A Performant Approach to Visualize Large Scale Airline Network of United States

by

© Hailong Feng

(Supervised by Dr. Yuanzhu Chen)

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Computer Science

Department of Computer Science

Memorial University of Newfoundland

August, 2020

St. John's Newfoundland

Abstract

This project presents a highly performant approach to visualize the large scale airline network of United States. In order to achieve high performance on the large scale data set, a wide variety of visualization techniques are tested and compared to improve the efficiency of data display, such as data pre-processing, clustering, caching, dynamic content display, etc. The graphical user interface is also fine-tuned to further enhance information's density and conciseness. The final web application provides an efficient, concise and interactive environment to visualize the complex airline network of United States.

Keywords: Airline Network, Data Visualization, Large Scale Data

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Definition

Aviation and air travel has always been considered a key economic and social indicator in the modern world [1]. In the past decades, an increasing number of studies are conducted to explore the airline network and people's travel patterns [3] [4]. However, as the world population increases and becomes more interconnected, the visualization of such a large scale network becomes a significant challenge.

In this project, I attempt to design and develop a data visualization pipeline to efficiently display the airline network of United States (U.S.). The raw data is derived from the Bureau of Transportation Statistics [8] under the United States Department of Transportation, and contains thousands of airports as well as hundreds of thousands of domestic and international airline routes. Final result is displayed in a web interface via the Google Maps JavaScript API [7].

1.2 Motivation

The visualization of large scale data has always been considered a challenging yet meaningful task in the field of data science. As our society has entered a data-driven era, a lot of explorations and decisions are made through the visualization and analysis of collected data. Even simple statistics, when presented with visualization, can reveal interesting correlations [2].

By analyzing the visualization result of an airline network, researchers can explore passengers' travel pattern, improve the network's traffic efficiency, make predictions on the cities' economic growth, etc. In addition, the same geographical data visualization technique may also be applied to other application domains, such as the investigate of animal's migration or pandemic's transmission.

1.3 Proposed Solution

Our proposed solution utilizes the Google Maps JavaScript API to display airline routes on a map. As Figure 1.1 illustrates, the raw data is first cleaned and condensed by a pre-processing script, then the comma-separated values (CSV) files are parsed in the correct type, stored in a custom built data structure, and finally drawn on the map by the Google Maps API. Detailed implementation steps are discussed in Chapter 3 - Visualization of Airports and Chapter 4 - Visualization of Airline Network.

Figure 1.1: Visualization process

Chapter 2

Background

2.1 Google Maps API

The visualization of airline networks requires a maps API to display the map along with other user defined elements. In this project, the Google Maps JavaScript API [7] is chosen due to its easy learning curve, flexible API and accurate data.

The documentation provided by Google Maps API contains detailed explanation and comprehensive examples about the usage of the API, which makes the developer easy to learn and test. In addition, elements in the Google Maps API are highly customizeable, making it as the perfect tool for geographical data visualization.

2.2 Source of Data

The data to be visualized in this project is of two types, one is the airport's geolocation, the other one is airline traffic (i.e. edge lists). Each data type contains one U.S. domestic version and one international version. The data is obtained from

the Bureau of Transportation Statistics [8] under the United States Department of Transportation and stored as CSV file type.

As shown in Figure 2.1, airports' records are stored in row-major, with each record contains 3 attributes, i.e. International Air Transport Association (IATA) airport code, latitude and longitude.

```
BTI,70.13400268550001,-143.582000732
LUR,68.87509918,-166.1100006
PIZ,69.73290253,-163.00500490000002
ITO,19.7213993072509 77,-155.04800415039062
ORL,28.545499801635998,-81.332901000977
BTT,66.91390228,-151.529007
Z84,64.301201,-149.119995
UTO,65.99279785,-153.7039948
FYU,66.57150268554689,-145.25
SVW,61.09740067,-155.5740051
```

Figure 2.1: Format of airport data

An IATA airport code is a unique three-letter geocode for identifying a global airport. In most cases, the airport codes are named after the name of the city or airport [9], such as ATL for Hartsfield-Jackson Atlanta International Airport, MSP for Minneapolis-Saint Paul International Airport, JFK for John F. Kennedy International Airport, etc. Besides airport codes for individual airports, in some large metropolitan areas with more than one airports, a separated code is also assigned to represent the metropolitan areas as a whole, such as Beijing (BJS) for Capital (PEK) and Daxing (PKX), Toronto (YTO) for Pearson (YYZ), Bishop (YTZ), Hamilton (YHM), and

Waterloo (YKF), or Washington, D.C. (WAS) for Dulles (IAD), Reagan (DCA), and Baltimore–Washington (BWI).

Figure 2.2 demonstrates the format of the edge list data. The first two columns represents the departure and destination airports of the route, the last column indicates the passengers' amount in thousands.

```
ALB,YUL,21975
ALB,YYZ,136220
ALB,FRA,1094
ALB,BDA,43
ALB,CUN,261
ALB,YOW,26276
ALB,NAS,202
ALB,CDG,1291
ALB,YHZ,1018
ALB,YQB,697
```

Figure 2.2: Format of edge list data

As Table 2.1 shows, both the airport data and the traffic data are at extremely large scale. Thus performance metrics should be closely monitored during the visualization process.

Table 2.1: Number of records

	Domestic	**International**
Airport	1,202	5,268
Edge list	211,136	658,572

2.3 Testing and Deployment

To ensure a consistent visual experience across multiple platforms, the project is built upon pure vanilla JavaScript codes in ES6 [14] syntax. The web page is tested in various web browsers (Chrome, Safari and Firefox) as well as on different devices (Personal Computer, iPhone and iPad). Git is used as the version control tool over the entire course of the development process.

The web project is deployed on the GitHub server through `GitHub Pages`. `GitHub Pages` is a free static site hosting service that can publish the website directly from a GitHub repository [6]. The visualization result can be viewed online at `https://dyckia.github.io/air-travel-patterns-visualization/`.

Chapter 3

Visualization of Airports

3.1 Basic Environment Setup

The first task for the visualization project is to display the map itself in the web browser, which involves the combination of HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript. By applying the principle of separation of concerns, the CSS and JavaScript components are created separately and are connected by referencing the respective file in the HTML `<head>` tag. This architecture pattern exhibits significant convenience in terms of maintainability and readability in the later stage.

3.1.1 HTML File Structure

As shown in Figure 3.1, the HTML source code contains a `head` element that loads necessary scripts and style sheets and a `body` element that holds the map.

In the `head` element, both Google Maps' JavaScript API and a custom script

`app.js` are loaded. The first two `script` tags specifies the Uniform Resource Locator (URL) of the Maps API as well as the user's API key. As stated in Section 2.1, user must have an API key in order to use the Maps JavaScript API. In general, a API key is a unique identifier that is used to authenticate requests associated with the user's project for usage and billing purposes. This API key is obtained through the `Google Cloud Platform Console` [5]. The third `script` tag points to the main JavaScript file that is developed in this project which initializes the map interface and sets the initial view position and zoom level, which is discussed in detail in Section 3.1.3.

In the `body` element, a `div` container is created which will be served as the viewport of our map. An `id` attribute is attached to this `div` container in order to be located and manipulated in the following stages by the JavaScript and CSS files.

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5    <title>Air Travel Patterns Visualization</title>
6    <script src="https://polyfill.io/v3/polyfill.min.js?features=default"></script>
7    <script defer src="https://maps.googleapis.com/maps/api/js?key=APIKEY&callback=initMap"></script>
8    <link rel="stylesheet" type="text/css" href="./style.css" />
9    <script src="./app.js"></script>
10 </head>
11
12 <body>
13   <div id="map"></div>
14 </body>
15
16 </html>
```

Figure 3.1: HTML file

3.1.2 CSS File Structure

The CSS file contains customized styles to the the `map` container. Due to the nature of data visualization project, a viewport with the maximized screen size is preferred. Thus the `map` container is set to take up 100% of the HTML `body`'s height. In standards mode, most current web browsers renders all elements with percentage-based sizes based on the size of their parent block elements [7]. In other words, if any of those ancestors fail to specify a size, they are assumed to be sized at 0 x 0 pixels. For that reason, both the heights of the `div` container and the `body` element have to be specified as 100%.

```
/* Always set the map height explicitly to define the size of the div
   element that contains the map. */
#map {
  height: 100%;
}

/* Makes the map fill the whole window. */
html,
body {
  height: 100%;
  margin: 0;
  padding: 0;
}
```

Figure 3.2: CSS file

3.1.3 JavaScript File Structure

The JavaScript file, namely `app.js`, comprises the function to initialize the map. Function `initMap()` will be called automatically when the web page is loaded. In

that function, a `google.maps.Map` instance is created which represents a single map on a page. By specifying the id of the element, the map is displayed inside the `div` container we created in the HTML file. In addition, two more options are passed in the JavaScript Object Notation (JSON) object as the second argument. In specific, `center` defines the initial view center of the map and `zoom` sets the initial zoom level of the map. Zoom level starts from 0, with a higher number indicating a more detailed location.

```
1  // main function to initialize the map
2  function initMap() {
3      //  a new map is created by instantiating a google.maps.Map object
4      const map = new google.maps.Map(document.getElementById("map"), {
5          center: { lat: 38.024009, lng: -97.081117 },
6          zoom: 4,
7      });
8  }
```

Figure 3.3: Main JavaScript file

As the goal of the project is to visualize airline traffic related to U.S., the center and zoom level are empirical set to {`lat: 38.024009, lng: -97.081117`} and `4` respectively.

3.2 Read Airports Data

As stated in Section 2.2, the airports data is attained from the The Bureau of Transportation Statistics [8] and is formatted as CSV file. Each row represents a record of airport's geolocation in the order of IATA airport code, latitude and longitude.

Figure 3.4: Initial map center

The airport data file is store in the path of `root/data/`. As shown in Figure 3.5, a custom middleware function is created to read the content of a given CSV file. The target CSV file is acquired by first making an AJAX call to the server, followed by a verification of the request status to ensure that only valid data is returned.

By making this middleware loosely coupled with the airport parser, same function can also be called by the traffic parser in the later stage, resulting in a concise and more maintainable data pre-processing pipeline.

The `readCSVFile` function returns the content of the CSV file as a string. In order to access the coordinates of each airport record, the string needs to be deserialized to an array of airport records. Figure 3.6 demonstrates this deserialization process in pseudo code. The function first splits the content string by the newline control character `\n` to get an array of airport records. The function further splits each record into three segments, namely airport code, latitude and longitude. Each segment is

```
// read given csv file via an AJAX call
function readCSVFile(file) {
    const rawFile = new XMLHttpRequest();
    rawFile.open("GET", file, false);
    let content;
    rawFile.onreadystatechange = function () {
        if (rawFile.readyState === 4) {
            if (rawFile.status === 200 || rawFile.status === 0) {
                content = rawFile.responseText;
            }
        }
    }
    rawFile.send();
    return content;
}
```

Figure 3.5: readCSVFile function

then formatted by removing any leading or trailing spaces and converting to the correct type. Finally, an airports array is created by appending each airport record `[code, lat, lag]`.

3.3 Visualize Airports on the Map

3.3.1 Visualize U.S. Airports

With the airports' geolocation successfully extracted from the CSV file, we can now visualize these airports on the map.

Google Maps identifies a location on the map with a marker. A marker by default is displayed as a red pin icon pointing to the targeted location. In Google Maps API, adding a marker is through creating a `google.maps.Marker` instance and attach

```
# return an array of airports' coordinates
# i.e. [[code, lat, lag], [JFK, 40.63980103, -73.77890015], ...]

func parse_airport(file_path):
    string_content = readCSVFile(file_path)
    rows = string_content.split_by(\n)
    airports = []
    for row in rows:
        segments = row.split_by(,)
        # remove any leading and trailing spaces
        airport_code = segments[0].trim()
        # convert string to number
        lat = double(segments[1].trim())
        lng = double(segments[2].trim())
        airports.append([airport_code, lat, lng])
    return airports
```

Figure 3.6: Pseudo code of parseAirport function

that marker to the desired map. As shown in Figure 3.7, the `google.maps.Marker` constructor can accepts a wide variety of arguments, including but not limited to:

- `position` - to identify the marker's geolocation (in JSON format {`lat: xx.xx, lng: xx.xx`}).
- `map` - to identify the map on which this marker is displayed.
- `label` - to label the icon with a user defined text.
- `icon` - to replace the default red pin with a customized icon.

During the test of visualizing airports markers on the map, the result is acceptable when the amount of markers is small (tens or hundreds of markers), however the performance started to decrease drastically as the markers' number grew. When the number of airports reached over one thousand, the move and zoom action of the map

```
// a new marker is created by instantiating a google.maps.Marker object
const marker = new google.maps.Marker({
    position: { lat: 40.6413, lng: -73.7781 },
    icon: pathOfIconFile,
    map: mapCreated,
    label: labelText
});
```

Figure 3.7: Marker's options

started to become unresponsive, which are detrimental to the user experience. In addition, the markers became too crowed to be recognizable by the user. Thus a more efficient approach is required to display the large number of airports.

3.3.2 Optimize Performance by Clustering Markers

Google Maps API provides a library called `MarkerClustererPlus` to display a large number of markers efficiently on a map by grouping markers in clusters. This library utilizes the grid-based clustering technique that divides the map into squares of a certain size.[10] The size of the grid changes dynamically based on the current zoom level. At each zoom level, it first generates a cluster at a particular marker, then iteratively combines markers that are in its bounds to the cluster. This process repeats until all markers within the certain grid size are assigned to the nearest marker clusters. If a marker is within the bounds of multiple clusters, `MarkerClustererPlus` library allocates it to the closest cluster in distance.

As Figure 3.8 shows, clusters are displayed as colored circles. Different colors representing different sizes of markers. The number on a cluster further indicates how many markers it contains. As the user zoom in or zoom out the map, clusters

are dynamically expanded or consolidated.

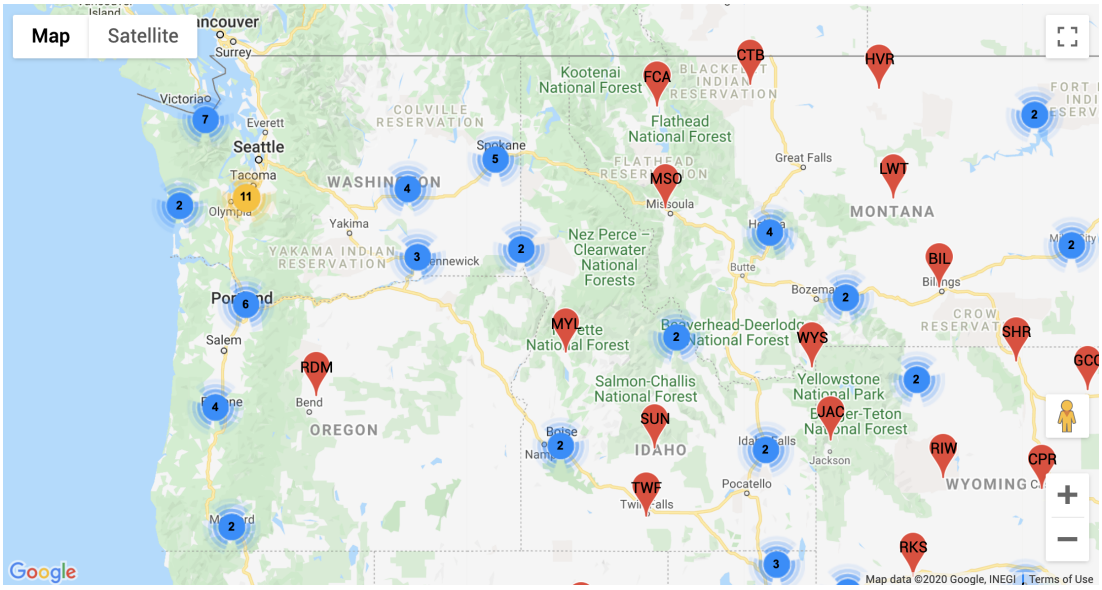

Figure 3.8: Markers in clusters

By integrating the `MarkerClustererPlus` library with the JavaScript API, the number of markers on the map has been significantly reduced, thus the visualization performance is substantially improved and the representation is greatly the simplified.

3.3.3 Visualize International Airports

With the pipeline built for the parsing and visualization of airports data, international airports can be conveniently displayed by simply plugging in the international airports data.

Figure 3.9 demonstrates the visualization result of both domestic airport data and international airport data.

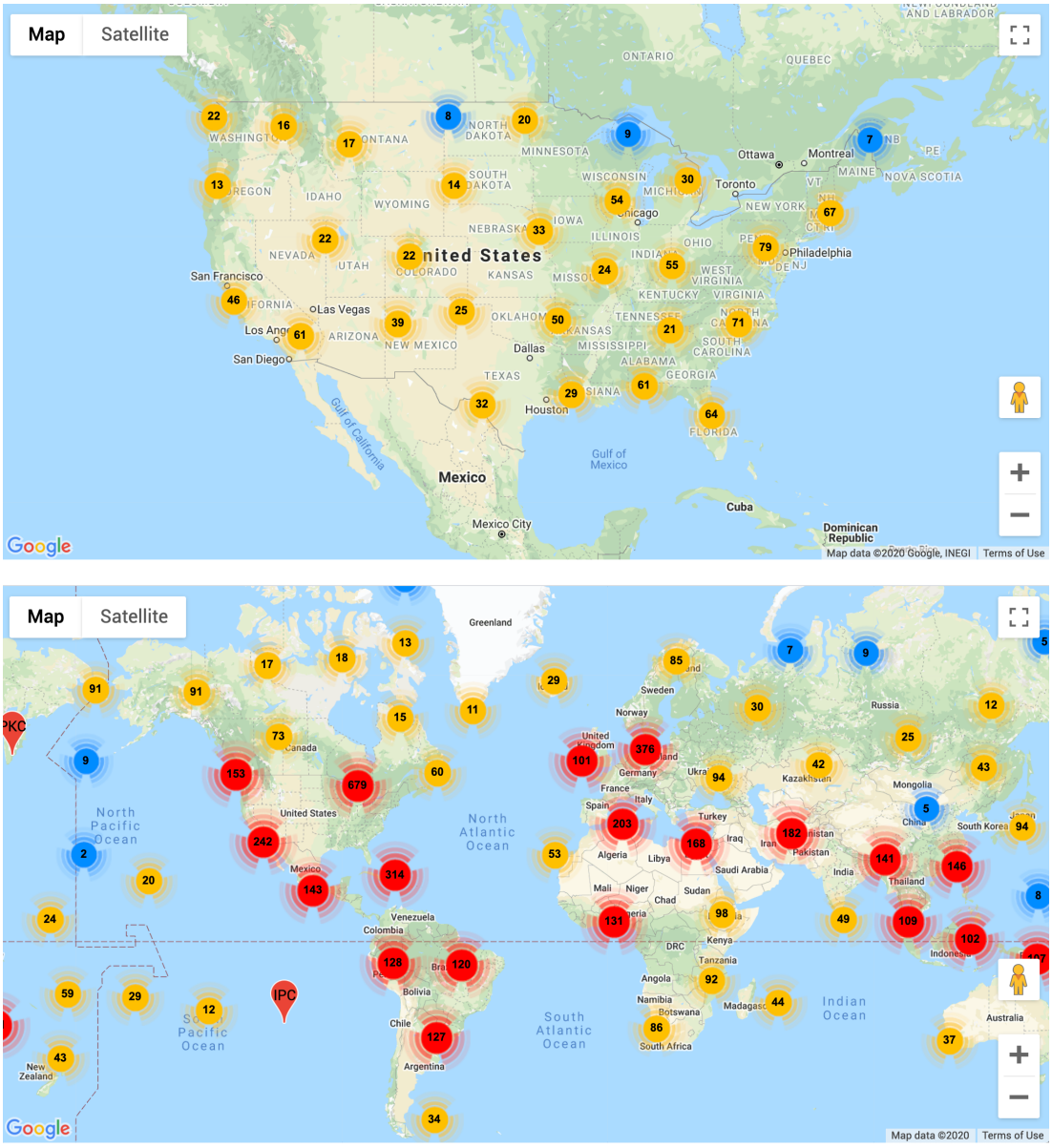

Figure 3.9: Airport visualization result (top: domestic airports, bottom: international airports)

Chapter 4

Visualization of Airline Network

4.1 Read Airline Traffic Data

As explored in Section 2.2, an airline traffic record contains the IATA airport codes for both the departure airport and the destination airport, as well as the number of passengers of that airline route (in thousands).

Thanks to the modular architecture of our visualization pipeline, same `readCSVFile` can be re-used for opening the edgelist CSV file and returning its whole content as a string, which avoids duplicated code and improves the maintainability of the project's source code.

Similar to Figure 3.6, a custom parsing function `parseEdgeList` is built to parse the CSV content and convert the value to the correct type. The function first splits the content string by the newline control character `\n` to get an array of edge list records. The function further splits each record into three segments, namely airport vertex one, airport vertex two and the edge's weight (i.e. passengers' count). Each

segment is then formatted by removing any leading or trailing spaces and converting to the correct type. Finally, an edges array is created by appending each edge record `[code1, code2, passenger]`.

```
# return an array of airlines' traffic
# i.e. [[code1, code2, passenger], [JFK, LAX, 335], ...]

func parse_edgelist(file_path):
    string_content = readCSVFile(file_path)
    rows = string_content.split_by(\n)
    edges = []
    for row in rows:
        segments = row.split_by(,)
        # remove any leading and trailing spaces
        airport_code1 = segments[1].trim()
        airport_code2 = segments[2].trim()
        # convert passengers' count from string to number
        count = int(segments[3].trim())
        edges.append([airport_code1, airport_code2, count])
    return edges
```

Figure 4.1: Pseudo code of parseEdgeList function

4.2 Visualize Airline Routes on the Map

4.2.1 Display Airline Routes as Polylines

The Google Maps JavaScript API provides a straightforward method to draw a line between two end points by utilizing the `Polyline` class. The `Polyline` class defines a linear overlay of connected line segments on the map. Users can specify custom styles (such as line color, opacity, or weight) for those line segments by passing the

`PolylineOptions` when constructing the `Polyline` object [11]. Figure 4.2 demonstrates the constructor arguments when creating a `Polyline` instance:

- `path` - to define the endpoints for each segment (a polyline can contain multiple line segments).
- `strokeColor` - to specify the colour of the of the line (in hexadecimal format).
- `strokeOpacity` - to specify the opacity of the line's color (value is between 0.0 and 1.0).
- `strokeWeight` - to specify the width of the line in pixels.

```
1  // define polyline's endPoints
2  endPoints = [
3      { lat: 37.772, lng: -122.214 },
4      { lat: 21.291, lng: -157.821 }
5  ];
6
7  // a new polyline is created by instantiating a google.maps.Polyline object
8  const line = new google.maps.Polyline({
9      path: endPoints,
10     geodesic: true,
11     strokeColor: '#78c2b7', // color in hexadecimal value
12     strokeOpacity: 0.8, // opacity ranges from 0 to 1.0
13     strokeWeight: 1 // weight of poly line in pixel
14 });
15
16 line.setMap(mapCreated);
```

Figure 4.2: Polyline's options

4.2.2 Improve Performance by Modifying Data Structure

As discussed in Section 4.2.1, in order to display airline routes on the map, we need to specify the end points of the line segment in the format of {`lat: xx.xx, lng: xx.xx`}. However, the `edges` array only specifies the end points as IATA airport codes, not as the geolocation coordinates. In other words, the coordinates must be retrieved through the airports array by referencing the airport codes.

As the pseudo code states in Figure 3.6, the airport records are stored as an array. Given a random airport code, the amortized time for retrieving the corresponding coordinate is $O(n/2)$, with n being the length of the `airports` array. As a result, the total time complexity for retrieving m airport codes would be $O(m * n/2)$, which is not ideal considering the large scale of our data records.

Because of this, a modification of the airports' data structure is proposed to improve the efficiency in retrieving airports' geolocation coordinates. The modification converts the `airports` array to a dictionary, with airport code being the key and its coordinates being the value. The dictionary data structure guarantees constant time for retrieving the coordinate through a given airport code, which reduces the total time complexity to be linear ($O(m)$).

4.2.3 Add Missing Airports' Coordinates

During the experimentation of airline traffic visualization, it is noticed that only partial traffic records are successfully displayed on the map. The reason of this display error is that there are airport codes in the edges list data that did not appear in the airports data, causing the system could not determine the end point's location

of the `Polyline`.

In order to resolve this issue, a custom script is written to generated all the airport codes whose coordinates are missing in the airports data. By examining the missing airports list, it is found that the missing airport codes are of two types: metropolitan area codes or individual airport codes. For a missing metropolitan area code, a coordinate is added by using the geographical center of that metropolitan area. For a missing individual airport code, a coordinate is added by referring to public online airport databases such as air-port-codes [12] or Wikipedia [9].

4.2.4 Improve Performance by Aggregating Records

Once all the airline routes are displayed on the map, interactions with the map becomes unresponsive again due the sheer number of airline routes. Optimization must be made to reduce the number of `Polyline`s.

By examining the edge list records stored in the CSV file, it is identified that there are multiple records with the same airport pair in the data. Thus a pre-processing method is proposed to combine passengers' count with same airport pair. Figure 4.3 shows the pre-processing steps in pseudo code. A nested dictionary data structure is used to store and add-up the passengers' count. For each edge list record, the function first checks the existence of the airport pair. If pair exists, the count for the same pair is retrieved and updated. If not, the dictionary will store that airport pair and its count.

As our edge list data is static, it is not necessary to pre-processing the same data every time when the page refreshes. Instead, we can cache the aggregated result by

```
# pre-process the edgelist data
# combine passengers' count of same airport pairs
# save aggregated results to a new csv file

func pre_process_edgelist(input_file, output_file):
    rows = parse_CSV_by_row(input_file)
    # nested dictionary to store unique airport pairs
    # i.e. {JFK: {DTW: 2, ATL: 3}}
    dic = {}
    # aggregate count
    for row in rows:
        # each row is an array of traffic record [JFK, DTW, 2]
        count = row[2]
        key1 = row[0]
        key2 = row[1]
        if key1 in dic:
            sub_dic = dic[key1]
            if key2 in sub_dic:
                # duplicated airport pairs found, combine count
                sub_dic[key2] += count
            else:
                # airport pair is unique, save count
                sub_dic[key2] = count
        else:
            # airport pair is unique, save count
             dic[key1] = {key2: count}
    # save results to the output csv file
    output_array = []
    for key1 in dic:
        sub_dic = dic[key1]
        for key2 in sub_dic:
            row = [key1, key2, sub_dic[key2]]
            output_array.append(row)
    save_array_to_file(output_array, output_file)
```

Figure 4.3: Pseudo code of pre-processing script

saving the processed data in a separated CSV file. In this case, the data pre-processing step is only executed once, which further improves the efficiency of the visualization

pipeline.

Table 4.1 shows the number of edge list records before and after data pre-processing. For both domestic data and international data, the number of `Polyline`s are reduced more than 98%.

Table 4.1: Number of records before and after processing

	Before processing	**After processing**	**Reduced By**
Domestic data	211,136	3,610	98.29%
International data	658,572	8,357	98.73%

With the implementation of the data pre-processing stage, the visualization performance is drastically increased and user's interactions with the map become highly responsive.

4.3 Visualize Passengers' Count

With the airline routes being efficiently displayed on the map, we can start visualizing the traffic amount. Based on the common data visualization practice and the style options offered by the Google Maps JavaScript API (Section 4.2.1), there are three approaches available to interpret the passengers' count:

1. Represent the passengers' count as the width of `Polyline`. The width of the line is set to be proportional to the number passengers. As shown in the top section of Figure 4.4, the result is not ideal due to the massive number and the immense density of airline routes. The route with wide line width (high

passenger volume) will interfere the nearby routes, making the less prominent routes hardly conceivable.

2. Represent the passengers' count as the color of `Polyline`. The rational behind this visualization technique is similar to representation of a heat map. Flight routes with high passenger volume are displayed in colors on the red side of the spectrum, while routes with low passenger volume are displayed in colors on the purple side. The middle section of Figure 4.4 demonstrates the visualization result in this approach. As the image shows, routes with different colors are not highly distinguishable from each other.

3. Represent the passengers' count as the opacity of `Polyline`. Similar to the width method, the opacity is set to be proportional to the passengers' count. As discussed in Section 4.2.1, the range of the opacity is from 0 to 1. Lines with higher opacity (less transparent) represent high volume route, while lower opacity (more transparent) represent route with less passengers' count. As shown in the bottom section of Figure 4.4, the result is both concise and visually perceivable.

4.4 Enhancements on User Experience

4.4.1 Customize Airport Marker Icon

When inspecting the result of airline routes, it is noticed that the `Polyline`s are not displayed properly together with `MarkerClusterer`s. In specific, if a `Polyline` is

drawn on top of a cluster, there is a high chance that `Polyline` is being cut by the cluster icon. Figure 4.5 demonstrates this display glitch.

In addition, with the airports' markers being hidden by the clusters, it is difficult for the user to determine which airport vertex the route edge belongs to. Because of the above two reasons, an alternate way to efficiently display the airport markers is required.

Through testing and investigation, it is found that the drastic drop of the map's performance in Section 3.3.1 is because the drawing of marker's icon and text label is extremely expensive.

To reduce the render cost of airport markers, a customized icon is used to replace the default pin icon. The new icon is a red dot in 5x5 pixel size. Comparing to the size of the original marker (22x40), the new icon reduces the pixel counts by over 97%.

In addition, airport labels are also removed to further improve the map's performance. Instead of displaying all the airport codes constantly on the map, an airport code is only shown when the user needs it. As shown in Figure 4.6, when the user hovers the mouse over an airport icon, an info window with the corresponding airport code will appears by the cursor. If the user moves the mouse away from the airport icon, the info window will disappear automatically.

With the above two modifications made to the markers' icon, the map is now able to render all the markers without the `MarkerClustererPlus` plugin. Besides, the end points of airline routes becomes more appreciable.

4.4.2 Prioritize Information Display

Google Maps by default displays a wide variety of information on the map, such as terrains, landmarks, roads, places of interest (POI), etc. This information is useful for navigation or exploration purposes. However, in our use case, this information is not necessary, and may even unintentionally creates noise to the visualization result. Thus map customization is essential to prioritize information of importance and remove unnecessary elements.

The Google Maps JavaScript API offers an convenient and comprehensive way to modify a map's style. As shown in Figure 4.7, the styles are specified by passing in an array of JSON objects, with each JSON object defines an `elementType` and its value. In addition, the Maps API provides a Styling Wizard [13] which allows the user to interactively generate a JSON styling object in the web browser.

As the primary purpose of this visualization project is to display airports and the airline network, the dominant color of the map is set to dark, density of roads and landmarks' labels is set to minimum. Figure 4.8 demonstrates the final visual style of the map. The appearance after customization is much cleaner and more recognizable comparing to the original theme.

Figure 3.9 illustrates the visualization result of both domestic airline routes and international airline routes.

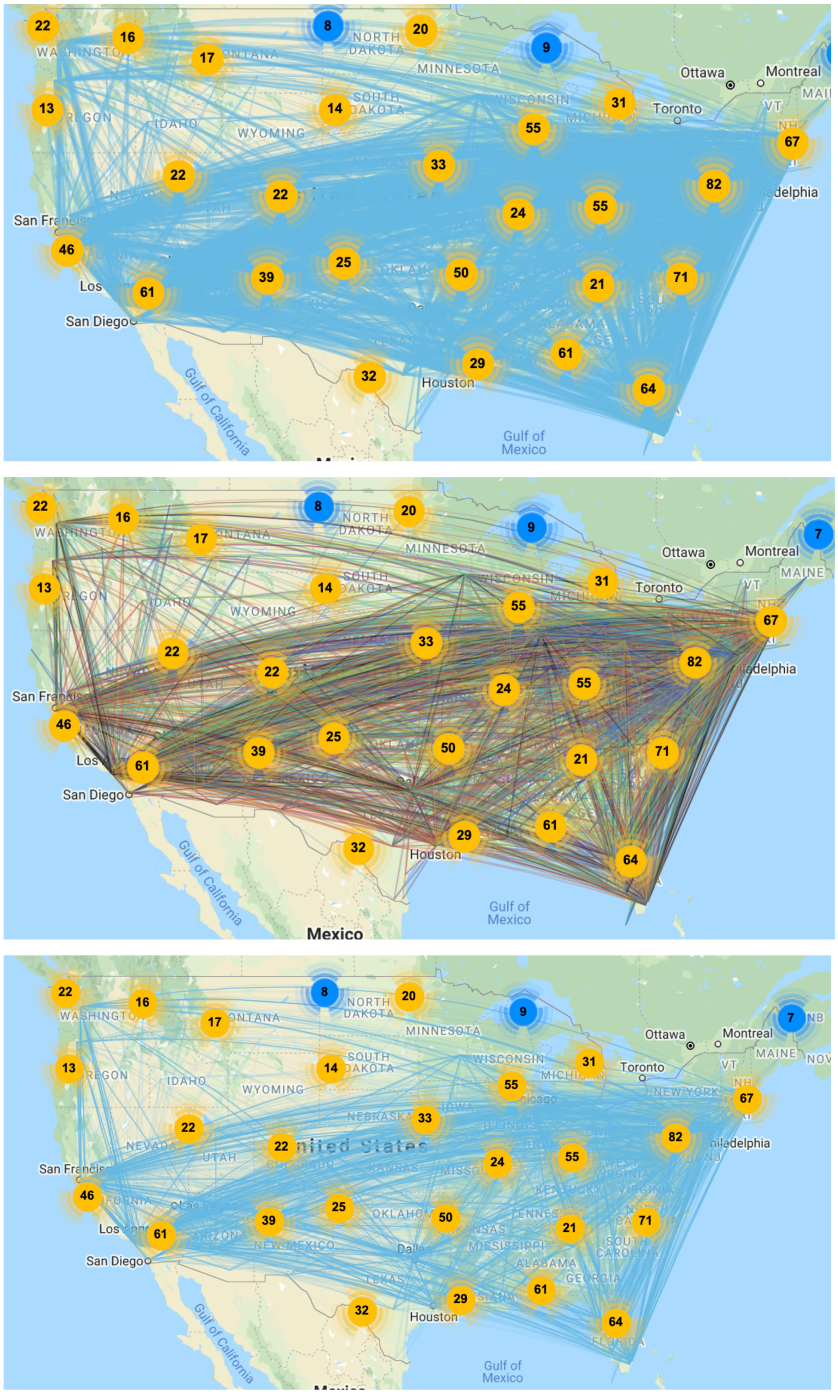

Figure 4.4: Different visualization strategies (top: width, middle: color, bottom: opacity)

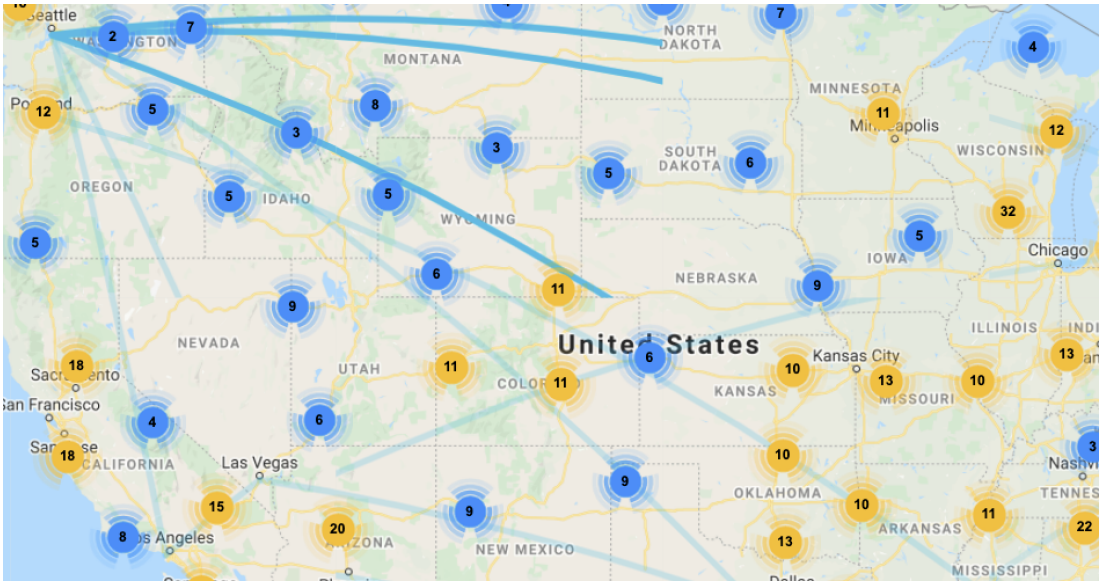

Figure 4.5: Display glitch

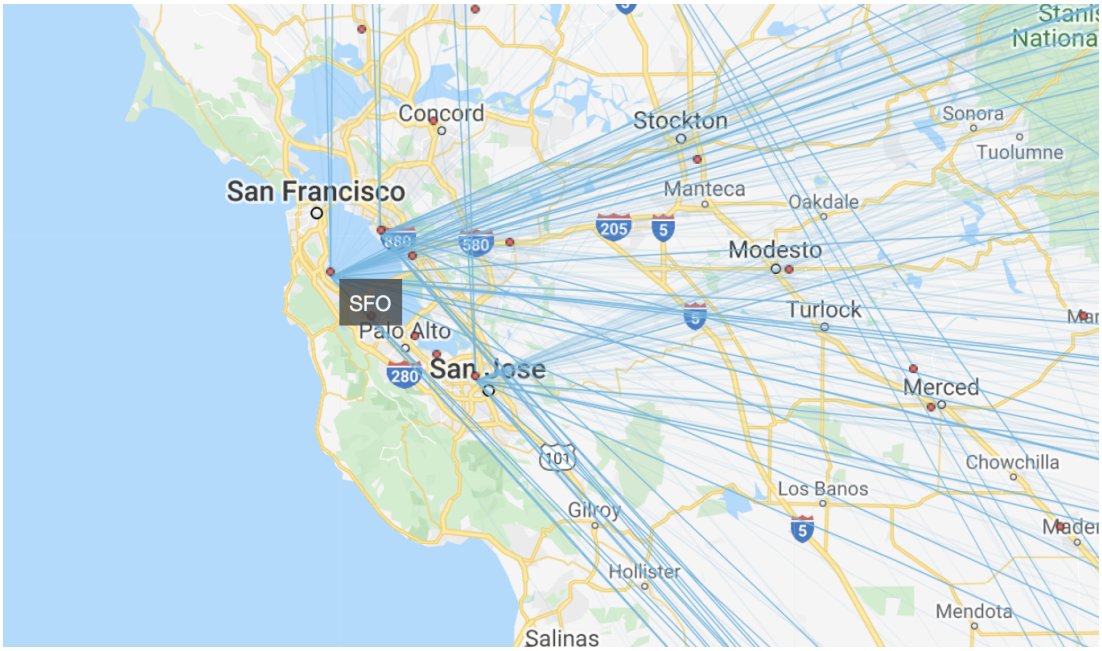

Figure 4.6: Info window

```
// define custom style for the map
const customStyle = [
    {
        "elementType": "geometry",
        "stylers": [
            {
                "color": "#242f3e"
            }
        ]
    },
    {
        "elementType": "labels.text.fill",
        "stylers": [
            {
                "color": "#746855"
            }
        ]
    },
    {
        "featureType": "water",
        "elementType": "labels.text.stroke",
        "stylers": [
            {
                "color": "#17263c"
            }
        ]
    }
]

// pass style as a parameter to the map
const map = new google.maps.Map(document.getElementById("map"), {
    center: { lat: 38.024009, lng: -97.081117 },
    zoom: 4,
    styles: customStyle
});
```

Figure 4.7: Style options

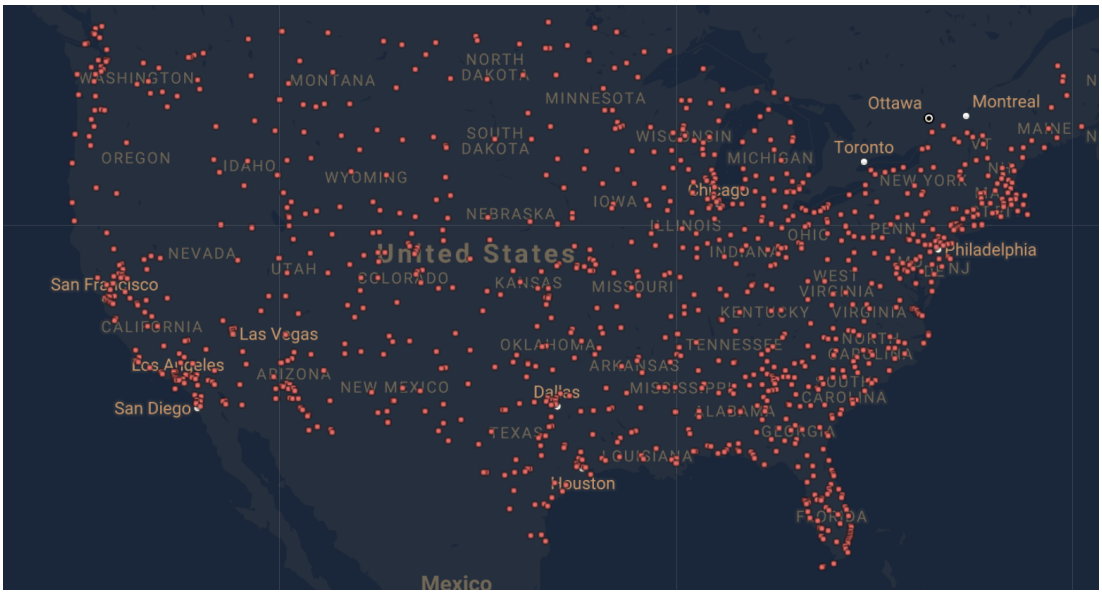

Figure 4.8: Map style after customization

Figure 4.9: Airline network visualization result (top: domestic routes, bottom: international routes)

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this project, I developed a highly performant data visualization pipeline for displaying the large scale airline network of U.S. In order to achieve high performance on the large scale data set, a variety of visualization techniques are used to enhance the efficiency, such as data pre-processing, caching, dynamic content display, etc. Different visual effects are compared to interpret the routes' traffic amount. In addition, the map's User Interface (UI) is customized and fine-tuned to further increase information's density and conciseness. The final application provides an efficient, concise and interactive environment to visualize the complex airline network of U.S.

This data visualization pipeline provides a generalized approach to display large scale complex network. Same pipeline can be easily applied to other application domains such as the investigate of animal's migration or pandemic's transmission.

5.2 Future Work

Besides the improvements stated in Section 4.4, some additional features could be implemented to further enhance the user's experience.

On the map, the airport's IATA code is dynamically displayed when the cursor is hovered upon an airport's marker. While the IATA code is short and precise for people who are familiar with it, it is obscure to normal users. We can utilize the info window that is created in Section 4.4.1 by adding additional information about the airport, such airport's full name and its location. In addition, such dynamic info window can also be applied on airline routes to display information such as airports of departure and landing and as well as the passengers' count of that route.

Moreover, for users who want to further investigate a specific airport or a certain type of routes. A filter sidebar can be developed which displays or hides the routes based on user's criteria. By interacting with the filter options, a user can select routes that are only from or to particular airport or in a certain range of passengers' amount, hiding unnecessary information on the map.

Bibliography

[1] G. Albrecht, H.-T. Lee, and A. Pang. Visual analysis of air traffic data using aircraft density and conflict probability. 06 2012.

[2] T. Dey, D. Phillips, and P. Steele. A graphical tool to visualize predicted minimum delay flights. *Journal of Computational and Graphical Statistics*, 20, 06 2011.

[3] H.-K. Liu and T. Zhou. Empirical study of chinese city airline network. *Acta Phys. Sinica*, 56:106–112, 2007.

[4] H.-K. Liu and T. Zhou. Review on the studies of airline networks. *Prog. Nat. Sci.*, 18:601, 2008.

[5] Get an api key - google maps javascript api. `https://developers.google.com/maps/documentation/javascript/get-api-key`.

[6] About github pages. `https://docs.github.com/en/github/working-with-github-pages/about-github-pages`.

[7] Overview - google maps javascript api. `https://developers.google.com/maps/documentation/javascript/overview`.

[8] Bureau of transportation statistics. https://www.bts.gov/.

[9] Wikipedia - iata airport code. https://en.wikipedia.org/wiki/IATA_airport_code.

[10] Marker clustering - google maps javascript api. https://developers.google.com/maps/documentation/javascript/marker-clustering.

[11] Shapes - google maps javascript api. https://developers.google.com/maps/documentation/javascript/shapes#polylines.

[12] Free airport codes api data feed. https://www.air-port-codes.com.

[13] Maps platform styling wizard. https://mapstyle.withgoogle.com/.

[14] Ecmascript. https://en.wikipedia.org/wiki/ECMAScript.