Graph Algorithms

1 What is a Graph?

A **graph** is a mathematical structure used to model pairwise relations between objects. A graph G consists of:

- A set of **vertices** (or nodes), V.
- A set of **edges**, E, representing connections between pairs of vertices.

Graphs may be:

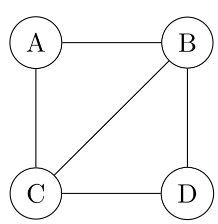
- **Undirected** or **Directed**
- **Weighted** or **Unweighted**
- **Cyclic** or **Acyclic**
- **Connected** or **Disconnected**

2 Why Graphs?

Graphs are widely used in:

- Computer networks (routers as vertices, links as edges)
- Social networks (people as vertices, relationships as edges)
- Maps and GPS routing (cities and roads)
- Scheduling and dependency analysis

3 Example Graph



4 Common Terminology

- **Vertex (Node):** A point in the graph.
- **Edge:** A connection between two vertices.
- **Degree:** Number of edges incident to a vertex.

- **Path:** A sequence of edges connecting a sequence of vertices.
- **Cycle:** A path that begins and ends at the same vertex.
- **Connected Graph:** A graph where there's a path between any two vertices.

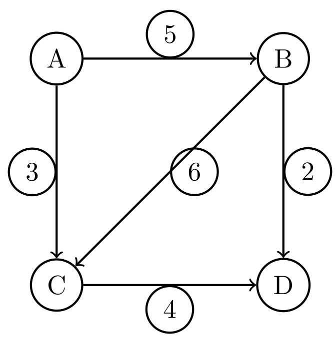
5 Directed vs. Undirected Graphs

- In an **undirected graph**, edges have no direction. An edge (u, v) implies that u is connected to v and vice versa.
- In a **directed graph (digraph)**, each edge has a direction. An edge (u, v) goes from u to v only.

Edge Weights

Some graphs associate a numerical value (called a **weight**) with each edge. These are known as **weighted graphs**. Weights might represent distances, costs, capacities, etc.

Illustration



Notes:

- The arrows indicate that the graph is directed.
- Numbers on the edges represent weights.
- If the same edges had no arrows and were bidirectional, it would represent an undirected weighted graph.

6 Graph Representation

1. Adjacency Matrix

- A 2D matrix of size $n \times n$, where n is the number of vertices.
- Entry (i, j) is 1 (or weight w) if there is an edge from i to j.

```
// Adjacency Matrix Representation
int n = 4; // number of vertices
vector<vector<int>> adjMatrix(n, vector<int>(n, 0));

// Add edge between u and v
adjMatrix[u][v] = 1;
adjMatrix[v][u] = 1; // if undirected
```

2. Adjacency List

- An array of lists, where index i stores the list of adjacent vertices to i.
- More space-efficient for sparse graphs.

```
// Adjacency List Representation
int n = 4; // number of vertices
vector<vector<int>> adjList(n);

// Add edge between u and v
adjList[u].push_back(v);
adjList[v].push_back(u); // if undirected
```

7 Breadth-First Search (BFS)

BFS explores a graph level by level, using a queue. It is useful for finding the shortest path in unweighted graphs.

```
// BFS from source vertex
void bfs(int src, vector<vector<int>>& adj, int n) {
    vector<bool> visited(n, false);
    queue<int> q;

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

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

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

Time Complexity: $O(V + E)$

8 Depth-First Search (DFS)

DFS explores as far as possible along a branch before backtracking. It can be implemented recursively or using a stack.

```
// DFS using recursion
void dfs(int node, vector<vector<int>>& adj, vector<bool>& visited) {
    visited[node] = true;
    cout << node << " ";

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

Time Complexity: $O(V + E)$

9 Applications of BFS and DFS

9.1 BFS for Unweighted Shortest Path

Breadth-First Search (BFS) can be used to compute the shortest path from a source vertex to all other vertices in an **unweighted graph**.

The idea is that BFS explores vertices in layers: it visits all nodes at distance 1 before distance 2, and so on. Thus, the first time we visit a node, we've found the shortest path to it in terms of number of edges.

```
// Compute shortest path from source using BFS
vector<int> bfsShortestPath(int src, const vector<vector<int>>& adj, int n) {
    vector<int> dist(n, -1);  // -1 indicates unreachable
    queue<int> q;

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

    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;
}
```

Note: This approach does not handle edge weights. Use Dijkstra's algorithm if weights are involved.

9.2 Checking Graph Connectivity

To determine if a graph is **connected**, we can use either BFS or DFS:

- Start from any node and perform BFS or DFS.
- After traversal, check if all nodes were visited.

```
// Simple DFS function for connectivity
void dfs(int u, const vector<vector<int>>& adj, vector<bool>& visited, int& count) {
    visited[u] = true;
    count++;
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs(v, adj, visited, count);
        }
    }
}

// Returns true if the graph is connected
bool isConnected(const vector<vector<int>>& adj, int n) {
    vector<bool> visited(n, false);
    int count = 0;

    dfs(0, adj, visited, count); // Start from node 0

    return count == n;
}
```

Note on Directed Graphs:

- A directed graph is **strongly connected** if there is a path from every vertex to every other vertex following the direction of edges.

- It is **weakly connected** if the graph is connected when edge directions are ignored (i.e., treated as undirected).

Determining strong connectivity requires more advanced algorithms such as Kosaraju's or Tarjan's algorithm.

9.3 Topological Sorting via DFS

Topological sorting is applicable to **Directed Acyclic Graphs (DAGs)**. It produces a linear ordering of vertices such that for every directed edge $(u \to v)$, vertex u appears before v in the ordering.

DFS-based topological sorting:

- Perform DFS on the graph.
- On finishing a node, push it to a stack or prepend to a list.
- Reverse the result at the end.

```cpp
// Topological sort using DFS
void topoDFS(int u, vector<vector<int>>& adj, vector<bool>& visited, vector<int>& result) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v])
            topoDFS(v, adj, visited, result);
    }
    result.push_back(u); // Add after visiting children
}

vector<int> topologicalSort(int n, vector<vector<int>>& adj) {
    vector<bool> visited(n, false);
    vector<int> result;

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

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

9.4 Spanning Trees and BFS/DFS Construction

A **spanning tree** of a connected, undirected graph is a subgraph that:

- Includes all the vertices of the original graph,
- Is a tree (i.e., contains no cycles),
- Has exactly $n - 1$ edges if there are n vertices.

Spanning trees are fundamental in graph theory and network design (e.g., building minimum-cost networks without cycles).

Key Property: Every connected undirected graph has at least one spanning tree.

Constructing a Spanning Tree Using BFS

You can build a spanning tree by performing a BFS and recording the edges used to first discover each vertex.

```cpp
// Returns list of edges in the BFS spanning tree
vector<pair<int, int>> bfsSpanningTree(int start, const vector<vector<int>>& adj, int n) {
    vector<bool> visited(n, false);
    vector<pair<int, int>> treeEdges;
    queue<int> q;

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

    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                treeEdges.push_back({u, v});
                q.push(v);
            }
        }
    }

    return treeEdges;
}
```

Constructing a Spanning Tree Using DFS

Likewise, DFS can be used to form a spanning tree by recording edges used during the traversal.

```cpp
// Returns list of edges in the DFS spanning tree
void dfsSpanningTree(int u, const vector<vector<int>>& adj, vector<bool>& visited,
                     vector<pair<int, int>>& treeEdges) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            treeEdges.push_back({u, v});
            dfsSpanningTree(v, adj, visited, treeEdges);
        }
    }
}
```

To use:

```cpp
vector<bool> visited(n, false);
vector<pair<int, int>> treeEdges;
dfsSpanningTree(0, adj, visited, treeEdges);
```

Note: These methods generate *some* spanning tree, not necessarily a minimum spanning tree (MST). For MSTs, use Prim's or Kruskal's algorithm.

10 Advanced Graph Algorithms: Greedy Approach

Many important graph problems can be solved efficiently using the **greedy paradigm**, where a local optimal choice is made at each step with the hope that it leads to a global optimum.

Two key graph algorithms based on this idea are:

- **Prim's Algorithm** for finding a Minimum Spanning Tree (MST),
- **Dijkstra's Algorithm** for finding the shortest path from a source node.

10.1 Prim's Algorithm: Minimum Spanning Tree

Prim's algorithm constructs a **minimum spanning tree** by growing the tree one edge at a time. At each step, it adds the minimum weight edge that connects a vertex in the tree to a vertex outside the tree.

Greedy Principle: At each step, choose the lightest edge that expands the tree without forming a cycle.

```cpp
// Prim's algorithm without priority queue
vector<int> primMST(const vector<vector<pair<int, int>>>& adj, int n) {
    vector<bool> inMST(n, false);
    vector<int> key(n, INT_MAX);  // Minimum weight to include node
    vector<int> parent(n, -1);    // Store MST edges
    key[0] = 0;

    for (int count = 0; count < n; ++count) {
        // Find the vertex u not in MST with minimum key[u]
        int u = -1;
        for (int i = 0; i < n; ++i) {
            if (!inMST[i] && (u == -1 || key[i] < key[u]))
                u = i;
        }

        inMST[u] = true;

        for (auto [v, weight] : adj[u]) {
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
            }
        }
    }

    return parent; // parent[i] gives the MST edge to i
}
```

Time Complexity: $O(V^2)$ due to the repeated minimum selection via linear scan.

Time Complexity: $O((V + E) \log V)$ with a priority queue. **Input:** Undirected, connected, weighted graph.

10.2 Dijkstra's Algorithm: Single-Source Shortest Path

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights.

Greedy Principle: At each step, pick the closest unvisited vertex and update distances to its neighbors.

```cpp
// Dijkstra's algorithm without priority queue
vector<int> dijkstra(int src, const vector<vector<pair<int, int>>>& adj, int n) {
    vector<int> dist(n, INT_MAX);
    vector<bool> visited(n, false);
    dist[src] = 0;

    for (int count = 0; count < n; ++count) {
        // Find unvisited vertex u with smallest dist[u]
        int u = -1;
        for (int i = 0; i < n; ++i) {
            if (!visited[i] && (u == -1 || dist[i] < dist[u]))
                u = i;
        }

        visited[u] = true;

        for (auto [v, weight] : adj[u]) {
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    return dist;
}
```

Time Complexity: $O(V^2)$ — acceptable for dense graphs or small inputs.

10.3 Why Greedy Works

Both Prim's and Dijkstra's algorithms build solutions incrementally:

- They always pick the next node or edge that seems best *at the moment.*
- They avoid revisiting or backtracking once a decision is made.
- Correctness is guaranteed due to the properties of MSTs (Prim) and non-negative weights (Dijkstra).

10.4 How Heaps Improve Efficiency

Both Prim's and Dijkstra's algorithms involve repeatedly selecting the vertex with the smallest key/distance. This operation is costly with a linear scan ($O(V)$ per iteration).

By using a **min-heap (priority queue)**, we reduce this selection time to $O(\log V)$:

- **Prim's with heap:** $O((V + E) \log V)$
- **Dijkstra with heap:** $O((V + E) \log V)$

11 Other Graph Algorithms (Brief Overview)

- **Kruskal's Algorithm:** Greedy algorithm for finding a minimum spanning tree using edge sorting and union-find.
- **Bellman-Ford Algorithm:** Computes shortest paths from a source, allowing negative edge weights.
- **Floyd-Warshall Algorithm:** Computes all-pairs shortest paths using dynamic programming.
- **Topological Sort (Kahn's Algorithm):** Iterative method to produce a topological ordering of a DAG.
- **Kosaraju's Algorithm:** Detects strongly connected components in a directed graph.
- **Tarjan's Algorithm:** Finds strongly connected components in a single DFS traversal.
- **Union-Find (Disjoint Set Union):** Efficient structure for handling connectivity queries and cycle detection.
- **Hamiltonian Cycle:** A cycle that visits each vertex exactly once and returns to the starting vertex.
- **Eulerian Cycle:** A cycle that visits every edge exactly once and returns to the start (exists if all degrees are even in an undirected graph).

12 Practice problem

Leetcode Problem 1971: *Find if Path Exists in Graph*

`https://leetcode.com/problems/find-if-path-exists-in-graph/description/`