Ramdeobaba University, Nagpur
Department of Computer Science and Engineering

Session: 2025-26

Subject: Design and Analysis of Algorithms (DAA) Lab Project

III Semester

LAB PROJECT REPORT

SUBMITTED BY-
1)Soumya Wasule(A1-38)
2)Sharvan Kotharu(A1-43)
3)Sujal Tembhare(A1-49)

Github repository

https://github.com/SoumyaWasule/Vector_path.git

TITLE : LOGISTICS ROUTE MANAGER

OBJECTIVES

- ❖ Minimize Total Delivery Cost and Time
- ❖ Optimize Route Planning
- ❖ Maximize Vehicle and Resource Utilization
- ❖ Automate and Visualize Decision Making
- ❖ Improve Scalability and Reliability
- ❖ Enhance User Experience and Communication:
- ❖ Bridge Theory and Practice

INTRODUCTION

In today's fast-paced e-commerce environment, companies need efficient ways to deliver goods to their customers quickly and at a low cost. The process of planning, organizing, and carrying out deliveries across multiple warehouses and customer locations is called supply chain logistics. Many challenges can arise during deliveries, such as delays, high costs, or poor use of delivery vehicles.

Our project aims to solve these problems by using advanced computer algorithms. We have created a web application that helps plan the best delivery routes, decide which packages should go on each trip, and make sure every delivery vehicle is working at its full capacity. The system uses well-known algorithms like Multi-Stage Graph, Traveling Salesman Problem (TSP), and Fractional Knapsack. The app also provides an interactive and animated visual interface so users or firms can clearly see how deliveries are optimized. This project will help e-commerce companies save money, deliver faster, and make their supply chain much more effective.

ALGORITHMS/TECHNIQUES USED

1) **TravellingSalesmanProblem Algorithm***(shortest and most efficient delivery route for a vehicle that needs to visit multiple delivery points or customer locations and return to the starting point)*

2) **Multistaged graph Algorithm***(optimize the delivery routes across a series of mandatory warehouses or distribution centers arranged in sequential stages)*

3) **Fractional Knapsack***(optimize the loading of delivery vehicles by selecting packages with the highest value-to-weight ratio to maximize overall profit within the vehicle’s capacity constraints)*

Algorithms/Pseudocode :

<u>TSP</u>

```
function nearestNeighborTSP(distanceMatrix):

    n = number of cities

    visited = array of boolean initialized to False

    route = list initialized with starting city (e.g., 0)

    visited[0] = True

    currentCity = 0

    totalDistance = 0

        for i in range(1 to n-1):

        nearestCity = None

        shortestDistance = infinity

        for city in range(n):

            if not visited[city] and
distanceMatrix[currentCity][city] < shortestDistance:

                nearestCity = city

     shortestDistance = distanceMatrix[currentCity][city]

     route.append(nearestCity)
```

```
        visited[nearestCity] = True
        totalDistance += shortestDistance
        currentCity = nearestCity
totalDistance += distanceMatrix[currentCity][0]  # Return to
start
    route.append(0)  # Complete cycle
    return route, totalDistance
```

Time Complexity : $O(n^2)$

This is because for each city, the algorithm checks distances to all unvisited cities to find the nearest one, which takes $O(n)$ time per city. Since this process is repeated for all n cities, the total is $O(n^2)$, This makes it much faster than exact algorithms, although it does not always guarantee the optimal solution.

MSG

```
function multiStageGraph(graph, stages, n):

    // n = total number of nodes

    // stages = list of stages with nodes

    // graph[i][j] = cost from node i to node j (infinity if no
edge)


    cost = array of size n initialized to infinity

    path = array of size n to store next node in optimal path


    // Start from the last stage (destination node)

    destination = last node in final stage

    cost[destination] = 0


    // Process stages from second-last to first (backward)

    for stage from (number of stages - 2) down to 0:

        for each node u in current stage:
```

```
            minCost = infinity

            nextNode = None

            for each node v in next stage:

                if graph[u][v] exists and graph[u][v] + cost[v] <
minCost:

                    minCost = graph[u][v] + cost[v]

                    nextNode = v

            cost[u] = minCost

            path[u] = nextNode


    // Trace the optimal path from source to destination

    optimalPath = []

    currentNode = source (first node in first stage)

    optimalPath.append(currentNode)


    while currentNode != destination:

        currentNode = path[currentNode]

        optimalPath.append(currentNode)
```

```
return optimalPath, cost[source]
```

Time Complexity : O(n×e)

The time complexity of the Multi-Stage Graph (MSG) algorithm using dynamic programming is typically $O(n \times e)$, where n is the number of nodes and e is the number of edges between stages. The algorithm processes each node in the graph and computes the minimum cost by examining all outgoing edges from that node to the next stage. This ensures an efficient calculation of the optimal path by building solutions incrementally from the destination back to the source, avoiding redundant calculations and making it suitable for stage-wise structured problems like supply chain logistics

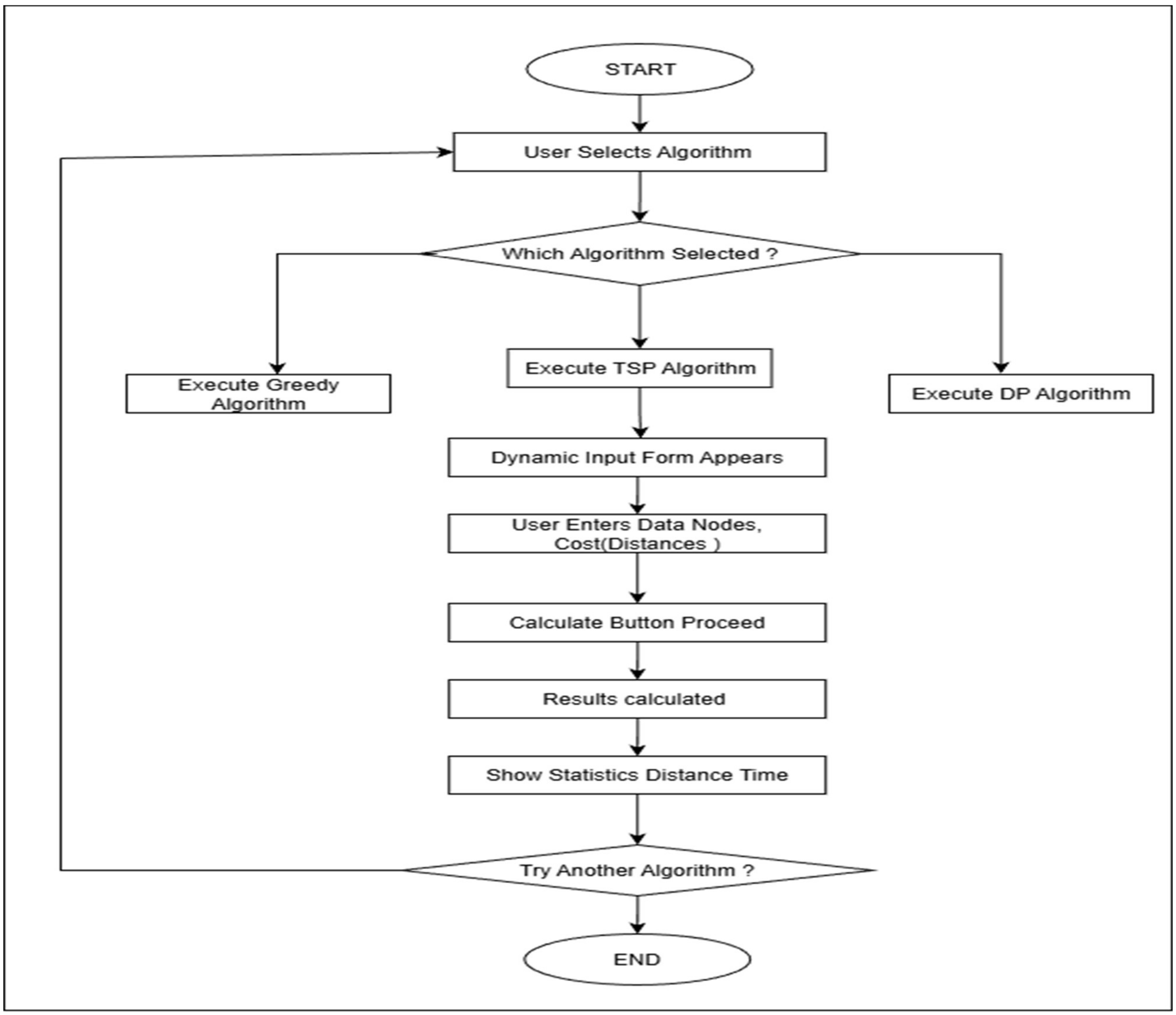
Fractional Knapsack

```
function fractionalKnapsack(values, weights, capacity):

    n = number of items

    items = array of (value, weight, value/weight)

    for i in 0 to n-1:

        items[i] = (values[i], weights[i], values[i]/weights[i])

    sort items by value/weight ratio in descending order

      totalValue = 0

    remainingCapacity = capacity

    for item in items:

        if item.weight <= remainingCapacity:

            totalValue += item.value

            remainingCapacity -= item.weight

        else:

            fraction = remainingCapacity / item.weight

            totalValue += item.value * fraction

            break

    return totalValue
```

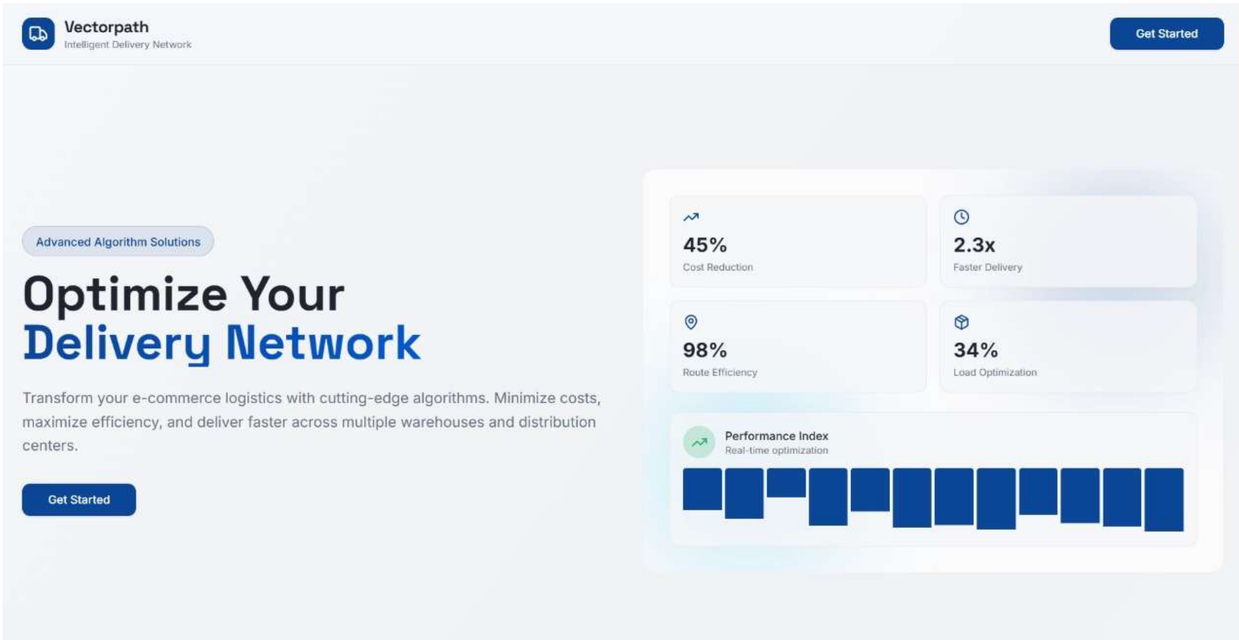
Time Complexity

The Fractional Knapsack algorithm has a time complexity of O(n log n). This is because sorting items by their value-to-weight ratio takes O(n log n), and the subsequent single pass to select items adds O(n). Sorting dominates, so the overall complexity is O(n logn).

LOGISTICS SUPPLY FLOW

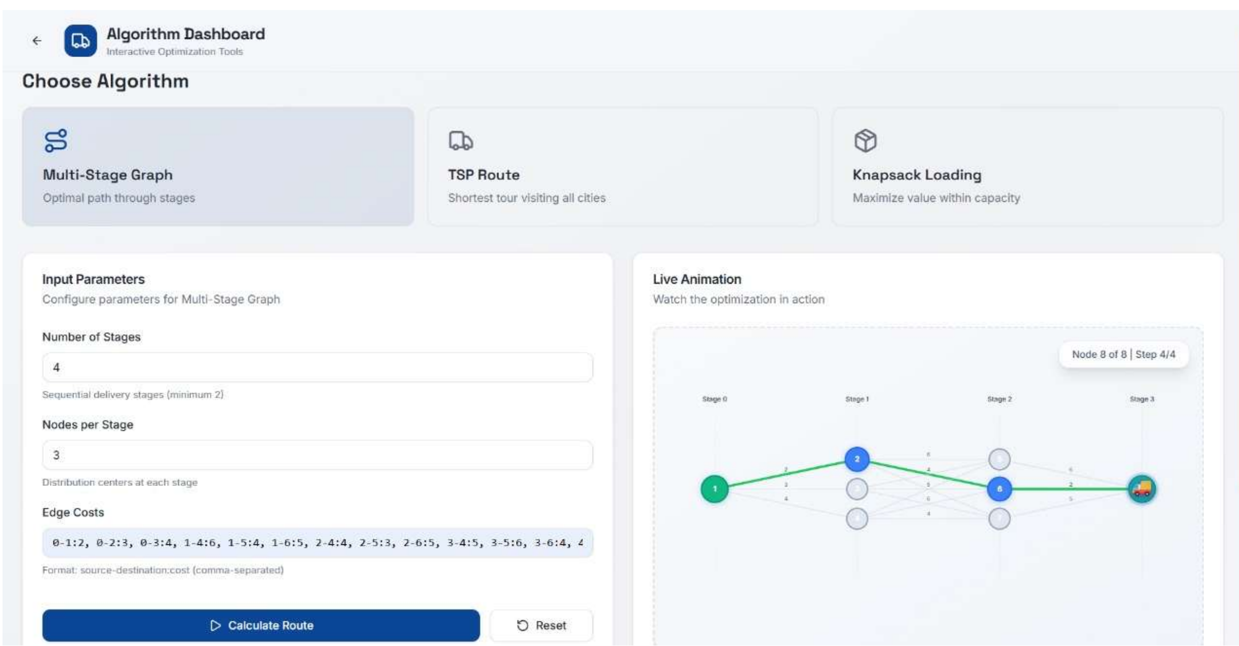


RESULTS



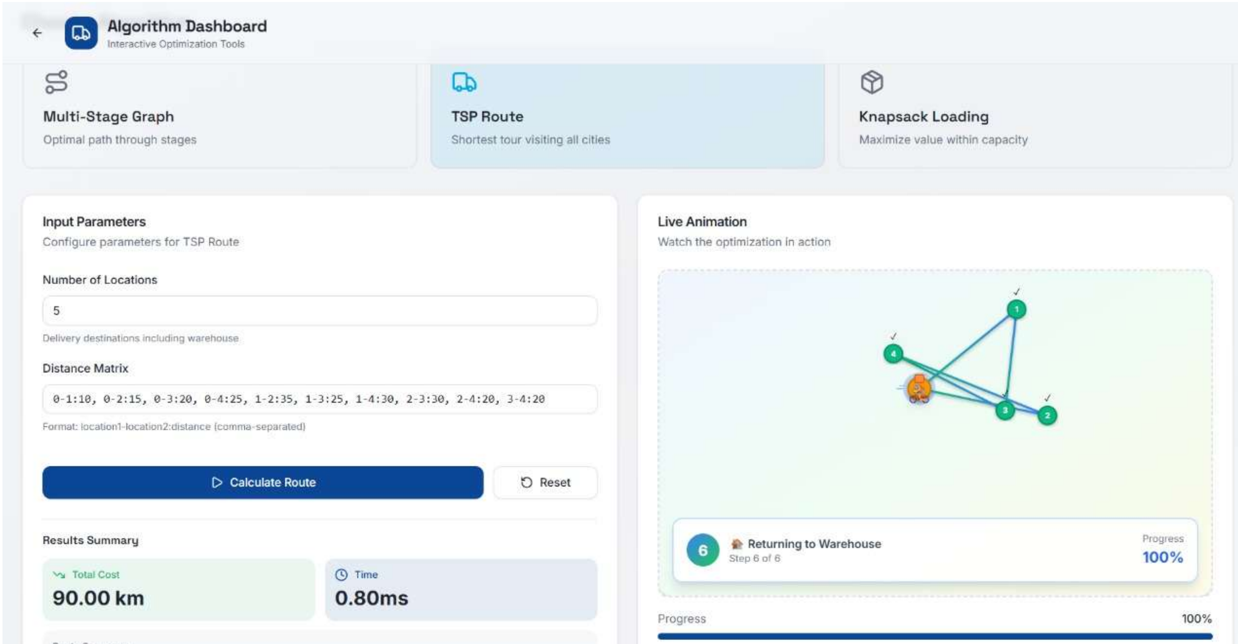
Home Page UI

1)MSG:Finds optimal path connecting warehouses

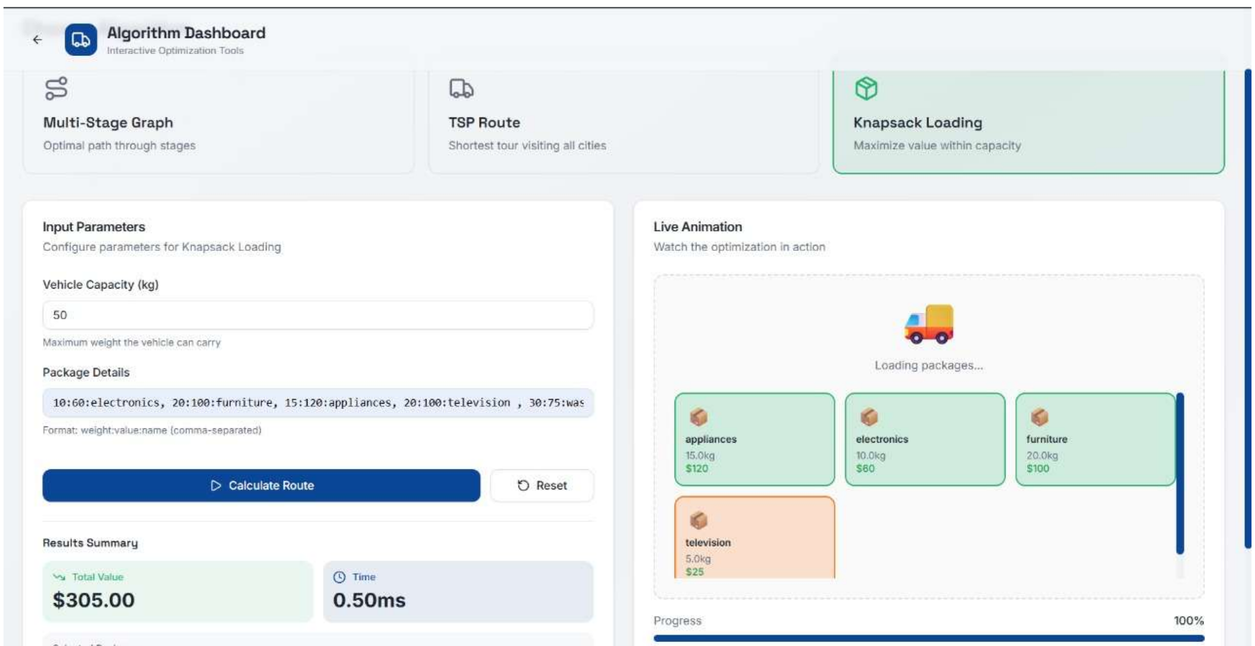


Users/Firms have the option to select a particular algorithm depending on their use case

2)TSP:-Finds the shortest path connecting locations of different customers



3)FractionalKnapsack:Finds the best possible combination of goods for efficient packaging of goods



CONCLUSION & FUTURE SCOPE

This project successfully demonstrates how advanced algorithms like Multi-Stage Graph (Dynamic Programming), Traveling Salesman Problem (TSP), and Fractional Knapsack can optimize a complex e-commerce supply chain. By efficiently planning delivery routes, scheduling vehicle trips, and maximizing package loading, the system minimizes overall delivery costs and time. The interactive application with animations helps visualize the logistics process clearly, making it easier for users and stakeholders to understand and apply these optimizations in real-world scenarios.

Future Scope:

- Integrate real-time traffic and weather data to dynamically adjust routes for better accuracy and responsiveness.
- Expand the model to support multiple vehicles and simultaneous deliveries using advanced vehicle routing problem (VRP) algorithms.
- Incorporate machine learning-based demand forecasting to proactively plan logistics resources.
- Enhance UI with map APIs (Google Maps, Mapbox) for precise geolocation and delivery tracking.
- Develop mobile apps for on-the-go monitoring and route adjustments by field drivers.
- Explore IoT integration to automatically track vehicle status, deliveries, and optimize loads in real time.