	Pimpri Chinchwad Education Trust's **Pimpri Chinchwad College of Engineering**
Assignment No: 5	

1. Problem Statement

A logistics company, SwiftCargo, specializes in delivering packages across multiple cities. To optimize its delivery process, the company divides the transportation network into multiple stages (warehouses, transit hubs, and final delivery points). Each package must follow the most cost-efficient or fastest route from the source to the destination while passing through these predefined stages. As a logistics optimization engineer, you must: 1. Model the transportation network as a directed, weighted multistage graph with multiple intermediate stages. 2. Implement an efficient algorithm (such as Dynamic Programming or Dijkstra's Algorithm) to find the optimal delivery route. 3. Ensure that the algorithm scales for large datasets (handling thousands of cities and routes).

4. Analyze and optimize route selection based on real-time constraints, such as traffic conditions, weather delays, or fuel efficiency.

Constraints & Considerations:

- The network is structured into N stages, and every package must pass through at least one node in each stage.
- There may be multiple paths with different costs/times between stages.
- The algorithm should be flexible enough to handle real-time changes (e.g., road closures or rerouting requirements).
- The system should support batch processing for multiple delivery requests

Tasks:

1. Model the network as a directed weighted multistage graph.
2. Use Dynamic Programming (DP) to compute optimal delivery routes.
3. Ensure scalability for thousands of cities/routes.
4. Consider real-time constraints like traffic, weather, or fuel efficiency.

Constraints:

1. Each package must pass through at least one node per stage.
2. Multiple paths may exist between stages.
3. Support batch processing and dynamic rerouting.

2. Course Objective

1. To know the basics of computational complexity of various algorithms.
2. To select appropriate algorithm design strategies to solve real-world problems

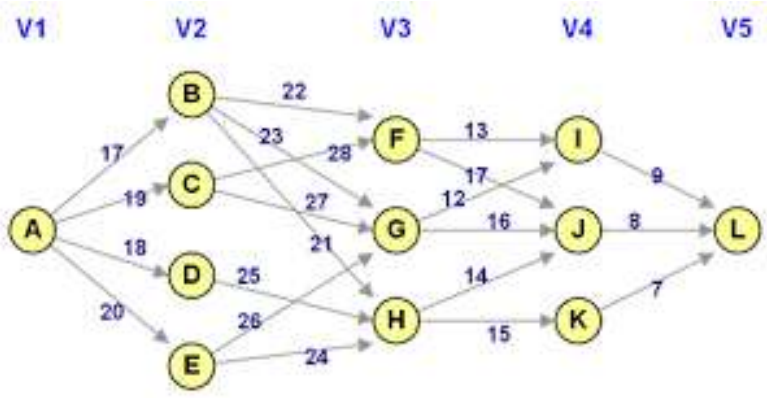
3. Course Outcome

1. Analyze the asymptotic performance of algorithms
2. Generate optimal solutions by applying Dynamic Programming strategy.

4. Theory -

1. Multistage Graph

A multistage graph is a directed graph where the vertices are partitioned into a set of disjoint stages, V1,V2,…,Vk, such that all edges go from a vertex in one stage to a vertex in the next stage. This structure is commonly used to model dynamic programming problems.



- **Structure:** The image displays a directed graph with vertices partitioned into five stages labeled V1 to V5.
    - V1={A} (The source node).
    - V2={B,C,D,E}.
    - V3={F,G,H}.
    - V4={I,J,K}.
    - V5={L} (The sink node).

- **Edges and Costs:** The edges are directed (indicated by arrows) and only connect a vertex in stage Vi to a vertex in the immediately succeeding stage Vi+1. Each edge has an

associated numerical value, which represents the cost or distance of traversing that specific edge.

- **Goal:** This type of graph is typically used to find the shortest path (or sometimes the exactly one vertex from each intermediate stage (V2,V3,V4). This sequential, stage-by-stage progression is what defines it as a "multistage" graph.

- **Application (Dynamic Programming):** The shortest path problem on a multistage graph is a classic appllongest path) from the source node (A in V1) to the sink node (L in V5).
- **Multistage Property:** The critical feature is that a path from A to L must pass through ication of the principle of optimality used in dynamic programming.
    - The problem can be broken down into a sequence of smaller, overlapping subproblems.
    - The shortest path from A to L is found by iteratively calculating the shortest path to each vertex in a stage from all vertices in the previous stage. For example, to find the shortest path to F∈V3, you consider the shortest paths to its predecessors (B,C,D∈V2) and add the cost of the edge from the predecessor to F.
    - The final answer is the shortest path to the sink, L.

- **Use Cases of Multistage Graphs in Real Life**

    - Logistics: Routing parcels through hubs (like in SwiftCargo).
    - Speech Recognition: Word sequence prediction (language models).
    - Manufacturing: Sequential processes with cost/time optimization.
    - Network Routing: Data packet transmission in layered protocols.
    - Finance: Investment planning through time periods.

2. <u>Dynamic Programming (DP)</u>

Definition:
Dynamic Programming is an optimization technique used to solve complex problems by breaking them into smaller overlapping subproblems, solving each subproblem once, and storing its result.

Key Properties:

- Optimal Substructure: The solution to the main problem can be built from solutions to subproblems.

- Overlapping Subproblems: The same subproblems are solved multiple times.

Why DP is Better than Recursion Here:

- Avoids recomputation of same subproblems.
- Works in bottom-up fashion (used in multistage graph).
- Saves time and memory (using memoization or tabulation).

Difference Between Greedy, DP, and Backtracking -

Strategy	**Suitable When**	**Pros**	**Cons**
Greedy	Local choice leads to global optimum	Fast, simple	May give suboptimal solution
DP	Optimal substructure + overlapping subproblems	Guarantees optimal, efficient	More memory use, needs planning
Backtracking	Combinatorial problems with constraints	Finds all/optimal combinations	Slow for large inputs

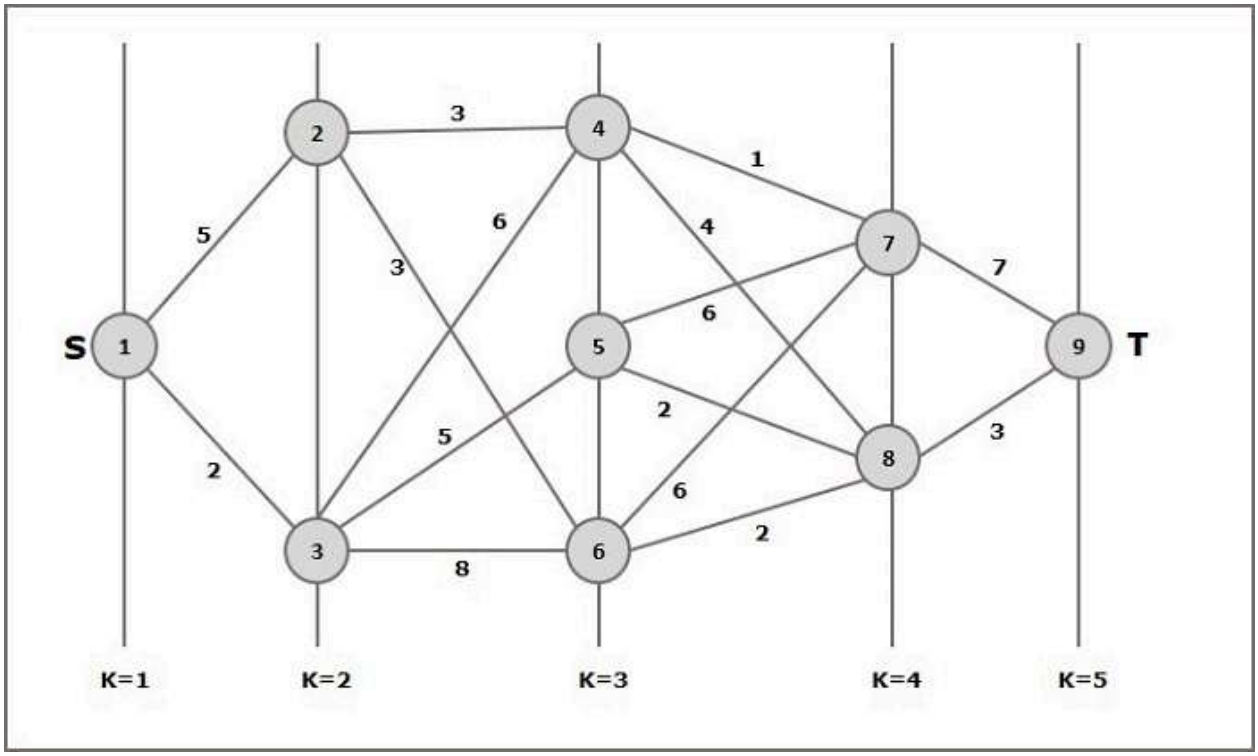
For this problem -

1. Graph is acyclic → no cycles.
2. Problem has optimal substructure → best path from node i depends on best paths from nodes in next stage.
3. DP avoids redundant calculations by storing partial results.
4. Efficient for large multistage networks (thousands of nodes).

3. How DP Works in a Multistage Graph?

1. Number stages 1 → N.
2. Initialize destination cost = 0.
3. Move backward stage by stage:
   $cost[i] = min\{weight(i,j) + cost[j]\}$, where $j \in V_{k+1}$ and $(i,j) \in E$:
4. Store next[] array to track path.
5. At the end, cost[source] gives the minimum cost, and path can be reconstructed.

Example Dry Run -



According to the formula, we have to calculate the cost (i, j) using the following steps

Step 1: Cost (K-2, j)

In this step, three nodes (node 4, 5. 6) are selected as j. Hence, we have three options to choose the minimum cost at this step.

Cost(3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7

Cost(3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5

Cost(3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5

Step 2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9),c(2, 6) +

Cost(6, 8) + Cost(8, 9)} = 8

Cost(2, 3) = {c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8)+ Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)} = 10

Step 3: Cost (K-4, j)

Cost (1, 1) = {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))} = 12

c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13

Hence, the path having the minimum cost is 1→ 3→ 5→ 8→ 9.

- **Algorithm**

Algorithm MultiStageDP(G)
1. cost[destination] = 0
2. for i = n-1 downto 1:
3. cost[i] = ∞
4. for each (i,j) in edges:
5. if cost[i] > weight(i,j) + cost[j]:
6. cost[i] = weight(i,j) + cost[j]
7. next[i] = j
8. print cost[source], path

- **Pseudocode: Multistage Graph Shortest Path (Backward DP)**

This algorithm assumes the graph G=(V,E) is partitioned into K stages, V1 (source s) to VK (sink L). N is the total number of vertices.

SHORTEST_PATH_MULTISTAGE(G,K,V1,…,VK)

- **Initialize Cost and Path Arrays**
  - Create an array cost[1…N] to store the shortest distance from each node i to the sink L.
  - Create an array path[1…N] to store the next node on the shortest path from i to L.
  - Let L be the single node in the final stage VK.
- **Base Case (Stage VK)**
  - cost[L]←0
- **Iterate Backward Through Stages (K−1 down to 1)**
  - **For** stage k from K−1 down to 1 **do**:
    - **For** each node i in Vk **do**:

- min_cost←∞
- min_next_node←NULL
- **For** each successor j of i (where j∈Vk+1 and (i,j)∈E) **do**:
    - current_cost←weight(i,j)+cost[j]
    - **If** current_cost<min_cost **then**:

      Min_cost←current_cost

      min_next_node←j

    - **End If**
- **End For**
- cost[i]←min_cost
- path[i]←min_next_node
- **End For**

- **End For**

- **Construct and Return the Shortest Path**
    - Shortest Path Cost←cost[s] (where s is the source node in V1).
    - i←s
    - **While** i=L **do**:

      Output node i

      i←path[i]

    - **End While**
    - Output node L

Output:

- The minimum cost from the source to the sink.
- The sequence of nodes that form the shortest path.

- **Time & Space Complexity**

Measure	**Complexity / Description**
Time	O(V + E)
Space	O(V)
Suitable for	Large multistage logistics graphs

Code

```
import java.util.*;

class Edge {
    int to;
    double baseCost;
    double curCost;

    Edge(int to, double baseCost) {
        this.to = to;
        this.baseCost = baseCost;
        this.curCost = baseCost;
    }
}

public class SwiftCargoLogistics {

    static final double INF = 1e30;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input: number of stages
        System.out.print("Enter number of stages: ");
        int S = sc.nextInt();

        int[] stageCount = new int[S];
        long N = 0;

        System.out.print("Enter number of nodes in each stage (" + S + " values): ");
        for (int i = 0; i < S; i++) {
            stageCount[i] = sc.nextInt();
            N += stageCount[i];
        }

        int[] stageStart = new int[S];
        int idx = 0;
        for (int i = 0; i < S; i++) {
            stageStart[i] = idx;
            idx += stageCount[i];
        }

        // Input: number of edges
```

```
System.out.print("Enter number of edges: ");
int M = sc.nextInt();

List<List<Edge>> adj = new ArrayList<>();
List<List<Integer>> revAdj = new ArrayList<>();
for (int i = 0; i < N; i++) {
    adj.add(new ArrayList<>());
    revAdj.add(new ArrayList<>());
}

System.out.println("Enter each edge as: u v cost");
for (int i = 0; i < M; i++) {
    int u = sc.nextInt();
    int v = sc.nextInt();
    double cost = sc.nextDouble();
    if (u < 0 || u >= N || v < 0 || v >= N) {
        System.out.println("Invalid edge node id");
        return;
    }
    adj.get(u).add(new Edge(v, cost));
    revAdj.get(v).add(u);
}

// Initialize best costs and next nodes
double[] bestCost = new double[(int) N];
int[] nextNode = new int[(int) N];
Arrays.fill(bestCost, INF);
Arrays.fill(nextNode, -1);

// Sink stage nodes have cost = 0
int lastStage = S - 1;
for (int k = 0; k < stageCount[lastStage]; k++) {
    int node = stageStart[lastStage] + k;
    bestCost[node] = 0.0;
    nextNode[node] = -1;
}

// Dynamic Programming (Backward)
for (int st = S - 2; st >= 0; st--) {
    for (int k = 0; k < stageCount[st]; k++) {
        int u = stageStart[st] + k;
        double best = INF;
        int bestv = -1;
        for (Edge e : adj.get(u)) {
```

```
            int v = e.to;
            double cost = e.curCost;
            if (bestCost[v] + cost < best) {
                best = bestCost[v] + cost;
                bestv = v;
            }
        }
        bestCost[u] = best;
        nextNode[u] = bestv;
    }
}

// Output results from Stage 0
System.out.println("\nBest costs from Stage-0 nodes:");
for (int k = 0; k < stageCount[0]; k++) {
    int u = stageStart[0] + k;
    if (bestCost[u] >= INF / 2)
        System.out.println("Node " + u + ": unreachable");
    else
        System.out.println("Node " + u + ": cost = " + bestCost[u]);
}

// Retrieve a path
System.out.print("\nEnter a source node id (in stage 0) to print path, or -1 to skip: ");
int src = sc.nextInt();
if (src >= 0 && src < N) {
    if (bestCost[src] >= INF / 2) {
        System.out.println("No route from " + src);
    } else {
        printPath(src, nextNode, adj);
        System.out.println("Total route cost: " + bestCost[src]);
    }
}

// Handle real-time updates
System.out.print("\nEnter number of live updates to edge costs (0 to finish): ");
int Q = sc.nextInt();
while (Q-- > 0) {
    System.out.print("Enter edge update (u v multiplier): ");
    int u = sc.nextInt();
    int v = sc.nextInt();
    double multiplier = sc.nextDouble();

    // Update edge weights
```

```
            for (Edge e : adj.get(u)) {
                if (e.to == v) {
                    e.curCost = e.baseCost * multiplier;
                }
            }

            // Recompute using incremental updates
            recomputeAfterUpdate(adj, revAdj, bestCost, nextNode, u);
        }

        // Final output after updates
        System.out.println("\nAfter updates, best costs from Stage-0 nodes:");
        for (int k = 0; k < stageCount[0]; k++) {
            int u = stageStart[0] + k;
            if (bestCost[u] >= INF / 2)
                System.out.println("Node " + u + ": unreachable");
            else
                System.out.println("Node " + u + ": cost = " + bestCost[u]);
        }

        // Reprint path after updates
        System.out.print("\nEnter a source node id (in stage 0) to print path, or -1 to skip: ");
        src = sc.nextInt();
        if (src >= 0 && src < N) {
            if (bestCost[src] >= INF / 2) {
                System.out.println("No route from " + src);
            } else {
                printPath(src, nextNode, adj);
                System.out.println("Total route cost: " + bestCost[src]);
            }
        }

        sc.close();
    }

    // Path reconstruction
    static void printPath(int src, int[] nextNode, List<List<Edge>> adj) {
        System.out.print("Path from " + src + ": ");
        int cur = src;
        while (cur != -1) {
            System.out.print(cur);
            int nxt = nextNode[cur];
            if (nxt != -1) {
                double edgeCost = 0;
```

```
            for (Edge e : adj.get(cur)) {
                if (e.to == nxt) {
                    edgeCost = e.curCost;
                    break;
                }
            }
            System.out.print(" -> ");
        } else break;
        cur = nxt;
    }
    System.out.println();
}

// Recompute after traffic update
static void recomputeAfterUpdate(List<List<Edge>> adj, List<List<Integer>> revAdj,
                                 double[] bestCost, int[] nextNode, int u) {
    Queue<Integer> queue = new LinkedList<>();
    double newCostU = recomputeNode(u, adj, bestCost, nextNode);
    if (Math.abs(newCostU - bestCost[u]) > 1e-9) {
        bestCost[u] = newCostU;
        queue.add(u);
    }

    while (!queue.isEmpty()) {
        int node = queue.poll();
        for (int pred : revAdj.get(node)) {
            double newCost = recomputeNode(pred, adj, bestCost, nextNode);
            if (Math.abs(newCost - bestCost[pred]) > 1e-9) {
                bestCost[pred] = newCost;
                queue.add(pred);
            }
        }
    }
}

static double recomputeNode(int node, List<List<Edge>> adj, double[] bestCost, int[] nextNode) {
    double best = INF;
    int bestv = -1;
    for (Edge e : adj.get(node)) {
        if (bestCost[e.to] >= INF / 2) continue;
        double cand = e.curCost + bestCost[e.to];
        if (cand < best) {
            best = cand;
            bestv = e.to;
```

```
            }
        }
        nextNode[node] = bestv;
        return best;
    }
}
```

Output

```
PS D:\DSA> cd "d:\DSA\" ; if ($?) { javac SwiftCargoLogistics.java } ; if ($?) { java
Enter number of stages: 4
Enter number of nodes in each stage (4 values): 2 2 2 1
Enter number of edges: 6
Enter each edge as: u v cost
0 2 3.0
0 3 2.5
1 2 2.0
1 3 4.0
2 4 1.5
3 4 2.0

Best costs from Stage-0 nodes:
Node 0: cost = 4.000000
Node 1: cost = 3.500000

Enter a source node id (in stage 0) to print path, or -1 to skip: 0
Path from 0 : 0 -> 3 -> 4
Total route cost: 4.0

Enter number of live updates to edge costs (0 to finish): 1
Enter edge update (u v multiplier): 3 4 0.5

After updates, best costs from Stage-0 nodes:
Node 0: cost = 3.000000
Node 1: cost = 3.500000

Enter a source node id (in stage 0) to print path, or -1 to skip: 0
Path from 0 : 0 -> 3 -> 4
Total route cost: 3.0

PS D:\DSA>
```

5. Conclusion

- Multistage graphs model stepwise logistics efficiently.
- Dynamic Programming finds the optimal route by backward computation using optimal substructure.
- Approach is scalable, flexible, and adapts to real-time constraints.
- Ensures minimum delivery cost/time, improving SwiftCargo's operational efficiency.