Algorithms for solving tree problems

Depth-First Search (BFS) on Trees

Note on Terminology: You mentioned "Depth-First Search (BFS)". To clarify, **Depth-First Search** is abbreviated as **DFS**, while **Breadth-First Search** is abbreviated as **BFS**. Since you spelled out "Depth-First," this guide focuses on **DFS**.

What is Depth-First Search (DFS)?

Imagine you are in a maze. You want to find the exit.

- **DFS Strategy:** You pick a path and keep walking forward until you hit a dead end. Once you hit a wall, you backtrack (retracing your steps) to the last intersection and try a different path. You repeat this until you find the exit.
- **The Key Idea:** "Go Deep." Dive as far down a branch as possible before coming back up.

Why do we need it? (The Problem it Solves)

DFS is essential for problems where you need to:

1. **Explore every possible path** (e.g., solving a maze, finding all solutions to a puzzle).
2. **Reach leaf nodes** (the bottom of the tree) quickly.
3. **Check for connectivity** (can I get from A to B?).

Visual Walkthrough: How DFS Works

Let's use a simple binary tree.

The Tree:

```
       (1)   <-- Root
      /   \
    (2)   (3)
    / \     \
  (4) (5)   (6)
```

The Goal: Visit every node using DFS. We will prioritize the **Left** child, then the **Right** child.

Step-by-Step Execution

Step 1: Start at Root **(1)**.

```
Current: (1)
Stack/Path: [1]
Action: Go Left to (2).
```

Step 2: At Node **(2)**.

```
        (1)
       /
     (2)*
     / \
   (4) (5)

Current: (2)
Stack/Path: [1, 2]
Action: Go Left to (4).
```

Step 3: At Node **(4)**.

```
        (1)
       /
     (2)
     /
   (4)*

Current: (4)
Stack/Path: [1, 2, 4]
Action: No left child. No right child. Dead end! Backtrack to (2).
```

Step 4: Back at Node **(2)**.

```
Current: (2)
Action: We already visited Left (4). Now go Right to (5).
```

Step 5: At Node **(5)**.

```
        (1)
       /
     (2)
       \
       (5)*

Current: (5)
Stack/Path: [1, 2, 5]
Action: No children. Dead end! Backtrack to (2).
```

Step 6: Back at Node **(2)**.

```
Current: (2)
Action: Left done. Right done. Backtrack to (1).
```

Step 7: Back at Node **(1)**.

```
Current: (1)
Action: Left side is completely done. Go Right to (3).
```

Step 8: At Node **(3)**.

```
        (1)
           \
           (3)*
             \
             (6)
```

```
Current: (3)
Action: No left child. Go Right to (6).
```

Step 9: At Node **(6)**.

```
             (6)*
```

```
Current: (6)
Action: Dead end. Backtrack to (3).
```

Step 10: Back at (3), then Back at (1). Done.

Final Order of Visits: 1 -> 2 -> 4 -> 5 -> 3 -> 6

Time and Space Complexity

Time Complexity: O(N)

- **N** = Number of nodes in the tree.
- We visit every node exactly once.

```
Time Taken
^
|      /
|     /   (Linear Growth)
|    /
|   /
|  /
| /
|/
+------------------>
  Number of Nodes (N)
```

Space Complexity: O(H)

- **H** = Height of the tree.
- In the worst case (a straight line tree), the recursion stack (or manual stack) holds all nodes.
- In a balanced tree, H = log(N).

```
Stack Memory Usage
^
|    |      (Worst Case: Skewed Tree)
|    |
|    |
|    |
|    |
|____|____  (Best Case: Balanced Tree - shorter stack)
+----------------->
  Tree Height (H)
```

The Algorithms (Python & JavaScript)

There are two main ways to write DFS: **Recursive** (easier to write) and **Iterative** (uses a manual stack).

1. Python Implementation

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# --- Recursive Approach (Most Common) ---
def dfs_recursive(root):
    if not root:
        return []

    result = []

    def traverse(node):
        if not node:
            return

        # 1. Process current node (Pre-order)
        result.append(node.val)
        # 2. Go Left
        traverse(node.left)

        # 3. Go Right
        traverse(node.right)

    traverse(root)
    return result

# --- Iterative Approach (Using a Stack) ---
def dfs_iterative(root):
    if not root:
        return []

    stack = [root]
    result = []

    while stack:
        # Pop the top element
        node = stack.pop()
        result.append(node.val)

        # Add children to stack.
        # Note: We push RIGHT first, so LEFT is popped first!
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)

    return result
```

2. JavaScript Implementation

```
// Definition for a binary tree node.
function TreeNode(val, left, right) {
    this.val = (val===undefined ? 0 : val)
    this.left = (left===undefined ? null : left)
    this.right = (right===undefined ? null : right)
}

// --- Recursive Approach ---
var dfsRecursive = function(root) {
    let result = [];

    function traverse(node) {
        if (!node) return;

        // 1. Process
        result.push(node.val);

        // 2. Left
        traverse(node.left);

        // 3. Right
        traverse(node.right);
    }

    traverse(root);
    return result;
};

// --- Iterative Approach ---
var dfsIterative = function(root) {
    if (!root) return [];

    let stack = [root];
    let result = [];

    while (stack.length > 0) {
        let node = stack.pop();
        result.push(node.val);

        // Push Right first so Left is processed next
        if (node.right) stack.push(node.right);
        if (node.left) stack.push(node.left);
    }

    return result;
};
```

How to Use DFS for LeetCode Problems

When you see a problem, ask yourself: **"Does the answer lie at the end of a path?"**

Mental Template

1. **Base Case:** What happens when I hit a `null` node? (Usually return 0, False, or None).

2. **Recursive Step:** What do I need from my children? (e.g., "Ask left child for its depth, ask right child for its depth").
3. **Combination:** How do I combine those answers to solve it for *myself* (the current node)?

Example: Maximum Depth of Binary Tree

- **Thought:** "My depth is 1 (myself) + the bigger depth of my two children."
- **Code logic:** `return 1 + max(dfs(left), dfs(right))`

Comparison: DFS vs. BFS

Feature	DFS (Depth-First)	BFS (Breadth-First)
Concept	Plunge to the bottom, then backtrack.	Explore layer by layer (neighbors first).
Data Structure	**Stack** (LIFO - Last In First Out).	**Queue** (FIFO - First In First Out).
Memory	Good for deep, narrow trees.	Good for shallow, wide trees.
Shortest Path?	No (might find a long path first).	**Yes** (always finds shortest path in unweighted graphs).
Diagram	`|` (Vertical exploration)	

Breadth-First Search (BFS) on Trees

Here is a comprehensive guide to the Breadth-First Search (BFS) algorithm, designed to be visual and easy to follow.

---

1. The Problem: Why do we need BFS?

Imagine you are at the top of a corporate hierarchy (the root of a tree) and you want to broadcast a message.

- **Option A:** You tell one person, they tell their subordinate, who tells their subordinate, going all the way down to the interns before coming back up to tell the next manager. This is **Depth-First Search (DFS)**. It goes deep quickly.
- **Option B:** You tell all your direct reports first. Once they all know, they tell all of their direct reports. The news spreads layer by layer. This is **Breadth-First Search (BFS)**.

We use BFS when:

1. We need to find the **shortest path** from the start to a target (in an unweighted graph/tree).
2. We need to process items **level by level** (e.g., printing a directory structure).
3. We want to find nodes close to the root before exploring nodes far away.

---

2. What is BFS?

BFS is a traversal algorithm that visits nodes in "waves" or "layers." It starts at the root and explores all neighbors at the current depth (level) before moving on to nodes at the next depth level.

Visual Analogy: Think of dropping a stone into a pond. The ripples move outward in perfect circles. The ripple doesn't shoot out in one straight line; it expands equally in all directions.

```
      (Root)       <- Ripple 1 (Time 0)
     /      \
   (A)      (B)    <- Ripple 2 (Time 1)
   / \      / \
 (C) (D)  (E) (F)  <- Ripple 3 (Time 2)
```

---

3. How It Works: The Queue

To make BFS work, we need a specific data structure: a **Queue**.

A Queue follows the **FIFO** rule: **First In, First Out**. Think of a line at a grocery store. The first person in line is the first one to be served.

- **Step 1:** Add the root to the Queue.
- **Step 2:** While the Queue is not empty:
- Take the first item out of the Queue (dequeue).
- "Visit" it (print it, check it, etc.).
- Add all of its children to the back of the Queue (enqueue).

---

4. Visual Walkthrough

Let's traverse this tree using BFS.

The Tree:

```
       [1]
      /   \
    [2]    [3]
```

```
     / \       \
   [4] [5]     [6]
```

Goal: Visit every node level-by-level.

State 0: Initialization We start by putting the Root [1] into the Queue.

```
Current Node: None
Output: []

Queue:
Front [ 1 ] Back
```

Step 1: Process Node [1] We remove [1] from the front. We visit it. Then, we add its children ([2] and [3]) to the back.

```
Current Node: [1]
Output: [1]

   Remove [1] ->  Add children [2], [3]

Queue:
Front [ 2 | 3 ] Back
```

Step 2: Process Node [2] We remove [2] from the front. We visit it. We add its children ([4] and [5]) to the back.

```
Current Node: [2]
Output: [1, 2]

   Remove [2] ->  Add children [4], [5]
                  (Notice [3] is still waiting!)

Queue:
Front [ 3 | 4 | 5 ] Back
```

Step 3: Process Node [3] We remove [3] from the front. We visit it. We add its child ([6]) to the back.

```
Current Node: [3]
Output: [1, 2, 3]

   Remove [3] ->  Add child [6]

Queue:
Front [ 4 | 5 | 6 ] Back
```

Step 4: Process Node [4] We remove [4]. It has no children (leaf node), so we add nothing.

```
Current Node: [4]
Output: [1, 2, 3, 4]

Queue:
Front [ 5 | 6 ] Back
```

Step 5 & 6: Process [5] and [6] Both are leaves. We simply remove them and visit them.

```
Final Output: [1, 2, 3, 4, 5, 6]
Queue is now empty. Done!
```

---

5. Implementation Code

Here is how you write this in code.

Python Implementation

```python
from collections import deque

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def bfs(root):
    if not root:
        return []

    result = []
    # Initialize Queue with the root
    queue = deque([root])

    while queue:
        # Remove from the left (front)
        current_node = queue.popleft()

        # "Visit" the node
        result.append(current_node.val)

        # Add children to the right (back)
        if current_node.left:
            queue.append(current_node.left)
        if current_node.right:
            queue.append(current_node.right)

    return result
```

JavaScript Implementation

```
function bfs(root) {
  if (!root) return [];

  const result = [];
  // Initialize Queue with the root
  const queue = [root];

  while (queue.length > 0) {
    // Remove from the front (index 0)
    // Note: shift() is O(n) in JS arrays, a real Deque is preferred
    // for production, but this is standard for interviews.
    const currentNode = queue.shift();

    // "Visit" the node
    result.push(currentNode.val);

    // Add children to the back
    if (currentNode.left) {
      queue.push(currentNode.left);
    }
    if (currentNode.right) {
      queue.push(currentNode.right);
    }
  }
  return result;
}
```

---

6. Complexity Analysis (Visualized)

Time Complexity: O(N) N = Number of nodes. Since every node enters the queue exactly once and leaves the queue exactly once, the work is directly proportional to the number of nodes.

```
Graph:  (A) -- (B) -- (C) ... (N)

Operations:
Push A -> Pop A -> Push B -> Pop B ...
```

```
  1        1         1        1

Total Ops = 2 * N  =>  O(N)
```

Space Complexity: O(W)   W = Maximum Width of the tree. In BFS, the Queue holds the "frontier" or the current level being explored. The Queue gets heaviest when we are at the widest part of the tree.

Case 1: A Line (Linked List Tree) Width is 1. Space is minimal.

```
    (A)
    /
  (B)
  /
(C)     Queue Max Size: 1 node
```

Case 2: A Perfect Triangle The bottom layer is the widest. It holds roughly N/2 nodes.

```
        (A)           Level 1 (1 node)
       /   \
     (B)   (C)        Level 2 (2 nodes)
     / \   / \
   (D)(E) (F)(G)      Level 3 (4 nodes) <- Queue holds all these at once!
```

In the worst case (a full tree), the width is proportional to N. So, often we say space is **O(N)**, or more specifically **O(Width)**.

---

7. LeetCode Strategy: How to use this

When solving problems (like on LeetCode), use BFS if you see these keywords or patterns:

1. "Shortest Path" or "Fewest Steps"

- Because BFS spreads layer-by-layer, the first time you reach your target, you are guaranteed to have taken the shortest path (in unweighted graphs).
- *Example: Word Ladder, Minimum Depth of Binary Tree.*

2. "Level Order Traversal"

- If the output format requires grouping nodes by their depth (e.g., [[root], [level2_a, level2_b], ...]).
- *Example: Binary Tree Level Order Traversal.*

3. "Connected Components" (in grids)

- Simulating a virus spreading or rot spreading in a grid.
- *Example: Rotting Oranges, Number of Islands.*

How to structure your Loop for "Levels": Sometimes you need to track which "level" you are on (e.g., to count steps). Use a `for` loop *inside* your `while` loop.

```
# Pattern for tracking levels/steps
steps = 0
while queue:
    size = len(queue)
    # This inner loop processes one entire level
    for i in range(size):
        node = queue.popleft()
        # check if node is target...
        # add neighbors...
    steps += 1
```

---

8. BFS vs. DFS (Comparison)

Feature	BFS (Breadth-First)	DFS (Depth-First)
Data Structure	Queue (FIFO)	Stack (LIFO) or Recursion
Motion	Wide and shallow (Layer by Layer)	Deep and narrow (Plunge to bottom)
Best for...	Shortest path, nodes close to root	Mazes, puzzles requiring backtracking
Space Cost	High (stores width of tree)	Low (stores height of tree)

Visual Comparison:

BFS (Spreading Water): Fills the room evenly.

```
Root -> Level 1 -> Level 2 -> ...
```

DFS (A Snake in a Maze): Goes as far as possible, hits a wall, then backtracks.

```
Root -> Left -> Left -> Left (hit bottom) -> Back -> Right ...
```