# BFS

## (Breadth First Search)

---

### ➢ What is BFS?

BFS is a technique used to systematically explore all the nodes in a tree or graph. It begins at a chosen starting node and examines every node at the same depth before moving on to the next layer. The process relies on a queue to maintain the order of exploration. Since it reaches the closest nodes first, BFS is particularly effective for finding the shortest path in unweighted graphs.

### ➢ Key Points:

- Works on **graphs and trees**.
- Explores nodes **level by level**.
- Uses a **queue** to keep track of the next nodes to visit.
- Gives the **shortest path** in unweighted graphs.
- Very useful for solving problems like shortest path, finding connected components, and level-order traversal.

### ➢ Why Do We Use BFS?

A few reasons make BFS extremely useful:

1. **Shortest Path (Unweighted Graph)**
   BFS guarantees the shortest number of edges from start to end
2. **Level-Based Problems**
   BFS naturally processes nodes level by level.
3. **Checking Connectivity**
   You can find which nodes are reachable.
4. **Building Trees Layer by Layer**
5. **Real-World Uses**
   - Social network friend suggestion
   - GPS navigation (when roads are considered equal)
   - Web crawlers

### ➢ How BFS Works:

Imagine a person standing at a starting node in a graph.
Instead of moving deep into one long chain of nodes, the person:

1. First visits all the nodes directly connected to the starting node.
2. Then moves to the next set of nodes that are connected to those nodes.
3. Continues this process layer by layer, expanding outward through the graph.

## ➢ Concepts Needed for BFS

To understand BFS clearly, these concepts help build the foundation needed to follow how the algorithm works.

- ✓ **Graph Basics**
  - Nodes (vertices): The individual points or positions in a graph.
  - Edges: The links or connections between nodes.
  - Directed vs Undirected Graphs: Whether edges have a specific direction or not.
- ✓ **Data Structures**
  - Queue: The primary structure used in BFS (FIFO — First In, First Out).
  - Adjacency List / Adjacency Matrix: Common ways to represent a graph in code.
  - Visited Array: Keeps track of which nodes have already been explored.
- ✓ **Important Terms**
  - Source Node: The node from where the BFS begins.
  - Neighbors: Nodes directly connected to the current node.
  - Levels: The distance from the source measured in steps or layers.

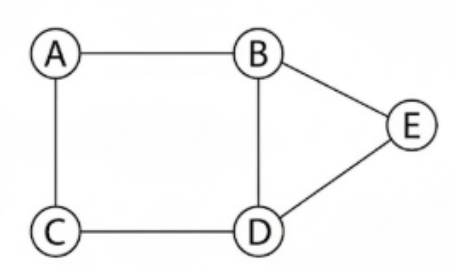
## ➢ BFS Algorithm — Step-by-Step

1. Pick a starting node.
2. Mark it as visited.
3. Push it into a queue.
4. While the queue is not empty:
   - Remove the front node.
   - Visit all its unvisited neighbors.
   - Mark neighbors as visited.
   - Push neighbors into the queue.

## ➢ BFS Pseudocode:

```
BFS(G, s) {
    for each u in V {
        color[u] = white
        d[u] = infinity
        pred[u] = null
    }
    color[s] = gray
    d[s] = 0
    Q = {s}
  while (Q is nonempty) {
    u = Q.Dequeue()
  for each v in Adj[u] {
        if (color[v] == white) {
            color[v] = gray
            d[v] = d[u]+1
            pred[v] = u
            Q.Enqueue(v)
        }
    }
 color[u] = black
}
    }
```

- **Example Graph:**




**BFS from source node A:**

- ✓ **Step 1:** Start from A
  Visited: A
  Queue: [B, C]
- ✓ **Step 2:** Visit B
  Visited: A, B
  Queue: [C, D, E]
- ✓ **Step 3:** Visit C
  Visited: A, B, C
  Queue: [D, E]
- ✓ **Step 4:** Visit D
  Visited: A, B, C, D
  Queue: [E]
- ✓ **Step 5:** Visit E
  Visited: A, B, C, D, E
  Queue: []

**BFS Order:** A → B → C → D → E

- **Time & Space Complexity**

  - Time Complexity: O(V + E)
  - Space Complexity: O(V)

- **When Not to Use BFS?**

Although BFS is great, not every problem fits it. Use BFS only when:

- Graph is unweighted or all weights are equal.
- You want the shortest path in terms of number of edges.

Use Dijkstra if weights differ. Use DFS for deep exploration or recursion-based tasks.

- **Summary:**

After learning BFS, one can traverse graphs, trees, and grids, find shortest paths in unweighted graphs, count levels or connected components, check whether a graph is bipartite, solve maze or grid-based problems, and implement advanced BFS variations such as multi-source BFS or BFS with path reconstruction.