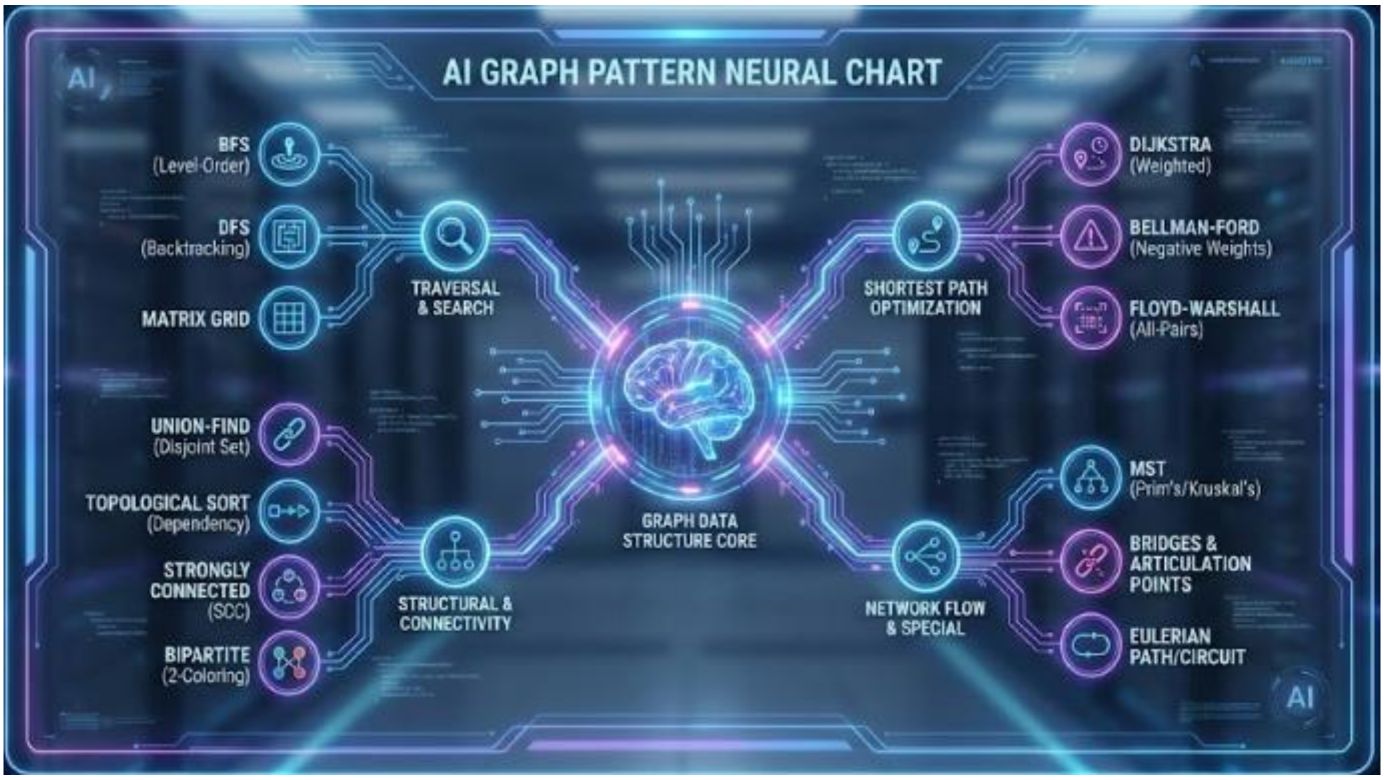
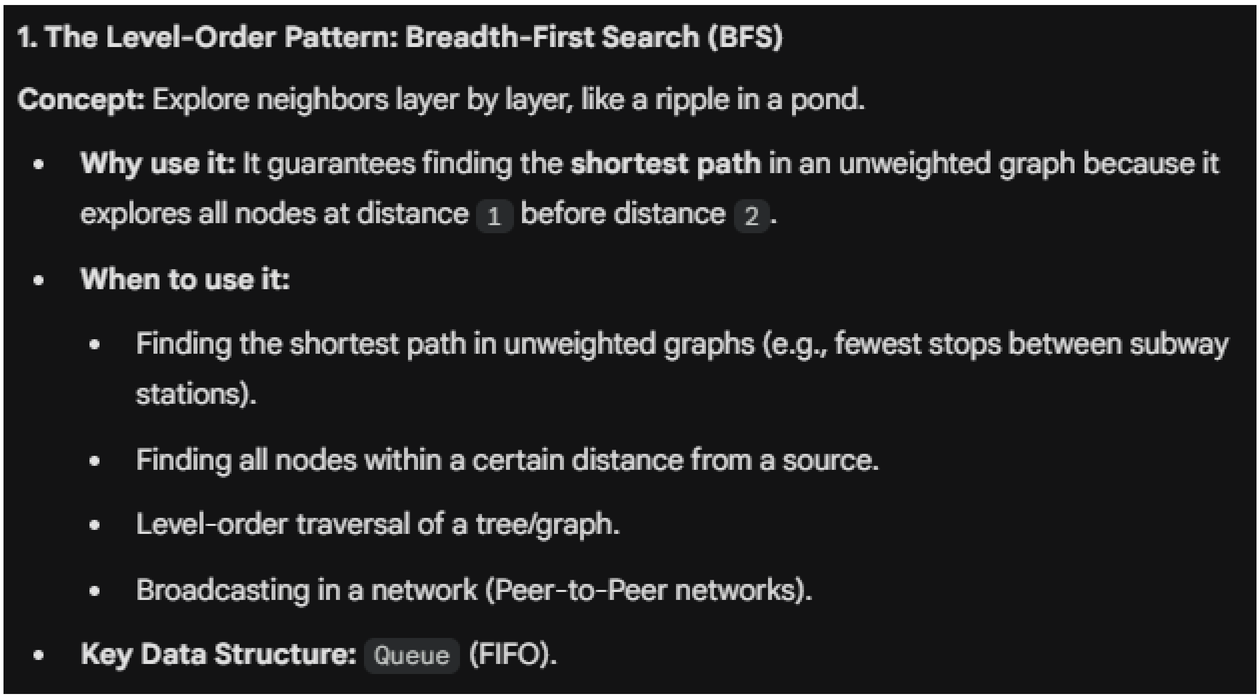
Graph Notes By Aman

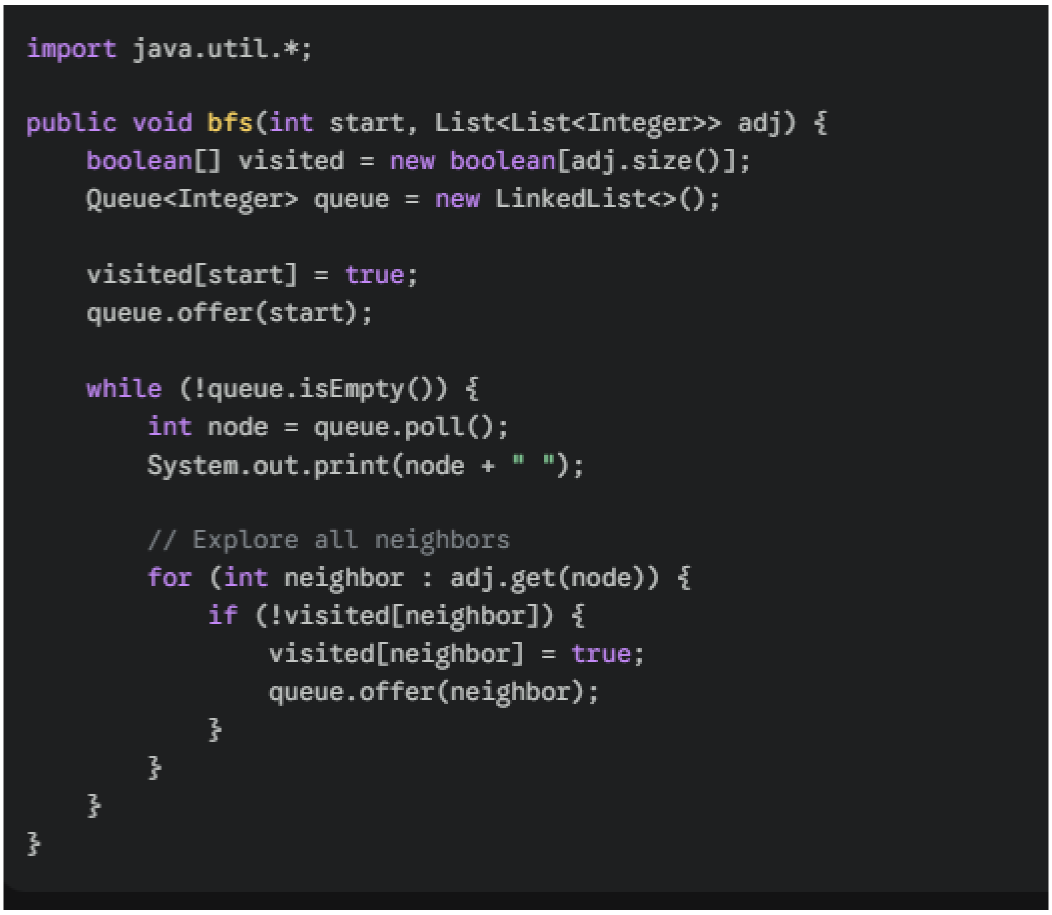

AI GRAPH PATTERN NEURAL CHART
AI
BFS (Level-Order)
DFS (Backtracking)
MATRIX GRID
TRAVERSAL & SEARCH
UNION-FIND (Disjoint Set)
TOPOLOGICAL SORT (Dependency)
STRONGLY CONNECTED (SCC)
BIPARTITE (2-Coloring)
STRUCTURAL & CONNECTIVITY
GRAPH DATA STRUCTURE CORE
SHORTEST PATH OPTIMIZATION
DIJKSTRA (Weighted)
BELLMAN-FORD (Negative Weights)
FLOYD-WARSHALL (All-Pairs)
NETWORK FLOW & SPECIAL
MST (Prim's/Kruskal's)
BRIDGES & ARTICULATION POINTS
EULERIAN PATH/CIRCUIT
AI



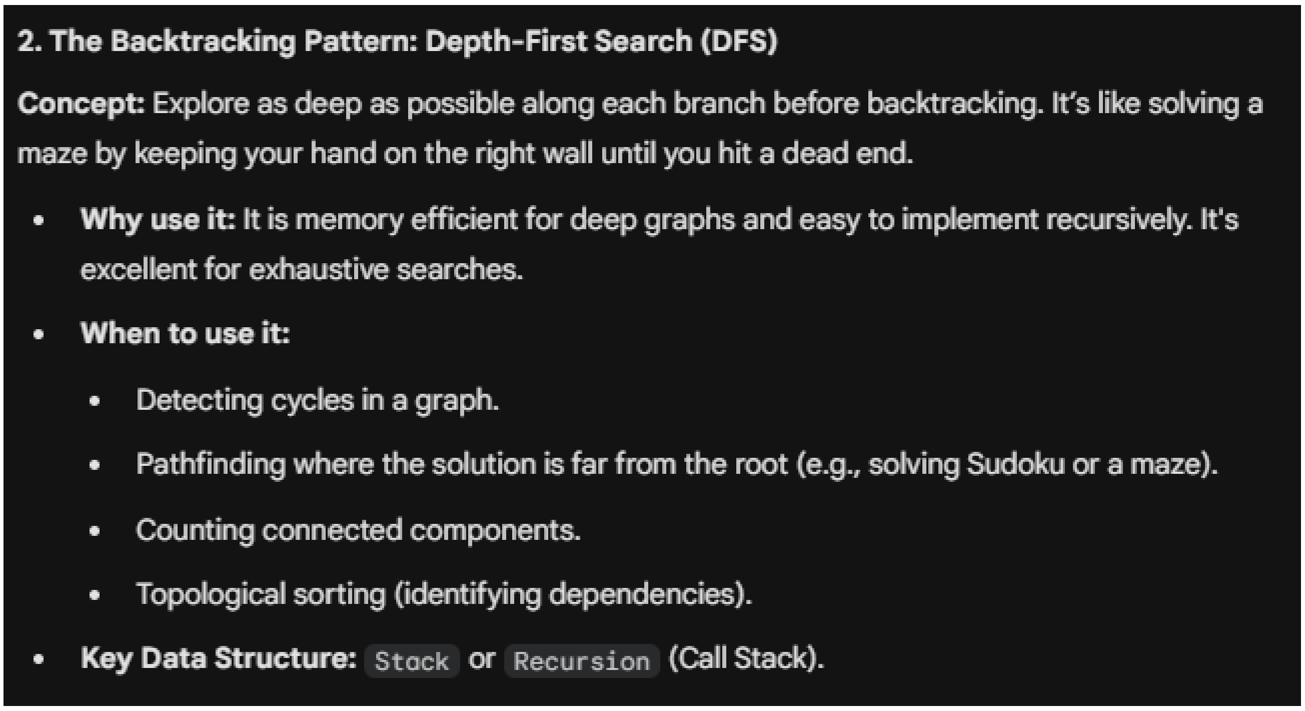
1. The Level-Order Pattern: Breadth-First Search (BFS)

Concept: Explore neighbors layer by layer, like a ripple in a pond.

- **Why use it:** It guarantees finding the **shortest path** in an unweighted graph because it explores all nodes at distance `1` before distance `2`.
- **When to use it:**
  - Finding the shortest path in unweighted graphs (e.g., fewest stops between subway stations).
  - Finding all nodes within a certain distance from a source.
  - Level-order traversal of a tree/graph.
  - Broadcasting in a network (Peer-to-Peer networks).
- **Key Data Structure:** `Queue` (FIFO).



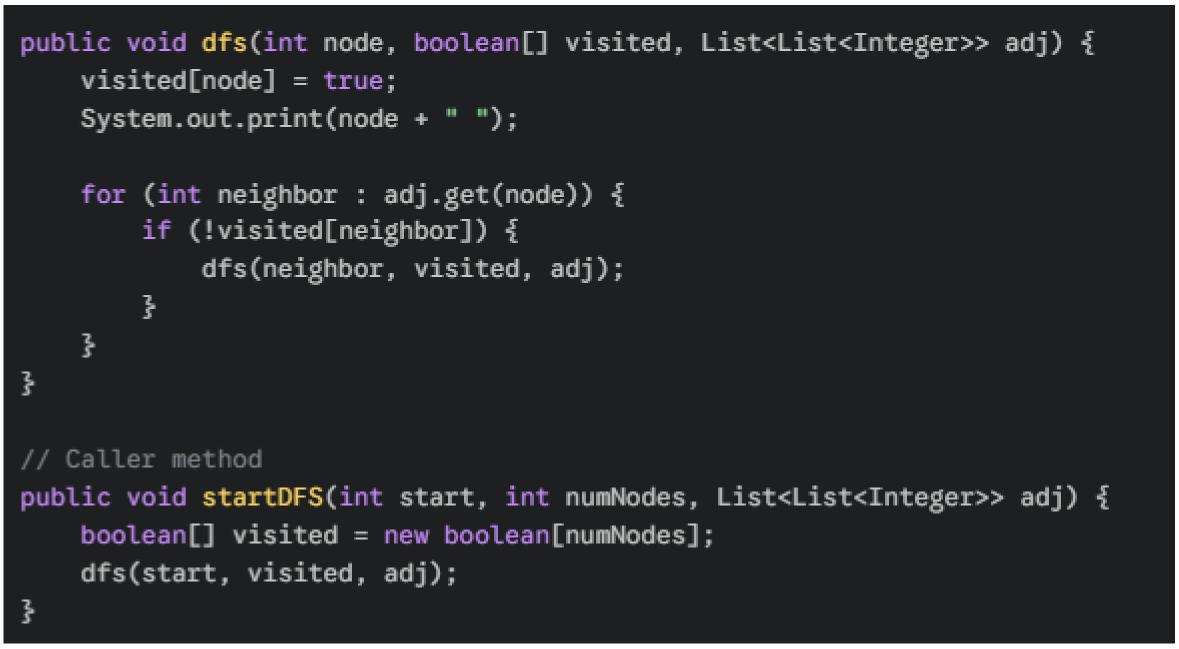
```java
import java.util.*;

public void bfs(int start, List<List<Integer>> adj) {
    boolean[] visited = new boolean[adj.size()];
    Queue<Integer> queue = new LinkedList<>();

    visited[start] = true;
    queue.offer(start);

    while (!queue.isEmpty()) {
        int node = queue.poll();
        System.out.print(node + " ");

        // Explore all neighbors
        for (int neighbor : adj.get(node)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                queue.offer(neighbor);
            }
        }
    }
}
```



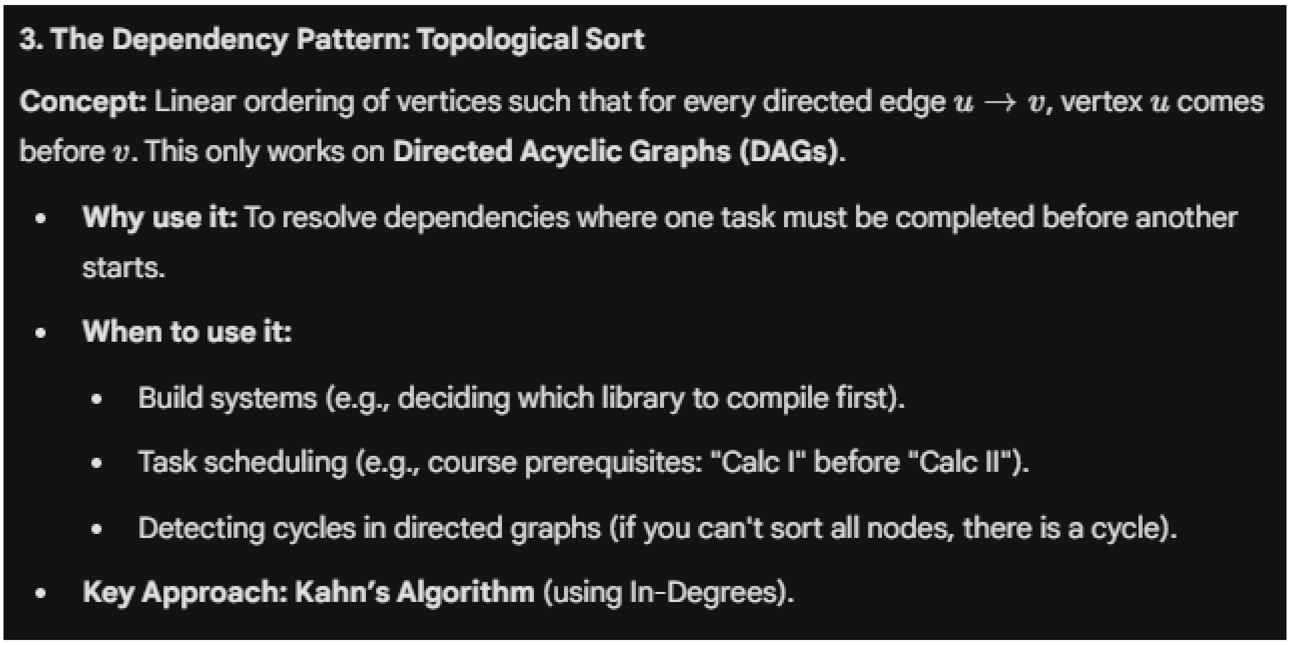
2. The Backtracking Pattern: Depth-First Search (DFS)

Concept: Explore as deep as possible along each branch before backtracking. It's like solving a maze by keeping your hand on the right wall until you hit a dead end.

- **Why use it:** It is memory efficient for deep graphs and easy to implement recursively. It's excellent for exhaustive searches.
- **When to use it:**
  - Detecting cycles in a graph.
  - Pathfinding where the solution is far from the root (e.g., solving Sudoku or a maze).
  - Counting connected components.
  - Topological sorting (identifying dependencies).
- **Key Data Structure:** `Stack` or `Recursion` (Call Stack).



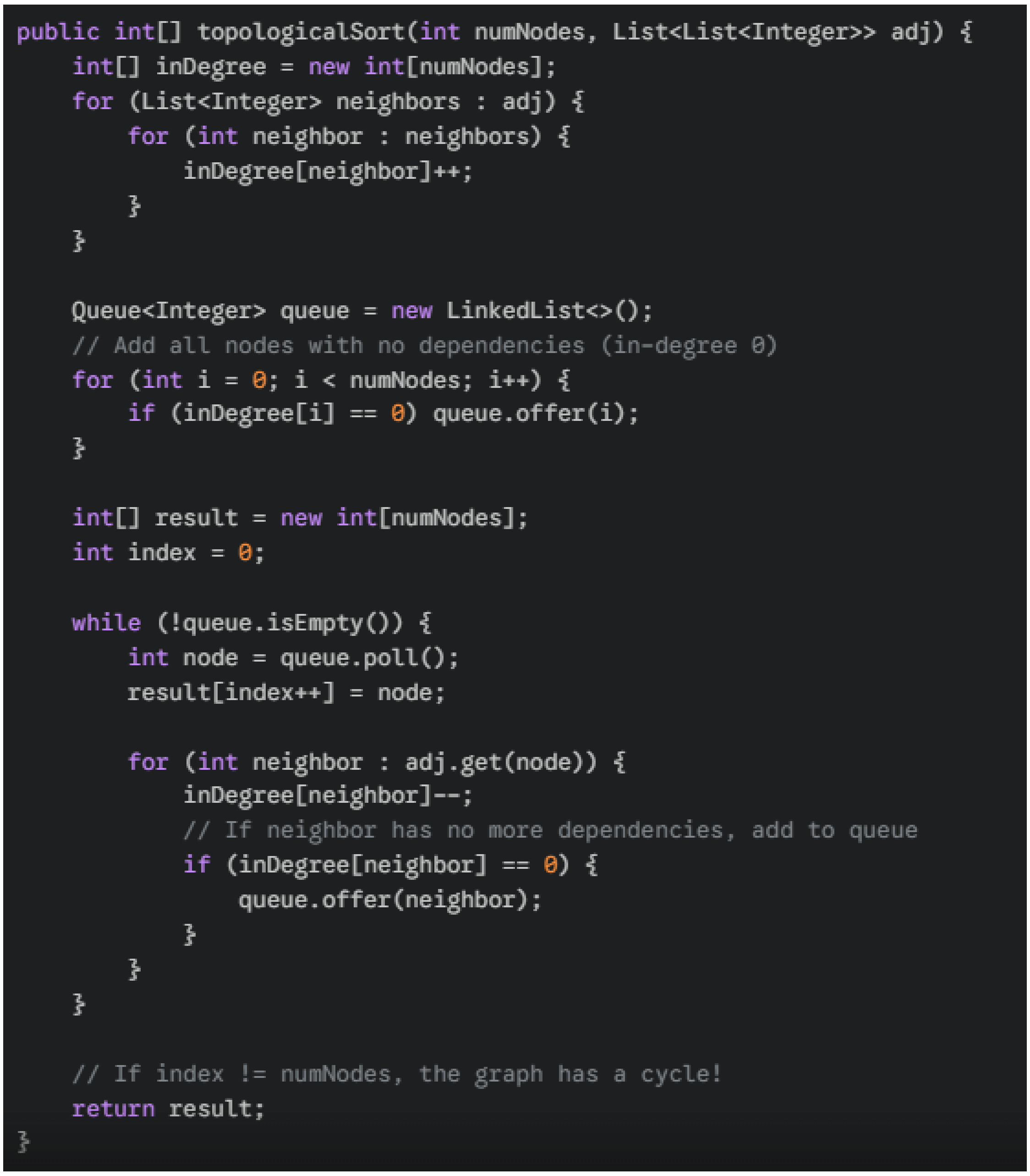
```java
public void dfs(int node, boolean[] visited, List<List<Integer>> adj) {
    visited[node] = true;
    System.out.print(node + " ");

    for (int neighbor : adj.get(node)) {
        if (!visited[neighbor]) {
            dfs(neighbor, visited, adj);
        }
    }
}

// Caller method
public void startDFS(int start, int numNodes, List<List<Integer>> adj) {
    boolean[] visited = new boolean[numNodes];
    dfs(start, visited, adj);
}
```



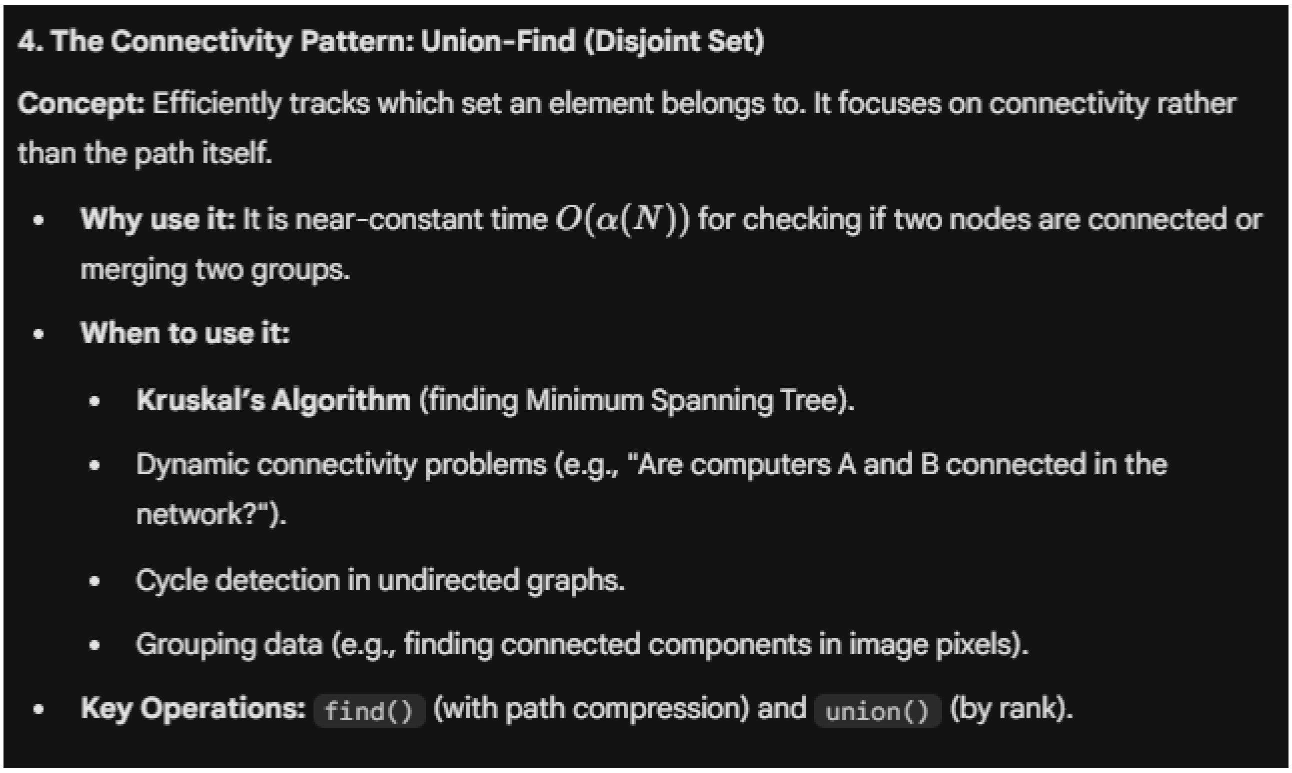
3. The Dependency Pattern: Topological Sort

Concept: Linear ordering of vertices such that for every directed edge $u \to v$, vertex u comes before v. This only works on **Directed Acyclic Graphs (DAGs)**.

- **Why use it:** To resolve dependencies where one task must be completed before another starts.
- **When to use it:**
  - Build systems (e.g., deciding which library to compile first).
  - Task scheduling (e.g., course prerequisites: "Calc I" before "Calc II").
  - Detecting cycles in directed graphs (if you can't sort all nodes, there is a cycle).
- **Key Approach: Kahn's Algorithm** (using In-Degrees).



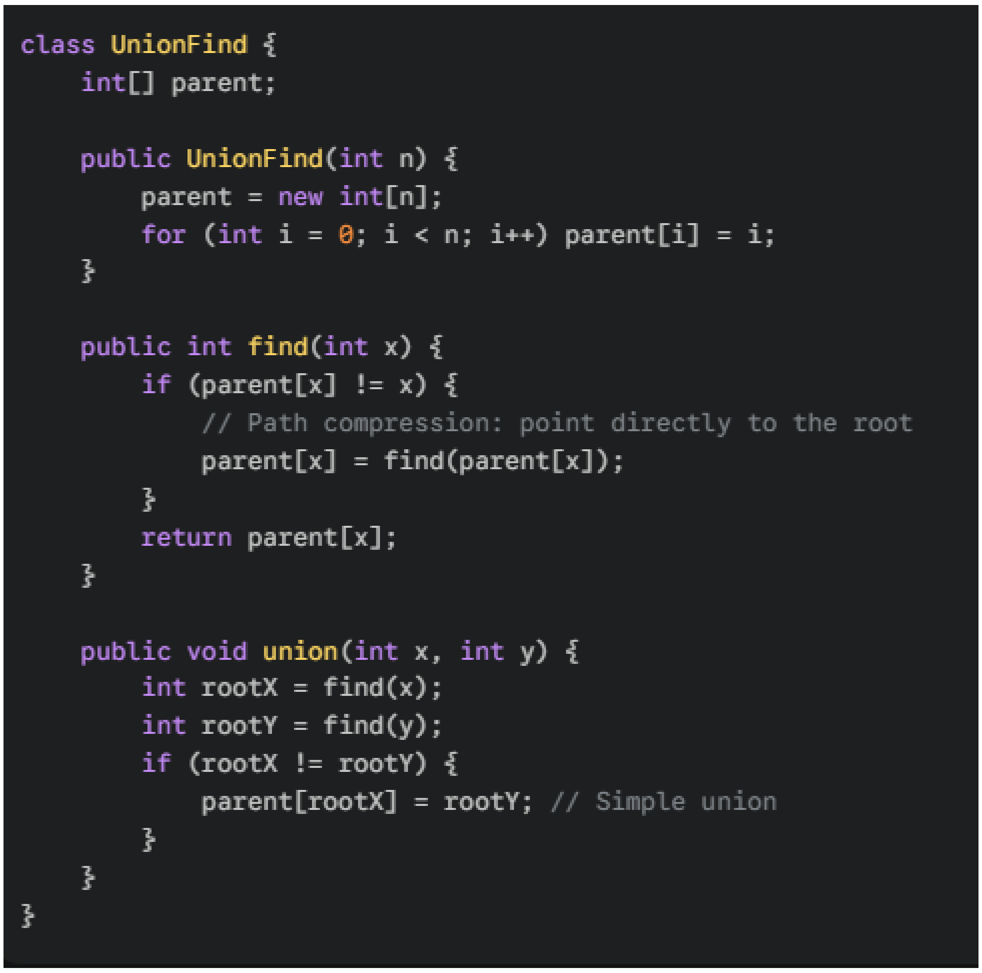
```java
public int[] topologicalSort(int numNodes, List<List<Integer>> adj) {
    int[] inDegree = new int[numNodes];
    for (List<Integer> neighbors : adj) {
        for (int neighbor : neighbors) {
            inDegree[neighbor]++;
        }
    }

    Queue<Integer> queue = new LinkedList<>();
    // Add all nodes with no dependencies (in-degree 0)
    for (int i = 0; i < numNodes; i++) {
        if (inDegree[i] == 0) queue.offer(i);
    }

    int[] result = new int[numNodes];
    int index = 0;

    while (!queue.isEmpty()) {
        int node = queue.poll();
        result[index++] = node;

        for (int neighbor : adj.get(node)) {
            inDegree[neighbor]--;
            // If neighbor has no more dependencies, add to queue
            if (inDegree[neighbor] == 0) {
                queue.offer(neighbor);
            }
        }
    }

    // If index != numNodes, the graph has a cycle!
    return result;
}
```



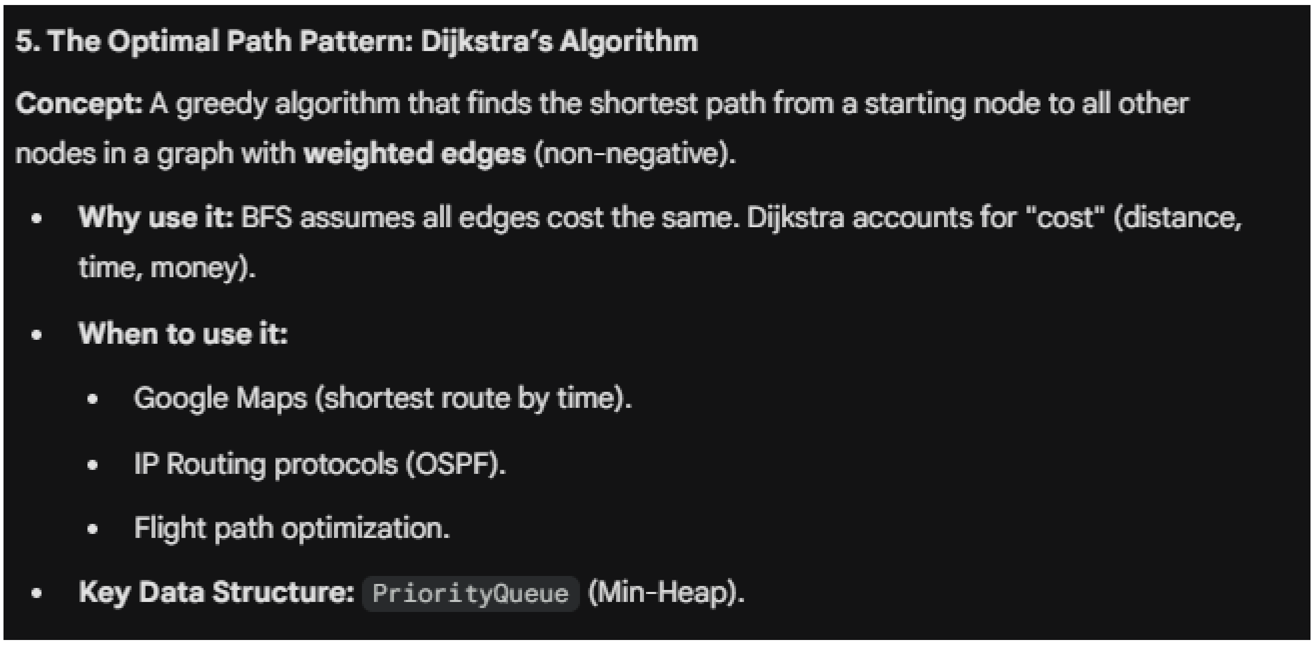
4. The Connectivity Pattern: Union-Find (Disjoint Set)

Concept: Efficiently tracks which set an element belongs to. It focuses on connectivity rather than the path itself.

- **Why use it:** It is near-constant time $O(\alpha(N))$ for checking if two nodes are connected or merging two groups.
- **When to use it:**
  - **Kruskal's Algorithm** (finding Minimum Spanning Tree).
  - Dynamic connectivity problems (e.g., "Are computers A and B connected in the network?").
  - Cycle detection in undirected graphs.
  - Grouping data (e.g., finding connected components in image pixels).
- **Key Operations:** `find()` (with path compression) and `union()` (by rank).



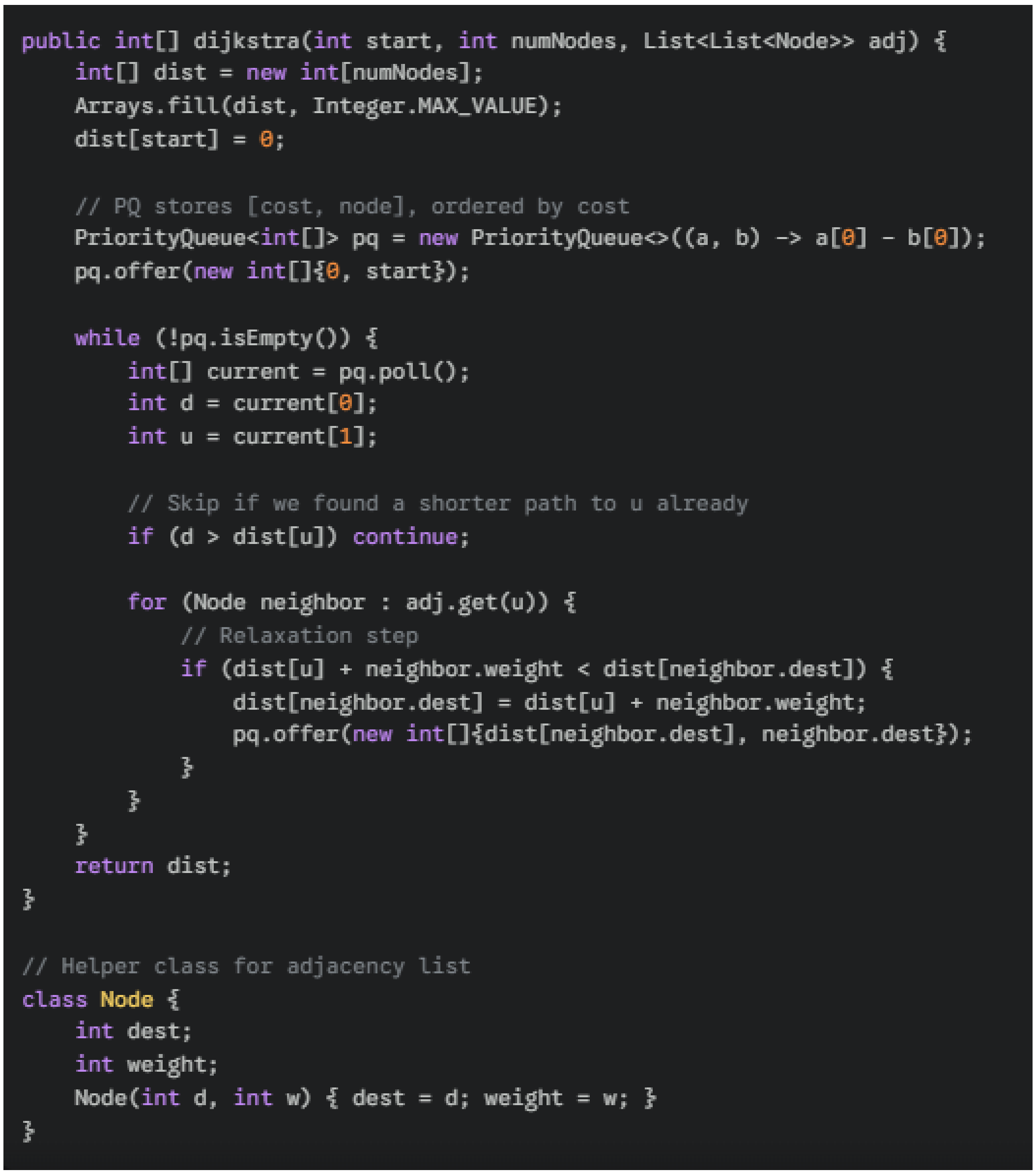
```java
class UnionFind {
    int[] parent;

    public UnionFind(int n) {
        parent = new int[n];
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    public int find(int x) {
        if (parent[x] != x) {
            // Path compression: point directly to the root
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            parent[rootX] = rootY; // Simple union
        }
    }
}
```



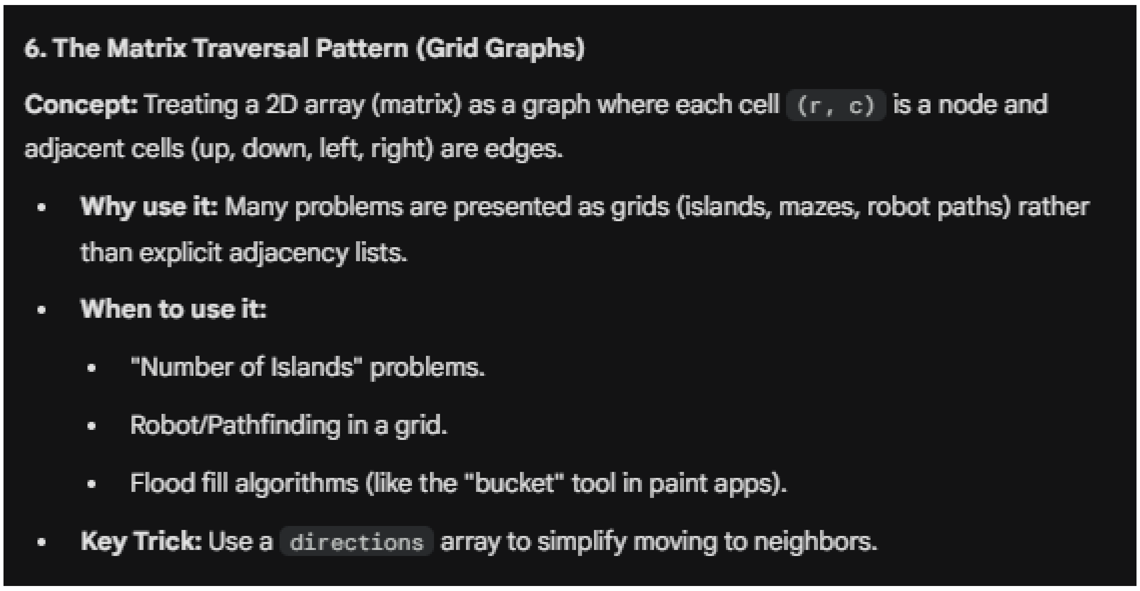
5. The Optimal Path Pattern: Dijkstra's Algorithm

Concept: A greedy algorithm that finds the shortest path from a starting node to all other nodes in a graph with **weighted edges** (non-negative).

- **Why use it:** BFS assumes all edges cost the same. Dijkstra accounts for "cost" (distance, time, money).
- **When to use it:**
  - Google Maps (shortest route by time).
  - IP Routing protocols (OSPF).
  - Flight path optimization.
- **Key Data Structure:** `PriorityQueue` (Min-Heap).



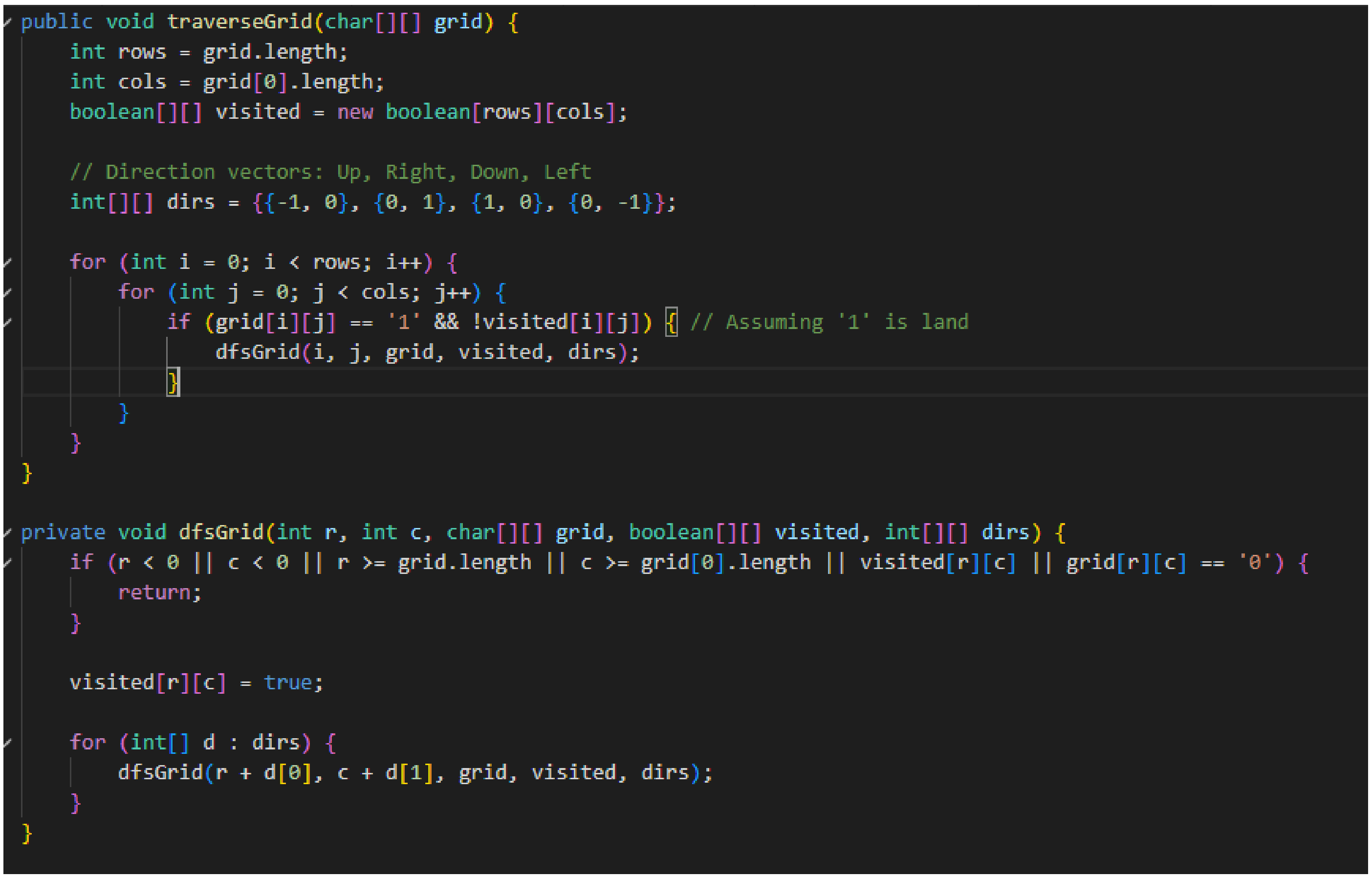
```java
public int[] dijkstra(int start, int numNodes, List<List<Node>> adj) {
    int[] dist = new int[numNodes];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[start] = 0;

    // PQ stores [cost, node], ordered by cost
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]);
    pq.offer(new int[]{0, start});

    while (!pq.isEmpty()) {
        int[] current = pq.poll();
        int d = current[0];
        int u = current[1];

        // Skip if we found a shorter path to u already
        if (d > dist[u]) continue;

        for (Node neighbor : adj.get(u)) {
            // Relaxation step
            if (dist[u] + neighbor.weight < dist[neighbor.dest]) {
                dist[neighbor.dest] = dist[u] + neighbor.weight;
                pq.offer(new int[]{dist[neighbor.dest], neighbor.dest});
            }
        }
    }
    return dist;
}

// Helper class for adjacency list
class Node {
    int dest;
    int weight;
    Node(int d, int w) { dest = d; weight = w; }
}
```



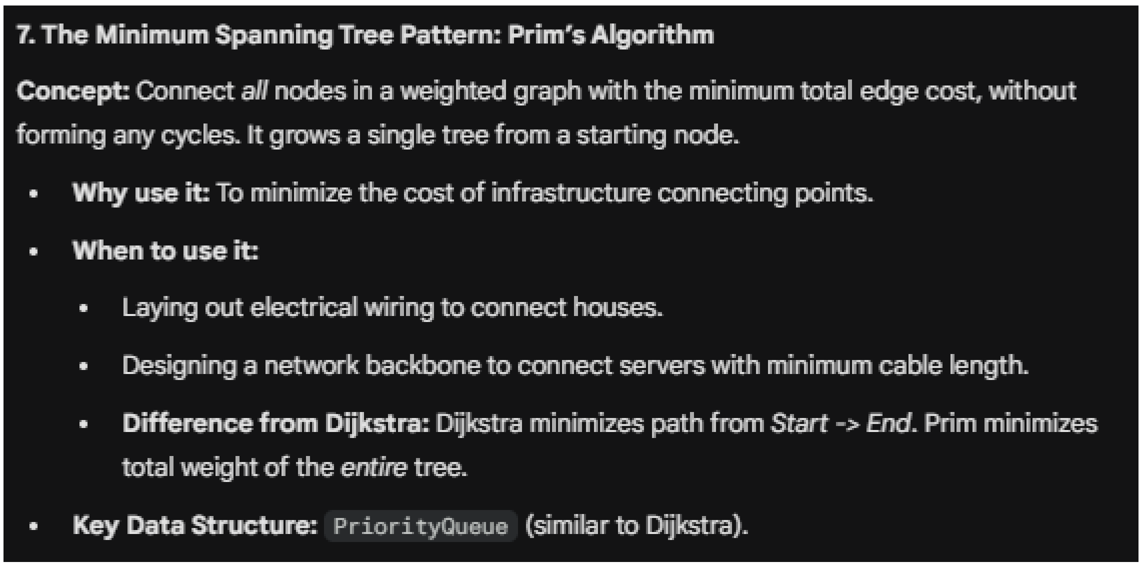
6. The Matrix Traversal Pattern (Grid Graphs)

Concept: Treating a 2D array (matrix) as a graph where each cell `(r, c)` is a node and adjacent cells (up, down, left, right) are edges.

- **Why use it:** Many problems are presented as grids (islands, mazes, robot paths) rather than explicit adjacency lists.
- **When to use it:**
  - "Number of Islands" problems.
  - Robot/Pathfinding in a grid.
  - Flood fill algorithms (like the "bucket" tool in paint apps).
- **Key Trick:** Use a `directions` array to simplify moving to neighbors.



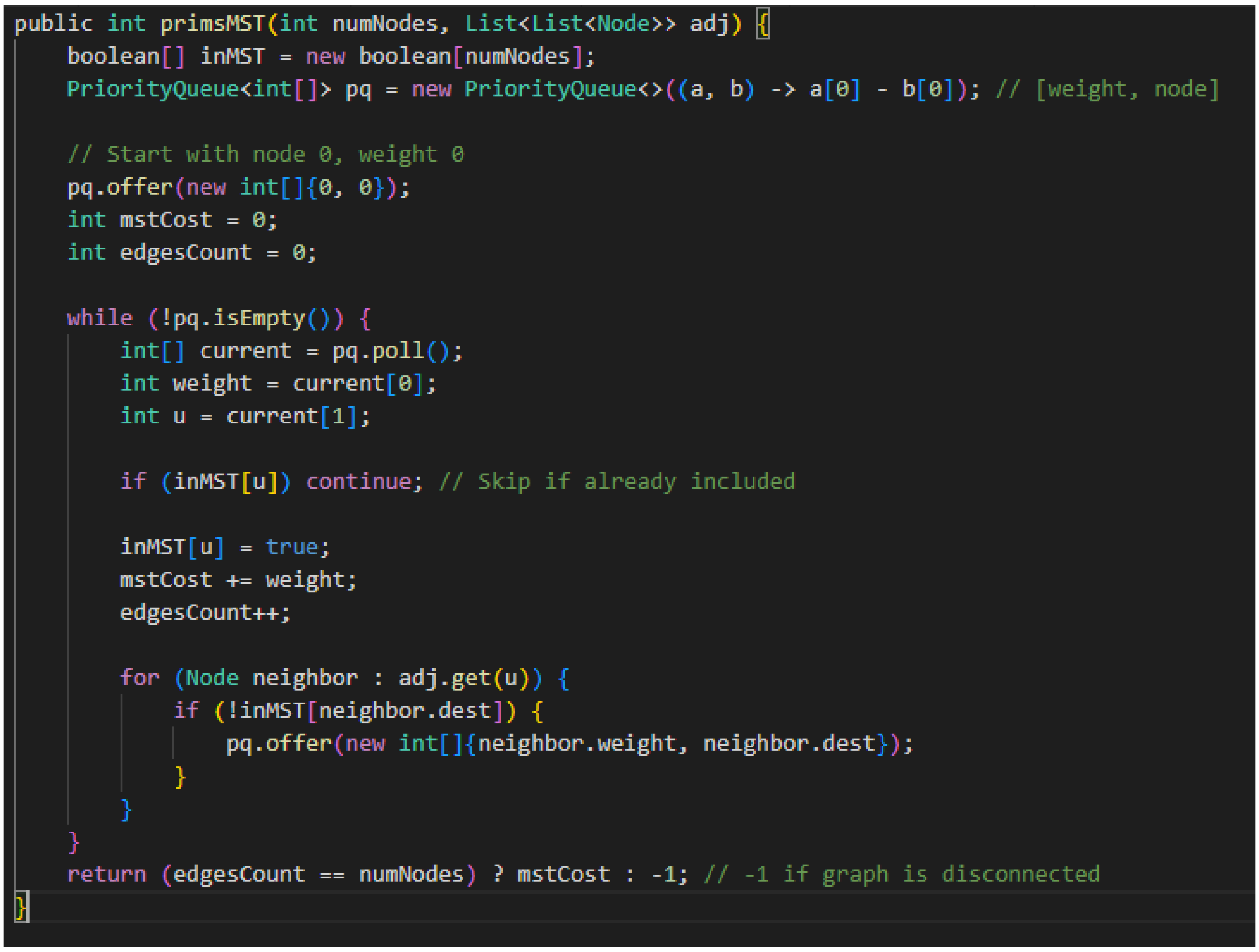
```java
public void traverseGrid(char[][] grid) {
    int rows = grid.length;
    int cols = grid[0].length;
    boolean[][] visited = new boolean[rows][cols];

    // Direction vectors: Up, Right, Down, Left
    int[][] dirs = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == '1' && !visited[i][j]) { // Assuming '1' is land
                dfsGrid(i, j, grid, visited, dirs);
            }
        }
    }
}

private void dfsGrid(int r, int c, char[][] grid, boolean[][] visited, int[][] dirs) {
    if (r < 0 || c < 0 || r >= grid.length || c >= grid[0].length || visited[r][c] || grid[r][c] == '0') {
        return;
    }

    visited[r][c] = true;

    for (int[] d : dirs) {
        dfsGrid(r + d[0], c + d[1], grid, visited, dirs);
    }
}
```



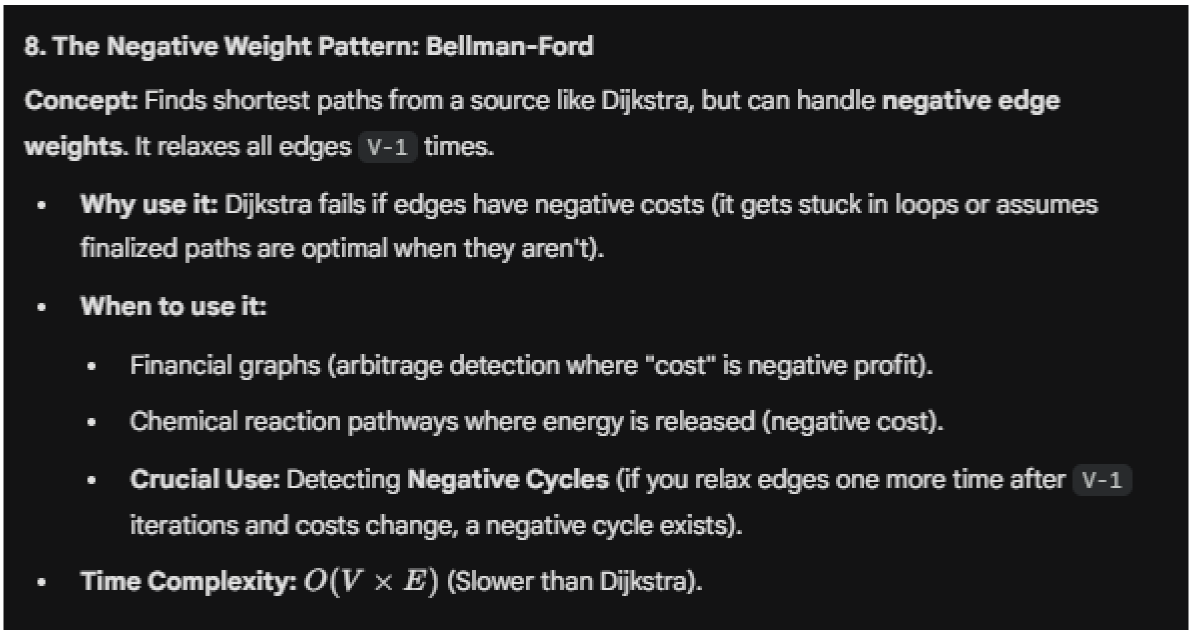
7. The Minimum Spanning Tree Pattern: Prim's Algorithm

Concept: Connect *all* nodes in a weighted graph with the minimum total edge cost, without forming any cycles. It grows a single tree from a starting node.

- **Why use it:** To minimize the cost of infrastructure connecting points.
- **When to use it:**
  - Laying out electrical wiring to connect houses.
  - Designing a network backbone to connect servers with minimum cable length.
  - **Difference from Dijkstra:** Dijkstra minimizes path from *Start* -> *End*. Prim minimizes total weight of the *entire* tree.
- **Key Data Structure:** `PriorityQueue` (similar to Dijkstra).



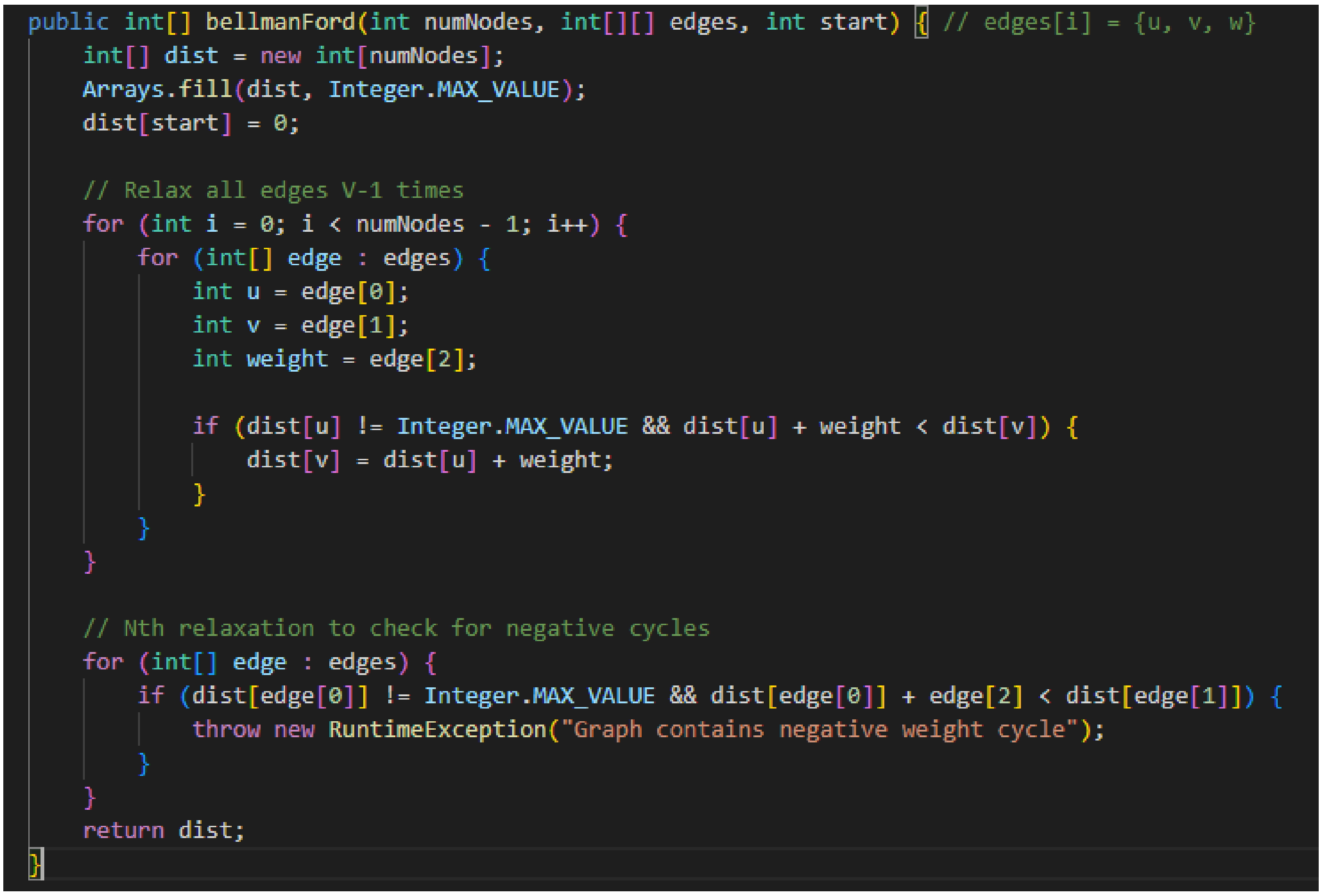
```java
public int primsMST(int numNodes, List<List<Node>> adj) {
    boolean[] inMST = new boolean[numNodes];
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]); // [weight, node]

    // Start with node 0, weight 0
    pq.offer(new int[]{0, 0});
    int mstCost = 0;
    int edgesCount = 0;

    while (!pq.isEmpty()) {
        int[] current = pq.poll();
        int weight = current[0];
        int u = current[1];

        if (inMST[u]) continue; // Skip if already included

        inMST[u] = true;
        mstCost += weight;
        edgesCount++;

        for (Node neighbor : adj.get(u)) {
            if (!inMST[neighbor.dest]) {
                pq.offer(new int[]{neighbor.weight, neighbor.dest});
            }
        }
    }
    return (edgesCount == numNodes) ? mstCost : -1; // -1 if graph is disconnected
}
```



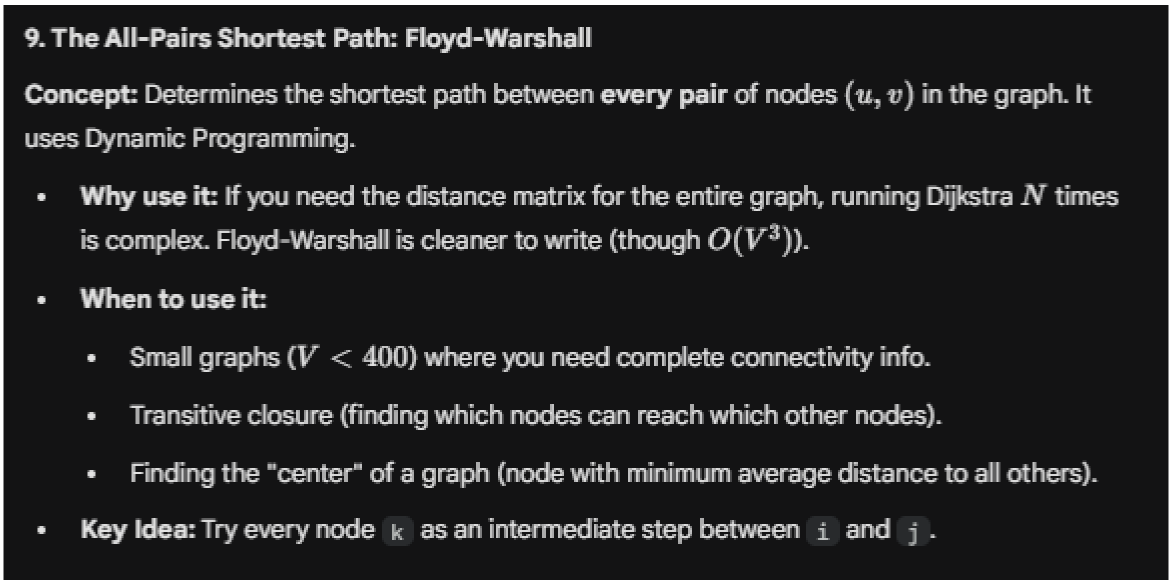
8. The Negative Weight Pattern: Bellman-Ford

Concept: Finds shortest paths from a source like Dijkstra, but can handle **negative edge weights**. It relaxes all edges `V-1` times.

- **Why use it:** Dijkstra fails if edges have negative costs (it gets stuck in loops or assumes finalized paths are optimal when they aren't).
- **When to use it:**
  - Financial graphs (arbitrage detection where "cost" is negative profit).
  - Chemical reaction pathways where energy is released (negative cost).
  - **Crucial Use:** Detecting **Negative Cycles** (if you relax edges one more time after `V-1` iterations and costs change, a negative cycle exists).
- **Time Complexity:** $O(V \times E)$ (Slower than Dijkstra).



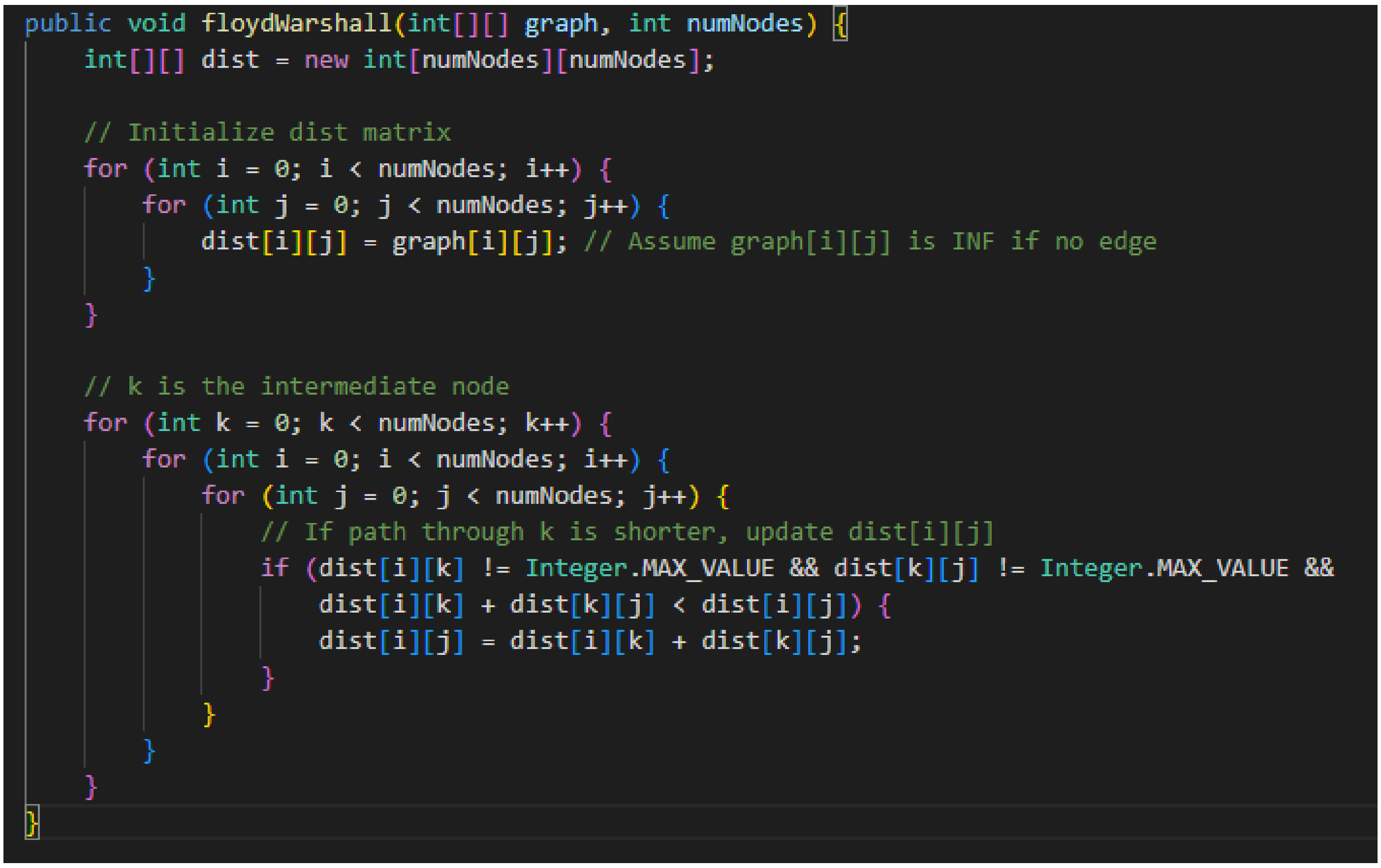
```java
public int[] bellmanFord(int numNodes, int[][] edges, int start) { // edges[i] = {u, v, w}
    int[] dist = new int[numNodes];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[start] = 0;

    // Relax all edges V-1 times
    for (int i = 0; i < numNodes - 1; i++) {
        for (int[] edge : edges) {
            int u = edge[0];
            int v = edge[1];
            int weight = edge[2];

            if (dist[u] != Integer.MAX_VALUE && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    // Nth relaxation to check for negative cycles
    for (int[] edge : edges) {
        if (dist[edge[0]] != Integer.MAX_VALUE && dist[edge[0]] + edge[2] < dist[edge[1]]) {
            throw new RuntimeException("Graph contains negative weight cycle");
        }
    }
    return dist;
}
```



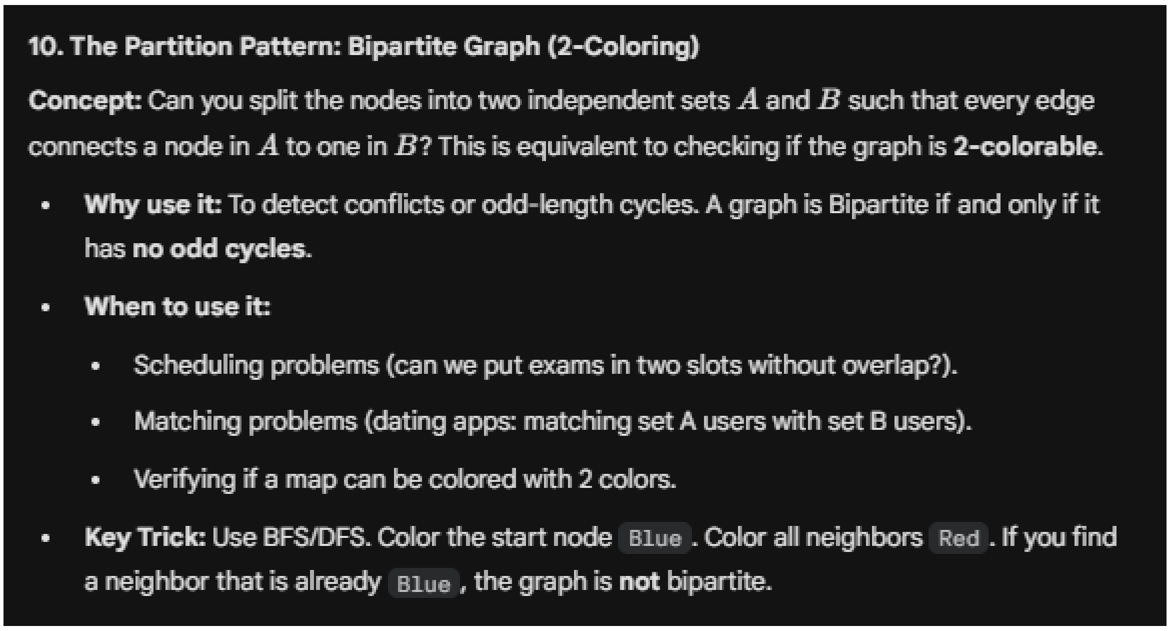
9. The All-Pairs Shortest Path: Floyd-Warshall

Concept: Determines the shortest path between **every pair** of nodes (u, v) in the graph. It uses Dynamic Programming.

- **Why use it:** If you need the distance matrix for the entire graph, running Dijkstra N times is complex. Floyd-Warshall is cleaner to write (though $O(V^3)$).
- **When to use it:**
  - Small graphs ($V < 400$) where you need complete connectivity info.
  - Transitive closure (finding which nodes can reach which other nodes).
  - Finding the "center" of a graph (node with minimum average distance to all others).
- **Key Idea:** Try every node `k` as an intermediate step between `i` and `j`.



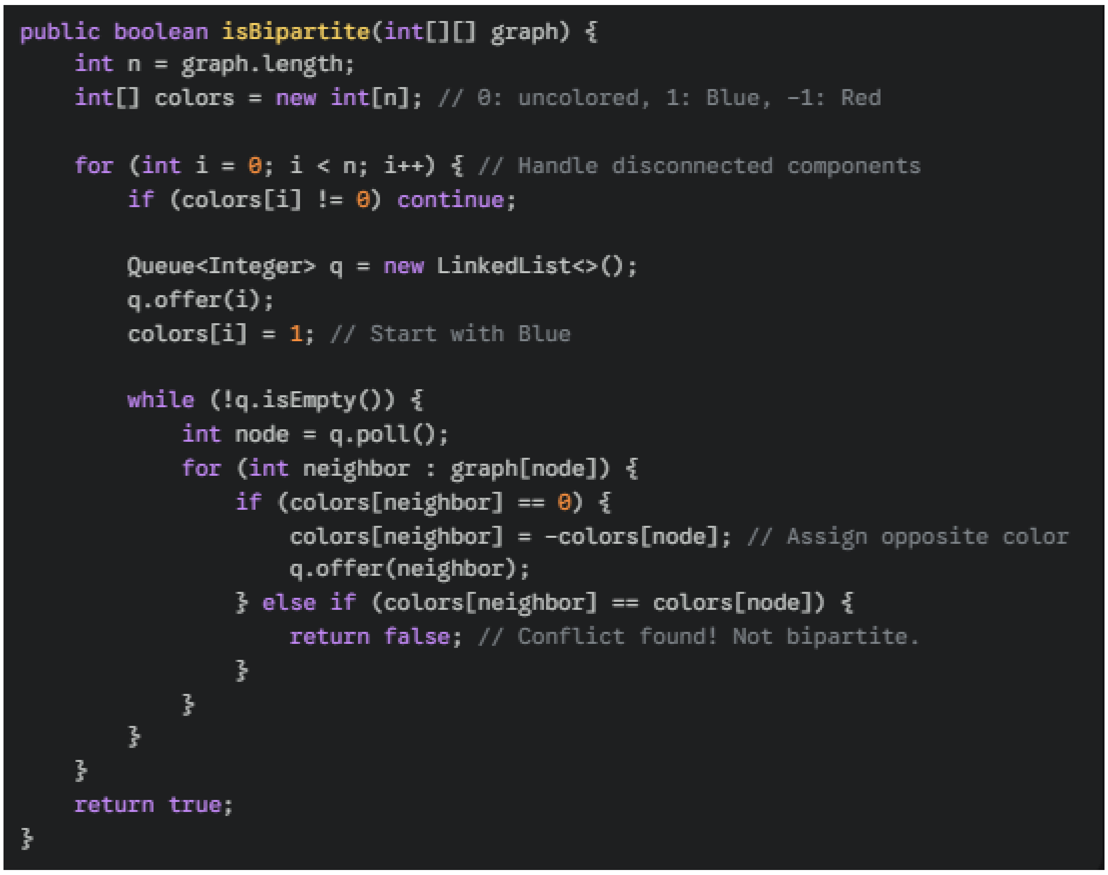
```java
public void floydWarshall(int[][] graph, int numNodes) {
    int[][] dist = new int[numNodes][numNodes];

    // Initialize dist matrix
    for (int i = 0; i < numNodes; i++) {
        for (int j = 0; j < numNodes; j++) {
            dist[i][j] = graph[i][j]; // Assume graph[i][j] is INF if no edge
        }
    }

    // k is the intermediate node
    for (int k = 0; k < numNodes; k++) {
        for (int i = 0; i < numNodes; i++) {
            for (int j = 0; j < numNodes; j++) {
                // If path through k is shorter, update dist[i][j]
                if (dist[i][k] != Integer.MAX_VALUE && dist[k][j] != Integer.MAX_VALUE &&
                    dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}
```



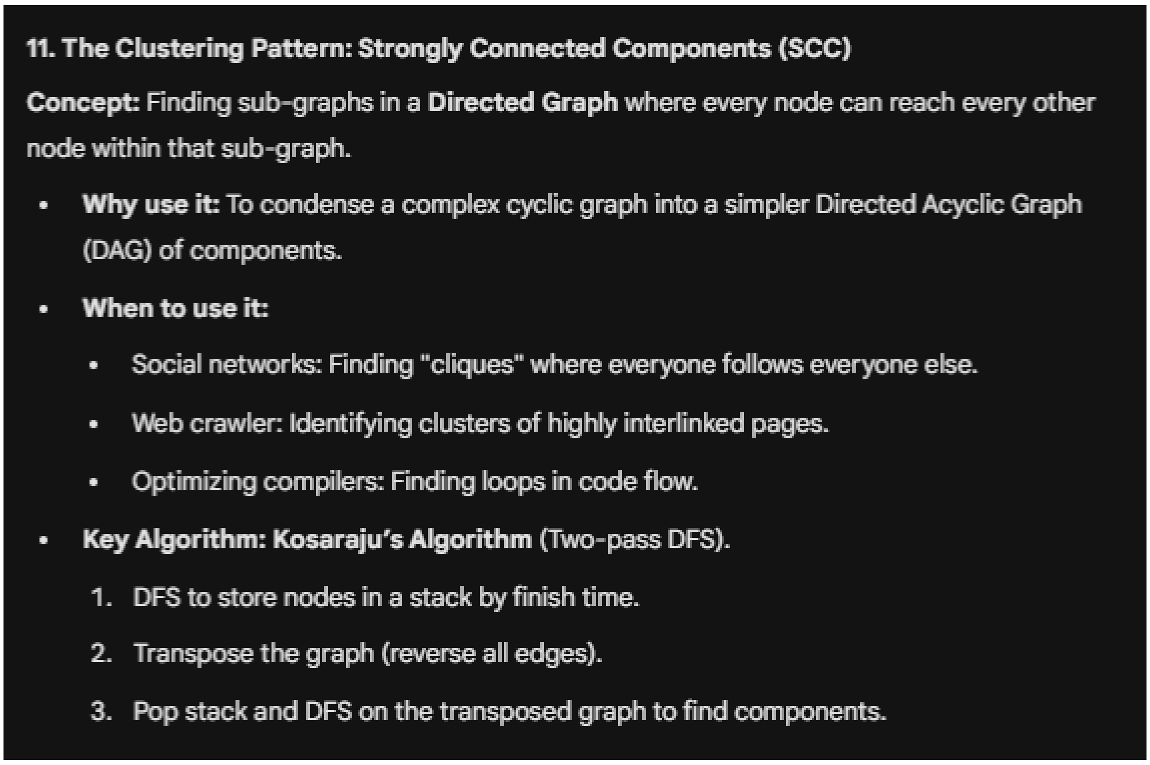
10. The Partition Pattern: Bipartite Graph (2-Coloring)

Concept: Can you split the nodes into two independent sets A and B such that every edge connects a node in A to one in B? This is equivalent to checking if the graph is **2-colorable**.

- **Why use it:** To detect conflicts or odd-length cycles. A graph is Bipartite if and only if it has **no odd cycles**.
- **When to use it:**
  - Scheduling problems (can we put exams in two slots without overlap?).
  - Matching problems (dating apps: matching set A users with set B users).
  - Verifying if a map can be colored with 2 colors.
- **Key Trick:** Use BFS/DFS. Color the start node `Blue`. Color all neighbors `Red`. If you find a neighbor that is already `Blue`, the graph is **not** bipartite.



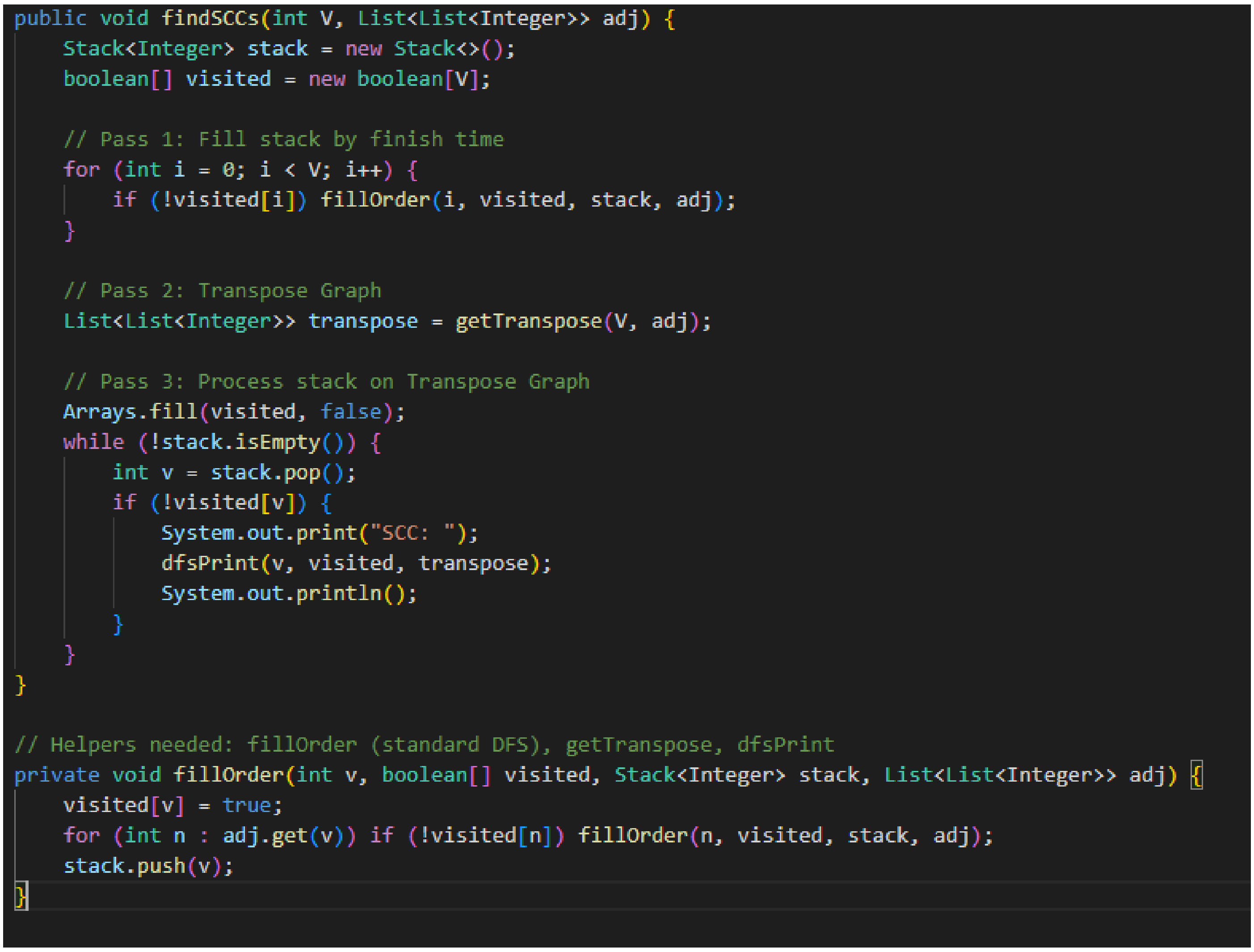
```java
public boolean isBipartite(int[][] graph) {
    int n = graph.length;
    int[] colors = new int[n]; // 0: uncolored, 1: Blue, -1: Red

    for (int i = 0; i < n; i++) { // Handle disconnected components
        if (colors[i] != 0) continue;

        Queue<Integer> q = new LinkedList<>();
        q.offer(i);
        colors[i] = 1; // Start with Blue

        while (!q.isEmpty()) {
            int node = q.poll();
            for (int neighbor : graph[node]) {
                if (colors[neighbor] == 0) {
                    colors[neighbor] = -colors[node]; // Assign opposite color
                    q.offer(neighbor);
                } else if (colors[neighbor] == colors[node]) {
                    return false; // Conflict found! Not bipartite.
                }
            }
        }
    }
    return true;
}
```



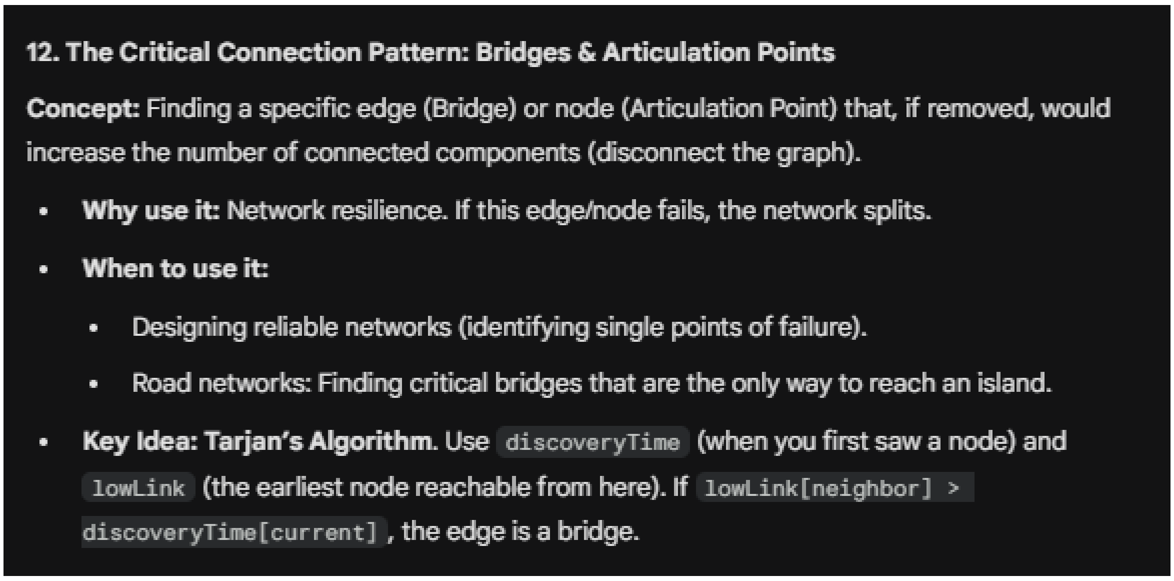
11. The Clustering Pattern: Strongly Connected Components (SCC)

Concept: Finding sub-graphs in a **Directed Graph** where every node can reach every other node within that sub-graph.

- **Why use it:** To condense a complex cyclic graph into a simpler Directed Acyclic Graph (DAG) of components.
- **When to use it:**
  - Social networks: Finding "cliques" where everyone follows everyone else.
  - Web crawler: Identifying clusters of highly interlinked pages.
  - Optimizing compilers: Finding loops in code flow.
- **Key Algorithm: Kosaraju's Algorithm** (Two-pass DFS).
  1. DFS to store nodes in a stack by finish time.
  2. Transpose the graph (reverse all edges).
  3. Pop stack and DFS on the transposed graph to find components.



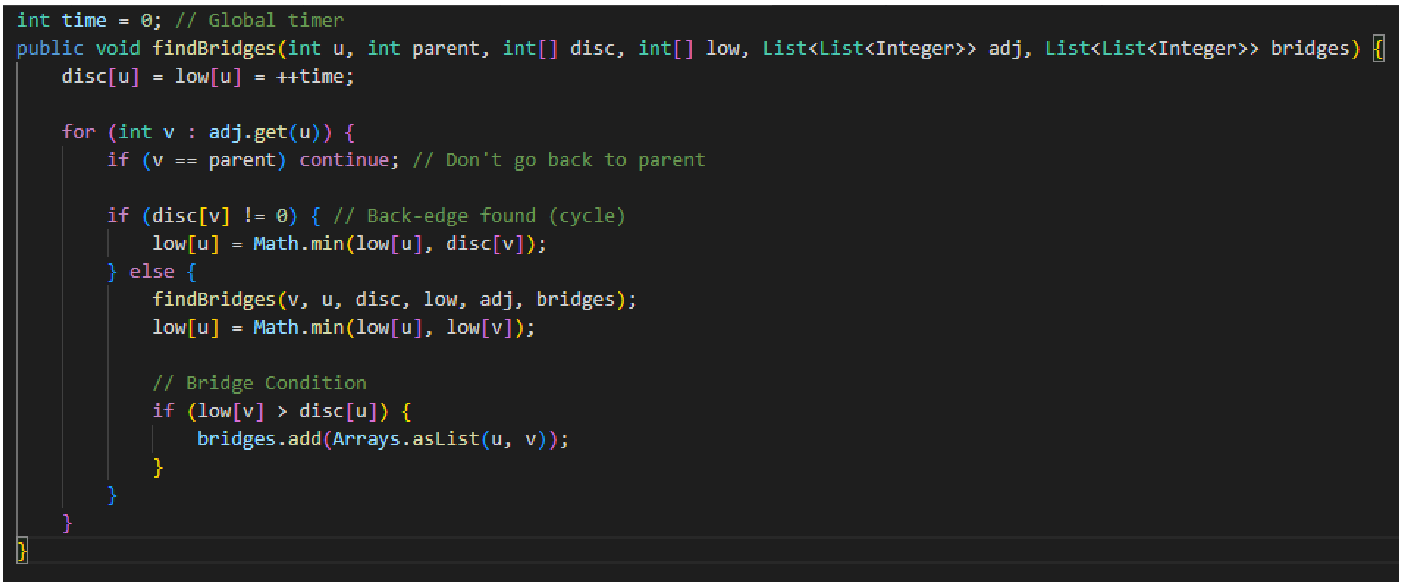
```java
public void findSCCs(int V, List<List<Integer>> adj) {
    Stack<Integer> stack = new Stack<>();
    boolean[] visited = new boolean[V];

    // Pass 1: Fill stack by finish time
    for (int i = 0; i < V; i++) {
        if (!visited[i]) fillOrder(i, visited, stack, adj);
    }

    // Pass 2: Transpose Graph
    List<List<Integer>> transpose = getTranspose(V, adj);

    // Pass 3: Process stack on Transpose Graph
    Arrays.fill(visited, false);
    while (!stack.isEmpty()) {
        int v = stack.pop();
        if (!visited[v]) {
            System.out.print("SCC: ");
            dfsPrint(v, visited, transpose);
            System.out.println();
        }
    }
}

// Helpers needed: fillOrder (standard DFS), getTranspose, dfsPrint
private void fillOrder(int v, boolean[] visited, Stack<Integer> stack, List<List<Integer>> adj) {
    visited[v] = true;
    for (int n : adj.get(v)) if (!visited[n]) fillOrder(n, visited, stack, adj);
    stack.push(v);
}
```



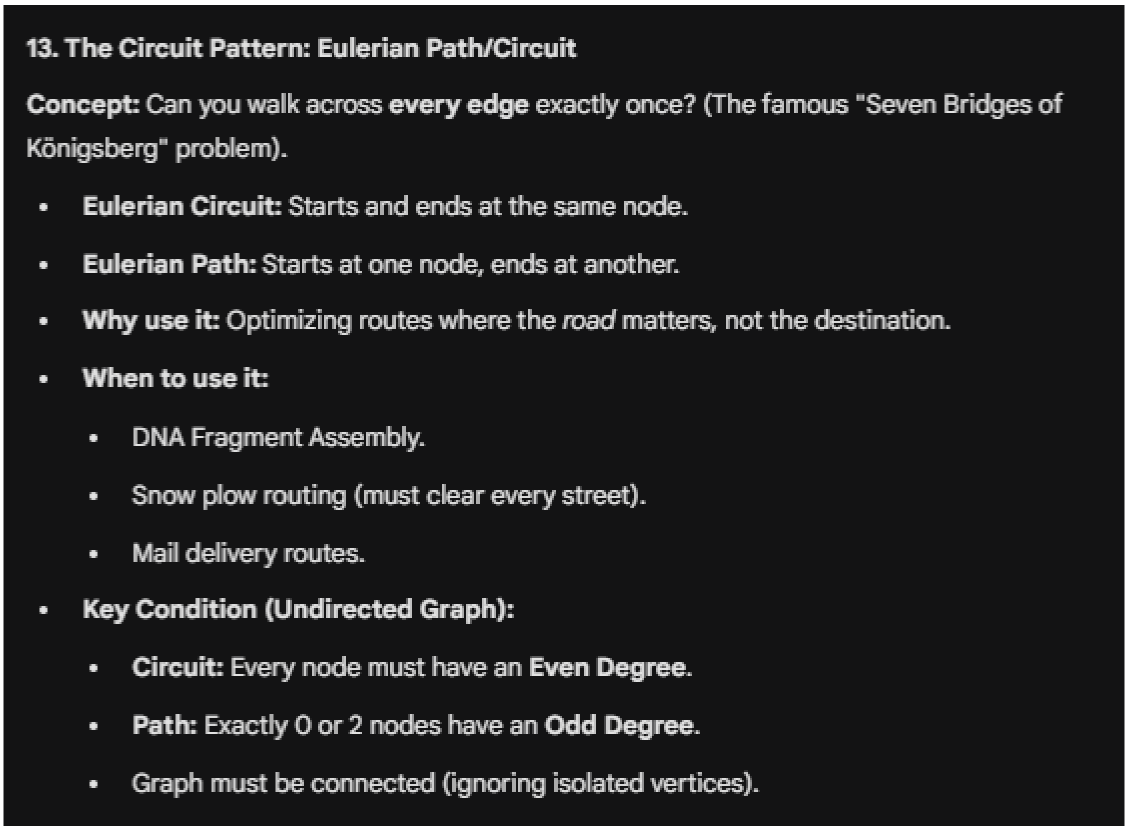
12. The Critical Connection Pattern: Bridges & Articulation Points

Concept: Finding a specific edge (Bridge) or node (Articulation Point) that, if removed, would increase the number of connected components (disconnect the graph).

- **Why use it:** Network resilience. If this edge/node fails, the network splits.
- **When to use it:**
    - Designing reliable networks (identifying single points of failure).
    - Road networks: Finding critical bridges that are the only way to reach an island.
- **Key Idea: Tarjan's Algorithm.** Use `discoveryTime` (when you first saw a node) and `lowLink` (the earliest node reachable from here). If `lowLink[neighbor] > discoveryTime[current]`, the edge is a bridge.



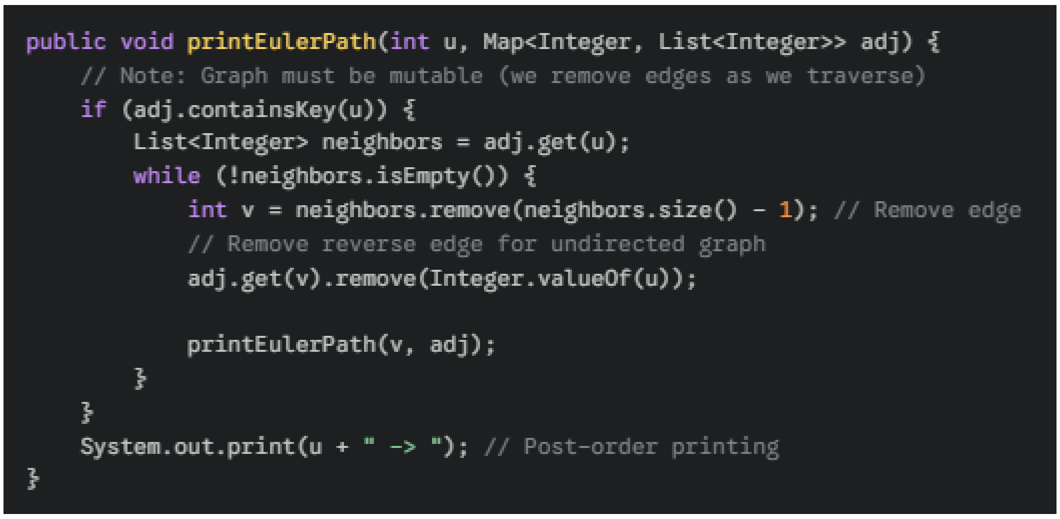
```java
int time = 0; // Global timer
public void findBridges(int u, int parent, int[] disc, int[] low, List<List<Integer>> adj, List<List<Integer>> bridges) {
    disc[u] = low[u] = ++time;

    for (int v : adj.get(u)) {
        if (v == parent) continue; // Don't go back to parent

        if (disc[v] != 0) { // Back-edge found (cycle)
            low[u] = Math.min(low[u], disc[v]);
        } else {
            findBridges(v, u, disc, low, adj, bridges);
            low[u] = Math.min(low[u], low[v]);

            // Bridge Condition
            if (low[v] > disc[u]) {
                bridges.add(Arrays.asList(u, v));
            }
        }
    }
}
```



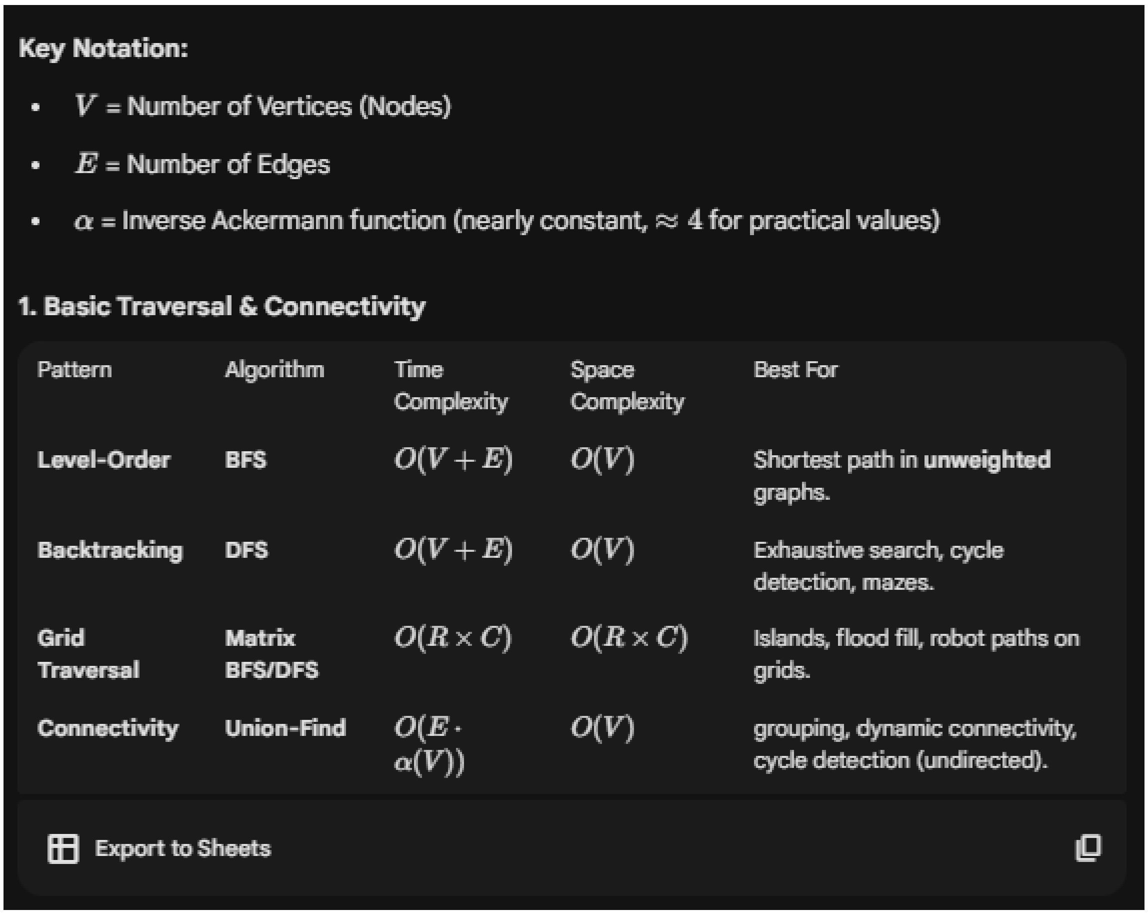
13. The Circuit Pattern: Eulerian Path/Circuit

Concept: Can you walk across **every edge** exactly once? (The famous "Seven Bridges of Königsberg" problem).

- **Eulerian Circuit:** Starts and ends at the same node.
- **Eulerian Path:** Starts at one node, ends at another.
- **Why use it:** Optimizing routes where the *road* matters, not the destination.
- **When to use it:**
  - DNA Fragment Assembly.
  - Snow plow routing (must clear every street).
  - Mail delivery routes.
- **Key Condition (Undirected Graph):**
  - **Circuit:** Every node must have an **Even Degree**.
  - **Path:** Exactly 0 or 2 nodes have an **Odd Degree**.
  - Graph must be connected (ignoring isolated vertices).



```java
public void printEulerPath(int u, Map<Integer, List<Integer>> adj) {
    // Note: Graph must be mutable (we remove edges as we traverse)
    if (adj.containsKey(u)) {
        List<Integer> neighbors = adj.get(u);
        while (!neighbors.isEmpty()) {
            int v = neighbors.remove(neighbors.size() - 1); // Remove edge
            // Remove reverse edge for undirected graph
            adj.get(v).remove(Integer.valueOf(u));

            printEulerPath(v, adj);
        }
    }
    System.out.print(u + " -> "); // Post-order printing
}
```



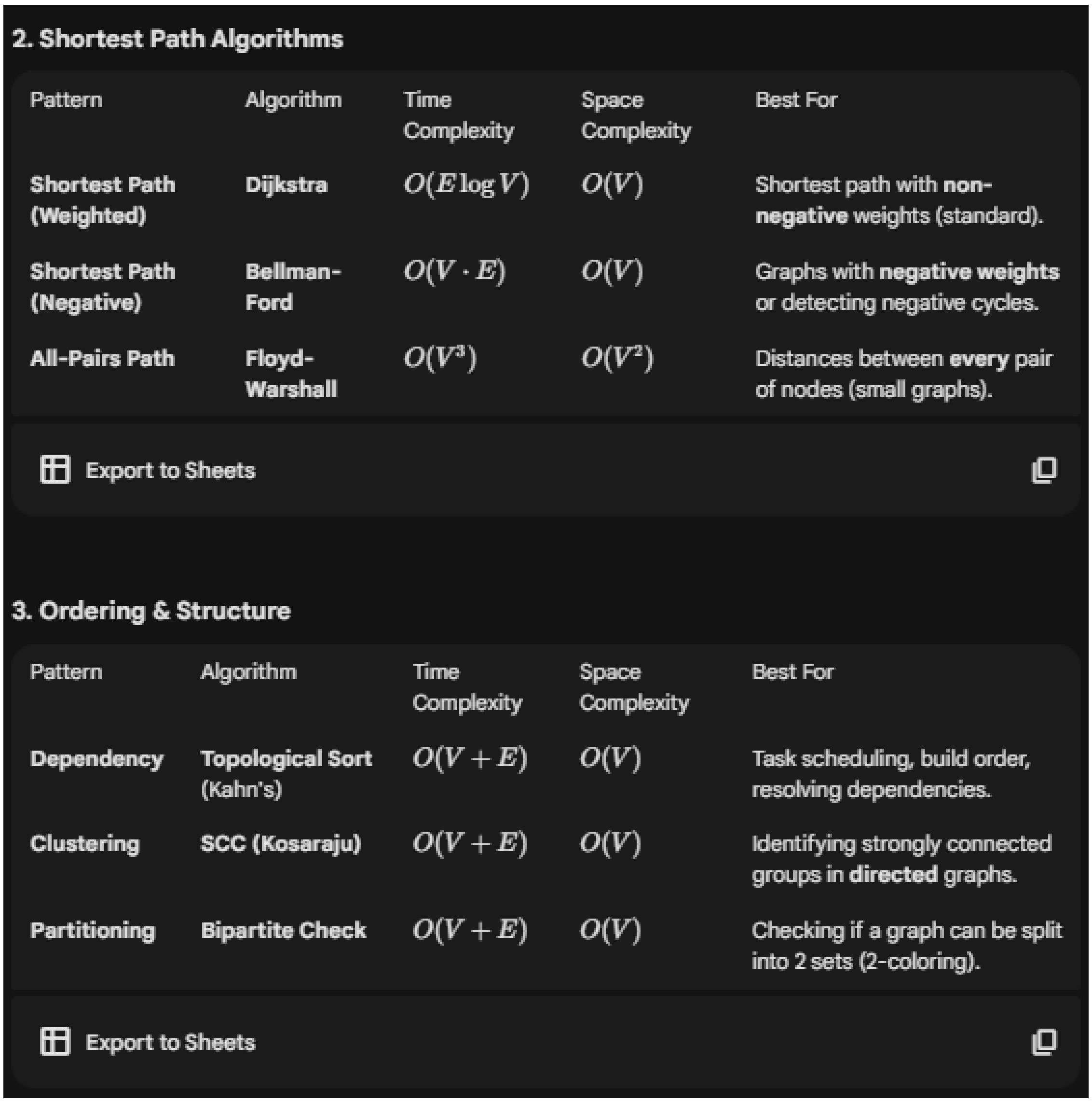
Key Notation:

- V = Number of Vertices (Nodes)
- E = Number of Edges
- α = Inverse Ackermann function (nearly constant, ≈ 4 for practical values)

1. Basic Traversal & Connectivity

Pattern	Algorithm	Time Complexity	Space Complexity	Best For
Level-Order	**BFS**	$O(V+E)$	$O(V)$	Shortest path in **unweighted** graphs.
Backtracking	**DFS**	$O(V+E)$	$O(V)$	Exhaustive search, cycle detection, mazes.
Grid Traversal	**Matrix BFS/DFS**	$O(R \times C)$	$O(R \times C)$	Islands, flood fill, robot paths on grids.
Connectivity	**Union-Find**	$O(E \cdot \alpha(V))$	$O(V)$	grouping, dynamic connectivity, cycle detection (undirected).

Export to Sheets

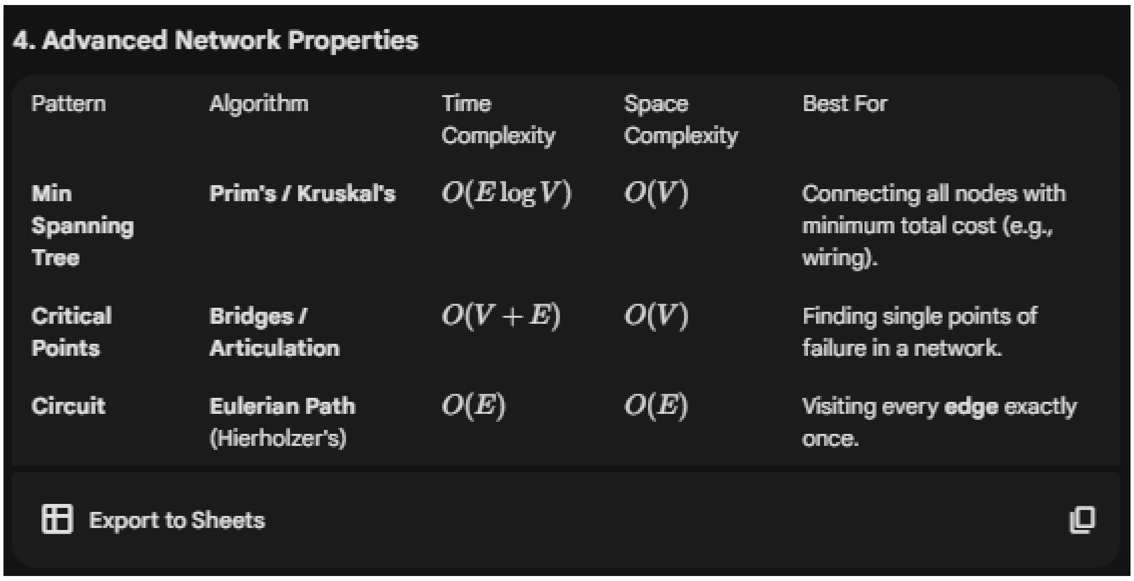


2. Shortest Path Algorithms

Pattern	Algorithm	Time Complexity	Space Complexity	Best For
Shortest Path (Weighted)	**Dijkstra**	$O(E \log V)$	$O(V)$	Shortest path with **non-negative** weights (standard).
Shortest Path (Negative)	**Bellman-Ford**	$O(V \cdot E)$	$O(V)$	Graphs with **negative weights** or detecting negative cycles.
All-Pairs Path	**Floyd-Warshall**	$O(V^3)$	$O(V^2)$	Distances between **every** pair of nodes (small graphs).

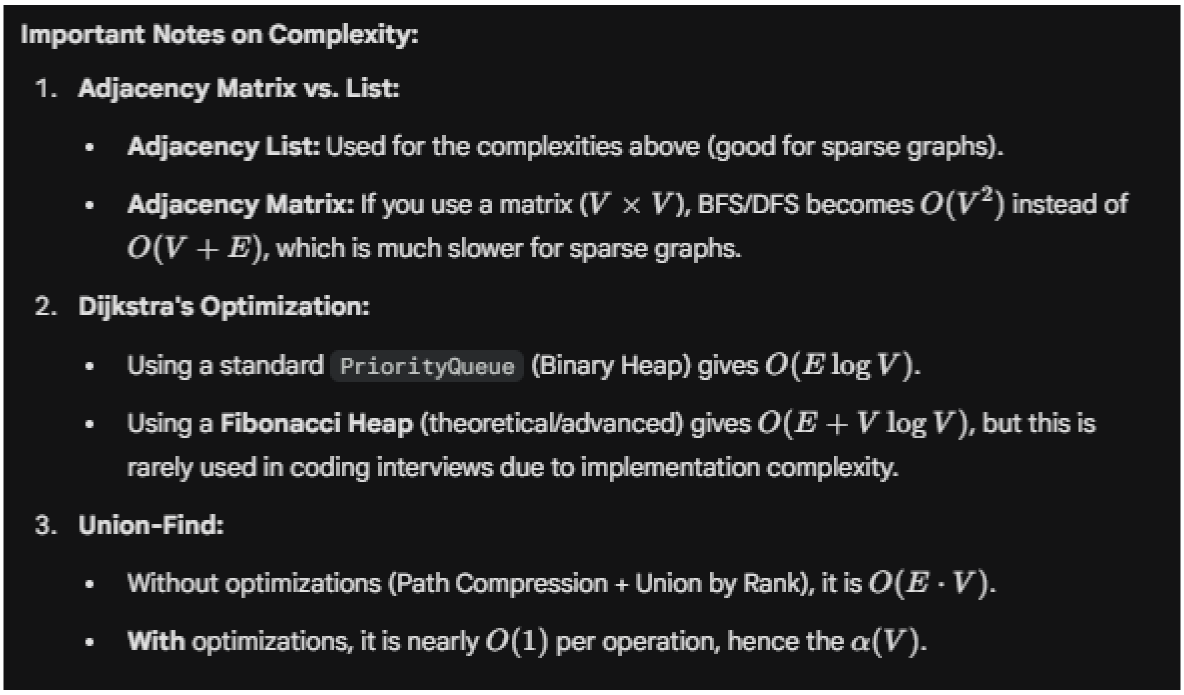
3. Ordering & Structure

Pattern	Algorithm	Time Complexity	Space Complexity	Best For
Dependency	**Topological Sort** (Kahn's)	$O(V+E)$	$O(V)$	Task scheduling, build order, resolving dependencies.
Clustering	**SCC (Kosaraju)**	$O(V+E)$	$O(V)$	Identifying strongly connected groups in **directed** graphs.
Partitioning	**Bipartite Check**	$O(V+E)$	$O(V)$	Checking if a graph can be split into 2 sets (2-coloring).



4. Advanced Network Properties

Pattern	Algorithm	Time Complexity	Space Complexity	Best For
Min Spanning Tree	**Prim's / Kruskal's**	$O(E \log V)$	$O(V)$	Connecting all nodes with minimum total cost (e.g., wiring).
Critical Points	**Bridges / Articulation**	$O(V+E)$	$O(V)$	Finding single points of failure in a network.
Circuit	**Eulerian Path** (Hierholzer's)	$O(E)$	$O(E)$	Visiting every **edge** exactly once.


Important Notes on Complexity:

1. **Adjacency Matrix vs. List:**
    - **Adjacency List:** Used for the complexities above (good for sparse graphs).
    - **Adjacency Matrix:** If you use a matrix ($V \times V$), BFS/DFS becomes $O(V^2)$ instead of $O(V + E)$, which is much slower for sparse graphs.
2. **Dijkstra's Optimization:**
    - Using a standard `PriorityQueue` (Binary Heap) gives $O(E \log V)$.
    - Using a **Fibonacci Heap** (theoretical/advanced) gives $O(E + V \log V)$, but this is rarely used in coding interviews due to implementation complexity.
3. **Union-Find:**
    - Without optimizations (Path Compression + Union by Rank), it is $O(E \cdot V)$.
    - **With** optimizations, it is nearly $O(1)$ per operation, hence the $\alpha(V)$.