Mego Logistics Project Documentation

by Sushma Suresh

Efficient logistics and route planning are crucial for reducing transportation costs and ensuring timely deliveries. We try to implement **Simulated Annealing**, a probabilistic technique for approximating global optimization.

Steps and Implementation

1. Define Cities and Coordinates

- A dictionary stores city names with their respective latitude and longitude values.
- The dataset includes major Indian state capitals and union territories.

```
# Step 1: Define the coordinates (latitude, longitude) of each capital/UT
cities = {

    "Panaji": (15.4909, 73.8278),
    "Daman": (20.3974, 72.8311),
    "Mumbai Dock": (18.5204, 73.8567),
    "Kavaratti": (10.5625, 72.6350),
    "Kolkata": (22.5726, 88.3639),
    "Chennai": (13.0827, 80.2707),
    "Bengaluru": (12.9716, 77.5946),
    "Hyderabad": (17.3850, 78.4867),
    "Gandhinagar" :(23.2156, 72.6369),
    "Jaipur": (26.9124, 75.7873),
    "Lucknow": (26.8467, 80.9462),
    "Chandigarh": (30.7333, 76.7794),
    "Bhopal": (23.2599, 77.4126),
    "Patna": (25.5941, 85.1376),
    "Ranchi": (23.3441, 85.3096),
    "Thiruvananthapuram": (8.5241, 76.9366),
    "Imphal": (24.8
    "Shillong": (25.5788, 91.8933),
```


2. Distance Calculation using Haversine Formula

- The Haversine formula computes the great-circle distance between two points on Earth.
- This ensures accurate distance measurements between any two cities.

3. Generate Initial Route

- A random initial route is created, ensuring that:

The tour starts and ends in Mumbai Dock.

The rest of the cities are shuffled randomly.

```python
# Step 2: Calculate the distance between two cities using Haversine Formula
def haversine(coord1, coord2):
    lat1, lon1 = coord1
    lat2, lon2 = coord2
    R = 6371  # Radius of Earth in km

    # Convert degrees to radians
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

    # Haversine formula
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    return R * c
```

4. Calculate Route Distance

- The total distance of a given route is computed by summing up pairwise distances between consecutive cities.

```python
# Step 3: Generate a random route (excluding Mumbai which is fixed at the start and end)
def create_initial_route(cities_list):
    cities_list = [city for city in cities_list if city != "Mumbai Dock"]  # Exclude Mumbai
    route = cities_list[:]
    random.shuffle(route)
    route = ["Mumbai Dock"] + route + ["Mumbai Dock"]  # Start and end at Mumbai
    return route
```


5. Simulated Annealing Optimization

i. Cooling Schedule: Temperature starts high and gradually reduces based on a cooling rate.

ii. Neighboring Solution Generation: Two random cities (excluding Mumbai Dock) are swapped.

iii. Acceptance Criteria:

- If the new route is shorter, it is accepted.
- If the new route is longer, it is accepted with some probability (to escape local optima).

i. Iterations: The algorithm runs for 10,000 iterations to refine the solution.

```python
# Step 3: Simulated Annealing Algorithm
def simulated_annealing(cities_list, initial_temperature, cooling_rate, iterations):
    current_route = create_initial_route(cities_list)
    current_distance = calculate_total_distance(current_route)

    best_route = list(current_route)
    best_distance = current_distance
    initial_distance=current_distance
    temperature = initial_temperature

    for i in range(iterations):
        # Create a neighboring solution by swapping two cities (excluding Mumbai)
        new_route = list(current_route)
        idx1, idx2 = random.sample(range(1, len(cities_list)), 2)  # Only swap cities excluding Mumbai
        new_route[idx1], new_route[idx2] = new_route[idx2], new_route[idx1]

        new_distance = calculate_total_distance(new_route)

        # Accept the new solution based on the Metropolis criterion
        if new_distance < current_distance or random.uniform(0, 1) < math.exp((current_distance - new_distance) / temperature):
            current_route = list(new_route)
            current_distance = new_distance

            # Update the best solution found so far
            if current_distance < best_distance:
                best_route = list(current_route)
                best_distance = current_distance
```

6. Execution & Results

i. The algorithm outputs:

- The best route found.
- The total optimized distance in km.
- The initial (random) route distance for comparison.

🚀 Best Route:

Mumbai Dock → Gandhinagar → Daman → Panaji → Kavaratti → Thiruvananthapuram → Puducherry → Chennai → Bengaluru → Hyderabad → Bhopal → Raipur → Patna → Gangtok → Dispur → Shillong → Itanagar → Kohima → Imphal → Aizawl → Agartala → Kolkata → Bhubaneswar → Ranchi → Lucknow → New Delhi → Dehradun → Chandigarh → Shimla → Leh → Srinagar → Jammu → Jaipur → Mumbai Dock

📏 **Total Optimized Distance: 12,069.80 km**

🎯 **Initial Random Route Distance: 48,267.41 km**

✨ **Efficiency Gain: 🚀 ~75% distance reduction!**

🚀 **Best Route:**

Mumbai Dock → Gandhinagar → Daman → Panaji → Kavaratti → Thiruvananthapuram → Puducherry → Chennai → Bengaluru → Hyderabad → Bhopal → Raipur → Patna → Gangtok → Dispur → Shillong → Itanagar → Kohima → Imphal → Aizawl → Agartala → Kolkata → Bhubaneswar → Ranchi → Lucknow → New Delhi → Dehradun → Chandigarh → Shimla → Leh → Srinagar → Jammu → Jaipur → **Mumbai Dock**

📏 **Total Optimized Distance: 12,069.80 km**

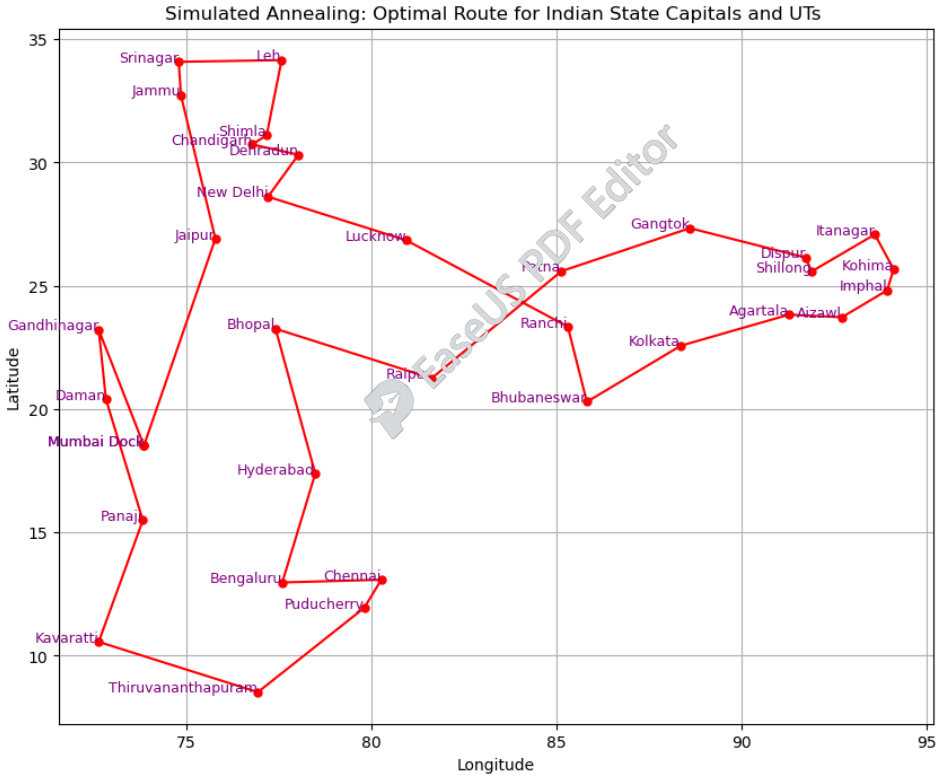
📍 **Initial Random Route Distance: 48,267.41 km**

✨ **Efficiency Gain: 🚀 ~75% distance reduction!**

-

7. Visualization

- Uses matplotlib to plot the optimized route on a coordinate map.
- Each city is labeled on the plot.



-

Shipment Allocation:

Group items by unloading sequence: If each stop in the shipment corresponds to a specific group of items, you can organize them into distinct categories based on the unloading order.

This code loads shipment route data, sorts the shipment stops in reverse order (from last stop to first), and saves the optimized loading plan in a new CSV file.

[9]:
```python
import pandas as pd

# Load shipment data
df = pd.read_csv("shipment_routes.csv")

# Sort in reverse order (last stop first)
df_sorted = df.sort_values(by="Stop_Order", ascending=False)

# Save the optimized loading plan
df_sorted.to_csv("optimized_loading_plan.csv", index=False)

print("Loading plan created successfully! 🚛")
```

```
Loading plan created successfully! 🚛
```

Fuel Efficiency:

The Volvo FH16 is a popular heavy-duty truck used for long-haul logistics and transportation. Here's more information about the Volvo FH16:

Fuel Tank Capacity:

The Volvo FH16 typically has a fuel tank capacity of around 500 to 1,000 liters, depending on the configuration and the number of fuel tanks. Most common configurations include 2 fuel tanks, with each tank typically holding between 250 to 500 liters, allowing for a total capacity of up to 1,000 liters.

Fuel Efficiency:

The fuel efficiency of a Volvo FH16 varies depending on the weight of the cargo, road conditions, and driving style, but on average, these trucks can achieve 6 to 8 km per liter of fuel.

3]:
```python
cost_of_fuel_in_mumbai = 103.50  # Cost per liter in Rupees
fuel_efficiency = 7    # Average fuel efficiency in km per liter

# Assuming best_distance is the total distance for the route
total_liters = best_distance / fuel_efficiency

print(f'Total number of liters the vehicle may require: {total_liters} L')

dist = []  # To store the distances between cities
refuelin = 1000  # Threshold for refueling
total_distance = 0  # To accumulate the total distance
tot_dist=0
for i in range(len(best_route)-1 ):
    tot_dist=0
    segment_distance = haversine(cities[best_route[i]], cities[best_route[i+1]])
    total_distance += segment_distance
    dist.append(segment_distance)
    tot_dist+= segment_distance
    # Check if the accumulated distance exceeds the refuel threshold
    if tot_dist > 1000:
        print(f"Refuel at city {best_route[i+1]} after traveling {total_distance} km.")
        tot_dist = 0  # Reset distance for the next segment

print(f"Total distances between cities: {total_distance} ")

# Calculate the total fuel cost for the route
fuel_cost = total_liters * cost_of_fuel_in_mumbai
print(f'The fuel cost comes up to be: {fuel_cost} Rupees')
```

```
Total number of liters the vehicle may require: 1724.2566804078194 L
Total distances between cities: 12069.796762854736
The fuel cost comes up to be: 178460.56642220932 Rupees
```