Recap on Weighted Graphs and Their Components

A **weighted graph** is a graph in which each edge has an associated numerical value called a weight. These weights can represent various things like distances, costs, or capacities depending on the context of the problem.

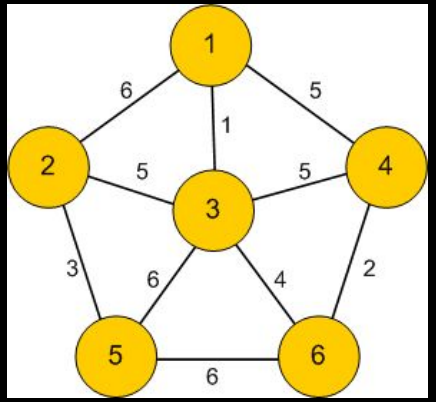
Components of Weighted Graphs

Vertices (Nodes) : Represent entities or objects. Can be labeled or unlabeled.

Edges (Arcs) : Connect pairs of vertices, Can be directed (one-way) or undirected (two-way).

Weights : Numerical values assigned to edges, Represent the cost, distance, or

any quantitative measure between the connected vertices.



Importance of Weighted Graphs

Weighted graphs are crucial in various applications :

- **Navigation systems:** Finding the shortest path between locations.
- **Network design:** Optimizing routes and resources in communication networks.
- **Logistics:** Planning efficient routes for transportation and delivery services.
- **Game development:** Implementing pathfinding algorithms for game characters.

Transition to Dijkstra's Algorithm

Understanding the components and importance of weighted graphs sets the stage for discussing Dijkstra's algorithm, which is a fundamental shortest path algorithm used to find the shortest path from a single source vertex to all other vertices in a weighted graph. This algorithm is particularly useful in scenarios where you need to minimize travel time or cost.

Priority Queue in Dijkstra's Algorithm

Priority Queue : This is a data structure used to efficiently manage and retrieve the next node to process based on the smallest known distance from the source. It ensures that the node with the smallest tentative distance is always processed next.

Here are the key steps involved in Dijkstra's Algorithm, highlighting the role of the priority queue:

Initialization : Set the distance of the source node to 0 and all other nodes to infinity, Insert the source node into the priority queue.

Extract Minimum : Remove the node with the smallest distance from the priority queue (starting with the source).

Relaxation : For each neighboring node of the extracted node, If the distance to the neighbor can be shortened through the extracted node, update the neighbor's distance, then it must be inserted or updated in the priority queue to ensure it can be processed in the future. The priority queue keeps track of nodes based on their current shortest distance.

Repeat : Repeat the extract and relax steps until the priority queue is empty.

Termination : The algorithm finishes when all nodes have been processed, resulting in the shortest paths from the source to all other nodes.