4. Graph Traversal: Breadth-First Search (BFS)

Breadth-First Search (BFS) is a traversal algorithm that explores the graph "**level by level**." It starts at a source vertex, explores all of its immediate neighbors, then all of their neighbors, and so on.

4.1 BFS Analogy and Illustration

Imagine a **ripple effect in a pond**. When you drop a stone (the **source vertex**), the ripple expands:

- It first hits all the water molecules immediately next to it (Level 1 Neighbors).
- Then, the ripple expands to hit the next layer of molecules (Level 2 Neighbors).

BFS works exactly this way. It is excellent for finding the **shortest path** in an **unweighted graph**.

Example: We want to trace BFS starting from node `0` on the following graph:

```
      0
     / \
    1---2
   /     \
  3       4
```

The traversal order will be `0, 1, 2, 3, 4`.

4.2 The Role of `std::queue`

The FIFO (First-In, First-Out) nature of a queue is perfect for BFS.

1. We add the `source` (Level 0) to the queue.
2. We dequeue the `source`, and add all its neighbors (Level 1) to the queue
3. We dequeue all the Level 1 nodes one by one, and as we do, we add their neighbors (Level 2) to the **back** of the queue.

This ensures we process all nodes at the current level **before** moving to the next level, just like the ripple effect.

4.3 BFS Implementation

BFS is implemented using an **iterative approach** with a `std::queue`. This approach is natural to the algorithm's "level-by-level" logic. The queue ensures that nodes are processed in the order they are discovered. We also use a visited array (or `std::vector<bool>`) to keep track of nodes we have already processed or added to the queue, which is crucial for preventing infinite loops in graphs with cycles.

```
void BFS(int s) { // 's' is the source vertex
    // 1. Create a visited array, initialized to all false
    std::vector<bool> visited(V, false);

    // 2. Create a Queue for BFS
    std::queue<int> q;

    // 3. Mark the source vertex as visited and enqueue it
    visited[s] = true;
    q.push(s);

    std::cout << "BFS Traversal: ";

    // 4. Loop as long as the queue is not empty
    while (!q.empty()) {
        // 5. Dequeue a vertex from the front and print it
        int u = q.front();
        q.pop();
        std::cout << u << " ";

        // 6. Get all neighbors of the dequeued vertex 'u'
        for (auto& neighbor : adj[u]) {

            // 7. If a neighbor has not been visited:
            if (!visited[neighbor]) {
                // Mark it as visited
                visited[neighbor] = true;
                // Enqueue it (add to the back of the queue)
                q.push(neighbor);
```

```
            }
        }
    }
    std::cout << std::endl;
}
```

Code Explanation:

- Initialize a `visited` array (all `false`) and an empty `queue`.
- Mark the starting node `s` as visited and add it to the queue.
- While the queue is **not empty**:
  - Dequeue `a` node `u` (this is the **current node**).
  - Print `u`.
  - Iterate through all neighbors of `u`.
  - If a neighbor has not been visited, mark it as visited and enqueue it.

Revision #3
Created 2025-11-11 11:27:00 UTC by RE
Updated 2025-11-11 11:55:16 UTC by RE