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 → v)
- **Undirected:** edges bidirectional (u ↔ v)
- **Weighted:** edges have costs/weights
- **Unweighted:** all edges equal weight (usually 1)

1 Graph Representation

Adjacency List (Most Common)

```cpp
// Unweighted graph
vector<vector<int>> adj(n);
adj[u].push_back(v);

// Weighted graph
vector<vector<pair<int,int>>> adj(n);
adj[u].push_back({v, weight});

// Input: m edges
int n, m;
cin >> n >> m;
vector<vector<int>> adj(n);
for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u); // for undirected
}
```

Adjacency Matrix

```cpp
// For dense graphs or fast edge queries
int adj[MAXN][MAXN];

// Add edge
adj[u][v] = 1; // or weight

// Check edge in O(1)
if (adj[u][v]) { /* edge exists */ }
```

Edge List

```cpp
// For algorithms like Kruskal's MST
struct Edge {
    int u, v, weight;
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

vector<Edge> edges;
edges.push_back({u, v, w});
```

2 Graph Traversal

DFS - Depth First Search

Complexity

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

```cpp
// Recursive DFS
vector<bool> visited;

void dfs(int u, vector<vector<int>>& adj) {
    visited[u] = true;
    cout << u << " ";

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

// Iterative DFS
void dfsIterative(int start,
    vector<vector<int>>& adj) {
    int n = adj.size();
    vector<bool> visited(n, false);
    stack<int> st;

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

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

```
        st.pop();
        cout << u << " ";

        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                st.push(v);
            }
        }
    }
}

// DFS with timestamps (for topological sort)
vector<int> visited, finish_time;
int timer = 0;

void dfsTime(int u, vector<vector<int>>& adj) {
    visited[u] = 1;

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

    finish_time[u] = timer++;
}
```

BFS - Breadth First Search

Complexity

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

```
// BFS with distance tracking
vector<int> bfs(int start,
    vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> dist(n, -1);
    queue<int> q;

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

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

        for (int v : adj[u]) {
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
    return dist;
}

// BFS with parent tracking (for path)
vector<int> bfsPath(int start, int end,
    vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> parent(n, -1);
    vector<bool> visited(n, false);
    queue<int> q;

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

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

        if (u == end) break;

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

    // Reconstruct path
    vector<int> path;
    if (!visited[end]) return path;

    for (int v = end; v != -1; v = parent[v]) {
        path.push_back(v);
    }
    reverse(path.begin(), path.end());
    return path;
}
```

0-1 BFS (0-1 Weighted Edges)

```
// 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

```
vector<bool> visited;

void dfs(int u, vector<vector<int>>& adj) {
   visited[u] = true;
   for (int v : adj[u]) {
      if (!visited[v]) {
         dfs(v, adj);
      }
   }
}

int countComponents(int n,
   vector<vector<int>>& adj) {
   visited.assign(n, false);
   int count = 0;

   for (int i = 0; i < n; i++) {
      if (!visited[i]) {
         dfs(i, adj);
         count++;
      }
   }
   return count;
}
```

Strongly Connected Components (SCC)

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

```
// Kosaraju's Algorithm
vector<bool> visited;
vector<int> order, component;

void dfs1(int u, vector<vector<int>>& adj) {
   visited[u] = true;
   for (int v : adj[u]) {
      if (!visited[v]) {
         dfs1(v, adj);
      }
   }
   order.push_back(u);
}

void dfs2(int u, vector<vector<int>>& radj) {
   visited[u] = true;
   component.push_back(u);
   for (int v : radj[u]) {
      if (!visited[v]) {
         dfs2(v, radj);
      }
   }
}

vector<vector<int>> findSCC(int n,
   vector<vector<int>>& adj) {
   visited.assign(n, false);
   order.clear();

   // First DFS: compute finish times
   for (int i = 0; i < n; i++) {
      if (!visited[i]) {
         dfs1(i, adj);
      }
   }

   // Build reverse graph
   vector<vector<int>> radj(n);
   for (int u = 0; u < n; u++) {
      for (int v : adj[u]) {
         radj[v].push_back(u);
      }
   }

   // Second DFS on reverse graph
   visited.assign(n, false);
   vector<vector<int>> sccs;

   reverse(order.begin(), order.end());
   for (int u : order) {
      if (!visited[u]) {
         component.clear();
         dfs2(u, radj);
         sccs.push_back(component);
      }
   }

   return sccs;
}
```

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.

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

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

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

      if (d > dist[u]) continue;

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

// Dijkstra with path reconstruction
pair<vector<int>, vector<int>>
dijkstraWithPath(int start, int n,
   vector<vector<pair<int,int>>>& adj) {
   vector<int> dist(n, INT_MAX);
   vector<int> parent(n, -1);
   priority_queue<pair<int,int>,
      vector<pair<int,int>>,
      greater<pair<int,int>>> pq;

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

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

      if (d > dist[u]) continue;

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

Bellman-Ford Algorithm

Complexity

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

```
struct Edge {
   int u, v, weight;
};

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

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

   // Check for negative cycle
   bool hasNegCycle = false;
   for (auto& e : edges) {
      if (dist[e.u] != INT_MAX &&
         dist[e.u] + e.weight < dist[e.v]) {
         hasNegCycle = true;
         break;
      }
   }

   return {dist, hasNegCycle};
}
```

Floyd-Warshall Algorithm

Complexity

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

```
void floydWarshall(vector<vector<int>>& dist) {
   int n = dist.size();

   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] != INT_MAX &&
               dist[k][j] != INT_MAX) {
               dist[i][j] = min(dist[i][j],
                  dist[i][k] + dist[k][j]);
            }
         }
      }
   }
}

// Initialize distance matrix
vector<vector<int>> buildDistMatrix(int n,
   vector<tuple<int,int,int>>& edges) {
   vector<vector<int>> dist(n,
      vector<int>(n, INT_MAX));

   // Distance to self is 0
   for (int i = 0; i < n; i++)
      dist[i][i] = 0;

   // Add edges
   for (auto [u, v, w] : edges) {
      dist[u][v] = w;
   }
```

```
    return dist;
}
```

5 Topological Sort

Key Insight

Works only for: Directed Acyclic Graphs (DAG)

Kahn's Algorithm (BFS-based)

```
vector<int> topologicalSort(int n,
    vector<vector<int>>& adj) {
    vector<int> indegree(n, 0);

    // Compute indegrees
    for (int u = 0; u < n; u++) {
        for (int v : adj[u]) {
            indegree[v]++;
        }
    }

    queue<int> q;
    for (int i = 0; i < n; i++) {
        if (indegree[i] == 0)
            q.push(i);
    }

    vector<int> result;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);

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

    // If result.size() != n, cycle exists
    if (result.size() != n)
        return {}; // cycle detected

    return result;
}
```

DFS-based Topological Sort

```
vector<bool> visited;
vector<int> topoOrder;

void dfs(int u, vector<vector<int>>& adj) {
    visited[u] = true;

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

    topoOrder.push_back(u);
}

vector<int> topologicalSortDFS(int n,
    vector<vector<int>>& adj) {
    visited.assign(n, false);
    topoOrder.clear();

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

    reverse(topoOrder.begin(), topoOrder.end());
    return topoOrder;
}
```

6 Cycle Detection

Undirected Graph

```
vector<bool> visited;

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

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

bool hasCycle(int n, vector<vector<int>>& adj) {
    visited.assign(n, false);

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

Directed Graph

```
vector<int> color; // 0: white, 1: gray, 2: black

bool hasCycleDFS(int u,
    vector<vector<int>>& adj) {
    color[u] = 1; // gray (in progress)

    for (int v : adj[u]) {
        if (color[v] == 1) {
            return true; // back edge to gray node
        }
        if (color[v] == 0 && hasCycleDFS(v, adj)) {
            return true;
        }
    }

    color[u] = 2; // black (completed)
    return false;
}

bool hasCycleDirected(int n,
    vector<vector<int>>& adj) {
    color.assign(n, 0);

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

```
}
```

7 Minimum Spanning Tree

Kruskal's Algorithm

Complexity

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

```
struct Edge {
   int u, v, weight;
   bool operator<(const Edge& other) const {
      return weight < other.weight;
   }
};

// Union-Find (DSU)
class DSU {
   vector<int> parent, rank;
public:
   DSU(int n) : parent(n), rank(n, 0) {
      iota(parent.begin(), parent.end(), 0);
   }

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

   bool unite(int x, int y) {
      int px = find(x), py = find(y);
      if (px == py) return false;

      if (rank[px] < rank[py])
         swap(px, py);
      parent[py] = px;
      if (rank[px] == rank[py])
         rank[px]++;
      return true;
   }
};

pair<int, vector<Edge>> kruskal(int n,
   vector<Edge>& edges) {
   sort(edges.begin(), edges.end());

   DSU dsu(n);
   vector<Edge> mst;
   int totalWeight = 0;

   for (auto& e : edges) {
      if (dsu.unite(e.u, e.v)) {
         mst.push_back(e);
         totalWeight += e.weight;
         if (mst.size() == n - 1)
            break;
      }
   }

   return {totalWeight, mst};
}
```

Prim's Algorithm

Complexity

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

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

   int totalWeight = 0;
   int edgesUsed = 0;

   // Start from node 0
   pq.push({0, 0}); // {weight, node}

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

      if (inMST[u]) continue;

      inMST[u] = true;
      totalWeight += w;
      edgesUsed++;

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

   return totalWeight;
}
```

8 Bipartite Graphs

Check if Bipartite

```
vector<int> color;

bool isBipartiteDFS(int u, int c,
   vector<vector<int>>& adj) {
   color[u] = c;

   for (int v : adj[u]) {
      if (color[v] == -1) {
         if (!isBipartiteDFS(v, 1 - c, adj))
            return false;
      } else if (color[v] == color[u]) {
         return false;
      }
   }
   return true;
}

bool isBipartite(int n,
   vector<vector<int>>& adj) {
   color.assign(n, -1);

   for (int i = 0; i < n; i++) {
      if (color[i] == -1) {
         if (!isBipartiteDFS(i, 0, adj))
            return false;
      }
   }
   return true;
}

// BFS version
bool isBipartiteBFS(int n,
   vector<vector<int>>& adj) {
   vector<int> color(n, -1);

   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];
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return false;
                }
            }
        }
    }
    return true;
}
```

9 Union-Find (DSU)

Complexity

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

```
class UnionFind {
    vector<int> parent, rank, size;
public:
    UnionFind(int n) : parent(n), rank(n, 0),
        size(n, 1) {
        iota(parent.begin(), parent.end(), 0);
    }

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

    bool unite(int x, int y) {
        int px = find(x), py = find(y);
        if (px == py) return false;

        // Union by rank
        if (rank[px] < rank[py])
            swap(px, py);
        parent[py] = px;
        size[px] += size[py];
        if (rank[px] == rank[py])
            rank[px]++;
        return true;
    }

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

    int getSize(int x) {
        return size[find(x)];
    }
};
```

Problem: Number of Islands

Count connected components of 1's in binary matrix.

Solution

```
class Solution {
    int dx[4] = {-1, 1, 0, 0};
    int dy[4] = {0, 0, -1, 1};

    void dfs(int i, int j,
        vector<vector<char>>& grid) {
        int n = grid.size(), m =
            grid[0].size();
        if (i < 0 || i >= n || j < 0 || j >= m
            ||
            grid[i][j] == '0')
            return;

        grid[i][j] = '0'; // mark visited

        for (int k = 0; k < 4; k++) {
            dfs(i + dx[k], j + dy[k], grid);
        }
    }

public:
    int numIslands(vector<vector<char>>&
        grid) {
        int count = 0;
        int n = grid.size(), m =
            grid[0].size();

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

10 Advanced Problems

Bridges (Cut Edges)

```
vector<vector<int>> adj;
vector<bool> visited;
vector<int> disc, low;
vector<pair<int,int>> bridges;
int timer = 0;

void findBridges(int u, int parent) {
    visited[u] = true;
    disc[u] = low[u] = timer++;

    for (int v : adj[u]) {
        if (v == parent) continue;

        if (visited[v]) {
            low[u] = min(low[u], disc[v]);
        } else {
            findBridges(v, u);
            low[u] = min(low[u], low[v]);

            if (low[v] > disc[u]) {
                bridges.push_back({u, v});
            }
        }
    }
}
```

Articulation Points (Cut Vertices)

```cpp
vector<bool> isArticulation;
int children = 0;

void findArticulationPoints(int u, int parent) {
   visited[u] = true;
   disc[u] = low[u] = timer++;

   for (int v : adj[u]) {
      if (v == parent) continue;

      if (visited[v]) {
         low[u] = min(low[u], disc[v]);
      } else {
         findArticulationPoints(v, u);
         low[u] = min(low[u], low[v]);

         if (parent != -1 && low[v] >= disc[u]) {
            isArticulation[u] = true;
         }
         children++;
      }
   }

   if (parent == -1 && children > 1) {
      isArticulation[u] = true;
   }
}
```

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.