Kruskal's Algorithm to Find the Minimum Spanning Tree (MST)

Step 1: Initialization

- Initialize an empty set to store the edges of the MST.
- Sort all edges in ascending order by their weight.

Step 2: Process Each Edge

- For each edge `(u, v)` in the sorted list of edges:
    - If adding the edge `(u, v)` does not form a cycle in the MST (check if `u` and `v` are in the same set):
        - Add the edge `(u, v)` to the MST.

Step 3: Repeat Until MST is Complete

- Repeat this process until the MST contains `n - 1` edges, where `n` is the number of nodes or vertices in the graph.

Step 4: Stop

- The MST is now complete, and the algorithm terminates.

Prim's Algorithm to Find the Minimum Spanning Tree (MST)

Step 1: Initialization

- Choose any vertex as the starting point of the algorithm.
- Initialize an empty set for the MST.
- Mark the chosen vertex as included in the MST.

Step 2: Repeat Until All Vertices Are Included

- While not all vertices are included in the MST:
    - Find the smallest edge that connects a vertex in the MST to a vertex outside the MST.
    - Add this edge and the new vertex to the MST.

Step 3: Stop

- The algorithm stops when all vertices are included in the MST, and the MST contains `n - 1` edges, where `n` is the total number of vertices.

Dijkstra's Algorithm

Step 1: Initialization

- Set the distance to the starting vertex as 0.
- Set the distance to all other vertices as infinity (∞).
- Mark all vertices as unvisited.

Step 2: Repeat Until All Vertices Are Visited

- While there are unvisited vertices:
    a. Pick the unvisited vertex with the smallest distance.
    b. For each neighboring vertex of the selected vertex, calculate the new distance.
        - **New Distance** = **Current Distance** + **Edge Weight**.
    c. If the calculated new distance is smaller than the current distance, update the distance.
    d. Mark the current vertex as visited (it won't be checked again).

Step 3: Stop

- The algorithm stops when all vertices have been visited.

Floyd-Warshall Algorithm

Step 1: Initialization

- Define a large value `inf` (for example, 1000000) to represent infinity.
- Create a distance matrix `dis[i][j]`:
  - If `i == j`, then `dis[i][j] = 0` (the distance from a vertex to itself is zero).
  - If there is an edge between vertex `i` and vertex `j` with weight `w`, set `dis[i][j] = w`.
  - If there is no direct edge between `i` and `j`, set `dis[i][j] = ∞` (infinity).

Step 2: Update Distances

- For each vertex `k` acting as an intermediate node:
  - For each pair of vertices `i` and `j`, update the distance matrix as follows:
    - `dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j])`
    - This step checks if a shorter path exists from `i` to `j` through vertex `k`.

Step 3: Stop

- The algorithm stops when all intermediate vertices have been processed.

- The matrix `dis[i][j]` now contains the shortest distance from vertex `i` to vertex `j`, considering all possible paths.

Bellman-Ford Algorithm

Step 1: Start

- Initialize the algorithm.

Step 2: Initialization

- Set the distance to the starting vertex as 0: `distance[start] = 0`.
- Set the distance to all other vertices as infinity (∞): `distance[v] = ∞` for all `v ≠ start`.

Step 3: Repeat (V - 1) Times

- Where `V` is the number of vertices in the graph:
    - For each edge `(u, v)` with weight `w`:
        - If `distance[u] + w < distance[v]`, then update the distance: `distance[v] = distance[u] + w`.

Step 4: Check for Negative Weight Cycle

- After (V - 1) iterations, check for negative weight cycles:
    - For each edge `(u, v)` with weight `w`:
        - If `distance[u] + w < distance[v]`, then a negative weight cycle is detected.

Step 5: Stop

- The algorithm stops, and either the shortest paths from the start vertex to all other vertices are found, or a negative weight cycle is detected.