**1. How to use DFS to check undirected graph connected or not ?**

To use Depth-First Search (DFS) to check if an undirected graph is connected or not, you can start DFS from any arbitrary node and traverse all nodes reachable from that node. If during the DFS traversal, you visit all nodes in the graph, then the graph is connected; otherwise, it's not.

Here's a step-by-step approach to implement this:

Implement the DFS algorithm: DFS is a recursive algorithm that explores as far as possible along each branch before backtracking. You'll maintain a visited set or array to keep track of visited nodes to avoid visiting the same node multiple times.

Start DFS from an arbitrary node: Choose any node in the graph as the starting point for DFS traversal.

Traverse the graph: During DFS traversal, visit all nodes reachable from the starting node.

Check visited nodes: After DFS traversal, if the number of visited nodes is equal to the total number of nodes in the graph, then the graph is connected; otherwise, it's not.

Here's a Java code example implementing this approach:

import java.util.\*;

public class ConnectedGraphChecker {

public static boolean isConnected(Map<Integer, List<Integer>> graph) {

Set<Integer> visited = new HashSet<>();

int startNode = graph.keySet().iterator().next(); // Start from an arbitrary node

dfs(graph, startNode, visited);

return visited.size() == graph.size();

}

private static void dfs(Map<Integer, List<Integer>> graph, int node, Set<Integer> visited) {

visited.add(node); // Mark the current node as visited

for (int neighbor : graph.getOrDefault(node, new ArrayList<>())) {

if (!visited.contains(neighbor)) { // Visit unvisited neighbors

dfs(graph, neighbor, visited);

}

}

}

public static void main(String[] args) {

Map<Integer, List<Integer>> graph = new HashMap<>();

graph.put(0, Arrays.asList(1, 2));

graph.put(1, Arrays.asList(0, 3));

graph.put(2, Arrays.asList(0, 3));

graph.put(3, Arrays.asList(1, 2));

boolean connected = isConnected(graph);

System.out.println("Is the graph connected? " + connected);

}

}

In this example, the isConnected method takes an adjacency list representation of an undirected graph as input and returns true if the graph is connected, and false otherwise. The dfs method performs depth-first search traversal of the graph. Finally, in the

main method, we create a sample graph and check if it's connected using the isConnected method.

**2. How to use DFS to check directed graph connected (strongly connected) or not ?**

To check if a directed graph is strongly connected (i.e., every vertex is reachable from every other vertex), you can use Depth-First Search (DFS) with a slight modification.

Here's how you can do it:

Perform DFS from a starting vertex: Start DFS from any vertex in the graph and explore as far as possible along each branch.

Track visited vertices: Maintain a set or array to keep track of visited vertices to avoid visiting the same vertex multiple times.

Check if all vertices are visited: After DFS traversal, if the visited set/array contains all vertices in the graph, then the graph is strongly connected; otherwise, it's not.

If the graph is not strongly connected, try DFS from other vertices: If the initial DFS traversal doesn't cover all vertices, try performing DFS from other unvisited vertices until all vertices are visited or until it's evident that the graph is not strongly connected.

Here's a Java code example implementing this approach:

import java.util.\*;

public class StronglyConnectedGraphChecker {

public static boolean isStronglyConnected(Map<Integer, List<Integer>> graph) {

for (int startNode : graph.keySet()) {

Set<Integer> visited = new HashSet<>();

dfs(graph, startNode, visited);

if (visited.size() != graph.size()) {

return false; // Not all vertices are reachable from the startNode

}

}

return true; // All vertices are reachable from every other vertex

}

private static void dfs(Map<Integer, List<Integer>> graph, int node, Set<Integer> visited) {

visited.add(node); // Mark the current node as visited

for (int neighbor : graph.getOrDefault(node, new ArrayList<>())) {

if (!visited.contains(neighbor)) { // Visit unvisited neighbors

dfs(graph, neighbor, visited);

}

}

}

public static void main(String[] args) {

Map<Integer, List<Integer>> graph = new HashMap<>();

graph.put(0, Arrays.asList(1));

graph.put(1, Arrays.asList(2));

graph.put(2, Arrays.asList(0, 3));

graph.put(3, Arrays.asList(4));

graph.put(4, new ArrayList<>());

boolean stronglyConnected = isStronglyConnected(graph);

System.out.println("Is the graph strongly connected? " + stronglyConnected);

}

}

In this example, the isStronglyConnected method takes an adjacency list representation of a directed graph as input and returns true if the graph is strongly connected (i.e., every vertex is reachable from every other vertex), and false otherwise. The

dfs method performs depth-first search traversal of the graph. In the main method, we create a sample directed graph and check if it's strongly connected using the isStronglyConnected method.

**3. What's the difference between DFS check if directed graph connected and DFS check if undirected graph connected ?**

The primary difference between using Depth-First Search (DFS) to check if a directed graph is connected versus checking if an undirected graph is connected lies in the way we interpret the results of the DFS traversal.

Directed Graph (Strongly Connected):

In a directed graph, to determine if it's strongly connected (every vertex is reachable from every other vertex), we perform DFS from every vertex and ensure that all vertices are visited during each traversal. If all vertices are visited from every vertex, the graph is strongly connected.

In other words, in a strongly connected directed graph, there should be a directed path from any vertex to any other vertex. Therefore, we need to perform DFS from each vertex to verify this property.

Undirected Graph (Connected):

In an undirected graph, to determine if it's connected (there exists a path between every pair of vertices), we typically perform DFS from a single starting vertex and ensure that all vertices are visited during the traversal. If all vertices are visited, the graph is connected.

In an undirected graph, the presence of an edge between vertices implies bidirectional connectivity. Therefore, starting DFS from any vertex should traverse the entire connected component if the graph is connected.

In summary, the key difference lies in the interpretation of the DFS traversal results:

For directed graphs, we need to perform DFS from every vertex to ensure strong connectivity.

For undirected graphs, starting DFS from any vertex and traversing all vertices suffices to determine connectivity.

<https://www.ics.uci.edu/~thornton/ics46/Notes/GraphConnectedness/>