1.Consider the following adjacency matrix representing a graph with 4 nodes (0, 1, 2, 3):

0 1 0 1

1 0 1 0

0 1 0 1

1 0 1 0

Which of the following is the correct graph representation?

a) A graph with 2 edges

b) A graph with 4 edges

c) A graph with 5 edges

d) A graph with 6 edges

Answer:

b) A graph with 4 edges

Explanation:

The adjacency matrix represents a graph with 4 nodes, where the edges are: (0,1), (1,2), (2,3), and (0,3).

The matrix is symmetric, which indicates an undirected graph.

2.Consider the following directed graph represented by its adjacency list:

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

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

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

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

What will be the output of the following DFS traversal starting from node 0?

dfs(0, visited, graph);

a) 0 1 2 3

b) 0 2 1 3

c) 0 1 3 2

d) 1 0 2 3

Answer:

a) 0 1 3 2

Explanation:

In DFS, the algorithm explores a path as deeply as possible before backtracking.

Starting from node 0, it will visit node 1, then 3 (the neighbor of node 1), then backtrack to node 0 and visit node 2 and then finally visit node 3 again.

3.Consider the following Java code snippet to detect a cycle in an undirected graph:

public boolean hasCycle(int node, int parent, Set<Integer> visited, Map<Integer, List<Integer>> graph) {

visited.add(node);

for (int neighbor : graph.get(node)) {

if (!visited.contains(neighbor)) {

if (hasCycle(neighbor, node, visited, graph)) {

return true;

}

} else if (neighbor != parent) {

return true;

}

return false;

}

In the context of this code, which of the following statements is TRUE?

a) The code works for directed graphs only.

b) The code works for undirected graphs and detects cycles.

c) The code cannot detect cycles in graphs with multiple components.

d) The code will throw an error because it doesn't handle visited nodes correctly.

Answer:

b) The code works for undirected graphs and detects cycles.

Explanation:

This code performs a Depth-First Search (DFS) and checks for cycles in an undirected graph.

It avoids a false positive by ensuring that if a node is revisited, it must not be the parent of the current node.

This approach works for undirected graphs, where revisiting a node that's not the immediate parent indicates a cycle.

4. Let G be a simple graph with 20 vertices and 8 components. If we delete a vertex in G, then number of components in G should lie between \_\_\_\_.

A.8 and 20

B.8 and 19

C.7 and 19

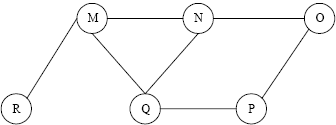
D.7 and 20

Answer: C

**Explanation :**

Case 1: If the vertex we are deleting from G is an isolated vertex, which is a component by itself, then number of components in G becomes 7.  
Case 2: If G is a start Graph, then by deleting the cut vertex of G, we get 19 components.  
Hence, number of components in G should lie between 7 and 19.

6. The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is 



A.MNOPQR

B.NQMPOR

C.QMNPRO

D.QMNPOR

Answer: C

**Explanation :**

Option (A) is MNOPQR. It cannot be a BFS as the traversal starts with M, but O is visited before N and Q. In BFS all adjacent must be visited before adjacent of adjacent. Option (B) is NQMPOR. It also cannot be BFS, because here, P is visited before O. (C) and (D) match up to QMNP. We see that M was added to the queue before N and P (because M comes before NP in QMNP). Because R is M's neighbor, it gets added to the queue before the neighbor of N and P (which is O). Thus, R is visited before O.

Hence (C) is the correct answer.

7. What is the output of this adjacency list code?

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

graph.put("A", Arrays.asList("B", "C"));

graph.put("B", Arrays.asList("A", "D"));

graph.put("C", Arrays.asList("A"));

graph.put("D", Arrays.asList("B"));

System.out.println(graph.get("B"));

A. [A, C]

B. [B, D]

C. [A, D]

D. [D, A]

Correct Answer: C. [A, D]

Explanation:

The adjacency list for node "B" is explicitly defined as ["A", "D"]. So graph.get("B") returns [A, D].

8. What traversal algorithm is implemented here?

public void traverse(String start) {

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

Queue<String> queue = new LinkedList<>();

queue.add(start);

visited.add(start);

while (!queue.isEmpty()) {

String node = queue.poll();

System.out.print(node + " ");

for (String neighbor : graph.get(node)) {

if (!visited.contains(neighbor)) {

queue.add(neighbor);

visited.add(neighbor);

}

}

}

}

A. DFS

B. BFS

C. Topological Sort

D. Dijkstra

Correct Answer: B. BFS

Explanation:

This uses a queue to explore neighbors level-by-level, which is the defining trait of Breadth-First Search (BFS).

9. What does this method do?

public boolean hasPath(String src, String dest, Set<String> visited) {

if (src.equals(dest)) return true;

visited.add(src);

for (String neighbor : graph.get(src)) {

if (!visited.contains(neighbor)) {

if (hasPath(neighbor, dest, visited)) return true;

}

}

return false;

}

A. Finds shortest path

B. Detects a cycle

C. Checks if a path exists using DFS

D. Prints the graph

Correct Answer: C. Checks if a path exists using DFS

Explanation:

This is a recursive DFS that returns true as soon as it finds the destination node. It's not calculating the shortest path or detecting cycles.

10.Which scenario causes a cycle in an undirected graph?

A. All nodes are visited exactly once

B. A node connects back to a visited node that is not its parent

C. Graph has a node with degree 1

D. Graph has a node with no outgoing edge

Correct Answer:

B. A node connects back to a visited node that is not its parent

Explanation:

In undirected graphs, a cycle exists when a node points to a previously visited node that is not its immediate parent in DFS.

11. What is the time complexity of BFS in an adjacency list representation?

A. O(V²)

B. O(E + V)

C. O(E log V)

D. O(V log E)

Correct Answer: B. O(E + V)

Explanation:

Each node (V) is visited once, and all edges (E) are explored once. Hence, time complexity = O(V + E).

12.What does this method count?

public int countComponents() {

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

int count = 0;

for (String node : graph.keySet()) {

if (!visited.contains(node)) {

dfs(node, visited);

count++;

}

}

return count;

}

A. Number of cycles

B. Number of leaf nodes

C. Number of connected components

D. Number of dead ends

Correct Answer: C. Number of connected components

Explanation:

Each unvisited node starts a DFS, which means it's part of a new disconnected component.

13. What is a dead-end node in an undirected graph?

A. Node with no neighbors

B. Node with only one neighbor

C. Node in a cycle

D. Node with maximum degree

Correct Answer: B. Node with only one neighbor

Explanation:

A dead-end has only one connection (degree = 1), making it a leaf in an undirected graph.

14. What will this method return for a disconnected graph?

public boolean hasCycle(String node, String parent, Set<String> visited) {

visited.add(node);

for (String neighbor : graph.get(node)) {

if (!visited.contains(neighbor)) {

if (hasCycle(neighbor, node, visited)) return true;

} else if (!neighbor.equals(parent)) {

return true;

}

}

return false;

}

A. May return false even if there is a cycle in another component

B. Always detects a cycle

C. Only works for directed graphs

D. Only detects self-loops

Correct Answer: A. May return false even if there is a cycle in another component

Explanation:

This checks for a cycle from a single connected component. If the graph is disconnected, other components aren’t explored.

15. The following code snippet is the function to insert a string in a trie. Find the missing line.

**private** void insert(String str){

TrieNode node = root;

**for** (int i = 0; i < length; i++){

int index = key.charAt(i) - 'a';

**if** (node.children[index] == **null**)

node.children[index] = **new** TrieNode();

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

node.isEndOfWord = **true**;

}

A. node = node.children[index];

B. node = node.children[str.charAt(i + 1)];

C. node = node.children[index++];

D. node = node.children[index++];

Answer: A

**Explanation :**

After checking if the current node has a child at the given index, we need to move to the next node in the trie.

The missing line moves the current target node to next node after checking the current node.

node = node->children[index]; / node = node.children[index];

Hence (A) is the correct option.