FINAL PROJECT REPORT
CS 5800 – ALGORITHMS

Summer 2023 semester

Efficient Customer Clustering and Path Planning Algorithms for Service Optimization

Prithiv Rajkumar, Aishwarya Suyamindra, Haritha Selvakumaran, Ravi Shankar Sankara Narayanan

Project context

At present, we rely a lot on transportation services such as food, grocery deliveries, and cab services. It is necessary for these types of services to be efficient to save time and resources. Our objective is to implement an algorithm that aims to group customers with similar characteristics and plan optimal routes to minimize costs, time, and resources while maximizing customer satisfaction.

Introduction

1. **Prithiv**:

All of us use food delivery services, as a customer I always prefer to receive my order in the shortest time. Aside from the time taken for food preparation, there is a lot of time spent delivering the food to the customers as well. Even when I order food from a restaurant close to my house, I will be receiving the food after the delivery person completes 4 or 5 orders in various locations. A takeaway is much more efficient than relying on delivery services in these instances. Most of the time is spent in delivering the food to customers than in preparation and there can be ways where this could be optimized to reduce the total time taken to deliver the food to the customer.

We aim to optimize the food delivery service by reducing the delivery time by implementing clustering and a shortest-path algorithm. We can cluster customers in similar neighborhoods and assign a delivery person to each cluster and among the clusters we can use algorithms to find the shortest path between these locations. By implementing this we can optimize the food delivery duration by reducing resources and serving more customers at the same time.

2. **Aishwarya**:

As a student, I am always trying to minimize the amount of time spent on commute. I would like to get to my destination quickly and rely on a shared cab service as a viable option. However, a lot of times, this ends up taking more time than if I chose a different means of transportation, such as public transport, or even walking. I have observed that often, users that have close-by destinations are not necessarily grouped together. As a result, a lot of time is spent on travel. This also leads to inefficient route planning, taking longer routes and even re-visiting some areas. These inefficiencies have driven me to delve deeper and explore solutions that can reduce the time taken to commute and enhance the overall ride-sharing experience.

I aim to draw upon the knowledge gained from the coursework, where I learnt about various algorithms that can help solve this problem in an efficient manner. By clustering users with similar destinations, we can create groups of users for each ride, so that the route is mapped efficiently without any unnecessary detours. This also has broader implications, as it can be used to optimize networks such as food and groceries delivery, that require a wide range of audiences to be serviced, in the shortest amount of time. Through this project, my team and I intend to optimize the delivery and scheduling network, reducing the overall travel time for everyone.

3. **Haritha**:

Efficient customer clustering and path planning algorithms hold immense potential for optimizing the RedEye shuttle service for Northeastern students. The algorithm behind the shuttle service at various times allocates students from quite different localities to the same RedEye vehicle, which leads to increased travel time and inconvenience. It also fails to identify the shortest path between drop destinations, which in turn is cost-ineffective.

Using efficient algorithms that strategically groups students based on their locations can significantly reduce such wait times and travel distances, which are some of the most prevalent issues with the shuttle service. Clustering students who live in close proximity allows the shuttle to serve multiple students in a single trip efficiently. Additionally, the implementation of the shortest path algorithm ensures that the shuttle follows the most optimized routes, taking the clustered student locations as stops.

Moreover, this also contributes to reduced fuel consumption and can be identified as a cost-effective methodology. By optimizing the routes, the shuttle can cover more students with fewer resources, leading to cost savings for the service provider. Our proposed algorithm can help the service provider streamline operations, increase efficiency, and provide a better experience for the students, making the service more beneficial for the university community.

4. **Ravi**:

RedEye Shuttle Service has been a valuable asset for our university community, providing a safe and convenient transportation option for students living off-campus or working late hours. As a student who frequently relies on the RedEye Shuttle, I often encounter extended wait times and lengthy travel durations. Despite living only 2 miles away from the university, the commute can take up to 80 to 90 minutes, whereas a public transport ride would be significantly shorter, around 20 minutes. This discrepancy in travel times has prompted me to investigate ways to optimize the shuttle service.

One observation I have made is that students living in opposite corners of Boston are often grouped together in the same shuttle. This practice leads to inefficient routing, as the shuttle travels from one corner to another, picking up and dropping off students along the way. Consequently, the shuttle follows smaller streets instead of utilizing major thoroughfares like Columbus Avenue, resulting in longer travel times.

To address these issues and enhance the overall experience for all users, my proposal is to implement an efficient customer clustering algorithm and optimize path planning for the RedEye Shuttle Service. By pooling students from the same or nearby neighborhoods together, we can significantly reduce wait times and ensure quicker and more direct routes to their respective destinations. Moreover, by strategically utilizing major streets, we can streamline the shuttle's path and achieve faster travel times for all

passengers. This optimization will not only benefit students like me but also enhance the overall efficiency and satisfaction of the RedEye Shuttle Service for the entire university community.

Question

Given data of student's or customer's details like their first and last name, their address, and the co-ordinates (latitude and longitude), How can we leverage various clustering and path planning algorithms to enhance the efficiency of serving them?

Our objective is to design two algorithms that effectively group students/customers based on their geographical data (latitude and longitude) and optimize the travel route for reaching the destinations. The first algorithm will focus on intelligently clustering students/customers together. The second algorithm will determine the most optimal path and order of destinations, considering their addresses, to significantly reduce the total travel time for the service.

Analysis

This project involves the implementation of efficient customer clustering and path planning algorithms to optimize services operations. We will employ MST-based clustering for customer clustering, creating groups based on geographical proximity. This will facilitate optimal route planning. Additionally, we will be using Travelling Salesman Problem algorithm, ensuring the shortest routes between locations. By combining these algorithms, the project aims to reduce travel time, fuel consumption, and environmental impact while enhancing customer satisfaction and overall service efficiency.

How do we gather our data?

To gather data within the geographical confines of Boston, the process involves defining the specific latitude and longitude boundaries that encompass the city. In this case, the minimum and maximum latitude values are set at 42.2279 and 42.4061 respectively, while the minimum and maximum longitude values are established as -71.1912 and -70.9865. To collect a dataset of geospatial coordinates within these limits, a randomization approach is employed using the NumPy library in Python. By utilizing the `np.random.uniform()` function, 60 sets of latitude and longitude coordinates are generated, adhering to the specified boundaries. The randomization is consistent across sessions due to a predetermined seed value (16 in this instance). Both customer clustering and path planning algorithms rely heavily on accurate geospatial data, and the randomly generated coordinates within Boston's geographical boundaries provide a representative dataset for testing and refining these algorithms.

```
# Define latitude and longitude bounds for Boston
min_latitude = 42.2279
max_latitude = 42.4061
min_longitude = -71.1912
max_longitude = -70.9865
```

```
# Generate 60 random latitude and longitude coordinates within Boston bounds
num_points = 60
np.random.seed(16)
latitudes = np.round(np.random.uniform(min_latitude, max_latitude, num_points), 6)
longitudes = np.round(np.random.uniform(min_longitude, max_longitude, num_points), 6)
```

Fig 1 Code Snippet

```
longitudes
```

```
array([-71.103045, -70.995153, -71.181424, -71.0412  , -71.052891,
       -71.083322, -71.075138, -71.043213, -71.051471, -71.028399,
       -71.168101, -71.150431, -71.043546, -71.110575, -71.121769,
       -71.15979 , -71.170612, -71.107837, -71.071877, -71.039575,
       -71.013037, -70.993352, -71.097946, -71.109233, -71.130007,
       -71.021635, -71.004956, -71.034372, -71.177216, -71.067146,
       -70.998942, -71.170291, -70.991098, -71.118186, -71.057288,
       -71.166066, -70.994411, -71.057903, -70.999462, -71.005163,
       -71.101745, -71.000161, -71.046637, -71.136517, -71.154402,
       -71.057008, -71.022931, -71.005351, -71.063022, -71.15788 ,
       -71.133547, -71.070794, -71.168325, -71.136029, -71.190794,
       -71.007673, -71.000762, -71.121554, -71.104698, -71.054011])
```

```
latitudes
```

```
array([42.26769 , 42.321128, 42.326035, 42.236026, 42.292182, 42.267653,
       42.350631, 42.257077, 42.240432, 42.395588, 42.328348, 42.241798,
       42.356675, 42.256136, 42.2725  , 42.280199, 42.352036, 42.310632,
       42.266224, 42.307524, 42.249441, 42.391432, 42.328841, 42.312014,
       42.283956, 42.235601, 42.361986, 42.324711, 42.364613, 42.368758,
       42.24077 , 42.345461, 42.237398, 42.349971, 42.306905, 42.311297,
       42.345701, 42.350852, 42.274825, 42.229946, 42.365345, 42.373337,
       42.252784, 42.330901, 42.288608, 42.269281, 42.263298, 42.27628 ,
       42.401975, 42.266289, 42.291412, 42.231179, 42.337812, 42.268958,
       42.360187, 42.332013, 42.302762, 42.354289, 42.401119, 42.372249])
```

```
# Combine latitudes and longitudes into a feature matrix
coordinates = np.column_stack((latitudes, longitudes))
```

Fig 2 Code Snippet II

Markers for each coordinate on the map -

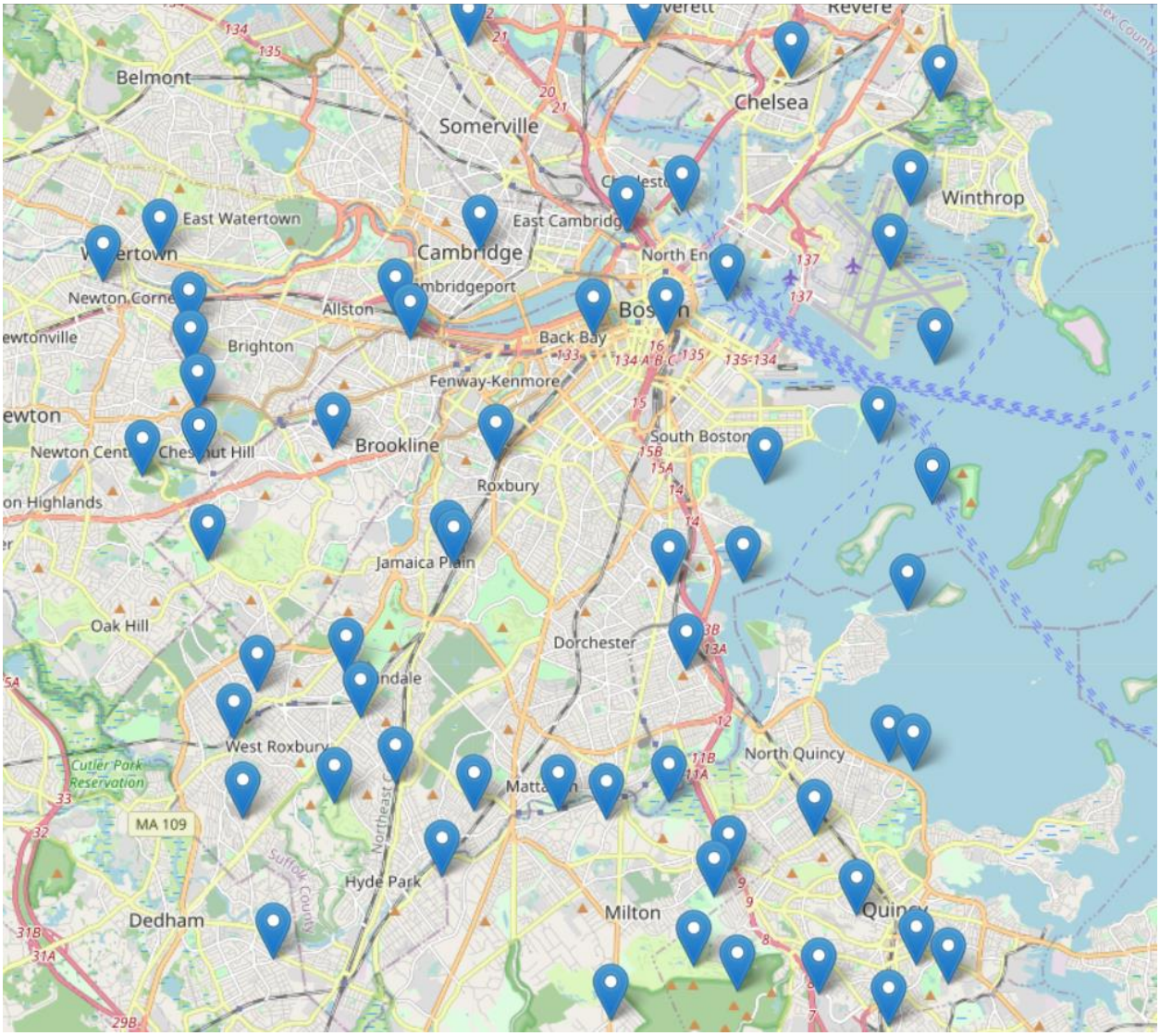

Fig 3 Map with markers

What do we do and why?

As the first step, we focused on clearly defining the problem statement, the objective and the various use cases of the problem statement.

We then looked into modelling the problem statement. The problem of determining the shortest route between various destinations can be modelled as a graph, where the nodes of the graph represent the various locations that users have scheduled a ride to. The edges on the graph represent the distance between the locations. The problem of clustering close by destinations to each other can be modelled to be X-Y co-ordinates on a 2D plane, where X and Y represent the latitude and longitude data respectively, of the user's destination.

We narrowed the scope of the project and established the relevant factors and constraints. Keeping the time constraint in mind, we narrowed the scope of the project to restrict the user to schedule a ride only within a set radius from the destination. We intend to consider the number of available cabs and the capacity of the cab as factors during clustering.

Further, we have done our research on the various algorithms that can be used to achieve the objective. As our project aims to first cluster locations that are closer to each other, we investigated the various clustering algorithms that can be used to implement this. K- means is an efficient clustering algorithm for large data points. As we are dealing with a small number of data points, the MST based clustering technique is also efficient, so we explored this as well. The other main objective of the project is to determine the shortest route between all the locations, so we looked into algorithms for the same.

Clustering

K-means

We used the K-means clustering algorithm to group a set of latitude and longitude coordinates into a specified number of clusters. K-means is an iterative algorithm that aims to partition data points into clusters where each point belongs to the cluster with the nearest mean (centroid). The code initialized the KMeans model and fitted it to the given coordinates. The cluster labels for each point were obtained, and cluster centers were calculated. Additionally, the code ensured that each cluster contains a maximum of 10 points by modifying the data allocation. It then visualized the clusters using scatter plots, where each point was colored according to its cluster label and cluster centers were marked with black 'X' markers. This approach allowed for spatial data to be grouped and visualized based on their geographical coordinates, aiding in insights and analysis.

The code provided demonstrates the use of the K-means clustering algorithm on a set of geographical coordinates (latitude and longitude). Explanation of the code along with its time complexity:

1. Data Generation and Preparation: The code begins by generating or using provided latitude and longitude data. These coordinates are combined into a feature matrix using `np.column_stack()`.

2. K-means Initialization and Fitting: The KMeans model is initialized with the specified number of clusters (`num_clusters`). The `fit()` method is then called to fit the model to the data points. The initialization step involves random placement of cluster centroids.

3. Assigning Data Points to Clusters: After fitting, the `kmeans.labels_` attribute holds the cluster labels for each data point. This attribute is used to assign each point to its nearest cluster.

4. Cluster Centers: The `kmeans.cluster_centers_` attribute contains the coordinates of the final cluster centers.

5. Ensuring Max Points per Cluster: To ensure each cluster contains a maximum of 10 points, a dictionary `cluster_coordinates` is created to store the coordinates of each cluster. A loop iterates through each data point, assigning it to a cluster while checking the maximum limit.

6. Plotting Clusters: The code uses `matplotlib` to create a scatter plot of the data points. Each point is colored according to its cluster label, and cluster centers are marked with black 'X' markers.

Time Complexity Analysis:

1. Initialization: O(k * d), where k is the number of clusters and d is the dimensionality (2 in this case).
2. Assignment Step: O(n * k * d), where n is the number of data points.
3. Update Step: O(n * d)
4. Iteration: The number of iterations varies, and each iteration involves assignment and update steps.

Overall, the time complexity can be approximated as O(I * n * k * d), where I is the number of iterations needed for convergence. In practice, the number of iterations tends to be relatively small, and the algorithm's efficiency depends on the number of data points, clusters, and the distribution of data. While it's possible to have worst-case scenarios that exhibit quadratic behavior (O(n^2)), it's more common for the algorithm to behave closer to linear time complexity **(O(n * k * d))** in practical applications.

Other reviewed approaches to clustering:

MST based clustering.

Kruskal's algorithm is a greedy algorithm that can be used to find the minimum spanning tree (MST) of an undirected, weighted graph. A minimum spanning tree is a subgraph that contains all the vertices of the original graph with the minimum possible total edge weight, without forming any cycles. Its greedy approach guarantees that each step is locally optimal, leading to a globally optimal solution. To find the minimum spanning tree, the following steps are taken:

1. Sort edges by their weight, in a non-decreasing order.
2. Add the edge with the lowest edge weight to the minimum spanning tree, unless adding this edge creates a cycle.
3. Add edges until all vertices are connected.

To check if adding an edge will result in a cycle, the Union Find data structure is used. Initially, each node is considered aa a separate component. As the edges are traversed and added to the minimum spanning tree, the disjoint set data structure is used to keep track of which nodes are in the same component. It is used to maintain partitions of a set into disjoint subsets and two functions ‘Union’ and ‘find’ are performed. The Union operation is used to connect two components of a graph, when an edge is added to the minimum spanning tree. The subsets containing the two nodes are merged. The Find operation is used to determine if two elements belong to the same component, it finds the component that a given node is part of. In this manner, the connectivity information of the components of the graph is maintained, which helps identify if adding an edge to the MST results in a cycle.

The process of clustering aims to find clusters with maximum spacing between data points. Given a graph, the connected components represent clusters. Edges are added between the closest pairs of nodes, in increasing order of distance between the nodes, ensuring that no cycles are added. So, each time an edge is added between two different components, it is similar to merging the two corresponding clusters. This is termed single-link clustering, which is a special case of hierarchical agglomerative clustering. The procedure is similar to how a minimum spanning tree is built using Kruskal’s algorithm. To obtain k clusters, the minimum spanning tree is built until k connected components are obtained. Kruskal’s algorithm is run and stopped before the last k – 1 edges are added, which is equivalent to building the

whole minimum spanning tree and then deleting the k – 1 most expensive edges. The connected components obtained then represent the k clusters.

To obtain clusters for our data nodes, which are a set of latitude and longitudes representing the user's location, first a graph is built that connects all these points, with the edge weight representing the Euclidean distance between the two nodes. Kruskal's algorithm is then used to obtain the minimum spanning tree for the connected graph. Then, on deleting the k – 1 most expensive edges from the minimum spanning tree, k clusters are obtained. This approach gives us the required k clusters but does not regulate the number of nodes in each cluster. Depending on the data nodes, the number of nodes in each cluster varies, sometimes with a large number of nodes in one cluster and a relatively smaller number of nodes in other clusters.

The time complexity:

Initialization: To build the connected graph, as we add an edge from every coordinate to every other coordinate, this takes $O(n^2)$ time.

Find the MST: Kruskal's algorithm takes O(ElogE) time, where E is the number of edges in the graph. As we have a fully connected graph, we have n^2 edges.

Nodes clustered using this approach:

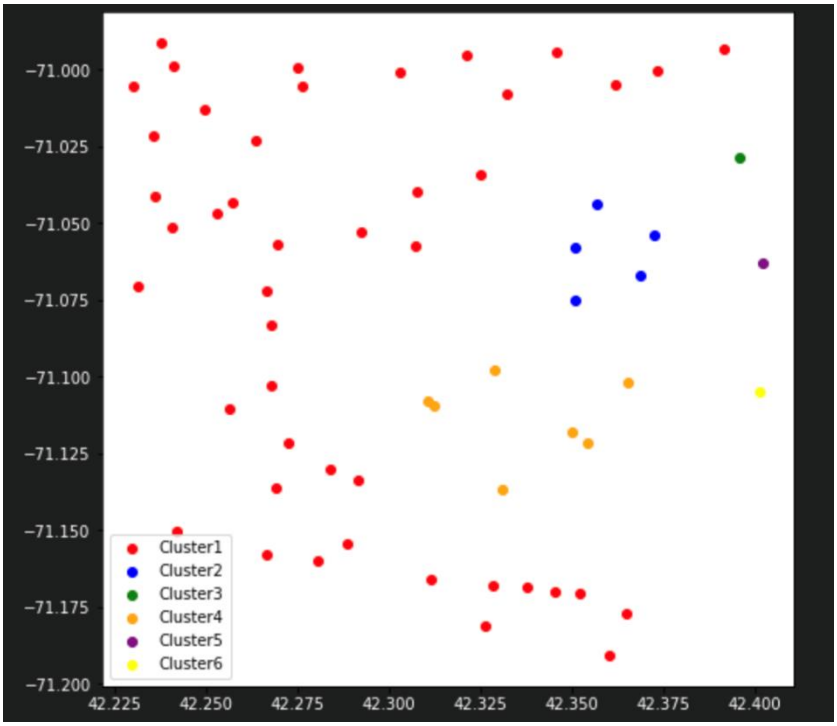

Fig 4 Clustered data points

For our use case, as we would like to ensure no more than 10 coordinates per cluster, we tried a variation of this algorithm. While building the minimum spanning tree using Kruskal's algorithm, the edges are first sorted in the decreasing order of their weights. As each of these edges are iterated over, we validate if removing this edge results in clusters that have less than 10 nodes in a cluster. If it does, this edge is

maintained in the MST. If not, this edge is removed, which gives us disconnected components. This process is repeated until either all the edges are iterated over, or the required number of k clusters are obtained. This approach does not result in guaranteed k number of clusters, but the number of co-ordinates in each cluster is relatively the same. This approach is also computationally intensive, as it requires that for each edge in the MST, the clusters be determined to validate the number of nodes in each cluster.

In summary, the MST – based clustering approach effectively clusters the coordinate data, but it does not factor in our required constraint. So, it is not suited to our use case as we aim to have a specified number of clusters and a specified number of coordinates in each cluster.

Nodes clustered using the second approach:

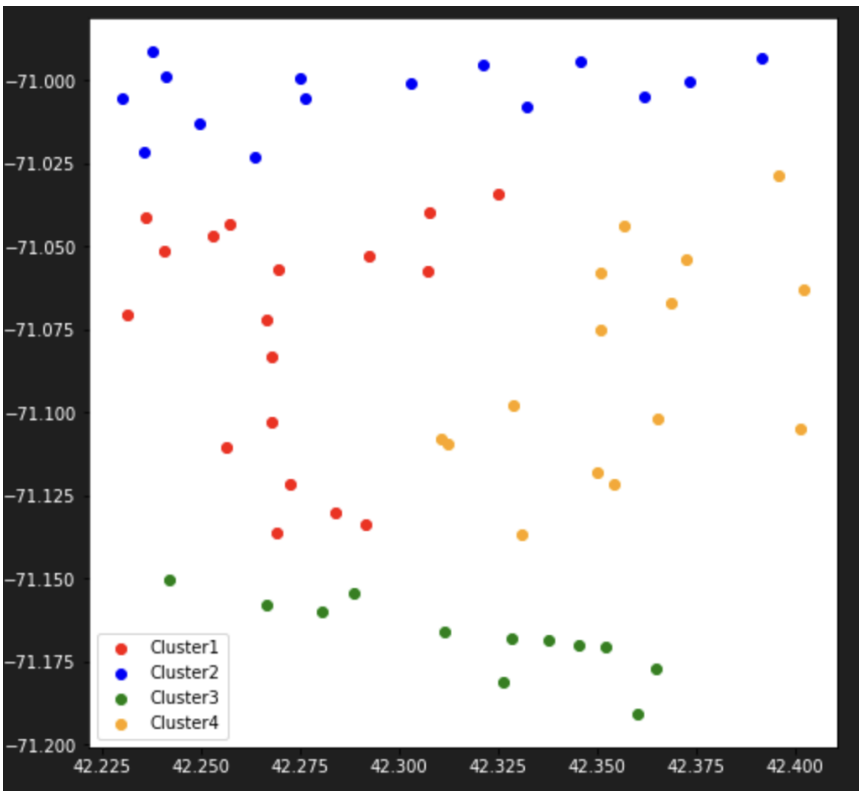

Fig 5 Clustered data points using an alternative approach

Path Finding

The objective of using the path planning algorithm in our project is to ensure that the vehicle starts from the base location, reaches all the customer locations, and returns back to the base in the most efficient way to save time and fuel. The path planning algorithm will help the project to decide the order of the stops to make.

Once the clustering algorithm separates all data points into clusters, we can implement the path-planning algorithm to work on each cluster. As a result, we will obtain a particular order of customer locations for the vehicle to make and return to the base location.

We have taken the RedEye Shuttle service as a particular example for designing the algorithm Various path planning algorithms were studied to find the perfect one that will meet our needs. On studying various algorithm, we found that The Traveling Salesman Problem (TSP) algorithm is well-suited for the problem of finding an efficient path for a vehicle to reach multiple customer locations and return to the base location.

The primary objective of the TSP algorithm is to find the shortest possible route that visits all customer locations exactly once and returns to the starting point (university). This aligns perfectly with our problem of efficiently visiting all customers and returning to the university in the shortest distance.

Two versions of the TSP algorithm were tested for getting the optimal path, one using brute force method and the other using the dynamic programming approach. The advantages and disadvantages of both versions were deeply reviewed before implementing them.

The decision of which version of the code (brute force or dynamic approach) is better depends on several factors, including the size of the dataset, time constraints, and the level of optimality required. Let's compare these approaches:

TSP Brute Force version

TSP brute force is a method to solve the Traveling Salesman Problem (TSP) by systematically considering all possible permutations of the cities and evaluating the total distance for each permutation. The TSP is an optimization problem where a salesperson must visit a given set of cities exactly once and return to the starting city, aiming to find the shortest possible route that visits all cities.

- Pros:
    - **Guarantees finding the optimal solution:** The brute force method explores all possible permutations of the customer locations, ensuring that it finds the shortest possible path.

- Cons:
    - **Exponential time complexity:** The time complexity of the brute force method is O(n!), where n is the number of customer locations. This makes it impractical for larger datasets, as the number of permutations grows rapidly with the input size.

- **Inefficient for larger datasets:** As the number of customer locations increases, the brute force approach becomes computationally expensive and may not provide a solution within a reasonable time frame.

The brute force approach explores all possible permutations, making it exponential in time complexity. While it guarantees finding the optimal solution, it becomes computationally impractical for larger datasets due to the huge number of permutations to explore. The working of the algorithm is explained below:

1. 'euclidean_distance(point1, point2)': This is a helper function that calculates the Euclidean distance between two points represented as tuples (latitude, longitude).
2. 'tsp_brute_force(university, cluster)': This is the main function that takes the university coordinates and a cluster of customer locations as input and returns the best path and its distance.
3. Append University: The university coordinates are appended to the 'cluster', as the TSP requires visiting the university at the end of the path.
4. Initialize Variables: 'best_distance' is set to infinity, and 'best_path' is initialized as an empty list to store the best path and its distance found during the algorithm.
5. 'permute(locations)': This is a recursive helper function that generates all possible permutations of the locations to explore different orders to visit the customer locations.
6. Base Case Check: If all locations are visited (except the university), the current permutation is a valid path. The function calculates the total distance of this path and updates the 'best_distance' and 'best_path' if it's shorter than the previously found best path.
7. Recursive Permutation: If not all locations are visited, the function iterates through the remaining unvisited customer locations and generates permutations by swapping the locations.
8. Swap and Recursive Call: It swaps the current location with another unvisited location and calls 'permute' recursively with the updated set of locations to generate the next permutation.
9. Backtrack: After calculating the distance for a particular permutation, the function backtracks by swapping the locations back to their original positions, so it can explore other permutations.
10. Execute 'permute': The 'permute' function is called initially with the 'cluster' (including university) to start the process of generating all possible permutations.
11. Return Best Path and Distance: After exploring all permutations, the 'tsp_brute_force' function returns the 'best_path' and its corresponding 'best_distance'.
12. Sample Input and Output: The sample input with university coordinates and customer locations is provided, and the best path and its distance are printed as output.

Derivation of the time complexity:

The time complexity of the brute force method for the Traveling Salesman Problem (TSP) is **O(n!),** where n is the number of customer locations (including the university).

1. The 'permute' function generates all possible permutations of the customer locations. In the worst case, there are **n!** permutations to explore.

2. For each permutation, the algorithm calculates the total distance of the path by summing the distances between consecutive customer locations and the university. This distance calculation takes **O(n)** time.

3. Overall, for each of the n! permutations, the algorithm performs **O(n)** distance calculations.

Therefore, the total time complexity is **O(n! * n),** which simplifies to **O(n!**).

As n grows, the number of permutations increases rapidly, making the brute force approach highly impractical for even moderately sized datasets. For small datasets with only a few customer locations (clusters), the brute force method may be feasible, but it quickly becomes computationally infeasible for larger datasets.

TSP Dynamic Programming version

TSP dynamic programming is an algorithmic technique used to solve the Traveling Salesman Problem (TSP) in a more efficient way compared to the brute force approach. Dynamic programming breaks down the TSP into smaller subproblems and stores their solutions to avoid redundant calculations, which significantly reduces the computational complexity.

The TSP dynamic programming algorithm is based on the principle of optimality, which states that an optimal tour of the entire set of cities can be constructed from optimal tours of subsets of those cities.

- Pros:
    - **Faster for small datasets:** The TSP DP method is more efficient than the brute force approach for small datasets with a limited number of customer locations. It avoids redundant calculations through dynamic programming, leading to a time complexity of O(2^n * n^2).
    - **Suitable for small clusters:** Since each cluster in your problem contains a maximum of 10 data points, the TSP DP approach is likely to provide a good solution in a reasonable time frame.
    - **Near-optimal solution:** While the TSP DP approach may not always find the globally optimal solution, it generally provides a near-optimal solution due to the use of dynamic programming.
- Cons:
    - **Not always globally optimal:** The TSP DP method may not always find the globally optimal solution for larger datasets. Depending on the specific dataset, it might give suboptimal results in some cases.

In this version of the TSP algorithm, we use dynamic programming to find the optimal path that visits all customer locations and returns to the university in the shortest distance. The DP table ‘dp_table’ is filled iteratively using subproblems, and the ‘best_order’ table is used to reconstruct the optimal path. The working of the algorithm is explained below:

1. 'euclidean_distance(point1, point2)': This is a helper function that calculates the Euclidean distance between two points represented as tuples (latitude, longitude).

2. 'tsp_dp(university, cluster)': This is the main function that takes the university coordinates and a cluster of customer locations as input and returns the efficient path and its total distance.

3. Append University: The university coordinates are appended to the 'cluster', as the TSP requires visiting the university at the end of the path.

4. Initialize Variables: The function initializes the DP table 'dp_table' and the 'best_order' table. These tables will store intermediate results to avoid redundant calculations and help reconstruct the optimal path later.

5. Base Case: The function sets the base case in the DP table. It initializes the distance from the university to all customer locations. For each location 'i', 'dp_table[1 << i][i]' is set to the Euclidean distance between the university and location 'i'.

6. Iterating Through Subproblems: The algorithm iterates through all subproblems using a binary mask representation. Each bit in the mask represents whether a customer location is visited (1) or not (0).

7. Fill DP Table: For each subproblem represented by the current mask, the algorithm considers all customer locations 'i' that are already visited in the mask. It then iterates through all possible next customer locations 'j', which are not yet visited. For each combination of 'i' and 'j', the algorithm calculates the distance to visit location 'j' from location 'i' and updates the 'dp_table' and 'best_order' if this path yields a shorter distance.

8. Reconstruct the Optimal Path: Once the DP table is filled, the algorithm uses the 'best_order' table to reconstruct the efficient path. It starts from the last node (university) and follows the pointers in the 'best_order' table to add customer locations to the path.

9. Calculate Total Distance: After reconstructing the path, the algorithm calculates the total distance for the efficient path by summing the Euclidean distances between consecutive customer locations and the university.

10. Return Efficient Path and Total Distance: The 'tsp_dp' function returns the efficient path and its total distance.

11. Sample Input and Output: The sample input with university coordinates and customer locations is provided, and the efficient path and its total distance are printed as output.

By using dynamic programming and cleverly breaking down the problem into subproblems, the algorithm avoids redundant calculations and finds the optimal or near-optimal path efficiently for small datasets.

Derivation of the time complexity:

The time complexity of the dynamic programming (DP) version of the Traveling Salesman Problem (TSP) algorithm is **O(2^n * n^2),** where n is the number of customer locations (including the university).

1. Filling the DP table 'dp_table': The algorithm fills the DP table for all possible subsets of customer locations. There are 2^n subsets of n customer locations, each represented by a binary mask. For each subset, the algorithm iterates through all n customer locations to calculate the distance and update the DP table. This takes O(n) time for each subset.

2. Overall, the time complexity for filling the DP table is O(2^n * n).

3. Reconstructing the path: After the DP table is filled, the algorithm reconstructs the efficient path by backtracking through the 'best_order' table. This requires traversing the 'best_order' table to find the optimal path, which takes O(n) time.

4. Calculating the total distance: The algorithm calculates the total distance for the efficient path by summing the distances between consecutive customer locations and the university. This requires iterating through the path, which takes O(n) time.

5. Overall, the time complexity for reconstructing the path and calculating the total distance is O(n).

6. Therefore, the overall time complexity of the TSP DP algorithm is **O(2^n * n^2).**

The TSP DP algorithm is significantly more efficient than the brute force approach, which has a time complexity of O(n!).

However, as the number of customer locations grows, the TSP DP algorithm still becomes computationally expensive due to the exponential growth in the number of subsets (2^n) to explore. It remains suitable for small datasets with a limited number of customer locations (clusters with <= 10 data points).

In the case of TSP-DP the time complexity becomes O(2^n * n^2), which makes it computationally infeasible for TSP instances with a large number of cities. It becomes impractical for more than a dozen cities and the TSP dynamic programming algorithm requires storing a large table of subproblem solutions, leading to significant memory consumption for large TSP instances. As we are trying to balance between finding the shortest route and reducing the computing resources we have to find an algorithm that satisfies both these requirements, in this case, we will be using the “Nearest Neighbors” algorithm for our project.

Run time difference between Brute force and DP:

The python implementation of both the methods are submitted as separate files. To understand the difference between these two methods, we calculated the runtime for both the algorithm with sample input and the results are attached below:

```
In [75]: runfile('E:/MSCS/Algorithms/Final project/Files/brute_force.py', wdir='E:/MSCS/Algorithms/Final project/Files')
Time taken: 0.018451452255249023 seconds
Optimal Path using Brute force method: ((42.401119, -71.104698), (42.349971, -71.118186), (42.330901, -71.136517), (42.312014,
-71.109233), (42.310632, -71.107837), (42.328841, -71.097946), (42.3398, -71.0903))
Total Distance: 0.14858056218341445

In [76]: runfile('E:/MSCS/Algorithms/Final project/Files/tsp_dp_final.py', wdir='E:/MSCS/Algorithms/Final project/Files')
Time taken: 0.0 seconds
Efficient Path using DP: [(42.3398, -71.0903), (42.310632, -71.107837), (42.312014, -71.109233), (42.330901, -71.136517), (42.349971,
-71.118186), (42.401119, -71.104698), (42.328841, -71.097946), (42.3398, -71.0903)]
Total Distance: 0.23448537461890892
```

Fig 6: Run time of both algorithms.

As you can see the time taken by DP is zero because the execution time is very short for ‘time.time()‘ function in python to capture the runtime. But as mentioned earlier, DP approach will only give sub-optimal results whereas Brute force is guaranteed to give optimal results but with a higher runtime than DP. So, if we want to scale up the algorithm by increasing the dataset, the time complexity of brute force algorithm will be very large when compared to DP.

When it comes to choosing between DP and Brute force, it is better to choose DP approach. However, both fail when the dataset is huge, we need to find an algorithm that is efficient and provides an optimal or sub-optimal solution hence NN was chosen to be implemented in our project.

Nearest Neighbors(NN)

The Nearest Neighbor Algorithm is a heuristic method used to find a solution to the Traveling Salesman Problem (TSP). The TSP is an optimization problem in which a salesperson must visit a given set of cities exactly once and return to the starting city, aiming to find the shortest possible route that visits all cities.

The Nearest Neighbor Algorithm starts with an arbitrary city (the starting city) and follows a greedy approach to construct the tour. At each step, it selects the nearest unvisited city from the current city and moves to that city. The process continues until all cities are visited, and then the salesperson returns to the starting city to complete the tour.

- Pros:
    - The Nearest Neighbor Algorithm is simple and easy to implement. It doesn't require complex mathematical computations or data structures.
    - The algorithm is relatively fast, especially for small to medium-sized instances of the TSP. Its time complexity is O(n^2), where n is the number of cities.

    - The algorithm only requires storing the current and remaining unvisited cities' distances, leading to minimal memory consumption.
- Cons:
    - **Not always globally optimal:** The Nearest Neighbor Algorithm does not guarantee finding the optimal solution for the TSP. Its greedy nature can lead to suboptimal tours, especially for instances with many cities.

As we are trying to find the optimal algorithm that balances finding the shortest distance and reducing the usage of resources we can use the Nearest Neighbors algorithm to find the shortest distance between the nodes.

Here is the algorithm for nearest neighbors:

- Input: Given a set of n cities (C) and a distance matrix (D) representing the distances between all pairs of cities.
- Initialization: Start from an arbitrary city as the current city (the starting city). Let the current tour be empty, and the current city's index be currentCityIndex.
- Tour Construction:
    - Add the currentCity to the current tour.
    - Mark currentCity as visited.
    - Find the nearest unvisited city (not in the current tour) to currentCity. This can be done by finding the city with the smallest distance value in the row corresponding to currentCity in the distance matrix D.
    - Set the nearest unvisited city as the new currentCity.
    - Repeat the above two steps until all cities are visited.
- Completing the Tour: Once all cities are visited, return to the starting city to complete the tour. Add the starting city to the current tour, closing the loop.
- Output: The current tour obtained from the algorithm represents an approximate solution to the TSP.

Derivation of Time complexity:

The time complexity of this algorithm will be:

- To initialize the variable the time complexity will be O(1).
- Main While Loop (num_cities - 1 iterations): O(num_cities)
- Finding the Nearest City (in each iteration): O(num_cities)
- Updating Tour and Unvisited Cities (in each iteration): O(1)
- Return to Starting City: O(1)

The total time complexity will be:

O(1) + O(num_cities) * O(num_cities) + O(num_cities) + O(1) + O(1)

As the dominant term here is O(num_cities^2), that will be its time complexity.

Other reviewed algorithms:

While analyzing multiple algorithms that suits our problem, we also understood why other algorithms like Floyd-Warshall, D* and A* are not suitable for this problem.

1. Floyd-Warshall Algorithm:

- The Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a graph.
- It is designed for problems with a single source and destination, where you need to find the shortest paths between all pairs of vertices.
- In our problem, the objective is to find a single path that visits all customer locations exactly once and returns to the university. Floyd-Warshall does not consider the constraint of visiting all locations only once, making it unsuitable for this specific problem.

2. D* Algorithm:

- The D* algorithm is a popular pathfinding algorithm used for mobile robots or agents in environments with dynamic changes.
- It relies on the concept of incremental search, where it updates the path as new information is obtained during exploration.
- In our problem, there are no dynamic changes in the environment, and the goal is to find an optimal path that visits all customer locations and returns to the university in a single go. D* is not designed for this kind of static path planning problem and would be overcomplicated for this use case.

3. A* Algorithm:

- The A* algorithm is a widely used pathfinding algorithm that finds the shortest path from a starting node to a goal node in a graph or grid.
- While A* could potentially be used for our problem, it would require a modified heuristic and graph representation to consider visiting all customer locations and returning to the university as part of the goal.
- Moreover, A* may not guarantee finding the globally optimal solution for the Traveling Salesman Problem (TSP), and it can be computationally expensive for large datasets.

In summary, the Floyd-Warshall algorithm is not suitable because it finds the shortest paths between all pairs of vertices and doesn't account for the constraints of our problem. The D* algorithm is not necessary as our problem doesn't involve dynamic changes in the environment. While A* could be adapted, the Traveling Salesman Problem requires specialized algorithms like the TSP DP approach to find the optimal or near-optimal solution for visiting all locations exactly once and returning to the university.

Project Implementation

We have created a model of our project in Python.

i. Generate Random points within the location of Boston.
ii. Form clusters with points that are close using the **K-means clustering** algorithm**.**
iii. Use **Nearest Neighbours** algorithm to find the shortest path between them.

Here is the code for our project:

1) Modules we are using for our project.

```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from math import radians, cos, sin, asin, sqrt
```

Fig 7 Libraries used

2) Co-ordinate Generator Function.
This function is used to generate random coordinates in the city of Boston which will be used as input. The boundary conditions are set within the radius of Boston to get the random points inside the city of Boston.

```
def coordinate_generator():
    """
    This function generates the coordinates in random as input.

    Parameters:
        None

    :return: coordinates
    """
    # Coordinates for Boston
    min_latitude = 42.2279
    max_latitude = 42.4061
    min_longitude = -71.1912
    max_longitude = -70.9865
    num_clusters = 7
```

Fig 8 Generating sample co-ordinates

```
    #Converting the min and max latitude to an area with length and height
    min_length = 0
    min_width = 0
    max_length = (max_latitude - min_latitude) * 69  #As one degree is 69 miles
    max_height = (max_longitude - min_longitude) * 69

    num_points = 60
    np.random.seed(16)
    latitudes = np.round(np.random.uniform(min_length, max_length, num_points), 6)
    longitudes = np.round(np.random.uniform(min_width, max_height, num_points), 6)
    coordinates = np.column_stack((latitudes,longitudes))
    K_Means(coordinates,num_clusters,longitudes,latitudes)
```

Fig 9 Generating sample co-ordinates

3) K_Means.

This function is used to cluster all the points which are close into one cluster so that we can use the shortest path algorithm in each of these clusters.

```
def K_Means(coordinates,num_clusters,longitudes,latitudes):
    """
    This function takes the input parameters from the previous code and clusters all the coordinates which are together

    :param coordinates : Random generated coordinates.
    :return: value,cluster_number
    """
    kmeans = KMeans(n_clusters=num_clusters)
    cluster_assignment = kmeans.fit_predict(coordinates)
    cluster_labels = kmeans.labels_
    cluster_centers = kmeans.cluster_centers_
    cluster_counts = {i: 0 for i in range(num_clusters)}

    modified_latitudes = []
    modified_longitudes = []

    # Loop through the data points and assign them to clusters with a maximum of 10 points
    for i, (lat, lon) in enumerate(zip(latitudes, longitudes)):
        cluster_id = cluster_labels[i]
        if cluster_counts[cluster_id] < 10:
            modified_latitudes.append(lat)
            modified_longitudes.append(lon)
            cluster_counts[cluster_id] += 1

    # Convert the modified latitudes and longitudes back to arrays
    modified_coordinates = np.column_stack((modified_latitudes, modified_longitudes))
```

Fig 10 Using K means clustering

```python
# Convert the modified latitudes and longitudes back to arrays
modified_coordinates = np.column_stack((modified_latitudes, modified_longitudes))

# Initialize a new KMeans model for the modified data
kmeans_modified = KMeans(n_clusters=num_clusters)

# Fit the modified model to the data
kmeans_modified.fit(modified_coordinates)

# Get the modified cluster labels and centers
cluster_labels_modified = kmeans_modified.labels_
cluster_centers_modified = kmeans_modified.cluster_centers_

cluster_coordinates = {i: [] for i in range(num_clusters)}

# Populate the dictionary with coordinates
for i, (lat, lon) in enumerate(zip(modified_latitudes, modified_longitudes)):
    cluster_id = cluster_labels[i]
    if len(cluster_coordinates[cluster_id]) < 10:
        cluster_coordinates[cluster_id].append((lat, lon))

cluster_number = 0
for key, val in cluster_coordinates.items():
    cluster_number = cluster_number + 1
    Distance = matrix_distance(val, cluster_number)
```

Fig 11 Using K means clustering II

4) matrix_distance

In each of these clusters, we should find the distance between these points and have them in the form of a symmetric matrix. This function is used to generate a symmetric matrix.

```python
def matrix_distance(value,cluster_number):
    """
    To create a symmetric matrix that calculates the distance between each nodes in a cluster.
    :param value: A cluster with nodes
    :param cluster_number: Which cluster number we are referring
    :return: dist_matrix, cluster_number
    """
    dist_matrix = np.zeros((len(value), len(value)), dtype=int)

    for i in range(len(value)):
        for j in range(i+1,len(value)):
            x1,y1 = value[i]
            x2,y2 = value[j]
            distance = sqrt((x2 - x1)**2 + (y2 - y1)**2)
            dist_matrix[i][j] = distance * 1000
            dist_matrix[j][i] = distance * 1000
    close_neighbours = nearest_neighbor_tsp(dist_matrix,cluster_number)
```

Fig 12 Creating Distance matrix

5) “nearest_neighbor_tsp” and “main” function

This function is used to find the nearest neighbor of each point by using the distance_matrix. This part also contains the main function which is used to call the other functions.

```
def nearest_neighbor_tsp(distance_matrix,cluster_number):
    """
    This function is used to find the nearest neighbor for a cluster
    :param distance_matrix: Symmetric matrix containing the distance between each points
    :param cluster_number: Cluster number
    :return: None
    """
    num_nodes = len(distance_matrix)
    unvisited_nodes = set(range(1, num_nodes))  # Excluding the starting node (index 0)
    tour = [0]  # Start the tour from node 0
    current_node = 0
    total_distance = 0

    while unvisited_nodes:
        nearest_neighbor = min(unvisited_nodes, key=lambda node: distance_matrix[current_node][node])
        total_distance += distance_matrix[current_node][nearest_neighbor]
        tour.append(nearest_neighbor)
        unvisited_nodes.remove(nearest_neighbor)
        current_node = nearest_neighbor

    # Return to the starting node to complete the tour
    tour.append(0)
    print("For the cluster "+str(cluster_number)+ " the optimal route is "+str(tour)+"and the total distance covered is "+str(total_distance)+"miles")
```

Fig 13 Applying NN algorithm

6) Output

```
For the cluster 1 the optimal route is [0, 2, 6, 4, 1, 3, 5, 0]and the total distance covered is 8miles
For the cluster 2 the optimal route is [0, 4, 7, 3, 8, 1, 6, 5, 2, 9, 0]and the total distance covered is 12miles
For the cluster 3 the optimal route is [0, 2, 3, 7, 8, 4, 9, 1, 5, 6, 0]and the total distance covered is 8miles
For the cluster 4 the optimal route is [0, 1, 2, 3, 7, 4, 5, 6, 8, 0]and the total distance covered is 15miles
For the cluster 5 the optimal route is [0, 1, 2, 3, 5, 6, 4, 7, 0]and the total distance covered is 6miles
For the cluster 6 the optimal route is [0, 3, 2, 1, 4, 5, 0]and the total distance covered is 14miles
For the cluster 7 the optimal route is [0, 1, 3, 4, 2, 5, 0]and the total distance covered is 9miles
```

Fig 14 Output of the code

Conclusion

In this project, we aimed to enhance the efficiency of serving students or customers by leveraging clustering and path planning algorithms. We designed two algorithms that effectively utilize geographical data (latitude and longitude) to achieve this objective.

The first algorithm focused on intelligent clustering of students/customers. By employing clustering algorithms like K-means, we grouped individuals with similar geographical locations into clusters. This allowed for optimized resource allocation and minimized travel distances for service providers. Clustering enabled us to identify geographical concentrations, facilitating efficient deployment of services while reducing the overall travel requirements.

The second algorithm tackled the challenge of optimizing travel routes. By considering addresses and geographical coordinates, we employed path planning techniques like the Traveling Salesman Problem (TSP) solver. This algorithm determined the most optimal sequence of destinations to minimize travel time and distance. It aimed to ensure that service providers follow a logical route that reduces backtracking and optimizes their journeys, leading to quicker service delivery.

It's important to acknowledge the project's limitations. Firstly, the clustering and path planning algorithms presented are optimized for the specific geographical context of Boston. These algorithms might not perform as effectively in other regions with different geographical layouts and transportation patterns. Additionally, due to time constraints, the project does not include the development of a web application for real-time interaction. A web app would allow students or customers to input their information, visualize cluster assignments, and receive optimized routes dynamically. While these limitations are recognized, the project serves as a valuable demonstration of the potential benefits that intelligent clustering and path planning can offer to enhance efficiency in service delivery scenarios.

PROJECT SELF REFLECTIONS

1. Prithiv:

From this project, I was able to understand the practical applications of these algorithms. Initially, I was under the impression that algorithms with the best results will always be preferred over other factors but through this project, I was able to understand that aside from the results I was able to understand the importance of time complexity.

When we had to finalize the algorithm for finding the shortest path visiting all nodes, we chose nearest neighbors as even though theoretically the “Travelling Salesman Brute Force” problem always yielded the best result, it had the worst time complexity which made the algorithm obsolete for higher nodes. As we had to balance between the computational resource and yielding results, we found out nearest neighbors were able to balance both. Through this project, I was able to understand the practical constraints while implementing this algorithm and how to choose an algorithm taking all the parameters such as application and available resources into consideration.

2. Aishwarya:

Working on this project provided me with the opportunity to explore the practical applications of various clustering and path-finding algorithms. I gained a deep insight into the workings of these algorithms and their adaptability to address specific use cases. As I explored these algorithms, it helped me learn about how they perform in different scenarios and what their limitations are. This also enabled me to comprehend how these algorithms scale when dealing with larger datasets.

Having learnt about Kruskal's algorithm in the course, I had the chance to construct a minimum spanning tree for our particular use case. This allowed me to grasp the nuances of the algorithm and discover how modifications can be made to obtain the desired number of clusters or data nodes within each cluster. It also helped me understand the need for good representation and visualization of data, to extract meaningful information from it and analyse how the algorithm can be modified to achieve a specific task.

I also learned that the emphasis should be on finding the most effective approach to solve the problem at hand, which led me to explore other approaches and algorithms to solve the problem.

Overall, this hands-on experience has helped me develop the skills necessary to comprehend and apply various algorithms in diverse contexts, tailoring them to fit specific use cases and effectively apply them to real-world scenarios.

3. Haritha:

Participating in this clustering project has provided me with valuable insights into several crucial aspects of data analysis and algorithm application. One of the key takeaways is understanding the practical implementation of clustering algorithms like K-means in real-world scenarios involving geographical data. These algorithms allowed me to identify and group similar data points efficiently. Additionally, I realized the importance of visualizing and interpreting clustering results. Creating visual representations of clusters enabled me to comprehend the distribution and patterns within the data, which is essential for decision-making and strategy development.

Overall, this project provided hands-on experience in applying clustering algorithms to geographical data, selecting appropriate features, and gaining insights through result interpretation and visualization. These skills are invaluable for optimizing service delivery and resource allocation in real-world scenarios.

4. Ravi:

During the course of this project, I have acquired valuable insights into a wide range of path planning algorithms and their distinct applications. While crafting the TSP (Traveling Salesman Problem) algorithm, I conducted an in-depth exploration of various approaches, thoroughly analyzing their strengths and weaknesses to identify the most suitable one for our specific project requirements.

Additionally, I delved into the essential auxiliary factors that play a crucial role in designing a robust path planning algorithm. I focused on factors such as cluster size, the number of nodes within each cluster, and the appropriate representation of clusters for efficient path planning. Although these factors may not directly pertain to the path planning algorithm itself, their consideration becomes indispensable during the coding process to achieve optimal results in practical scenarios.

Moreover, I gained a profound understanding of the significant disparity in time complexity between Brute force and Dynamic Programming (DP) approaches. Despite the minor difference in runtime observed with our sample points, I acknowledged the potential impact of this distinction when dealing with extensive datasets. Furthermore, I acquired the ability to select an algorithm not solely based on its runtime but also based on its capability to provide the most optimal solution for the problem at hand.

In essence, working on this project has bestowed me with invaluable skills in implementing algorithms to address real-life challenges effectively. The knowledge and experiences gained during this endeavor will undoubtedly serve as a solid foundation for tackling various algorithmic problems in the future.

Appendix

MST – based clustering:

The python implementation of the MST based clustering is attached below:

```python
def build_connected_graph(data):
    num_points = len(data)
    graph = {(elem.latitude, elem.longitude): [] for elem in data}

    for i in range(num_points):
        for j in range(i + 1, num_points):
            weight = math.dist([data[i].latitude, data[j].longitude],[data[j].latitude, data[i].longitude])
            key_i = (data[i].latitude, data[i].longitude)
            key_j = (data[j].latitude, data[j].longitude)
            graph[key_i].append((key_j, weight))
            graph[key_j].append((key_i, weight))

    return graph

graph = build_connected_graph(data)
```

✓ 0.0s

Fig 15 MST based clustering I

```python
def find(parent, node):
    if parent[node] != node:
        parent[node] = find(parent, parent[node])
    return parent[node]

def union(parent, rank, node1, node2):
    root1 = find(parent, node1)
    root2 = find(parent, node2)

    if rank[root1] < rank[root2]:
        parent[root1] = root2
    elif rank[root1] > rank[root2]:
        parent[root2] = root1
    else:
        parent[root2] = root1
        rank[root1] += 1

def build_mst(graph):
    edges = []
    for node, neighbors in graph.items():
        for neighbor, weight in neighbors:
            edges.append((weight, node, neighbor))

    edges.sort()
    mst = []
    parent = {node: node for node in graph}
    rank = {node: 0 for node in graph}

    for weight, node1, node2 in edges:
        if find(parent, node1) != find(parent, node2):
            mst.append((node1, node2, weight))
            union(parent, rank, node1, node2)

    return mst

minimum_spanning_tree = build_mst(graph)
```

Fig 16 MST based clustering II

```python
# determine clusters
def dfs_recursive(graph, node, visited, nodes):
    if node not in visited:
        nodes.append(node)
        visited.add(node)
        for neighbor in graph[node]:
            dfs_recursive(graph, neighbor, visited, nodes)

def dfs(graph):
    clusters = {}
    count = 0
    visited = set()
    for node in graph:
        if node not in visited:
            nodes = []
            dfs_recursive(graph, node, visited, nodes)
            count += 1
            clusters[count] = nodes
    return clusters
```

Fig 17 MST based clustering III

```python
    # Cluster by removing k - 1 most expensive edges from the MST
    def kcluster(minimum_spanning_tree, k):
        # Sort the MST edges based on weights
        minimum_spanning_tree.sort(key=lambda x: x[2], reverse=True)

        node_cluster = {node: set() for node in graph.keys()}
        for u, v, weight in minimum_spanning_tree:
            node_cluster[u].add(v)
            node_cluster[v].add(u)

        clustered_nodes = {}
        for i in range(k - 1):
            node1, node2, weight = minimum_spanning_tree.pop(0)
            node_cluster[node1].remove(node2)
            node_cluster[node2].remove(node1)
        clustered_nodes = dfs(node_cluster)

        return clustered_nodes
```

Fig 18 MST based clustering IV

```python
def isToDeleteEdge(clustered_nodes):
    for value in clustered_nodes.values():
        if len(value) < 10:
            return False
    return True

# Cluster by removing edges that result in relatively equal number of nodes in each cluster
def cluster(minimum_spanning_tree, k):
    minimum_spanning_tree.sort(key=lambda x: x[2], reverse=True)
    print(minimum_spanning_tree)
    print(len(minimum_spanning_tree))
    node_cluster = {node: set() for node in graph.keys()}
    for u, v, weight in minimum_spanning_tree:
        node_cluster[u].add(v)
        node_cluster[v].add(u)

    clustered_nodes = {}
    edge = 0
    while edge < len(minimum_spanning_tree):
        node1, node2, weight = minimum_spanning_tree[edge]
        node_cluster[node1].remove(node2)
        node_cluster[node2].remove(node1)
        clustered_nodes = dfs(node_cluster)
        if isToDeleteEdge(clustered_nodes):
            minimum_spanning_tree.pop(edge)
        else:
            # retain the edge, move to the next one
            edge += 1
            node_cluster[node1].add(node2)
            node_cluster[node2].add(node1)
            continue
        if len(clustered_nodes) == k:
            break
    clustered_nodes = dfs(node_cluster)
    return clustered_nodes
```

Fig 19 MST based clustering V

```python
clustered_nodes = cluster(minimum_spanning_tree, 6)
minimum_spanning_tree = build_mst(graph)
k_clustered_nodes = kcluster(minimum_spanning_tree, 6)
```

Fig 20 MST based clustering V

TSP – Brute Force Approach:

The pseudocode for the TSP Brute force algorithm is attached below:

```
function euclidean_distance(point1, point2):
    Calculate the Euclidean distance between point1 and point2

function tsp_brute_force(university, cluster):
    Append the university coordinates to the cluster

    best_distance = INFINITY
    best_path = []

    function permute(locations):
        if all locations are visited:
            current_distance = 0
            for i from 0 to number of locations - 1:
                current_distance += euclidean_distance(locations[i], locations[i + 1])
            if current_distance < best_distance:
                best_distance = current_distance
                best_path = copy of locations
        else:
            for i from 1 to number of locations - 1:
                swap locations[1] and locations[i]
                permute(locations)
                swap locations[1] and locations[i]

    permute(cluster)

    return best_path, best_distance

# Sample input
university = (42.3398, -71.0903)  # University coordinates (latitude, longitude)
cluster = [
    (42.310632, -71.107837),
    (42.328841, -71.097946),
    (42.312014, -71.109233),
    (42.349971, -71.118186),
    (42.330901, -71.136517),
    (42.401119, -71.104698)
]

best_path, best_distance = tsp_brute_force(university, cluster)
print("Best Path:", best_path)
print("Best Distance:", best_distance)
```

Fig 21 Pseudocode for TSP Brute force

The time complexity of the brute force method for the Traveling Salesman Problem (TSP) is **O(n!),** where n is the number of customer locations (including the university).

The python implementation of the TSP Brute force code is attached below:

```python

import itertools
import math
import time

#def calculate_distance(loc1, loc2):
#    return dist(loc1, loc2)

def euclidean_distance(point1, point2):
    lat1, lon1 = point1
    lat2, lon2 = point2
    return math.sqrt((lat2 - lat1)**2 + (lon2 - lon1)**2)

def find_optimal_path(university, cluster):
    start_time = time.time()
    cluster.append(university)
    min_distance = float('inf')
    optimal_path = []

    # Generate all permutations of customer locations
    for path_permutation in itertools.permutations(cluster):
        total_distance = 0

        # Calculate the total distance for the current permutation
        for i in range(len(path_permutation) - 1):
            total_distance += euclidean_distance(path_permutation[i], path_permutation[i + 1])

        # Add the distance from the last customer location back to the university
        total_distance += euclidean_distance(path_permutation[-1], university)

        # Update the optimal path if the current permutation yields a shorter distance
        if total_distance < min_distance:
            min_distance = total_distance
            optimal_path = path_permutation

    end_time = time.time()
    time_taken = end_time - start_time
    print("Time taken:", time_taken, "seconds")

    return optimal_path

# Sample input
university = (42.3398, -71.0903)  # University coordinates (latitude, longitude)
cluster = [
    (42.310632, -71.107837),
    (42.328841, -71.097946),
    (42.312014, -71.109233),
    (42.349971, -71.118186),
    (42.330901, -71.136517),
    (42.401119, -71.104698)
]

optimal_path = find_optimal_path(university, cluster)
print("Optimal Path using Brute force method:", optimal_path)


total_distance = 0
for i in range(1, len(optimal_path)):
    total_distance += euclidean_distance(optimal_path[i-1], optimal_path[i])

print("Total Distance:", total_distance)
```

Fig 22 Python implementation of TSP Brute force

TSP – Dynamic Programming version

The pseudocode for the TSP DP algorithm is attached below:

```
function euclidean_distance(point1, point2):
    Calculate the Euclidean distance between point1 and point2

function tsp_dp(university, cluster):
    Append the university coordinates to the cluster
    n = number of customer locations (including the university)

    Initialize a 2D DP table dp_table of size (2^n) x n with all entries set to -1
    Initialize a 2D best_order table of size (2^n) x n with all entries set to -1

    # Base case: distance from university to all customers
    for i from 0 to n - 1:
        dp_table[1 << i][i] = euclidean_distance(university, cluster[i])

    # Iterate through all subproblems
    for mask from 1 to (2^n) - 1:
        for i from 0 to n - 1:
            if mask & (1 << i) != 0:
                for j from 0 to n - 1:
                    if i != j and dp_table[mask][i] != -1:
                        new_mask = mask | (1 << j)
                        if dp_table[new_mask][j] == -1 or dp_table[new_mask][j] > dp_table[mask][i] + euclidean_distance(cluster[i], cluster[j]):
                            dp_table[new_mask][j] = dp_table[mask][i] + euclidean_distance(cluster[i], cluster[j])
                            best_order[new_mask][j] = i

    # Find the optimal path using the DP table and best_order
    mask = (2^n) - 1
    last_node = -1
    for i from 0 to n - 1:
        if best_order[mask][i] != -1:
            last_node = i
            break

    path = [university]
    while last_node != -1:
        path.append(cluster[last_node])
        new_mask = mask ^ (1 << last_node)
        last_node = best_order[mask][last_node]
        mask = new_mask

    # Calculate the total distance for the efficient path
    total_distance = 0
    for i from 1 to length(path) - 1:
        total_distance += euclidean_distance(path[i-1], path[i])

    return path, total_distance

# Sample input
university = (42.3398, -71.0903)  # University coordinates (latitude, longitude)
cluster = [    (42.310632, -71.107837),    (42.328841, -71.097946),    (42.312014, -71.109233),    (42.349971, -71.118186),    (42.330901, -71.136517),    (42.401119, -71.104698)]

efficient_path, total_distance = tsp_dp(university, cluster)
print("Efficient Path:", efficient_path)
print("Total Distance:", total_distance)
```

Fig 23 Pseudocode for TSP DP approach

The time complexity of the dynamic programming (DP) version of the Traveling Salesman Problem (TSP) algorithm is **O(2^n * n^2),** where n is the number of customer locations (including the university).

The python implementation of the TSP DP algorithm is attached below:

```python
import math
import time


def euclidean_distance(point1, point2):
    lat1, lon1 = point1
    lat2, lon2 = point2
    return math.sqrt((lat2 - lat1)**2 + (lon2 - lon1)**2)

def tsp_dp(university, cluster):
    start_time = time.time()
    cluster.append(university)
    n = len(cluster)

    # Initialize the DP table with -1 (unvisited)
    dp_table = [[-1] * n for _ in range(1 << n)]
    best_order = [[-1] * n for _ in range(1 << n)]

    # Initialize the base case: distance from university to all customers
    for i in range(n):
        dp_table[1 << i][i] = euclidean_distance(university, cluster[i])

    # Iterate through all subproblems
    for mask in range(1, (1 << n)):
        for i in range(n):
            if mask & (1 << i) != 0:
                for j in range(n):
                    if i != j and dp_table[mask][i] != -1:
                        new_mask = mask | (1 << j)
                        if dp_table[new_mask][j] == -1 or dp_table[new_mask][j] > dp_table[mask][i] + euclidean_distance(cluster[i], cluster[j]):
                            dp_table[new_mask][j] = dp_table[mask][i] + euclidean_distance(cluster[i], cluster[j])
                            best_order[new_mask][j] = i

```

Fig 24 Python code for TSP DP algorithm

```python
                            dp_table[new_mask][j] = dp_table[mask][i] + euclidean_d
                            best_order[new_mask][j] = i

    # Find the optimal path using the DP table and best_order
    mask = (1 << n) - 1
    last_node = -1
    for i in range(n):
        if best_order[mask][i] != -1:
            last_node = i
            break

    path = [university]
    while last_node != -1:
        path.append(cluster[last_node])
        new_mask = mask ^ (1 << last_node)
        last_node = best_order[mask][last_node]
        mask = new_mask

    # Calculate the total distance for the efficient path
    total_distance = 0
    for i in range(1, len(path)):
        total_distance += euclidean_distance(path[i-1], path[i])

    end_time = time.time()
    time_taken = end_time - start_time
    print("Time taken:", time_taken, "seconds")

    return path, total_distance

# Sample input
university = (42.3398, -71.0903)  # University coordinates (latitude, longitude)
cluster = [
    (42.310632, -71.107837),
    (42.328841, -71.097946),
    (42.312014, -71.109233),
    (42.349971, -71.118186),
    (42.330901, -71.136517),
    (42.401119, -71.104698)
]

efficient_path, total_distance = tsp_dp(university, cluster)
print("Efficient Path using DP:", efficient_path)
print("Total Distance:", total_distance)

```

Fig 25 Python code for TSP DP Algorithm II