Trajectory Inspector

Skye Im (ki539@nyu.edu)

1. Introduction

Background. Web-based aircraft visualizations pose a unique technical challenge. There is a wealth of data freely available for aircraft positional data, such as that from ADS-B Exchange[0]; many web apps exist that display current position. Sites such as ADS-B Exchange are possible because most modern aircraft are equipped with ADS-B transponders, which broadcast the ICAO code (unique to each aircraft) as well as optionally a veritable array of data, including position, altitude, airspeed and heading.

Much fewer web-based visualizations are available for trajectories, because of various issues. For one, data transfer rates are growing but finite, while aircraft positional datasets are often multiple gigabytes large for a single day. There are upwards of 40000 aircraft in the sky at any given moment. Additionally, plane trajectory distributions are often very dense—over continental Europe, for example. Naively displaying each trajectory on top of each other is simply visual clutter and will not allow the user to discover patterns within in, and this is without even considering technical limitations of how many trajectories *can* be displayed at once.

This paper proposes a method for visualizing aircraft trajectories to help discover high level flight path trends, explore trend members, and locate & compare trends by operator or country. A simple heuristic is used for stitching sequences of points into trajectories, and EDwP is used for calculating the high-dimensional distance between any two given trajectories. The *k*-nearest neighbor (*k*-NN) algorithm proposed by Lu & Fu[9,10] is selected for the clustering approach, and clusters are again clustered in a bottom-up hierarchical aggregation approach[8]. The prototype is coded using d3.js[11] because of its powerful visualization libraries, and hosted online.

Care was taken to make sure the design is suitable for an online application, where data is transmitted to the client on each load, as opposed to a local application which is guaranteed a traditional database with high bandwidth. Although no new visualization techniques are proposed, Trajectory Inspector is a novel combination of existing techniques that proposes a potential new method to visualize aircraft trajectories.

In **Related Work**, some existing aircraft visualizations are introduced and analyzed for design hints for Trajectory Inspector. Then, four trajectory distance functions are introduced and compared, as well as two clustering algorithms. Presented is the rationale behind selecting each algorithm. In **Process**, the data preprocessing and transformation method used for Trajectory Inspector is described in detail. In **System Overview**, the visual design and interaction design of Trajectory Inspector is described. In **Evaluation**, some interesting trends are introduced. Finally, in **Conclusion**, future work that could be done to improve Trajectory Inspector is introduced.

2. Related work

Related work—visualizations. Planefinder[1] is an online visualization that maps real-time aircraft position to position, aircraft type to icon shape, and aircraft heading to angle. Clicking on each icon creates a side-view that displays information such as operator, aircraft type, altitude, an image of the aircraft, and so on. Despite the fact that Planefinder only displays the current position of aircraft, it conveys an illusion of trajectories due to how icons often follow each other in distinct trails across oceans. However, it is difficult to read the visualization near airports due to the increased density of the marks on the map.

UK24[2] is a non-interactive visualization produced by NATS. Aircraft over the skies of the United Kingdom are displayed as moving dots that leave a transparent, fading trail. The visualization is in 3D, rendered into a 2D video with a moving perspective. UK24 shows that trajectory information can yield insight into traffic patterns that a discrete point view cannot, but also shows how dense traffic near airports makes the visualization harder to understand.

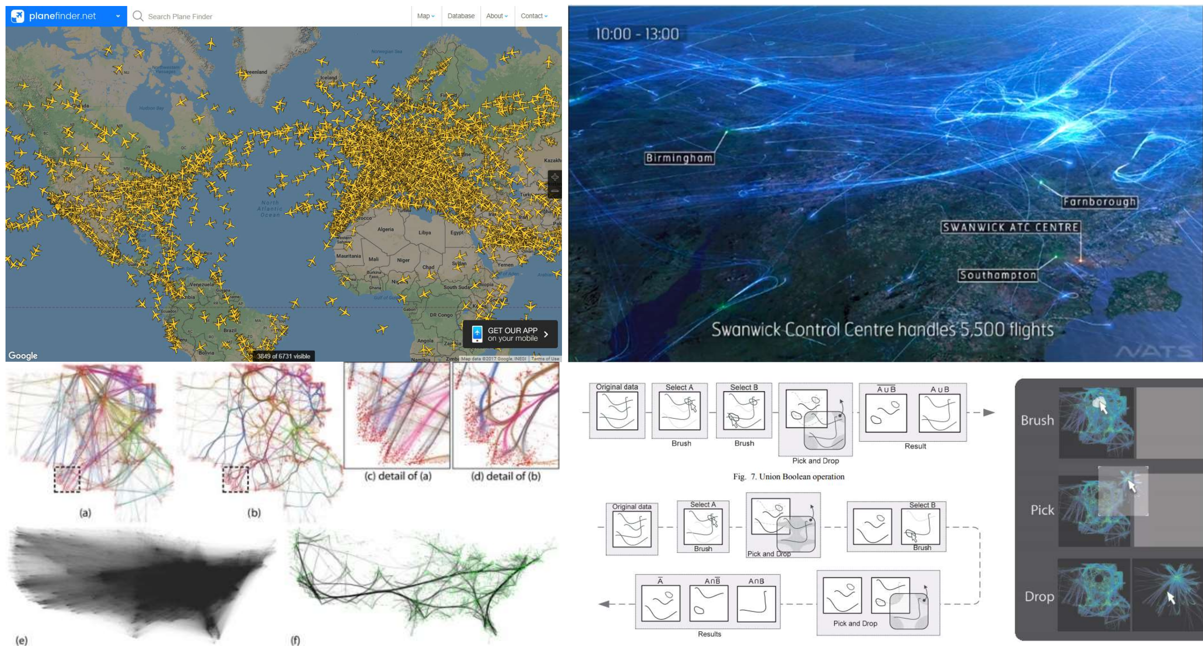

Figure 1. Top-left, Planefinder[1]. Top-right, UK24[2]. Bottom-left, edge bundling; figure from [3]. Bottom-right, FromDaDy[4].

In Figure 1-BL, we can see that edge bundling on start-destination graphs for the United States is very effective at reducing clutter and exposing patterns. While the trajectory data this paper uses is positional, not point-to-point, it is possible to take edge bundling as an inspiration for a similar process. The concept of the process is thus:

1. Using some distance measure, put similar segments into a cluster
2. Using some trajectory combination algorithm, derive a representative trajectory from each cluster
3. For each cluster, use some cluster–cluster distance measure to find clusters of clusters.
4. For each cluster of clusters, use the representative trajectory of each cluster to find a representative trajectory for this cluster of clusters.
5. Repeat.

This process is essentially bottom-up hierarchical aggregation[8] with a representative for each cluster level.

Last, FromDaDy[4] is a sophisticated plane trajectory viewer that utilizes sweeping or brushing motions to allow the user to reduce the currently displayed set of data to what the user thinks is interesting. In other words, it trusts that the user has the best idea of how to discover interesting

patterns and facilitates this process of discovery. Although data transformation is an important part of infoviz, the role of the user in analyzing data must not be forgotten.

Related work—distance algorithms. Trajectory Inspector requires a certain number of algorithms: a trajectory distance comparer, a clustering algorithm, and a trajectory combination algorithm. Due to technical limitations and time constraints, the combination algorithm was not finished for the demo and is excluded from this paper, but it is likely that some force-directed or data ink-reducing method would be used.

In this paper, a *trajectory* is a sequence of two or more points in some coordinate space (typically, latitude and longitude). A trajectory in the form $T = [p_1, p_2, p_3, p_4 \ldots p_n]$ would have length *n*. Since Trajectory Inspector uses the lat/long information broadcast by aircraft, when comparing trajectories one cannot assume that trajectories have the same length, sampling rate, or accuracy. This presents significant challenge in determining how similar two trajectories are.

A few candidates were considered for the distance algorithm. The first is a naïve element-by-element comparison of a trajectory, calculating the Euclidian distance between matching pairs and returning infinite distance for trajectories of different lengths. It is obvious that this is not a viable algorithm because two visually similar trajectories may have wildly different lengths, depending on the number of samples for each trajectory.

One is Edit Distance on Real sequence[5] (EDR). EDR is a simple, recursive *O(nm)* complexity trajectory comparison algorithm. For each element in the two sequences, the algorithm attempts to • find the distance between the rest of sequence 1 and sequence 2, plus whether the first element of each matches • find the distance between sequence 2 and the rest of sequence 1 and add one • find the distance between sequence 1 and the rest of sequence 2, and add one. The minimum of the three values is selected as the distance. A *match* is best described as an approximate Manhattan distance. If $(|p_{1x} - p_{2x}| < \epsilon) \wedge (|p_{1x} - p_{2x}| < \epsilon)$, the two points are considered to match. For more details, consult [5]. EDR is thus resilient against noise. However, Figure 2 shows that for two trajectories with different sample rates, EDR provides vastly

inaccurate results. Although the trajectories *a* and *b* are visually similar, their EDR distance is 3. The trajectories *b* and *c* are dissimilar but their EDR distance is 1.

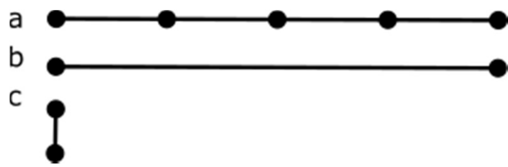

Figure 2. EDR(a, b) = 3, EDR(b, c) = 1.

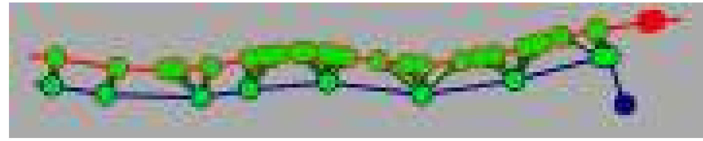

Figure 3, Multiple Assignment, taken from [6] Figure 6(d).

Another algorithm is Multiple Assignment[6] (MA). MA is an *O(nm)* algorithm that considers the similarity between each trajectory as a directed, bipartite graph of assignments, taking inspiration from the popular discrete Fréchet distance algorithm, aka the 'dog-walking algorithm,' as well as DNA analysis techniques. Although the authors assert that MA is robust against noise and sampling rate variation, and also provide a trajectory segmentation-and-matching algorithm, MA was too difficult to implement so Trajectory Inspector uses a third algorithm with similar properties to MA.

The algorithm Trajectory Inspector uses is Edit Distance with Projection[7] (EDwP). EDwP is an extension of EDR. Instead of giving a flat cost depending on whether a point matches, EDwP attempts to insert new, intermediary points on each trajectory and match on those. Additionally, the distance is scaled by the proportion to the total trajectory each segment is. Thus, EDwP is less sensitive to sample rate variations.

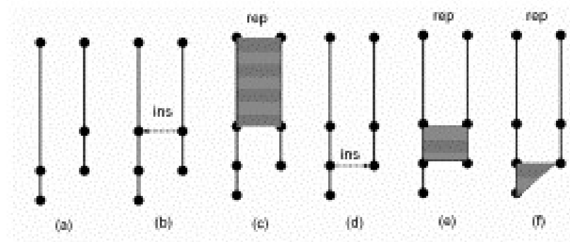

Figure 4. EDwP illustration, from Fig.3 of [7]

EDwP is an *O(nm)* comparison algorithm, where *n* and *m* are the lengths of the trajectories being compared. So order to create a distance matrix for clustering *L* trajectories, the complexity of the distance matrix function would be $O(L^2n^2)$, where *n* is the average length of all the trajectories. The authors of EDwP propose a tree structure, TrajTree that is optimized for *k*-NN lookups that

would provide sub-$O(L^2n^2)$ performance. However, once more due to technical and time constraints, TrajTree was not implemented in Trajectory Inspector.

Related work—clustering. Trajectory Inspector requires a clustering algorithm that does not require the number of clusters to find as an input, because even with hierarchical aggregation, arbitrarily defining a number of clusters to find would likely cause very misleading results. Thus, two algorithms were considered for clustering. First is the popular, often-effective DBSCAN[12], which scans for areas with high density and grows clusters from each seed. DBSCAN does not require the number of clusters to be provided, and is able to cluster non-circular patterns well (eg, the half-moon dataset), compared to *k*-means. It is also capable of detecting outliers and excluding them from clusters. For testing purposes, the Python library sklearn[13]'s implementation was used.

The second algorithm is the *k*-NN clustering algorithm proposed by Lu & Fu[10], as described in [9]. The algorithm is as follows:

```
foreach element in dataset:
     if no clusters exist:
          new cluster ← element
     foreach cluster:
          d = min(distances of element and each cluster element)
          c = argmin(distances of element and each cluster
          element)
     if d < threshold:
          c ← element
     else:
          new cluster ← element
foreach cluster of size 1:
     outlierCluster ← cluster element
```

Similarly to DBSCAN, Lu & Fu's *k*-NN has the desirable properties of detecting any number of clusters without having to provide a number, and having the ability to detect outlier elements.

However, as Lu & Fu's *k*-NN's cluster size increases, subsequent elements are more likely to be put in a given cluster regardless of how similar they actually are, due to the *min* clause.

For Trajectory Inspector, technical limitations meant that only fifty rows were used. Under these conditions, Lu & Fu's *k*-NN produced better results than DBSCAN.

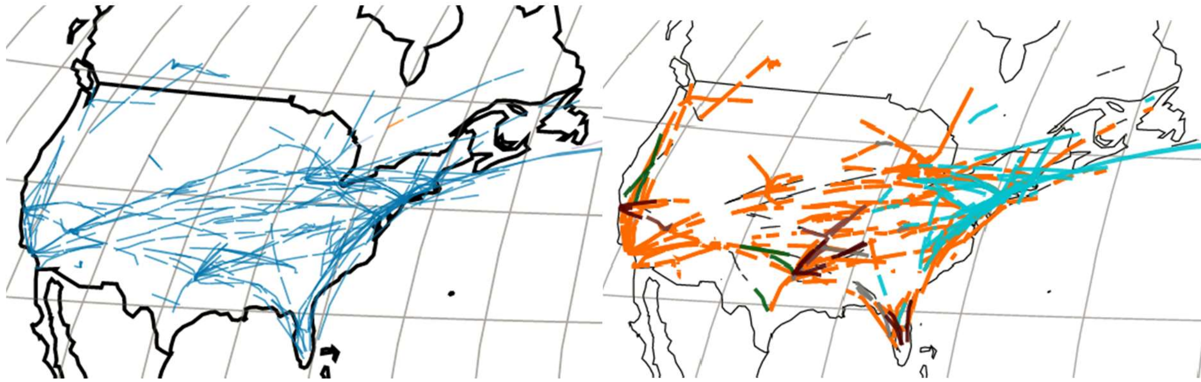

Figure 5. Left, DBSCAN results on data. Right, Lu & Fu's *k*-NN's results.

DBSCAN tended to cluster every trajectory in the United States into one group and produce a small quantity of very small groups. Lowering the threshold simply caused more elements to be considered outliers. Lu & Fu's algorithm produced slightly more differentiated results, so was used for the demo. When there are a higher number of rows used, it is possible that DBSCAN would perform better.

3. Process

Trajectory Inspector's data processing is majority written in Python, while segments that require performance at C#.

The first step of a visualization is acquiring the data. ADS-B Exchange[0] provides data archives for sensor data collected by volunteer transponders, available per day. The compressed archive is roughly 8GB for each day, and uncompressed is 40GB of JSON files. The data used for the demo is August 29, 2017, and has 25,665,057 'pings' of sensor data. Each ping has a wealth of data. Some of the data is pulled from the signal an aircraft broadcasts, while others are found by looking up the unique ICAO code each aircraft on the ICAO database. Some of the data fields are introduced in Table 1. One problem is that almost every data field is optional, even including

longitude, latitude and timestamp. Roughly 7 million pings (28.9% of total) lacked longitude or latitude information, and were excluded.

From ping	From ICAO database
ICAO code (six digit hex)	Operator
Timestamp	Country of registration
Altitude	Aircraft type
Speed	Callsign
Heading	Year of registration
Squawk	Number of engines
Latitude	Wake turbulence class
Longitude	Destination
…	…

Table 1. Information from ADS-B Exchange.

The data provided by ADS-B Exchange is in JSON format, but each file often contains minor syntax errors. Most common was the trailing comma error, and second most common was duplicated {s. A script was ran to fix these errors, then the JSON was inserted into a PostgreSQL database and deduplicated. As a result, the data's size was reduced to 3GB.

Next, a simple heuristic was applied to construct trajectories from sequences of points per aircraft. Each sequence is sorted by the timestamp in ascending order. Points are added to a trajectory, until the next point has a timestamp that is more than 1 minute since the previous one. Additionally, to aid in segment clustering each trajectory is limited to 30 minutes of flight time.

The average number of points per trajectory at this point is ~200 at this stage. Considering that EDwP is a *O(nm)* algorithm, and that each trajectory is only up to thirty minutes long, it is possible that many of the data points are redundant and can be removed. The Visvalingam-Whyatt line simplification algorithm[14] is applied to each trajectory to simplify the path. Each triplet of consecutive points is treated as a triangle, and its area is calculated. Then, points that have the smallest area are repeatedly removed until the line is sufficiently simplified. Trajectory

Inspector's VW algorithm is a port of the reference implementation produced by Mike Bostock of d3.js fame[15]. After this stage, the path lengths are on average 15 long.

The EDwP algorithm is run on every pair of trajectories to produce a distance matrix. Trajectory Inspector's implementation was written in Python originally, then ported to C# for increased performance, while retaining the readability that a high-level language has—EdwP lends itself well to an object-oriented design, and a C implementation would be messy at best.

Finally, Lu & Fu's *k*-NN algorithm is applied to produce clusters of trajectories. Due to the nature of clustering, the distance matrix in advance has, at worst, the same runtime as an online calculation. Once an initial set of clusters is created, a 'representative' trajectory from each cluster is created. Trajectory Inspector simply selects the trajectory that minimizes the squared distance to every other trajectory as the representative. The clustering process is repeated, with each cluster sorted into a fewer number of meta clusters. The outlier cluster is not included in the second aggregation stage. In the demo there were roughly fifty clusters after the first stage, and less than ten by the second stage. The bottom-up hierarchical approach minimizes the largest drawback of Lu & Fu's *k*-NN, in that different thresholds can be used for cluster-to-cluster grouping while preventing a single mega-cluster from forming.

Many of these process choices were made for simplicity rather than any deep significance. These choices will be discussed in the Future Work section.

4. System Overview

Trajectory Inspector is targeted for a browser environment, to lower the threshold of difficulty in accessing the visualization. This means that performance is a priority, both in data transmission and rendering speed. Browser-rendered SVG is a mature technology and is fairly optimized, but even so it is difficult to display more than some thousand individual paths. It is in nobody's interest to render thousands of paths at once, because of the massive visual clutter and overlap generated. Also, transmitting the entire dataset upfront is unlikely to be practical, because of the sheer size of the data. Therefore, the hierarchical aggregation employed in the data processing

step is leveraged to reduce visual clutter and improve performance, and the concept of edge bundling is also applied to make high-level patterns more obvious.

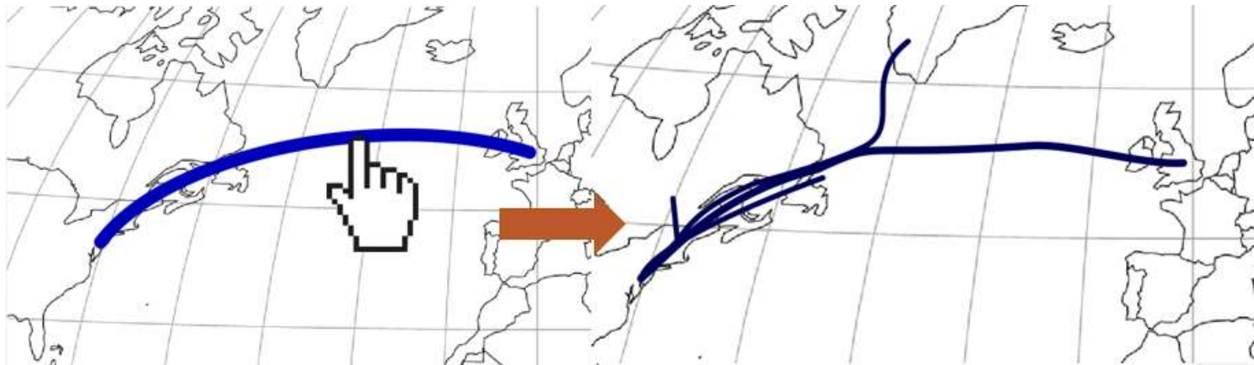

Figure 6. Mockup of trajectory splitting.

The initial view would have a small number of thick trajectories displayed. Accordingly, the initial data transmission would be the first two or three layers of the cluster hierarchy. The user would click a trajectory to display its child clusters, and information about the levels below the clicked clusters would be streamed. The demo of Trajectory Inspector does not have the trajectory combination feature, though it does have hierarchical aggregation.

Aside from the technical aspects, Trajectory Inspector also attempts to target the following tasks: *discover* flight trends, *explore* members of a trend, *locate & compare* trends by X (country, operator…). Figure 7 shows a typical screenshot of Trajectory Inspector in use.

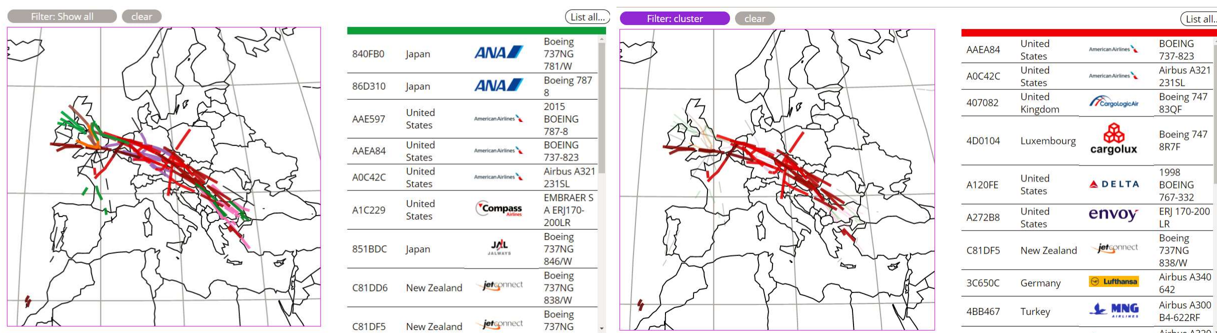

Figure 7. Trajectory Inspector demo. Left, initial view. Right, focused on cluster.

To the left is the **map view**. On the map, trajectories are displayed as lines, and the color channel is mapped to the cluster. Aircraft positions are mapped to the position channel via a projection (the Winkel projection in the demo), since that is the most intuitive for geospatial data.

Each cluster is effectively a trend, and the user can discover new trends by mousing over a cluster. Clicking on a cluster highlights the cluster and de-emphasizes other clusters, thus

reducing clutter. Clicking on a cluster also sets the **filter** in the top left corner to “cluster”, and the **selection** to the right is updated with the contents of the cluster, the details of each aircraft selected. The top bar of the selection takes on the color of the cluster to provide a hint as to what the current selection is. Clicking on an ICAO code will change the filter to “icao” and emphasize only the selected aircraft. Clicking on country of registration or operator will also change the filter, respectively to “country” and “operator”. An important part of the design is that interacting with a selection does *not* change the selection. This allows the user to rapidly compare various elements within a selection. For example, the user could view each American Airlines flight to see where the aircraft is coming from and where it is going to, so that it would be in the selection.

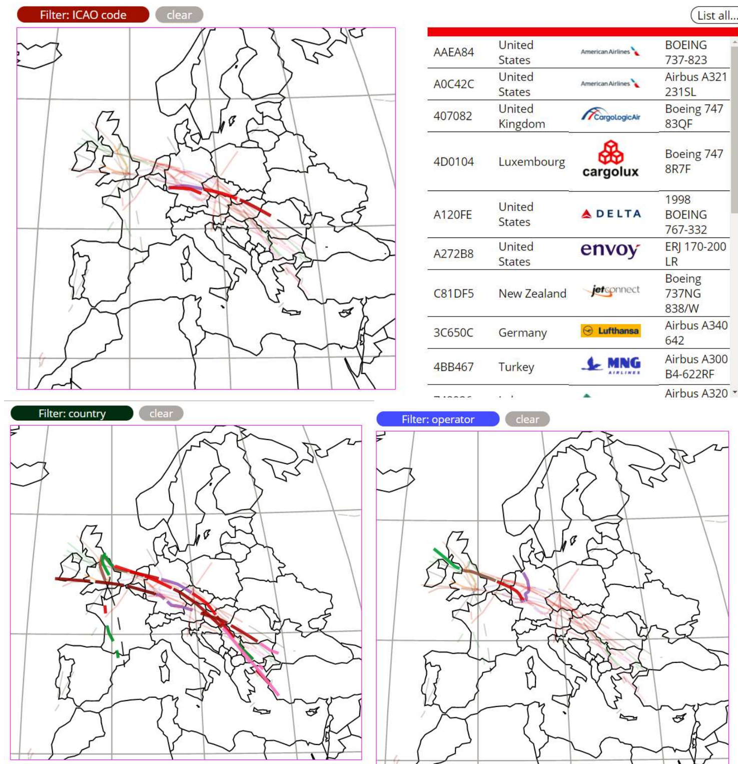

Figure 8. Top, selecting ICAO. Notice that the selection does not change. Bottom left, selecting United Kingdoms flights. Bottom right, selecting United Airlines flights.

If the user wishes to view all flights from an operator or a country, they can click the *list all* button in the top right corner and select the preferred type.

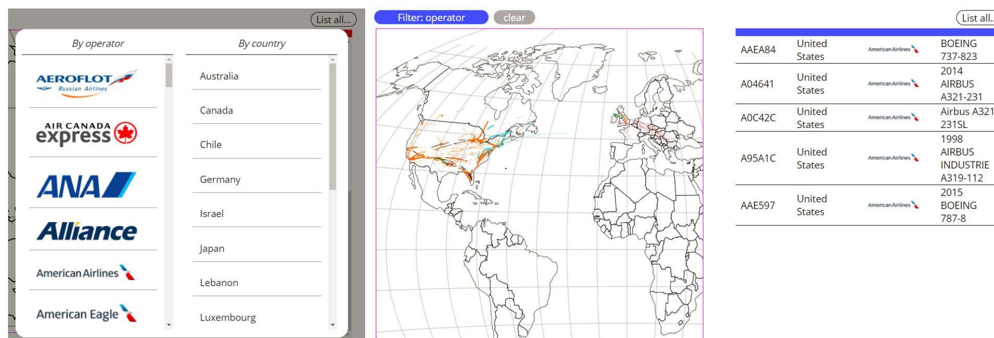

Figure 9. Left, *list all* menu. Right, changed selection to American Airlines.

Note that the top bar of the selection changes color to match the filter. This is, again, an attempt to provide the user with a hint of what the current selection is. There is some risk of a cluster color matching up with a filter color, and choosing filter colors that avoid this issue may be a

preferred method. Alternatively, filter-selections may be patterned, while cluster selections are solid colored.

A not insignificant part of how effective a visualization is lies in the polish. There are many features in the demo for maximum usability. One of the features is adaptive line widths, where the latitude/longitude graticules and the country outlines change width depending on how zoomed in the user is. If a single line width is used for every zoom level, at high zoom the outlines would dwarf the trajectories, while at low zoom there will be severe aliasing which hurts readability. Another feature is the responsive design, where the map and selection change size and position depending on screen width. Mousing over trajectories causes an animation to play, where a trajectory grows slightly; changing a filter transitions all non-selected trajectories to be thinner. Changing the map view uses the intuitive gestures of “click and drag” and “scroll,” for panning and zooming, respectively. Finally, elements that you can interact with in the selection change color and gain an underline when moused over. All of these factors contribute towards a more intuitive user experience, with lower time to learn how to use the system.

5. Evaluation

Usability. In order to gain a rough idea of how intuitive the visualization is, five people were selected at random among the author’s acquaintences. Their backgrounds varied, but none worked in the field of information visualization. Without any extra explanation—even of what the visualization was *of*—all five were able to access most of the features of the visualization.

However, the *list all* button was cited as an example of unintuitive design, because the button is small and unemphasized.

Findings. Because of the limited data actually used in the demo, it is difficult to find meaningful, large-scale patterns. However, there were some interesting local patterns, mostly in the form of outliers.

Figure 10. Jin Air flight path in orange.

The orange line is a Jin Air flight that was mostly filtered from the demo as an outlier. Jin Air is a South Korean low-cost airline that services Korean domestic flights as well as a small selection of Japanese lines. However, one aircraft was sighted flying from a location in Europe, across Russia to Seoul. And there are indeed no commerical flights from Korea to Europe serviced by

Jin Air. However, low-cost airlines often purchase used aircraft rather than commissioning new ones. The plane may have been en route from the buyer's hangar in Europe to Korea.

Another outlier is a private jet operated by a multilevel marketing company—more commonly known as a "pyramid scheme." The ICAO code A10F9E is registered to SUNRIDER CORP, SUNRIDER INTERNATIONAL, and is identified as a McDonnell Douglas MD-80.

Figure 11. Left, A10F9E flight path. Right, a Dogulas MD-8x model also owned by Sunrider.

The author speculates that Sunrider simply used the company jet as a method to "wow" recruitees, flying it on a short loop around the airport.

It is relatively easy to spot private aircraft between the commercial airlines. Operators are not necessarily airliners or corporations, either. One Kevin W. McCue was discovered flying a 1957 Cessna 172 in the region of Arizona, and two Eurocopter helicopters owned by Mustang Leasing LLC were found in Nevada, drawing circular paths rather than long ones.

6. Conclusion

Trajectory Inspector is a combination of existing methods to create a novel, web-suitable visualization. However, many of the choices made were not the most optimal choice, and much work can be done to improve Trajectory Inspector. For example, the EDwP authors also propose a data structure dubbed TrajTree[7] which supported sub-n^2 complexity for comparing two trajectories, which would greatly improve runtime. In fact, due to the hierarchical nature of TrajTree the separate clustering algorithm might not be required, instead collapsing TrajTree into clusters. Lu & Fu's *k*-NN still has difficulty producing distinct clusters in the North America region. Be it by modifying the algorithm or selecting a new one, there is no doubt that

improvements can happen. The segmentation heuristic currently used is quite arbitrary—we may be missing patterns which are not obvious from trajectories that are sliced into thirty minute segments. Perhaps a segmentation algorithm similar to that proposed by the authors of MA[6] may yield better results.

The visualization itself could also benefit from the trajectory combination feature, which was not implemented in the demo. Currently, each segment is displayed individually, and this causes visual clutter which is only partially ameliorated by coloring each cluster. Also, filters for time periods, as well as allowing for combinations of filters (eg, "American Airlines OR United Airlines") may also improve what patterns could be discovered using Trajectory Inspector. The number of colors for clusters is sufficient for now, but with increased number of trajectories, colors will not be a sufficient method to distinguish between clusters. Clarifying which channel clusters are mapped to will improve the visualization.

Even with all the work that can be done to improve Trajectory Inspector, it is the author's opinion that the current implementation still presents a unique and interesting, if rough, method of visualizing aircraft trajectory information in a way that is suitable for the web format.

Works referenced

[0] *ADS-B Exchange*. (ADSBexchange.com)

[1] *Planefinder.* (planefinder.net)

[2] National Air Traffic Control Services, *UK24*, 2014.

[3] C.Hurtera, S.Conversya, D.Gianazzaa, A.C.Teleab, *Interactive image-based information visualization for aircraft trajectory analysis.* Transportation Research Part C: Emerging Technologies, Volume 47, Part 2, October 2014, Pages 207-227. Used Figure 11

[4] C. Hurter, B. Tissoires, S. Conversy. *FromDaDy.* IEEE Transactions on Visualization and Computer Graphics, Volume: 15, Issue: 6, Nov.-Dec. 2009.

[5] L. Chen, M. T. Özsu, V. Oria; *Robust and Fast Similarity Search for Moving Object Trajectories,* SIGMOD/PODS '05.

[6] S. Sankararaman et al. *Model-driven matching and segmentation of trajectories*, SIGSPATIAL'13; p234-243.

[7] S. Ranu et al, *Indexing and Matching Trajectories under Inconsistent Sampling Rates,* 2015 IEEE International Conference on Data Engineering; p999-1010.

[8] N. Elmqvist, J. Fekete, *Hierarchical Aggregation for Information Visualization: Overview, Techniques, and Design Guidelines*, IEEE Transactions on Visualization and Computer Graphics, Institute of Electrical and Electronics Engineers, 2010, 16(3), p439-454.

[9] W. Peng, M. O. Ward, E. A. Rundensteiner; *Clutter Reduction in Multi-Dimensional Data Visualization Using Dimension Reordering*, IEEE Symposium on Information Visualization (2004) ref 15; is summary of [10]

[10] S. Y. Lu and K. S. Fu. *A sentence-to-sentence clustering procedure for pattern analysis*, IEEE Transactions on Systems, Man and Cybernetics, 8:381–389, 1978.

[11] M. Bostock, V. Ogievetsky, J. Heer. *D3 Data-Driven Documents,* IEEE Transactions on Visualization and Computer Graphics, Volume 17 Issue 12, December 2011. p2301-2309.

[12] M. Ester et al. *A density-based algorithm for discovering clusters in large spatial databases with noise.* Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining. 1996.

[13] Pedregosa et al. *Scikit-learn: Machine Learning in Python*, JMLR 12, pp. 2825-2830, 2011.

[14] M. Visvalingam, J. D. Whyatt. *Line generalisation by repeated elimination of points*, Cartographic Journal 1993, 30, 46–51.

[15] M. Bostock, *simplify.js (VW reference)*, 2012. Accessed 2017-12-08. (bost.ocks.org/mike/simplify/simplify.js)