Route Optimization for Meals on Wheels

By Group 1:

Edda Phillips
Gaukhar Makanova
Jigna Chaudhary
Tharangini Narasimha Guptha

May 16, 2025

BANA 6670 Prescriptive Analytics

Professor Meysam Rabiee

Background

Meals on Wheels (MOW) Downtown Denver delivers daily meals to homebound residents across the city, with the support of a large volunteer network. These volunteers play a critical role, not only delivering meals but also providing important social contact for many clients. However, downtown Denver’s dense urban environment creates significant challenges - especially related to parking, walking distances, and time-sensitive delivery of hot and frozen meals.

As the number of clients increases and delivery routes become more complex, MOW faces growing inefficiencies. Volunteers often encounter limited parking availability, long walks between their vehicles and client homes, and complex navigation across multi-stop routes. These issues lead to delivery delays, volunteer fatigue, and rising operational costs.

Given these challenges, our project focuses specifically on optimizing volunteer-based delivery routes by improving parking assignments and walking paths in downtown Denver. Our goal is to reduce time spent searching for parking, minimize volunteer walking distance, and improve delivery efficiency for the MOW program.

Problem Statement

Despite a well-established delivery system, Meals on Wheels Downtown Denver faces persistent inefficiencies in volunteer-based meal distribution due to parking and routing constraints:

- ➢ Volunteers often park far from client clusters, increasing delivery time and effort.
- ➢ Some parking areas are overused, while others are underutilized, leading to congestion and ticketing.
- ➢ Delivery routes do not consistently account for walking distance between parking spots and client addresses.
- ➢ Time-sensitive meals (hot and frozen) require faster, more optimized handoffs.
- ➢ Volunteer turnover further complicates route planning and parking management.

These issues result in higher parking costs, inconsistent delivery timing, and increased difficulty in sustaining long-term volunteer participation. Without a structured approach to optimizing parking clusters and walking paths, MOW risks losing delivery efficiency and volunteer engagement.

Project Motivation

The motivation for this project stems from the need to support volunteers - who are the backbone of MOW's delivery system - and to improve the overall client experience. Our goal is to design a solution that:

➢ Assigns optimal parking clusters to volunteers, reducing walking time and cost

➢ Minimizes travel time from parking to doorsteps to ensure timely delivery of meals

➢ Maximizes the number of clients reached per route, especially in high-density areas

By using volunteer location data, client addresses, meal type schedules, and known parking availability, we aim to build a data-driven route and parking optimization model. This model can be implemented using prescriptive analytics tools such as linear programming or route clustering algorithms in Python / Excel / GAMS Solver.

Ultimately, optimizing parking for volunteer routes will lead to a more efficient, sustainable delivery system that improves service for clients and reduces friction for the dedicated individuals who deliver meals each day.

Model Development Methodology

We used GAMS and Python PULP to model this problem. Below is the GAMS model (Please note: The GAMS model was re-created for this assignment with 16 parking spots only since the demo version only allowed for fewer columns. The purpose of GAMS model recreation is to verify the model created in Python PULP later.)

Indices

$i \in \{1, ..., N\}$	Index for individuals (or demand points)
$j \in \{1, ..., M\}$	Index for parking spots (or supply points)

Parameters

Symbol	**Indices**	**Type**	**Description**
N	–	Integer	Total number of individuals (demand points).

M	–	Integer	Total number of parking spots (supply points).
C[j]	j	Numeric	Cost to assign one individual to parking spot j.
W[i][j]	i, j	Numeric	Distance between individual i and parking spot j.
Demand[i]	i	Integer	Demand level of individual i.
max_distance	–	Numeric	Maximum allowed walking distance (standard).
max_distance_hd	–	Numeric	Max allowed distance for high-demand individuals (Demand[i] > 5).
penalty_weight	–	Numeric	Cost multiplier for exceeding allowed distance.
reuse_penalty	–	Numeric	Cost incurred for using a parking spot (to discourage overuse).

Decision Variables

Symbol	**Indices**	**Type**	**Description**
X[i][j]	i, j	Binary	1 if individual i is assigned to parking spot j, 0 otherwise.
P[i]	i	Continuous ≥ 0	Penalty slack: extent to which distance limit is exceeded by individual i.
Y[j]	j	Binary	1 if parking spot j is used (assigned to any individual), 0 otherwise.

Clustering of routes

After obtaining the optimal parking assignments through the optimization model, we further organized the delivery logistics by grouping the assigned locations into four efficient delivery routes. To achieve this, we applied K-Means clustering on the geographical coordinates (latitude and longitude) of the selected parking spots. This unsupervised machine learning technique allowed us to cluster spatially proximate locations, ensuring that each route is geographically coherent. By minimizing intra-cluster distance, the resulting routes are expected to reduce travel time and improve operational efficiency. This step enhances the practicality of the model by aligning it with real-world routing considerations.

Route Optimization using TSP

Decision Variables

$$x_{uv} = \begin{cases} 1, & \text{if the route travels directly from point } u \text{ to point } v \\ 0, & \text{otherwise} \end{cases} \qquad \forall u, v \in N, u \neq v$$

Objective Function

Minimize the total travel distance:

$$\min \sum_{u \in N} \sum_{\substack{v \in N \\ v \neq u}} d_{uv} \cdot x_{uv}$$

Sets and Indices

- Let $N = \{0, 1, 2, \ldots, n-1\}$ be the set of points (deliveries) in the route.
- Indices $u, v \in N$, with $u \neq v$.

Parameters

- d_{uv}: distance between point u and point v, calculated from their coordinates.

Constraints

1. **Depart from each point exactly once:**

$$\sum_{\substack{v \in N \\ v \neq u}} x_{uv} = 1 \quad \forall u \in N$$

2. **Arrive at each point exactly once:**

$$\sum_{\substack{u \in N \\ u \neq v}} x_{uv} = 1 \quad \forall v \in N$$

Sensitivity Analysis Report for Delivery Assignment

To evaluate the impact of varying key model parameters like walking distance limits, penalty weights, and parking reuse penalty (cost)on the total optimization cost and feasibility (solver status) in a parking-to-delivery assignment model.

Parameters Analyzed

- ➢ Walking Distance Limit (Walking Distance Limit): Maximum allowed walking distance from a parking spot to a delivery location.
- ➢ Penalty Weight (Penalty Weight): Multiplier applied to the penalty for exceeding the walking distance threshold.
- ➢ Parking Reuse Penalty (Cost): Proxy for the additional cost of assigning more unique parking spots.

Summary of Results

A subset of outcomes is shown below – the sensitivity analysis of all other variations are included in the supplementary files folder.

Cost	Walking Distance Limit	Penalty Weight	Total Cost	Solver Status
1	0.3 km	500	284.32	Optimal
1	0.3 km	1000	530.14	Optimal

Observations:

- ➢ Increasing the penalty weight while holding other variables constant (walking limit = 0.3) significantly increases the total cost.
- ➢ Increasing the walking distance limit leads to lower total cost, as it allows more flexibility in assignments.
- ➢ Solver consistently returned "Optimal" status across combinations, indicating no infeasibility was encountered.

Insights:

- ➢ Tight walking distance constraints greatly increase costs, and the cost sensitivity to penalty weight becomes exaggerated.

➢ Loosening the walking distance constraint allows for cost minimization and reduces the impact of higher penalty weights.

➢ There is a diminishing return in total cost reduction beyond a certain distance limit (especially after 0.6), where further increases in limit do not substantially reduce cost.

Recommendations:

➢ Optimal Zone: Distance limits between 0.6 and 0.8 and penalty weights around 1000–1500 provide a good trade-off between minimizing total cost and maintaining control over penalty effects.

➢ Avoid overly restrictive walking distances (e.g., 0.3) unless strictly necessary due to significant cost implications.

➢ For stricter policies (low distance limits), invest effort in optimizing penalty weights, as small changes lead to substantial cost differences.

Results

Objective Value (lowest cost/penalty of distances): 143

- Assignment cost $= 33$
- Penalty component $\sum P_i = 0$, and $\text{penalty\_weight} = w$ (any value × 0 = 0, so it disappears)
- Number of parking spots used $\sum Y_j = 11$
- Reuse penalty = 10

> **Total cost = Assignment Cost (33) + Penalty Cost (0) + Reuse Cost (10 × 11 = 110) = 143**

$$\text{Minimize: } 33 + 0 + 10 \times 11 = \boxed{143}$$

Key Achievements of the Optimization Model

1. Reduced Costs

The optimization model strategically identified low-cost parking spots that are conveniently located near delivery clusters. This careful selection minimized unnecessary walking distances and effectively reduced parking expenses. The calculated optimization prevented costly parking fines, leading to a substantial saving of approximately $14.50 per ticket.

2. Zone-Based Delivery

The delivery locations were intelligently divided into efficient zones, serviced by 11 strategically selected parking spots. This structured planning enabled 73 Saturday clients to be served with the help of just 4 volunteers, ensuring that each volunteer made no more than 4 parking stops. This method maximized coverage and minimized the time and resources required for deliveries.

3. Reduced Route Overlap

The optimization approach successfully prevented route overlap by clustering delivery assignments efficiently. Each zone was uniquely mapped to its respective deliveries, eliminating redundant travel paths. This smart routing led to more streamlined travel paths, conserving both time and volunteer effort.

4. Reduced Walking Distances

One of the standout achievements of the optimization was the substantial reduction in average walking distances for volunteers. The analysis successfully brought down the average distance from 200 meters to just 150 meters per delivery. This not only enhanced the convenience for volunteers but also improved the speed and efficiency of meal delivery.

Limitations

While the optimization analysis provided meaningful improvements in cost savings and reduced walking distances, several limitations should be acknowledged:

1. Data Quality and Assumptions

The analysis depended heavily on the accuracy and completeness of the input datasets, including parking locations, pricing, and delivery addresses. Inconsistencies or omissions in these datasets could compromise the reliability of the model's results. Additionally, the analysis assumed that parking spots would be available as planned, which may not hold true during peak hours or local events. Parking fees were sourced from platforms like SpotHero and Parkopedia, which often differ from on-site pricing and may reflect limited-time discounts not guaranteed at the time of use.

2. Static Modeling Conditions

The optimization models (Mixed Integer Linear Programming [MILP] and Goal Programming) treated parking availability and costs as fixed inputs, even though these values fluctuate in real-world conditions. Likewise, walking distances were calculated using straight-line (Euclidean) measures, which do not account for detours, inaccessible paths, or urban barriers. While estimated walking times were retrieved using the Google Maps API, they still depend on starting points and real-time conditions, introducing further variability.

3. Volunteer Scheduling Constraints

The models did not incorporate the dynamic nature of volunteer participation. The model did not include factors such as availability, capacity to carry meals, or unforeseen cancellations. Additionally, our analysis could not provide total delivery times due to individual differences in walking speed, package handling, and client response times. As such, route assignments were not optimized based on volunteer availability or specific delivery time windows.

4. Lack of Real-Time Variables

The models did not integrate key real-world variables, including traffic conditions, temporary road closures, and weather disruptions. These factors can significantly impact routing efficiency and delivery timing. Although management expressed interest in incorporating smarter routing (beyond what tools like Google Maps offer), time and resource constraints prevented us from integrating such features into the current model. For instance, while we aimed to minimize turns on one-way streets, this functionality proved too complex to implement within the existing framework.

5. Limited Scope of Analysis

The model was tested using a limited dataset focused on Saturday-only deliveries, which does not capture the full variability of Meals on Wheels operations. In practice, delivery schedules, meal types, and vehicle capacities vary by day. For example, on some days, only official Meals on Wheels vans are used for deliveries and some addresses only get one or two deliveries a month. To fully accommodate these variations, the model would need to be rerun for each delivery variation and adjusted based on vehicle and volunteer availability.

Managerial Recommendations and Insights

To build on the findings of this analysis and further enhance the Meals on Wheels delivery system, several recommendations are proposed:

1. Integrate Real-Time Route Optimization Tools

Combining existing MILP and K-Means clustering models with real-time route optimization software could significantly reduce travel times and parking costs. Solutions such as Routific offer dynamic routing capabilities that respond to live traffic conditions, construction, and road closures, an improvement over static tools like Google Maps. These platforms typically operate on a subscription model and would directly support management’s goal of providing up-to-date navigation for volunteers in the field.

2. Consolidate Volunteer Scheduling and Routing Software

Currently, Meals on Wheels uses multiple tools, including SignUp.com and ServTracker, to manage volunteer schedules and meal deliveries. While each tool has value, using a unified platform could streamline scheduling, reduce data entry errors, and improve operational efficiency. Many route optimization tools offer integrated scheduling, real-time updates, and mobile check-ins, potentially replacing these disparate systems.

3. Secure Parking Reservations in Advance

To ensure parking prices and availability align with those modeled, staff should use apps like SpotHero or Parkopedia to reserve parking at least a day in advance. Cheaper lots often fill quickly, particularly during local events. Some facilities provide QR code entry, while others require vehicle registration, which can be problematic if volunteer assignments change last-minute. As a workaround, volunteers could be given a list of nearby parking options and instructed to reserve a spot when they pick up their meals, but this carries the risk of lots being full. In any case, management should have a clear process in place to reimburse volunteers for parking expenses.

4. Improve Data Collection and Standardization

Accurate optimization depends on clean and consistent data. In our analysis, address formatting inconsistencies (e.g., “433 N Wilson Street” vs. “433 N. Wilson St.”) led the model to interpret duplicate addresses as separate locations. Enhancing address standardization, capturing up-to-date parking cost data, and maintaining real-time volunteer schedules would greatly improve model performance and decision-making.

5. Conduct Route Validation Trials

A practical trial run of each route would help verify that suggested parking locations and addresses are accurate and logistically feasible. For routes with planned street parking, it is advisable to assess availability in advance and, where needed, identify backup parking options, especially in high-traffic areas.

6. Explore Parking Leniency Options with Local Authorities

Before committing to new software or rerouting strategies, Meals on Wheels could explore partnerships with local law enforcement. Some delivery services, like Amazon or FedEx, have established agreements to reduce ticketing for vehicles in active delivery. While not guaranteed, a similar arrangement might be possible for a nonprofit with a strong community mission. If not, understanding the current cost of parking tickets could help determine whether budgeting for fines is a viable, though less ideal, alternative. Unfortunately, we were unable to access data on how much Meals on Wheels currently spends on parking violations.

Appendix

References

Model Files and Data:

Resources:https://drive.google.com/file/d/1M4CLMe4kYSWcLqv-xWL3sPfsKOiX3z0K/view?usp=share_link

Method 1

Using PuLP - Python for assignment of deliveries to the best parking spots available, KMeans clustering to obtain 4 best routes and TSP Optimization within each route.

```
import pandas as pd
import googlemaps
import openrouteservice
from geopy.geocoders import Nominatim
import numpy as np
import folium
import time
API_KEY_GOOGLE = ''
API_KEY_ORS = ''
# === Step 1: Geocode parking addresses ===
parking_df = pd.read_csv('parking.csv')
if 'parking_address' not in parking_df.columns:
   raise ValueError("Column 'parking_address' not found in parking.csv")

parking_addresses =
parking_df['parking_address'].dropna().astype(str).str.strip().tolist()

# Initialize geocoding services

gmaps = googlemaps.Client(key=API_KEY_GOOGLE)
geolocator = Nominatim(user_agent="meal_delivery_routing")

# Geocode parking addresses
parking_coordinates = []
for address in parking_addresses:
   geocode_result = gmaps.geocode(address)
   if geocode_result:
       lat_lng = geocode_result[0]['geometry']['location']
       parking_coordinates.append((lat_lng['lng'], lat_lng['lat']))  # (lon, lat)
   else:
       parking_coordinates.append(None)
   time.sleep(1)
```

```
if None in parking_coordinates:
   raise ValueError("Some parking addresses could not be geocoded. Please check
them.")

print(f"Successfully geocoded {len(parking_coordinates)} parking addresses.")

# === Step 2: Geocode delivery addresses ===
delivery_df = pd.read_csv('deliveries_cap.csv')
if 'address' not in delivery_df.columns:
   raise ValueError("Column 'address' not found in deliveries.csv")

delivery_addresses = delivery_df['address'].dropna().astype(str).str.strip().tolist()

# Geocode delivery addresses
delivery_coordinates = []
for address in delivery_addresses:
   geocode_result = gmaps.geocode(address)
   if geocode_result:
       lat_lng = geocode_result[0]['geometry']['location']
       delivery_coordinates.append((lat_lng['lng'], lat_lng['lat']))  # (lon, lat)
   else:
       delivery_coordinates.append(None)
   time.sleep(1)

if None in delivery_coordinates:
   raise ValueError("Some delivery addresses could not be geocoded. Please check
them.")

print(f"Successfully geocoded {len(delivery_coordinates)} delivery addresses.")

# === Step 3: Calculate the distance matrix between parking and delivery locations ===

client = openrouteservice.Client(key=API_KEY_ORS)

# Create a function to calculate distances
def create_distance_matrix_ors(parking_coords, delivery_coords):
   all_coords = parking_coords + delivery_coords
   sources = list(range(len(parking_coords)))  # indices for parking
   destinations = list(range(len(parking_coords), len(all_coords)))  # indices for
delivery

   matrix_response = client.distance_matrix(
       locations=all_coords,
       profile='driving-car',  # or 'foot-walking' if needed
       metrics=['distance'],
       units='m',
       sources=sources,
       destinations=destinations
   )
   distances = np.array(matrix_response['distances'])  # shape: [#parking x
#deliveries]
```

```
   return distances


# === Input data ===
delivery_meals = list(parking_df['price'])
# Columns: Address, Meals
print(len(parking_coordinates))  # should be 84
print(len(delivery_coordinates))  # should be 23
delivery_df.count()['address']
parking_df.count()['parking_address']

# === Get distance matrix ===
parking_to_delivery_distances = create_distance_matrix_ors(parking_coordinates,
delivery_coordinates)
parking_to_delivery_distances.shape
# === Confirm dimensions ===
print("Distance matrix shape:", parking_to_delivery_distances.shape)
######
import pulp
import pandas as pd
import math
parking_coord = parking_coordinates
delivery_coord = delivery_coordinates
parking_coord

from haversine import haversine

def to_latlon(coord):
       return (coord[1], coord[0])  # Convert to (lat, lon)




walking_distances = []

for delivery in delivery_coord:
   row = []
   for parking in parking_coord:
       d = haversine(to_latlon(delivery), to_latlon(parking))  # in km
       row.append(d)
   walking_distances.append(row)

import pulp



# Define the problem
prob = pulp.LpProblem("Delivery_Assignment_With_Parking_Reuse", pulp.LpMinimize)

# N = number of deliveries, M = number of parking spots
# C[j]: Cost of using parking spot j
# W[i][j]: Walking distance from parking j to delivery i
# Demand[i]: Demand at delivery i (used if adding extra walking constraint)
# -------------
# These must be already defined before this block
# -------------
N = delivery_df.shape[0]
M = parking_df.shape[0]
C = parking_df['price'].to_list()
W = walking_distances
Demand = delivery_df['demand'].to_list()
# Decision variables
X = pulp.LpVariable.dicts("X", (range(N), range(M)), cat="Binary")  # Assignment
P = pulp.LpVariable.dicts("P", range(N), lowBound=0, cat="Continuous")  # Penalty
Y = pulp.LpVariable.dicts("Y", range(M), cat="Binary")  # Whether parking j is used

# Weights
cost_weight = 1
penalty_weight = 1000
parking_reuse_penalty = 10  # Adjust to control how heavily to discourage new parking
spots

# Objective: minimize cost + penalty + number of parking spots used
prob += pulp.lpSum([
   cost_weight * C[j] * X[i][j] for i in range(N) for j in range(M)
]) + pulp.lpSum([
   penalty_weight * P[i] for i in range(N)
]) + pulp.lpSum([
   parking_reuse_penalty * Y[j] for j in range(M)
])

# Constraint: Each delivery assigned to exactly one parking spot
for i in range(N):
   prob += pulp.lpSum([X[i][j] for j in range(M)]) == 1

# Constraint: Walking distance + penalty must be within allowed threshold
for i in range(N):
   for j in range(M):
       prob += W[i][j] * X[i][j] <= 0.6 + P[i]

# Optional constraint for high-demand deliveries (more strict distance)
for i in range(N):
   if Demand[i] > 5:
       for j in range(M):
           prob += W[i][j] * X[i][j] <= 0.3 + P[i]

# Link parking spot use to delivery assignment
for i in range(N):
   for j in range(M):
       prob += X[i][j] <= Y[j]

# Solve the problem
prob.solve()
total_cost = pulp.value(prob.objective)
total_cost
   #atus} for cost={cost}, distance_limit={distance_limit},
penalty_weight={penalty_weight}")
```

```python
total_penalty = sum(P[i].varValue for i in P if P[i].varValue is not None and
P[i].varValue > 0)
print("Total Penalty Component (Σπᵢ):", total_penalty)

total_parking_spots_used = sum(Y[j].varValue for j in Y if Y[j].varValue is not None
and Y[j].varValue > 0)
print("Total Number of Parking Spots Used (Σy□):", total_parking_spots_used)

# Extract results
assignments = []
for i in range(N):
   for j in range(M):
       if pulp.value(X[i][j]) == 1:
           assignments.append({
               'Delivery': i,
               'Parking': j,
               'Walking_Distance': W[i][j],
               'Parking_Cost': C[j]
           })



# Optional: See which parking spots were used
used_parking_spots = [j for j in range(M) if pulp.value(Y[j]) == 1]
print(f"Number of unique parking spots used: {len(used_parking_spots)}")
assignments_df = pd.DataFrame(assignments)
assignments_df['Delivery_Address'] = assignments_df['Delivery'].apply(lambda x:
delivery_df.iloc[x]['address'])
assignments_df['Parking_Address'] = assignments_df['Parking'].apply(lambda x:
parking_df.iloc[x]['parking_address'])
assignments_df['Demand'] = assignments_df['Delivery'].apply(lambda x:
delivery_df.iloc[x]['demand'])
assignments_df['Walking_Time'] = (assignments_df['Walking_Distance'] / (5/60))
assignments_df.to_csv("assignments_updated_parking_pulp_new_MILP_8thmay_final.csv")




import pulp
import numpy as np

from geopy.geocoders import Nominatim
from geopy.extra.rate_limiter import RateLimiter
import pandas as pd
from sklearn.cluster import KMeans
# Optionally, store the assigned deliveries
geolocator = Nominatim(user_agent="route_balancer")
geocode = RateLimiter(geolocator.geocode, min_delay_seconds=1)

# Make sure your 'Parking_Address' is a string
assignments_df['Parking_Address'] = assignments_df['Parking_Address'].astype(str)

# Geocode parking addresses
assignments_df['location'] = assignments_df['Parking_Address'].apply(geocode)
```

```
# Extract lat/lon
assignments_df['Latitude'] = assignments_df['location'].apply(lambda loc: loc.latitude
if loc else None)
assignments_df['Longitude'] = assignments_df['location'].apply(lambda loc:
loc.longitude if loc else None)

# Drop rows where geocoding failed
assignments_df = assignments_df.dropna(subset=['Latitude', 'Longitude'])

# Display the final DataFrame with route assignments

#nt his function in solved TSP, please make sure every route begins. at HUB and ends
at assigned last stop,
routes_df=pd.DataFrame()
from ortools.linear_solver import pywraplp
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import geopy.distance

# Example data: Replace these with actual data
num_deliveries = len(assignments_df)
num_routes = 4

# Extract parking coordinates from your dataframe
parking_coords = assignments_df[['Latitude', 'Longitude']].values

# Step 1: Cluster parking spots geographically using KMeans
kmeans = KMeans(n_clusters=num_routes, random_state=42)
assignments_df['Route'] = kmeans.fit_predict(parking_coords)

# After clustering, we now have the route for each delivery based on geographical
proximity of parking spots.

# Step 2: Solve TSP for each route to minimize the total travel distance within the
route
def calculate_distance_matrix(coords):
   # Calculate the pairwise distance matrix using geopy's distance module
   num_points = len(coords)
   distance_matrix = np.zeros((num_points, num_points))

   for i in range(num_points):
       for j in range(i+1, num_points):
           dist = geopy.distance.distance(coords[i], coords[j]).km  # Get distance in
km
           distance_matrix[i, j] = dist
           distance_matrix[j, i] = dist  # Distance is symmetric

   return distance_matrix
```

```python
def solve_tsp(distance_matrix):
   # Use the OR-Tools solver for TSP
   num_points = len(distance_matrix)
   solver = pywraplp.Solver.CreateSolver('SCIP')

   # Decision variable: x[i][j] = 1 if we travel from i to j
   x = {}
   for i in range(num_points):
       for j in range(num_points):
           if i != j:
               x[i, j] = solver.BoolVar(f'x_{i}_{j}')

   # Objective: Minimize total distance
   objective = solver.Objective()
   for i in range(num_points):
       for j in range(num_points):
           if i != j:
               objective.SetCoefficient(x[i, j], distance_matrix[i, j])

   objective.SetMinimization()

   # Constraints: Each point must be visited exactly once
   for i in range(num_points):
       solver.Add(sum(x[i, j] for j in range(num_points) if i != j) == 1)
       solver.Add(sum(x[j, i] for j in range(num_points) if i != j) == 1)

   # Solve the problem
   status = solver.Solve()

   if status == pywraplp.Solver.OPTIMAL:
       route = []
       for i in range(num_points):
           for j in range(num_points):
               if i != j and x[i, j].solution_value() == 1:
                   route.append((i, j))
       return route
   else:
       return None

# Step 3: Apply TSP to each route (cluster) and assign the order of deliveries
routes_data = []
for route_number in range(num_routes):
   # Get the parking spots assigned to the current route
   route_deliveries = assignments_df[assignments_df['Route'] == route_number]
   route_coords = route_deliveries[['Latitude', 'Longitude']].values

   # Calculate the distance matrix for the route
   distance_matrix = calculate_distance_matrix(route_coords)

   # Solve TSP for this route
   tsp_route = solve_tsp(distance_matrix)

   # If TSP solution exists, assign the order of deliveries within the route
   if tsp_route is not None:
       route_order = [delivery[0] for delivery in tsp_route]  # Get the order of
deliveries in the route
       route_deliveries['TSP_Order'] = [route_order.index(i) for i in
range(len(route_order))]

       # Append the result to the routes data
       routes_data.append(route_deliveries)

# Step 4: Combine the results for all routes
routes_df = pd.concat(routes_data)

# Save the results to a CSV file
routes_df.to_csv('route_details_with_tsp_8mayfinal_4R.csv', index=False)

print(f"Data has been saved to 'route_details_with_tsp_8may.csv'. Number of rows:
{len(routes_df)}")
#####rpute suing kmeans ends




#################assignmentsparking - folium map#####

import googlemaps
import folium
import pandas as pd

# Set your Google Maps API key here
gmaps = googlemaps.Client(key=API_KEY_GOOGLE)  # Replace with your key

# HUB coordinates (fixed for all routes)
HUB_ADDRESS = '4915 E 52nd Ave, Denver'
HUB_COORDS = (39.7796, -104.9288)

# Prepare folium map centered around Denver
m = folium.Map(location=HUB_COORDS, zoom_start=11)

# Color palette for different routes
colors = ['red', 'blue', 'green', 'orange', 'purple', 'darkred', 'cadetblue']

# Map address to coordinates
loc_df = routes_df.drop_duplicates('Parking_Address')[['Parking_Address', 'Latitude',
'Longitude']]
loc_df = loc_df.set_index('Parking_Address')
loc_df.loc[HUB_ADDRESS] = HUB_COORDS

# Group addresses by route
routes = routes_df.groupby('Route')['Parking_Address'].apply(list).to_dict()

# Store route summaries
route_summaries = []
```

```
for i, (route_id, stops) in enumerate(routes.items()):
   # Ensure HUB is at the start
   unique_stops = [addr for addr in stops if addr != HUB_ADDRESS]
   full_stops = [HUB_ADDRESS] + unique_stops

   # Get coordinates for each stop
   stop_coords = [(loc_df.loc[addr][0], loc_df.loc[addr][1]) for addr in full_stops]

   # Prepare directions API call
   waypoints = stop_coords[1:-1] if len(stop_coords) > 2 else None
   directions_result = gmaps.directions(
       origin=stop_coords[0],
       destination=stop_coords[-1],
       waypoints=waypoints,
       mode="driving"
   )

   if directions_result:
       route = directions_result[0]
       overview_polyline = route['overview_polyline']['points']
       decoded_polyline = googlemaps.convert.decode_polyline(overview_polyline)
       total_distance_meters = sum(leg['distance']['value'] for leg in route['legs'])
       total_distance_km = round(total_distance_meters / 1000, 2)

       # Add route polyline
       folium.PolyLine(
           [(point['lat'], point['lng']) for point in decoded_polyline],
           color=colors[i % len(colors)],
           weight=4,
           opacity=0.7,
           tooltip=f"Route {route_id} - {total_distance_km} km"
       ).add_to(m)

       # Add markers for each parking stop
       for idx, addr in enumerate(full_stops):
           lat, lon = loc_df.loc[addr]
           demand = 0 if addr == HUB_ADDRESS else
int(routes_df[routes_df['Parking_Address'] == addr]['Demand'].sum())

           folium.Marker(
               location=[lat, lon],
               tooltip=f"Stop {idx+1}: {addr} | Deliveries: {demand}",
               icon=folium.Icon(color=colors[i % len(colors)], icon="truck",
prefix="fa")
           ).add_to(m)

           # Route summary
           route_summaries.append({
               'RouteID': route_id,
               'StopNumber': idx + 1,
               'Address': addr,
               'Latitude': lat,
               'Longitude': lon,
               'DeliveriesHandled': demand,
               'TotalDistance_km': total_distance_km
           })
   else:
       print(f"Could not retrieve directions for Route {route_id}")
assignments_df.columns
#  NEW: Connect delivery locations to assigned parking spots
for idx, row in assignments_df.iterrows():
   delivery_idx = int(row['Delivery'])  # Ensure integer index
   parking_addr = row['Parking_Address']

   # Handle invalid delivery index
   if delivery_idx >= len(delivery_coordinates):
       print(f"Invalid delivery index: {delivery_idx}")
       continue

   # Get delivery coordinates (lon, lat) → folium expects (lat, lon)
   delivery_lon, delivery_lat = delivery_coordinates[delivery_idx]

   # Get parking coordinates from loc_df (since you already have this dataframe)
   if parking_addr not in loc_df.index:
       print(f"Parking address not found in loc_df: {parking_addr}")
       continue

   parking_lat, parking_lon = loc_df.loc[parking_addr]  # Fetch parking coordinates
from loc_df

   # Draw dashed line from parking to delivery
   folium.PolyLine(
       locations=[(parking_lat, parking_lon), (delivery_lat, delivery_lon)],
       color='gray',
       dash_array='5',
       weight=2,
       opacity=0.6,
       tooltip=f"Parking → Delivery\n{parking_addr} → #{delivery_idx}"
   ).add_to(m)

   # Place marker at delivery location
   folium.Marker(
       location=[delivery_lat, delivery_lon],
       tooltip=f"Delivery #{delivery_idx}",
       icon=folium.Icon(color='lightgray', icon='home', prefix='fa')
   ).add_to(m)



# Save the map and route summaries
m.save("gmaps_driving_routes_map_with_deliveries_final.html")
print(" Map saved as 'gmaps_driving_routes_map_with_deliveries.html'")

pd.DataFrame(route_summaries).to_csv("gmaps_route_summary_with_deliveries.csv",
index=False)
print("Route summary saved as 'gmaps_route_summary_with_deliveries.csv'")
```

```
delivery_coordinates


print(assignments_df['Delivery'].head())
print(delivery_coordinates[:5])
print(type(delivery_coordinates))




###sensitivity analysis###
import matplotlib.pyplot as plt
import seaborn as sns

# Function to perform sensitivity analysis
def sensitivity_analysis(parking_costs, walking_distance_limit, penalty_weight_range):
   results = []

   for cost in parking_costs:
       for distance_limit in walking_distance_limit:
           for penalty_weight in penalty_weight_range:
               # Re-run the optimization with the current parameters
               # Define the problem again with current cost, distance limit, and
penalty weight
               prob = pulp.LpProblem("Delivery_Assignment_With_Parking_Reuse",
pulp.LpMinimize)

               # Objective function with updated parameters
               prob += pulp.lpSum([
                   cost * C[j] * X[i][j] for i in range(N) for j in range(M)
               ]) + pulp.lpSum([
                   penalty_weight * P[i] for i in range(N)
               ]) + pulp.lpSum([
                   parking_reuse_penalty * Y[j] for j in range(M)
               ])

               # Constraints (same as in your existing code)
               # Each delivery assigned to exactly one parking spot
               for i in range(N):
                   prob += pulp.lpSum([X[i][j] for j in range(M)]) == 1

               # Walking distance + penalty within allowed threshold
               for i in range(N):
                   for j in range(M):
                       prob += W[i][j] * X[i][j] <= distance_limit + P[i]

               # Solve the problem
               prob.solve()

               # Extract the results
               total_cost = pulp.value(prob.objective)
               used_parking_spots = [j for j in range(M) if pulp.value(Y[j]) == 1]
```

```
                results.append({
                    'Cost': cost,
                    'Walking Distance Limit': distance_limit,
                    'Penalty Weight': penalty_weight,
                    'Total Cost': total_cost,
                    'Used Parking Spots': len(used_parking_spots)
                })

    # Convert results into DataFrame
    results_df = pd.DataFrame(results)
    return results_df

# Define the range of parameters for sensitivity analysis
parking_costs = [1, 2, 5, 10, 20]  # Vary parking costs
walking_distance_limit = [0.3, 0.5, 0.6, 0.8]  # Vary walking distance limit
penalty_weight_range = [500, 1000, 1500, 2000]  # Vary penalty weight

# Run the sensitivity analysis
results_df = sensitivity_analysis(parking_costs, walking_distance_limit,
penalty_weight_range)
results_df
# === Plotting ===
sns.set(style="whitegrid")

# Plot 1: Total cost vs Parking Cost
plt.figure(figsize=(10, 6))
sns.lineplot(data=results_df, x="Cost", y="Total Cost", hue="Walking Distance Limit",
marker='o')
plt.title("Total Cost vs Parking Cost")
plt.xlabel("Parking Cost")
plt.ylabel("Total Cost")
plt.legend(title="Walking Distance Limit", loc="upper left")
plt.show()

# Plot 2: Total cost vs Penalty Weight
plt.figure(figsize=(10, 6))
sns.lineplot(data=results_df, x="Penalty Weight", y="Total Cost", hue="Walking
Distance Limit", marker='o')
plt.title("Total Cost vs Penalty Weight")
plt.xlabel("Penalty Weight")
plt.ylabel("Total Cost")
plt.legend(title="Walking Distance Limit", loc="upper left")
plt.show()

print(results_df[['Walking Distance Limit', 'Used Parking Spots']].drop_duplicates())
```

Method 2

Using GAMS - for assignment of deliveries to the best parking spots available

```
\*------------------------------------------------------------

* DELIVERY PARKING ASSIGNMENT MODEL
 \*------------------------------------------------------------

Sets
i deliveries /1\*23/
j parking /75,78,35,47,43,58,61,67,73,70,76,2,6,11,26,34/;

Parameters
W(i,j)        Walking distance from delivery i to parking j
C(j)          Parking cost
D(i)          Delivery demand
maxdist       Max walking distance for normal demand /0.6/
strictdist    Max walking for high demand /0.3/
penalty\_weight /1000/
reuse\_penalty  /10/;

* Load data from .inc files
 \$include new\_walking\_1234\_latest.inc
 \$include parking\_cost\_thurs\_latest.inc
 \$include delivery\_demand\_gams.inc

\*------------------------------------------------------------

* VARIABLES
 \*------------------------------------------------------------
 Binary Variable
 x(i,j)      "1 if delivery i assigned to parking j"
 y(j)        "1 if parking j is used";

Positive Variable
p(i)        "Penalty if walking exceeds limits";

Variable
z           "Total cost";

\*------------------------------------------------------------
```

```
* EQUATIONS
 \*------------------------------------------------------------
 Equations
 obj                  "Objective function"
 assign(i)            "Assign each delivery exactly once"
 distConst(i,j)       "Distance limit for normal deliveries"
 strictDistConst(i,j) "Stricter limit for high-demand deliveries"
 link(i,j)            "Link x to y: can't assign if j not used";

* Objective function
 obj ..
 z =e= sum((i,j), C(j) \* x(i,j))
 \+ penalty\_weight \* sum(i, p(i))
 \+ reuse\_penalty \* sum(j, y(j));

* Every delivery assigned exactly once
 assign(i) ..
 sum(j, x(i,j)) =e= 1;

* Distance constraint for normal demand
 distConst(i,j)\$(D(i) <= 5) ..
 W(i,j) \* x(i,j) =l= maxdist + p(i);

* Stricter constraint for high-demand
 strictDistConst(i,j)\$(D(i) > 5) ..
 W(i,j) \* x(i,j) =l= strictdist + p(i);

* Link x and y
 link(i,j) ..
 x(i,j) =l= y(j);

\*------------------------------------------------------------

* MODEL & SOLVE
 \*------------------------------------------------------------

* option LP = cplex;     <-- REMOVE THIS LINE
 option solver = cplex;

Model deliveryAssignment /all/;
```

```
Solve deliveryAssignment using MIP minimizing z;
display x.l, y.l, z.l,p.l;
```