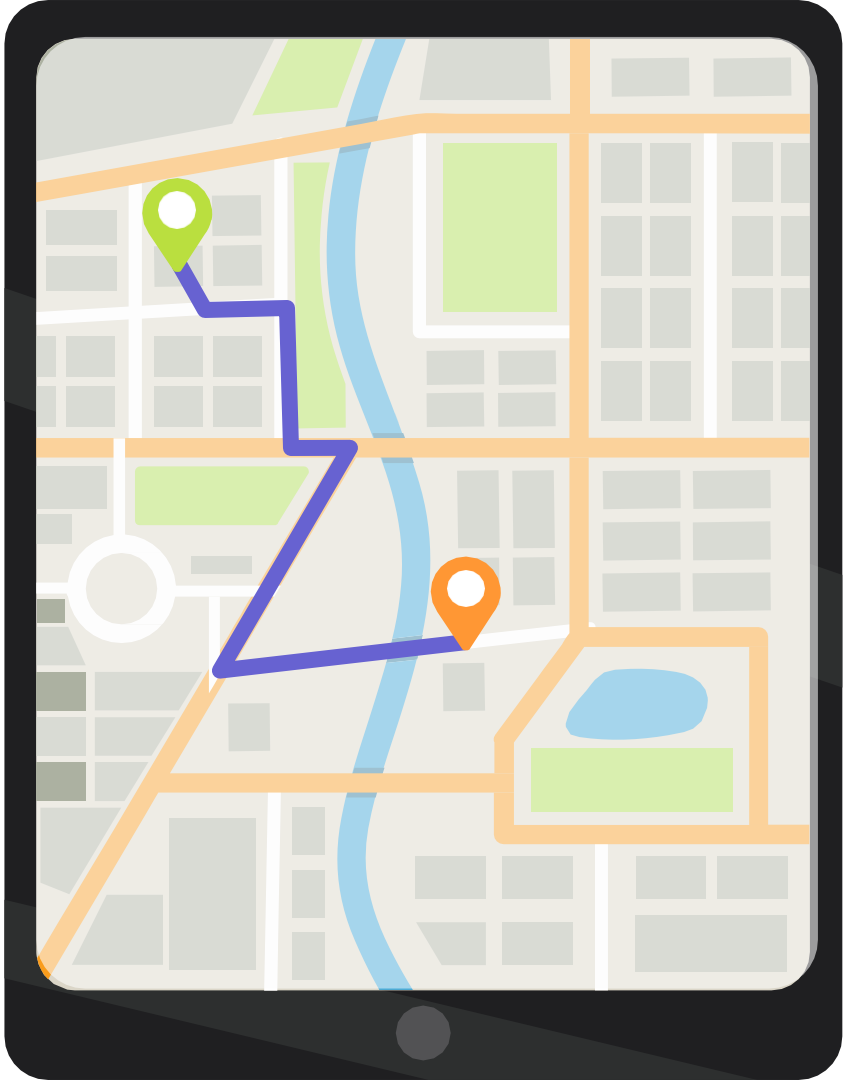
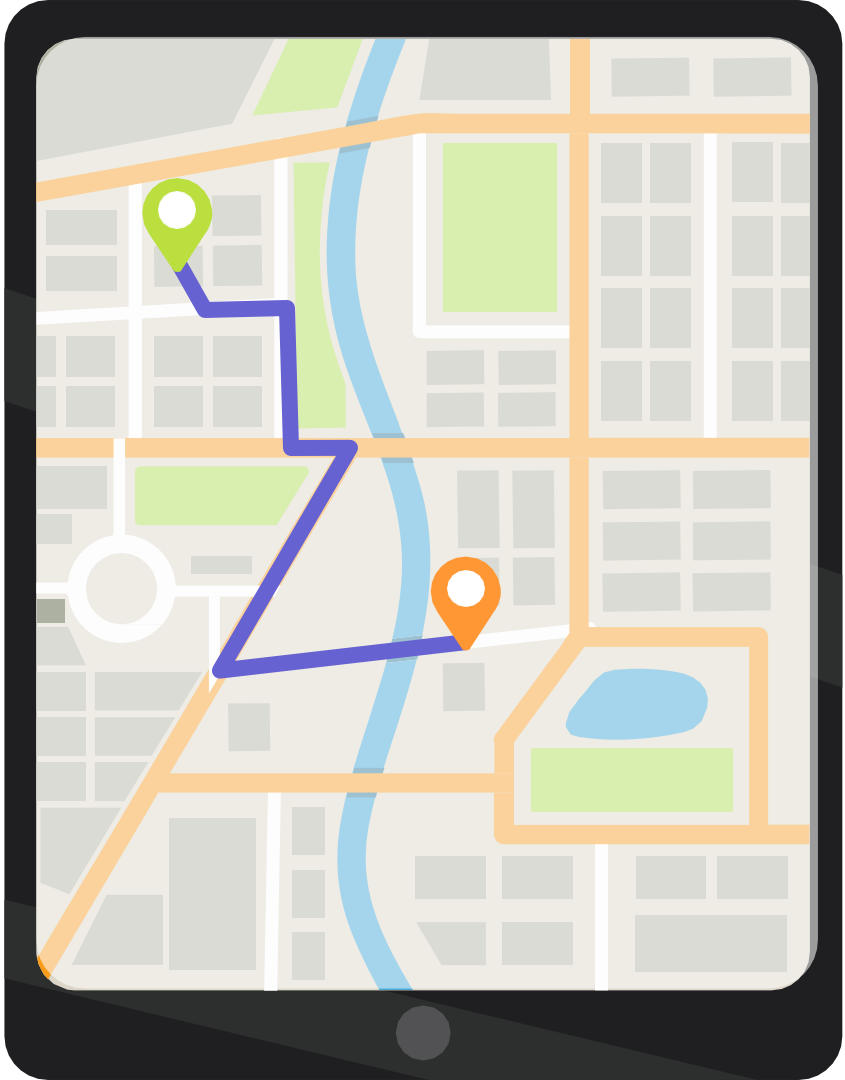
Trajectories Case Study



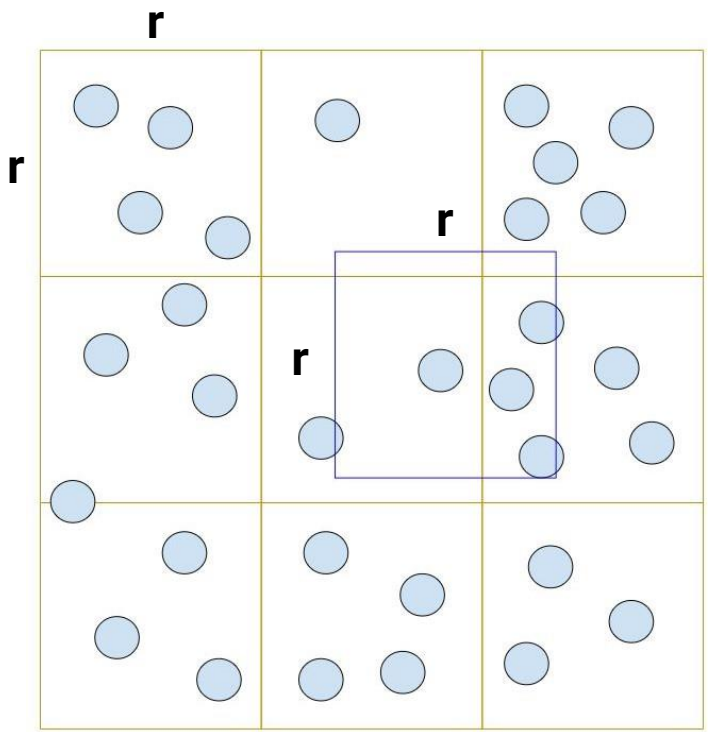
Task 1

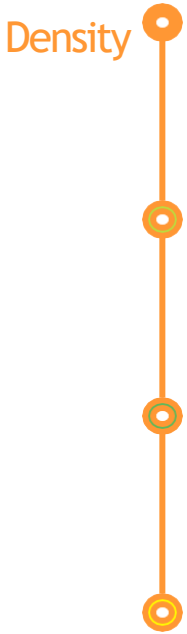
Hub Identification



Explanation of Density Function

r = 0.1





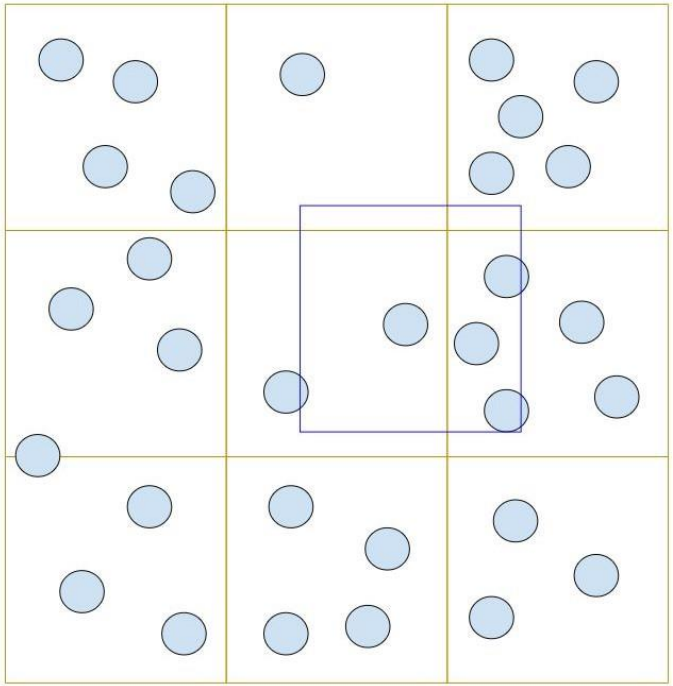
Defined density as the number of points within an r x r square centered at desired point (x, y)

The density function used in this script is a simple fixed-radius density estimation function. It takes as input a 2D grid, which is essentially a partitioning of the space into cells that contain their respective (x, y) coordinates. It returns the number of points within a fixed distance r of those coordinates

Specifically, the function considers all cells that are within a square of side length 2r Centered at the input coordinate, iterating over a finite area that includes all neighbors, and counting points within the r * r finite area

Runtime: O(1)
Parameters: grid, x, y

Reasoning and Set of Parameters Used



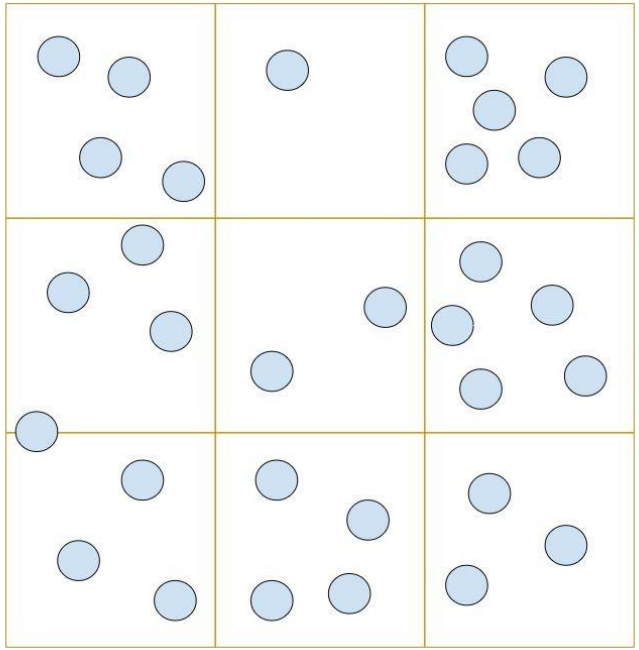
- This density function is simple and easy to implement. It creates a finite area to loop over with easy O(1) access in comparison to spatial-oriented data structures like kb-trees
- It is also effective in identifying clusters of points that can be closely located together. However, it may not perform well in cases where the density of points varies widely across the space, as the performance decays for larger square sizes or less distributed point values
- **Reasoning for R = 0.1:**
  - Capture fine-grained patterns with smaller neighborhoods – data went to 6 decimal places
  - Did not fully minimize for runtime and memory. The smaller the R, the more memory and time required to initialize the grid that spanned the 2D plane
- The parameters for density consist of the pre-processed grid (see next slide), the x value of the desired point (float), and the y value of the desired point (float)

Example Density Calculation

Assume you have the points: [(0, 0), (1, 1), (2, 1), (3, 2), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9, 9)] and r = 1. Say we want to calculate density of (1, 2).

- The preprocess_grid() creates a grid structure that divides the 2D space into cells of size 1 and places each point into the corresponding cell based on its x and y coordinates.
- The minimum x and y values are 0, and the maximum x and y values are 9: minimum x and y values are 0, and the maximum x and y values are 9. Points are populated into their respective boxes depending on their (x, y) coordinates.
- After grid creation, we can use the density() function to see which grid cell the point (x, y) falls into. The x and y coordinates are (x=1, y=2), so the point falls into the cell at position (1, 2) in the grid.
- It looks at the current cell, as well as all neighboring cells at cell positions (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2), (0, 3), (1, 3), and (2, 3).
- The function would count the number of points within the radius of 1 within these neighboring cells and return the number of points within a radius of 1 of (1, 2).
- In this case the point (1, 1) is the only one, so density would return just 1.

Preprocessing Algorithm



Runtime: O(N) with respect to number of points N: Only iterate over all points twice with O(1) arithmetic

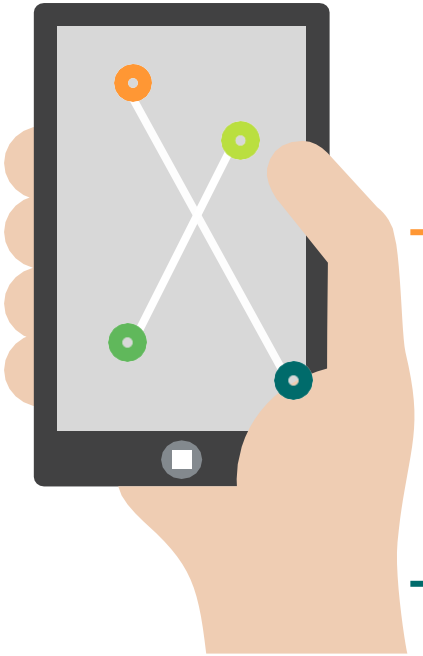
Partitions the space into a 2D grid and assigning each point to a cell in the grid. The algorithm takes as input the list of points, and the fixed radius parameter r that defines the size of the cells

Uses min/max of x/y to determine the size of the grid. Then calculates the number of cells needed in the x and y directions by dividing the range of x-y coordinates by the cell size r

The algorithm then creates an empty 2D list with dimensions equal to the number of cells in the x and y directions, and assigns each point to the cell that contains it.

Calculates the cell indices of each point by dividing its x-y coordinates by r, rounding down to the nearest integer, and using the resulting indices to index into the 2D list and append the point to the list at that index

hubs(P;k;r) Procedure and Runtime



Initialize the centers using K-means++ algorithm to select k initial centers that are far apart from each other

Loop until valid centers are found. Assign each point to the closest center, creating k clusters. For each cluster, consider each point in the cluster as a candidate for the new center

To choose the possible points, compute the distance between each point in the cluster. For each candidate center, check if the new center would be valid (distance between any two centers is at least r)

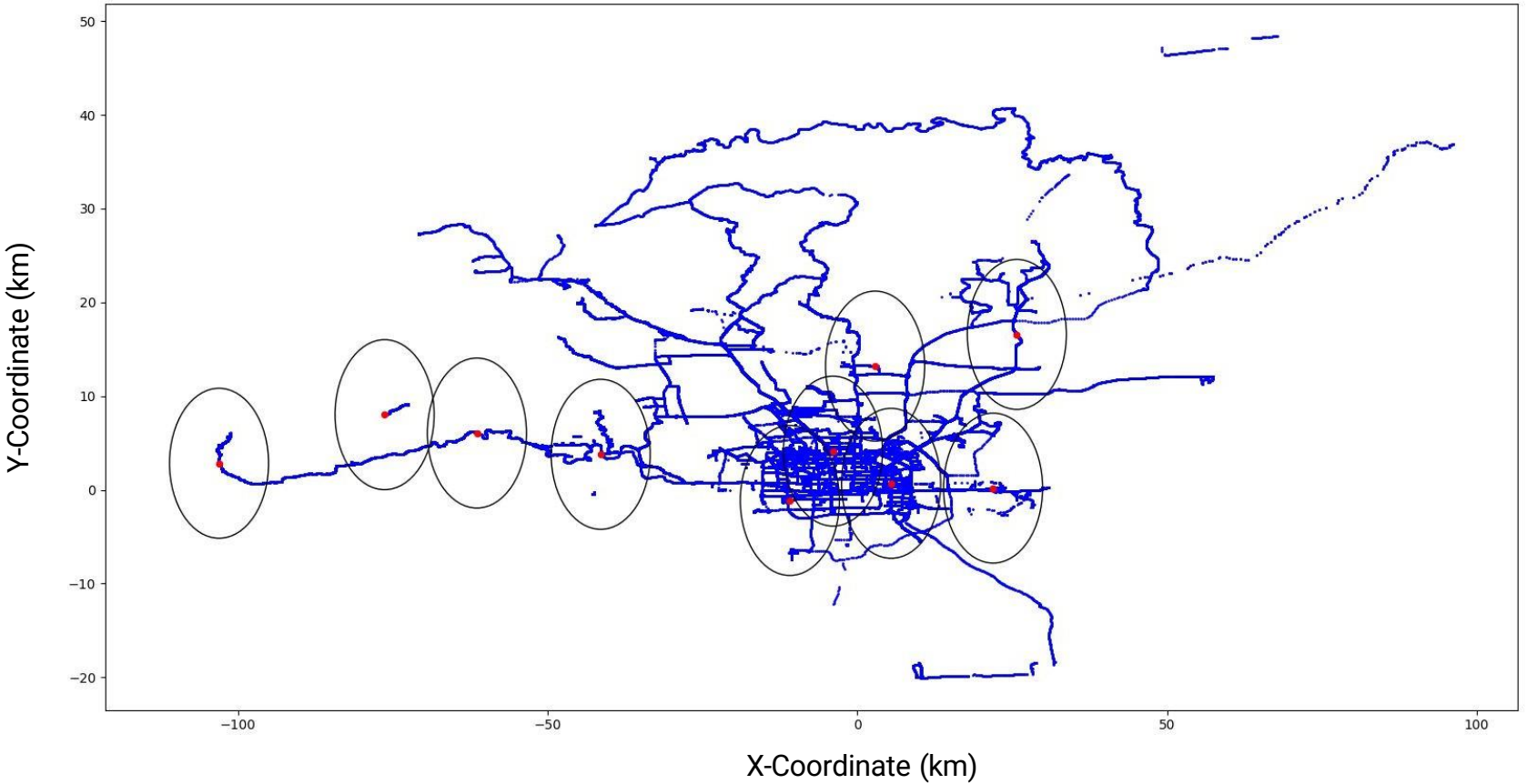
If the new center is valid, update the center. Check if the centers are valid (distance between any two centers is at least r)

Runtime: O(n) with respect to number of input points P. Has polynomial dependencies however, on number of hubs k and some dependence on convergence time T (excluded)

K = 10
R = 8 km

Experimental Results (Section 3.2)

Hub Locations (and a defined Radius) Among the Full Dataset



Runtime Analysis - Run on Full Dataset

K (hubs)	R (kms)	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)
5	2	2212.12	2265.00	2230.43
10	2	5829.01	5955.02	5862.33
20	2	17255.50	33499.32	18003.00
40	2	111385.00	408395.03	351948.10

K = 40 will run. But may take time as runtime can grow quickly as number of hubs k grows

Runtime Analysis Part II - Varying Input Data

K (Hubs)	r (km)	10% (ms)	30% (ms)	60% (ms)	Full (ms)
10	8	7158.23	7820.01	6720.21	7292.21

Our hubs() function is not dependent on preprocessing nor density(). Runtimes are measured from the execution of hubs

Advantages, Disadvantages, and Future Improvements

Advantages:

- The Hubs algorithm can effectively identify top k dense hubs in a dataset.
- It is relatively simple to implement and does not depend on density or preprocessing functionality
- Can act on unsorted or unmanipulated data.
- Randomization can at times can improve diversity of centers

Disadvantages:

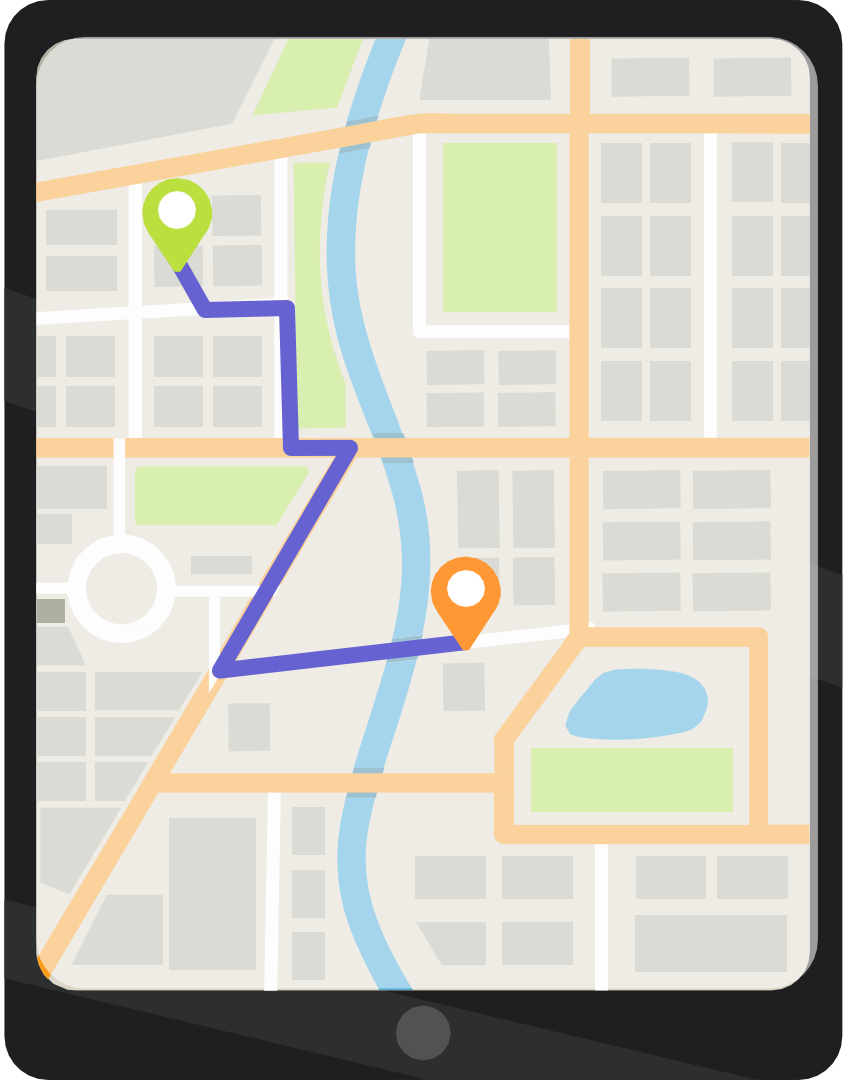
- The algorithm assumes that clusters are defined by the density of points around a center – this is not necessarily true for all datasets
- The algorithm may not work well for trajectories with irregular structures.
- The algorithm is sensitive to the choice of the number of hubs and its runtime can grow at a rather large rate due to the runtime being polynomial with respect to k.

Future Improvements:

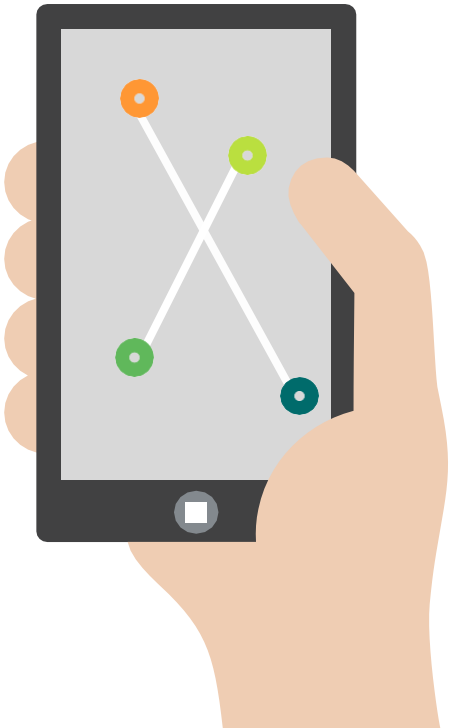
- One way to improve the Hubs algorithm is to use more sophisticated methods for center selection and initialization. Maybe use less randomization and utilize the density algorithm so hubs can remain static on subsequent runs
- Possibly using estimation as a way to identify the hubs could help runtime. We could use a grid structure and ultimately group each point into a grid (similar to our own preprocessing). We base the hub densities via these grid densities. The count for the density would be the number of points in the grid.

Task 2

Trajectory Simplification



Proposed Trajectory Simplification Algorithm TS-greedy



minDistanceFromPoint & TS_greedy

- To calculate the minimum distance from a point, a separate class was created called Segment to define a segment line. Inside this class, minDistanceFromPoint is used to calculate the minimum distance from the segment (the class itself) to a point q. It then returns the minimum distance. Furthermore, any methods which pertain to calculating segment values are also in the Segment class (example dist, which returns the length of the segment).

- The TS Greedy algorithm uses recursion to divide the trajectory into simpler line segments until the error of these new subdivisions is within the error of epsilon (divide and conquer). This program runs in O(|N|log|N|) time complexity where N is a trajectory of points.

Breakdown of TS Greedy program

Step 1

Make line segment formed by the first and last points of the trajectory

Step 2

Finds the point farthest from the above line segment

Step 3

If point is within the error (epsilon) return endpoints of trajectory, else continue.

Step 4

Split into two sub-trajectories at farthest point and recursively call the TS Greedy on both

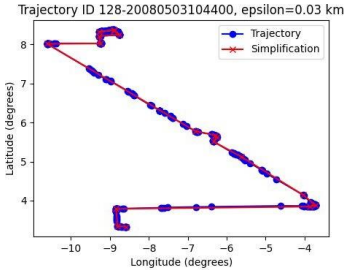
Step 5

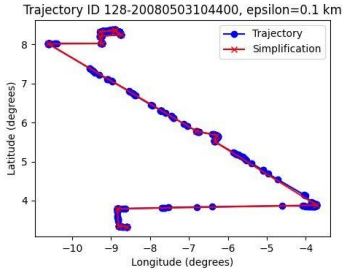
Combine two simplified sub-trajectories by concatenating the left and the right sub-trajectories

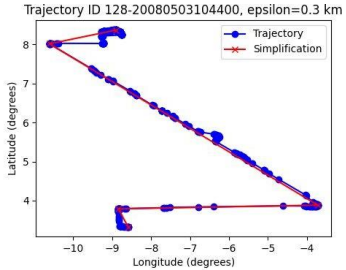
Step 6

Repeat this until all the trajectories are within the error range

Experimental Results (Section 4.2)







Experimental Results (Section 4.2)

	Original Points	Simplified Points	Compression Ratio
Trajectory ID: 128-20080503104400	321	18	17.83
Trajectory ID: 010-20081016113953	543	11	49.36
Trajectory ID: 115-20080520225850	1330	37	35.95
Trajectory ID: 115-20080615225707	1442	38	37.95

Advantages, Disadvantages, and Future Improvements

Benefits

1) With divide and conquer the algorithm has a time complexity of O(NlogN), which is faster than a simple Greedy which is O(N^2).

2) The error tolerance parameter epsilon to control the amount of simplification gives users more control over the output.

Drawbacks

1) The algorithm is sensitive to noise in the trajectory data, as a few outlier points can cause the algorithm to subdivide the trajectory excessively and produce a highly simplified output.

2) By using Divide and Conquer, it uses more memory usage due its recursive properties.

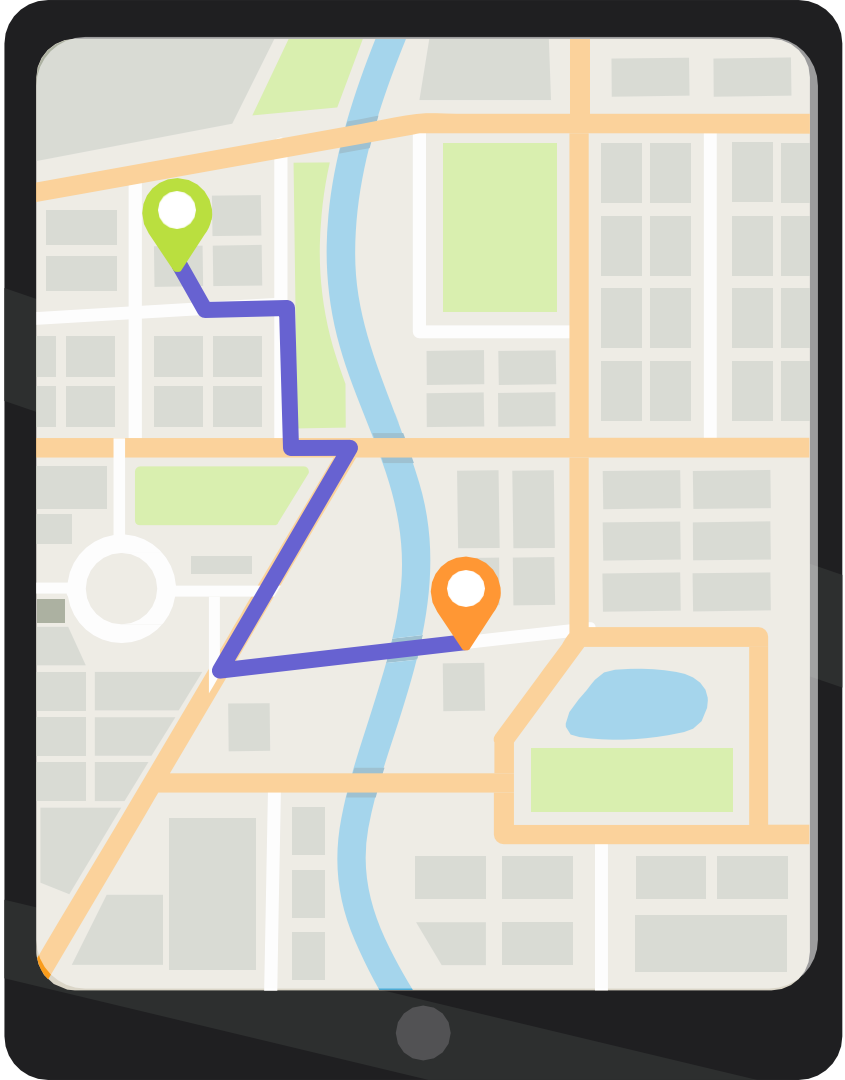
Future Improvements

1) Using an adaptive error tolerance for TS Greedy, based on the local curvature of the trajectory could produce better results in certain situations.

2) Since the TS Greedy uses recursive calls, we could use parallelization to make the program run faster.

Task 3

Comparing Trajectories



$|P| = m \quad |Q| = n$

Proposed Frechet Distance Algorithm

Intuition: the frechet distance is like the maximum length of the leash you need while walking your dog on one curve and you walking on another curve

Algorithm:

01 Start with a zero as minimum distance between all pairs of points

02 For each possible pair of points in P and Q, calculate the distance between the points in the pair

03 Choose the maximum distance on the minimum path to this point and mark the previous location on the path chosen

04 Trace back your path to accumulate all the points in the optimal path

Algorithm time complexity: $O(mn)$ Algorithm space complexity: $O(mn)$

Proposed DTW Distance Algorithm

Algorithm:

01 Start with a zero as minimum distance between all pairs of points

02 For each possible pair of points in P and Q, calculate the distance between the points in the pair

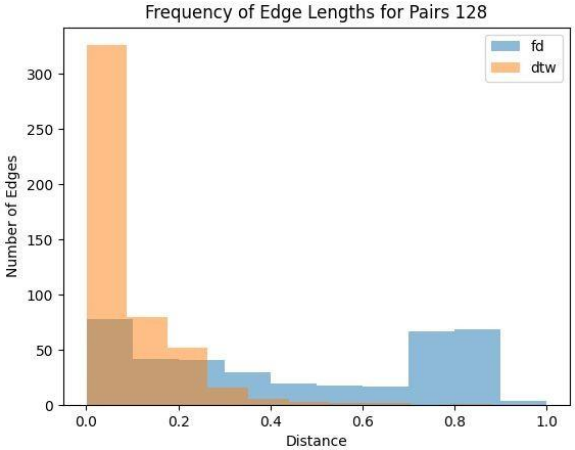
03 Add the distance to the minimum path to this point and mark the previous location on the path chosen

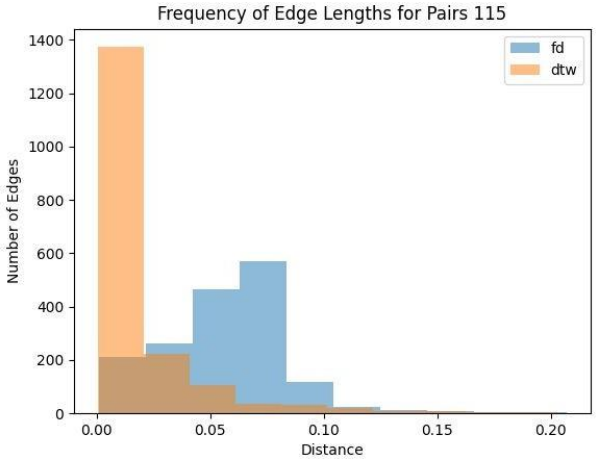
04 Trace back your path to accumulate all the points in the optimal path

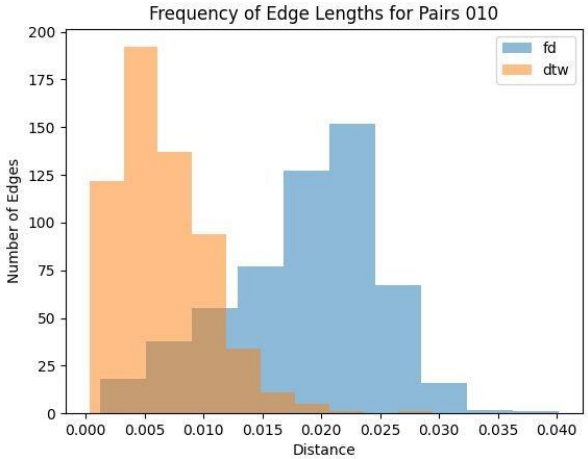
Algorithm time complexity: $O(mn)$ Algorithm space complexity: $O(mn)$

$|P| = m$ $|Q| = n$

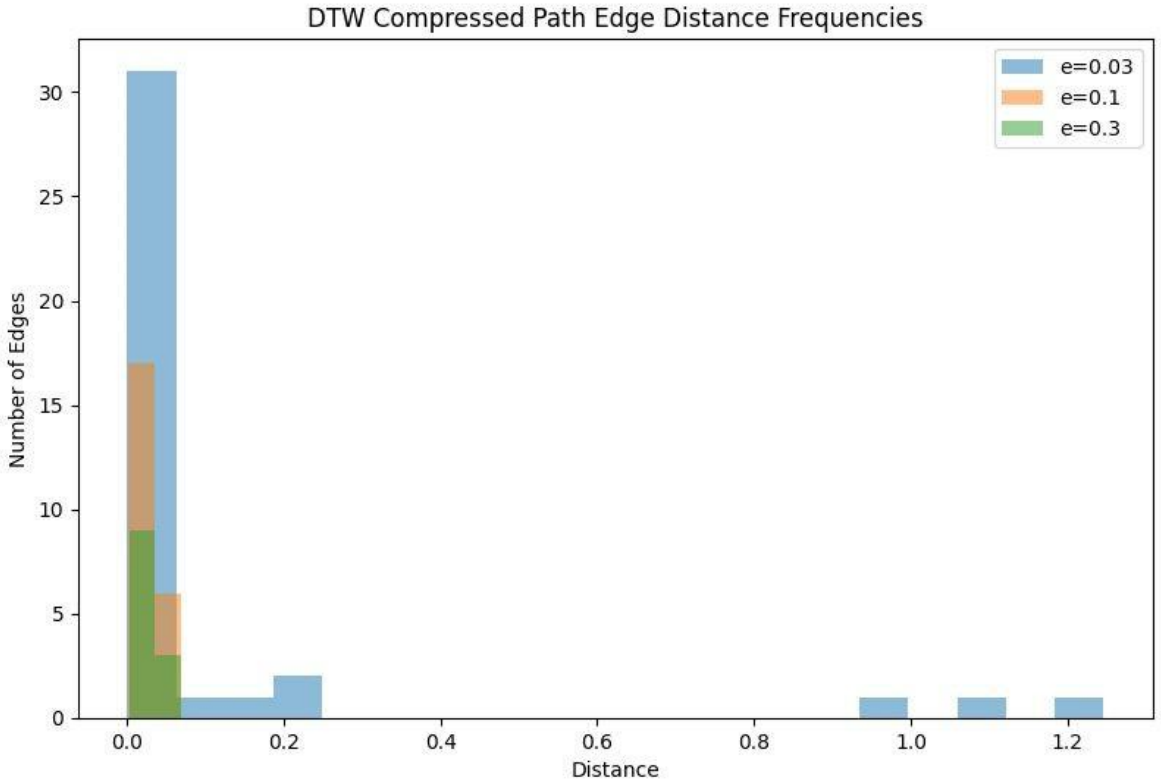
Experimental Results Part 1







Experimental Results Part 2



Proposed Algorithm Advantages and Drawbacks

Advantages:

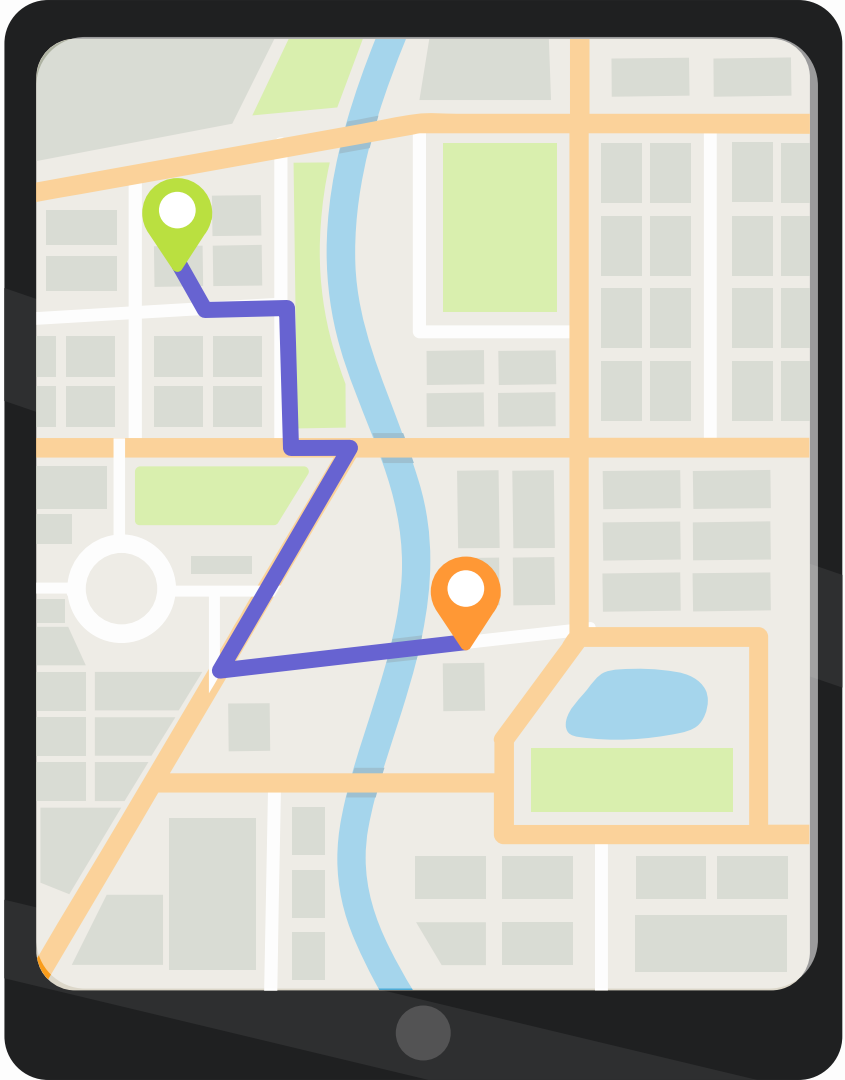
- Since the algorithm runs through each possible alignment of point pairs, it makes the most optimal decision in the end.
- The algorithm is dynamic, so it avoids repeated work on calculating the distances and comparing them again and again.

Drawbacks:

- The main part of the algorithm works to find the optimal alignment but then the path must be traced back. This causes an increase in the runtime and space complexity
- The algorithm takes up a lot of space because it stores the minimum distances, the sizes, and the previous pair in the path for each pair in the path.

Task 4

Center Trajectories



Approach II: Explanation and Justification

High Level Overview: Input: A dictionary of trajectories (lists of 2D points) called trajectories_dict

- Calculate the average number of points per trajectory
- Initialize sums and counts lists to store the sums of x and y coordinates and their counts, respectively
- Iterate through the trajectories and update the sums and counts based on a linear interpolation approach
- Calculate the central trajectory by dividing the sums by the counts for each point – Return central trajectory

Linear Interpolation:

- Linear interpolation is a method of estimating a value between two known points on a straight line
- Used it to resample each trajectory to have approximately the same number of points as the average number of points in all trajectories
    a. Using the average: Balances algorithmic complexity with simplicity (accuracy vs speed). By using the average, the function ensures a balanced representation of each trajectory when calculating the central trajectory.
- It assumes that the points in a trajectory can be approximated by a piecewise linear function, which is reasonable when the points are close together and the underlying function is relatively smooth
- The linear interpolation approach makes it possible to compute the central trajectory by averaging the x and y coordinates of the resampled points – but it is an estimation as we are not looking at every point in all trajectories

Approach II Information Continued

- **Runtime**: O(N) where N = $\sum_{i=1}^{n} |T_i|$ for all input trajectories
  - The outer loop iterates through the number of trajectories n
  - The inner loop iterates average_points times which is equal to the average number of points among all trajectories in $\mathcal{T}$
  - Essentially, it performs average points iterations n trajectory times
    - average points * trajectories = total number of points among all trajectories
- **Time Standardization:**
  - We are assuming that each trajectory represents some object moving at a constant speed, but speed can differ between trajectories
  - We are also assuming points in each trajectory are measured at roughly equal time intervals. This assumption allows the linear interpolation method to approximate time standardization when resampling the trajectories based on the average number of points.

Approach I & Approach II Outputs 3.2.2

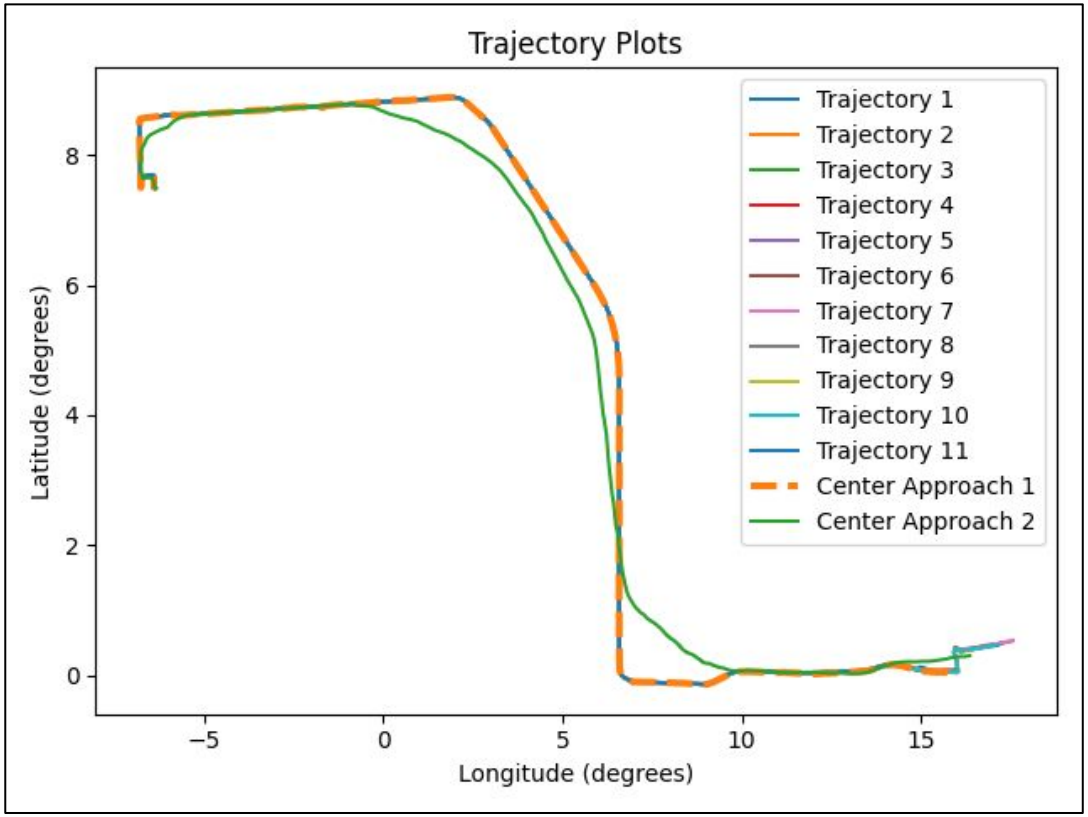
Avg distance from **Approach 1** center trajectory to all other Trajectories:

0.11037828477500894 km

Trajectory id in $\mathcal{T}$: 115-20080706230401

Avg distance from **Approach 2** center trajectory to all other Trajectories:

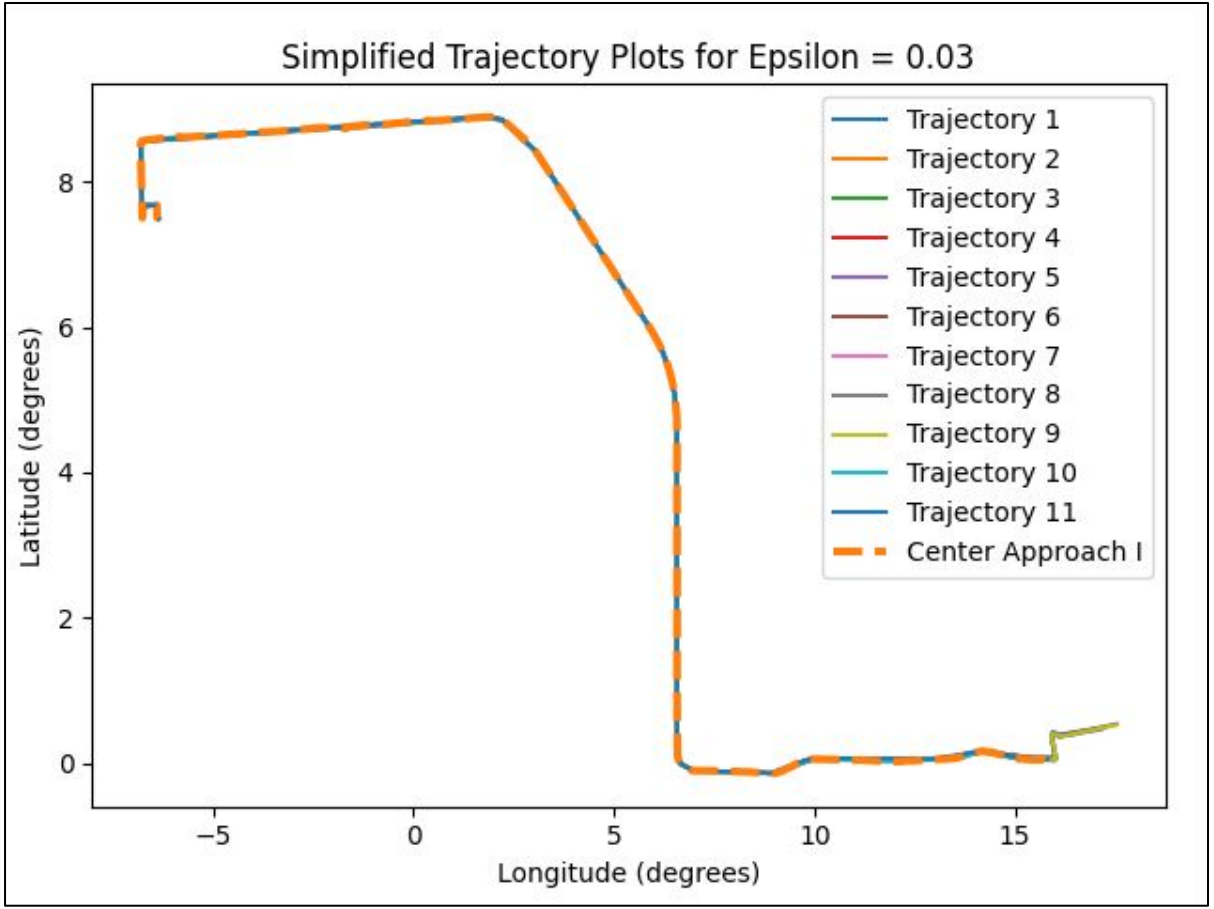
8.53300675881177 km



Approach I
Epsilon = 0.03

Avg distance from calculated center to simplified trajectories is:

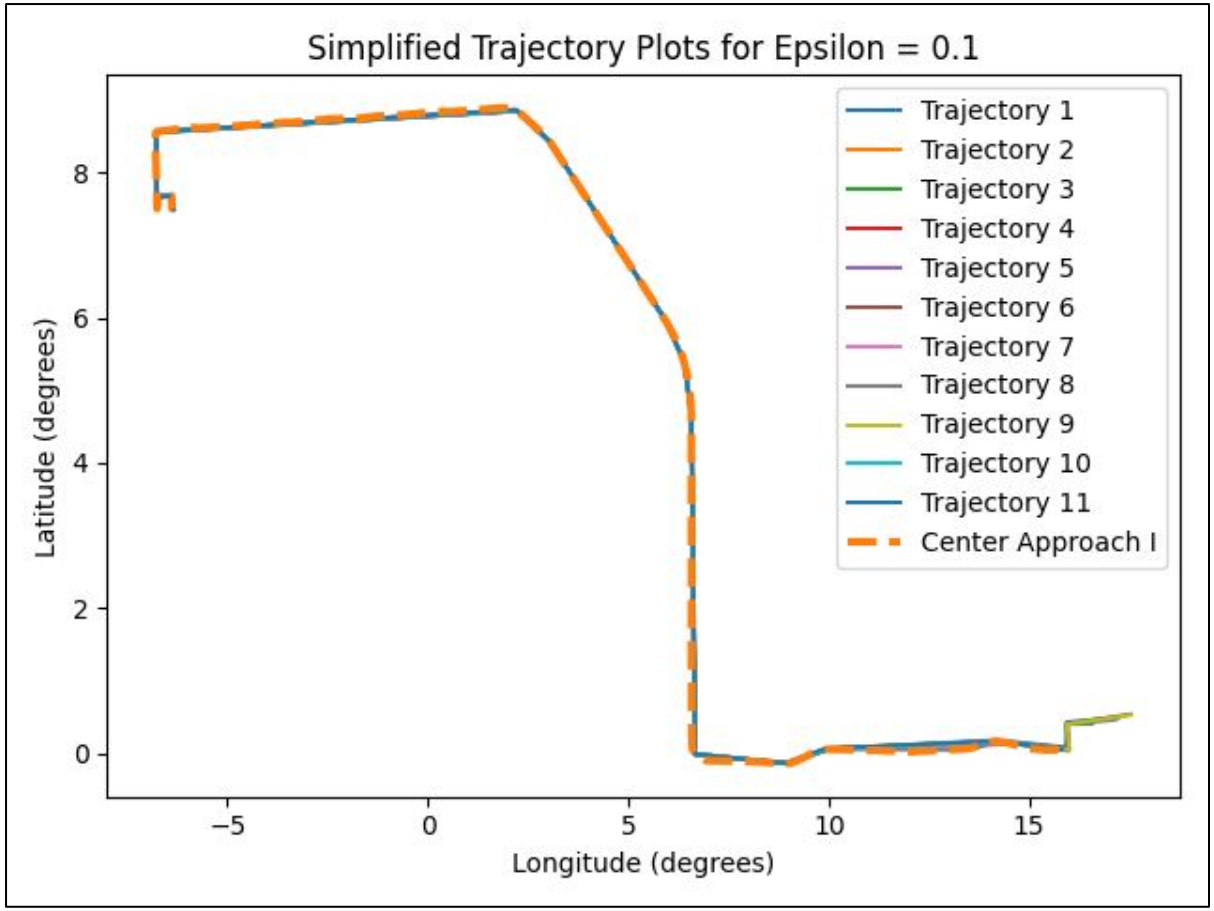
1.512953266226576 km



Approach I
Epsilon = 0.1

Avg distance from calculated center to simplified trajectories is:

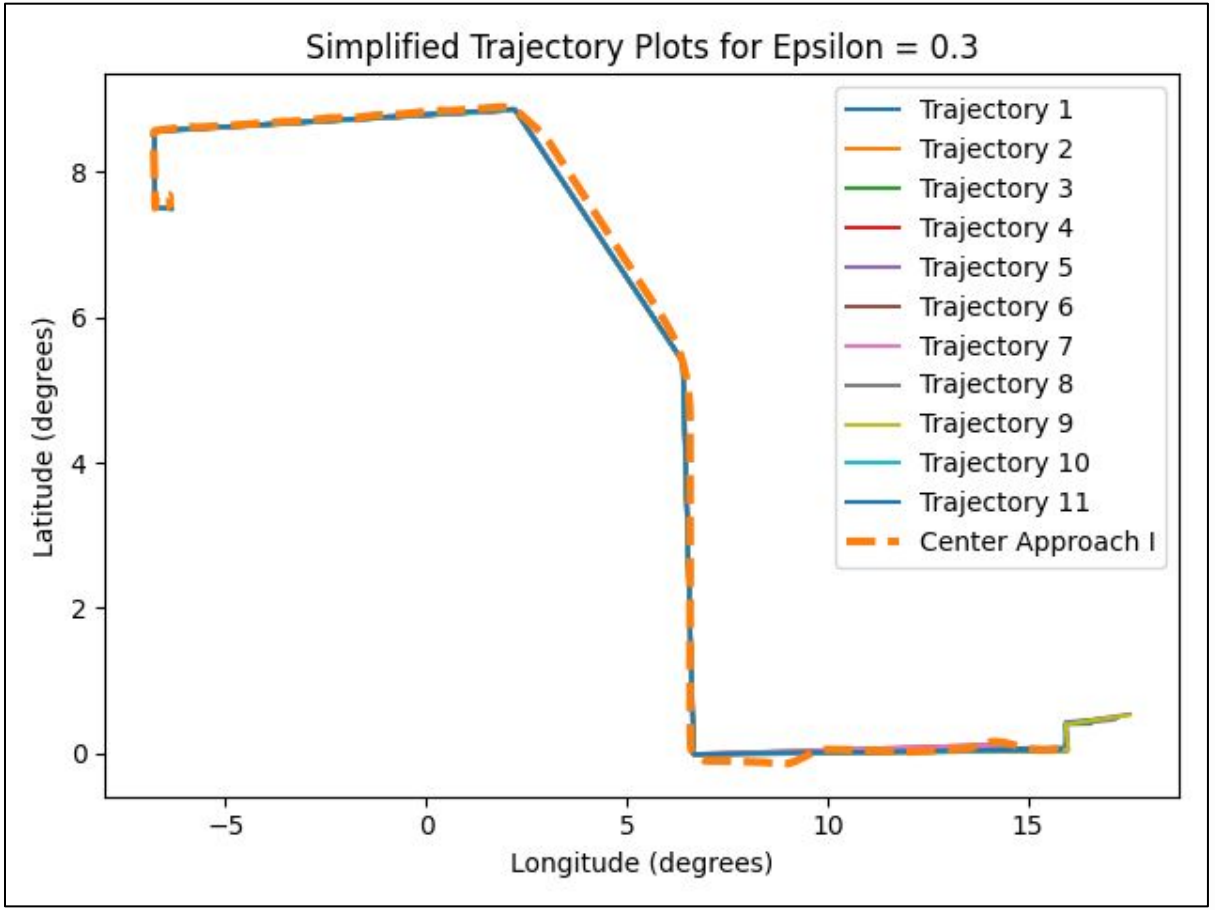
1.8990705719391308 km



Approach I
Epsilon = 0.3

Avg distance from calculated center to simplified trajectories is:

2.5890967068666875 km



Approach II

Advantages:

- Fast runtime: linear with respect to total number of points in all trajectories – no dependence on external distance measures that increase runtime (i.e. DTW)
- Linear interpolation assumes points in a trajectory can be approximated by a piecewise linear function. Via pre-graphing the trajectories, this is reasonable when the points are close together and the functions are smooth. This estimation is efficient and often strikes a good balance between accuracy and simplicity.
- The method can account for small variations in the input trajectories (i.e. size), as it calculates the central trajectory by averaging the resampled points across all trajectories.

Disadvantages:

- Only estimation bc timestamp column is not guaranteed
  - Unsure the accuracy of the timestamp column and how it reflects true relation
  - Assume equal time interval measurements for each trajectory, but this is not guaranteed and can lead to an egregious estimation
- If the points are not uniformly distributed (i.e. constant speed in each trajectory), the estimated central trajectory may not accurately represent the true center of input
- If the distribution of points is not proportionally spaced across trajectories, the computer central trajectory may be biased towards trajectories with more densely spaced points

Approach I

Advantages:

- This approach can handle nonlinear trajectory variations, such as complex shapes
- Minimizing the total DTW distance between trajectories, in order to find the central trajectory, can involve memoization that efficiently reuses previously computed DTW distances → reduces the overall computation time and necessary resources
- Flexibility in comparing trajectories of varying lengths and sampling rates

Disadvantages:

- If the trajectories are not closer to each other in the 2D space, this approach might egregiously estimate the trajectory because the center trajectory is one of the trajectories included in set that is being testing
    - So if the trajectories are not similar, the approach can just spit out an estimated trajectory that is not centralized at all
- Runtime is very slow ($O(N^2m^2)$)
    - n = number of trajectories
    - m = max number points in one of the input trajectories

Approach I Versus Approach II

Similarities

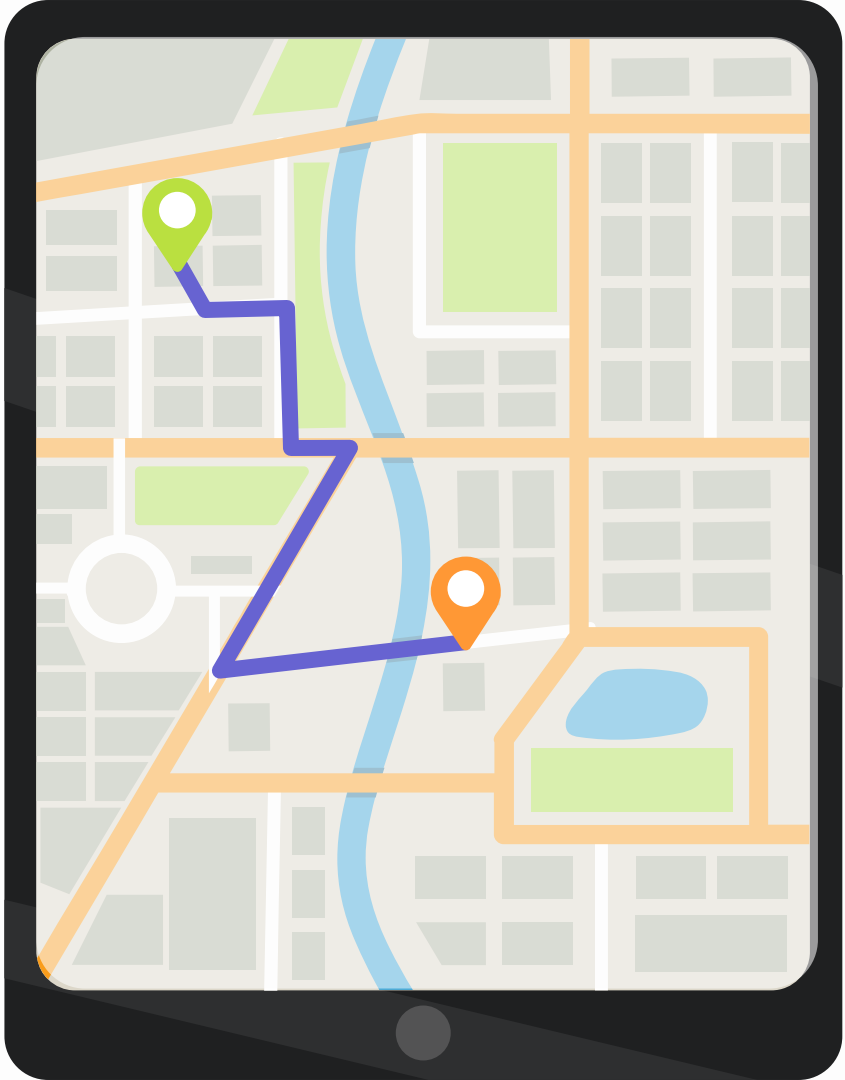
- Both approaches can handle input trajectories of varying sizes/point densities
- Both functions assume that the input trajectories can be aligned in some way to enable estimation of a central trajectory
- Both functions can be extended to handle more complex or higher-dimensional trajectory data.

Differences

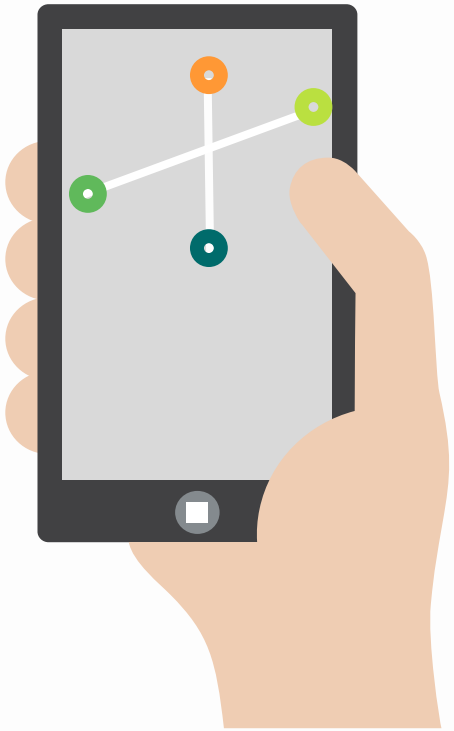
- Very different runtimes/efficiencies as discussed previously
- Approach I outputs a center that is included in the input set while the solution to approach II may or may not be included in the input set
- Approach I depends on distance formula DTW whereas approach II uses linear interpolation
- Approach II assumes that the points in each trajectory are measured at roughly equal time intervals. Approach I assumes that the trajectories can be aligned using DTW to measure the paired distances – and that this alignment is meaningful in representing the trajectories

Task 5

Clustering Trajectories



State and Describe Proposed Seeding Method



- The proposed seeding methods first selects one center trajectory at random.
- Then it calculates the distance between a trajectory and the closest selected center.
- The next center trajectory is randomly chosen with the probability of a trajectory being chosen center correlating to the distance from the closest chosen center.
- Using this method allows for randomness and allows for choosing centers a greater distance from each other. If the centers are far away from each other, then the cluster assignments are less likely to reshuffle when a new center is assigned.
    - For example if all trajectories assigned to center trajectory i are all very close to it from the previous calculation, choosing a new center trajectory far from the center trajectory i wouldn't change the cluster assignments by much. Therefore, the cost of clustering would be less.
- This process is repeated until k cluster center trajectories are chosen

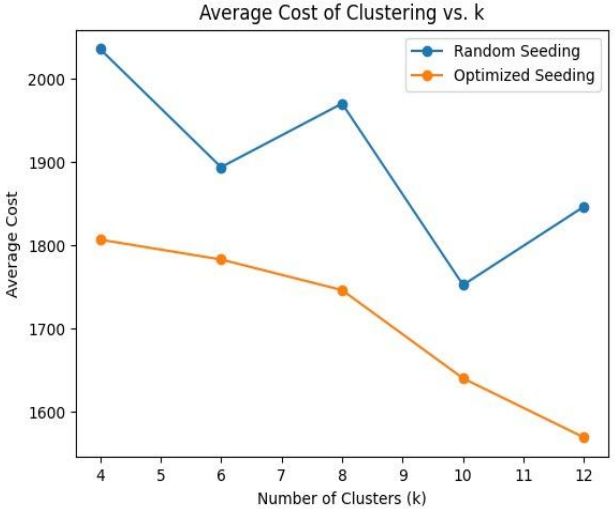
Advantages and Disadvantages of Proposed vs Random Seeding

- Proposed to use k-means++ algorithm for seeding.

- **Faster convergence:** Optimized seeding spreads clusters out more evenly across the data space, leads to quicker convergence as it requires fewer iterations to reach a good clustering solution. As seen in Figure 2 on the next slide, the optimized algorithms converges quicker (will not change later), compared to the random algorithm.
- **Improved cluster quality:** The k-means++ seeding creates initial centers that lead to better quality clusters, as evidenced by the lower average cost on Figure 1 (sum of squared distances between points and their assigned cluster centers).
- **Reduced sensitivity to initialization:** Random seeding can lead to poor clustering results if initial centers are not well-distributed. Optimized seeding reduces such chances of unfavorable initializations by selecting centers that are spread out across the data space.

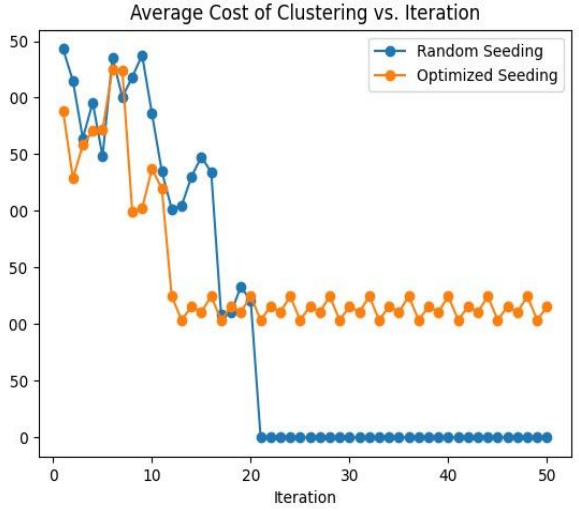
Experimental Results (Section 4.2)

Figure 1



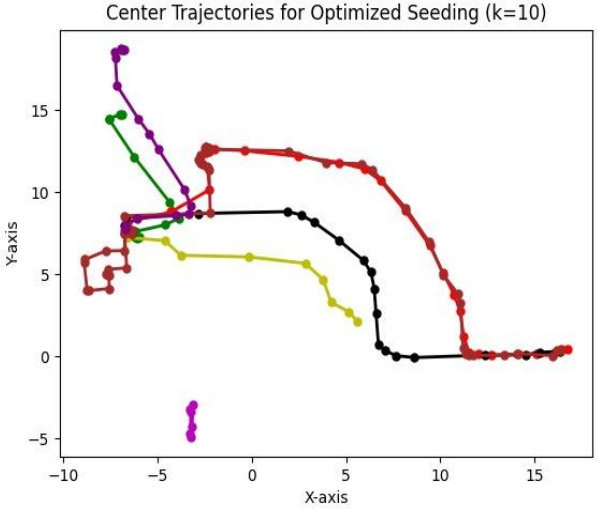
The random and proposed seeding both show a downward trend in cost as number of clusters increases. Some upward trends in random seeding could be attributed to it's randomness. Lloyd's algorithm does not guarantee always decreasing costs for increase in clusters.

Figure 2



The average cost of random clustering converges on the 21st iteration while the cost of the optimized clustering converges on the 12th iteration. After that the optimized clustering sees repeating results which means the optimal cluster was achieved

Figure 3



The center trajectories for k = 10 clusters is shown. Each cluster is represented in a different color. k=10 clusters was chosen because the decrease in average cost for proposed seeding from 10 to 12 wasn't as big as it was in previous changes between k values and it was the lowest average cost for randomized seeding

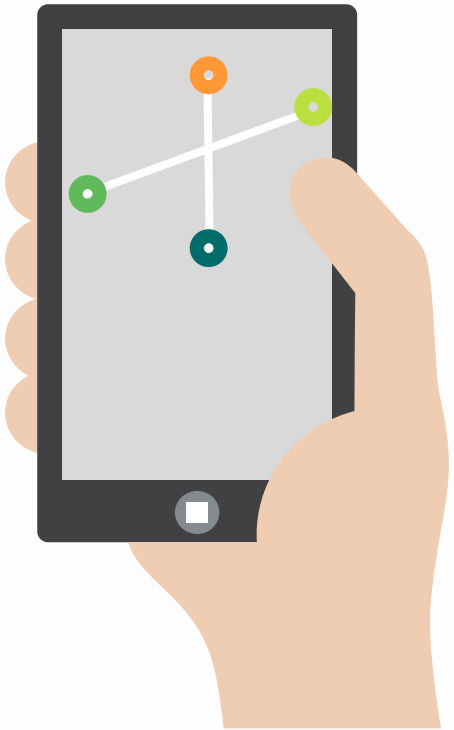
Experimental Results (Avg cost per cluster)

K Values	Random Seeding	Optimized Seeding
4	2036	1807
6	1894	1783
8	1970	1746
10	1752	1640
12	1846	1569

t_max, epsilon, and chosen k values

- The t_max value was chosen by running multiple trials of clustering with random and the proposed seeding methods for the given k values (4, 6, 8, 10, 12). For those multiple trials with different random seeds, I plotted the costs over multiple iterations and observed that most if not all cluster assignments converged after around 20 iterations. After testing, decided to use a t_max value of 50 iterations which was more than double to be safe. Using this t_max value allowed all cases tested to converge
- The epsilon value chosen was 0.05. Multiple trials were run for different values of epsilon (0.01, 0.03, 0.05, 0.1, 0.3) and considered how long each run took versus how the graphs looked. The value 0.5 was chosen because that allows us to run faster experiments.
- Our chosen value of k after running the experiments was 10 because it showed the best results for the random seeding method and the decrease in cost from 10 to 12 wasn't as great as it was in previous changes between k values.

Choosing number of clusters - k values



1. Given a number of data points, choose k values ranging from 1 to at least 10 or 1% of the total number of data points you have (if you have a large dataset).
2. Run lloyd's algorithm for each of the k values 3 times and take the average cost for all the runs for each k value.
3. Choose the k value which produces an "elbow" or the lowest cost of clustering before it starts increasing again.

Chose k = 10 because that was the value that produced an elbow with random seeding and had a lower decrease in cost when changing the k from k = 10 to k = 12. I tested the for k values of k = 4, 6, 8, 10, 12. The max k value tested was 12 which is about 1% of the total trajectories in our dataset (1190)

Alternate Approaches to Clustering Trajectories

1. DBSCAN: DBSCAN is a clustering method that relies on spatial density to group trajectories, identifying low-density regions as noise. This approach is particularly suited for clustering trajectories with diverse densities and forms since it doesn't necessitate a predetermined cluster count or specific cluster shapes.
2. DTW-based Clustering: DTW computes similarity among trajectories by enabling non-linear alignments, accounting for temporal variations. Using a clustering technique (such as hierarchical clustering) on the derived distance matrix, DTW-based clustering can aggregate trajectories exhibiting similar patterns, even if they are temporally misaligned or exhibit varying velocities.

Sources:

- **Analytics Vidhya**. (2020, September). *How DBSCAN Clustering Works*. Retrieved from https://www.analyticsvidhya.com/blog/2020/09/how-dbscan-clustering-works/
- **RStudio Pubs**. *Trajectory Clustering with DTW-based Techniques*. Retrieved from https://rstudio-pubs-static.s3.amazonaws.com/474160_0761428e8683401e941a92a4177254a4.html