Ultimate Guide to Solving Graph Problems with DFS and BFS

🧠 Introduction

Graph problems come in many flavors: pathfinding, cycle detection, connectivity, ordering, and optimization. Most of them boil down to using either **Depth-First Search (DFS)** or **Breadth-First Search (BFS)**. This document gives you a comprehensive and crystal-clear way to identify which algorithm to use, how to structure your code, and how to modify it based on specific problem constraints.

🔢 Identify Graph Problem Types

Before choosing DFS or BFS, ask:

1. **What is the goal?**
   - All paths?
   - Shortest path?
   - Any valid path?
   - Topological order?
   - Use all edges once?
2. **What constraints exist?**
   - Lexicographical order?
   - Weighted/unweighted edges?
   - Can nodes/edges be reused?
   - Directed or undirected?

🧰 DFS: Depth-First Search

Used when:

- You want to explore all possibilities (backtracking)
- You want post-order behavior (e.g., use all edges before adding to result)
- You want to find cycles (by tracking visited in current path)
- You want all paths
- Exploring connected components

DFS uses a visited set to keep track of nodes that have already been explored. This is crucial for preventing infinite loops in graphs that contain cycles.

📚 DFS Recursion Template

```
visited = set()

def dfs(node):
```

```
    visited.add(node)
    for neighbor in graph.get(node, []):
        if neighbor not in visited:
            dfs(neighbor)
```

📚 DFS Iterative Template

```
def dfs_iterative(start_node):
    stack = [start_node]
    visited = set()
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            # Process the node here (e.g., add to result)
            for neighbor in reversed(graph.get(node, [])):  # Process neighbors in reverse
order for similar exploration to recursive DFS
                if neighbor not in visited:
                    stack.append(neighbor)
    return visited
```

🔹 DFS with Path Tracking (All Paths)

```
path = []
result = []

def dfs_paths(node):
    path.append(node)
    if node == TARGET:
        result.append(path[:])
    else:
        for neighbor in graph.get(node, []):
            dfs_paths(neighbor)
    path.pop()
```

🔹 Post-order DFS (e.g., Leetcode 332 - Reconstruct Itinerary)

In post-order DFS, you recursively visit all destinations before adding the current node to the result. This is especially useful when you need to construct an itinerary or

topological order after consuming all outgoing edges.

Here’s the implementation:

```
route = []

def dfs_postorder(node):
    while graph[node]:
        next_city = heapq.heappop(graph[node])
        dfs_postorder(next_city)
    route.append(node)  # post-order
```

Visual Stack Unwinding Example
Imagine this graph:
JFK -> ATL
JFK -> SFO
ATL -> LAX

DFS starts from JFK and goes deep:

- dfs_postorder("JFK")
  - dfs_postorder("ATL")
    - dfs_postorder("LAX") → append "LAX"
      → append "ATL"
  - dfs_postorder("SFO") → append "SFO"
    → append "JFK"

So the stack unwinds in the reverse visiting order: ['LAX', 'ATL', 'SFO', 'JFK'], and you return [::-1] to get the itinerary.

```
route = []


def dfs_postorder(node):
    while graph[node]:
        next_city = heapq.heappop(graph[node])
        dfs_postorder(next_city)
    route.append(node)  # post-order
```

🔹 **DFS Topological Sort**

```
visited = set()
stack = []


def dfs_topo(node):
    visited.add(node)
    for neighbor in graph.get(node, []):
        if neighbor not in visited:
            dfs_topo(neighbor)
    stack.append(node)  # post-order


for node in graph:
    if node not in visited:
        dfs_topo(node)

stack.reverse()
```

🌐 BFS: Breadth-First Search

Used when:

- You want the shortest path in **unweighted** graphs
- You want level-by-level traversal
- Exploring connected components
- Multi-source shortest path problems

Visual Example: Level-wise Traversal

Graph:

```
   1
  / \
 2   3
/     \
4      5
```

Level 0: [1]
Level 1: [2, 3]
Level 2: [4, 5]

BFS explores level by level:

- Visit node 1 ⟶ enqueue [2, 3]
- Visit node 2 ⟶ enqueue [4]
- Visit node 3 ⟶ enqueue [5]
- Visit nodes 4 and 5 ⟶ done

📚 BFS Template

```
from collections import deque


def bfs(start):
    queue = deque([start])
    visited = set([start])

    while queue:
        node = queue.popleft()
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

🔹 BFS for Shortest Path

```
from collections import deque


def bfs_shortest_path(start, target):
    queue = deque([(start, [start])])
    visited = set([start])

    while queue:
        node, path = queue.popleft()
        if node == target:
            return path
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                visited.add(neighbor)
```

```
            queue.append((neighbor, path + [neighbor]))
    return None  # Or handle case where target is not reachable
```

🔹 **Multi-Source BFS (e.g., Rotting Oranges)**

In multi-source BFS, the queue is initialized with *all* the starting nodes.

```
from collections import deque


def multi_source_bfs(initial_sources):
    queue = deque(initial_sources)
    visited = set(initial_sources)
    # Additional logic to track distance/time

    while queue:
        node = queue.popleft()
        for neighbor in get_neighbors(node):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
                # Update distance/time for the neighbor
```

⚖️ Choosing DFS vs BFS

Goal	Choose	Notes
Find **any path** A ⟶ B	DFS	Use recursion or iterative
Find **shortest path**	BFS	Unweighted graph
All paths	DFS	Use path list + backtracking
Topological sort	DFS / BFS	DFS (post-order), BFS (Kahn's algorithm)
Cycle detection	DFS	Track recursion stack or visited in path
Use all edges once	DFS	Eulerian path (Hierholzer's

		Algorithm)
Connected components	DFS / BFS	Explore all reachable nodes

📈 DFS vs BFS Comparison Table

Feature	**DFS**	**BFS**
Space	O(h) (h = height, recursive), O(n) worst-case iterative	O(w) (w = width, can be O(n) worst)
Use Case	Backtracking, Topo, All Paths, Cycle Detection, Connected Components	Shortest Path (unweighted), Level Traversal, Multi-Source BFS
Supports pruning	Yes	Sometimes (target found early)
Easy to go deep	Yes	No
Implementation	Often recursive	Typically iterative (queue)

🔄 Decision Tree for DFS vs BFS

Track full path?
  |
 Yes
  |
 DFS
  |
Best result = first solution found?
  | \
 Yes No
  | |
Return early Explore all paths

🧠 Pro Tips for Interviews

- **"Use all tickets"** → Think post-order DFS (LeetCode 332)
- **"Shortest path"** → Think BFS unless weights involved (then Dijkstra)
- **"All paths"** → Think DFS with backtracking

- **"Lexical order"** → Sort neighbors or use heapq (priority queue)
- **Weighted shortest** → Use Dijkstra (not DFS/BFS)
- **Edge constraints** → Be mindful of visited sets; sometimes mark edges as used
- **Disconnected graphs** → Iterate through all nodes and start traversal if unvisited

```
def traverse_all(graph, algorithm='dfs'):
    visited = set()
    results = []
    for node in graph:
        if node not in visited:
            if algorithm == 'dfs':
                component = []
                def dfs_component(u):
                    visited.add(u)
                    component.append(u)
                    for v in graph.get(u, []):
                        if v not in visited:
                            dfs_component(v)
                dfs_component(node)
                results.append(component)
            elif algorithm == 'bfs':
                component = []
                queue = deque([node])
                visited.add(node)
                while queue:
                    u = queue.popleft()
                    component.append(u)
                    for v in graph.get(u, []):
                        if v not in visited:
                            visited.add(v)
                            queue.append(v)
                results.append(component)
    return results
```

🔗 Graph Representation Tips

- Use defaultdict(list) or defaultdict(heapq)
- For undirected graphs: add edges in both directions: graph[a].append(b) and graph[b].append(a)

- For weighted graphs: graph[u].append((v, weight))

🎡 Practice Problems

Visual Walkthrough: Leetcode 797 - All Paths from Source to Target

Graph:
0 -> 1 -> 3
\ ^
-> 2 --

DFS Traversal (starting from 0):

- Visit 0
  - Visit 1
    - Visit 3 ⟶ Path: [0, 1, 3] (Target reached, add to result) ⟶ Backtrack
  - Backtrack from 1
  - Visit 2
    - Visit 3 ⟶ Path: [0, 2, 3] (Target reached, add to result) ⟶ Backtrack
  - Backtrack from 2
- Backtrack from 0

Result:

[[0, 1, 3], [0, 2, 3]]

This shows backtracking in action as DFS explores all possible routes.

Problem	Type	Algorithm
Leetcode 797	All paths	DFS + path
Leetcode 332	Use all edges once	Post-order DFS
Leetcode 207 / 210	Topological Sort	DFS / BFS
Leetcode 200 / 547	Connected Components	DFS / BFS
Leetcode 743	Weighted Shortest	Dijkstra
Leetcode 133 / 417 / 994	Multi-source BFS	BFS

🔮 Final Mindset

> "Graph problems test your ability to track state over time. DFS is surgical, BFS is systematic. Know when to go deep, and when to go wide."

Practice recognizing the patterns. Build DFS/BFS muscle memory. You'll go from confused to confident.

Let me know if you want a visual markdown version or quiz-based practice next!