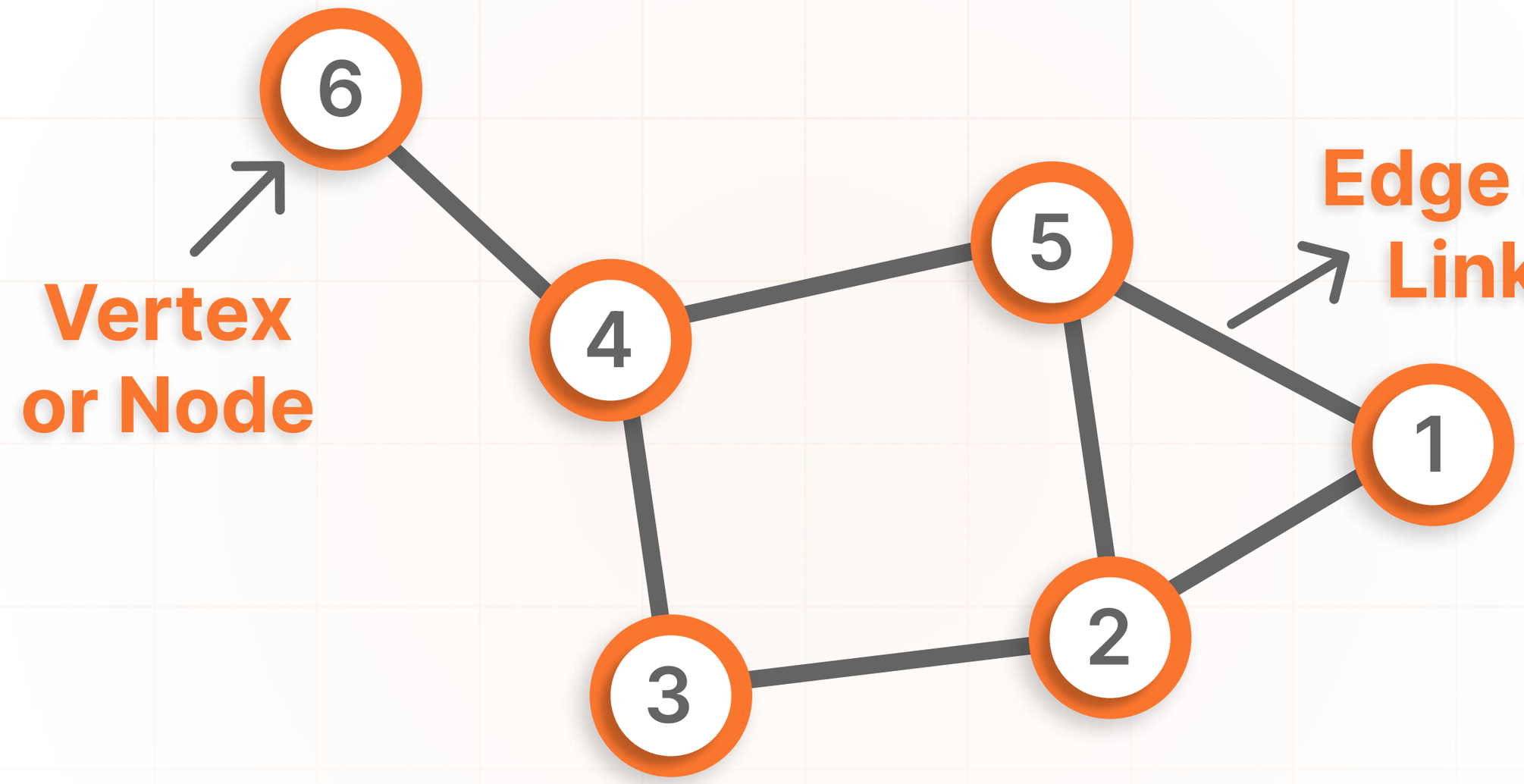
BOSSCODER ACADEMY

GRAPHS

REVISION GUIDE

Practical Questions



Disclaimer

Everyone learns uniquely.

What matters is developing the problem solving ability to solve new problems.

This Doc will help you with the same.

Introduction to Graphs

Graphs are a fundamental data structure used to model pairwise relationships between objects. They consist of vertices (nodes) and edges (connections). Graphs can represent various real-world problems, including social networks, transportation systems, and web page linking.

Concepts in Graphs

1. GRAPH TERMINOLOGY

- **Vertex (Node):** The fundamental unit of a graph.
- **Edge:** A connection between two vertices.
- **Adjacency:** Two vertices are adjacent if they are connected by an edge.
- **Degree:** The number of edges connected to a vertex.
- **Path:** A sequence of edges that connects two vertices.
- **Cycle:** A path that starts and ends at the same vertex.
- **Connected Graph:** A graph in which there is a path between any two vertices.
- **Disconnected Graph:** A graph in which some vertices are not connected by paths.
- **Directed Graph (Digraph):** A graph in which edges have directions.
- **Undirected Graph:** A graph in which edges have no direction.

2. GRAPH REPRESENTATIONS

- **Adjacency Matrix:** A 2D array where the element at (i, j) is true if there is an edge from vertex i to vertex j.
- **Adjacency List: An array of lists.** The index represents the vertex, and the list at each index contains the adjacent vertices.

3. GRAPH TRAVERSALS

- **Depth-First Search (DFS):** Explores as far as possible along each branch before backtracking.
- **Breadth-First Search (BFS):** Explores all neighbors at the present depth before moving on to nodes at the next depth level.

4. SHORTEST PATH ALGORITHMS

- **Dijkstra's Algorithm:** Finds the shortest paths from a single source vertex to all other vertices in a graph with non-negative edge weights.
- **Bellman-Ford Algorithm:** Finds the shortest paths from a single source vertex to all other vertices in a graph, even if the graph has negative weight edges.
- **Floyd-Warshall Algorithm:** Finds shortest paths between all pairs of vertices.

5. MINIMUM SPANNING TREE (MST)

- **Kruskal's Algorithm:** Finds the MST by adding edges in increasing order of weight, ensuring no cycles are formed.
- **Prim's Algorithm:** Finds the MST by starting from a single vertex and repeatedly adding the smallest edge that connects a vertex in the MST to a vertex outside it.

6. TOPOLOGICAL SORTING

- Ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge uv, vertex u comes before vertex v.

7. STRONGLY CONNECTED COMPONENTS (SCC)

- A maximal subgraph of a directed graph where there is a path between any two vertices.

Practical Implication-Based Practice Questions

TOPIC-1

Detecting a Cycle in a Directed Graph

Q- Implement an algorithm to detect if a directed graph has a cycle.

Approach:

Use Depth-First Search (DFS) and maintain a recursion stack to detect back edges.

Code:

Python

```
def hasCycle(graph):
    def dfs(node):
        if node in recursion_stack:
            return True
        if node in visited:
            return False
    visited.add(node)
        recursion_stack.add(node)
```

```
            for neighbor in graph[node]:
                if dfs(neighbor):
                    return True
            recursion_stack.remove(node)
            return False

    visited = set()
    recursion_stack = set()
    for node in graph:
        if dfs(node):
            return True
    return False
```

TOPIC-2

Detecting a Cycle in an Undirected Graph

Q- Implement an algorithm to detect if an undirected graph has a cycle.

Approach:

Use DFS and keep track of the parent node to avoid counting the reverse edge.

Code:

```python
def hasCycle(graph):
    def dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif parent != neighbor:
```

```
                return True
        return False


visited = set()
for node in graph:
    if node not in visited:
        if dfs(node, -1):
            return True
return False
```

TOPIC-3

Finding the Shortest Path in an Unweighted Graph

Q- Use BFS to find the shortest path between two vertices in an unweighted graph.

Approach:

Use Breadth-First Search (BFS) to find the shortest path.

Code:

```python
from collections import deque

def shortestPath(graph, start, end):
    queue = deque([(start, [start])])
    visited = set([start])

    while queue:
        current, path = queue.popleft()
```

```
        if current == end:
            return path
        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path +
                [neighbor]))
    return None
```

TOPIC-4

Finding the Shortest Path in a Weighted Graph

Q- Use Dijkstra's algorithm to find the shortest path between two vertices in a weighted graph.

Approach:

Use Dijkstra's algorithm to find the shortest path.

Code:

```python
import heapq

def dijkstra(graph, start):
    min_heap = [(0, start)]
    distances = {vertex: float('infinity') for vertex
    in graph}
    distances[start] = 0

    while min_heap:
```

```
        current_distance, current_vertex =
        heapq.heappop(min_heap)

        if current_distance >
distances[current_vertex]:
            continue

        for neighbor, weight in
graph[current_vertex]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(min_heap,
(distance, neighbor))
    return distances
```

TOPIC-5

Clone a Graph

Q- Implement an algorithm to clone a graph.

Approach:

Use BFS to clone a graph.

Code:

```python
def cloneGraph(node):
    if not node:
        return node
    visited = {}
    queue = deque([node])
    visited[node] = Node(node.val, [])

    while queue:
        n = queue.popleft()
```

```
    for neighbor in n.neighbors:
                if neighbor not in visited:
                    visited[neighbor] =
Node(neighbor.val, [])
                    queue.append(neighbor)

visited[n].neighbors.append(visited[neighbor])
    return visited[node]
```

TOPIC-6

Topological Sorting of a Directed Acyclic Graph (DAG)

Q- Implement topological sorting for a DAG using DFS.

Approach:

Use DFS and maintain a stack to get the topological order.

Code:

```python
def topologicalSort(graph):
    def dfs(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor)
        stack.append(node)

    visited = set()
```

```
    stack = []
    for node in graph:
        if node not in visited:
            dfs(node)
    return stack[::-1]
```

TOPIC-7

Finding All Strongly Connected Components

Q- Implement Kosaraju's algorithm to find all SCCs in a directed graph.

Approach:

Use Kosaraju's algorithm.

Code:

```python
def kosaraju(graph):
    def dfs(node, visited, stack):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor, visited, stack)
        stack.append(node)

    def reverseGraph(graph):
        reversed_graph = {node: [] for node in graph}
        for node in graph:
            for neighbor in graph[node]:
```

```
                reversed_graph[neighbor].append(node)
        return reversed_graph


def fillOrder(graph, visited, stack):
        for node in graph:
                if node not in visited:
                        dfs(node, visited, stack)
def printSCCs(reversed_graph, stack):
    visited = set()
    while stack:
        node = stack.pop()
        if node not in visited:
            component = []
            dfs(node, visited, component)
            print(component)
            stack = []
            visited = set()
            fillOrder(graph, visited, stack)
reversed_graph = reverseGraph(graph)
printSCCs(reversed_graph, stack)
```

TOPIC-8

Finding the Minimum Spanning Tree Using Kruskal's Algorithm

Q- Implement Kruskal's algorithm to find the MST of a graph.

Approach:

Use the union-find data structure to find the MST.

Code:

```python
def kruskal(graph):
    def find(parent, i):
        if parent[i] == i:
            return i
        return find(parent, parent[i])

    def union(parent, rank, x, y):
        root_x = find(parent, x)
        root_y = find(parent, y)
        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
```

```
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_y] = root_x
            rank[root_x] += 1

    result = []
    i, e = 0, 0
    graph = sorted(graph, key=lambda item:
item[2])
    parent = []
    rank = []

    for node in range(len(graph)):
        parent.append(node)
        rank.append(0)

    while e < len(graph) - 1:
        u, v, w = graph[i]
        i += 1
        x = find(parent, u)
```

```
            y = find(parent, v)
            if x != y:
                e += 1
                result.append([u, v, w])
                union(parent, rank, x, y)

    for u, v, weight in result:
        print(f"{u} -- {v} == {weight}")
```

TOPIC-9

Finding the Minimum Spanning Tree Using Prim's Algorithm

Q- Implement Prim's algorithm to find the MST of a graph.

Approach:

Use a priority queue to find the MST.

Code:

```python
import heapq

def prim(graph, start):
    min_heap = [(0, start)]
    mst = []
    visited = set()

    while min_heap:
        weight, node = heapq.heappop(min_heap)
        if node in visited:
            continue
```

```
    visited.add(node)
            mst.append((weight, node))
            for neighbor, cost in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(min_heap, (cost,
neighbor))

    return mst
```

TOPIC-10

Check Bipartiteness of a Graph

Q- Implement an algorithm to check if a graph is bipartite.

Approach:

Use BFS to check if a graph is bipartite.

Code:

```python
def isBipartite(graph):
    color = {}

    for node in graph:
        if node not in color:
            queue = deque([node])
            color[node] = 0
            while queue:
                node = queue.popleft()
                for neighbor in graph[node]:
                    if neighbor not in color:
```

```
                color[neighbor] = color[node] ^ 1
                  queue.append(neighbor)
            elif color[neighbor] == color[node]:
    return False
return True
```

TOPIC-11

Network Delay Time

Q- Calculate how long it will take for all nodes to receive a signal sent from a given node in a weighted graph.

Approach:

Use Dijkstra's algorithm to find the network delay time.

Code:

```python
def networkDelayTime(times, N, K):
    graph = collections.defaultdict(list)
    for u, v, w in times:
        graph[u].append((v, w))

    min_heap = [(0, K)]
    visited = {}

    while min_heap:
        time, node = heapq.heappop(min_heap)
        if node in visited:
```

```
            continue
        visited[node] = time
            for neighbor, t in graph[node]:
                if neighbor not in visited:
        heapq.heappush(min_heap, (time + t, neighbor))

    if len(visited) == N:
        return max(visited.values())
    return -1
```

TOPIC-12

Course Schedule

Q- Determine if it is possible to finish all courses given the prerequisites as a graph.

Approach:

Use topological sorting to determine if all courses can be finished.

Code:

```python
def canFinish(numCourses, prerequisites):
    graph = collections.defaultdict(list)
    for u, v in prerequisites:
        graph[u].append(v)

    visited = [0] * numCourses

    def dfs(node):
        if visited[node] == -1:
            return False
        if visited[node] == 1:
```

```
            return True
        visited[node] = -1
        for neighbor in graph[node]:
            if not dfs(neighbor):
                return False
        visited[node] = 1
        return True

    for course in range(numCourses):
        if not dfs(course):
            return False
    return True
```

TOPIC-13

Number of Connected Components in an Undirected Graph

Q- Implement an algorithm to count the number of connected components in an undirected graph.

Approach:

Use DFS to count the number of connected components.

Code:

```python
def countComponents(n, edges):
    graph = collections.defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = set()

    def dfs(node):
```

```
            stack = [node]
            while stack:
                n = stack.pop()
                for neighbor in graph[n]:
                    if neighbor not in visited:
                        visited.add(neighbor)
                        stack.append(neighbor)


        count = 0
        for i in range(n):
            if i not in visited:
                dfs(i)
                count += 1

        return count
```

TOPIC-14

Word Ladder

Q- Given two words and a dictionary, find the shortest transformation sequence from start to end.

Approach:

Use BFS to find the shortest transformation sequence.

Code:

```python
def ladderLength(beginWord, endWord, wordList):
    wordList = set(wordList)
    if endWord not in wordList:
        return 0
    queue = deque([(beginWord, 1)])
    while queue:
        word, length = queue.popleft()
        if word == endWord:
            return length
        for i in range(len(word)):
```

```
            for c in 'abcdefghijklmnopqrstuvwxyz':
                new_word = word[:i] + c + word[i+1:]
                    if new_word in wordList:
                        wordList.remove(new_word)
                queue.append((new_word, length + 1))
    return 0
```

TOPIC-15

Friend Circles

Q- Find the number of friend circles in a matrix representing friendships.

Approach:

Use DFS to find the number of friend circles.

Code:

```python
def findCircleNum(M):
    def dfs(student):
        for friend in range(len(M)):
            if M[student][friend] == 1 and friend not
            in visited:
                visited.add(friend)
                dfs(friend)

    visited = set()
    count = 0
```

```
    for student in range(len(M)):
            if student not in visited:
                dfs(student)
                count += 1
      return count
```

BOSSCODER ACADEMY

WHY BOSSCODER?

1000+ Alumni placed at Top Product-based companies.

More than **136% hike** for every 2 out of 3 Working Professional.

Average Package of **24LPA**.

The syllabus is most **up-to-date** and the list of problems provided covers **all important topics.**

Lavanya
Meta

Course is very well **structured** and **streamlined** to crack any **MAANG** company

Rahul
Google

EXPLORE MORE