Complete Graph DSA - Deep Mastery Guide

Table of Contents

Theoretical Foundation

What is a Graph?

Definition: A Graph G = (V, E) is a mathematical structure consisting of:

- **V**: Set of vertices (nodes)
- **E**: Set of edges (connections between vertices)

Historical Context

- **Leonhard Euler (1736)**: Seven Bridges of Königsberg - Birth of Graph Theory
- **Gustav Kirchhoff (1845)**: Tree analysis in electrical circuits
- **Arthur Cayley (1857)**: Enumeration of chemical isomers using trees
- **Dénes Kőnig (1936)**: First textbook on Graph Theory

- **Modern Era**: Network analysis, social graphs, route optimization, AI pathfinding

Core Principle

> Graphs model **relationships** and **connections**. Any problem involving entities and their relationships can be represented as a graph.

Graph Types Taxonomy

```
GRAPHS
├── By Direction
│   ├── Undirected: Edge (u,v) = (v,u)
│   └── Directed (Digraph): Edge (u,v) ≠ (v,u)
│
├── By Weight
│   ├── Unweighted: All edges equal
│   └── Weighted: Edges have costs/distances
│
├── By Connectivity
│   ├── Connected: Path exists between any two vertices
│   ├── Disconnected: Multiple components
│   └── Strongly Connected (Directed): Path exists both ways
│
├── Special Types
│   ├── Tree: Connected acyclic graph (V-1 edges)
│   ├── DAG: Directed Acyclic Graph
│   ├── Bipartite: Vertices in 2 sets, edges only between sets
│   ├── Complete: Every vertex connected to every other
│   └── Cyclic: Contains at least one cycle
```

Mathematical Analysis

Graph Properties - Mathematical Definitions

Degree of a Vertex

Undirected Graph:

- Degree(v) = Number of edges incident to v

- **Handshaking Lemma**: $\Sigma \deg(v) = 2|E|$
  - Proof: Each edge contributes 2 to total degree sum

Directed Graph:

- In-degree(v) = Number of edges entering v
- Out-degree(v) = Number of edges leaving v
- Σ in-degree(v) = Σ out-degree(v) = $|E|$

Path and Cycle

Path: Sequence of vertices where each adjacent pair is connected

- Simple Path: No vertex repeated
- Length: Number of edges in path

Cycle: Path where first and last vertex are the same

- Simple Cycle: No vertex repeated except first/last

Theorem: Tree with n vertices has exactly n-1 edges

Proof:

1. Tree is connected → at least n-1 edges needed
2. Tree is acyclic → at most n-1 edges possible
3. Therefore, exactly n-1 edges ■

Complexity Analysis Framework

Space Complexity

Representation	Space	Best For
Adjacency Matrix	$O(V^2)$	Dense graphs, quick edge lookup
Adjacency List	$O(V + E)$	Sparse graphs, memory efficient
Edge List	$O(E)$	Algorithms that iterate edges

Time Complexity by Operation

Adjacency Matrix:

- Add Edge: $O(1)$
- Remove Edge: $O(1)$
- Check if Edge Exists: $O(1)$
- Get All Neighbors: $O(V)$
- Space: $O(V^2)$

Adjacency List:

- Add Edge: $O(1)$
- Remove Edge: O(degree(v))
- Check if Edge Exists: O(degree(v))
- Get All Neighbors: O(degree(v))
- Space: $O(V + E)$

Graph Density:

- **Sparse**: $E \approx V$ (Example: Trees, social networks)
- **Dense**: $E \approx V^2$ (Example: Complete graphs)

Rule of Thumb:

- If $E > V \log V$ → Use Adjacency List
- If $E \approx V^2$ and need fast lookups → Use Adjacency Matrix

Graph Representations

Pattern 1: Adjacency List (Most Common)

Structure: Array of lists where index i contains neighbors of vertex i

Implementation Choices:

```cpp
// Choice 1: Vector of Vectors (Most flexible)
vector<vector<int>> adj(n);
adj[u].push_back(v);

// Choice 2: Unordered Map (Sparse, non-sequential vertices)
unordered_map<int, vector<int>> adj;

// Choice 3: Array of Lists (Fixed size, fastest)
vector<int> adj[MAX_N];
```

When to Use:

- ✅ Sparse graphs (most interview problems)
- ✅ Need to iterate through neighbors
- ✅ Dynamic graph (add/remove edges)
- ❌ Need quick "does edge exist?" checks

Problem Indicators:

- "Find all neighbors"
- "Traverse the graph"
- "N nodes numbered 0 to N-1"

Pattern 2: Adjacency Matrix

Structure: 2D array where matrix[i][j] = 1 if edge exists

```cpp
vector<vector<int>> matrix(n, vector<int>(n, 0));
matrix[u][v] = 1; // or weight for weighted graphs
```

When to Use:

- ✅ Dense graphs (E close to V^2)
- ✅ Need fast edge existence checks
- ✅ Graph algorithms requiring matrix operations (Floyd-Warshall)
- ❌ Large sparse graphs (memory waste)

Problem Indicators:

- "Given matrix representation"
- "Grid problems" (2D grid is implicit adjacency matrix)

- "All pairs shortest path"

Pattern 3: Edge List

Structure: List of edges as (u, v) or (u, v, weight) tuples

```
vector<pair<int, int>> edges;
// or for weighted
vector<tuple<int, int, int>> edges; // {u, v, weight}
```

When to Use:

- ✅ Kruskal's MST (sort edges by weight)
- ✅ Bellman-Ford algorithm
- ✅ Union-Find problems
- ❌ Need frequent neighbor lookups

Conversion to Adjacency List:

```
vector<vector<int>> buildGraph(int n, vector<pair<int,int>>& edges) {
    vector<vector<int>> adj(n);
    for (auto [u, v] : edges) {
        adj[u].push_back(v);
        adj[v].push_back(u); // if undirected
    }
    return adj;
}
```

Pattern 4: Grid as Implicit Graph

Concept: 2D grid where each cell is a vertex, edges to adjacent cells

Standard Directions:

```
// 4-directional
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};

// 8-directional (includes diagonals)
int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1};
int dy[] = {-1, 0, 1, -1, 1, -1, 0, 1};
```

Coordinate to Index Conversion:

```
// Grid: m x n
// Cell (i, j) → index: i * n + j
// Index k → cell: (k / n, k % n)
```

When to Use:

- Matrix problems (islands, flood fill)
- Pathfinding on grids
- 2D maze problems

Core Traversal Patterns

Pattern 1: DFS (Depth-First Search)

Theoretical Foundation

Concept: Explore as far as possible along each branch before backtracking

Mathematical Property:

- DFS creates a **DFS tree/forest**
- Edges classified as: Tree, Back, Forward, Cross
- Time complexity: O(V + E)

Key Insight: DFS uses **stack** (implicit via recursion or explicit)

Core DFS Template

```cpp
// Recursive DFS (Most common in interviews)
void dfs(int node, vector<vector<int>>& adj, vector<bool>& visited) {
    visited[node] = true;

    // Process current node
    // ... processing logic ...

    // Explore neighbors
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, adj, visited);
        }
    }
}

// Iterative DFS (Stack-based)
void dfs_iterative(int start, vector<vector<int>>& adj, int n) {
    vector<bool> visited(n, false);
    stack<int> st;
    st.push(start);

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

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

        // Process node

        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                st.push(neighbor);
            }
        }
    }
}
```

DFS Sub-Pattern 1.1: Connected Components

Problem: Count number of connected components in undirected graph

Mathematical Insight:

- Each DFS call explores one complete component
- Number of DFS calls = Number of components

Invariant: After k DFS calls, k components have been fully explored

```
int countComponents(int n, vector<vector<int>>& adj) {
    vector<bool> visited(n, false);
    int components = 0;

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i, adj, visited);
            components++; // Each DFS explores one component
        }
    }

    return components;
}
```

Complexity:

- Time: O(V + E) - each vertex visited once, each edge examined twice
- Space: O(V) - visited array + recursion stack

Applications:

- Number of Provinces (LC 547)
- Number of Islands (LC 200)
- Friend Circles
- Network connectivity

DFS Sub-Pattern 1.2: Cycle Detection

A. Cycle Detection in Undirected Graph

Theorem: Undirected graph has cycle ⟺ DFS finds a back edge to visited (non-parent) node

Why parent check needed?: In undirected graph, edge A→B means B is in A's adjacency list AND A is in B's list. Without parent check, we'd falsely detect cycle.

```cpp
bool hasCycleDFS(int node, int parent, vector<vector<int>>& adj,
                 vector<bool>& visited) {
    visited[node] = true;

    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            if (hasCycleDFS(neighbor, node, adj, visited))
                return true;
        }
        else if (neighbor != parent) {
            return true; // Back edge found → Cycle exists
        }
    }

    return false;
}
```

Mathematical Proof:

1. If back edge exists → obvious cycle
2. If cycle exists → DFS must encounter back edge:
   - When exploring cycle, DFS will reach a visited node
   - That visited node is not parent (cycle has ≥3 nodes)
   - Therefore, back edge detected ■

B. Cycle Detection in Directed Graph

Theorem: Directed graph has cycle ⟺ DFS finds back edge to node in **current recursion stack**

Three-Color DFS:

- White (0): Unvisited
- Gray (1): In recursion stack (being processed)
- Black (2): Completely processed

```cpp
bool hasCycleDirected(int node, vector<vector<int>>& adj,
                      vector<int>& color) {
    color[node] = 1; // Mark as being processed (gray)

    for (int neighbor : adj[node]) {
        if (color[neighbor] == 1) {
            return true; // Back edge to gray node → Cycle
        }
        if (color[neighbor] == 0) {
            if (hasCycleDirected(neighbor, adj, color))
                return true;
        }
    }

    color[node] = 2; // Mark as completely processed (black)
    return false;
}
```

Why Three Colors?:

- Color 1 (gray) tracks **current path** in DFS tree
- Back edge to gray node means cycle in current path
- Black nodes are finished; edges to them are cross/forward edges (no cycle)

Applications:

- Course Schedule (LC 207)
- Detect cycle in directed graph
- Dependency resolution

DFS Sub-Pattern 1.3: Backtracking on Graphs

Concept: Explore all paths by marking→exploring→unmarking

Template:

```cpp
void backtrack(int node, vector<vector<int>>& adj,
               vector<bool>& visited, vector<int>& path) {
    visited[node] = true;
    path.push_back(node);

    // Base case (found solution)
    if (isGoal(node)) {
        processPath(path);
    }

    // Explore all neighbors
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            backtrack(neighbor, adj, visited, path);
        }
    }

    // BACKTRACK: Unmark for other paths
    visited[node] = false;
    path.pop_back();
}
```

Key Difference from Regular DFS:

- Regular DFS: visited[node] stays true (explore each node once)
- Backtracking: visited[node] reset to false (explore multiple paths)

Applications:

- All Paths from Source to Target (LC 797)
- Word Search (LC 79)
- Rat in a Maze
- N-Queens (graph interpretation)

Pattern 2: BFS (Breadth-First Search)

Theoretical Foundation

Concept: Explore graph level by level (all neighbors before going deeper)

Mathematical Properties:

- BFS creates **BFS tree** with shortest paths (unweighted)
- Distance from source to node = level in BFS tree
- Time complexity: O(V + E)

Key Insight: BFS uses **queue** and guarantees shortest path in unweighted graphs

Shortest Path Guarantee:

- **Theorem**: In unweighted graph, BFS finds shortest path from source to all reachable nodes
- **Proof**: Nodes visited in increasing order of distance. When node at distance d is visited, all nodes at distance <d already visited ■

Core BFS Template

```
void bfs(int start, vector<vector<int>>& adj, int n) {
    vector<bool> visited(n, false);
    queue<int> q;

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

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

        // Process current node

        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

Critical Points:

1. Mark visited **when pushing** to queue (not when popping)
2. Why? Prevents duplicate entries in queue
3. Process node when popping from queue

BFS Sub-Pattern 2.1: Shortest Path (Unweighted)

Problem: Find shortest path from source to target in unweighted graph

Implementation with Distance Tracking:

```
int shortestPath(int start, int end, vector<vector<int>>& adj, int n) {
    if (start == end) return 0;

    vector<bool> visited(n, false);
    queue<int> q;
    int distance = 0;

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

    while (!q.empty()) {
        int size = q.size();
        distance++;

        // Process all nodes at current level
        for (int i = 0; i < size; i++) {
            int node = q.front();
            q.pop();

            for (int neighbor : adj[node]) {
                if (neighbor == end) {
                    return distance; // Found target
                }

                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
    }

    return -1; // Target not reachable
}
```

Level-by-Level Processing Pattern:

```
while (!q.empty()) {
    int levelSize = q.size(); // Key: capture current level size

    for (int i = 0; i < levelSize; i++) {
        int node = q.front();
        q.pop();
        // Process node at current level
    }

    level++; // Move to next level
}
```

Applications:

- Shortest Path in Binary Matrix (LC 1091)
- Word Ladder (LC 127)
- Minimum Knight Moves
- Snakes and Ladders (LC 909)

BFS Sub-Pattern 2.2: Multi-Source BFS

Concept: Start BFS from multiple sources simultaneously

Key Insight: Push all sources into queue initially, then run standard BFS

Mathematical Property: Finds shortest distance from **any** source to each node

```cpp
void multiSourceBFS(vector<pair<int,int>>& sources,
                    vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    queue<pair<int,int>> q;

    // Initialize: Push all sources
    for (auto [x, y] : sources) {
        q.push({x, y});
    }

    int level = 0;
    while (!q.empty()) {
        int size = q.size();

        for (int i = 0; i < size; i++) {
            auto [x, y] = q.front();
            q.pop();

            // Process and expand
            for (int d = 0; d < 4; d++) {
                int nx = x + dx[d];
                int ny = y + dy[d];

                if (isValid(nx, ny, m, n) && !visited[nx][ny]) {
                    visited[nx][ny] = true;
                    q.push({nx, ny});
                }
            }
        }
        level++;
    }
}
```

Applications:

- Rotten Oranges (LC 994) - All rotten oranges are sources
- 01 Matrix (LC 542) - All zeros are sources
- Walls and Gates (LC 286)
- As Far from Land as Possible (LC 1162)

BFS Sub-Pattern 2.3: Bidirectional BFS

Concept: Start BFS from both source and target, meet in middle

Why?: Reduces search space exponentially!

- Regular BFS: O(b^d) where b = branching factor, d = depth
- Bidirectional: O(b^(d/2) + b^(d/2)) = O(2 * b^(d/2))
- Speedup: b^d / (2 * b^(d/2)) ≈ b^(d/2) / 2 (exponential improvement!)

```cpp
int bidirectionalBFS(int start, int end, vector<vector<int>>& adj, int n) {
    if (start == end) return 0;

    unordered_set<int> frontQueue, backQueue;
    unordered_set<int> visitedFront, visitedBack;

    frontQueue.insert(start);
    backQueue.insert(end);
    visitedFront.insert(start);
    visitedBack.insert(end);

    int steps = 0;

    while (!frontQueue.empty() && !backQueue.empty()) {
        // Always expand smaller frontier (optimization)
        if (frontQueue.size() > backQueue.size()) {
            swap(frontQueue, backQueue);
            swap(visitedFront, visitedBack);
        }

        unordered_set<int> nextLevel;
        steps++;

        for (int node : frontQueue) {
            for (int neighbor : adj[node]) {
                if (visitedBack.count(neighbor)) {
                    return steps; // Frontiers met!
                }

                if (!visitedFront.count(neighbor)) {
                    visitedFront.insert(neighbor);
                    nextLevel.insert(neighbor);
                }
            }
        }

        frontQueue = nextLevel;
    }

    return -1;
}
```

Applications:

- Word Ladder
- Minimum Genetic Mutation
- Any problem requiring shortest path in large search space

Pattern 3: DFS vs BFS Decision Framework

When to Use DFS:

- ✅ Need to explore all paths
- ✅ Checking connectivity
- ✅ Cycle detection
- ✅ Topological sort
- ✅ Backtracking required
- ✅ Memory constrained (O(height) vs O(width))

When to Use BFS:

- ✅ Shortest path (unweighted)
- ✅ Minimum steps/moves
- ✅ Level-order processing
- ✅ Multi-source scenarios
- ✅ Finding "nearest" or "closest"

Problem Keywords:

Keyword	Likely Algorithm
"shortest path"	BFS
"minimum steps"	BFS
"all paths"	DFS + Backtracking
"connectivity"	DFS or Union-Find
"cycle"	DFS (both directed/undirected)
"minimum spanning tree"	Kruskal/Prim
"rotten/spreading"	Multi-source BFS

Advanced Algorithms & Patterns

Pattern 4: Topological Sort

Theoretical Foundation

Definition: Linear ordering of vertices such that for every directed edge (u,v), u comes before v

Existence Condition: Topological sort exists ⟺ Graph is a **DAG** (Directed Acyclic Graph)

Theorem: Every DAG has at least one topological ordering

Proof Sketch:

1. DAG has at least one vertex with in-degree 0 (no cycles)
2. Remove this vertex, remaining graph is still DAG
3. Repeat recursively
4. Order of removal gives topological sort ■

Algorithm 1: Kahn's Algorithm (BFS-based)

Intuition: Repeatedly remove vertices with in-degree 0

```
vector<int> topologicalSort(int n, vector<vector<int>>& adj) {
    // Step 1: Calculate in-degrees
    vector<int> indegree(n, 0);
    for (int u = 0; u < n; u++) {
        for (int v : adj[u]) {
            indegree[v]++;
        }
    }

    // Step 2: Initialize queue with 0 in-degree nodes
    queue<int> q;
    for (int i = 0; i < n; i++) {
        if (indegree[i] == 0) {
            q.push(i);
        }
    }

    // Step 3: BFS process
    vector<int> topoOrder;
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        topoOrder.push_back(node);

        // Reduce in-degree of neighbors
        for (int neighbor : adj[node]) {
            indegree[neighbor]--;
            if (indegree[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }

    // Step 4: Check for cycle
    if (topoOrder.size() != n) {
        return {}; // Cycle exists
    }

    return topoOrder;
}
```

Complexity:

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

Cycle Detection: If topoOrder.size() < n, cycle exists (some nodes never reached in-degree 0)

Algorithm 2: DFS-based Topological Sort

Intuition: Use post-order DFS traversal, reverse the result

```
void dfsTopo(int node, vector<vector<int>>& adj,
             vector<bool>& visited, stack<int>& st) {
    visited[node] = true;

    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfsTopo(neighbor, adj, visited, st);
        }
    }

    st.push(node); // Post-order: Push after exploring all descendants
}

vector<int> topologicalSortDFS(int n, vector<vector<int>>& adj) {
    vector<bool> visited(n, false);
    stack<int> st;

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfsTopo(i, adj, visited, st);
        }
    }

    vector<int> topoOrder;
    while (!st.empty()) {
        topoOrder.push_back(st.top());
        st.pop();
    }

    return topoOrder;
}
```

Why Reverse Post-Order Works:

- Post-order: Process node after all descendants
- If edge u→v, then v finishes before u
- Reversing gives u before v ✓

Kahn's vs DFS:

- Kahn's: Easier to detect cycles, more intuitive
- DFS: More compact code, natural recursion

Applications:

- Course Schedule I & II (LC 207, 210)
- Alien Dictionary (LC 269)
- Sequence Reconstruction (LC 444)
- Build order for dependencies

Pattern 5: Union-Find (Disjoint Set Union)

Theoretical Foundation

Purpose: Maintain disjoint sets with two operations:

- **Find**: Determine which set an element belongs to
- **Union**: Merge two sets

Applications:

- Connected components in dynamic graphs
- Cycle detection
- Minimum spanning trees (Kruskal)

Naive Implementation: $O(n)$ per operation ❌
Optimized: $O(\alpha(n)) \approx O(1)$ per operation ✓

- $\alpha(n)$ = Inverse Ackermann function (grows incredibly slowly)

Optimization 1: Path Compression

Idea: During Find, make nodes point directly to root

```cpp
int find(int x, vector<int>& parent) {
    if (parent[x] != x) {
        parent[x] = find(parent[x], parent); // Path compression
    }
    return parent[x];
}
```

Effect: Flattens tree structure, future operations faster

Optimization 2: Union by Rank/Size

Idea: Attach smaller tree under larger tree's root

```cpp
void unionSets(int x, int y, vector<int>& parent, vector<int>& rank) {
    int rootX = find(x, parent);
    int rootY = find(y, parent);

    if (rootX == rootY) return; // Already in same set

    // Union by rank
    if (rank[rootX] < rank[rootY]) {
        parent[rootX] = rootY;
    } else if (rank[rootX] > rank[rootY]) {
        parent[rootY] = rootX;
    } else {
        parent[rootY] = rootX;
        rank[rootX]++;
    }
}
```

Complete Union-Find Template:

```cpp
class UnionFind {
private:
    vector<int> parent;
    vector<int> rank;
    int components;

public:
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        components = n;
        for (int i = 0; i < n; i++) {
            parent[i] = i; // Each node is its own parent initially
        }
    }

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

    bool unionSets(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] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }

        components--;
        return true;
    }
```

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

    int getComponents() {
        return components;
    }
};
```

Applications:

- Number of Connected Components (LC 323)
- Graph Valid Tree (LC 261)
- Redundant Connection (LC 684)
- Accounts Merge (LC 721)
- Most Stones Removed (LC 947)

Union-Find vs DFS:

- Union-Find: Better for dynamic graphs (edges added over time)
- DFS: Better for static graphs, simpler for one-time queries

Pattern 6: Shortest Path in Weighted Graphs

Algorithm 1: Dijkstra's Algorithm

Purpose: Single-source shortest path with **non-negative** weights

Core Idea: Greedy algorithm - repeatedly pick unvisited node with minimum distance

Mathematical Guarantee:

- **Theorem**: Dijkstra correctly computes shortest paths if all edge weights ≥ 0
- **Proof Sketch**: By induction on number of nodes finalized. Each node finalized with minimum possible distance ■

```cpp
vector<int> dijkstra(int start, vector<vector<pair<int,int>>>& adj, int n) {
    vector<int> dist(n, INT_MAX);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;

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

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

        if (d > dist[u]) continue; // Already found better path

        for (auto [v, w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

Complexity:

- Time: O((V + E) log V) with binary heap
- Space: O(V)

Why Priority Queue?:

- Need to efficiently extract minimum distance node
- Heap operations: O(log V)

Applications:

- Network Delay Time (LC 743)
- Path with Maximum Probability (LC 1514)
- Cheapest Flights within K Stops (LC 787) - Modified Dijkstra
- Minimum Cost to Make Connections

Algorithm 2: Bellman-Ford Algorithm

Purpose: Single-source shortest path with **negative weights** allowed

Core Idea: Relax all edges V-1 times

```
vector<int> bellmanFord(int start, vector<tuple<int,int,int>>& edges,
                        int n) {
    vector<int> dist(n, INT_MAX);
    dist[start] = 0;

    // Relax all edges V-1 times
    for (int i = 0; i < n - 1; i++) {
        for (auto [u, v, w] : edges) {
            if (dist[u] != INT_MAX && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
            }
        }
    }

    // Check for negative cycles
    for (auto [u, v, w] : edges) {
        if (dist[u] != INT_MAX && dist[u] + w < dist[v]) {
            // Negative cycle detected
            return {}; // or mark affected nodes
        }
    }

    return dist;
}
```

Complexity:

- Time: O(VE)
- Space: O(V)

Why V-1 Iterations?:

- Shortest path has at most V-1 edges (no cycles in shortest path)
- After k iterations, shortest paths with ≤k edges are correct
- After V-1 iterations, all shortest paths computed

Negative Cycle Detection: If can still relax after V-1 iterations, negative cycle exists

Applications:

- Graphs with negative weights
- Detecting negative cycles
- Arbitrage detection (currency exchange)

Algorithm 3: Floyd-Warshall

Purpose: All-pairs shortest path

Core Idea: Dynamic programming with intermediate vertices

```
void floydWarshall(vector<vector<int>>& dist, int n) {
    // dist[i][j] = direct edge weight or INF

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }

    // Check negative cycles
    for (int i = 0; i < n; i++) {
        if (dist[i][i] < 0) {
            // Negative cycle exists
        }
    }
}
```

Complexity:

- Time: $O(V^3)$
- Space: $O(V^2)$

When to Use: Small graphs ($V \leq 400$), need all pairs distances

Applications:

- Find the City with Smallest Number of Neighbors (LC 1334)

- Transitive closure
- Graph connectivity queries

Algorithm Comparison:

Algorithm	Time	Space	Works With	Best For
BFS	O(V+E)	O(V)	Unweighted	Unweighted graphs
Dijkstra	O((V+E)logV)	O(V)	Non-negative	Single-source, positive weights
Bellman-Ford	O(VE)	O(V)	Any weights	Negative weights, detect cycles
Floyd-Warshall	$O(V^3)$	$O(V^2)$	Any weights	All-pairs, small graphs

Pattern 7: Minimum Spanning Tree (MST)

Definition: Spanning tree with minimum total edge weight

Properties:

- Connects all vertices
- Has exactly V-1 edges
- No cycles
- Minimum sum of edge weights

Algorithm 1: Kruskal's Algorithm

Idea: Sort edges by weight, greedily add if no cycle created

Uses: Union-Find for cycle detection

```cpp
int kruskalMST(int n, vector<tuple<int,int,int>>& edges) {
    // edges = {weight, u, v}
    sort(edges.begin(), edges.end());

    UnionFind uf(n);
    int mstWeight = 0;
    int edgesAdded = 0;

    for (auto [w, u, v] : edges) {
        if (uf.unionSets(u, v)) {
            mstWeight += w;
            edgesAdded++;
            if (edgesAdded == n - 1) break; // MST complete
        }
    }

    return (edgesAdded == n - 1) ? mstWeight : -1; // -1 if not connected
}
```

Complexity:

- Time: O(E log E) - dominated by sorting
- Space: O(V) for Union-Find

Algorithm 2: Prim's Algorithm

Idea: Start from arbitrary vertex, greedily add minimum weight edge to tree

Uses: Priority queue (similar to Dijkstra)

```cpp
int primMST(int n, vector<vector<pair<int,int>>>& adj) {
    vector<bool> inMST(n, false);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;

    pq.push({0, 0}); // {weight, node}
    int mstWeight = 0;
    int edgesAdded = 0;

    while (!pq.empty() && edgesAdded < n) {
        auto [w, u] = pq.top();
        pq.pop();

        if (inMST[u]) continue;

        inMST[u] = true;
        mstWeight += w;
        edgesAdded++;

        for (auto [v, weight] : adj[u]) {
            if (!inMST[v]) {
                pq.push({weight, v});
            }
        }
    }

    return (edgesAdded == n) ? mstWeight : -1;
}
```

Complexity:

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

Kruskal vs Prim:

- Kruskal: Better for sparse graphs, simpler with Union-Find
- Prim: Better for dense graphs, similar to Dijkstra

Applications:

- Min Cost to Connect All Points (LC 1584)
- Connecting Cities with Minimum Cost (LC 1135)

- Network design optimization

Pattern 8: Bipartite Graphs

Definition: Graph whose vertices can be divided into two sets such that every edge connects vertices from different sets

Key Property: Graph is bipartite ⟺ it has no odd-length cycles

Theorem: Graph is bipartite ⟺ it can be 2-colored

Detection using BFS (Coloring)

```
bool isBipartite(vector<vector<int>>& adj, int n) {
    vector<int> color(n, -1); // -1 = uncolored

    for (int start = 0; start < n; start++) {
        if (color[start] != -1) continue;

        queue<int> q;
        q.push(start);
        color[start] = 0;

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

            for (int v : adj[u]) {
                if (color[v] == -1) {
                    color[v] = 1 - color[u]; // Opposite color
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return false; // Same color → not bipartite
                }
            }
        }
    }

    return true;
}
```

Complexity:

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

Applications:

- Is Graph Bipartite? (LC 785)
- Possible Bipartition (LC 886)
- Matching problems
- Scheduling conflicts

Problem Categories

Category 1: Connected Components

Core Pattern: Count/identify separate connected regions

Standard Problems:

1. **Number of Provinces** (LC 547)
    - Pattern: Connected Components via DFS
    - Approach: DFS from each unvisited node, count calls
    - Complexity: $O(n^2)$ for matrix representation
2. **Number of Islands** (LC 200)
    - Pattern: Connected Components on Grid
    - Approach: DFS/BFS on each '1', mark visited
    - Complexity: $O(m \times n)$
3. **Graph Valid Tree** (LC 261)
    - Pattern: Connectivity + Cycle Detection
    - Approach: Check n-1 edges AND fully connected
    - Complexity: O(V + E)

Mental Model: "How many separate pieces?"

Category 2: Shortest Path Problems

Core Pattern: Minimum distance/steps between points

Problem Matrix:

Problem	Weight Type	Algorithm	Key Insight
Word Ladder	Unweighted	BFS	Each transformation = 1 step
Network Delay Time	Weighted (+)	Dijkstra	Signal propagation
Cheapest Flights	Weighted (+)	Modified Dijkstra	K-stops constraint
Bellman Ford variant	Weighted (±)	Bellman-Ford	Handle negatives

4. **Shortest Path in Binary Matrix** (LC 1091)
   - Pattern: BFS on Grid
   - Approach: 8-directional BFS from (0,0) to (n-1,n-1)
   - Complexity: $O(n^2)$
5. **Word Ladder** (LC 127)
   - Pattern: Unweighted Shortest Path
   - Approach: BFS with word transformations
   - Optimization: Bidirectional BFS
   - Complexity: $O(N \times M^2)$ where N=words, M=word length

Category 3: Cycle Detection

Core Pattern: Determine if graph contains cycle

6. **Course Schedule** (LC 207)
   - Pattern: Cycle Detection in Directed Graph
   - Approach: Topological sort (Kahn's or DFS)
   - Complexity: $O(V + E)$
7. **Redundant Connection** (LC 684)
   - Pattern: Cycle Detection via Union-Find
   - Approach: Process edges, find first causing cycle
   - Complexity: $O(E \times \alpha(V)) \approx O(E)$

Category 4: Topological Ordering

Core Pattern: Order tasks with dependencies

8. **Course Schedule II** (LC 210)
   - Pattern: Topological Sort
   - Approach: Kahn's algorithm with order tracking
   - Complexity: $O(V + E)$

9. **Alien Dictionary** (LC 269)
    - Pattern: Build Graph + Topological Sort
    - Approach: Compare adjacent words → build order → topo sort
    - Complexity: O(C) where C = total characters

Category 5: Advanced Graph Algorithms

10. **Cheapest Flights within K Stops** (LC 787)
    - Pattern: Modified Dijkstra with Constraints
    - Approach: State = (city, cost, stops)
    - Complexity: $O(E \times K \times \log(V \times K))$
11. **Network Delay Time** (LC 743)
    - Pattern: Single-Source Shortest Path
    - Approach: Dijkstra from source
    - Result: max(distances)
    - Complexity: $O((V+E) \log V)$
12. **Min Cost to Connect All Points** (LC 1584)
    - Pattern: Minimum Spanning Tree
    - Approach: Kruskal or Prim
    - Complexity: $O(N^2 \log N)$

60+ Curated Problems

Beginner Level (Foundation)

Connected Components (5 problems):

1. Number of Provinces (LC 547) ⭐ Start here
2. Number of Islands (LC 200) ⭐ Grid DFS
3. Max Area of Island (LC 695)
4. Flood Fill (LC 733)
5. Island Perimeter (LC 463)

Basic Traversal (5 problems):
6. All Paths from Source to Target (LC 797) ⭐ DFS Backtracking
7. Find if Path Exists (LC 1971)
8. Clone Graph (LC 133) ⭐ DFS/BFS

9. Keys and Rooms (LC 841)
10. Find Center of Star Graph (LC 1791)

Intermediate Level

Shortest Path - Unweighted (8 problems):
11. Shortest Path in Binary Matrix (LC 1091) ⭐ Grid BFS
12. Word Ladder (LC 127) ⭐ Classic BFS
13. Minimum Genetic Mutation (LC 433)
14. Snakes and Ladders (LC 909)
15. Shortest Bridge (LC 934)
16. Nearest Exit from Entrance (LC 1926)
17. Rotting Oranges (LC 994) ⭐ Multi-source BFS
18. As Far from Land as Possible (LC 1162)

Cycle Detection (5 problems):
19. Course Schedule (LC 207) ⭐ Directed cycle
20. Course Schedule II (LC 210) ⭐ Topo sort
21. Redundant Connection (LC 684) ⭐ Union-Find
22. Redundant Connection II (LC 685)
23. Find Eventual Safe States (LC 802)

Topological Sort (6 problems):
24. Course Schedule II (LC 210)
25. Alien Dictionary (LC 269) ⭐ Hard variant
26. Sequence Reconstruction (LC 444)
27. Parallel Courses (LC 1136)
28. Minimum Height Trees (LC 310)
29. Sort Items by Groups (LC 1203)

Union-Find (7 problems):
30. Number of Connected Components (LC 323)
31. Graph Valid Tree (LC 261) ⭐ Tree validation
32. Accounts Merge (LC 721)
33. Most Stones Removed (LC 947)
34. Number of Islands II (LC 305) - Dynamic
35. Largest Component Size (LC 952)
36. Satisfiability of Equality Equations (LC 990)

Advanced Level

Shortest Path - Weighted (8 problems):

37. Network Delay Time (LC 743) ⭐ Dijkstra
38. Cheapest Flights within K Stops (LC 787) ⭐ Modified Dijkstra
39. Path with Minimum Effort (LC 1631)
40. Path with Maximum Probability (LC 1514)
41. Minimum Cost to Connect Cities (LC 1135)
42. Swim in Rising Water (LC 778)
43. Reachable Nodes (LC 882)
44. Find the City (LC 1334) - Floyd-Warshall

MST Problems (4 problems):

45. Min Cost to Connect All Points (LC 1584) ⭐ Prim/Kruskal
46. Connecting Cities with Minimum Cost (LC 1135)
47. Optimize Water Distribution (LC 1168)
48. Build Highways (Custom)

Bipartite (4 problems):

49. Is Graph Bipartite? (LC 785) ⭐ 2-coloring
50. Possible Bipartition (LC 886)
51. Divide Nodes Into Two Groups (LC 2493)
52. Maximum Matching (Custom)

Grid Advanced (6 problems):

53. Surrounded Regions (LC 130) ⭐ Capture regions
54. Number of Enclaves (LC 1020)
55. Number of Closed Islands (LC 1254)
56. Shortest Path with Alternating Colors (LC 1129)
57. Shortest Path to Get All Keys (LC 864)
58. Minimum Moves to Reach Target (LC 1263)

Hard Challenges (6 problems):

59. Word Ladder II (LC 126) - All shortest paths
60. Bus Routes (LC 815) - Implicit graph
61. Reconstruct Itinerary (LC 332) - Eulerian path
62. Critical Connections (LC 1192) ⭐ Bridges
63. Evaluate Division (LC 399) - Weighted graph queries
64. Alien Dictionary (LC 269)

65. Largest Color Value (LC 1857) - DP on graphs
66. Minimum Number of Days to Disconnect Island (LC 1568)

Optimization Techniques

1. Early Termination

BFS Shortest Path:

```
// Stop immediately when target found
if (node == target) {
    return distance; // Don't continue exploring
}
```

Why It Works: BFS guarantees shortest path on first encounter

2. Visited Marking Strategies

Strategy A: Mark When Pushing (Standard)

```
visited[neighbor] = true;
q.push(neighbor);
```

- Prevents duplicate queue entries
- More efficient

Strategy B: Mark When Popping

```
int node = q.front(); q.pop();
if (visited[node]) continue;
visited[node] = true;
```

- Allows for priority queue updates (Dijkstra)
- Necessary when node can be reached multiple times

3. State Compression

Problem: Shortest Path to Get All Keys (LC 864)

Instead of: `visited[row][col][key1][key2][key3]...` ❌

Use bitmask: `visited[row][col][keysState]` where keysState is integer ✓

```
// Example: 3 keys (a, b, c)
// State 101 (binary) = have keys a and c
int newState = state | (1 << keyIndex); // Pick up key
bool hasKey = (state & (1 << keyIndex)) != 0; // Check key
```

4. Bidirectional Search

For problems with defined start and end:

- Reduces search space from O(b^d) to O(b^(d/2))
- Significant speedup for large graphs

5.