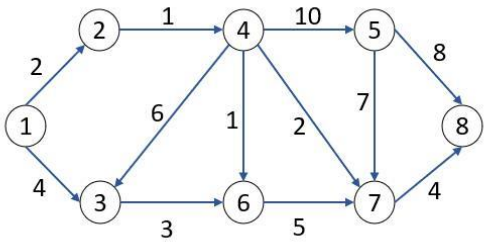
FOUNDATIONS OF ALGORITHM DESIGN AND MACHINE LEARNING
DATE: 30th January 2025

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 using **Depth-First Search (DFS) with Memoization**.



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 **recursive depth-first approach** with **topological sorting** and **memoization** for optimization.

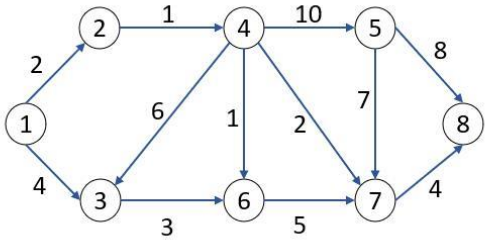
Algorithm: Shortest Path in DAG using Depth-First Recursive Approach

The algorithm implements **Shortest Path in a DAG** using **Depth-First Search (DFS)** and **Memoization**. This approach efficiently finds the shortest path by traversing the graph recursively, storing previously computed results to avoid redundant calculations.

Pseudo Code:

```
visited[N];   // Boolean array to track visited nodes, initially all 0
cost[N];      // Stores the shortest known path to each node, initially ∞
succ[N];      // Adjacency list storing neighbors of each node
C[N][N];      // Cost matrix for edges between nodes


DFSP(G,node, g) {
   if (node == g) {
      cost[node] = 0;  // Goal node has zero cost to itself
   }

   visited[node] = 1;
   int value = ∞;  // Local variable to store the shortest cost

   for each n in succ[node] do {
      if (visited[n] == 0) {
         DFSP(n, g);  // Recursive DFS call to explore further paths
      }
      value = min(value, cost[n] + C[node][n]);
   }

   cost[node] = value;  // Update the shortest known cost to reach this node
   return cost[node];
}
```

Execution of DFSP(G,1, 8)



1. **Start at Node 1**
   - **visited[1] = 1**
   - **Explore neighbors: Node 2 (cost 2) & Node 3 (cost 4)**
2. **Go to Node 2**
   - **visited[2] = 1**
   - **Explore neighbor: Node 4 (cost 1)**
3. **Go to Node 4**
   - **visited[4] = 1**
   - **Explore neighbors: Node 3 (cost 6) , Node 5 (cost 10), Node 6 (cost 1) & Node 7 (cost 2)**
4. **Go to Node 5**
   - **visited[5] = 1**
   - **Explore neighbors: Node 7 (cost 7) & Node 8 (cost 8)**
5. **Go to Node 7**
   - **visited[7] = 1**
   - **Explore neighbor: Node 8 (cost 4)**
6. **Reach Goal Node 8**
   - **cost[8] = 0**
7. **Backtrack & Update Costs**
   - **cost[7] = min(cost[8] + C[7,8]) = min(0 + 4) = 4**
   - **cost[5] = min(cost[7] + C[5,7], cost[8] + C[5,8]) = min(4 + 7, 0 + 8) = 8**
8. **Continue Backtracking**
   - **Compute for cost[6], cost[3], cost[4], cost[2] and cost[1] similarly.**

Final Shortest Path Cost Calculation

After computing all values, the shortest path **from Node 1 to Node 8** will be:

Path: 1 → 2 → 4 → 7 → 8
Total Cost: 9

Time Complexity Analysis:

- **Each node is visited once** in the recursive DFS traversal.
- **Each edge is processed once** when updating distances.
- **Total Complexity**: O(V+E)

This approach is efficient for **DAGs** since it avoids the overhead of priority queues and extra edge relaxations found in **Dijkstra's Algorithm**.

Shortest Path in DAG using Best-First Search

Introduction:

Shortest path algorithms are crucial for efficiently finding the lowest-cost path between nodes. In a **Directed Weighted Acyclic Graph (DAG)**, the **Best-First Search (BFS) technique** can be used to find the shortest path more efficiently by always expanding the **most promising node first** based on a priority queue (min-heap).

Problem Statement:

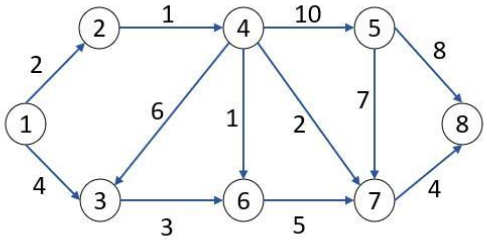
Given a **DAG** and a **source node**, we aim to find the **shortest path to a destination node**. Instead of exploring all paths equally, the **Best-First Search (BFS) technique** prioritizes nodes based on the **minimum cost** encountered so far, always expanding the lowest-cost node first.

A**lgorithm: Shortest Path in DAG using Best-First Search**

This approach utilizes a **priority queue (min-heap)** to always expand the **least-cost node** first.

Pseudo code:

```
SP_BestFirst(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 is not empty)
   {
      (n, val) <- extract_min(Q)  // Extract node with the smallest distance
      visit[n] = True
      for all m in successors(n)
      {
         if (!visit[m]) INSERT(Q, (m, dist[m]))
         if 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
         }}}
   return dist[g]  // Return the shortest path to the goal node}
```



Algorithm Explanation:

The algorithm follows these steps:

1. Initialization:

- **Create a dist[] array** initialized to **∞** for all nodes except the source node (**dist[s] = 0**).
- **Create a visit[] array** initialized to **False** for all nodes.
- **Initialize a priority queue (min-heap)** with the source node (s, 0).

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 neighbor, **update its distance** and **enqueue it** in the priority queue.

3. Computing the Shortest Path:

- The **priority queue ensures** that nodes are processed in the order of **minimum distance**, leading to an optimal solution.

Applying Best-First Search

We will track the **distance** array (dist[]) and **priority queue (Q)** updates carefully.

Step 1: Initialization

- Distance array: dist = [0,∞,∞,∞,∞,∞,∞,∞]
- Priority queue: Q=[(1,0)]

Processing Node	Queue (Q) Before Extraction	Queue (Q) After Updates	Distance (dist[])
1	[(1, 0)]	[(2, 2), (3, 4)]	[0, 2, 4, ∞, ∞, ∞, ∞, ∞]
2	[(2, 2), (3, 4)]	[(3, 4), (4, 3)]	[0, 2, 4, 3, ∞, ∞, ∞, ∞]
4	[(3, 4), (4, 3)]	[(3, 4), (6, 4), (7, 5), (5, 13)]	[0, 2, 4, 3, 13, 4, 5, ∞]
7	[(3, 4), (6, 4), (7, 5), (5, 13)]	[(3, 4), (6, 4), (5, 13), (8, 9)]	[0, 2, 4, 3, 13, 4, 5, 9]
8	[(3, 4), (6, 4), (5, 13), (8, 9)]	[(3, 4), (6, 4), (5, 13)]	[0, 2, 4, 3, 13, 4, 5, 9]

Final Shortest Path Calculation

- The shortest path from **node 1 to node 8** is:
  1 → 2 → 4 → 7 → 8
- **Total cost = 2 + 1 + 2 + 4 = 9**

Thus, the shortest path is **1 → 2 → 4 → 7 → 8** with a cost of **9**.