**Comprehensive Report on Solving the Travelling Salesman Problem for Product Distribution in Kenya**

**1. Introduction**

This report documents a full technical and research-driven implementation of a Travelling Salesman Problem (TSP) solver to determine the most optimal route for distributing a market product across six counties in Kenya. The solution is coded in Python and leverages open-source APIs and libraries for geographical data, distance calculation, and visualization. The project demonstrates how classical search algorithms and modern APIs can address real-world logistics.

**2. Problem Statement**

The objective is to determine the most efficient route for distributing goods that:

* Starts and ends in Nairobi
* Visits each of the other counties (Meru, Nyeri, Nandi, Kericho, Nakuru) only once
* Minimizes total travel distance

Requirements:

* Implemented in Python
* Executable via command line (CMD)
* Uses a virtual environment for package isolation

**3. Project Goals**

* ✅ Develop a working solution for TSP using a brute-force DFS algorithm
* ✅ Accurately fetch real-world road distances using the OpenRouteService API
* ✅ Convert town names to geocoordinates
* ✅ Visualize the route using graph representations
* ✅ Package the scripts for command-line execution inside a virtual environment

**4. Methodology and Research**

**4.1 Overview of TSP**

The Travelling Salesman Problem is a classic NP-hard problem in computer science. It requires finding the shortest route that visits each location once and returns to the starting point. With only six towns, we can compute all permutations feasibly.

**4.2 Algorithms Considered**

**a) Depth First Search (DFS) – Brute Force**

We implemented a custom DFS that recursively explores all possible routes starting from Nairobi.

* 🔍 Maintains visited set, path list, and cumulative distance
* 📈 Returns to Nairobi when a complete path is formed
* 🏆 Updates the shortest path if a better one is found

✅ Advantages:

* Exact optimal solution
* Simple and deterministic

⚠️ Limitations:

* O(n!) time complexity
* Not scalable beyond small datasets

**b) Greedy Algorithm (Not Implemented)**

Starts from Nairobi and chooses the nearest unvisited town.

👍 Pros:

* Very fast, low memory use

👎 Cons:

* Suboptimal routes
* No backtracking

**c) Genetic Algorithms (Reviewed)**

Simulates natural selection to evolve route quality over generations.

👍 Pros:

* Scales well
* Often approaches optimal solutions

👎 Cons:

* Non-deterministic
* Needs parameter tuning

**d) Google OR-Tools (Reviewed)**

High-performance optimization toolkit by Google.

👍 Pros:

* Built-in TSP solver
* Efficient and scalable

👎 Cons:

* Complex to set up
* Requires detailed modeling

🧠 Final Decision: DFS chosen for guaranteed correctness within a small dataset.

**5. System Architecture**

**5.1 Scripts Developed**

* ai2.py: Implements DFS-based TSP route computation
* ai3.py: Visualizes computed routes using matplotlib + networkx

**5.2 API Integration**

* Uses OpenRouteService for geocoding and distance matrix
* Converts town names to geographic coordinates
* Retrieves real-world road distances

**5.3 Environment Setup**

* Python 3.11
* Virtual environment created for dependency management

**6. Step-by-Step Implementation Guide**

**Step 1: Create the Project Folder**

We used file explorer to create this.

**Step 2: Add Scripts**

Place ai2.py and ai3.py inside the folder.You can open terminal from this folder by right clicking anywhere in the space and select ‘open in terminal’.

**Step 3: Create Virtual Environment**

python -m venv venv

**Step 4: Activate the Virtual Environment**

Windows PowerShell:

.\venv\Scripts\Activate.ps1

Command Prompt:

venv\Scripts\activate.bat

**Step 5: Install Required Packages**

pip install openrouteservice networkx matplotlib

**Step 6: Run the Scripts**

Run the TSP solver:

python ai2.py

Run the visualizer:

python ai3.py

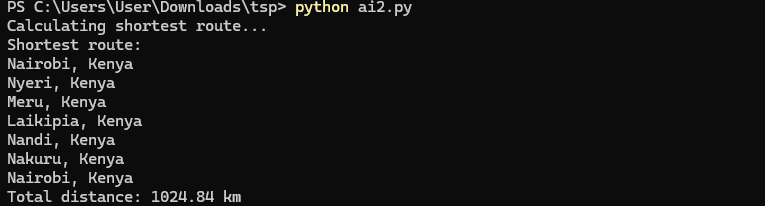
**7. Output and Interpretation**

🟢 ai2.py Output:

* Shows optimal route including Nairobi
* Displays total travel distance in kilometers

OpenRouteService used here.

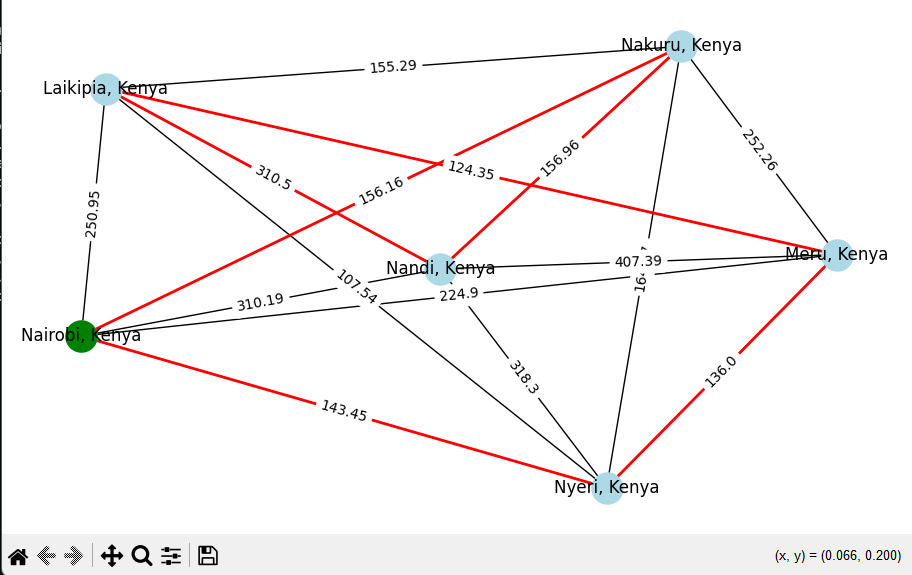
The output should look like this if done correctly;



📊 ai3.py Output:

* Graph with towns as nodes
* Red edges show optimized traversal sequence

The output will both be in terminal same as as the first script but with a visual representation as shown below.Here matpotlib and network have beem used.



**8. Validation and Testing**

* Verified coordinates using map services
* Cross-checked distance matrix values
* Ran tests on smaller sets to validate DFS logic
* Compared output to expected results manually

**9. Challenges and Resolutions**

* ❗ pip timeout errors resolved by using prefer-binary wheels
* ❗ SSL issues fixed by switching to python.org installer
* ❗ Long install times for matplotlib mitigated by pinning version to 3.5.3

**10. Visualization Techniques**

* Used networkx to construct the route graph
* Applied spring\_layout for clean node positioning
* Enabled interactive route animation in matplotlib
* Differentiated visited nodes using distinct colors

**11. Conclusion**

This project presents a functional TSP solution tailored to Kenyan town logistics using accurate road distance data. While brute-force DFS has scalability limits, it ensures correctness and serves as an educational foundation. The visual output enhances comprehension and utility for logistics demonstrations.

**12. Future Enhancements**

* 🔁 Replace DFS with Google OR-Tools for large-scale routing
* 🖥 Add GUI or web interface for usability
* 🕓 Integrate live traffic/time constraints
* ✏️ Enable dynamic town list editing from user input

**13. References**

* TSP\_Methods\_Explanation.docx: Algorithm comparison
* APPROACH.docx: Project plan and structure
* TSP.pdf: Problem formalization and mathematical basis
* ai2.py and ai3.py as the python script files.