Graph traversal is a method of visiting each node (vertex) of a graph systematically. Two popular traversal techniques are Breadth-First Search (BFS) and Depth-First Search (DFS). Let's go through each in detail, with code and visual representations.

**1. Breadth-First Search (BFS)**

**Explanation**:

* BFS traverses a graph layer by layer, exploring all nodes at the present depth level before moving on to nodes at the next depth level.
* BFS uses a **queue** data structure to keep track of nodes to visit.
* To avoid processing a node more than once (especially if the graph contains cycles), we use a **boolean visited array**.
* BFS begins from a specified **source node**.

**Algorithm Steps**:

1. Start from the source node, mark it as visited, and enqueue it.
2. While the queue is not empty:
   * Dequeue a node from the queue.
   * For each unvisited adjacent node, mark it as visited and enqueue it.

**Example Code for BFS**:

from collections import deque

# Function for BFS traversal

def bfs(graph, start\_node):

visited = [False] \* len(graph) # Create a visited list

queue = deque([start\_node]) # Initialize queue with the start node

visited[start\_node] = True # Mark the start node as visited

while queue:

# Dequeue a node and print it

node = queue.popleft()

print(node, end=" ")

# Enqueue all unvisited adjacent nodes

for neighbor in graph[node]:

if not visited[neighbor]:

visited[neighbor] = True

queue.append(neighbor)

# Example graph represented as an adjacency list

graph = {

0: [1, 2],

1: [0, 3, 4],

2: [0, 5],

3: [1],

4: [1],

5: [2]

}

# Calling BFS starting from node 0

bfs(graph, 0)

**Commit Steps**:

1. **Initialize BFS Algorithm**: Initialize visited list and queue with the start node.

Commit: "Initialize BFS with start node and visited list"

1. **Process Nodes in Queue**: While queue is not empty, dequeue the front node and print it.

Commit: "Dequeue node and print during BFS traversal"

1. **Visit Adjacent Nodes**: For each adjacent node, if unvisited, mark as visited and enqueue it.

Commit: "Enqueue unvisited adjacent nodes and mark them as visited"

**Visualization of BFS**: Suppose we start from node 0, the traversal order will be 0 -> 1 -> 2 -> 3 -> 4 -> 5.

Copy code

0

/ \

1 2

/ \ \

3 4 5

**Traversing Order**: 0 -> 1 -> 2 -> 3 -> 4 -> 5 (Level by level)

**2. Depth-First Search (DFS)**

**Explanation**:

* DFS explores a graph by going as deep as possible along each branch before backtracking.
* DFS uses a **stack** data structure (explicitly or implicitly via recursion).
* Similar to BFS, to avoid reprocessing nodes in a cyclic graph, a **visited list** is used.

**Algorithm Steps**:

1. Start from the source node, mark it as visited.
2. For each unvisited adjacent node, recursively apply DFS.
3. Continue until all nodes are visited.

**Example Code for DFS (Using Recursion)**:

# Function for DFS traversal

def dfs(graph, node, visited):

# Mark the current node as visited and print it

visited[node] = True

print(node, end=" ")

# Recursively visit all unvisited neighbors

for neighbor in graph[node]:

if not visited[neighbor]:

dfs(graph, neighbor, visited)

# Example graph represented as an adjacency list

graph = {

0: [1, 2],

1: [0, 3, 4],

2: [0, 5],

3: [1],

4: [1],

5: [2]

}

# Initialize visited list and call DFS starting from node 0

visited = [False] \* len(graph)

dfs(graph, 0, visited)

**Commit Steps**:

1. **Initialize DFS and Visited List**: Create a visited list and mark nodes during traversal.

Commit: "Initialize DFS traversal with visited list"

1. **Visit Current Node and Mark**: Mark the current node as visited and process it.

Commit: "Mark current node as visited and print it"

1. **Recursively Traverse Neighbors**: For each unvisited neighbor, call DFS recursively.

Commit: "Recursively visit unvisited neighbors"

**Visualization of DFS**: With the same graph as above, starting from node 0, the traversal could go as deep as possible before backtracking.

0

/ \

1 2

/ \ \

3 4 5

**Traversing Order**: 0 -> 1 -> 3 -> 4 -> 2 -> 5 (Depth-wise)

**Key Differences Between BFS and DFS**

* **Data Structure**: BFS uses a **queue**, while DFS uses a **stack** (or recursion).
* **Traversal Order**: BFS explores nodes level-by-level, while DFS goes deep along each branch.
* **Applications**:
  + BFS is used in shortest-path algorithms (like in unweighted graphs).
  + DFS is useful in topological sorting and detecting cycles.