NAME: THEURI BONFACE KARUE

REG_NO: SCT211-0573/2022

UNIT: ADTs & ALGORITHMS

UNIT_CODE:ICS 2300

ASSIGNMENT : GRAPHS

QUESTION

Describe giving examples the following algorithms that calculate the shortest path between the vertices of a graph G:

a) Minimum spanning tree

b) Dijkstra's algorithm

c) Warshall's algorithm

Hint: Using ChatGPT make summary notes on the following and submit in 1 Weeks time

1. Minimum spanning tree

Purpose: Unlike shortest-path algorithms, an MST connects all vertices in a graph with the minimum possible total edge weight without forming any cycles.

Key characteristics:

- Creates a tree that connects all vertices with minimum total weight
- Contains exactly V-1 edges (where V is the number of vertices)
- Has no cycles

How it works:

1. Start from any vertex
2. Repeatedly select the minimum weight edge that connects a visited vertex to an unvisited one
3. Add the selected edge to the MST
4. Continue until all vertices are connected

Example Algorithms:

- **Prim's Algorithm**: Starts from an arbitrary vertex and grows the spanning tree by adding the smallest edge that connects the tree to a new vertex.
- **Kruskal's Algorithm**: Sorts all edges by weight and adds them one by one to the tree, avoiding cycles.

Example:

Consider the following weighted graph:
Vertices: A, B, C, D
Edges and weights:

- A-B: 4
- A-C: 2
- B-C: 5
- B-D: 1
- C-D: 8

Using Prim's Algorithm:

1. Start at A. Add the smallest edge, A-C (weight 2).
2. From C, add the smallest connecting edge, C-B (weight 5).
3. From B, add the smallest connecting edge, B-D (weight 1).

MST: Edges are A-C, C-B, and B-D with a total weight of **2 + 5 + 1 = 8**.

Use Case: Network design problems like building efficient road networks or communication links.

2. Dijkstra's Algorithm

- **Purpose**: Finds the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights.

Key characteristics:

- Works with weighted graphs
- Finds shortest paths from a single source
- Cannot handle negative weights
- Uses a priority queue for efficiency

- **Steps**:
    1. Start with the source vertex, assigning it a distance of 0, and set all other distances to infinity.
    2. Use a priority queue to repeatedly select the vertex with the smallest tentative distance.
    3. Update distances to neighboring vertices if a shorter path is found.
    4. Mark the vertex as visited
    5. Repeat until all vertices are visited

Example

- Source vertex: A
- Edges and weights:
    - A-B: 1
    - A-C: 4
    - B-C: 2
    - B-D: 6
    - C-D: 3

Steps:

1. Start at A. Set distance to A as 0, and all others as infinity.
2. Visit A. Update distances: B = 1, C = 4.
3. Visit B (smallest distance). Update distances: C = 3 (via B), D = 7 (via B).
4. Visit C. Update distance: D = 6 (via C).
5. Visit D. No updates needed.

Shortest distances: A-B = 1, A-C = 3, A-D = 6.

- **Example**: Calculating shortest delivery routes for logistics companies.
- **Limitation**: Inefficient for graphs with negative edge weights.

3. Warshall's Algorithm

- **Purpose**: Computes the transitive closure of a directed graph or finds shortest paths between all pairs of vertices in a graph (Floyd-Warshall).

Key characteristics:

- Finds shortest paths between all pairs of vertices
- Can handle negative edge weights (but not negative cycles)
- Uses dynamic programming
- Time complexity is $O(V^3)$

How it works:

1. Create a distance matrix with direct edge weights
2. For each intermediate vertex k:
   - For each pair of vertices (i,j):
     - Check if path through k is shorter than current path
     - If so, update the shortest path

Example:

Graph with vertices A, B, C:
Initial distance matrix:

	A	**B**	**C**
A	0	3	∞
B	∞	0	1
C	4	∞	0

Steps:

1. Use A as an intermediate vertex: Update B to C via A.
2. Use B as an intermediate vertex: Update A to C via B.
3. Use C as an intermediate vertex: Update B to A via C.

Final distance matrix:

	A	**B**	**C**
A	0	3	4
B	5	0	1
C	4	7	0

Result: Shortest paths between all pairs of vertices.

- **Example**: Finding shortest paths in a city grid network for all source-destination pairs.
- **Limitation**: May not work with graphs having negative weight cycles.

Further notes

Their Key Differences.

The following section focuses on their key differences.

Aspect	**Minimum**	**Dijkstra's**	**Warshall's**

	Spanning Tree (MST)	Algorithm	Algorithm (Floyd-Warshall)
1. Purpose	Finds a tree that connects all vertices with the minimum total edge weight.	Finds the shortest path from a single source to all other vertices.	Finds the shortest paths between all pairs of vertices.
2. Graph Type	Undirected, weighted graphs.	Directed/undirected, weighted graphs (non-negative weights).	Directed/undirected, weighted graphs (may include negative weights, but no negative weight cycles).
3. Algorithm Type	Greedy algorithm.	Greedy algorithm.	Dynamic programming.
4. Output	A spanning tree (set of edges).	Shortest distances from the source to all vertices.	Shortest distances between all pairs of vertices.
5. Edge Weights	Requires positive weights.	Requires non-negative weights.	Can handle negative weights but not negative cycles.
6. Applications	Network design (e.g., road, electrical, or communication networks).	Navigation, routing (e.g., GPS systems).	Network analysis, all-pairs shortest path in dense graphs.
7. Efficiency	Efficient for sparse graphs (e.g., Prim's or Kruskal's algorithm).	Efficient for sparse graphs with priority queues.	Suitable for dense graphs.
8. Complexity	**O**(**E**log**V**) for Prim/Kruskal	**O**(**V**^2) or **O**(**V**log**V**+**E**)with priority queue.	**O**(**V**^3) for adjacency matrix implementation.