FOUNDATIONS OF ALGORITHM DESIGN AND MACHINE LEARNING – NOTES

DATE: 28th January 2025 | TIME SLOT: 11AM-12PM

Directed Weighted Acyclic Graphs

Introduction:
Shortest path algorithms are fundamental in graph theory, used for finding the most efficient route between nodes. In this chapter, we will discuss the shortest path problem in Directed Weighted Acyclic Graphs (DAGs) and explain an efficient approach to solving it.

Problem Statement:
Given a Directed Weighted Acyclic Graph (DAG) and a source node, we aim to find the shortest path to a destination node. Unlike general graphs, where Dijkstra's or Bellman-Ford algorithms are used, DAGs allow for a more optimized approach using Topological Sorting.

Algorithm: Shortest Path in DAG
The algorithm implements **Shortest Path in a Directed Acyclic Graph (DAG)** using a **priority queue (min-heap)** to always process nodes in the order of their **shortest known distance** from the source node.

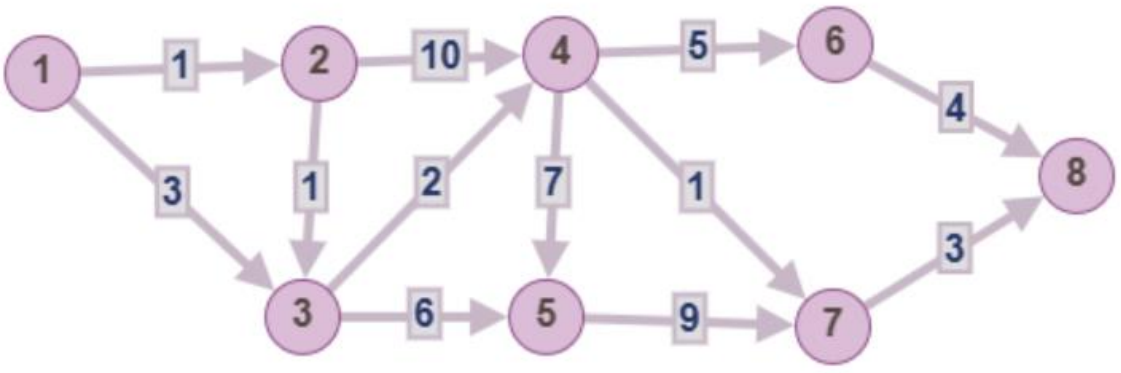
Pseudo Code:

```
SP(G, s, g)
{
   Initialize dist[] to ∞ for all nodes except dist[s] = 0
   visit[s] = False
   Q <- {(s, 0)}  // Priority queue storing (node, cost)

   while (Q !empty)
   {
      (n, val) <- extract min(Q)  // Extract node with smallest distance
      visit[n] = True

      for all m in successors(n)
      {
         if (!visit[m]) INSERT(Q, (m, dist[m]))

         if m already in Q and dist[n] + cost(n → m) < dist[m]
         {
            dist[m] = dist[n] + cost(n → m)  // Update cost
            UPDATE(Q, (m, dist[m]))  // Update priority queue with new cost
         }
      }
   }
}
```

Algorithm Explanation:
The algorithm uses a **queue** to process nodes in increasing order of their shortest distance from the source. The key steps are:

1. **Initialization**:
    - Create a dist[] array initialized to ∞ for all nodes except the source node (dist[1] = 0).
    - Create a visit[] array initialized to False for all nodes.
    - Initialize a queue Q and enqueue the source node (1, dist[1]).
2. **Processing Nodes**:
    - While the queue is not empty:
        - Extract the node with the smallest distance from the queue.
        - Mark it as visited.
        - Relax all outgoing edges from this node:
            - If a shorter path is found to a neighbour, update its distance and enqueue it into the queue.

Explanation through an example:



We will compute the shortest path from node 1 to node 8 in the given **DAG** using the **queue-based approach.**

Initialisation:

- The dist array (distance from source node 1 to all nodes) is initialized as: dist = [0, ∞, ∞, ∞, ∞, ∞, ∞, ∞]
- The visit array (whether a node has been processed) is initialized as:
  visit = [False, False, False, False, False, False, False, False]
- The queue starts with the source node:
  Q = [(1, 0)]

Processing Node 1:

- Extract node **1** from the queue.
- Mark node **1** as visited.

visit = [True, False, False, False, False, False, False, False]

- Relax outgoing edges from node **1**:
  - Edge (1 → 2) updates dist[2] to **1** and enqueues (2, 1).
  - Edge (1 → 3) updates dist[3] to **3** and enqueues (3, 3).

  dist = [0, 1, 3, ∞, ∞, ∞, ∞, ∞]
  Q = [(2, 1), (3, 3)]

Processing Node 2:

- Extract node **2** from the queue.
- Mark node **2** as visited.
  visit = [True, True, False, False, False, False, False, False]
- Relax outgoing edges from node **2**:
  - Edge (2 → 4) updates dist[4] to **11** and enqueues (4, 11).
  - Edge (2 → 3) updates dist[3] from **3** to **2**.

    dist = [0, 1, 2, 11, ∞, ∞, ∞, ∞]
    Q = [(3, 2), (4, 11)]

Processing Node 3:

- dist = [0, 1, 2, 4, 8, ∞, ∞, ∞]
  Q = [(4, 4), (5, 8)]

Processing Node 4:

- dist = [0, 1, 2, 4, 8, 9, 5, ∞]
  Q = [(5, 8), (6, 9), (7, 5)]

Processing Node 7:

- dist = [0, 1, 2, 4, 8, 9, 5, 8]
  Q = [(5, 8), (6, 9), (8, 8)]

Processing Node 5:

- dist = [0, 1, 2, 4, 8, 9, 5, 8]
  Q = [(6, 9), (8, 8)]

Processing Node 8:

- dist = [0, 1, 2, 4, 8, 9, 5, 8]
  Q = [(6, 9)]

Processing Node 8:

- Edge (6 → 8) does not update dist[8] since it is already **8**.
  dist = [0, 1, 2, 4, 8, 9, 5, 8]
  Q = []

The shortest path from node 1 to 8 is: 1 → 2 → 3 → 4 → 7 → 8 with a total cost of 8

Time Complexity:

The time complexity of finding the shortest path in a Directed Acyclic Graph (DAG) using the topological sorting and edge relaxation approach is ***O*(*V*+*E*)**, where:

- V: Number of vertices (nodes) in the graph.
- E: Number of edges in the graph.

<u>Recursive Formulation for Dijkstra's Shortest Path</u>

The recursive formulation of Dijkstra's algorithm provides a mathematical way to compute the shortest path from a source node s*s* to a goal node g*g* in a weighted graph. It is based on the principle of **optimal substructure**, which means that the shortest path from **s to g** can be broken down into smaller subproblems involving intermediate nodes.
This formulation is particularly useful for understanding the theoretical underpinning of Dijkstra's algorithm and can also be adapted to handle cases involving cyclic graphs.

Recursive Formula:

The shortest path $SP(G, s, g)$ is defined as:

$$SP(G,s,g) = \begin{cases} 0 & \text{if } s = g \\ \infty & \text{if } G = NULL \\ \min_{m \in succ(s)} \left(wt(s,m) + SP(G - \{s\}, m, g) \right) & \text{otherwise} \end{cases}$$

Explanation of Terms:

1. Base Cases:
   - If s=*g*: The shortest path from a node to itself is 0.
   - If G=NULL: This means there are no nodes or edges left in the graph, and the goal node is unreachable. The shortest path is infinity (∞).
2. Recursive Case:
   - For each successor m of node s:
     - Compute the cost of traveling from s to m, which is given by the edge weight wt(s,m).
     - Add this cost to the shortest path from m to the goal node g, which is recursively computed as SP(G−{s},m,g).
   - Take the minimum value across all successors m, as we are looking for the shortest path.

Key Observations:

- The recursive approach works by exploring all possible paths from the source to the goal and selecting the one with the smallest cost.
- It assumes that edge weights are non-negative (a requirement for Dijkstra's algorithm).
- This formulation can also work in cyclic graphs if cycles are properly handled (e.g., by maintaining a visited set).

Steps in Recursive Approach:

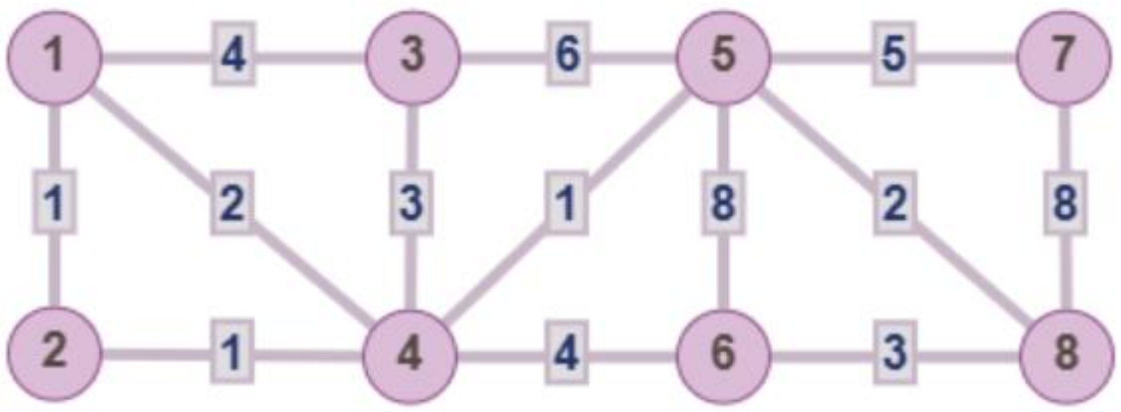
1. Start at the source node s.
2. Explore all successors of s.
3. For each successor m:
   - Compute the cost of traveling from s to m.
   - Recursively calculate the shortest path from m to the goal node g.
4. Return the minimum cost among all possible paths.

Spanning Trees and Minimum Spanning Trees (MST)

Definition:
A **spanning tree** of a graph spans across every vertex of the graph but contains no cycles. A graph can have multiple spanning trees.
Objective:
Our goal is to find the **minimum cost spanning tree** (MST), which is the spanning tree with the least total edge weight.

Applications:
MSTs have numerous applications, including in network design, circuit design, and as a sub-problem in the **Travelling Salesman Problem**.

Now, we will deep dive into the spanning tree problem based on an example:



Prim's Method for spanning tree:

Idea: At each step, select the minimum weight edge that connects a vertex inside the growing spanning tree to a vertex outside, ensuring no cycles are formed.

Pseudocode:

```
 MST(G) {
    Select an arbitrary starting vertex 's';
    G1 = {s}              // G1: Set of vertices included in MST
    G2 = G - {s}          // G2: Remaining vertices

    MST(G1, G2);
}

MST(G1, G2) {
    if (G2 is empty) {
        return (T_G1, c)  // T_G1 is the spanning tree, c is the total cost
    }
    else {
        // Find the minimum weight edge 'e' connecting G1 to G2
        e = minimum edge from G1 to G2;

        G1 = G1 ∪ {e}                          // Add the edge 'e' to G1 (MST set)
        G2 = G2 - {vertex connected by e}  // Remove the vertex from G2

        MST(G1, G2)          // Recursively repeat the process
    }
}
```

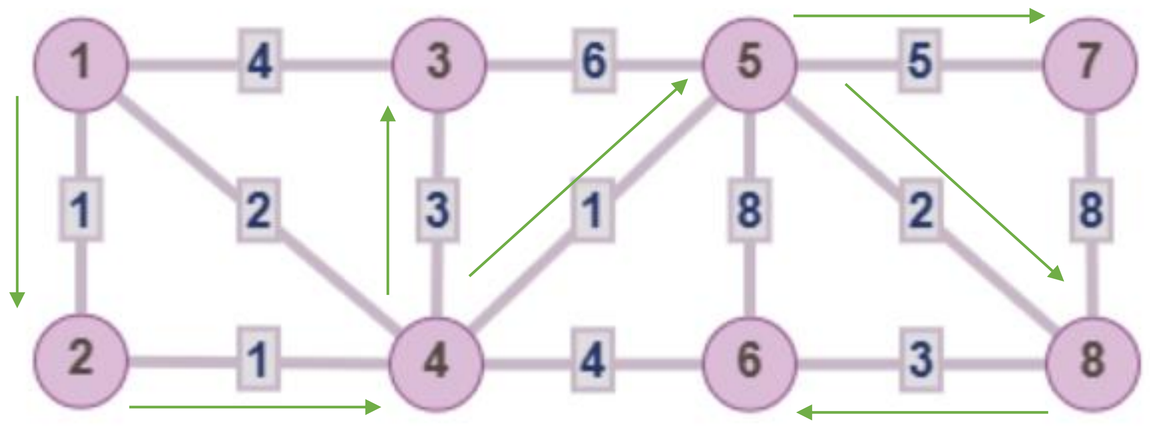
Where:

- G = Original graph
- G1 = Set of vertices included in MST (initially {s}, where s is the starting vertex)
- G2 = Remaining vertices (initially G - {s})

Step-by-Step Execution on Example Graph:

1. Start with vertex 1. G1 = {1}, G2 = {2, 3, 4, 5, 6, 7, 8}.
2. Choose the minimum edge from G1 to G2, which is (1, 2) with weight 1.
3. G1 = {1, 2}, G2 = {3, 4, 5, 6, 7, 8}.
4. Next minimum edge is (2, 4) with weight 1.
5. G1 = {1, 2, 4}, G2 = {3, 5, 6, 7, 8}.
6. Next minimum edge is (4, 5) with weight 1.
7. G1 = {1, 2, 4, 5}, G2 = {3, 6, 7, 8}.
8. Next minimum edge is (5, 8) with weight 2.
9. G1 = {1, 2, 4, 5, 8}, G2 = {3, 6, 7}.
10. Next minimum edge is (8, 6) with weight 3.
11. G1 = {1, 2, 4, 5, 6, 8}, G2 = {3, 7}.
12. Next minimum edge is (4, 3) with weight 3.
13. G1 = {1, 2, 3, 4, 5, 6, 8}, G2 = {7}.
14. Next minimum edge is (5, 7) with weight 5.
15. G1 = {1, 2, 3, 4, 5, 6, 7, 8}, G2 = {}.

Total Cost of MST = 1 + 1 + 1 + 2 + 3 + 3 + 5 = 16



Time complexity for generic greedy algorithm or Prim's algorithm is **O(Elog V)**, where E is the number of edges and V is the number of vertices.

Minimum spanning of trees can also be achieved by **Kruskal's algorithm** which is a **greedy method** for finding the MST. It works by **sorting all edges by weight** and adding the smallest edges to the MST **while avoiding cycles**. The algorithm uses the **union-find (disjoint-set) data structure** to efficiently manage connected components and detect cycles. Time complexity for Kruskal's method is **O(Elog V)**