

Graph Theory

Tawhid Bin Omar

Graph Fundamentals

A graph $G = (V, E)$ consists of vertices (nodes) V and edges E connecting them.

Key Insight

Graph Types:

- **Directed:** edges have direction ($u \rightarrow v$)
- **Undirected:** edges bidirectional ($u \leftrightarrow v$)
- **Weighted:** edges have costs/weights
- **Unweighted:** all edges equal weight (usually 1)

1 Graph Representation

Adjacency List (Most Common)

```
1 // Unweighted graph
2 vector<vector<int>> adj(n);
3 adj[u].push_back(v);
4
5 // Weighted graph
6 vector<vector<pair<int,int>>> adj(n);
7 adj[u].push_back({v, weight});
8
9 // Input: m edges
10 int n, m;
11 cin >> n >> m;
12 vector<vector<int>> adj(n);
13 for (int i = 0; i < m; i++) {
14     int u, v;
15     cin >> u >> v;
16     adj[u].push_back(v);
17     adj[v].push_back(u); // for undirected
18 }
```

Adjacency Matrix

```
1 // For dense graphs or fast edge queries
2 int adj[MAXN][MAXN];
3
4 // Add edge
5 adj[u][v] = 1; // or weight
6
7 // Check edge in O(1)
8 if (adj[u][v]) { /* edge exists */ }
```

Edge List

```
1 // For algorithms like Kruskal's MST
2 struct Edge {
3     int u, v, weight;
4     bool operator<(const Edge& other) const {
```

```
5         return weight < other.weight;
6     }
7 };
8
9 vector<Edge> edges;
10 edges.push_back({u, v, w});
```

2 Graph Traversal

DFS - Depth First Search

Complexity

Time: $O(V + E)$ | **Space:** $O(V)$

```
1 // Recursive DFS
2 vector<bool> visited;
3
4 void dfs(int u, vector<vector<int>>& adj) {
5     visited[u] = true;
6     cout << u << " ";
7
8     for (int v : adj[u]) {
9         if (!visited[v]) {
10             dfs(v, adj);
11         }
12     }
13 }
14
15 // Iterative DFS
16 void dfsIterative(int start,
17 vector<vector<int>>& adj) {
18     int n = adj.size();
19     vector<bool> visited(n, false);
20     stack<int> st;
21
22     st.push(start);
23     visited[start] = true;
24
25     while (!st.empty()) {
26         int u = st.top();
```

```

27     st.pop();
28     cout << u << " ";
29
30     for (int v : adj[u]) {
31         if (!visited[v]) {
32             visited[v] = true;
33             st.push(v);
34         }
35     }
36 }
37 }
38
39 // DFS with timestamps (for topological sort)
40 vector<int> visited, finish_time;
41 int timer = 0;
42
43 void dfsTime(int u, vector<vector<int>>& adj) {
44     visited[u] = 1;
45
46     for (int v : adj[u]) {
47         if (!visited[v]) {
48             dfsTime(v, adj);
49         }
50     }
51
52     finish_time[u] = timer++;
53 }

```

```

43     if (!visited[v]) {
44         visited[v] = true;
45         parent[v] = u;
46         q.push(v);
47     }
48 }
49 }
50
51 // Reconstruct path
52 vector<int> path;
53 if (!visited[end]) return path;
54
55 for (int v = end; v != -1; v = parent[v]) {
56     path.push_back(v);
57 }
58 reverse(path.begin(), path.end());
59 return path;
60 }

```

0-1 BFS (0-1 Weighted Edges)

BFS - Breadth First Search

Complexity

Time: $O(V + E)$ | **Space:** $O(V)$

```

1 // BFS with distance tracking
2 vector<int> bfs(int start,
3 vector<vector<int>>& adj) {
4     int n = adj.size();
5     vector<int> dist(n, -1);
6     queue<int> q;
7
8     dist[start] = 0;
9     q.push(start);
10
11     while (!q.empty()) {
12         int u = q.front();
13         q.pop();
14
15         for (int v : adj[u]) {
16             if (dist[v] == -1) {
17                 dist[v] = dist[u] + 1;
18                 q.push(v);
19             }
20         }
21     }
22     return dist;
23 }
24
25 // BFS with parent tracking (for path)
26 vector<int> bfsPath(int start, int end,
27 vector<vector<int>>& adj) {
28     int n = adj.size();
29     vector<int> parent(n, -1);
30     vector<bool> visited(n, false);
31     queue<int> q;
32
33     visited[start] = true;
34     q.push(start);
35
36     while (!q.empty()) {
37         int u = q.front();
38         q.pop();
39
40         if (u == end) break;
41
42         for (int v : adj[u]) {

```

```

// For graphs with edge weights 0 or 1
vector<int> bfs01(int start,
vector<vector<pair<int, int>>& adj) {
    int n = adj.size();
    vector<int> dist(n, INT_MAX);
    deque<int> dq;

    dist[start] = 0;
    dq.push_front(start);

    while (!dq.empty()) {
        int u = dq.front();
        dq.pop_front();

        for (auto [v, w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                if (w == 0)
                    dq.push_front(v);
                else
                    dq.push_back(v);
            }
        }
    }
    return dist;
}

```

3 Connected Components

Count Components (Undirected)

Problem

Count number of connected components in undirected graph.

Solution

```

1  vector<bool> visited;
2
3  void dfs(int u, vector<vector<int>>& adj) {
4      visited[u] = true;
5      for (int v : adj[u]) {
6          if (!visited[v]) {
7              dfs(v, adj);
8          }
9      }
10 }
11
12 int countComponents(int n,
13 vector<vector<int>>& adj) {
14     visited.assign(n, false);
15     int count = 0;
16
17     for (int i = 0; i < n; i++) {
18         if (!visited[i]) {
19             dfs(i, adj);
20             count++;
21         }
22     }
23     return count;
24 }

```

```

45     visited.assign(n, false);
46     vector<vector<int>> sccs;
47
48     reverse(order.begin(), order.end());
49     for (int u : order) {
50         if (!visited[u]) {
51             component.clear();
52             dfs2(u, radj);
53             sccs.push_back(component);
54         }
55     }
56 }
57
58     return sccs;
59 }

```

Strongly Connected Components (SCC)

For directed graphs - use Kosaraju's or Tarjan's algorithm.

```

1  // Kosaraju's Algorithm
2  vector<bool> visited;
3  vector<int> order, component;
4
5  void dfs1(int u, vector<vector<int>>& adj) {
6      visited[u] = true;
7      for (int v : adj[u]) {
8          if (!visited[v]) {
9              dfs1(v, adj);
10         }
11     }
12     order.push_back(u);
13 }
14
15 void dfs2(int u, vector<vector<int>>& radj) {
16     visited[u] = true;
17     component.push_back(u);
18     for (int v : radj[u]) {
19         if (!visited[v]) {
20             dfs2(v, radj);
21         }
22     }
23 }
24
25 vector<vector<int>> findSCC(int n,
26 vector<vector<int>>& adj) {
27     visited.assign(n, false);
28     order.clear();
29
30     // First DFS: compute finish times
31     for (int i = 0; i < n; i++) {
32         if (!visited[i]) {
33             dfs1(i, adj);
34         }
35     }
36
37     // Build reverse graph
38     vector<vector<int>> radj(n);
39     for (int u = 0; u < n; u++) {
40         for (int v : adj[u]) {
41             radj[v].push_back(u);
42         }
43     }
44
45     // Second DFS on reverse graph

```

4 Shortest Path Algorithms

Dijkstra's Algorithm

Complexity

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

Key Insight

Use when: Non-negative edge weights, single source shortest path.

```

1 vector<int> dijkstra(int start, int n,
2   vector<vector<pair<int,int>>& adj) {
3   vector<int> dist(n, INT_MAX);
4   priority_queue<pair<int,int>,
5     vector<pair<int,int>>,
6     greater<pair<int,int>>> pq;
7
8   dist[start] = 0;
9   pq.push({0, start});
10
11  while (!pq.empty()) {
12    auto [d, u] = pq.top();
13    pq.pop();
14
15    if (d > dist[u]) continue;
16
17    for (auto [v, w] : adj[u]) {
18      if (dist[u] + w < dist[v]) {
19        dist[v] = dist[u] + w;
20        pq.push({dist[v], v});
21      }
22    }
23  }
24  return dist;
25 }
26
27 // Dijkstra with path reconstruction
28 pair<vector<int>, vector<int>>
29 dijkstraWithPath(int start, int n,
30   vector<vector<pair<int,int>>& adj) {
31   vector<int> dist(n, INT_MAX);
32   vector<int> parent(n, -1);
33   priority_queue<pair<int,int>,
34     vector<pair<int,int>>,
35     greater<pair<int,int>>> pq;
36
37   dist[start] = 0;
38   pq.push({0, start});
39
40   while (!pq.empty()) {
41     auto [d, u] = pq.top();
42     pq.pop();
43
44     if (d > dist[u]) continue;
45
46     for (auto [v, w] : adj[u]) {
47       if (dist[u] + w < dist[v]) {
48         dist[v] = dist[u] + w;
49         parent[v] = u;
50         pq.push({dist[v], v});
51       }
52     }
53   }
54   return {dist, parent};
55 }
```

Bellman-Ford Algorithm

Complexity

Time: $O(VE)$ | Can detect negative cycles

```

1 struct Edge {
2   int u, v, weight;
3 };
4
5 pair<vector<int>, bool> bellmanFord(int start,
6   int n, vector<Edge>& edges) {
7   vector<int> dist(n, INT_MAX);
8   dist[start] = 0;
9
10  // Relax edges n-1 times
11  for (int i = 0; i < n - 1; i++) {
12    for (auto& e : edges) {
13      if (dist[e.u] != INT_MAX &&
14        dist[e.u] + e.weight < dist[e.v]) {
15        dist[e.v] = dist[e.u] + e.weight;
16      }
17    }
18  }
19
20  // Check for negative cycle
21  bool hasNegCycle = false;
22  for (auto& e : edges) {
23    if (dist[e.u] != INT_MAX &&
24      dist[e.u] + e.weight < dist[e.v]) {
25      hasNegCycle = true;
26      break;
27    }
28  }
29
30  return {dist, hasNegCycle};
31 }
```

Floyd-Warshall Algorithm

Complexity

Time: $O(V^3)$ | All-pairs shortest path

```

1 void floydWarshall(vector<vector<int>>& dist) {
2   int n = dist.size();
3
4   for (int k = 0; k < n; k++) {
5     for (int i = 0; i < n; i++) {
6       for (int j = 0; j < n; j++) {
7         if (dist[i][k] != INT_MAX &&
8           dist[k][j] != INT_MAX) {
9           dist[i][j] = min(dist[i][j],
10             dist[i][k] + dist[k][j]);
11         }
12       }
13     }
14   }
15 }
16
17 // Initialize distance matrix
18 vector<vector<int>> buildDistMatrix(int n,
19   vector<tuple<int,int,int>>& edges) {
20   vector<vector<int>> dist(n,
21     vector<int>(n, INT_MAX));
22
23   // Distance to self is 0
24   for (int i = 0; i < n; i++)
25     dist[i][i] = 0;
26
27   // Add edges
28   for (auto [u, v, w] : edges) {
29     dist[u][v] = w;
30   }
31 }
```

```

31     return dist;
32 }
33

```

5 Topological Sort

Key Insight

Works only for: Directed Acyclic Graphs (DAG)

Kahn's Algorithm (BFS-based)

```

1 vector<int> topologicalSort(int n,
2   vector<vector<int>>& adj) {
3   vector<int> indegree(n, 0);
4
5   // Compute indegrees
6   for (int u = 0; u < n; u++) {
7     for (int v : adj[u]) {
8       indegree[v]++;
9     }
10  }
11
12  queue<int> q;
13  for (int i = 0; i < n; i++) {
14    if (indegree[i] == 0)
15      q.push(i);
16  }
17
18  vector<int> result;
19  while (!q.empty()) {
20    int u = q.front();
21    q.pop();
22    result.push_back(u);
23
24    for (int v : adj[u]) {
25      indegree[v]--;
26      if (indegree[v] == 0)
27        q.push(v);
28    }
29  }
30
31  // If result.size() != n, cycle exists
32  if (result.size() != n)
33    return {}; // cycle detected
34
35  return result;
36 }

```

DFS-based Topological Sort

```

1 vector<bool> visited;
2 vector<int> topoOrder;
3
4 void dfs(int u, vector<vector<int>>& adj) {
5   visited[u] = true;
6
7   for (int v : adj[u]) {
8     if (!visited[v]) {
9       dfs(v, adj);
10    }
11  }
12
13  topoOrder.push_back(u);
14 }
15
16 vector<int> topologicalSortDFS(int n,
17   vector<vector<int>>& adj) {
18   visited.assign(n, false);
19   topoOrder.clear();
20
21   for (int i = 0; i < n; i++) {

```

```

22     if (!visited[i]) {
23       dfs(i, adj);
24     }
25   }
26
27   reverse(topoOrder.begin(), topoOrder.end());
28   return topoOrder;
29 }

```

6 Cycle Detection

Undirected Graph

```

1 vector<bool> visited;
2
3 bool hasCycleDFS(int u, int parent,
4   vector<vector<int>>& adj) {
5   visited[u] = true;
6
7   for (int v : adj[u]) {
8     if (!visited[v]) {
9       if (hasCycleDFS(v, u, adj))
10        return true;
11     } else if (v != parent) {
12       return true; // back edge found
13     }
14   }
15   return false;
16 }
17
18 bool hasCycle(int n, vector<vector<int>>& adj) {
19   visited.assign(n, false);
20
21   for (int i = 0; i < n; i++) {
22     if (!visited[i]) {
23       if (hasCycleDFS(i, -1, adj))
24         return true;
25     }
26   }
27   return false;
28 }

```

Directed Graph

```

1 vector<int> color; // 0: white, 1: gray, 2: black
2
3 bool hasCycleDFS(int u,
4   vector<vector<int>>& adj) {
5   color[u] = 1; // gray (in progress)
6
7   for (int v : adj[u]) {
8     if (color[v] == 1) {
9       return true; // back edge to gray node
10    }
11    if (color[v] == 0 && hasCycleDFS(v, adj)) {
12      return true;
13    }
14  }
15
16  color[u] = 2; // black (completed)
17  return false;
18 }
19
20 bool hasCycleDirected(int n,
21   vector<vector<int>>& adj) {
22   color.assign(n, 0);
23
24   for (int i = 0; i < n; i++) {
25     if (color[i] == 0) {
26       if (hasCycleDFS(i, adj))
27         return true;
28     }
29   }
30   return false;

```

7 Minimum Spanning Tree

Kruskal's Algorithm

Complexity

Time: $O(E \log E)$ | Uses Union-Find

```

1 struct Edge {
2     int u, v, weight;
3     bool operator<(const Edge& other) const {
4         return weight < other.weight;
5     }
6 };
7
8 // Union-Find (DSU)
9 class DSU {
10     vector<int> parent, rank;
11 public:
12     DSU(int n) : parent(n), rank(n, 0) {
13         iota(parent.begin(), parent.end(), 0);
14     }
15
16     int find(int x) {
17         if (parent[x] != x)
18             parent[x] = find(parent[x]);
19         return parent[x];
20     }
21
22     bool unite(int x, int y) {
23         int px = find(x), py = find(y);
24         if (px == py) return false;
25
26         if (rank[px] < rank[py])
27             swap(px, py);
28         parent[py] = px;
29         if (rank[px] == rank[py])
30             rank[px]++;
31         return true;
32     }
33 };
34
35 pair<int, vector<Edge>> kruskal(int n,
36     vector<Edge>& edges) {
37     sort(edges.begin(), edges.end());
38
39     DSU dsu(n);
40     vector<Edge> mst;
41     int totalWeight = 0;
42
43     for (auto& e : edges) {
44         if (dsu.unite(e.u, e.v)) {
45             mst.push_back(e);
46             totalWeight += e.weight;
47             if (mst.size() == n - 1)
48                 break;
49         }
50     }
51
52     return {totalWeight, mst};
53 }

```

Prim's Algorithm

Complexity

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

```

1 int prim(int n,
2     vector<vector<pair<int,int>>& adj) {
3     vector<bool> inMST(n, false);
4     priority_queue<pair<int,int>,
5         vector<pair<int,int>>,
6         greater<pair<int,int>>> pq;
7
8     int totalWeight = 0;
9     int edgesUsed = 0;
10
11     // Start from node 0
12     pq.push({0, 0}); // {weight, node}
13
14     while (!pq.empty() && edgesUsed < n) {
15         auto [w, u] = pq.top();
16         pq.pop();
17
18         if (inMST[u]) continue;
19
20         inMST[u] = true;
21         totalWeight += w;
22         edgesUsed++;
23
24         for (auto [v, weight] : adj[u]) {
25             if (!inMST[v]) {
26                 pq.push({weight, v});
27             }
28         }
29     }
30
31     return totalWeight;
32 }

```

8 Bipartite Graphs

Check if Bipartite

```

1 vector<int> color;
2
3 bool isBipartiteDFS(int u, int c,
4     vector<vector<int>>& adj) {
5     color[u] = c;
6
7     for (int v : adj[u]) {
8         if (color[v] == -1) {
9             if (!isBipartiteDFS(v, 1 - c, adj))
10                 return false;
11         } else if (color[v] == color[u]) {
12             return false;
13         }
14     }
15     return true;
16 }
17
18 bool isBipartite(int n,
19     vector<vector<int>>& adj) {
20     color.assign(n, -1);
21
22     for (int i = 0; i < n; i++) {
23         if (color[i] == -1) {
24             if (!isBipartiteDFS(i, 0, adj))
25                 return false;
26         }
27     }
28     return true;
29 }
30
31 // BFS version
32 bool isBipartiteBFS(int n,
33     vector<vector<int>>& adj) {
34     vector<int> color(n, -1);
35
36     for (int start = 0; start < n; start++) {
37         if (color[start] != -1) continue;

```

```

39 queue<int> q;
40 q.push(start);
41 color[start] = 0;
42
43 while (!q.empty()) {
44     int u = q.front();
45     q.pop();
46
47     for (int v : adj[u]) {
48         if (color[v] == -1) {
49             color[v] = 1 - color[u];
50             q.push(v);
51         } else if (color[v] == color[u]) {
52             return false;
53         }
54     }
55 }
56 }
57 return true;
58 }
    
```

9 Union-Find (DSU)

Complexity

Time: Nearly $O(1)$ per operation (amortized)

```

1 class UnionFind {
2     vector<int> parent, rank, size;
3 public:
4     UnionFind(int n) : parent(n), rank(n, 0),
5         size(n, 1) {
6         iota(parent.begin(), parent.end(), 0);
7     }
8
9     int find(int x) {
10        if (parent[x] != x)
11            parent[x] = find(parent[x]); // compression
12        return parent[x];
13    }
14
15    bool unite(int x, int y) {
16        int px = find(x), py = find(y);
17        if (px == py) return false;
18
19        // Union by rank
20        if (rank[px] < rank[py])
21            swap(px, py);
22        parent[py] = px;
23        size[px] += size[py];
24        if (rank[px] == rank[py])
25            rank[px]++;
26        return true;
27    }
28
29    bool connected(int x, int y) {
30        return find(x) == find(y);
31    }
32
33    int getSize(int x) {
34        return size[find(x)];
35    }
36 };
    
```

Problem: Number of Islands

Count connected components of 1's in binary matrix.

Solution

```

1 class Solution {
2     int dx[4] = {-1, 1, 0, 0};
3     int dy[4] = {0, 0, -1, 1};
4
5     void dfs(int i, int j,
6         vector<vector<char>>& grid) {
7         int n = grid.size(), m =
8             grid[0].size();
9         if (i < 0 || i >= n || j < 0 || j >= m
10            || grid[i][j] == '0')
11             return;
12
13         grid[i][j] = '0'; // mark visited
14
15         for (int k = 0; k < 4; k++) {
16             dfs(i + dx[k], j + dy[k], grid);
17         }
18     }
19 public:
20     int numIslands(vector<vector<char>>&
21         grid) {
22         int count = 0;
23         int n = grid.size(), m =
24             grid[0].size();
25
26         for (int i = 0; i < n; i++) {
27             for (int j = 0; j < m; j++) {
28                 if (grid[i][j] == '1') {
29                     dfs(i, j, grid);
30                     count++;
31                 }
32             }
33         }
34         return count;
35     };
36 };
    
```

10 Advanced Problems

Bridges (Cut Edges)

```

1 vector<vector<int>> adj;
2 vector<bool> visited;
3 vector<int> disc, low;
4 vector<pair<int,int>> bridges;
5 int timer = 0;
6
7 void findBridges(int u, int parent) {
8     visited[u] = true;
9     disc[u] = low[u] = timer++;
10
11     for (int v : adj[u]) {
12         if (v == parent) continue;
13
14         if (visited[v]) {
15             low[u] = min(low[u], disc[v]);
16         } else {
17             findBridges(v, u);
18             low[u] = min(low[u], low[v]);
19
20             if (low[v] > disc[u]) {
21                 bridges.push_back({u, v});
22             }
23         }
24     }
25 }
    
```

Articulation Points (Cut Vertices)

```
1 vector<bool> isArticulation;
2 int children = 0;
3
4 void findArticulationPoints(int u, int parent) {
5     visited[u] = true;
6     disc[u] = low[u] = timer++;
7
8     for (int v : adj[u]) {
9         if (v == parent) continue;
10
11         if (visited[v]) {
12             low[u] = min(low[u], disc[v]);
13         } else {
14             findArticulationPoints(v, u);
15
16             low[u] = min(low[u], low[v]);
17
18             if (parent != -1 && low[v] >= disc[u]) {
19                 isArticulation[u] = true;
20             }
21             children++;
22         }
23
24         if (parent == -1 && children > 1) {
25             isArticulation[u] = true;
26         }
27     }
28 }
```

Practice Problems

Traversal & Basics:

- CSES - Counting Rooms
- LeetCode 200 - Number of Islands
- Codeforces 115A - Party

Shortest Path:

- CSES - Shortest Routes I (Dijkstra)
- LeetCode 743 - Network Delay Time
- Codeforces 20C - Dijkstra?

Topological Sort:

- LeetCode 207 - Course Schedule
- CSES - Course Schedule
- Codeforces 510C - Fox And Names

MST:

- CSES - Road Reparation
- LeetCode 1584 - Min Cost to Connect Points

Advanced:

- CSES - Planets and Kingdoms (SCC)
- LeetCode 1192 - Critical Connections (Bridges)
- CSES - Road Construction (Union-Find)

Master graphs to solve connectivity, optimization, and path problems efficiently.