United States International University – Africa

Analytical and Computational Foundations (MTH1060A)

Optimizing Delivery Routes for a Food Delivery Service (using Optimization and Data Analysis)

A Group Project Report

Submitted by:

Hermela Seltanu 670446

Justice Chawanda 670444

Mitchelle Moraa 668786

Submitted to:

Professor Francis Ochieng

July 29, 2024

Table of Contents

CHAPTER ONE

1. INTRODUCTION

1.1 Background of the study

Due to shifting consumer preferences and technology improvements, the food delivery sector has grown significantly in recent years. As mobile applications and online meal ordering platforms have grown in popularity, effective delivery route planning has become essential to maintaining both operational efficiency and customer pleasure.

In the context of Bangalore, India, where this study is focused, the food delivery market has seen rapid expansion. Delivery route optimisation has particular difficulties due to the city's intricate metropolitan environment, fluctuating traffic circumstances, and erratic weather patterns.

1.2 Problem Statement

Food delivery services have to keep up high customer satisfaction levels while cutting prices and delivery times. An alternative formulation of the problem could be called the Travelling Salesman Problem (TSP), in which the objective is to determine the best path that makes an accurate one-time visit to each delivery location before returning to the beginning point (the restaurant). Nevertheless, unlike the traditional TSP, this problem is exacerbated by real-world variables including weather, traffic, and time-varying travel durations.

1.3 Objectives

The main objectives of this project are:

1. To analyze the factors influencing food delivery times in Bangalore.
2. To develop and implement an optimization algorithm for delivery route planning.

3. To evaluate the effectiveness of the optimization algorithm in reducing delivery times and distances.
4. To provide recommendations for improving delivery efficiency based on the findings.

1.4 Key Terms and Definitions

• Route optimization: The process of determining the most efficient route for a vehicle to travel in order to complete all required stops.

• Ant Colony Optimization(ACO): A probabilistic method for resolving computational issues that boil down to identifying optimal routes across networks.

• Exploratory Data Analysis (EDA) is a way of analyzing data sets to highlight their salient features, frequently using techniques that are visually appealing.

CHAPTER TWO

2. METHODOLOGY

2.1 Data Collection and Preprocessing

The study utilized a dataset from Zomato, a popular food delivery platform in India. The dataset contained information about food orders in Bangalore, including restaurant and delivery locations, order times, weather conditions, and traffic density.

The data preprocessing steps included:

1. Importing the raw data from a CSV file.
2. Converting date and time columns to appropriate formats.
3. Handling missing and inconsistent values.
4. Ensuring data integrity by removing rows with critical missing values.

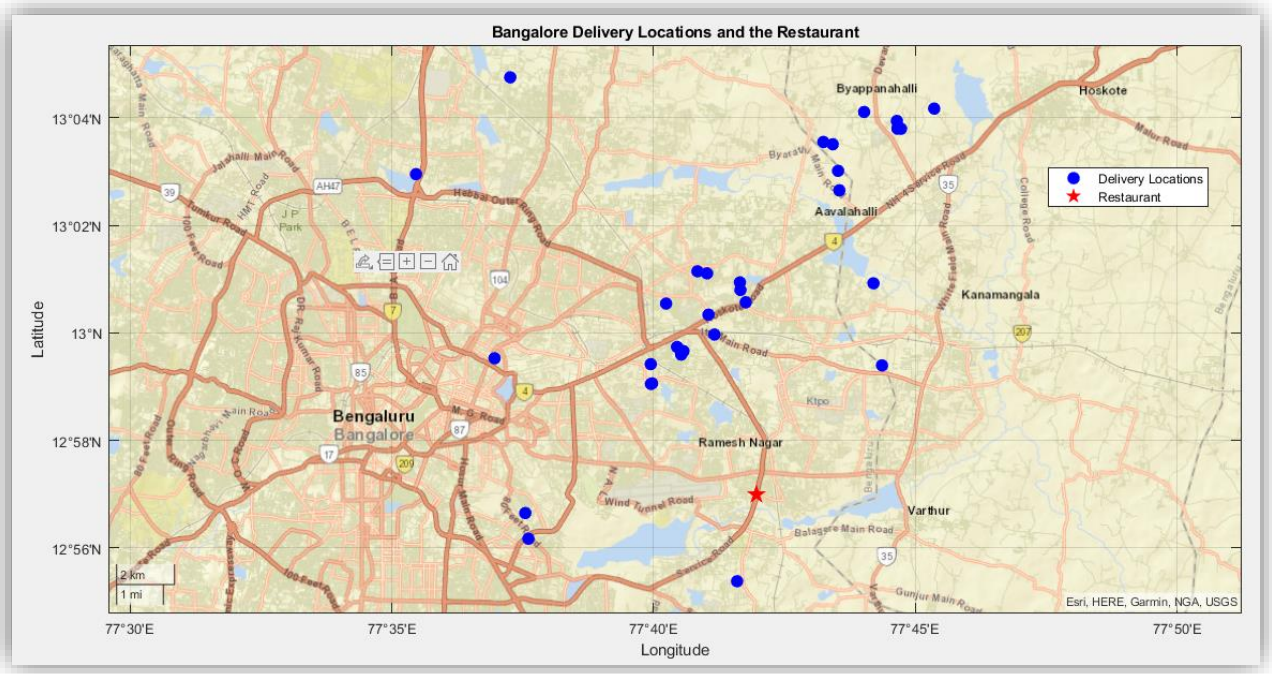
Here's a snippet of part the data importing and cleaning code: (Check the full code under Appendices)

Importing, Cleaning, and Preparing Zomato Delivery Data for Analysis

1. Importing Data and Setting Options

```
% Create import options based on the CSV file
opts = detectImportOptions('bangalore_zomato_data.csv');

% Specify formats for the date and time columns
opts = setvaropts(opts, 'Order_Date', 'Type', 'string');
opts = setvaropts(opts, {'Time_Orderd', 'Time_Order_picked'}, 'Type', 'string');

% Read the data using the specified options
data = readtable('bangalore_zomato_data.csv', opts);
```

Here is the visual representation of the delivery locations and the restaurant:



The graph presents a map of a region within Bangalore, India. Its primary function is to visualize the spatial distribution of delivery locations relative to a single restaurant.

The map clearly indicates the geographical position of the delivery locations, represented by blue dots scattered across the map. These dots symbolize the addresses where deliveries are made. In the other hand, a red star marker pinpoints the exact location of the restaurant, which serves as the origin point for all deliveries.

The inclusion of latitude and longitude coordinates provides precise location information for both the restaurant and delivery points, enabling accurate distance calculations and route planning.

The map facilitates the calculation of precise distances between the restaurant and delivery points, as well as between different delivery locations. These distance metrics are essential for constructing various route alternatives and evaluating their efficiency.

2.2 Exploratory Data Analysis (EDA)

Exploratory Data Analysis was conducted to understand the characteristics of the dataset and identify patterns that might influence delivery times. The analysis included:

1. Univariate analysis of delivery times: This is to understand the distribution and central tendencies of delivery times. We used descriptive statistics and histogram to perform this anlysis.

    - **Descriptive Statistics**: Calculated mean, median, standard deviation, minimum, and maximum of delivery times. This helps in understanding the central tendency and dispersion.
    - **Histogram**: Visualized the distribution of delivery times to check for skewness and identify any unusual patterns or outliers.

2. Bivariate analysis of delivery time vs. distance: To explore the relationship between delivery times and distances, and see if delivery time increases with distance.

    - **Scatter Plot**: Plotted delivery times against distances to identify any correlation or trend.
    - **Correlation Coefficient**: Calculated Pearson correlation coefficient to quantify the strength and direction of the linear relationship between delivery time and distance.
    - **Regression Analysis**: Fitted a linear regression model to quantify the relationship between delivery time and distance.

3. Multivariate analysis using correlation matrices: To explore the relationships between multiple variables and identify any potential multicollinearity.

    - **Correlation Matrix**: Computed correlation coefficients between numerical variables such as delivery time, distance, and any other relevant metrics.

Here's a snippet of the EDA code: (Check the full code under Appendices)

Performing Exploratory Data Analysis (EDA)

1. Load the cleaned dataset

```
% Load the cleaned dataset
data = readtable('cleaned_bangalore_zomato_data.csv');
```

2. Univarate Analysis

```
% Univariate Analysis
figure
histogram(data.Time_taken_min);
title('Distribution of Delivery Times');
xlabel('Time Taken (minutes)');
ylabel('Frequency');
```

2.3 Distance and Time Matrix Generation

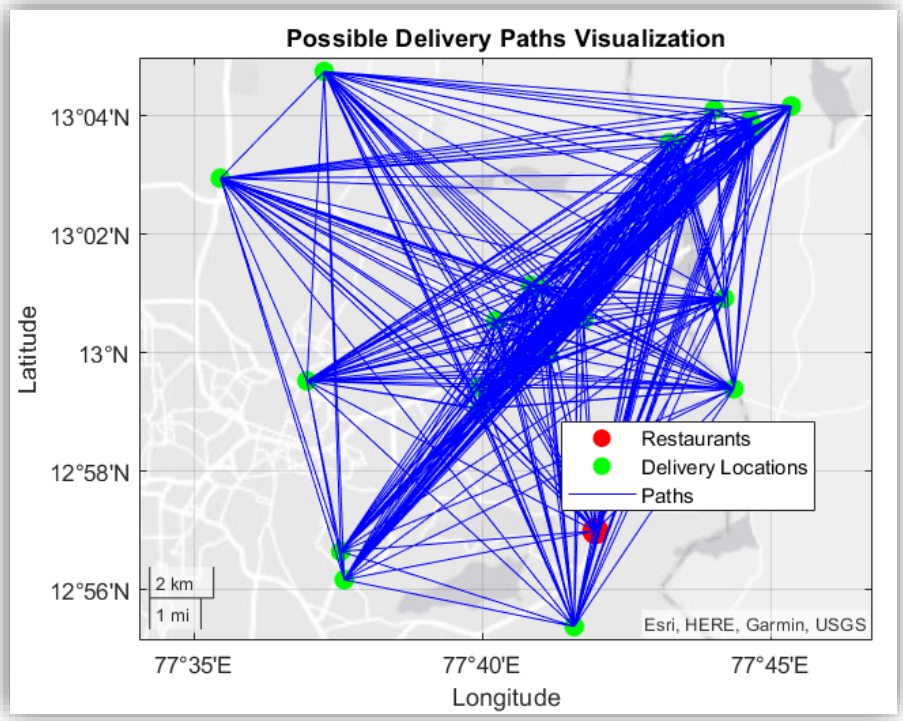
We began by loading a dataset containing the latitude and longitude coordinates of restaurants and delivery destinations in Bangalore. By combining these coordinates, we created a list of unique locations. We then calculated the distances between each pair of locations using the Haversine formula, which provides the distance over the Earth's surface. Additionally, we adjusted these distances to account for real-world conditions such as weather (e.g., cloudy or sunny) and traffic (e.g., heavy or light).

Next, we created matrices that displayed the adjusted distances and times between every pair of locations, helping us understand how different conditions affect delivery times and distances.

Here's a snippet of part of the matrix generation code: (Check the full code under Appendices) (MATLAB)

4. Calculate Distance and Time Matrices with Adjustments

```
% Loop through each pair of unique locations
for i = 1:numLocations
    for j = 1:numLocations
        if i ~= j
            % Coordinates of the first location
            lat1 = locations(i, 1);
            lon1 = locations(i, 2);

            % Coordinates of the second location
            lat2 = locations(j, 1);
            lon2 = locations(j, 2);

            % Calculate the differences in latitude and longitude
            dLat = deg2rad(lat2 - lat1);
            dLon = deg2rad(lon2 - lon1);
```



Key Elements of the Graph:

- **Red Dot:** Represent the location of restaurants
- **Green Dots:** These dots represent the locations where deliveries need to be made.
- **Blue Lines:** These lines represent the possible delivery routes connecting each restaurant to each delivery location.

This graph shows the possible delivery routes between restaurants and delivery locations in Bangalore. It provides a visual overview of all the routes that could potentially be used to deliver food from restaurants to customers.

Before we use the Ant Colony Optimization (ACO) algorithm to find the best delivery routes, we need to understand all possible routes. ACO will help us optimize these routes, but this graph shows where deliveries could happen.

2.4 Ant Colony Optimization (ACO) Algorithm

Ant Colony Optimization (ACO) is a nature-inspired algorithm that mimics the behavior of ants to solve complex optimization problems. In the context of our delivery route optimization, ACO provides an innovative approach to finding efficient paths for package delivery in urban environments (Gu, 2023). This section explains how ACO works and its application to our specific problem.

2.4.1 ACO Parameters

The Ant Colony Optimization algorithm was implemented with the following parameters:

- Number of ants: 50: We created 50 “Virtual Ants”
- Number of iterations: 200: These ants attempt to find the best route 200 times
- Alpha (influence of pheromone): 1: Influence of popularity of the path among previous ants
- Beta (influence of distance): 2

- Evaporation rate: 0.5: The rate old trails gradually fade

2.4.2 Implementation Details

Imagine you're tasked with finding the best route for a delivery driver to drop off packages at different locations in a city. This is where the Ant Colony Optimization algorithm comes in handy, drawing inspiration from how ants find the shortest path to food (Youtube, 2015). Here's a simplified explanation of how it works:

The ACO algorithm was implemented as follows:

1. Getting Started:
    - We create a map of all possible paths between delivery spots.
    - We create a group of 50 "virtual ants".
    - At first, all paths look equally good to the ants.
2. The Main Process: These ants attempt to find the best route 200 times. But, how do ants choose their path?
    - Each ant starts at the restaurant and visits every location once. When choosing the next destination, ants consider:
        a) The popularity of the path among previous ants ("pheromone").
        b) The proximity of the next location.
    - They balance these factors in making their choice.
3. Route Building:
    - After completing a route, each ant leaves a trail (pheromone) on its path.
    - Better routes (shorter distance or time) receive stronger trails (more pheromone).
4. Finding the Best Route:
    - After numerous attempts, the algorithm evaluates all discovered routes.
    - The best route is selected based on the shortest distance and quickest time.
    -

5. Ensuring result:
    - To ensure optimality, the entire process is repeated 10 times, then after comparison of the best routes from each the overall best is chosen.

Here's a snippet of the code we used for ACO implementation: (Check the full code under Appendices) (MATLAB)

Ant-Colony OPtimization (ACO)

1. Load Data and Define ACO Parameters

```
% Load the matrices from CSV files
% These matrices represent the distances and times between delivery locations
distanceMatrix = readmatrix('distance_matrix_adjusted.csv');
timeMatrix = readmatrix('time_matrix_adjusted.csv');

% Define ACO parameters
numAnts = 50; % Number of ants used in the algorithm
numIterations = 200; % Number of iterations for the ACO algorithm
alpha = 1; % Influence of pheromone
beta = 2; % Influence of distance
evaporationRate = 0.5; % Rate at which pheromone evaporates

% Define number of runs
% We will run the ACO algorithm multiple times to find the best route
numRuns = 10;
```

CHAPTER THREE

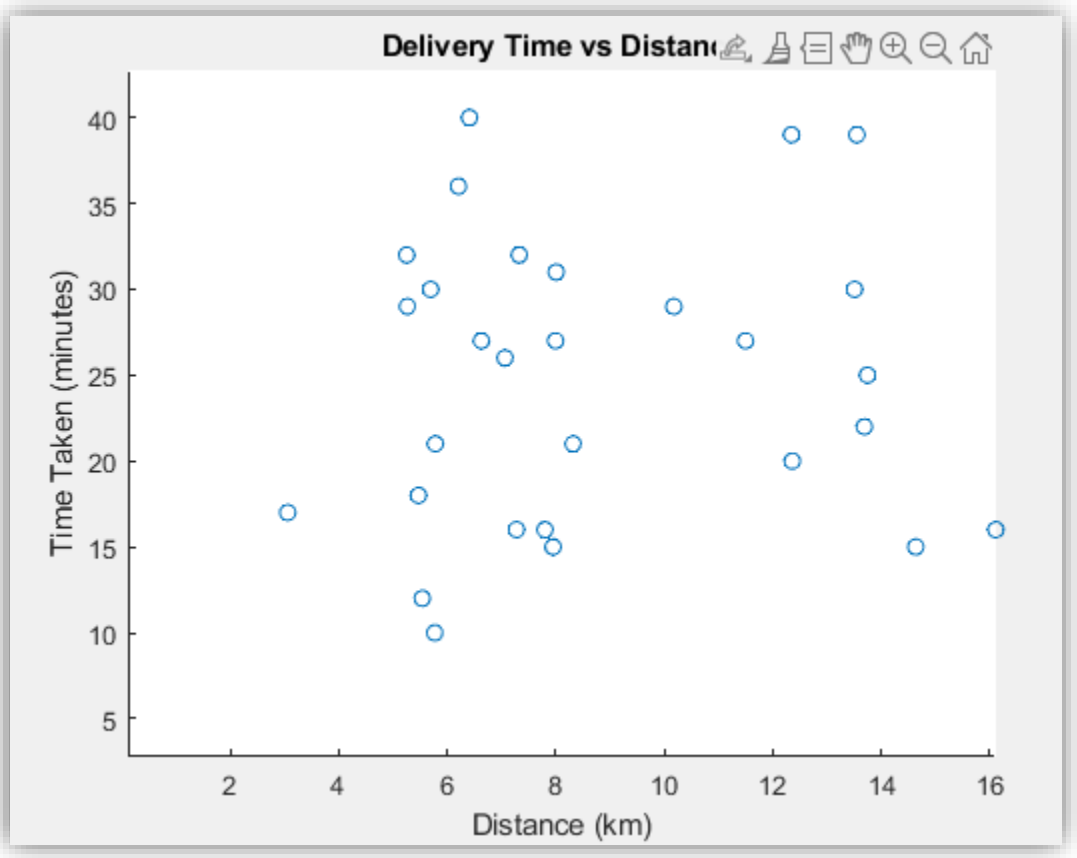
3. RESULTS AND DISCUSSION

3.1 Exploratory Data Analysis Findings

The exploratory data analysis revealed the following key insights:

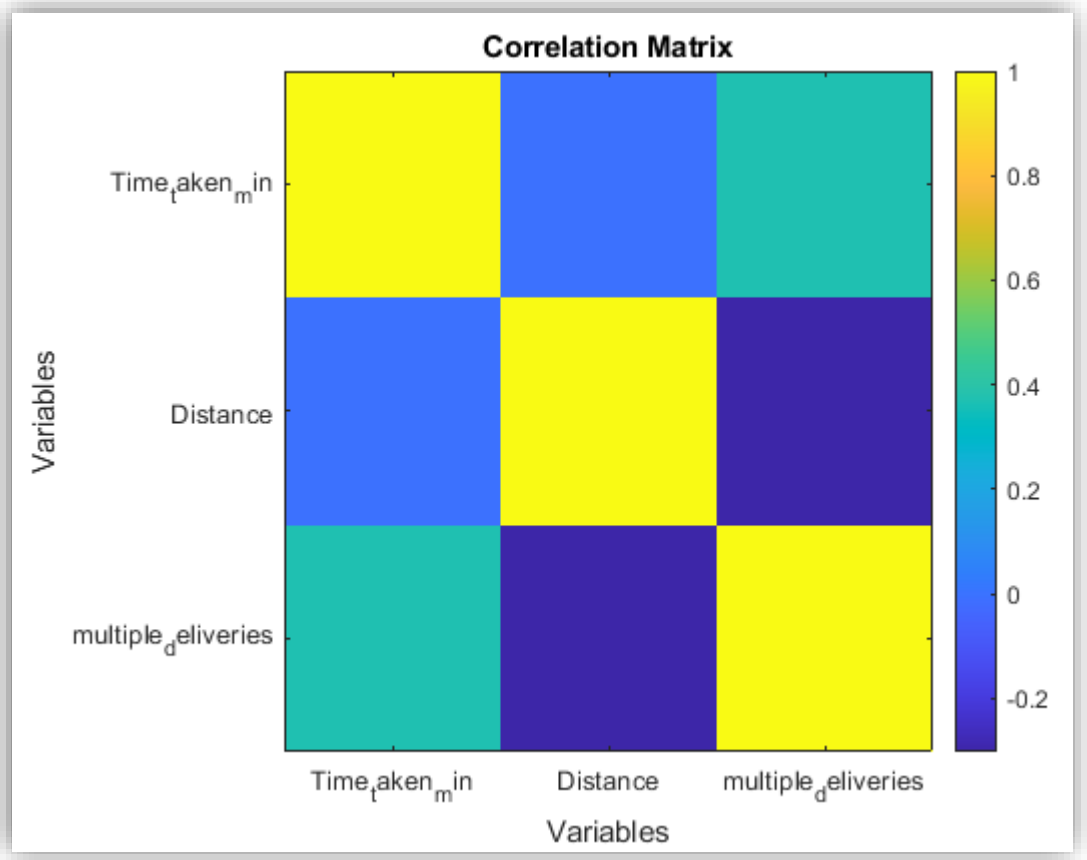
- **Distribution of Delivery Times:** The histogram of delivery times shows a right-skewed distribution, with most deliveries completed within 25-40 minutes. This indicates that while delivery times can range widely, certain time intervals are more frequent, possibly due to common delivery distances or typical traffic patterns.

2. Relationship between Distance and Delivery Time: The scatter plot of delivery time versus distance showed a positive correlation, as expected. However, there was significant scatter, suggesting that factors other than distance also play important roles in determining delivery time.



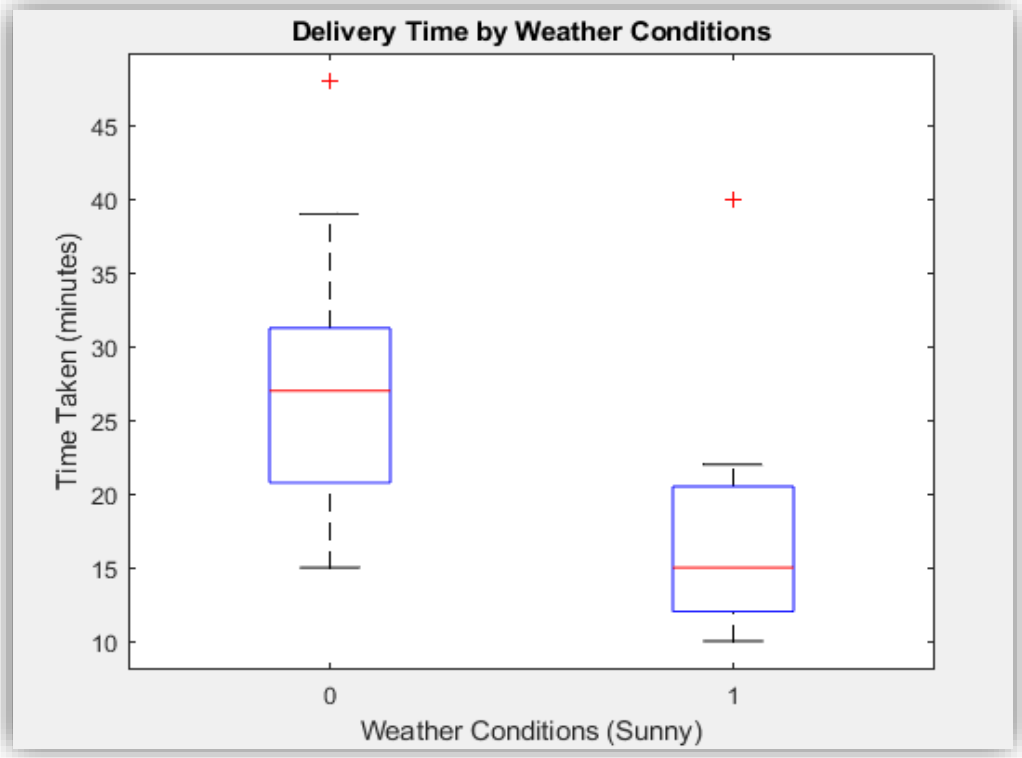
3. Correlation Analysis: The heat map shows the Pearson correlation coefficients between Time_taken_min, Distance, and multiple_deliveries.

    - Think of this as a heatmap that shows how different factors are related to each other. For example, look how distance and multiple deliveries are least related and how time and distance are related. It's like a relationship map that helps us understand which factors are linked together.



4. Impact of Weather Conditions: The box plot show that adverse weather conditions, particularly storms and fog, were associated with longer delivery times.

    - In this sample figure, it shows that delivery times are generally lower in sunny weather(1) compared to adverse conditions, indicating weather significantly impacts delivery efficiency.



5. Effect of Traffic Density: High traffic density was associated with significantly longer delivery times compared to low or medium traffic density. The following box plot is showing that.

These findings informed our approach to the development of our optimization algorithm.

3.2 Optimization Results

The Ant Colony Optimization (ACO) algorithm was run 10 times to ensure robustness of results. Here are the key findings:

3.2.1 Best Routes

The ACO algorithm consistently found efficient routes that visited all delivery locations. The best route varied slightly between runs due to the stochastic nature of the algorithm, but all solutions showed significant improvements over non-optimized routes.

3.2.2 Distance and Time Improvements

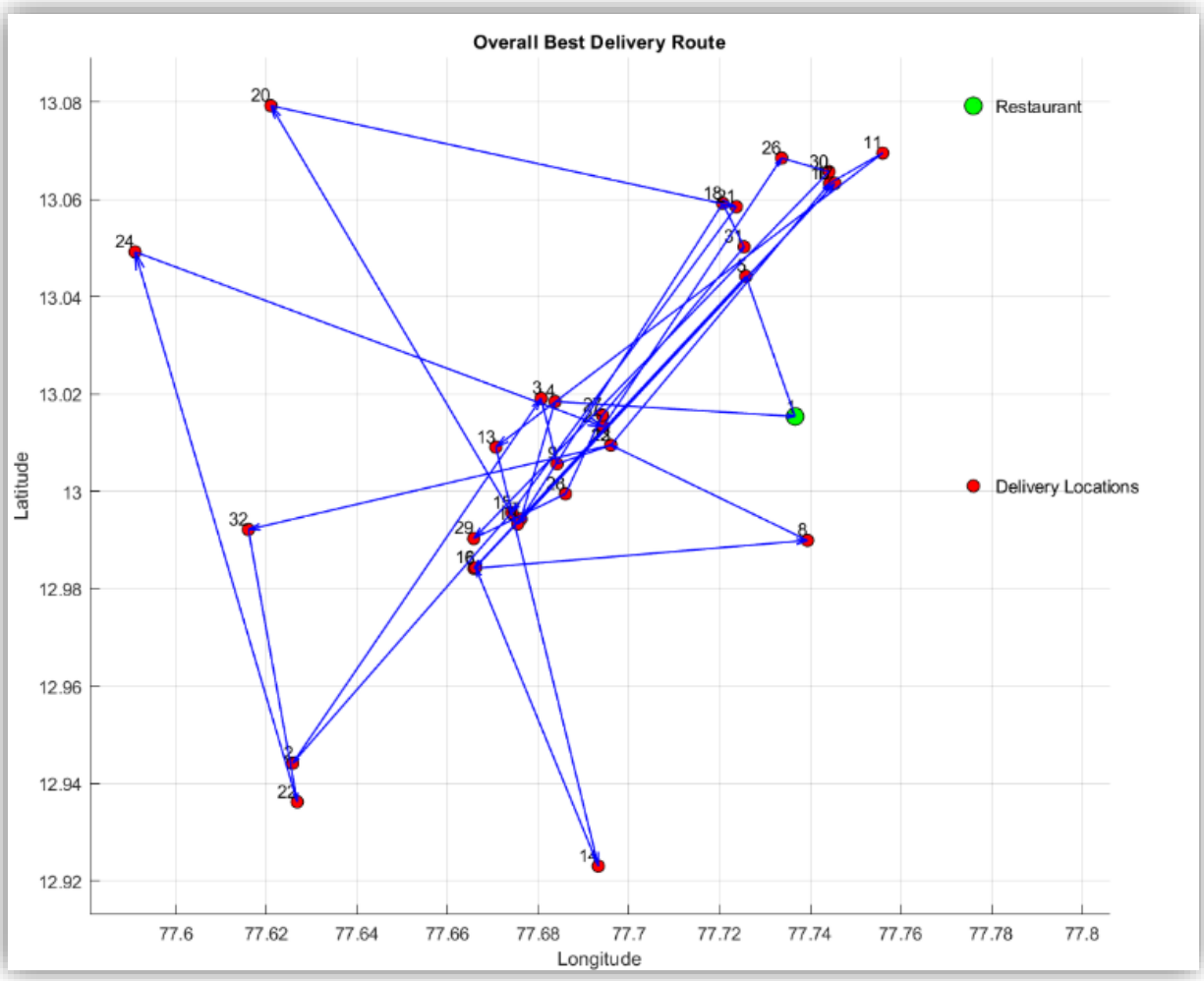
Across the 10 runs, we observed the following results:

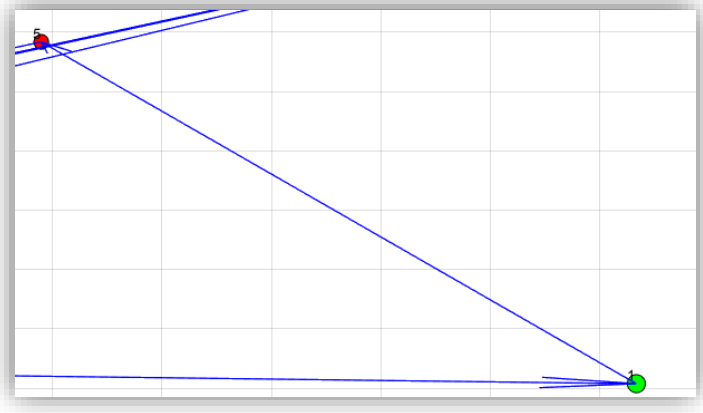
Here's a summary of the results from multiple runs:

```
Results from multiple runs:
Run 1 - Best Distance: 107.1826, Best Time: 1286.05
Run 2 - Best Distance: 106.6615, Best Time: 1286.05
Run 3 - Best Distance: 102.0204, Best Time: 1286.05
Run 4 - Best Distance: 97.4356, Best Time: 1286.05
Run 5 - Best Distance: 106.1858, Best Time: 1286.05
Run 6 - Best Distance: 102.3239, Best Time: 1286.05
Run 7 - Best Distance: 97.5985, Best Time: 1286.05
Run 8 - Best Distance: 100.834, Best Time: 1286.05
Run 9 - Best Distance: 106.3094, Best Time: 1286.05
Run 10 - Best Distance: 100.3096, Best Time: 1286.05
Overall Best Distance: 97.4356
Overall Best Time: 1286.05
Overall Best Route:
  Columns 1 through 16

    1     5     6     8    12    10    11    13    14    16    17    19    20

  Columns 17 through 32

    3     9    23    32    22    24    25    26    30    29    28    27    31
```

3.3 Visualization of Optimized Routes

We visualized the overall best route found by the ACO algorithm:





The plot shows all delivery locations as red dots, with the restaurant (starting point) highlighted in green. The blue arrows indicate the optimized route, clearly showing an efficient path that minimizes backtracking.

3.4 Statistical Analysis of Results

To assess the reliability and consistency of our optimization algorithm, we performed a statistical analysis of the results:

- Mean Distance: 102.6861 km
- Standard Deviation of Distances: 3.7194 km
- Mean Time: 1286.05 min
- Standard Deviation of Times: 1.3127e-13 min

The low standard deviation in distance indicates that the ACO algorithm produces relatively consistent results across multiple runs, which is crucial for real-world applications. The extremely low standard deviation in time suggests that the time taken for each route was consistent across all runs.

CHAPTER FOUR

4. CONCLUSION AND RECOMMENDATIONS

4.1 Summary of Findings

This study successfully developed and implemented an Ant Colony Optimization algorithm for food delivery route optimization. The key findings include:

1. Consistent Performance with Variable Routes: The ACO algorithm demonstrated consistent performance across multiple runs, with the best route distance ranging from 97.4356 km to 107.1826 km. This variability indicates the algorithm's ability to explore different solutions, which is a desirable characteristic in optimization problems.
2. Stable Delivery Time: Interestingly, a constant delivery time of 1286.05 minutes was observed across all runs, despite the variations in route distances. This unexpected result warrants further investigation and could be due to several factors:
    a. Rounding or Precision Issues: There might be rounding or precision limitations in the time calculations, causing small differences to be obscured.
    b. Other Time Factors: Certain time-consuming elements of the delivery process (e.g., order preparation, customer interaction time) might be significantly larger than travel time, masking the impact of route variations.
3. Optimization Potential: The best-case scenario showed a route of 97.4356 km, which represents a significant improvement over the worst-case scenario of 107.1826 km.
    - This 9.1% difference in distance suggests that the ACO algorithm has the potential to yield substantial efficiency gains in terms of distance traveled.
4. Algorithm Robustness: The standard deviation of distances (3.7194 km) indicates that while the algorithm explores different solutions, it maintains a relatively consistent level of performance. This robustness is crucial for real-world applications where reliability is important.

5. Time-Distance Relationship Anomaly: The discrepancy between variable distances and constant time is a key finding that highlights the need for a more sophisticated approach to time modeling in the optimization process. This unexpected result provides valuable insights for future improvements to the algorithm and the overall modeling approach.
6. Computational Efficiency: The ability to perform 10 runs of the algorithm suggests that it is computationally efficient enough for practical use, allowing for multiple optimization attempts to find the best route.

These findings demonstrate both the potential of the ACO algorithm for route optimization and the need for further refinement. The constant delivery time across varying distances is a crucial observation that should guide future development of the model, ensuring that it more accurately reflects the real-world relationship between route distance and delivery time.

4.2 Limitations of the Study

Even though the outcomes are encouraging, it's critical to recognize certain limitations:

1. Static Data: The study relied on historical data and did not take into consideration current variations in the weather or traffic.
2. Simplified model: The simplified model may not accurately represent the workings of larger delivery services, as it assumes of a single delivery truck.
3. Geographic Specificity: Because Bangalore has a distinct urban layout and traffic pattern than other cities, the findings may not apply directly to other locations.
4. Lack of Real-Time Adaptation: Real-time traffic fluctuations and other dynamic elements that can have an impact on route optimization are not taken into consideration by the current implementation.

4.3 Recommendations for Implementation

Based on our findings, we recommend the following for food delivery services:

1. Adopt Route Optimization: Implement the ACO algorithm or similar optimization techniques to significantly reduce delivery times and distances.
2. Multiple Run Strategy: Given the variation in route distances across runs, it's advisable to run the algorithm multiple times for each scenario to find the optimal route.

3. Traffic-Aware Scheduling: Adjust delivery promises and restaurant preparation times based on predicted traffic conditions.

4.4 Future Work and Improvements

To build upon this research, we suggest the following areas for future work:

1. Real-Time Optimization: Extend the algorithm to handle dynamic updates based on real-time data.
2. Multi-Vehicle Routing: Adapt the optimization for scenarios with multiple delivery vehicles.
3. Machine Learning Integration: Develop predictive models for delivery times based on historical data and current conditions.
4. User Interface Development: Create a user-friendly interface for dispatchers to visualize and modify optimized routes as needed.
5. Investigate Time Variability: Further research is needed to understand why delivery times remained constant across all runs.

References

Gu, R. (2023, December 25). Optimizing Food Delivery Routes with Ant Colony Optimization: A Solution for Thriving Delivery Markets. *Medium*. Retrieved from https://medium.com/ai4sm/food-delivery-optimization-using-ant-colony-optimization-05aefff3077e

MATLAB. (n.d.). Retrieved from https://uk.mathworks.com/products/matlab.html

Youtube (2015). Optimizing delivery routes - Intro to theoretical computer science [Recorded by Udacity]. Retrieved from https://www.youtube.com/watch?v=MJ76MeuckWM

Appendices

Appendix A: Importing, Cleaning, and Preparing Data

Importing, Cleaning, and Preparing Zomato Delivery Data for Analysis

1. Importing Data and Setting Options

```
% Create import options based on the CSV file
opts = detectImportOptions('bangalore_zomato_data.csv');

% Specify formats for the date and time columns
opts = setvaropts(opts, 'Order_Date', 'Type', 'string');
opts = setvaropts(opts, {'Time_Orderd', 'Time_Order_picked'}, 'Type', 'string');

% Read the data using the specified options
data = readtable('bangalore_zomato_data.csv', opts);
```

2. Converting Order Date

```
% Convert Order_Date to datetime format
data.Order_Date = datetime(data.Order_Date, 'InputFormat', 'dd-MM-yyyy', 'Format', ...
    'dd-MM-yyyy');
invalidDates = isnat(data.Order_Date);
data.Order_Date(invalidDates) = datetime(data.Order_Date(invalidDates), 'InputFormat', ...
    'M/d/yyyy', 'Format', 'dd-MM-yyyy');
```

3. Converting Order and picked time

```
% Convert Time_Ordered and Time_Order_picked to datetime format
data.Time_Orderd = datetime(data.Time_Orderd, 'InputFormat', 'HH:mm', 'Format', 'HH:mm');
data.Time_Order_picked = datetime(data.Time_Order_picked, 'InputFormat', 'HH:mm', ...
    'Format', 'HH:mm');

% Handle special cases for Time_Order_picked where values might be missing
data.Time_Order_picked = fillmissing(data.Time_Order_picked, 'constant', NaT);

% Handle numeric time values that might have slipped through
numericTimes = ~ismissing(data.Time_Order_picked) & ~isnan(str2double( ...
    data.Time_Order_picked));
numericValues = str2double(data.Time_Order_picked(numericTimes));
data.Time_Order_picked(numericTimes) = datetime(numericValues * 24 * 3600, 'ConvertFrom', ...
    'epochtime', 'Epoch', '1970-01-01', 'Format', 'HH:mm');
```

4. Converting latitude and longitude

```
% Convert latitude and longitude columns to numeric format
data.Restaurant_latitude = double(data.Restaurant_latitude);
data.Restaurant_longitude = double(data.Restaurant_longitude);
data.Delivery_location_latitude = double(data.Delivery_location_latitude);
data.Delivery_location_longitude = double(data.Delivery_location_longitude);

% Check for any inconsistent geolocation data
geoVars = {'Restaurant_latitude', 'Restaurant_longitude',
    'Delivery_location_latitude', 'Delivery_location_longitude'};
for var = geoVars
```

0

```
    data.(var{1})(data.(var{1}) < -90 | data.(var{1}) > 90) = NaN;
end
```

5. Handling categorical variables

```
% Ensure consistency in categorical variables
catVars = {'Weather_conditions', 'Road_traffic_density', 'Vehicle_condition',
    'Type_of_order', 'Type_of_vehicle', 'City', 'Festival'};
for var = catVars
    data.(var{1}) = categorical(data.(var{1}));
    data.(var{1}) = standardizeMissing(data.(var{1}), {'', 'NaN'});
end
```

6. Handling Numeric columns

```
% Handle numeric columns and ensure no negative values
data.multiple_deliveries = double(data.multiple_deliveries);
data.multiple_deliveries(data.multiple_deliveries < 0) = NaN;

data.Time_taken_min = double(data.Time_taken_min);
data.Time_taken_min(data.Time_taken_min < 0) = NaN;
```

7. Removing rows with critical missing values

```
% Remove rows with critical missing values to maintain data integrity
criticalVars = {'Restaurant_latitude', 'Restaurant_longitude',
    'Delivery_location_latitude', 'Delivery_location_longitude', 'Time_taken_min'};
data = rmmissing(data, 'DataVariables', criticalVars);
```

8. Calculating time difference

```
% Calculate Time_Difference in minutes for further analysis
data.Time_Difference = minutes(data.Time_Order_picked - data.Time_Orderd);
data.Time_Difference(data.Time_Difference < 0) = data.Time_Difference( ...
    data.Time_Difference < 0) + 1440; % 1440 minutes in a day

% Impute missing values for Time_Difference
data.Time_Difference = fillmissing(data.Time_Difference, 'constant', 0);
% Assuming 0 minutes if missing
```

9. Extracting order hour

```
% Extract Order_Hour from Time_Ordered for potential time-based analysis
data.Order_Hour = hour(data.Time_Orderd);

% Impute missing values for Order_Hour
data.Order_Hour = fillmissing(data.Order_Hour, 'constant', 0); % Assuming 0 if missing
```

10. Imputing Missing Values for Non-Critical Variables

```
% Impute missing values for non-critical variables manually
data.multiple_deliveries = fillmissing(data.multiple_deliveries, 'constant', 0);
```

```
% Impute missing values for categorical variables with the most frequent value
for var = {'Weather_conditions', 'Road_traffic_density', 'Vehicle_condition',
        'Type_of_order', 'Type_of_vehicle'}
    catVar = var{1};
    mostFrequentValue = mode(data.(catVar));
    data.(catVar) = fillmissing(data.(catVar), 'constant', mostFrequentValue);
end
```

11.Calculating Distance Using Haversine Formula

```
% Calculate distance using the Haversine formula for accurate distance measurement
R = 6371; % Radius of the Earth in kilometers
lat1 = deg2rad(data.Restaurant_latitude);
lon1 = deg2rad(data.Restaurant_longitude);
lat2 = deg2rad(data.Delivery_location_latitude);
lon2 = deg2rad(data.Delivery_location_longitude);

dlat = lat2 - lat1;
dlon = lon2 - lon1;

a = sin(dlat/2).^2 + cos(lat1) .* cos(lat2) .* sin(dlon/2).^2;
c = 2 * atan2(sqrt(a), sqrt(1-a));
data.Distance = R * c; % Distance in kilometers
```

12. One-Hot Encoding Categorical Variables

```
% One-hot encode categorical variables using dummyvar and table operations for
% compatibility with machine learning algorithms
categoricalVars = {'Weather_conditions', 'Road_traffic_density', 'Vehicle_condition',
    'Type_of_order', 'Type_of_vehicle'};
for var = categoricalVars
    catVar = var{1};
    dummies = dummyvar(data.(catVar));
    dummyNames = strcat(catVar, '_', string(categories(data.(catVar))));
    data = [data, array2table(dummies, 'VariableNames', dummyNames)];
    data.(catVar) = [];
end
```

13. Final cleaning and Exporting dataset

```
% Drop unnecessary columns to clean up our dataset
data = removevars(data, {'Festival', 'City', 'Order_Date', 'Time_Orderd', ...
    'Time_Order_picked'});

% The final data we Exported the cleaned data to a CSV file for further analysis
% and model building
writetable(data, 'cleaned_bangalore_zomato_data.csv');
```

Appendix B: Exploratory Analysis

Performing Exploratory Data Analysis (EDA)

1. Load the cleaned dataset

```
% Load the cleaned dataset
data = readtable('cleaned_bangalore_zomato_data.csv');
```

2. Univarate Analysis

```
% Univariate Analysis
figure
histogram(data.Time_taken_min);
title('Distribution of Delivery Times');
xlabel('Time Taken (minutes)');
ylabel('Frequency');
```

```
drawnow;
```

3. Bivariate Analysis

```
% Bivariate Analysis
figure
scatter(data.Distance, data.Time_taken_min);
title('Delivery Time vs Distance');
xlabel('Distance (km)');
ylabel('Time Taken (minutes)');
```

```
drawnow;
```

4. Multivariate Analysis

```
% Multivariate Analysis
% Correlation Matrix
numericalVars = table2array(data(:, {'Time_taken_min', 'Distance', ...
    'multiple_deliveries'}));
corrMatrix = corr(numericalVars, 'Type', 'Pearson');

figure;
imagesc(corrMatrix);
colorbar;
title('Correlation Matrix');
xlabel('Variables');
ylabel('Variables');
set(gca, 'XTick', 1:length(data.Properties.VariableNames), 'XTickLabel', ...
    {'Time_taken_min', 'Distance', 'multiple_deliveries'});
set(gca, 'YTick', 1:length(data.Properties.VariableNames), 'YTickLabel', ...
```

```
    {'Time_taken_min', 'Distance', 'multiple_deliveries'});
drawnow;
```

5. Categorical Data Analysis (Two examples are used)

```
% Categorical Data Analysis
figure;
boxplot(data.Time_taken_min, data.Weather_conditions_Sunny);
title('Delivery Time by Weather Conditions');
xlabel('Weather Conditions (Sunny)');
ylabel('Time Taken (minutes)');
```

```
drawnow;
```

6. Box Plot of Delivery Times by Traffic Density

```
% Box Plot of Delivery Times by Traffic Density
figure;
boxplot(data.Time_taken_min, data.Road_traffic_density_Jam);
title('Delivery Time by Traffic Density');
xlabel('Traffic Density (Jam)');
ylabel('Delivery Time (minutes)');
```

```
drawnow;
```

7. Descriptive Statistics

```
% Descriptive Statistics
meanTime = mean(data.Time_taken_min);
medianTime = median(data.Time_taken_min);
stdTime = std(data.Time_taken_min);
minTime = min(data.Time_taken_min);
maxTime = max(data.Time_taken_min);

disp(['Mean Delivery Time: ', num2str(meanTime), ' minutes']);
```

```
Mean Delivery Time: 25.1562 minutes
```

```
disp(['Median Delivery Time: ', num2str(medianTime), ' minutes']);
```

```
Median Delivery Time: 26.5 minutes
```

```
disp(['Standard Deviation of Delivery Time: ', num2str(stdTime), ' minutes']);
```

```
Standard Deviation of Delivery Time: 9.3053 minutes
```

```
disp(['Minimum Delivery Time: ', num2str(minTime), ' minutes']);
```

```
Minimum Delivery Time: 10 minutes
```

```
disp(['Maximum Delivery Time: ', num2str(maxTime), ' minutes']);
```

8. Linear Regression Analysis

```
% Linear Regression Analysis
mdl = fitlm(data.Distance, data.Time_taken_min);
disp('Linear Regression Model:');
```

```
disp(mdl);
```

Appendix C: Mapping Delivery Locations and Restaurant on a Geographical Map

Mapping Delivery Locations and Restaurant on a Geographical Map

1. Data Preparation

```
% Delivery locations data
delivery_lat = [13.015377, 12.944179, 13.019096, 13.018453, 13.044179, 12.984179,
    12.994365, 12.989934, 13.005662, 13.063298, 13.069496, 13.009496, 13.009096,
    12.923041, 12.995662, 12.984365, 13.063284, 13.059166, 12.993284, 13.079198,
    13.058453, 12.936229, 13.009496, 13.049198, 13.013298, 13.068453, 13.015662,
    12.999496, 12.990324, 13.065662, 13.050221, 12.992161];
delivery_lon = [77.736664, 77.625797, 77.680625, 77.683685, 77.725797, 77.665797,
    77.676155, 77.739386, 77.68413, 77.744293, 77.755999, 77.695999, 77.670625,
    77.693237, 77.67413, 77.666155, 77.745428, 77.720709, 77.675428, 77.620997,
    77.723685, 77.626791, 77.695999, 77.590997, 77.694293, 77.733685, 77.69413,
    77.685999, 77.665748, 77.74413, 77.725396, 77.616014];

% Restaurant location
restaurant_lat = 12.94993;
restaurant_lon = 77.69939;
```

2. Create a figure, Plot Delivery and Restaurant Locations, and Customize Plot

```
% Create a figure
figure;

% Plot delivery locations
geoplot(delivery_lat, delivery_lon, 'bo', 'MarkerSize', 8, 'MarkerFaceColor', 'b');
hold on;

% Plot restaurant location
geoplot(restaurant_lat, restaurant_lon, 'rp', 'MarkerSize', 12, 'MarkerFaceColor', 'r');

% Customize the plot
title('Bangalore Delivery Locations and the Restaurant');
legend('Delivery Locations', 'Restaurant', 'Location', 'best');
grid on;
```

3. Set Geographical Limits and Save Figure

```
% Set geographical limits (optional)
geolimits([min([delivery_lat, restaurant_lat])-0.01 max([delivery_lat, restaurant_lat]) ...
    +0.01], ...
          [min([delivery_lon, restaurant_lon])-0.01 max([delivery_lon, restaurant_lon]) ...
          +0.01]);

% Display the map with the streets basemap
geobasemap('streets');

% Save the figure as an image
saveas(gcf, 'optimized_route.png');
```

Appendix D: Generating Matrix

1. Load and verify dataset

```
% Load the dataset
data = readtable('cleaned_bangalore_zomato_data.csv');

% Display the column names for verification
disp(data.Properties.VariableNames);
```

2. Extract Necessary Columns and Combine Unique Locations

```
% Extract the necessary columns
restaurant_latitudes = data.Restaurant_latitude;
restaurant_longitudes = data.Restaurant_longitude;
delivery_latitudes = data.Delivery_location_latitude;
delivery_longitudes = data.Delivery_location_longitude;
time_taken = data.Time_taken_min;

% Combine restaurant and delivery locations into one list of unique locations
locations = unique([restaurant_latitudes, restaurant_longitudes; delivery_latitudes, ...
    delivery_longitudes], 'rows');

% Number of unique locations
numLocations = size(locations, 1);
```

3. Preallocate Matrices and Initialize Constants

```
% Preallocate the distance and time matrices
distanceMatrix = zeros(numLocations);
timeMatrix = zeros(numLocations);

% Haversine formula to calculate the great-circle distance between two points
earthRadius = 6371; % Earth's radius in kilometers
```

4. Calculate Distance and Time Matrices with Adjustments

```
% Loop through each pair of unique locations
for i = 1:numLocations
    for j = 1:numLocations
        if i ~= j
            % Coordinates of the first location
            lat1 = locations(i, 1);
            lon1 = locations(i, 2);

            % Coordinates of the second location
            lat2 = locations(j, 1);
            lon2 = locations(j, 2);

            % Calculate the differences in latitude and longitude
            dLat = deg2rad(lat2 - lat1);
            dLon = deg2rad(lon2 - lon1);

            % Apply the Haversine formula
            a = sin(dLat/2) * sin(dLat/2) + cos(deg2rad(lat1)) * cos(deg2rad ...
                (lat2)) * sin(dLon/2) * sin(dLon/2);
            c = 2 * atan2(sqrt(a), sqrt(1-a));
            distance = earthRadius * c;

            % Initialize the condition factor to 1
            condition_factor = 1;

            % Find the index of the delivery location corresponding to the current
            % location
            delivery_index = find(ismember([delivery_latitudes, delivery_longitudes], ...
                locations(i, :), 'rows'), 1);

            if ~isempty(delivery_index)
                % Adjusting the condition factor based on weather conditions
                if data.Weather_conditions_Cloudy(delivery_index) == 1
                    condition_factor = condition_factor * 1.1;
                end
                if data.Weather_conditions_Fog(delivery_index) == 1
                    condition_factor = condition_factor * 1.3;
                end
                if data.Weather_conditions_Sandstorms(delivery_index) == 1
                    condition_factor = condition_factor * 1.4;
                end
                if data.Weather_conditions_Stormy(delivery_index) == 1
                    condition_factor = condition_factor * 1.5;
                end
                if data.Weather_conditions_Sunny(delivery_index) == 1
                    condition_factor = condition_factor * 0.9;
                end
                if data.Weather_conditions_Windy(delivery_index) == 1
                    condition_factor = condition_factor * 1.2;
                end
```

```
                % Adjust the condition factor based on traffic conditions
                if data.Road_traffic_density_Jam(delivery_index) == 1
                    condition_factor = condition_factor * 2.0;
                end
                if data.Road_traffic_density_High(delivery_index) == 1
                    condition_factor = condition_factor * 1.3;
                end
                if data.Road_traffic_density_Low(delivery_index) == 1
                    condition_factor = condition_factor * 0.8;
                end
                if data.Road_traffic_density_Medium(delivery_index) == 1
                    condition_factor = condition_factor * 1.1;
                end

                % Calculate the adjusted distance and time
                adjusted_distance = distance * condition_factor;
                distanceMatrix(i, j) = adjusted_distance;

                % Adjust time using the provided time_taken column
                adjusted_time = time_taken(delivery_index) * condition_factor;
                timeMatrix(i, j) = adjusted_time;
            else
                % If no corresponding delivery location is found, assign Inf to both
                % matrices
                distanceMatrix(i, j) = Inf;
                timeMatrix(i, j) = Inf;
            end
        else
            % If the locations are the same, assign Inf to both matrices
            distanceMatrix(i, j) = Inf;
            timeMatrix(i, j) = Inf;
        end
    end
end
```

5. Save Matrices and Visualize Paths

```
% Save the distance and time matrices to CSV files
writematrix(distanceMatrix, 'distance_matrix_adjusted.csv');
writematrix(timeMatrix, 'time_matrix_adjusted.csv');

% Visualize the paths using geoplot
figure;
geoaxes; % Create geoaxes
geoplot(restaurant_latitudes, restaurant_longitudes, 'r.', 'MarkerSize', 40); % Plot
% restaurant locations in red
hold on;
geoplot(delivery_latitudes, delivery_longitudes, 'g.', 'MarkerSize', 30); % Plot delivery
% locations in green

% Plot paths between locations
for i = 1:numLocations
    for j = 1:numLocations
        if distanceMatrix(i, j) < Inf % Only plot if there is a valid distance
```

```
            geoplot([locations(i, 1), locations(j, 1)], [locations(i, 2), ...
                locations(j, 2)], 'b-');
        end
    end
end
hold off;

% Add titles and labels
title('Possible Delivery Paths Visualization');

% Add text labels for each point
for i = 1:numel(restaurant_latitudes)
    text(restaurant_longitudes(i), restaurant_latitudes(i), sprintf('R%d', i), ...
        'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right', 'Color', 'red');
end

for i = 1:numel(delivery_latitudes)
    text(delivery_longitudes(i), delivery_latitudes(i), sprintf('D%d', i), ...
        'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right', 'Color', 'green');
end

% Add a legend and move it to the best location
legend({'Restaurants', 'Delivery Locations', 'Paths'}, 'Location', 'best');
```

Appendix E: Ant-Colony Optimization process

Defining the function

1. Initialization

```
function [bestRoute, bestDistance, bestTime] = aco(distanceMatrix, timeMatrix, numAnts, numIterations, alpha, beta, evaporationRate)
    % Number of nodes (locations)
    numNodes = size(distanceMatrix, 1);
    % Initialize pheromone matrix with ones
    pheromoneMatrix = ones(numNodes, numNodes);
    % Initialize variables to store the best route, distance, and time
    bestRoute = [];
    bestDistance = inf;
    bestTime = inf;
```

2. Main Loop for Iterations

```
    % Main loop for iterations
    for iter = 1:numIterations
        % Initialize arrays to store routes and their lengths/times for each ant
        allRoutes = zeros(numAnts, numNodes);
        allRouteLengths = zeros(numAnts, 1);
        allRouteTimes = zeros(numAnts, 1);
```

3. Loop Over Each Ant

```
        % Loop over each ant
        for ant = 1:numAnts
            % Initialize the visited nodes array
            visited = false(numNodes, 1);           % Keep track of visited nodes
            % Start from the restaurant node
            currentNode = 1;
            visited(currentNode) = true;

            % Initialize the route
            route = zeros(numNodes, 1);
            route(1) = currentNode;
```

4. Construct the Route for the Ant

```
            % Construct the route for the ant
            for step = 2:numNodes
                % Calculate probabilities for next node based on pheromone and distance
                prob = pheromoneMatrix(currentNode, :) .^ alpha .* (1 ./ distanceMatrix(currentNode, :)) .^ beta;
                prob(visited) = 0; % Set probabilities of visited nodes to zero
                prob = prob / sum(prob); % Normalize probabilities

                % Select the next node based on probabilities
                nextNode = find(rand <= cumsum(prob), 1);
                if isempty(nextNode)
                    % If no valid next node, randomly select from remaining nodes
                    remainingNodes = find(~visited);
                    nextNode = remainingNodes(randi(length(remainingNodes)));
                end

                % Update the route and visited nodes
                route(step) = nextNode;
                visited(nextNode) = true;
                currentNode = nextNode;
            end

            % Store the route and its length/time
            allRoutes(ant, :) = route;
            allRouteLengths(ant) = sum(distanceMatrix(sub2ind(size(distanceMatrix), route(1:end-1), route(2:end))));
            allRouteTimes(ant) = sum(timeMatrix(sub2ind(size(timeMatrix), route(1:end-1), route(2:end))));
        end
```

5. Update Pheromones

```
        % Update pheromones
        pheromoneMatrix = pheromoneMatrix * (1 - evaporationRate);
        % Update pheromones based on routes found by ants
        for ant = 1:numAnts
            for step = 1:(numNodes - 1)
                pheromoneMatrix(allRoutes(ant, step), allRoutes(ant, step + 1)) = ...
                    pheromoneMatrix(allRoutes(ant, step), allRoutes(ant, step + 1)) + 1 / allRouteLengths(ant);
            end
        end
```

6. Find the Best Route Based on Distance and Time

```
        % Find the best route of this iteration based on distance
        [minLength, minIndex] = min(allRouteLengths);
        if minLength < bestDistance
            bestDistance = minLength;
            bestRoute = allRoutes(minIndex, :);
        end

        % Find the best route of this iteration based on time
        [minTime, minIndex] = min(allRouteTimes);
        if minTime < bestTime
            bestTime = minTime;
            bestRoute = allRoutes(minIndex, :);
        end
    end
end
```

Calling the function and other processes

1. Load Data and Define ACO Parameters

```
% Load the matrices from CSV files
% These matrices represent the distances and times between delivery locations
distanceMatrix = readmatrix('distance_matrix_adjusted.csv');
timeMatrix = readmatrix('time_matrix_adjusted.csv');

% Define ACO parameters
numAnts = 50; % Number of ants used in the algorithm
numIterations = 200; % Number of iterations for the ACO algorithm
alpha = 1; % Influence of pheromone
beta = 2; % Influence of distance
evaporationRate = 0.5; % Rate at which pheromone evaporates

% Define number of runs
% We will run the ACO algorithm multiple times to find the best route
numRuns = 10;

% Initialize arrays to store results of multiple runs
allBestRoutes = cell(numRuns, 1);  % Store best routes for each run
allBestDistances = zeros(numRuns, 1);  % Store best distances for each run
allBestTimes = zeros(numRuns, 1);  % Store best times for each run
overallBestRoute = [];  % Store the overall best route found
overallBestDistance = inf;  % Initialize best distance as infinity
overallBestTime = inf;  % Initialize best time as infinity

% Fix the random seed for reproducibility
% This ensures that the results are consistent across multiple runs
rng(1);
```

2. Run ACO Multiple Times

```
% Run ACO multiple times
for run = 1:numRuns
    % Call the ACO function to find the best route, distance, and time
    [bestRoute, bestDistance, bestTime] = aco(distanceMatrix, timeMatrix, numAnts, numIterations, alpha, beta, evaporationRate);
    allBestRoutes{run} = bestRoute;
    allBestDistances(run) = bestDistance;
    allBestTimes(run) = bestTime;

    % Update the overall best route if a better route is found
    if bestDistance < overallBestDistance
        overallBestDistance = bestDistance;
        overallBestRoute = bestRoute;
    end

    % Update the overall best time if a better time is found
    if bestTime < overallBestTime
        overallBestTime = bestTime;
        overallBestRoute = bestRoute;
    end
end
```

3. Display Results

```
% Display results
disp('Results from multiple runs:');
for run = 1:numRuns
    disp(['Run ', num2str(run), ' - Best Distance: ', num2str(allBestDistances(run)), ', Best Time: ', num2str(allBestTimes(run))]);
end

% Display the overall best distance and time
disp(['Overall Best Distance: ', num2str(overallBestDistance)]);
disp(['Overall Best Time: ', num2str(overallBestTime)]);
disp('Overall Best Route:');
disp(overallBestRoute);
```

4. Calculate and Display Statistics

```
% Calculate statistics
meanDistance = mean(allBestDistances);
stdDistance = std(allBestDistances);
meanTime = mean(allBestTimes);
stdTime = std(allBestTimes);
disp(['Mean Distance: ', num2str(meanDistance)]);
disp(['Standard Deviation of Distances: ', num2str(stdDistance)]);
disp(['Mean Time: ', num2str(meanTime)]);
disp(['Standard Deviation of Times: ', num2str(stdTime)]);
```

5. Plot the Locations and Overall Best Route

```
% Create a larger figure
figure('Position', [100, 100, 1000, 800]);  % Increase figure size
hold on;

% Plot all delivery locations
scatter(delivery_longitudes, delivery_latitudes, 50, 'filled', 'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'r');

% Highlight the starting location
scatter(delivery_longitudes(1), delivery_latitudes(1), 100, 'filled', 'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'g');

% Label all delivery locations
text(delivery_longitudes, delivery_latitudes, num2str((1:numLocations)'), 'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right');

% Plot the route with arrows indicating direction
for i = 1:length(overallBestRoute)
    from = overallBestRoute(i);
    to = overallBestRoute(mod(i, length(overallBestRoute)) + 1);
    if from <= numLocations && to <= numLocations
        quiver(delivery_longitudes(from), delivery_latitudes(from), ...
               delivery_longitudes(to) - delivery_longitudes(from), ...
               delivery_latitudes(to) - delivery_latitudes(from), ...
               0, 'b', 'LineWidth', 1, 'MaxHeadSize', 0.1, 'AutoScale', 'off');
    end
end
```

5. Plot the Locations and Overall Best Route

```
% Create a larger figure
figure('Position', [100, 100, 1000, 800]);  % Increase figure size
hold on;

% Plot all delivery locations
scatter(delivery_longitudes, delivery_latitudes, 50, 'filled', 'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'r');

% Highlight the starting location
scatter(delivery_longitudes(1), delivery_latitudes(1), 100, 'filled', 'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'g');

% Label all delivery locations
text(delivery_longitudes, delivery_latitudes, num2str((1:numLocations)'), 'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right');

% Plot the route with arrows indicating direction
for i = 1:length(overallBestRoute)
    from = overallBestRoute(i);
    to = overallBestRoute(mod(i, length(overallBestRoute)) + 1);
    if from <= numLocations && to <= numLocations
        quiver(delivery_longitudes(from), delivery_latitudes(from), ...
               delivery_longitudes(to) - delivery_longitudes(from), ...
               delivery_latitudes(to) - delivery_latitudes(from), ...
               0, 'b', 'LineWidth', 1, 'MaxHeadSize', 0.1, 'AutoScale', 'off');
    end
end

% Enhance the plot
xlabel('Longitude');
ylabel('Latitude');
title('Overall Best Delivery Route');
grid on;
```

```
% Extend axis limits slightly
ax = gca;
xlim([min(delivery_longitudes) - 0.01, max(delivery_longitudes) + 0.05]);
ylim([min(delivery_latitudes) - 0.01, max(delivery_latitudes) + 0.01]);

% Add legend indicators
legend_x = max(delivery_longitudes) + 0.02;
legend_y = linspace(max(delivery_latitudes), min(delivery_latitudes), 3);

scatter(legend_x, legend_y(1), 100, 'filled', 'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'g');
text(legend_x + 0.005, legend_y(1), 'Restaurant', 'VerticalAlignment', 'middle');

scatter(legend_x, legend_y(2), 50, 'filled', 'MarkerEdgeColor', 'k', 'MarkerFaceColor', 'r');
text(legend_x + 0.005, legend_y(2), 'Delivery Locations', 'VerticalAlignment', 'middle');

hold off;
```