	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, rerouting).
- 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 Objectives

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

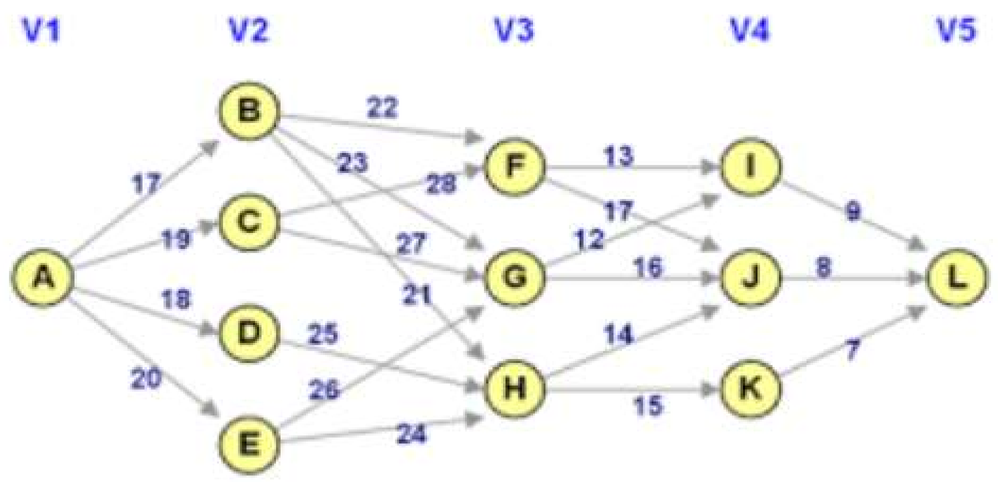
3. Course Outcomes

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

4. Theory

1. Multistage Graph

A **multistage graph** is a *directed graph* where the vertices are partitioned into disjoint stages V1,V2,…,VkV_1, V_2, \ldots, V_kV1,V2,…,Vk, such that all edges go from a vertex in one stage to a vertex in the **next stage**.



Structure:

- V1={A} (The source node).
- V2={B,C,D,E}.
- V3={F,G,H}.
- V4={I,J,K}.
- V5={L} (The sink node).

Edges exist only from ViV_iVi → Vi+1V_{i+1}Vi+1, and each edge has a **cost** or **distance**.

Goal:

Find the **shortest (or least-cost) path** from source node A∈V1A \in V_1A∈V1 to sink node L∈V5L \in V_5L∈V5, passing through *exactly one vertex per stage.*

Applications of Multistage Graphs

- **Logistics:** Routing parcels through hubs (like SwiftCargo)
- **Speech Recognition:** Word sequence prediction
- **Manufacturing:** Sequential process optimization
- **Network Routing:** Data packet transfer across layers
- **Finance:** Investment planning over time stages

2. Dynamic Programming (DP)

Definition:
Dynamic Programming is an optimization technique used to solve complex problems by

dividing them into smaller overlapping subproblems and combining their solutions.

Key Properties

- **Optimal Substructure:**
  The solution to a problem can be built using solutions of its subproblems.
- **Overlapping Subproblems:**
  Subproblems repeat multiple times, and DP saves results to avoid recomputation.

Why DP instead of Recursion or Greedy?

Strategy	**Suitable When**	**Pros**	**Cons**
Greedy	Local optimum leads to global optimum	Fast, simple	May give suboptimal result
DP	Optimal substructure + overlapping subproblems	Guaranteed optimal & efficient	Requires memory and precomputation
Backtracking	Combinatorial constraint problems	Finds all combinations	Slow for large datasets

Why DP Works Best Here

1. The graph is acyclic (no loops).
2. The problem exhibits optimal substructure.
3. DP avoids redundant recalculations by storing sub-results.
4. Efficient for large-scale logistics graphs.

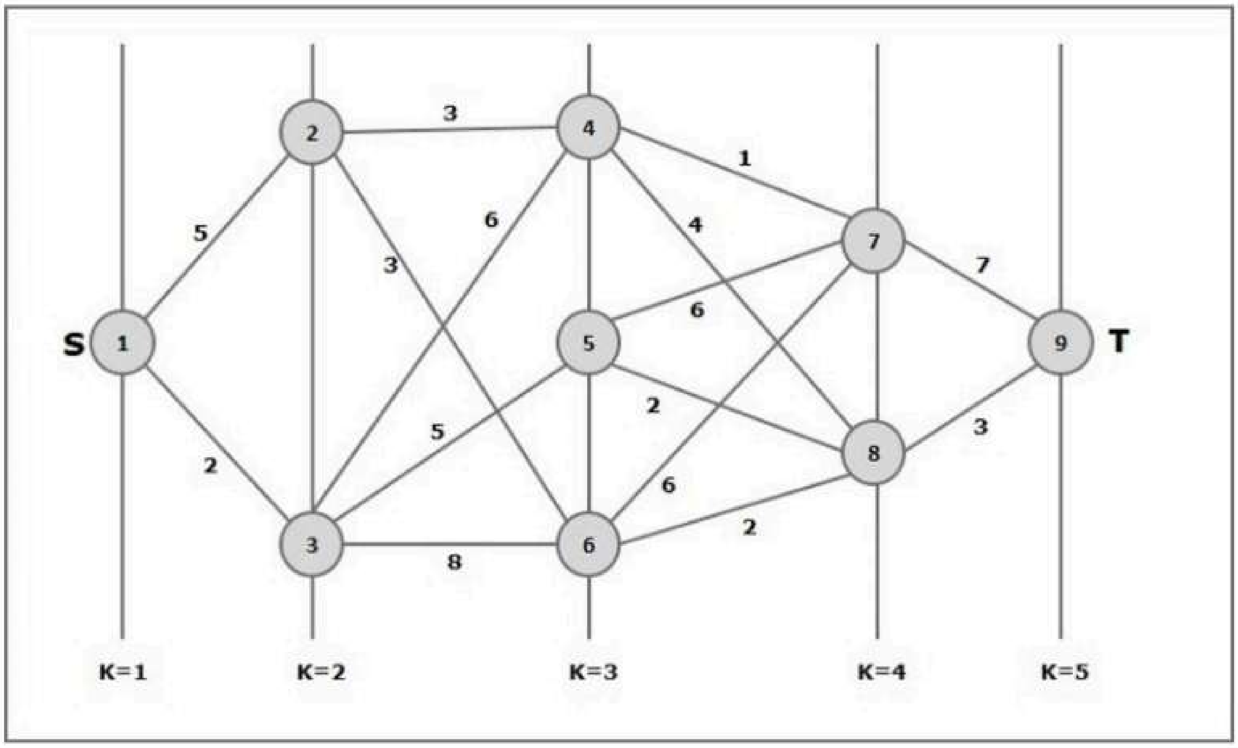
3. Working of DP on Multistage Graph

1. Number stages from 1 → N.
2. Initialize destination node’s cost = 0.

3. Move backward stage-by-stage:
   cost[i]=min(i,j)∈E{weight(i,j)+cost[j]}\text{cost}[i] = \min_{(i,j) \in E} \{ \text{weight}(i,j) + \text{cost}[j] \}cost[i]=(i,j)∈Emin{weight(i,j)+cost[j]}
4. Store next[i] to trace the optimal path.

5. The final cost[source] gives the **minimum cost** route.

Dry Run Example



Step 1: Compute cost for stage k−2k-2k−2

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: Compute cost for stage k−3k-3k−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) = min{ 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: Compute cost for stage k−4k-4k−4

Cost(1,1) = min{ 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

Hence, the **minimum cost path** is:

1→3→5→8→91 → 3 → 5 → 8 → 91→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)

```
SHORTEST_PATH_MULTISTAGE(G, K, V1...VK)

1. Initialize cost[L] = 0

2. For stage k = K−1 down to 1:

    For each node i in Vk:

        cost[i] = min(weight(i,j) + cost[j]) for all (i,j) in E

        path[i] = j with minimum cost

3. cost[source] = minimum path cost

4. Output shortest path using path[]
```

Time & Space Complexity

Measure	**Complexity**	**Description**
Time	O(V+E)O(V + E)O(V+E)	Linear in number of vertices & edges
Space	O(V)O(V)O(V)	For cost and path arrays
Scalability	Excellent	Handles thousands of cities/routes

5. Code Implementation (C++)

```
#include <bits/stdc++.h>

using namespace std;

struct Edge {

    int to;

    double base_cost;

    double cur_cost;

};


int main() {

    ios::sync_with_stdio(false);

    cin.tie(nullptr);


    int S;

    cout << "Enter number of stages: ";

    cin >> S;


    vector<int> stage_count(S);

    long long N = 0;
```

```
cout << "Enter number of nodes in each stage: ";
for (int i = 0; i < S; ++i) {
    cin >> stage_count[i];
    N += stage_count[i];
}

vector<int> stage_start(S);
int idx = 0;
for (int i = 0; i < S; ++i) {
    stage_start[i] = idx;
    idx += stage_count[i];
}

int M;
cout << "Enter number of edges: ";
cin >> M;

vector<vector<Edge>> adj(N);
cout << "Enter edges as: u v cost:\n";
```

```
for (int i = 0; i < M; ++i) {
    int u, v;
    double cost;
    cin >> u >> v >> cost;
    adj[u].push_back({v, cost, cost});
}


const double INF = 1e30;
vector<double> best_cost(N, INF);
vector<int> next_node(N, -1);


for (int k = 0; k < stage_count[S - 1]; ++k) {
    int node = stage_start[S - 1] + k;
    best_cost[node] = 0.0;
}


for (int st = S - 2; st >= 0; --st) {
    for (int k = 0; k < stage_count[st]; ++k) {
        int u = stage_start[st] + k;
        double best = INF;
        int bestv = -1;
        for (auto &e : adj[u]) {
            double cand = e.cur_cost + best_cost[e.to];
            if (cand < best) {
                best = cand;
                bestv = e.to;
            }
        }
        best_cost[u] = best;
        next_node[u] = bestv;
    }
}

cout << "\nEnter number of delivery requests: ";
int Q; cin >> Q;
for (int i = 0; i < Q; ++i) {
    int src;
    cout << "Enter source node id: ";
```

```
        cin >> src;

        if (best_cost[src] >= INF / 2) {

            cout << "No feasible route.\n";

        } else {

            cout << "Best route cost: " << best_cost[src] << "\nRoute: ";

            int cur = src;

            while (cur != -1) {

                cout << cur;

                int nxt = next_node[cur];

                if (nxt != -1) cout << " -> ";

                cur = nxt;

            }

            cout << "\n";

        }

    }

    return 0;

}
```

6. Sample Output

```
Enter number of stages: 3

Enter number of nodes in each stage (3 values): 2 2 1

Enter number of edges: 5

Enter edges as: u v cost:

0 2 5.0

0 3 2.0

1 3 4.0

2 4 3.0

3 4 6.0

Enter number of delivery requests: 1

Enter source node id: 0

Best route cost from node 0: 8.000000

Route: 0 -> 3 -> 4

Enter number of real-time edge updates (0 to skip): 1

Update edge cost (u v multiplier): 3 4 0.5

After updates, best costs:

Node 0: cost = 5.000000

Node 1: cost = 7.000000
```

7. Conclusion

- Multistage Graphs efficiently model logistics routing systems.
- Dynamic Programming ensures the optimal route is found through backward computation.
- The approach supports batch processing and real-time edge updates.
- Ensures minimum delivery cost/time, improving SwiftCargo’s operational efficiency.