Improving Connectivity of Airline Networks

Nayha Auradkar
Paul G. Allen School
University of Washington
nayhaa@cs.washington.edu

Anna Goncharenko
school of beep boop
University of Washington
angonch@cs.washington.edu

Logan Milandin
Paul G. Allen School
University of Washington
mi1andin@cs.washington.edu

Sanjana Sridhar
Paul G. Allen School
University of Washington
sanjgeek@cs.washington.edu

1 Introduction

There are many regions in the world that cannot be reached efficiently by airplane from relatively nearby regions, due to a lack of connecting airline routes. It may not always be obvious how a given region can best utilize its air travel infrastructure to better connect itself to surrounding regions of interest. This simple idea is the motivation for our work, where we present a system that uses existing airline route data to recommend new routes that increase the connectivity between pairs of weakly connected regions. Because adding airline routes can be expensive, our system also has the ability to assess a given well-connected region and recommend unimportant or redundant routes for removal (we reason that removing routes *between* regions that are already weaky connected would be couterproductive), with the intent of freeing up the resources a region would need to add the routes we recommend.

To make the notion of a “region" concrete yet flexible, our recommendation and deletion pipelines accept arbitrary subsets of airports (which we will call “communities") to operate on. We formulate both route recommendation and route removal as optimization problems over these communities, the first optimizing a variant of average shortest path length between communities (see section 4.2.1) and the latter optimizing this same metric within a single community. Since the number of potential new routes (pairs of airports) is orders of magnitude larger than the number of existing routes, we use a heuristic-based approach for the route recommendation pipeline but an exact optimization solution for the route deletion pipeline.

While we deem it important that our pipelines work with any subset(s) of airports (since a user may only have control over certain sets of airports, regardless of which we think are ideal to connect), we focus on sets that are densely connected within themselves but weakly connected to each other in developing and evaluating them. To obtain such communities, we clustered the global network of airports and airline routes using the Louvain algorithm. To tune our heuristic for recommending new routes, we manually selected several pairs of the resulting communities that have few existing routes between them but are relatively nearby geographically and found hyperparameters that optimized our shortest path metric over these. We assessed the quality of our tuned pipeline by comparing the reductions in path lengths resulting from adding routes it recommends to reductions in path lengths resulting from randomly selecting new routes. Our edge deletion system did not require hyperparameter tuning, but we evaluated its selection of redundant intra-community routes similarly by comparing the increase in path lengths resulting from removing routes it selects to the increase in path lengths resulting from randomly removing routes.

2 Related Work

Optimizing the airline network, as least as we're approaching it, is fundamentally a graph algorithm problem. As such, it draws on much prior work in this area from the past several decades. Perhaps most significant is the work of Blondel et al. [1] in developing the Louvain community detection algorithm, an agglomerative clustering algorithm for identifying communities in large sets of data that is now widely used. The Louvain algorithm is performed in two repeated phases that seek to optimize a modularity function. In phase one, each node is considered to be its own community. Then, communities are combined with neighboring communities in a way that gives the maximum modularity until no further modularity gain can be achieved. In phase two, nodes in each community are coalesced into "super nodes", producing a new network to pass back to phase one. The first and second phases are then reapplied, in theory until there is only one community left but in practice until modularity decreases by less than some threshold.

The Louvain algorithm is powerful and can operate efficiently on very large datasets. In comparison to other clustering algorithms such as Girvan et al's algorithm [2] or the Girvan-Newman algorithm, Clauset et al's algorithm [3] and simulated annealing, the Louvain algorithm is the fastest. Its time complexity on a graph with n nodes and m edges is $O(n \log n)$ in the average case, and linear time on sparse data. This is a significant improvement to the Girvan-Newman algorithm's $O(nm^2)$ runtime. Moreover, in addition to the final result of the algorithm, the hierarchical nature of the algorithm generates intermediate solutions that are often useful in analysis of community structure. Overall, the Louvain algorithm is attractive to us for its speed and interpetability. High quality library implementations exist in Python [4], so we utilize it to obtain the communities that we develop and evaluate our pipelines with.

In our exploration of other clustering alternatives, we came across a feature of one such algorithm that inspired the evaluation metric we developed for the quality of the connection between a pair of communities. The Girvan-Newman community detection algorithm works by deleting edges from the graph one at a time until it's separated into disjoint communities. It is far slower than the Louvain algorithm, but is often seen as a "gold standard" in community detection because it avoids greedy rules, instead performing an extensive analysis of every edge in the graph before picking the next one to remove. The way it decides which edges to remove on each step, which are the edges that go between the communities it eventually obtains, is what inspired our evaluation metric. It uses the notion of "betweenness centrality" [5] which is the number of shortest paths in a graph that pass through a given node. The idea is that for a quality partition, many shortest paths ought to pass through the edges that go between communities and fewer shortest paths ought to pass through the edges within a community. This led us to consider how adding routes between communities would produce many new shortest paths that utilize that new route, and how analyzing the lengths of those new paths compared to the length of the shortest paths before recommending an edge, we can get an idea of how well-connected two communities are.

We also looked for work similar to what we present here. While we found systems that attempt to optimize air travel infrastructure through path selection, taking into account properties of jet streams and flying altitudes [6], and systems that attempt to optimize air travel to maximize profits for airlines [7], we found none that attempt to recommend entirely new routes or the removal of existing routes to optimize travel from the standpoint of consumers. This illustrates the novelty of our work.

3 Data Collection

3.1 Datasets

The primary dataset we utilize is the publicly available Routes dataset from OpenFlights [8], which contains 67663 airline routes. Each route has the source and destination airport, which are of primary interest to us, as well as the airline.

We also utilized the publicly available Airports dataset, also obtained from OpenFlights [9]. The Airports dataset contains data regarding 7698 airports, including the city it is located in and the latitude and longitude of the airport. Joining these datasets, we build several dataframes to extract key information.

3.2 Preprocessing

We preprocess the raw datasets above extensively, both filtering out certain parts and computing new quantities that are of interest for our system.

3.2.1 Dataset Pruning

There is a handful of routes (less than 10) in the Routes dataset with the same source and destination airport. We're not interested in these (as taking off and landing in the same place will not help one travel between or within communities more effectively), so we removed them. Also, while the vast majority of routes present are bidirectional (planes fly both ways), about 1.5% are not. Because we use the Louvain algorithm to partition the airline network for its speed, simplicity, and interpretability, which operates on undirected graphs, we elected to remove these routes. We recognized this early on as an inherent limitation to our clustering method, but accepted it so we could focus our limited time on the recommendation pipeline rather than investigating clustering algorithms for directed graphs. Finally, we used NetworkX to find the largest connected component (LCC) of the global airline network and removed the 28 airports and 26 routes not belonging to it from our data. The reason for this is that our evaluation metric for recommended routes computes average shortest path lengths between pairs of airports, so comparing its value for a graph with and without our recommendations is only meaningful if the total number of shortest paths remains fixed. By definition, the LCC guarantees this; the number of shortest paths is fixed at $\binom{\text{\# airports in LCC}}{2}$.

3.2.2 Adding Great-Circle Distance

Our route data does not include flight distances, either in the form of actual distance flown or distance between the airports. We are still interested in these distances, however, so to approximate them, we compute the great circle distance [10] between pairs of airports using their latitude and longitude. The great circle distance, which we will denote between airports i and j as D_{ij} is defined as the shortest distance along the surface of a sphere, i.e. along a circumference of that sphere. Of course, we trivially have $D_{ij} = r\Delta\sigma$ given a radius r and central angle $\Delta\sigma$, so computing this distance reduces to computing the central angle $\Delta\sigma$ from latitude ϕ and longitude λ. We will not attempt to derive the formula here, but it's on the Wiki page linked above:

$$\Delta\sigma = \arccos(\sin\phi_i \sin\phi_j + \cos\lambda_i \cos\lambda_j \cos|\lambda_i - \lambda_j|)$$

We used this formula with the average radius of earth in kilometers $r = 6371$ and have verified that this distance approximation gives us an error on the order of 1% from true Earth distances. One major benefit of computing distance ourselves is that we can obtain the distances of candidate routes that don't exist yet during the edge recommendation step, which is information that we wouldn't have if we were using true flight path distances that deviate slightly from great circle paths. There is probably an API we could have used to get slightly more accurate Earth distances as a preprocessing step, but this would have been prohibitively slow to invoke for the many candidate edges that are considered during the recommendation step.

3.2.3 Partitioning Airports into Communities

To partition airports into communities used in the tuning and evaluation of our route recommendation and removal pipelines, we used the Louvain clustering algorithm. We summarized this algorithm above and now explain its steps in more formal detail, due to its pertinence to our system:

1. Assign each node its own community. Iterate through the nodes of the graph and assign each to the community of one of its neighbors (or keep the current community) according to which community change produces the highest modularity in the resulting partition. Modularity is defined as

$$Q(G,S) = \frac{1}{2m}\sum_{s\in S}\sum_{i\in s}\sum_{j\in s}\left(A_{ij} - \frac{k_i k_j}{2m}\right)$$

where $A_{i,j}$ is the edge weight between nodes i and j and k_i is the sum of weights of edges attached to node i. Repeat this iteration process until no more community changes yield a modularity gain (algorithm provides a theoretical guarantee of this convergence).

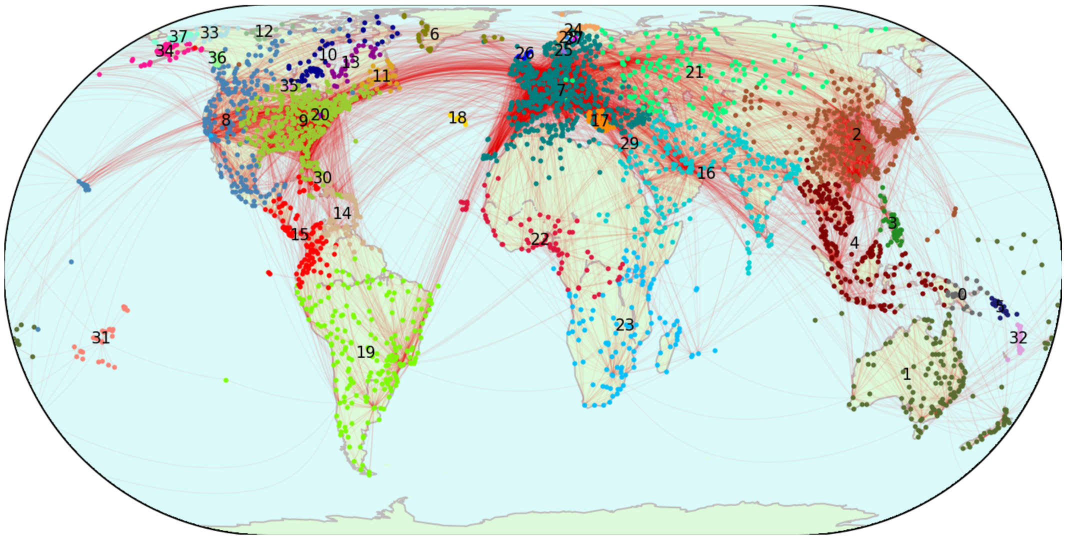


Figure 1: The partition of airports into communities obtained from running the Louvain algorithm. Communities are labeled with their euclidean centroids and airline routes are shown in red

2. Produce a new graph G' from the graph $G = (V, E)$ and partitioning of it S produced in the most recent iteration of step 1 in the following way. For each community $s \in S$, produce a single node v_s and give it a self edge with a weight equal to the sum of edge weights within that community:

$$w_s = \sum_{\{i,j\} \in E : i \in s, j \in s} A_{ij}$$

Create edges between these new nodes if there were any edges between the communities s, s' that produced them, with a weight equal to the sum of the edge weights between those communities:

$$w_{s,s'} = \sum_{\{i,j\} \in E : i \in s, j \in s'} A_{ij}$$

Go back to step 1 with this new graph as input.

Repeating the above two steps indefinitely will eventually yield a single super node, and by keeping track of the agglomeration tree we can select a number of communities to output by finding the level where the modularity stops increasing by as much (elbow point). The implementation provided in the package we are using does this for us, simply outputting the partition corresponding to this elbow point.

One decision we had to make here is what to use for the edge weights. We wanted this quantity to roughly reflect how convenient it is for a passenger to travel between a pair of airports, so we took into account route distance D as well as route multiplicity M (the number of airlines that fly it). We reasoned that raw multiplicity scales too fast (ten airlines flying a given route does not make using it ten times more convenient for a customer), so we settled on a weight function

$$w(r) = \frac{\sqrt{M_r}}{D_r}$$

Running the Louvain algorithm on our routes graph produced 38 communities, which are shown in figure 1.

3.3 Data Exploration

The following table present some statistics of interest regarding the original datasets before and after preprocessing:

Original Number of Routes	67663
Non-returning flights	949
Duplicate airline routes	29725
Routes not in LCC	46
Post-processing Number of Routes	36050
Original Number of Airports	3101
Airports in LCC	3073

The following tables present some statistics of interest regarding the communities obtained from the Louvain algorithm in preprocessing:

Cluster Sizes:

Number of Clusters	38
Largest Cluster Size	499 (Western Europe)
Smallest Cluster Size	2 (Eastern Greenland, Eastern Caribbean Islands)

Edges Between Communities:

Mean number inter-cluster edges	1.364
Std. of number inter-cluster edges	23.908
Max number inter-cluster edges	609
Min number inter-cluster edges	0
Number disconnected clusters	2

Edges Within Communities:

Mean number intra-cluster edges	712.053
Std. of number intra-cluster edges	1865.038
Max number intra-cluster edges	10454
Community with most intra-cluster edges	Western Europe
Min number intra-cluster edges	2
Community with least intra-cluster edges	Eastern Caribbean Islands

4 Methodology

4.1 Mathematical Background

In addition to a working knowledge of simple euclidean geometry (to follow our calculations of great circle distance above), readers will benefit from a basic understanding of heuristic functions and using them to guide a search or selection process. One difference worth noting between the heuristic function we use for route recommendation and the heuristic functions readers are likely to be familiar with is that we are combining features multiplicatively rather than additively. In other words, given a set of relevant features $f_1, ...f_n$ for some object of interest x, the typical way to combine those features into an overall heuristic score is

$$h(x) = \sum_{i=1}^{n} k_i f_i(x)$$

where k_i's are tuned coefficients representing the importance assigned to each feature. The heuristic we use, however, is of the form

$$h(x) = \prod_{i=1}^{n} f_i(x)^{k_i}$$

Our reason for using a multiplicative heuristic is to make the tuning easier when dealing with features that differ widely in magnitude; we can assign equal significance to each feature as a first step in the tuning process just by assigning each $k_i = 1$ (so doubling any feature doubles the result) whereas a linear combination would require us to find appropiately large coefficients for the smaller features before making much use of them.

4.2 Route Recommendation

The route recommendation component of our system accepts as input two subsets of airports C_1 and C_2 (a natural choice is two communities obtained by the Louvain algorithm in preprocessing, but these can be any two subsets) and a desired number of route recommendations k. We then generate a set of candidate routes

$$R = \{(i, j) : i \in C_1, j \in C_2, (i, j) \text{ isn't an existing route}\}$$

which is easily obtained by joining these two subsets of airports and filtering out any routes that already exist between them. We compute a heuristic score (see below) for each of these remaining candidate routes and return the top k.

4.2.1 Evaluation of Recommended Routes

In order to give us some way to compare any heuristics we might use for recommending routes, we require some notion of a "ground truth" score for the connectivity between pairs of communities that we can check against after adding our recommendations to the airline network. We reason that this quantity should reflect how must distance a person must cover or how much time they must spend to travel from airports in C_1 to airports in C_2 (or vice versa), and that it should be impossible to improve upon if the communities are fully connected, i.e. if there exists a direct flight between every pair of airports ($u \in C_1$, $v \in C_2$). We therefore use a version of the average shortest path length between such pairs of airports:

$$score\_pair(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{u \in C_1, v \in C_1} dist(u, v)$$

The obvious choice here is to define $dist(u, v)$ simply as the sum of great circle distances on the path from u to v. However, this is inadequate because it does not place any importance on the number of steps in the path; in reality, a path consisting of fewer flights is often far preferable to a path consisting of more flights even if the total distance covered is more for the path consisting of fewer flights. The reason for this, of course, is the overhead occurred during a layover; at the very least, a traveler has to get off the plane, board another one and wait for it to take off. We therefore introduce a layover penalty, a quantity which we assigned to routes during preprocessing for this purpose, to every flight (besides the first) of a given path between pairs of airports. For the new routes recommended by our pipeline, we assign a multplicity of two (this is what most routes in the existing network have, a single airline flying the route in each direction). We will use l to refer to layover penalties. As mentioned in the description of our preprocessing, these penalties are obtained by assigning a layover time between 1 and 3 hours based on how frequently a route is flown (which we estimate with route multiplicity) and then converting to the distance a typical plane could cover if it had kept flying for that amount of time. These penalties are somewhat crude and arbitrary, but they serve our purpose of weighting heavily against paths with more stops. We can formally express this total path "distance" in terms of the set of all paths $paths(u, v)$ (where a path is simply a sequence of edges/routes) between two airports u and v:

$$\min_{p \in paths(u,v)} \sum_{r \in p} (D_r + l_r \mathbf{1}[u \notin r])$$

Note the use of an indicator so that we only penalize stops (direct flights incur no penalty). Note also that under this definition, the distance of a shortest path can change slightly depending whether one is traveling from u to v or from v to u, so we instead call the above quantity dir_dist and define

$$dist(u, v) = \frac{1}{2}(dir\_dist(u, v) + dir\_dist(v, u))$$

4.2.2 Candidate Route Heuristic

Rather than immediately presenting the final utility heuristic we used, we will first provide the intuition and reasoning that led us to it. We experimented with a few quantities of interest in determining a good heuristic by which to order candidate routes. One is great circle distance; intuitively, we would prefer to add a route between disconnected communities that is short rather than long to optimize the evaluation metric above because if many inter-community shortest paths use this route (which is the intent), making it shorter makes at least one segment of those paths shorter. A short connecting route also makes it less likely that paths will have to backtrack, i.e. fly away from the intended destination in order to get to the airport that connects the communities. With this in mind, our utility function U should have the property

$$U(r) \propto D_r^{-k_D}$$

for some tunable nonnegative exponent k_D. Another quantity that turns out to be even more relevant is the degree of the airports we'd be connecting. If we add a route between two airports u and v that are densely connected to the rest of their respective communities, we create guaranteed three-hop paths between the many airports connected to u and the many airports connected to v, whereas connecting "fringe" airports only creates these three-hop paths between comparatively few pairs. To assign value to a candidate route based on this idea, we simply add this quantity for the two airports being connected (we informally call this sum "edge degree" and denote it with d). With this in mind, our utility function should also have the property

$$U(r) \propto d_r^{k_d}$$

for some tunable nonnegative exponent k_d. We also investigated the possibility of using a weighted degree, summing the multiplicities of an airport's routes rather than just counting the number of routes. The reasoning here is that if there are multiple airports with about the same unweighted degree, using any of them as one end of our recommended route would create about the same number of the efficient paths mentioned above, but the layover penalty would be worse for those with lower multiplicity routes attached. However, our experimentation showed that the original unweighted degree was the more informative quantity for optimizing our evaluation metric, indicating that the number of few-hop paths created is more important than the layovers for the paths created. This leads us to our final, ultimately rather simple heuristic:

$$U(r) = d_r^{k_d} D_r^{-k_D}$$

At this point, the savvy reader ought to ask, "why use a heuristic at all? Why not compute exactly the candidates which optimize the evaluation metric?" The reason is that $score\_pair$ takes on the order of a second to compute for most of the community pairs obtained from our Louvain partition. The number of airports per community is typically at least 100, yielding on the order of 10,000 candidate routes per community pair, so clearly we cannot afford a computation of $score\_pair$ for each of these candidates.

4.2.3 Heuristic Tuning

To tune the heuristic above, we manually selected 15 community pairs that we considered representative examples of what our system aims to fix. We first filtered to only communities with at least 50 airports (several of the communities obtained from the Louvain partition are rather tiny) and then selected the six closest pairs that have 0 connections, the three closest pairs that have 1 connection, and the two closest pairs that have each of 3, 4, and 5 connections. We reason that community pairs with significantly more connections than this are already reasonably well connected and not of much interest to our pipeline. We then randomly selected five of these pairs to be left out for a test set (since we're ultimately interested in how our heuristic performs on arbitrary pairs, not just those used to develop it), and used the remaining ten pairs to tune our hyperparameters k_d and k_D. Note that since our final heuristic only includes two of the features we experimented with, only the ratio of hyperparameters $\frac{k_D}{k_d}$ (not their magnitudes) matters for the ordering of candidate routes (since $d^{k_d} D^{-k_D} = \left(dD^{(-k_D/k_d)}\right)^{k_d}$ and $x < y \to x^k < y^k, \forall x, y, k > 0$). So we simply fixed $k_d = 1$ and tried many values for k_D ranging from 0 to 2 (we didn't bother checking any higher as performance was already dropping steeply at the upper end of this range). For each assignment to the hyperparameters, we computed route recommendations for each of the ten community pairs. Then, separately for pair of communities C_i and C_j, we added the recommended routes to the original airline network G to create a modified network G' and computed $score\_pair(C_1, C_2)$ over G'. We then simply averaged these ten scores to obtain an overall score for the hyperparameter assignment used. With this approach, the best value we settled on is $k_D = .3$ for a final tuned heuristic of

$$U(r) = d_r D_r^{-.3}$$

It is worth noting here that the optimal k_D was not exactly the same for different numbers of route recommendations, but we found that $k_D = .3$ was optimal for ≤ 3 route recommendations and near-optimal for more recommendations as well.

4.3 Removing Routes

One could argue that optimizing an airline network may involve not only adding new routes, but also removing existing redundant/unimportant routes. To support this use case, our system also includes a pipeline for suggesting existing routes for removal. The communities our pipelines are designed for are densely connected within themselves but sparsely connected to each other, so removing routes that pass between communities would be orthogonal to our system's intent. Thus, we focus our attention on the problem of finding and removing redundant routes between airports that reside within a single community. Our edge removal pipeline accepts a subset of airports C (representing a single community) and a desired number of route removals k and returns the k routes that produce the lowest evaluation score (see section 4.3.2) on the community subnetwork after their removal.

4.3.1 Determining Candidates for Removal

It's possible that removing certain routes within a community could entirely disconnect components of the airline network. Such routes certainly cannot be considered redundant or unimportant under any reasonable definition (in fact, we would argue these are the most important to keep), so we take special care to avoid considering these routes as candidates for removal. To this end, we obtain a subgraph $G_C = (C, E_C)$ consisting of all the nodes and edges within the community we are removing routes from, compute a Minimum Spanning Tree (MST) of G_C (with great circle distance as weights) and only consider routes in E_C for removal if they do not belong to the MST. An MST of a graph is defined as a subset of edges $E_{C,MST}$ that connects all the nodes in the graph and has a total edge weight no larger than any other spanning tree. As long as this MST for our community remains intact, we guarantee by definition that any airport in C remains reachable from any other airport in C. The edge removal problem thus reduces to ranking the non-essential $E_C \setminus E_{C,MST}$ edges to find the least important ones.

4.3.2 Removal Algorithm

Since the number of candidate edges for removal is quite small relative to the number of candidate new routes in our edge recommendation pipeline, our removal algorithm can afford to exactly optimize an evaluation metric rather than approximate an optimal solution with a heuristic. For each route $e \in (E_C \setminus E_{C,MST})$, we remove e from E_C to obtain a modified set of edges E'_C. We then compute the following evaluation metric for C using only the edges in E'_C when computing $dist$'s:

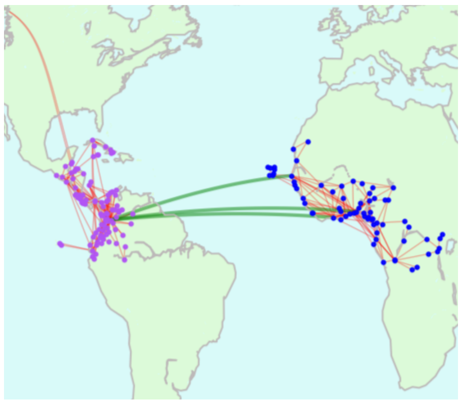
$$score\_single(C) = \frac{1}{\binom{|C|}{2}} \sum_{u \in C, v \in C : u \neq v} dist(u, v)$$

Note that this is the same as the $score\_pair$ (see section 4.2.1) metric, but defined over airport pairs within the same community rather than within different communities. Once we have computed this score for every route $e \in E_C \setminus E_{C,MST}$, we simply return the k routes that produced the lowest score.

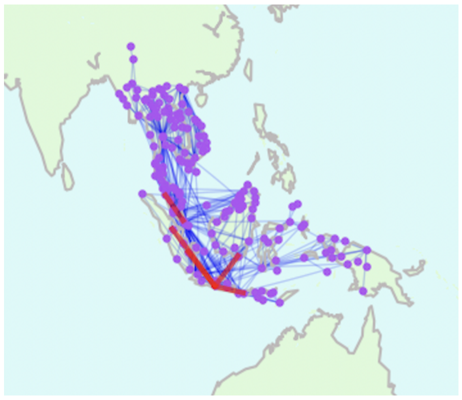
5 Results and Analysis

5.1 Route Recommendation Pipeline

As mentioned above, we selected 15 representative pairs of communities to develop our recommendation pipeline, leaving out five of them as a test set (the rest were used to tune our recommendation heuristic). To evaluate our tuned pipeline, we ran it on these five community pairs and computed an overall evaluation score (averaged over the pairs, in the same way we described doing in the tuning process) for each of one, two, three, four and five recommendations per pair. The evaluation score is only meaningful in comparison to other evaluation scores for the same community pairs, so to get an idea of whether our recommendations are any good, we compared these scores to what we get when recommending edges between community pairs at random instead of using our heuristic. The results of this test are summarized in figure 3. Evidently our recommendations produce scores that are quite a few standard deviations below what random recommendations produce, indicating that one would have to be quite lucky to guess routes that are as good as those recommended by our pipeline. This gives us some amount of confidence in our pipeline's quality. Figure 2 shows a few examples of community pairs from this test set with our system's recommended new routes.



(a) Existing routes shown in red, recommendations in green



(b) Existing routes shown in blue, suggested removals in red

Figure 2: Example route recommendations between community pairs (left) and removals within a single community (right)

5.2 Route Removal Pipeline

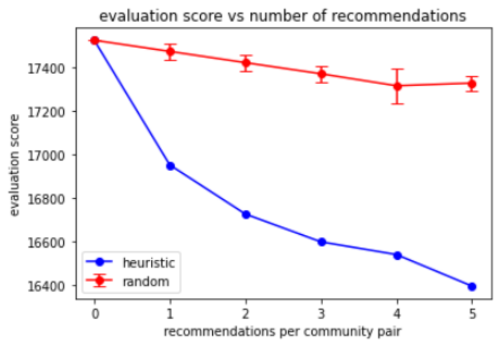
To evaluate our removal pipeline, we selected five mid-size communities and ran our pipeline to select one, two, three, four, and five routes for deletion from each, similar to how we evaluated our recommendation pipeline. We averaged the $score\_single$ values over each of the communities to obtain a single measure of performance. We also used a series of random trials here, randomly sampling from the set of non-MST edges for each community, to obtain a comparison indicating the quality of our removals. The results of this test are summarized in figure 3. Our route removals appear to increase the evaluation metric by much less than the random recommendations on average, indicating that our removals are indeed meaningful; however, we can see their scores are not nearly as far below the error bars for the random removals as our recommendation scores were below the random recommendations. In fact, for just one removal, our score actually overlaps the error bars. However, this trend makes sense if we consider the sparsity of the airline network; the number of candidate routes being considered for recommendation is always orders of magnitude larger than the number of existing routes being considered for deletion, so one would have to be much luckier to randomly guess the routes we recommend for the first pipeline than to randomly guess the routes we remove for the second pipeline. This is especially true when selecting fewer routes (hence the overlap with the error bars for just one recommendation); as we select more routes for removal, the random removals would have to get lucky more times in order to compete with our set of removals.

6 Limitations and Future Work

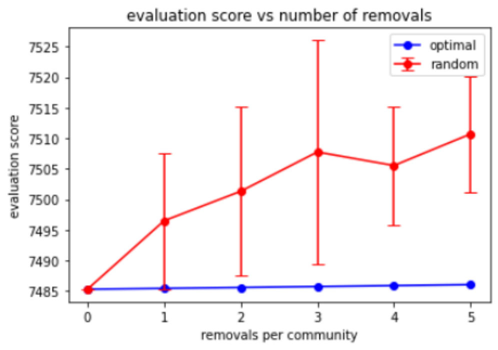
We believe that our our route removal and route recommendation pipelines together provide a usable toolset for optimizing airline networks. However, this work is far from perfect. For example, in Section 3.2.1, we describe how we had to prune some useful data, such as removing all routes that don't have flights in both directions in order to use an undirected clustering algorithm. We also made arbitrary assumptions, such as our assignment of layover penalties described in Section 4.2.1. Some additional limitations are listed below.

6.1 Route Recommendation Pipeline

For reasons described in our preprocessing, we use only the largest connected component of the airline network, meaning that our community partition does not contain any of the sub-networks that are disconnected from the remainder of the airline network. This is somewhat contradictory to the philosophy of our system because these isolated communities are in fact excellent candidates for becoming better connected with surrounding regions. The purpose of our recommendation pipeline is to facilitate travel from isolated communities to surrounding communities, but in order to use it for



(a) Route recommendation pipeline vs. random recommendations



(b) Route removal pipeline vs. random removals

Figure 3: A comparison of the performance of our route recommendation pipeline and route removal pipeline against random removals. We ran ten random trials for each number of recommendations (the error bars denote one standard deviation)

one of these extremely isolated communities, one would have to manually create a set containing those airports rather than using the communities obtained from our partition.

6.2 Route Removal Pipeline

Because we avoid the use of a heuristic in removing routes, instead electing to exactly compute the routes which optimize our evaluation metric, the route removal pipeline tends to be significantly slower than the recommendation pipeline (even though the number of candidate routes is much smaller). It becomes prohibitively slow to use on communities with more than a few hundred airports and more than a few thousand routes. In order to make the removal pipeline more universally functional, we would need to develop a heuristic for ranking routes for removal or come up with an evaluation metric that is faster to compute.

6.3 Future Work

Our system attempts to optimize the airline network using only data obtained from the structure of the network itself. We attempted to leverage other relevant data such as the population around each airport or within each region, but had issues with data cleanliness and were unable to integrate these statistics into our pipelines in a meaningful way. We therefore recommend that future work in this explore the use of other important data regarding the airline network, such as fuel costs, population, relationships between regions, and information about which parties are in control of which airports around the world.

7 Conclusion

We introduce a novel pipeline to recommend flight routes to add and remove from current airline networks to improve the connectivity of the world and use resources more efficiently. We utilize the Routes and Airports dataset from OpenFlights to build a graph where nodes are airports and edges are routes. We then use the Louvain algorithm to cluster parts of the graph into communities, use our heuristic to select top candidates routes to add, and use our evaluation metric to select top candidates routes to remove. Using our evaluation metric, we demonstrate that for both route addition and deletion recommendations, our algorithms perform significantly better than randomly choosing routes. In the future, we hope to support the use of our pipelines with more isolated communities, improve the efficiency of our algorithms, and include more data (such as population) to inform our algorithms. We are confident, however, that the system we present here can help regions better connect to the world.

8 Group Member Contributions

Nayha Auradkar: Researched Louvain algorithm and implemented clustering in code, helped with preprocessing data, implemented heuristics for choosing communities to connect, came up with and implemented edge removal algorithm. Wrote part of related work section, community clustering section, the multiple route removal sections, conclusion, helped with figure generation, and edited the final report. Worked on problem formulation, pipeline description, and route removal sections of the presentation.

Anna Goncharenko: Researched Girvan-Newman Algorithm. Implemented code for merging with various population datasets (omitted from final dataset), choosing pairs of communities for testing, helped create removal recommendation algorithm, implemented evaluation function for edge removal algorithm and helped write edge removal code, produced visuals for edge removal. Performed exploratory data analysis on processed datasets and clustered datasets. Worked on Louvain algorithm and final conclusion sections of the presentation, as well as editing the final video together.

Logan Milandin: Researched efficient implementations of clustering algorithms. Organized and implemented majority of preprocessing, route recommendation pipeline, and evaluation infrastructure including randomized trials. Documented pipeline infrastructure for ease of use. Helped design removal pipeline. Wrote majority of introduction, methodology, results, limitations and future work sections of this paper. Generated most figures included in this paper and our presentation.

Sanjana Sridhar: Researched betweenness centrality. Implemented code for establishing a layover penalty, hyperparameter tuning, and associated data preprocessing steps. Wrote part of the Data Collection section, created the citations, and edited the final report. Designed the final presentation and made the slides for the overview, preprocessing and evaluation sections and helped make the slides for route recommendation. Helped edit the final video.

9 References

[1] Blondel, V.D. & Guillaume, J. & Lambiotte, R & Lefebvre, E. (2008) Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment, vol. 2008, no. 10*, pp. P10008.

[2] Girvan, M. & Newman, M.E.J. (2002) Community structure in social and biological networks. *Proceedings of the National Academy of Sciences vol. 99, no. 12*, pp. 7821-7826.

[3] Clauset, A. & Newman, M.E.J. & Moore, C. (2004) Finding community structure in very large networks. *Phys. Rev. E vol. 70*, pp. 066111.

[4] “Louvain Algorithm.” PyPI, https://pypi.org/project/louvain/.

[5] Riondato, M. & Kornaropoulos, E. (2014) Fast Approximation of Betweenness Centrality through Sampling. *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM ’14)*, pp. 413–422. New York, NY: Association for Computing Machinery.

[6] Ng, H.K. & Sridhar, B. & Grabbe, S. (2014) Optimizing Aircraft Trajectories with Multiple Cruise Altitudes in the Presence of Winds. *Journal of Aerospace Information Systems vol. 11*, pp. 35-47.

[7] Lordan, O. & Sallan, J.M. & Escorihuela, N. & Gonzalez-Prieto, D. (2016) Robustness of airline route networks. *Physica A: Statistical Mechanics and its Applications vol. 445*, pp. 18-26.

[8] OpenFlights. (2014) Route database. *contentshare*. https://openflights.org/data.htmlroute. Accessed Jan 26 2023.

[9] OpenFlights. (2017) Airport database. *contentshare*. https://openflights.org/data.htmlairport. Accessed Jan 26 2023.

[10] “Great-Circle Distance.” Wikipedia, Wikimedia Foundation, 25 Jan. 2023, https://en.wikipedia.org/wiki/Great-circle_distance.