Project 2 Report

Haoliang (Jack) Hu, Haoxiang (Sean) Yi, Chaeyun Lim
McCombs School of Business, University of Texas at Austin
BAX358: Optimization Methods Decision Making
Dr. Daniel Mitchell
March 31, 2023

Introduction

The Traveling Salesman Problem (TSP) is classified as an NP-hard problem because of its complexity. That means it has no quick solution, and we could never get an optimal solution for the problem. Although we may not find the optimal result from it, using optimization methods definitely helps us to have reduced costs and lesser time. The following report tells an optimal route for delivering packages, how many trucks we should use, and what we should do with new packages.

The Optimal Route for Dataset 1

The first dataset we have includes a list of 500 delivery locations and their corresponding x and y coordinates. We are trying to find optimal paths that result in the minimum total distance traveled when using all 10 trucks. To make 10 trucks travel at their least distances, we observe 500 locations and divide them into 10 clusters so that the trucks won't trespass on other clusters.

Clustering

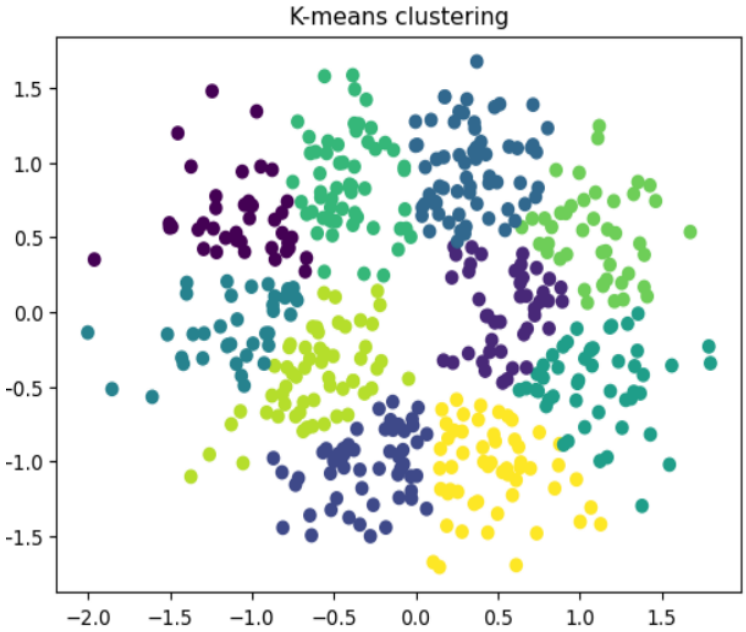
The clustering method we decided to use is **K-means**, where each location is grouped to the cluster with the nearest mean.

```
# Find the location clusters using K-means
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=10)
kmeans.fit(df1)
df1['cluster'] = kmeans.labels_

# Plot the clusters
plt.scatter(df1['x'], df1['y'], c=df1['cluster'])
plt.title('K-means clustering')
plt.show()
```

Graphical representation of the 10 clusters:



Then, we put each cluster into a data frame, add the distribution center to each cluster, and append all the cluster data frames into a list. We created a function that takes in a dataframe and outputs a distance matrix using the standard Euclidean distance formula. With the function, we were able to generate a distance matrix for each cluster.

```
def get_distance_matrix(df):
    ''' This function takes a dataframe with x and y coordinates and returns a distance matrix'''

    # Get the number of cities
    N = df.shape[0]

    distMat = np.zeros((N,N)) # distance matrix
    for i in range(N):
        for j in range(N):
            if i == j:
                # So we don't go from the same place to the same place
                distMat[i,j] = 1000
            else:
                # Euclidean distance
                distMat[i,j] = np.sqrt((df.x[i]-df.x[j])**2 + (df.y[i]-df.y[j])**2)
    return distMat
```

```
# Store the distance matrices in a list
clusters = 10
list_of_distance_matrices = []
for cluster in range(clusters):
    list_of_distance_matrices.append(get_distance_matrix(df_list[cluster]))
```

Optimization model

Now, we find out the optimal distance for each truck within the clusters. We use the **Miller-Tucker-Zemlin formulation**. Our objective is to minimize the total distance trucks traveled.

Objective function:

$$\min z = \sum_i \sum_j c_{ij} x_{ij}$$

(where i, j are the locations with numbers, x_{ij} variables are binary types that represent the path from location i to location j, and c_{ij} is the cost from location i to location j)

Constraints:

$$\begin{aligned}
\sum_{i=1, i\neq j}^{n} x_{ij} &= 1 && j = 1, \ldots, n; \\
\sum_{j=1, j\neq i}^{n} x_{ij} &= 1 && i = 1, \ldots, n; \\
u_i - u_j + (n-1)x_{ij} &\leq n-2 && 2 \leq i \neq j \leq n; \\
1 \leq u_i &\leq n-1 && 2 \leq i \leq n.
\end{aligned}$$

(where u variables are dummy variables that keep track of the order in which the locations are visited)

For each cluster, we create an empty gurobi model and add variables. And since the trucks can only enter and leave each location once, the sums of x_{ij}, where j = 1…n, and x_{ij}, where i = 1…n, are equal to 1.

In addition to the enter and leave constraints, the u constraints are used to enforce that there is only a single tour within each cluster, eliminating subloops.

With the objective function and constraints above, we are able to define the optimization model using Gurobi as shown below.

To prevent Gurobi from taking a long time to find the optimal solution, we created a time limit of 5 minutes for each model. This means the solution might not be the most optimal, but a good approximation. The optimal route for each cluster is then stored in a list for easier access later.

```
# Find the optimal distance for each cluster/truck
optimal_distances = []
route_for_each_cluster = []
for cluster in range(clusters):
    print('Working on cluster', cluster)
    N = df_list[cluster].shape[0]
    mod = gp.Model()
    # 25 x 25 variables
    x = mod.addMVar((N,N),vtype='B')
    # 25 u variables
    u = mod.addMVar(N)

    # enter each city once, j fixed
    # each column add to 1
    enter = mod.addConstrs((gp.quicksum(x[i,j] for i in range(N))==1) for j in range(N))
    # leave each city once, i fixed
    # each row add to 1
    leave = mod.addConstrs((gp.quicksum(x[i,j] for j in range(N))==1) for i in range(N))
    # u constraints
    ucons = mod.addConstrs((u[i] - u[j] + N*x[i,j] <= (N-1)) for i in range(1,N) for j in range(1,N) if i != j)

    # i is the row here, j is the column
    mod.setObjective(gp.quicksum(x[i,j]*list_of_distance_matrices[cluster][i,j] for i in range(N) for j in range(N)))

    mod.Params.OutputFlag = 0 # tell gurobi to shut up!!
    mod.setParam('TimeLimit', 5*60)
    mod.optimize()

    optimal_distances.append(mod.objVal)
    route_for_each_cluster.append(x.x)
```

Optimal solution

This is what we get for an optimal outcome of truck 1's traveling distance and its route. With the same method, we can obtain optimal results for the rest of the clusters/trucks.

```
Start at the distribution center, location = 0
Then go to location 292, traveling 0.11003965418426336 miles along the way.
Then go to location 272, traveling 0.06708434296327845 miles along the way.
Then go to location 258, traveling 0.07962235381947014 miles along the way.
Then go to location 342, traveling 0.0985530559830508 miles along the way.
Then go to location 333, traveling 0.019564784980763115 miles along the way.
Then go to location 317, traveling 0.19925990261124787 miles along the way.
Then go to location 251, traveling 0.10871182134936844 miles along the way.
Then go to location 298, traveling 0.03899731311921068 miles along the way.
Then go to location 257, traveling 0.0585245532777583 miles along the way.
Then go to location 327, traveling 0.0636242089211187 miles along the way.
Then go to location 267, traveling 0.07690116800158785 miles along the way.
Then go to location 273, traveling 0.0215151396965217 miles along the way.
Then go to location 299, traveling 0.07188740880259885 miles along the way.
Then go to location 217, traveling 0.03377815830228336 miles along the way.
Then go to location 218, traveling 0.1326382068734415 miles along the way.
Then go to location 209, traveling 0.08955102854791383 miles along the way.
Then go to location 213, traveling 0.1424841803372344 miles along the way.
Then go to location 287, traveling 0.14302526036349675 miles along the way.
Then go to location 296, traveling 0.0735684692413953 miles along the way.
Then go to location 256, traveling 0.23449095070437193 miles along the way.
Then go to location 240, traveling 0.11988927460182092 miles along the way.
Then go to location 236, traveling 0.11040891664494683 miles along the way.
Then go to location 252, traveling 0.0947522400199262 miles along the way.
Then go to location 500, traveling 0.2227290465978401 miles along the way.
Then go to location 328, traveling 0.2767784420628091 miles along the way.
Then go to location 264, traveling 0.07584166457477695 miles along the way.
Then go to location 207, traveling 0.1017087708636442 miles along the way.
Then go to location 235, traveling 0.04946136341761622 miles along the way.
Then go to location 339, traveling 0.1367509687049146 miles along the way.
Then go to location 347, traveling 0.024558179656212323 miles along the way.
Then go to location 330, traveling 0.04868825110523643 miles along the way.
Then go to location 295, traveling 0.11005162187321617 miles along the way.
Then go to location 315, traveling 0.04076736018165076 miles along the way.
Then go to location 397, traveling 0.0558988001889341 miles along the way.
Then go to location 290, traveling 0.16927916336757537 miles along the way.
Then go to location 275, traveling 0.0831971392395843 miles along the way.
Then go to location 337, traveling 0.1354870240153694 miles along the way.
Then go to location 374, traveling 0.055277060936884795 miles along the way.
Then go to location 343, traveling 0.16794821941358887 miles along the way.
Then go to location 434, traveling 0.001832570632468058 miles along the way.
Then go to location 312, traveling 0.08002159024779354 miles along the way.
Then go to location 283, traveling 0.08093384874602856 miles along the way.
Then go to location 318, traveling 0.05267694057126489 miles along the way.
Then go to location 200, traveling 0.1162498183637256 miles along the way.
For a total of 4.275010238108207 miles.
```

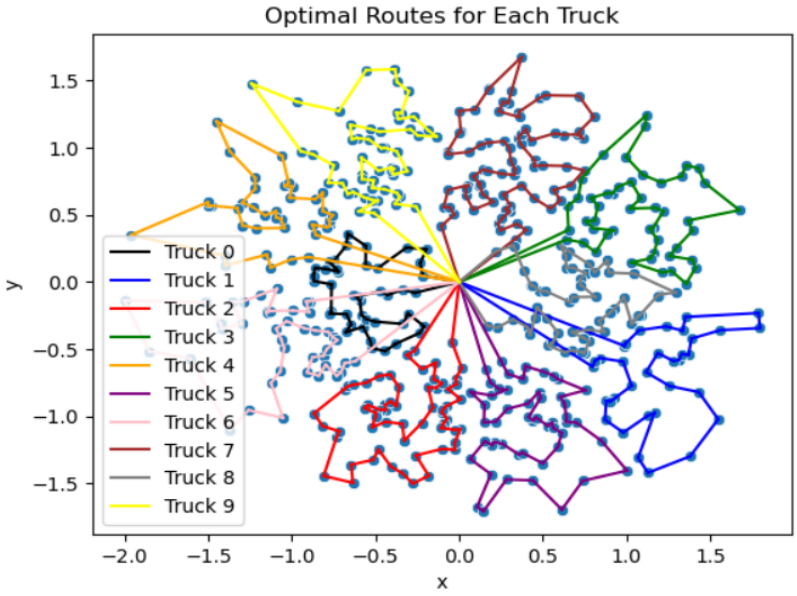
The optimal total distance with all trucks is around 65.23 miles.

```
# Sum of the optimal distances for all clusters/trucks
print('Total miles traveled: ', sum(optimal_distances))
```

✓ 0.0s

```
Total miles traveled:  65.2281135650637
```

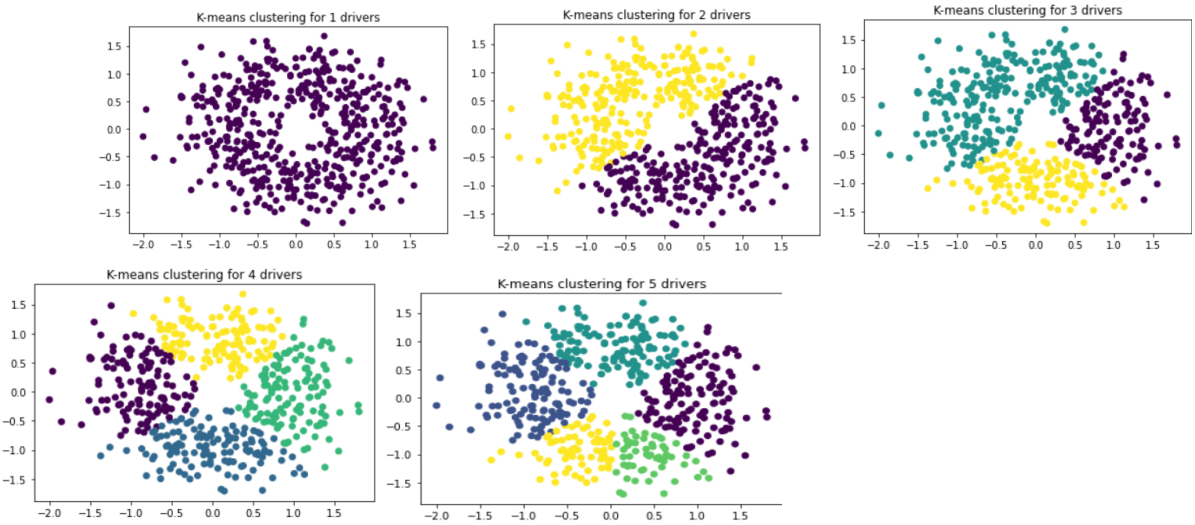
Here is a visualization of the optimal routes for each truck.

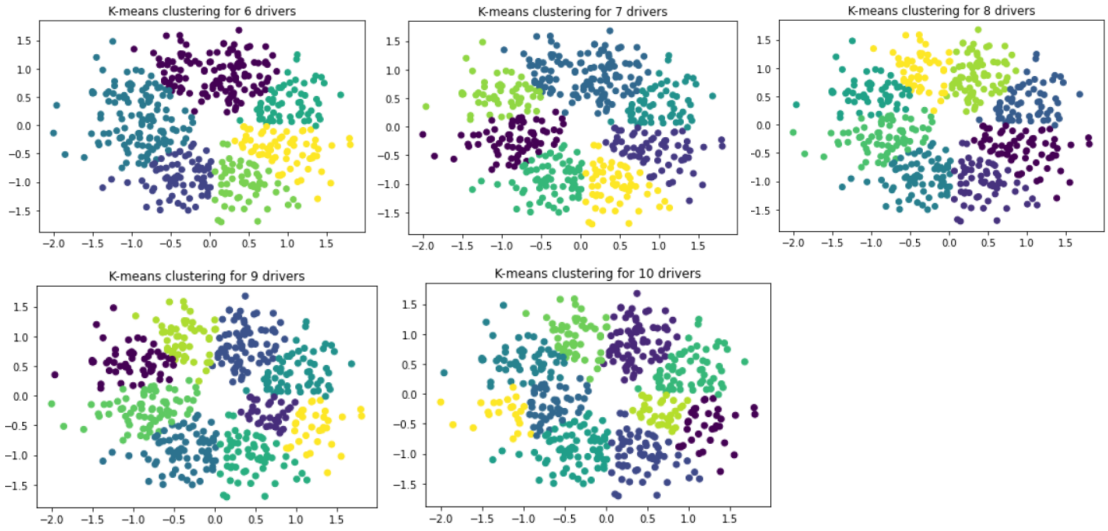


How Many Trucks Should We Use to Minimize the Cost?

Clustering

In order to find the number of trucks that minimize cost, we need to split dataset 1 into different clusters. Since we have up to 10 trucks, we split the locations into 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 k-classifications using k-means clustering. Here are the visualizations for each k-classification.





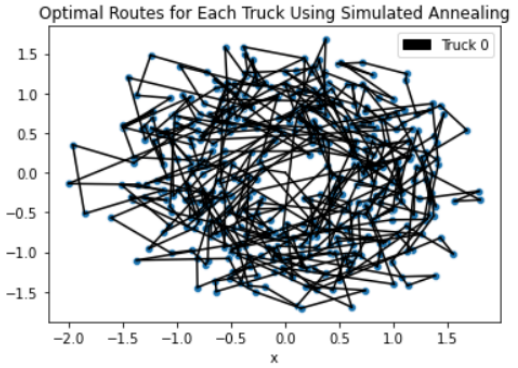
We then need to find the optimized routes for each cluster in each k-classification. We decided to do that by simulated annealing.

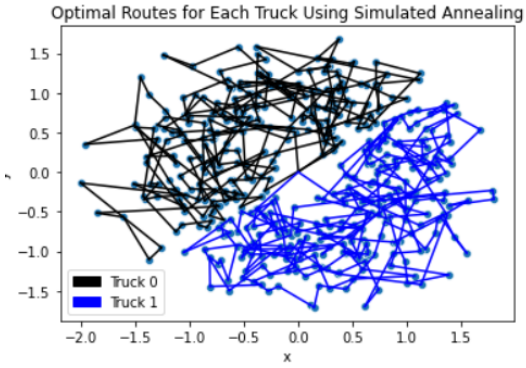
Simulated annealing

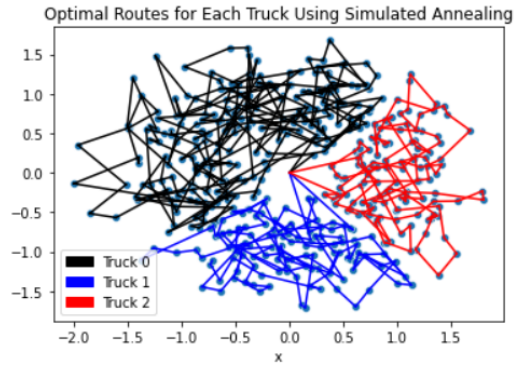
After splitting the data, we can then use simulated annealing on each cluster to find the optimal route for each cluster within each k-classification. Simulated annealing can help us save some time compared to Gurobi. The use of simulated annealing in our data allows for efficient exploration of a large number of possible routes while avoiding getting stuck due to the size and complexity of the locations. By using simulated annealing, we can find near-optimal solutions that are very close to the true optimal solution.

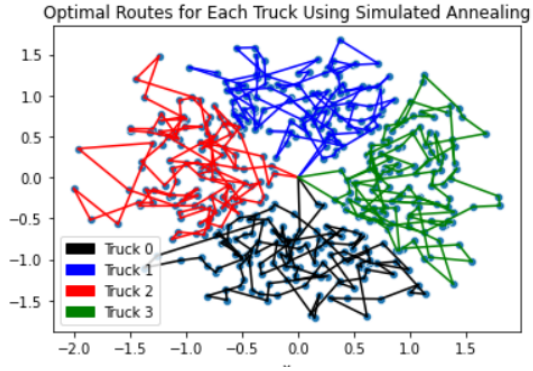
The simulated annealing algorithm starts by generating a random initial solution and iteratively searching for better solutions. At each iteration, the algorithm randomly disrupts the current solution and accepts the new solution if it is better than the current solution. The way our algorithm looks for new solutions is by randomly using either the reverse or transport method. However, if the new solution is worse, the algorithm may still accept it with a certain probability based on a temperature parameter that gradually decreases over time. This probabilistic acceptance of worse solutions allows the algorithm to avoid getting stuck in local optima and explore the search space more effectively.

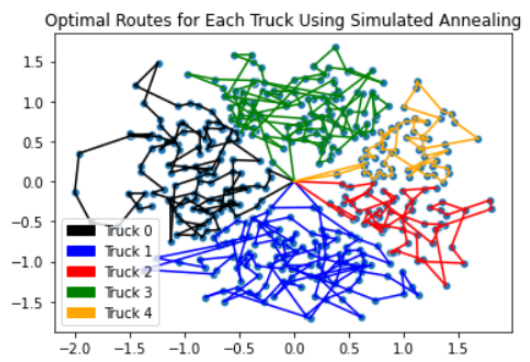
After doing simulated annealing, we are able to find the optimized routes for each cluster in each k-classification. Here are the visualizations of the optimal routes.

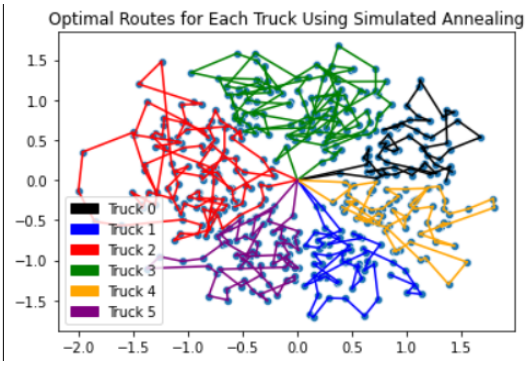


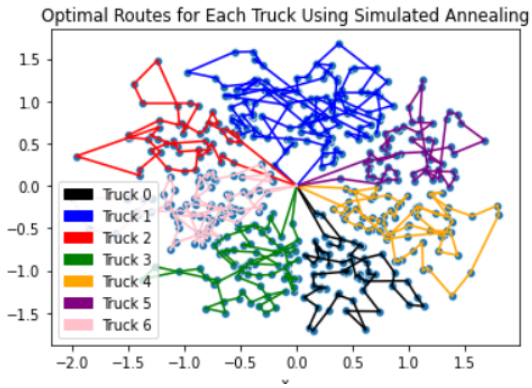


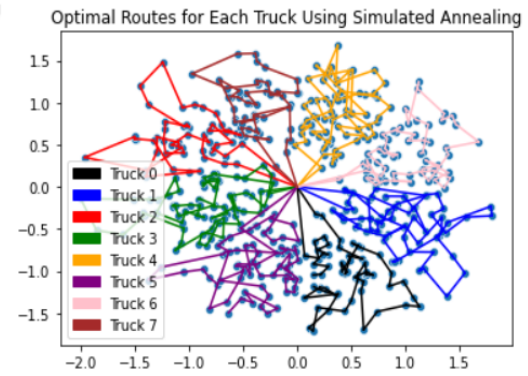


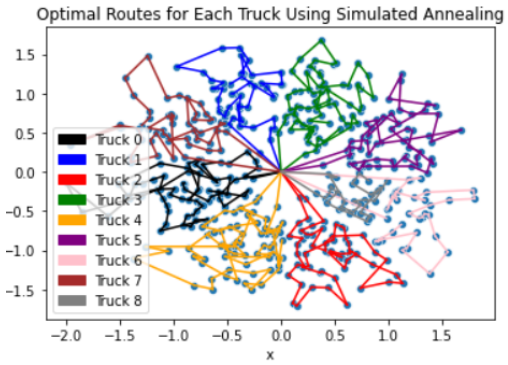


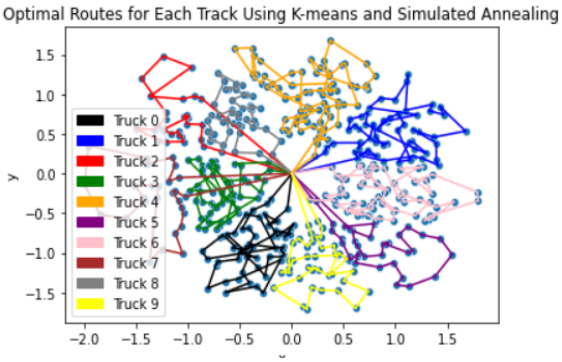












Our research reaffirms the reasonability of the statement given by the company that for every mile traveled delivering packages it costs $1 for gasoline, labor, and truck depreciation. Mileage depreciation is about $0.08 per mile and gasoline cost is about $0.65 per mile. Hence, we will use $1 as our cost in our models.

With driving each mile costing 1 dollar and each truck costing 300 dollars we are able to come up with the following report.

Number of trucks	1	2	3	4	5	6	7	8	9	10
Total distance	225.57	156.44	146.36	116.61	110.40	102.35	94	89.98	90.20	92.25
Cost of driver	300	600	900	1200	1500	1800	2100	2400	2700	3000
Total Cost	525.57	756.44	1046.36	1316.61	1610.40	1902.35	2194	2489.98	2790.20	3092.25

In the end, we find **one driver** will end up with the least cost of **529.77** dollars.

```
    # One driver is the optimal solution, with the least cost
    print('The least cost is',least_cost)
    print('The least k',least_k)
```

```
The least cost is 529.7731415701764
The least k 1
```

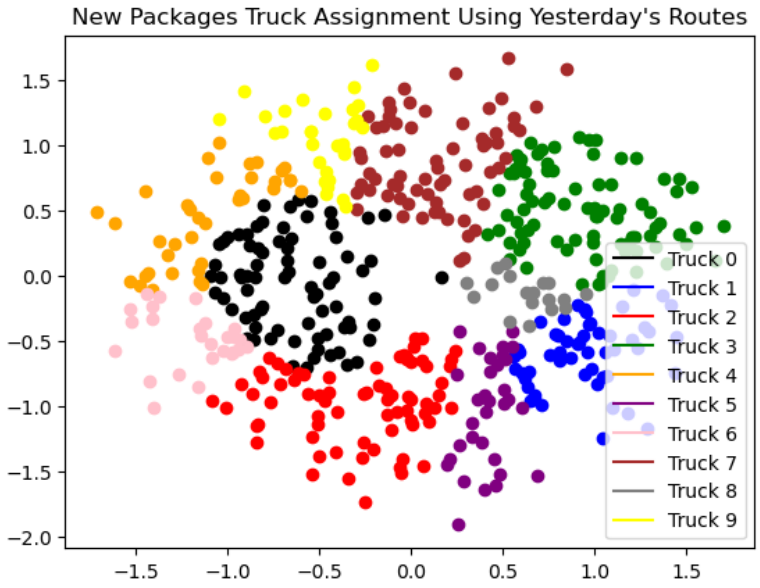
What Should We Do With New Packages?

For new packages on the next day (dataset 2), we can choose to use the same route as yesterday (dataset 1) or create a new model from scratch. We will see how each option performs below.

Using the same route:

If we choose to use the same route as yesterday, we can use a simple algorithm to group the locations in dataset 2 into clusters. For each location in each cluster of dataset 1, we look for the location in dataset 2 that are close and classify them with the same cluster as the location in dataset 1. We defined close as having a distance of less than 0.25 miles. We chose 0.25 as the distance threshold because after testing out different values, 0.25 is the value that can capture all 500 locations.

The clusters formed using yesterday's routes:



After grouping all locations in dataset 2 into clusters, we then found the total route distance for each cluster. The resulting total distance for all clusters is around **138.41** miles (will vary due to Gurobi's randomness)

The total distance traveled if using yesterday's routes:

```
Truck 0 will travel 21.46635845898244 miles.
Truck 1 will travel 25.25307239337034 miles.
Truck 2 will travel 24.736010604957883 miles.
Truck 3 will travel 14.663362018729767 miles.
Truck 4 will travel 6.017620644904638 miles.
Truck 5 will travel 7.450754075239372 miles.
Truck 6 will travel 9.707137216406737 miles.
Truck 7 will travel 14.543364853435397 miles.
Truck 8 will travel 9.17066797399629 miles.
Truck 9 will travel 5.398129450371523 miles.
```

```
based_on_yester_dist = sum(new_package_cluster_dist)
print('Total miles traveled: ', based_on_yester_dist)
```

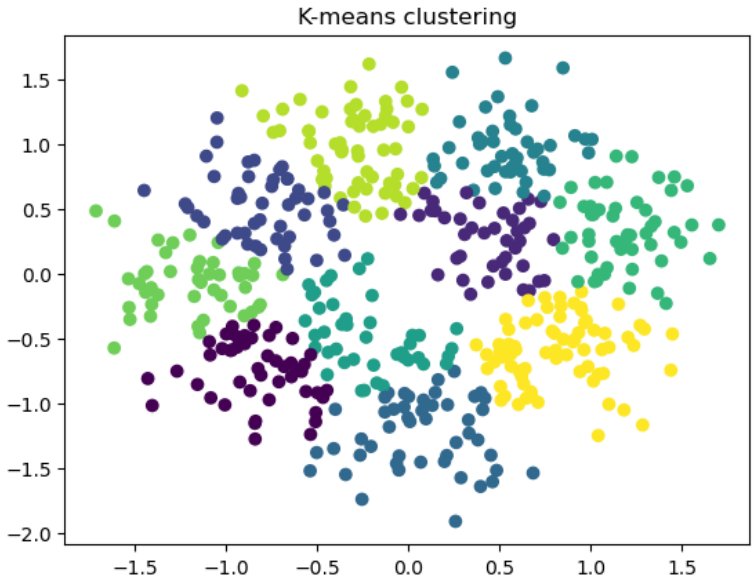
✓ 0.0s

```
Total miles traveled:  138.40647769039438
```

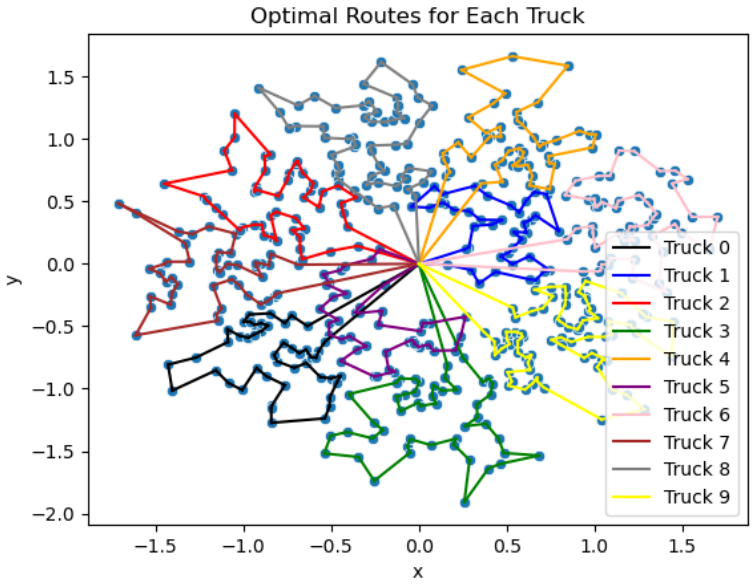
Building a new model from scratch:

If we build a new model from scratch using the same formulas and methods as the model for dataset 1, we will get a total distance of around **62.89** miles with all clusters (will vary due to Gurobi's randomness).

The clusters formed using a new model:



The optimal routes found using a new model:



The total distance traveled with all clusters if using a new model:

```
# Sum of the optimal distances for all clusters/trucks
based_on_today_dist = sum(optimal_distances2)
print('The total distance traveled by all trucks is', based_on_today_dist, 'miles.')
```

✓ 0.0s

```
The total distance traveled by all trucks is 62.892245172018036 miles.
```

Suggestion:

We can see the difference between using the optimal routes from yesterday vs. starting a new model is very large (138.41-62.89=**75.51** miles). We more than doubled our distance by not starting a new model, suggesting **starting a model each day is the best practice**.

The difference in total distance between the two options:

Difference in distance between using yesterday's route and setting up a new model:

```
based_on_yester_dist- based_on_today_dist
```

✓ 0.0s

```
75.51423251837635
```

Are There Other Methods to Split Up the Locations?

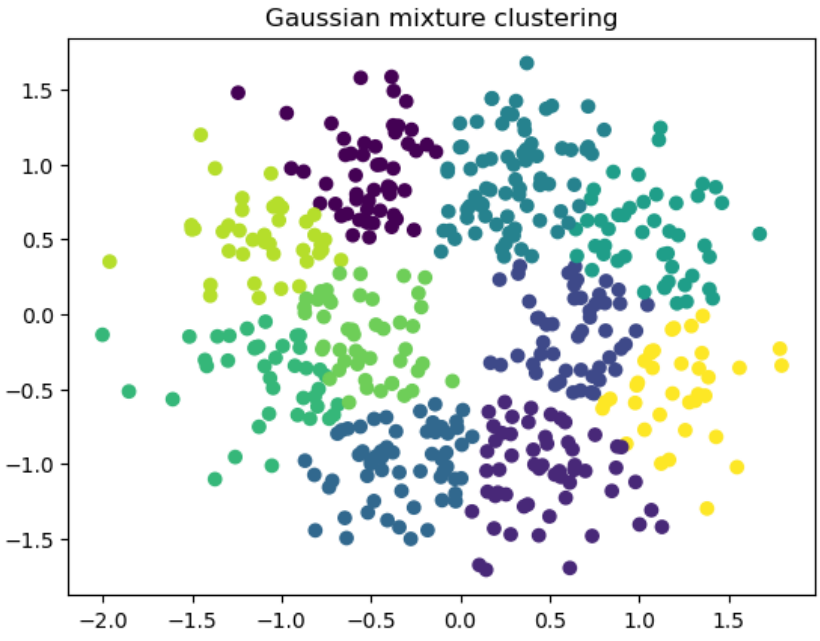
Another method to split up the locations is to use the **Gaussian mixture method**.
GMM may be better than K-means in certain scenarios for several reasons:

- Flexibility: GMM is more flexible in modeling the shape and size of clusters compared to K-means. K-means assumes that clusters are spherical and equally sized, whereas GMM can model clusters of different shapes and sizes.
- Soft clustering: GMM provides a soft clustering approach, which means that each data point is assigned a probability of belonging to each cluster. This is useful when a data point could belong to more than one cluster or when a cluster overlaps with another.
- Capturing variance: GMM models each cluster as a combination of Gaussian distributions, allowing it to capture the variance within each cluster. K-means only considers the mean of each cluster, which may not be representative of the entire cluster.
- Better for non-linearly separable data: K-means may struggle with non-linearly separable data, where data points in the same cluster may not be close to each other. GMM, on the other hand, can capture the complex relationship between data points and their probability of belonging to each cluster.

However, it's important to note that GMM is computationally more expensive than K-means and requires more tuning of its hyperparameters. The choice between the two algorithms ultimately depends on the nature of the data and the goals of the analysis.

Since our data do not have visually well-separated clusters, GMM might be a better approach than K-means.

Using the GMM classifier in scikit-learn, we formed the following clusters for dataset 1:



Simulated annealing

After splitting the data using GMM, we can then use simulated annealing again on each cluster to find the optimal route for each cluster.

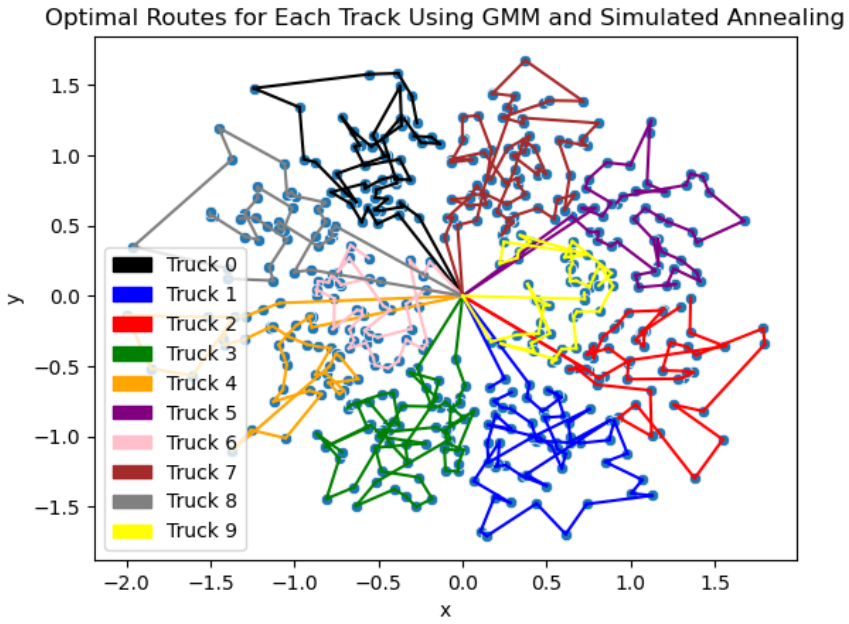
The optimal total distance of all clusters for dataset 1 is around **90.44** miles (varies due to the randomness of GMM), which is not as good as the Gurobi model of 65.23 miles, but still an acceptable approximation.

```
# Print the total distance for all clusters
print('Total distance for all clusters:',sum(optimal_dist_list))
```

✓ 0.0s

```
Total distance for all clusters: 90.43926390293748
```

The optimal routes found using simulated annealing:



Conclusion

In conclusion, we **suggest using optimization methods to solve the optimal routes each day**. The main advantage of using optimization methods is that finding the optimal routes can lead to **significant cost savings and efficiency improvements**. However, we must also understand finding the optimal routes is known to be an NP-hard problem, meaning that it **can be computationally infeasible** for a large number of locations, and the best available algorithms (e.g. simulated annealing) can only provide approximate solutions. Additionally, our delivery problem is sensitive to small changes in the input, and the quality of the solution can depend heavily on the specific locations and the chosen algorithm.