph G, Iterable<Integer> sources) {
        Queue<Integer> q = new Queue<Integer>();
        for (int s : sources) {
            marked[s] = true;
            distTo[s] = 0;
            q.enqueue(s);
        }
        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
                }
            }
        }
    }

    /**
     * Is there a path between the source vertex {@code s} (or sources) and vertex
{@code v}?
     * @param v the vertex
     * @return {@code true} if there is a path, and {@code false} otherwise
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public boolean hasPathTo(int v) {
        validateVertex(v);
        return marked[v];
    }


    /**
     * Returns the number of edges in a shortest path between the source vertex
{@code s}
     * (or sources) and vertex {@code v}?
     * @param v the vertex
     * @return the number of edges in a shortest path
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public int distTo(int v) {
        validateVertex(v);
        return distTo[v];
    }



    // check optimality conditions for single source
    private boolean check(Graph G, int s) {

        // check that the distance of s = 0
        if (distTo[s] != 0) {
            System.out.println("Distancia da fonte " + s + " para si mesma = " +
distTo[s]);
            return false;
        }

        // check that for each edge v-w dist[w] <= dist[v] + 1
        // provided v is reachable from s
        for (int v = 0; v < G.V(); v++) {
            for (int w : G.adj(v)) {
                if (hasPathTo(v) != hasPathTo(w)) {
                    System.out.println("Aresta " + v + "-" + w);
                    System.out.println("Exite caminho para(" + v + ") = " +
hasPathTo(v));
                        System.out.println("Exite caminho para(" + w + ") = " +
hasPathTo(w));
                        return false;
                    }
                    if (hasPathTo(v) && (distTo[w] > distTo[v] + 1)) {
                        System.out.println("Aresta " + v + "-" + w);
                        System.out.println("Distancia[" + v + "] = " + distTo[v]);
                        System.out.println("Distancia[" + w + "] = " + distTo[w]);
                        return false;
                    }
                }
            }

            // check that v = edgeTo[w] satisfies distTo[w] = distTo[v] + 1
            // provided v is reachable from s
            for (int w = 0; w < G.V(); w++) {
                if (!hasPathTo(w) || w == s) continue;
                int v = edgeTo[w];
                if (distTo[w] != distTo[v] + 1) {
                    System.out.println("aresta de menor caminho " + v + "-" + w);
                    System.out.println("distancia[" + v + "] = " + distTo[v]);
                    System.out.println("distancia[" + w + "] = " + distTo[w]);
                    return false;
                }
            }

            return true;
        }

        // throw an IllegalArgumentException unless {@code 0 <= v < V}
        private void validateVertex(int v) {
            int V = marked.length;
            if (v < 0 || v >= V)
                throw new IllegalArgumentException("vertice " + v + " nao esta entre 0
e " + (V-1));
        }

        // throw an IllegalArgumentException unless {@code 0 <= v < V}
        private void validateVertices(Iterable<Integer> vertices) {
            if (vertices == null) {
                throw new IllegalArgumentException("parametro nulo");
            }
            int V = marked.length;
            for (int v : vertices) {
                if (v < 0 || v >= V) {
                    throw new IllegalArgumentException("vertice " + v + " nao esta
entre 0 e " + (V-1));
                }
            }
        }

        public void printPath(int s,int v){
              if(s == v){
                   System.out.print(s);
              }
              else{
                   if(!marked[v]) return;
                   else{
                        printPath(s,edgeTo[v]);
                        System.out.print("-"+v);
                   }
              }
        }

        /**
```

```
     * Unit tests the {@code BreadthFirstPaths} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        Graph G = new Graph(args[0]);

        int s = Integer.parseInt(args[1]);
        BreadthFirstPaths bfs = new BreadthFirstPaths(G, s);

        for (int v = 0; v < G.V(); v++) {
            if (bfs.hasPathTo(v)) {
                System.out.print( s+" ate "+ v +" (" +bfs.distTo(v)+") - ");
                bfs.printPath(s,v);
                System.out.println();
            }

            else {
                System.out.print(s+ " e "+v+" nao estao ligados\n");
            }

        }
    }
}
```