Type of Graph

Graphs can be categorized based on their edge properties and overall structure:

- **Directed vs. Undirected Edges:**
    - **Directed:** Edges have a specific direction, meaning the connection flows from one vertex to another.
    - **Undirected:** Edges are bidirectional, meaning a connection between two vertices implies a connection in both directions.
- **Complete vs. Not Complete Graphs:**
    - **Complete Graph:** A graph where there is an edge between every pair of distinct vertices.
    - **Not Complete Graph:** A graph where at least one pair of vertices lacks a direct edge connection.
- **Connected vs. Not Connected Graphs:**
    - **Connected Graph:** A graph where it is possible to visit any vertex from any other vertex.
    - **Not Connected Graph:** A graph where some vertices are unreachable from others.

Graph Representations

Different methods are used to store graph data, each suited for varying graph structures and operational needs:

- **Adjacency Matrix:** A 2D array where an entry indicates the presence or absence of an edge between two vertices.
- **Incidence Matrix:** A 2D array that maps vertices to edges, indicating which vertices are endpoints of which edges.
- **Adjacency List:** A collection where each vertex maintains a list of its directly connected neighbors.

Graph Traversal Algorithms

These algorithms systematically explore the vertices and edges of a graph:

- **Depth First Search (DFS):** Explores as deeply as possible along each path before backtracking.

- **Breadth First Search (BFS):** Explores all neighbors at the current level before moving to the next level of neighbors, effectively exploring the graph layer by layer.

Shortest Path Algorithms

These algorithms are designed to find the path with the minimum total weight or distance between specified vertices:

- **Dijkstra’s Algorithm:** Computes the shortest path from a single source vertex to all other vertices in graphs with non-negative edge weights.

Minimum Spanning Tree (MST) Algorithms

An MST is a subgraph that connects all vertices in a weighted, undirected graph with the minimum possible total edge weight, without forming any cycles:

- **Kruskal’s Algorithm:** Constructs the MST by iteratively adding edges in increasing order of weight, ensuring no cycles are formed.
- **Prim’s Algorithm:** Builds the MST by starting from an arbitrary vertex and continuously adding the smallest-weight edge that connects a vertex in the growing tree to one outside it.

Applications of Graphs

Graph theory is widely applied across various fields:

- **Communication Networks:** Modeling and optimizing network structures.
- **Circuit Design:** Representing and analyzing electronic circuit layouts.
- **Highway Layouts:** Planning and optimizing transportation routes and infrastructure.

```cpp
#include <iostream>
#include <vector>

using namespace std;

class GraphMatrix
{
    int numVertices;
    vector<vector<int>> matrix;

public:
    GraphMatrix(int vertices)
    {
        numVertices = vertices;
        matrix.resize(vertices, vector<int>(vertices, 0));
    }

    void addEdge(int src, int dest, bool directed = false)
    {
        matrix[src][dest] = 1;
        if (!directed)
        {
            matrix[dest][src] = 1;
        }
    }

    void printGraph()
    {
        for (int i = 0; i < numVertices; i++)
        {
            for (int j = 0; j < numVertices; j++)
            {
                cout << matrix[i][j] << " ";
            }
            cout << endl;
        }
    }
};
#include <iostream>
#include <vector>
#include <list>
#include <queue>
#include <stack>

using namespace std;

class Graph
{
    int numVertices;
    vector<list<int>> adjList;

public:
    Graph(int vertices)
    {
        numVertices = vertices;
        adjList.resize(vertices);
    }

    void addEdge(int src, int dest, bool directed = false)
    {
        adjList[src].push_back(dest);
        if (!directed)
        {
            adjList[dest].push_back(src);
        }
    }

    void printGraph()
    {
        for (int i = 0; i < numVertices; i++)
        {
            cout << i << ":";
            for (int neighbor : adjList[i])
            {
                cout << " -> " << neighbor;
            }
            cout << endl;
        }
    }

    void DFS(int startVertex)
    {
        vector<bool> visited(numVertices, false);
        DFSUtil(startVertex, visited);
        cout << endl;
    }

    void DFSUtil(int v, vector<bool> &visited)
    {
        visited[v] = true;
        cout << v << " ";

        for (int neighbor : adjList[v])
        {
            if (!visited[neighbor])
            {
                DFSUtil(neighbor, visited);
            }
        }
    }

    void BFS(int startVertex)
    {
        vector<bool> visited(numVertices, false);
        queue<int> q;

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

        while (!q.empty())
        {
            int current = q.front();
            q.pop();
            cout << current << " ";

            for (int neighbor : adjList[current])
            {
                if (!visited[neighbor])
                {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
        cout << endl;
    }
};
int main()
{
    Graph g(5);

    g.addEdge(0, 1);
    g.addEdge(0, 4);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 3);
    g.addEdge(3, 4);

    g.printGraph();

    g.DFS(0);
    g.BFS(0);

    return 0;
}
```