# EE 610 Fundamentals of VLSI CAD

**Submitted By**

**Karan Markandey**

**210102042**

# Assignment 02:-

Write a program that analyzes a graph's structure, specifically focusing on the connectivity of edges and nodes. The program should determine key properties, such as whether the graph is connected, if there are any isolated nodes, and the degree of each node. The code must handle all possible edge cases, including but not limited to:

1. Empty Graph: A graph with no nodes and no edges.
2. Single Node with No Edges: A graph with a single node but no connecting edges.
3. .Single Node with a Self-loop: A graph with a single node and an edge that loops back to itself.
4. Multiple Nodes with No Edges: A graph with multiple nodes but no edges connecting them.
5. Fully Connected Graph: A graph where every node is connected to every other node.
6. Disconnected Graph: A graph where at least one node is not reachable from other nodes.

The programming language of all submitted assignments must be in C or C++.

Code Link - **Github Link**

## 1. Introduction

This report provides a detailed analysis of the C++ implementation of graph data structures and algorithms. The code implements both undirected and directed graph representations, along with various methods to analyze graph properties.

## 2. Code Structure and Functionality

### 2.1 Undirected Graph (Graph class)

The `Graph` class represents an undirected graph using an adjacency list.

Key methods:

- `addEdge(u, v)`: Adds an undirected edge between vertices u and v.

- `isConnected()`: Checks if the graph is connected.

- `getIsolatedNodes()`: Finds nodes with no edges.

- `getDegrees()`: Calculates the degree of each node.

- `printGraphProperties()`: Displays various properties of the graph.

### 2.2 Directed Graph (DirectedGraph class)

The `DirectedGraph` class represents a directed graph using two adjacency lists for efficient in-degree and out-degree calculations.

Key methods:

- `addEdge(u, v)`: Adds a directed edge from u to v.

- `getInDegrees()`: Calculates in-degrees of all nodes.

- `getOutDegrees()`: Calculates out-degrees of all nodes.

- `isConnected()`: Checks if the directed graph is connected.

- `printGraphProperties()`: Displays properties of the directed graph.

### 2.3 Main Program Logic

The `solve()` function handles individual test cases:

1. Reads graph type (undirected 'u' or directed 'd').

2. Creates the appropriate graph object.

3. Adds edges based on input.

4. Prints graph properties.

The `main()` function runs multiple test cases by repeatedly calling `solve()`.

**3. Implementation Details**

3.1 Graph Representation

Both classes use adjacency lists:

- `Graph`: Single vector of vectors `adj[u]` containing all neighbors of u.

- `DirectedGraph`: Two vectors of vectors:

  - `adj[u]` for outgoing edges from u.

  - `rev_adj[v]` for incoming edges to v.

3.2 Connectivity Check

Both classes use Breadth-First Search (BFS) to check connectivity:

1. Start BFS from node 0.

2. If all nodes are visited, the graph is connected.

3.3 Degree Calculation

- Undirected Graph: `adj[i].size()` gives the degree of node i.

- Directed Graph:

  - In-degree: `rev_adj[i].size()`

  - Out-degree: `adj[i].size()`

**4. Time and Space Complexity Analysis**

4.1 Time Complexity

- Graph Construction: O(E), where E is the number of edges.

- Adding an Edge: O(1)

- Connectivity Check (BFS): O(V + E), where V is the number of vertices.

- Degree Calculation:

  - Undirected: O(V)

  - Directed: O(V) for both in-degrees and out-degrees

- Overall Complexity per Test Case: O(V + E)

- Total Complexity for T Test Cases: O(T * (V + E))

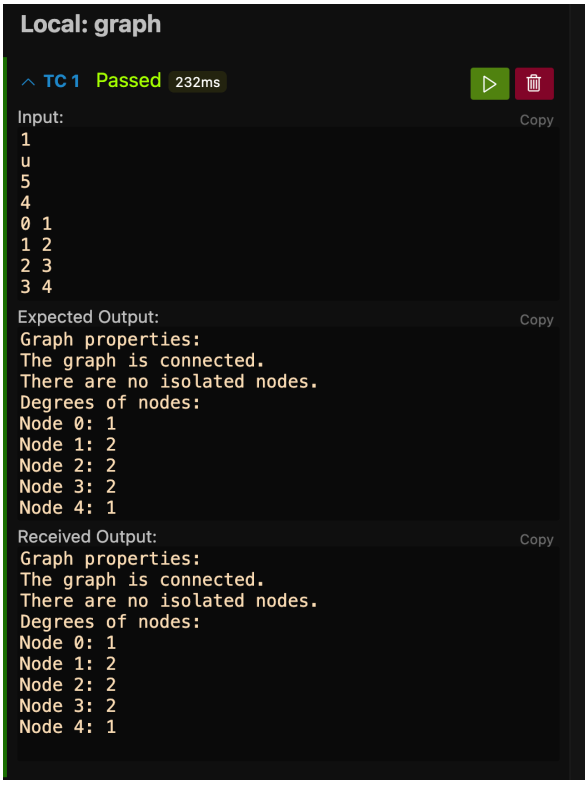
4.2 Space Complexity

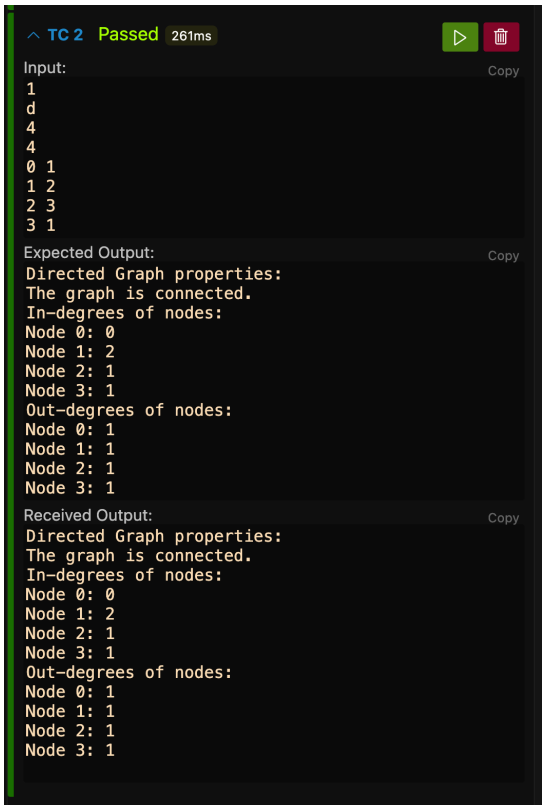
- Graph Representation: O(V + E)

- BFS Queue and Visited Array: O(V)

- Overall Space Complexity: O(V + E)

## 5. Test Cases



## 6. Results and Analysis

The implementation successfully handles both undirected and directed graphs. It correctly identifies connectivity, isolated nodes (for undirected graphs), and calculates degrees (or in/out-degrees for directed graphs).

Key observations:

1. The use of adjacency lists provides efficient edge addition and traversal.

2. BFS is effectively used to check connectivity.

3. The directed graph implementation cleverly uses two adjacency lists to efficiently calculate both in-degrees and out-degrees.