**MODE FAIR HOME ASSESSMENT**

**DOCUMENTATION**

**Prepared by**

**Pay Eong Waon (Stephanie)**

## 

## **Introduction**

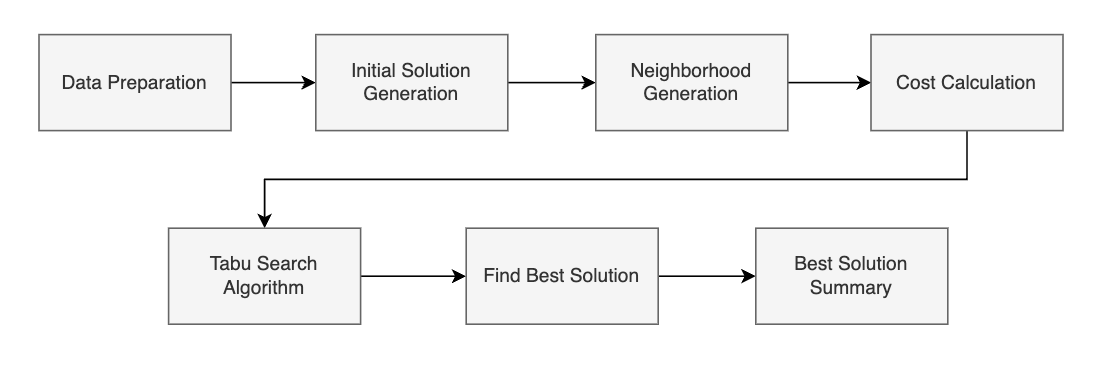
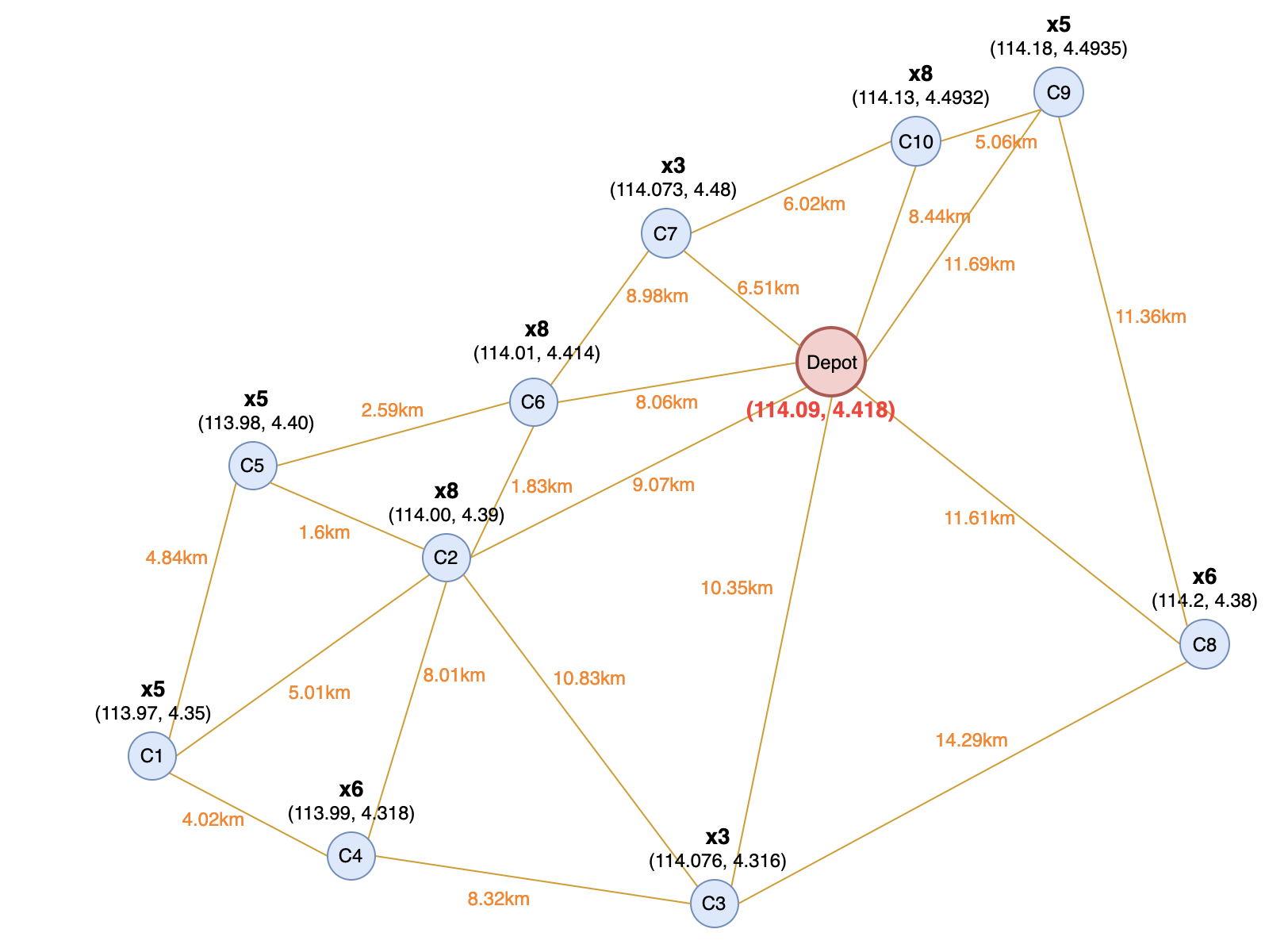
This project aims to optimize the routing of a fleet of vehicles to efficiently deliver goods to various customer locations while minimizing costs and following vehicle capacity constraints. This documentation outlines the approach, which involves data preparation, data preparation, solution generation, and optimization using the **Tabu Search algorithm**. Figure 1 shows a general illustration of the routes between customers and the depot for better understanding. Figure 2 displays a high level overview of flowchart for developing the algorithm.

Figure 1: General Illustration

Figure 2: High level overview of flowchart

## **Data Preparation**

### **Customer Data & Vehicle Data**

This assessment utilizes two main datasets: customer locations with their demands, vehicle types with capacities and costs. The depot coordinates are also defined.

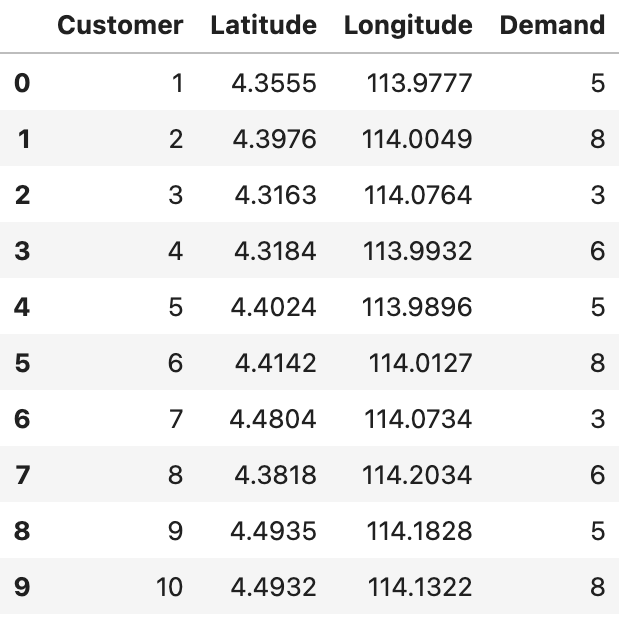


Figure 3: Customer Data

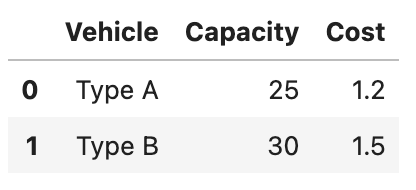


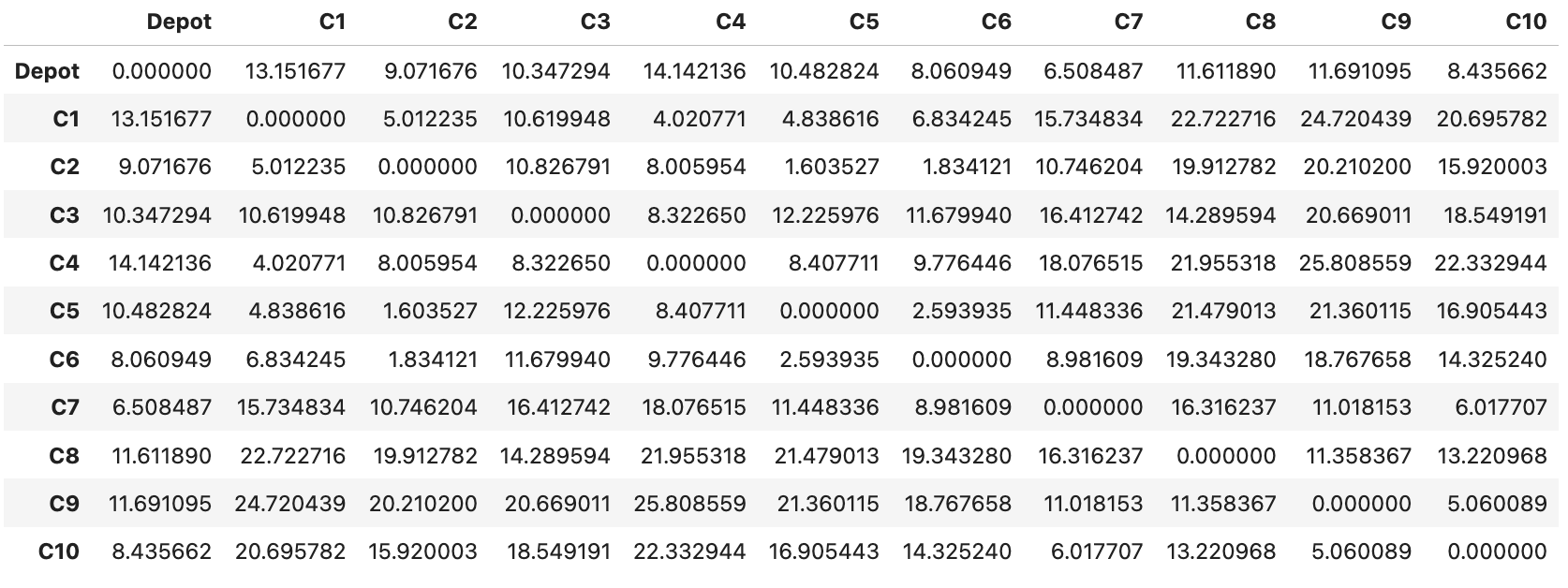
Figure 4: Vehicle Data

### **Distance Matrix**

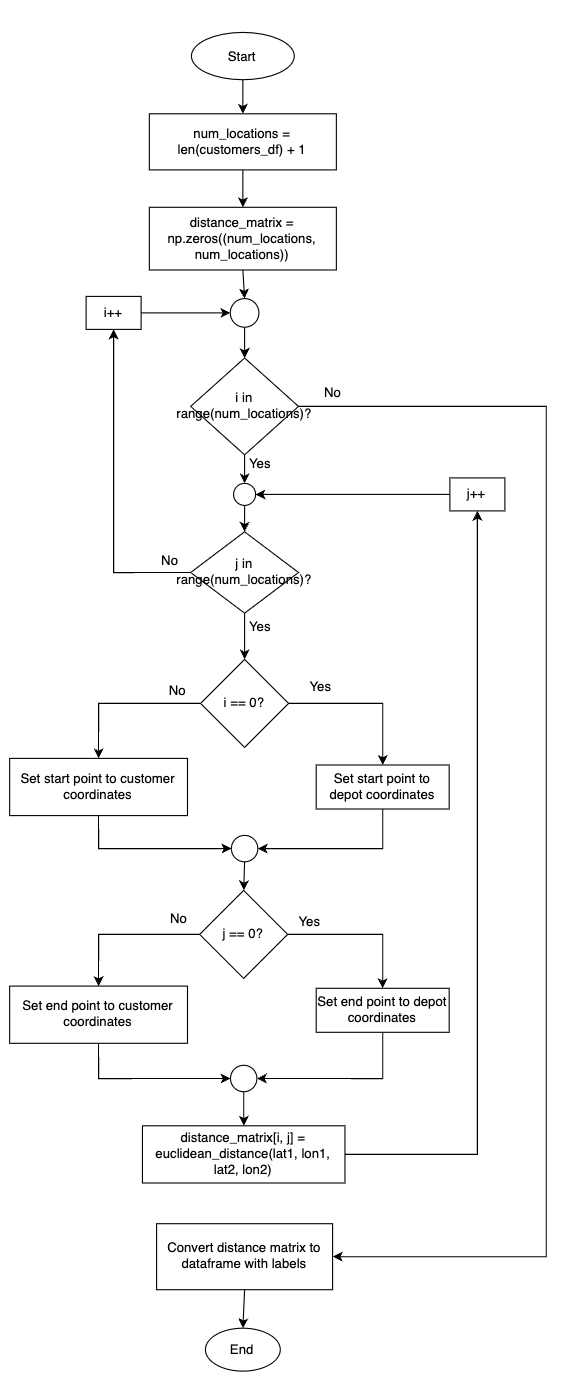
After creating dataframes for all customers and vehicles, we proceed to creating the function for calculating Euclidean distance and generating the distance matrix between the customers and the depot.

#### **Euclidean Distance Calculation**

|  |  |
| --- | --- |
| **Function Name** | euclidean\_distance |
| **Purpose** | To calculate the Euclidean distance between two coordinates. |
| **Formula** | 100 \* √((Longitude2-Longitude1)^2 + (Latitude2-Latitude1)^2) |
| **Function overview** | 1. Calculate the Euclidean distance between two points using the given formula. 2. Create a distance matrix to store distances between each pair of locations (customers and depot). |

Figure 5: Distance Matrix

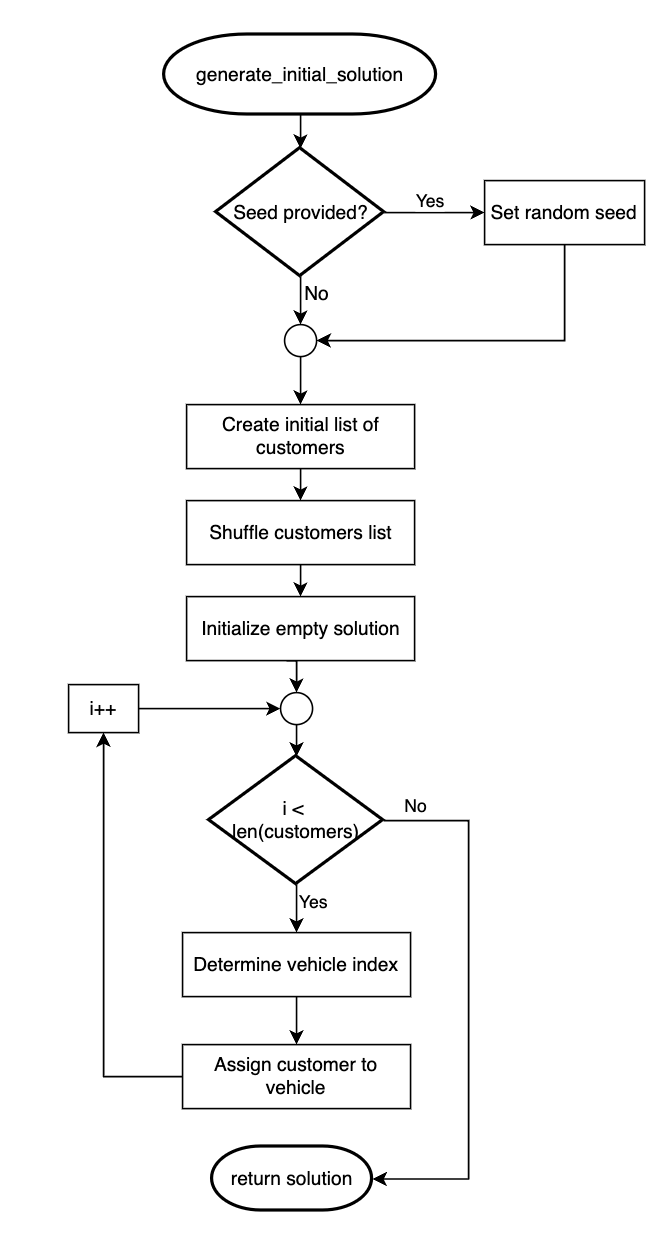
#### **Flowchart**



## **Generate Initial Solution**

|  |  |
| --- | --- |
| **Function Name** | generate\_initial\_solution |
| **Purpose** | To generate an initial solution by assigning customers to vehicles randomly using a round-robin approach. |
| **Inputs** | * num\_customers: Number of customers to be serviced * num\_vehicles: Number of vehicles available * seed: Optional seed for random number generation to ensure reproducibility |
| **Outputs** | * solution: Initial solution of the routes for each vehicle. |
| **Function Overview** | 1. Create an initial list of customers. 2. Shuffle the list of customers. 3. Assign customers to vehicles using round-robin arrangement. |

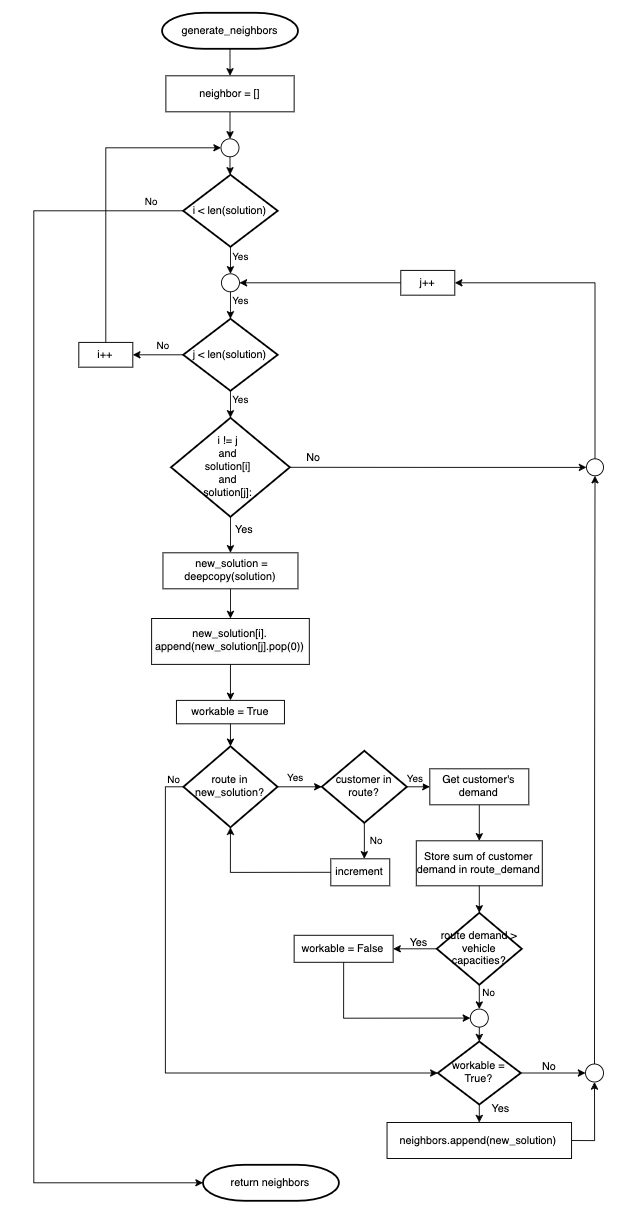
### **Flowchart**



## **Generate Neighbors**

|  |  |
| --- | --- |
| **Function Name** | generate\_neighbors |
| **Purpose** | To generate neighboring solutions by swapping customers between routes. |
| **Inputs** | * solution: Current solution * vehicle\_capacities: Capacity limits for each type of vehicle |
| **Outputs** | * neighbors: List of neighboring solutions. |
| **Function overview** | 1. Iterate over each pair of vehicles. 2. Swap the first customer from one vehicle's route to another's. 3. Check if the new solution is workable based on vehicle capacities. |

### **Flowchart**



## **Solution Cost Calculation**

|  |  |
| --- | --- |
| **Function Name** | calculate\_cost |
| **Purpose** | To calculate the total cost and distance of a given solution. |
| **Inputs** | * solution: Current solution * distance\_matrix: Matrix of distances between each location * vehicle\_costs: Costs of each type of vehicle * vehicle\_capacities: Capacity limits for each type of vehicle |
| **Outputs** | * total\_cost: The total cost of the solution * total\_distance: The total distance traveled in the solution |
| **Function overview** | 1. For each route, calculate the distance and demand. 2. Determine the vehicle type based on demand. 3. Calculate the route cost based on the vehicle type. |

### **Flowchart**

## 

## **Tabu Search Algorithm**

|  |  |
| --- | --- |
| **Function Name** | tabu\_search\_algorithm |
| **Purpose** | To find the optimal vehicle routing solution. It iterates over possible solutions, evaluating and updating the best solution found. |
| **Inputs** | * distance\_matrix: Matrix of distances between each location * vehicle\_costs: Costs of each type of vehicle * vehicle\_capacities: Capacity limits for each type of vehicle * num\_customers: Number of customers to be serviced * num\_vehicles: Number of vehicles available * max\_iterations: Maximum number of iterations for the search * seed: Optional seed for random number generation to ensure reproducibility |
| **Outputs** | * best\_solution: The best solution found * best\_cost: The cost of the best solution * iteration\_costs: The cost of the solution at each iteration |
| **Function Overview** | 1. Initialize a solution. 2. Iteratively explore neighboring solutions. 3. Update the best solution based on cost and Tabu list constraints. 4. Return the best solution after the maximum iterations or if no neighbors are found. |

## 

### **Flowchart**

## 

## **Finding The Best Solution**

|  |  |
| --- | --- |
| **Function Name** | find\_best\_solution |
| **Purpose** | To execute the Tabu Search algorithm multiple times to ensure the best possible solution is found. It returns the overall best solution from all runs. |
| **Inputs** | * num\_runs: Number of times to run the Tabu Search algorithm. * distance\_matrix: Matrix of distances between each location * vehicle\_costs: Costs of each type of vehicle. * vehicle\_capacities: Capacity limits for each type of vehicle * num\_customers: Number of customers to be serviced * num\_vehicles: Number of vehicles available * max\_iterations: Maximum number of iterations for each Tabu Search run |
| **Outputs** | * best\_overall\_solution: The best solution found across all runs * best\_overall\_cost: The cost of the best overall solution * best\_iteration\_costs: The cost of the solution at each iteration for the best run |
| **Function Overview** | 1. Run the Tabu Search algorithm multiple times. 2. Compare the best solution from each run. 3. Select the overall best solution and its associated cost. |

### **Flowchart**

## 

## **Summarizing Solution**

|  |  |
| --- | --- |
| **Function Name** | summarize\_solution |
| **Purpose** | After finding the best solution using the above two functions, this function is used to provide a detailed summary of the solution. It breaks down the routes, costs, and other details in the required format. |
| **Inputs** | * solution: The best solution found by the Tabu Search algorithm * distance\_matrix: Matrix of distances between each location * vehicle\_costs: Costs of each type of vehicle * vehicle\_capacities: Capacity limits for each type of vehicle |
| **Outputs** | * summary: A detailed breakdown of the solution, including routes, costs, and demands * total\_cost: The total cost of the solution * total\_distance: The total distance traveled in the solution |
| **Function Overview** | 1. Iterate through each route in the solution. 2. Calculate distances, demands, and costs for each route. 3. Determine the appropriate vehicle type based on demand. 4. Construct summary of the solution. |

### 

### **Flowchart**