Volume 3: Graph Algorithms & Advanced Data Structures

Table of Contents

BFS, DFS, Dijkstra, Union-Find, Topological Sort

Target: Infosys L3 Specialist Programmer
Edition: 2025
Focus: Graph problems constitute 15-20% of coding round

Preface

Graph problems test **algorithmic maturity**—the ability to model real-world scenarios as graphs and apply appropriate traversal/search strategies.

Infosys L3 expects:

- Clean BFS/DFS templates (codable in <5 minutes)
- Shortest path algorithms (Dijkstra for weighted graphs)
- Cycle detection in directed/undirected graphs
- Topological sorting for dependency resolution

Pattern Recognition: 80% of graph problems use BFS or DFS as foundation.

Table of Contents

Chapter 1: Graph Representations

1.1 Adjacency List vs Adjacency Matrix

Aspect	Adjacency List	Adjacency Matrix
Space	$O(V + E)$	$O(V^2)$
Check Edge	O(degree)	$O(1)$
Iterate Neighbors	O(degree)	$O(V)$
Best For	Sparse graphs ($E << V^2$)	Dense graphs, quick edge lookup

Sparse Graph (social network): $E \approx V$
Dense Graph (complete graph): $E \approx V^2$

Infosys Standard: Use adjacency list (most graphs are sparse).

1.2 C++ Implementation

Adjacency List:

```
#include &lt;vector&gt;
#include &lt;list&gt;
using namespace std;

// Unweighted graph
vector&lt;vector&lt;int&gt;&gt; graph(n); // n vertices
graph[u].push_back(v); // Edge u → v

// Weighted graph
vector&lt;vector&lt;pair&lt;int, int&gt;&gt;&gt; graph(n);
graph[u].push_back({v, weight}); // Edge u → v with weight
```

Adjacency Matrix:

```
vector&lt;vector&lt;int&gt;&gt; graph(n, vector&lt;int&gt;(n, 0));
graph[u][v] = 1; // Edge u → v

// Weighted:
graph[u][v] = weight;
```

1.3 Python Implementation

```
from collections import defaultdict

# Adjacency list<a></a>
graph = defaultdict(list)
graph[u].append(v)

# Weighted<a></a>
graph = defaultdict(list)
graph[u].append((v, weight))
```

Chapter 2: Breadth-First Search (BFS)

2.1 Concept

BFS explores level-by-level (like ripples in water).

Applications:

- Shortest path in **unweighted graph**
- Level-order traversal of tree
- Connected components
- Bipartite graph check

Time: O(V + E), **Space**: O(V)

2.2 Template (C++)

```
#include &lt;queue&gt;
#include &lt;vector&gt;
using namespace std;

void bfs(vector&lt;vector&lt;int&gt;&gt;&amp; graph, int start) {
    int n = graph.size();
    vector&lt;bool&gt; visited(n, false);
    queue&lt;int&gt; q;

    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        // Process u
        cout &lt;&lt; u &lt;&lt; " ";

        for (int v : graph[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

2.3 Shortest Path in Unweighted Graph

Problem: Find shortest distance from source to all vertices.

Solution:

```
vector&lt;int&gt; shortestPath(vector&lt;vector&lt;int&gt;&gt;&amp; graph, int start) {
    int n = graph.size();
    vector&lt;int&gt; dist(n, -1); // -1 = unreachable
    queue&lt;int&gt; q;

    q.push(start);
    dist[start] = 0;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : graph[u]) {
            if (dist[v] == -1) { // Not visited
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
    return dist;
}
```

Example:

```
Graph: 0 → 1 → 2
       ↓       ↓
       3 → 4 ← 5

Start = 0
dist = [0, 1, 2, 1, 2, 3]
```

2.4 Python Template

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        u = queue.popleft()
        print(u, end=' ')

        for v in graph[u]:
            if v not in visited:
                visited.add(v)
                queue.append(v)
```

2.5 Problem: Number of Islands (Infosys Medium)

Statement: Given 2D grid of '1's (land) and '0's (water), count islands.

Input:

```
[
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
```

Output: 3

Approach: BFS from each unvisited '1', mark all connected '1's.

C++ Solution:

```
int numIslands(vector&lt;vector&lt;char&gt;&gt;&amp; grid) {
    if (grid.empty()) return 0;

    int m = grid.size(), n = grid[0].size();
    int count = 0;

    auto bfs = [&amp;](int i, int j) {
        queue&lt;pair&lt;int, int&gt;&gt; q;
        q.push({i, j});
        grid[i][j] = '0'; // Mark visited

        int dirs[4][2] = {{-1,0}, {1,0}, {0,-1}, {0,1}};

        while (!q.empty()) {
            auto [x, y] = q.front();
            q.pop();

            for (auto&amp; d : dirs) {
                int nx = x + d[0], ny = y + d[1];
                if (nx &gt;= 0 &amp;&amp; nx &lt; m &amp;&amp; ny &gt;= 0 &amp;&amp; ny &l
                    &amp;&amp; grid[nx][ny] == '1') {
                    grid[nx][ny] = '0';
                    q.push({nx, ny});
                }
            }
        }
    };

    for (int i = 0; i &lt; m; i++) {
        for (int j = 0; j &lt; n; j++) {
            if (grid[i][j] == '1') {
                bfs(i, j);
                count++;
            }
        }
    }
    return count;
}
```

Complexity: O(m × n)

Chapter 3: Depth-First Search (DFS)

3.1 Concept

DFS explores as deep as possible before backtracking.

Applications:

- Cycle detection
- Topological sort
- Connected components
- Path finding (not necessarily shortest)

Time: O(V + E), **Space**: O(V) for recursion stack

3.2 Template (Recursive)

```
void dfs(vector&lt;vector&lt;int&gt;&gt;&amp; graph, int u, vector&lt;bool&gt;&amp; visite
    visited[u] = true;

    // Process u
    cout &lt;&lt; u &lt;&lt; " ";

    for (int v : graph[u]) {
        if (!visited[v]) {
            dfs(graph, v, visited);
        }
    }
}

// Wrapper
void dfsGraph(vector&lt;vector&lt;int&gt;&gt;&amp; graph, int start) {
    int n = graph.size();
    vector&lt;bool&gt; visited(n, false);
    dfs(graph, start, visited);
}
```

3.3 Template (Iterative with Stack)

```
void dfsIterative(vector&lt;vector&lt;int&gt;&gt;&amp; graph, int start) {
    int n = graph.size();
    vector&lt;bool&gt; visited(n, false);
    stack&lt;int&gt; st;

    st.push(start);

    while (!st.empty()) {
        int u = st.top();
        st.pop();

        if (visited[u]) continue;
        visited[u] = true;

        cout &lt;&lt; u &lt;&lt; " ";

        for (int v : graph[u]) {
            if (!visited[v]) {
                st.push(v);
```

```
            }
        }
    }
}
```

3.4 Cycle Detection (Undirected Graph)

Approach: DFS with parent tracking. If we visit a neighbor that's:

- Not parent → Cycle exists
- Not visited → Continue DFS

```
bool hasCycleDFS(vector&lt;vector&lt;int&gt;&gt;&amp; graph, int u,
                 int parent, vector&lt;bool&gt;&amp; visited) {
    visited[u] = true;

    for (int v : graph[u]) {
        if (!visited[v]) {
            if (hasCycleDFS(graph, v, u, visited))
                return true;
        } else if (v != parent) {
            return true; // Cycle found
        }
    }
    return false;
}

bool hasCycle(vector&lt;vector&lt;int&gt;&gt;&amp; graph) {
    int n = graph.size();
    vector&lt;bool&gt; visited(n, false);

    for (int i = 0; i &lt; n; i++) {
        if (!visited[i]) {
            if (hasCycleDFS(graph, i, -1, visited))
                return true;
        }
    }
    return false;
}
```

3.5 Cycle Detection (Directed Graph)

Approach: Use **color scheme**:

- White (0): Not visited
- Gray (1): In current DFS path
- Black (2): Completely processed

Cycle exists if we encounter Gray node (back edge).

```
bool hasCycleDFS(vector&lt;vector&lt;int&gt;&gt;&amp; graph, int u,
                 vector&lt;int&gt;&amp; color) {
    color[u] = 1; // Gray (in path)

    for (int v : graph[u]) {
        if (color[v] == 1) return true; // Back edge
        if (color[v] == 0 &amp;&amp; hasCycleDFS(graph, v, color))
            return true;
    }
```

```
    color[u] = 2; // Black (done)
    return false;
}

bool hasCycleDirected(vector&lt;vector&lt;int&gt;&gt;&amp; graph) {
    int n = graph.size();
    vector&lt;int&gt; color(n, 0); // White

    for (int i = 0; i &lt; n; i++) {
        if (color[i] == 0) {
            if (hasCycleDFS(graph, i, color))
                return true;
        }
    }
    return false;
}
```

Chapter 4: Shortest Path (Dijkstra)

4.1 Concept

Dijkstra's Algorithm: Shortest path in **weighted graph with non-negative weights**.

Greedy Strategy: Always expand nearest unvisited node.

Time: O((V + E) log V) with priority queue
Space: O(V)

4.2 Template (C++)

```
#include &lt;queue&gt;
#include &lt;vector&gt;
using namespace std;

vector&lt;int&gt; dijkstra(vector&lt;vector&lt;pair&lt;int, int&gt;&gt;&gt;&amp; graph,
                     int start) {
    int n = graph.size();
    vector&lt;int&gt; dist(n, INT_MAX);

    // Min-heap: {distance, node}
    priority_queue&lt;pair&lt;int, int&gt;,
                   vector&lt;pair&lt;int, int&gt;&gt;,
                   greater&lt;&gt;&gt; pq;

    dist[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();

        if (d &gt; dist[u]) continue; // Outdated entry

        for (auto [v, w] : graph[u]) {
            if (dist[u] + w &lt; dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
```

```
    return dist;
}
```

4.3 Python Template

```
import heapq

def dijkstra(graph, start):
    n = len(graph)
    dist = [float('inf')] * n
    dist[start] = 0

    pq = [(0, start)]  # (distance, node)

    while pq:
        d, u = heapq.heappop(pq)

        if d &gt; dist[u]:
            continue

        for v, w in graph[u]:
            if dist[u] + w &lt; dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))

    return dist
```

4.4 Problem: Network Delay Time (Infosys Medium)

Statement: Given network of n nodes, edges with travel times, find minimum time for signal to reach all nodes from source k.

Input: times = [[2,1,1], [2,3,1], [3,4,1]], n = 4, k = 2
Output: 2 (max distance from source)

Solution: Dijkstra + find maximum distance.

```
int networkDelayTime(vector&lt;vector&lt;int&gt;&gt;&amp; times, int n, int k) {
    vector&lt;vector&lt;pair&lt;int, int&gt;&gt;&gt; graph(n + 1);

    for (auto&amp; t : times) {
        graph[t[0]].push_back({t[1], t[2]});
    }

    vector&lt;int&gt; dist = dijkstra(graph, k);

    int maxDist = 0;
    for (int i = 1; i &lt;= n; i++) {
        if (dist[i] == INT_MAX) return -1; // Unreachable
        maxDist = max(maxDist, dist[i]);
    }
    return maxDist;
}
```

Chapter 5: Topological Sort

5.1 Concept

Topological Order: Linear ordering of vertices in DAG (Directed Acyclic Graph) where for every edge u → v, u comes before v.

Applications:

- Task scheduling with dependencies
- Course prerequisites
- Build systems (Makefile)

Algorithm: DFS with post-order traversal (reverse finish times).

5.2 Template (DFS-Based)

```
void topSortDFS(vector&lt;vector&lt;int&gt;&gt;&amp; graph, int u,
                vector&lt;bool&gt;&amp; visited, stack&lt;int&gt;&amp; st) {
    visited[u] = true;

    for (int v : graph[u]) {
        if (!visited[v]) {
            topSortDFS(graph, v, visited, st);
        }
    }

    st.push(u); // Add after all descendants
}

vector&lt;int&gt; topologicalSort(vector&lt;vector&lt;int&gt;&gt;&amp; graph) {
    int n = graph.size();
    vector&lt;bool&gt; visited(n, false);
    stack&lt;int&gt; st;

    for (int i = 0; i &lt; n; i++) {
        if (!visited[i]) {
            topSortDFS(graph, i, visited, st);
        }
    }

    vector&lt;int&gt; result;
    while (!st.empty()) {
        result.push_back(st.top());
        st.pop();
    }
    return result;
}
```

5.3 Kahn's Algorithm (BFS-Based)

Approach: Remove nodes with in-degree 0 repeatedly.

```
vector&lt;int&gt; topologicalSortKahn(vector&lt;vector&lt;int&gt;&gt;&amp; graph) {
    int n = graph.size();
    vector&lt;int&gt; inDegree(n, 0);

    for (int u = 0; u &lt; n; u++) {
        for (int v : graph[u]) {
```

```
            inDegree[v]++;
        }
    }

    queue&lt;int&gt; q;
    for (int i = 0; i &lt; n; i++) {
        if (inDegree[i] == 0) q.push(i);
    }

    vector&lt;int&gt; result;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);

        for (int v : graph[u]) {
            inDegree[v]--;
            if (inDegree[v] == 0) q.push(v);
        }
    }

    // If result.size() &lt; n, cycle exists
    return result;
}
```

5.4 Problem: Course Schedule (Infosys Medium)

Statement: Given numCourses and prerequisites (course pairs), determine if all courses can be finished.

Input: numCourses = 2, prerequisites = [[1,0]]
Output: true (take course 0, then 1)

Input: numCourses = 2, prerequisites = [[1,0], [0,1]]
Output: false (cycle: 0 → 1 → 0)

Solution: Check if topological sort produces all courses.

```
bool canFinish(int numCourses, vector&lt;vector&lt;int&gt;&gt;&amp; prerequisites) {
    vector&lt;vector&lt;int&gt;&gt; graph(numCourses);

    for (auto&amp; p : prerequisites) {
        graph[p[1]].push_back(p[0]); // p[1] → p[0]
    }

    vector&lt;int&gt; sorted = topologicalSortKahn(graph);
    return sorted.size() == numCourses;
}
```

Chapter 6: Union-Find (Disjoint Set)

6.1 Concept

Union-Find: Data structure for tracking disjoint sets with operations:

- **Find**: Which set does element belong to?
- **Union**: Merge two sets

Applications:

- Detect cycles in undirected graph
- Kruskal's MST algorithm
- Network connectivity

Optimizations:

- **Path Compression**: Flatten tree during Find
- **Union by Rank**: Attach smaller tree to larger

Time: O(α(n)) per operation (α = inverse Ackermann, practically constant)

6.2 Template (C++)

```
class UnionFind {
private:
    vector&lt;int&gt; parent, rank;

public:
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i &lt; n; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Path compression
        }
        return parent[x];
    }

    bool unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) return false; // Already connected

        // Union by rank
        if (rank[rootX] &lt; rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] &gt; rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
        return true;
    }

    bool connected(int x, int y) {
        return find(x) == find(y);
    }
};
```

6.3 Problem: Number of Connected Components

Statement: Count connected components in undirected graph.

Solution: Union-Find, count remaining roots.

```
int countComponents(int n, vector&lt;vector&lt;int&gt;&gt;&amp; edges) {
    UnionFind uf(n);

    for (auto&amp; e : edges) {
        uf.unite(e[0], e[1]);
    }

    unordered_set&lt;int&gt; roots;
    for (int i = 0; i &lt; n; i++) {
        roots.insert(uf.find(i));
    }

    return roots.size();
}
```

Chapter 7: Advanced: Strongly Connected Components (SCC)

7.1 Concept

SCC: Maximal subgraph where every vertex is reachable from every other vertex.

Algorithm: Kosaraju's (two DFS passes)

1. DFS on original graph, record finish times
2. DFS on transposed graph in reverse finish order
3. Each DFS tree in step 2 is one SCC

Time: O(V + E)

7.2 Template (Kosaraju's)

```
void dfs1(vector&lt;vector&lt;int&gt;&gt;&amp; graph, int u,
          vector&lt;bool&gt;&amp; visited, stack&lt;int&gt;&amp; st) {
    visited[u] = true;
    for (int v : graph[u]) {
        if (!visited[v]) dfs1(graph, v, visited, st);
    }
    st.push(u);
}

void dfs2(vector&lt;vector&lt;int&gt;&gt;&amp; graphT, int u,
          vector&lt;bool&gt;&amp; visited, vector&lt;int&gt;&amp; component) {
    visited[u] = true;
    component.push_back(u);
    for (int v : graphT[u]) {
        if (!visited[v]) dfs2(graphT, v, visited, component);
    }
}

vector&lt;vector&lt;int&gt;&gt; findSCCs(vector&lt;vector&lt;int&gt;&gt;&amp; graph) {
    int n = graph.size();

    // Step 1: DFS and stack
```

```
    vector&lt;bool&gt; visited(n, false);
    stack&lt;int&gt; st;
    for (int i = 0; i &lt; n; i++) {
        if (!visited[i]) dfs1(graph, i, visited, st);
    }

    // Step 2: Transpose graph
    vector&lt;vector&lt;int&gt;&gt; graphT(n);
    for (int u = 0; u &lt; n; u++) {
        for (int v : graph[u]) {
            graphT[v].push_back(u);
        }
    }

    // Step 3: DFS on transpose
    fill(visited.begin(), visited.end(), false);
    vector&lt;vector&lt;int&gt;&gt; sccs;

    while (!st.empty()) {
        int u = st.top();
        st.pop();
        if (!visited[u]) {
            vector&lt;int&gt; component;
            dfs2(graphT, u, visited, component);
            sccs.push_back(component);
        }
    }

    return sccs;
}
```

Practice Problem Set

Easy (BFS/DFS Basics)

1. Clone Graph
2. Flood Fill
3. Max Area of Island
4. Surrounded Regions
5. Pacific Atlantic Water Flow

Medium (Shortest Path, Topological Sort)

6. Course Schedule
7. Course Schedule II
8. Network Delay Time
9. Cheapest Flights Within K Stops
10. Minimum Height Trees

Hard (Advanced)

11. Word Ladder
12. Alien Dictionary
13. Critical Connections in Network
14. Reconstruct Itinerary

15. Swim in Rising Water

References

[1] GeeksforGeeks. (2023). Graph Algorithms Cheat Sheet. https://memgraph.com/blog/graph-algorithms-cheat-sheet-for-coding-interviews

[2] InterviewCake. (2025). Breadth-First Search (BFS). https://www.interviewcake.com/concept/java/bfs

[3] CLRS. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[4] DesignGurus. (2024). Algorithm Interview Preparation. https://www.designgurus.io/answers/detail/how-to-prepare-an-algorithm-interview