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

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 domestic and international airline network of the 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 the United States.

Keywords: Airline Network, Data Visualization, Large Scale Data

Acknowledgements

I would like to express my deepest gratitude to Professor Yuanzhu Chen, my master's project supervisor, for his patient guidance, enthusiastic encouragement and useful critiques of this project work. My sincere thanks go to the faculty and staff at Memorial University of Newfoundland, specially in the Department of Computer Science, for their excellent teaching and relentless support.

My grateful thanks are also extended to the Google Maps API team for building this convenient Maps JavaScript API and maintaining a high quality documentation. Special thanks should be given to the U.S. Bureau of Transportation Statistics for the collation of the data.

Finally, I wish to thank my parents for their support and encouragement throughout my master's study.

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 [2]. In the past decades, an increasing number of studies are conducted to explore the airline network and people's travel patterns [15] [16]. 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 domestic and international airline network of the United States (U.S.). The raw data is derived from the U.S. Bureau of Transportation Statistics [21], 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 [14].

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 [7].

By visualizing and analyzing the complex airline network on a map, researchers can reveal the passengers' travel pattern [20], improve the network's traffic efficiency by minimizing flight delays and optimizing flight capacity [6], and even make predictions on the cities' economic growth [18].

In addition, the same geographical data visualization technique may also be applied to other application domains, such as exploring an animal's migration pattern [19] or studying a pandemic's transmission model [4].

1.3 Proposed Solution

Figure 1.1 demonstrates an overview of the visualization process. First, the airport data is loaded and parsed by the JavaScript function and stored in a custom built data structure. Second, for each airport record, a Google Maps `marker` object is created by passing in the airport's information and other custom options. Last, the `marker`s are drawn on the map.

The edge list (flight routes) data has a similar process as the airport data, ex-

cept for an additional data pre-processing step, which is developed to improve the visualization performance.

Detailed implementation steps are discussed in Chapter 3 - Visualize Airports and Chapter 4 - Visualize Airline Network.

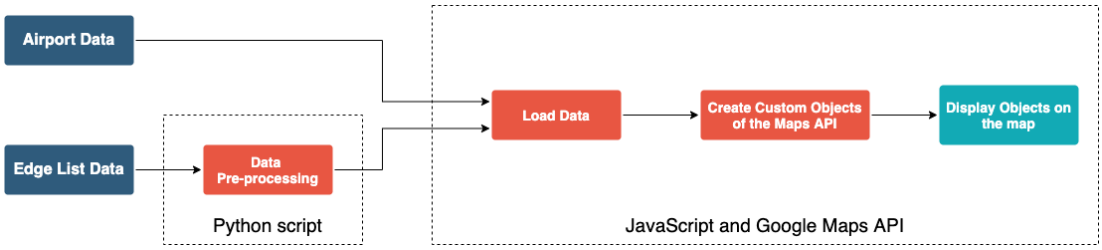

Figure 1.1: Visualization process

Chapter 2

Background

2.1 Google Maps API

The visualization of airline network requires a map layer to display the map along with other user defined elements. In this project, the Google Maps JavaScript API [14] is chosen as the base layer framework for map display.

The Google Maps JavaScript API is a JavaScript framework which allows developers to embed the Google Maps onto a third-party website, with user-defined elements and content displayed as overlays on the map. It provides an user friendly interface through which developers can interact with the map itself as well as elements on it easily through JavaScript methods.

The Google Maps API comes with a well-maintained documentation which contains a number of detailed explanations and comprehensive examples about the usage.

This high quality documentation makes the development and testing process fast and convenient. Besides, the Google Maps API comes with various third party libraries that provide additional functionalities and visual effects. These libraries are specifically beneficial to our visualization task. Moreover, the elements in the Google Maps API are highly customizable, making it as the perfect tool for the visualization of geographical data.

Some of the key elements in the Google Maps API are:

- `Map` - to represent the map element shown on the web page. Options such as the initial map center and zoom level can be specified when creating the map object. The appearance of the map can also be tailored during this process.
- `Marker` - to represent a point of interest (POI) on the map. By default the marker is displayed as a red pin icon. A `Marker` is used to indicate an airport in this project.
- `Polyline` - to represent one or more connected line segments on the map. The `Polyline` is defined by specifying the coordinates of the endpoints of each line segment to be drawn. The style of the line can be adjusted during the construction. In this project, a `Polyline` is used to demonstrate an airline route.
- Other shapes such as `Polygon`, `Rectangle` or `Circle` are also available through the Google Maps API though they are not used in this project.

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

A CSV file is a delimited text file that uses a comma to separate values [22]. The CSV file format is widely used in fields such as data science, business, banking, etc. to store tabular data (numbers and text) in plain text, in which case each line will have the same number of fields. In a CSV file, each row represents a data record, which is separated by the `newline` control character `\n`. Each data record consists of one or more fields which are separated separated by commas.

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

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 [24], 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)

```
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

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.

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.

```
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

Table 2.1: Number of records

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

2.3 Development and Testing platform

The development of this visualization project is on a local machine with a of 2.6 GHz 6-Core Intel Core i7 CPU and 16 GB 2400 MHz DDR4 memory. Git is used as the version control tool over the entire course of the development process.

To ensure a consistent visual experience across multiple platforms, the project is built upon pure vanilla JavaScript codes in ES6 [23] 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).

2.4 Deployment

In order to provide ubiquitous access to the visualization result, the application is deployed on the GitHub server through the `GitHub Pages` [8]. The `GitHub Pages` is a free static site hosting service that can publish the website directly from a GitHub repository. The visualization result can be viewed online at `https://dyckia.github.io/air-travel-patterns-visualization/`.

Chapter 3

Visualize 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 files 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` [13]. 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.

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 [14]. In other words, if

```html
<!DOCTYPE html>
<html>

<head>
  <title>Air Travel Patterns Visualization</title>
  <script src="https://polyfill.io/v3/polyfill.min.js?
features=default"></script>
  <script src="https://maps.googleapis.com/maps/api/js?
key=APIKEY&callback=initMap"></script>
  <link rel="stylesheet" type="text/css" href="./style.css"/>
  <script src="./app.js"></script>
</head>

<body>
  <div id="map"></div>
</body>

</html>
```

Figure 3.1: HTML file

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 explicitly set to 100%.

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`

```css
/* 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

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.

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

Figure 3.3: Main JavaScript file

As the goal of the project is to visualize airline network of the U.S., the center and

zoom level are empirical set to `center:{lat: 38.024009, lng: -97.081117}` and `zoom:4` respectively.

Figure 3.4: Initial map center

3.2 Read Airport Data

As stated in Section 2.2, the airport data is attained from the the U.S. Bureau of Transportation Statistics [21] 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.

The airport data file is stored on the server located at `root/data/`. As shown in Figure 3.5, a custom middleware function is created to read the content of a given CSV file.

In this middleware function, an Asynchronous JavaScript and XML (`Ajax`) call is sent to the server to get the target CSV file. `Ajax` is a set of web development techniques using the `XMLHttpRequest` object on the client side to communicate with the server [17]. It allows a web application to send and retrieve data from a server in various formats, including JSON, Extensible Markup Language (XML), HTML, and text files. Due to its "asynchronous" nature, it decouples the data exchange layer from the representation layer. In other words, actions like data exchange and element update can be made without refreshing the page.

Once the CSV file is retrieved by the `Ajax` call, the middleware verifies the status state to ensure that only valid data is returned.

```
1  // read given csv file via an AJAX call
2  function readCSVFile(file) {
3      const rawFile = new XMLHttpRequest();
4      rawFile.open("GET", file, false);
5      let content;
6      rawFile.onreadystatechange = function () {
7          if (rawFile.readyState === 4) {
8              if (rawFile.status === 200 || rawFile.status === 0) {
9                  content = rawFile.responseText;
10             }
11         }
12     }
13     rawFile.send();
14     return content;
15 }
```

Figure 3.5: readCSVFile function

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 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]`.

```
# return an array of airport 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

3.3 Visualize Airports on the Map

3.3.1 Visualize Domestic Airports

With the airport 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 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 airport 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 started to become unresponsive, which are detrimental to the user experience. In

```
// 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

addition, the markers became too crowded 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 comes with a third party 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.[11] 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.

To integrate this marker cluster feature in this project, first the `MarkerClustererPlus` library needs to be loaded via a `<script>` tag in the HTML file. Then a `MarkerClusterer` is instantiated by passing the following arguments:

```
// in HTML file
<script
src="https://unpkg.com/@google/markerclustererplus@4.0.1/dist
/markerclustererplus.min.js"></script>


// in JavaScript function
var markerCluster = new MarkerClusterer(
    map,
    markers,
    { imagePath:
'https://developers.google.com/maps/documentation/javascript/
examples/markerclusterer/m' }
);
```

Figure 3.8: MarkerClustererPlus library

- `map` - to identify the map on which the clusters are to be displayed.
- `markers` - to specify the markers to be grouped into clusters. The `markers` variable is an array that contains the airport markers created in Section 3.3.1. Markers not included in the `markers` variable will be always displayed. In our case, all the airport markers will be added to the `markers` variable.
- `imagePath` - to specify the location of the icons of the clusters to be displayed. The `imagePath` location should contain five image files named from `m1.png` to `m5.png`. These icons represent the volume of the clusters, with `m1.png` being the smallest in size and `m5.png` being the largest. By default, the `MarkerClustererPlus` library uses the color to indicate the size of the clusters.

As Figure 3.9 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.

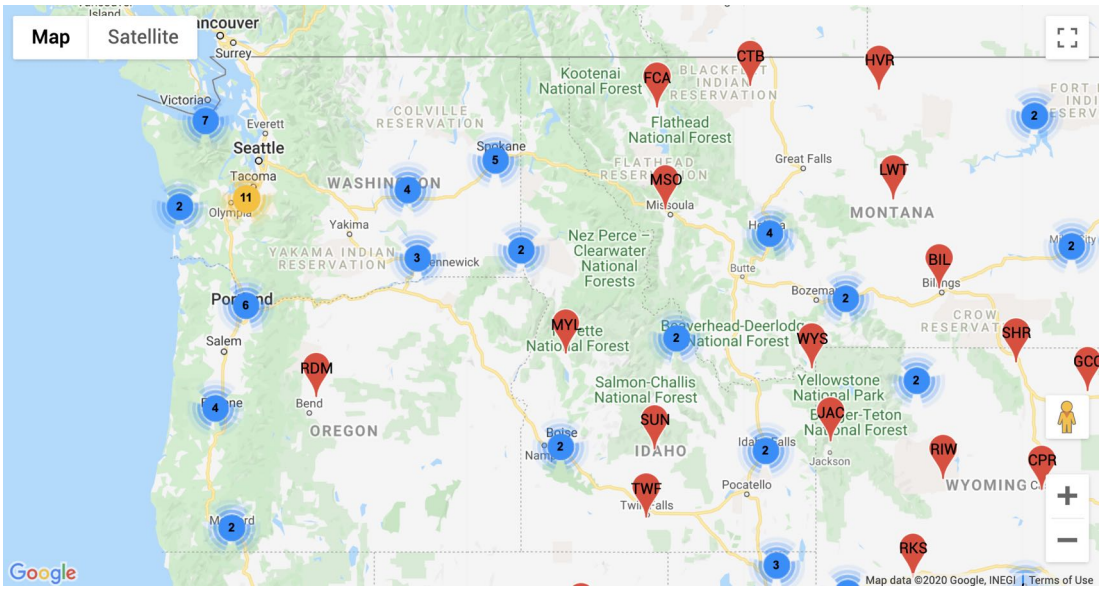

Figure 3.9: Markers grouped in clusters

By integrating the `MarkerClustererPlus` library in our project, the number of airport markers on the map has been significantly reduced, thus the visualization performance is substantially improved and the representation is significantly the simplified.

3.3.3 Visualize International Airports

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

3.4 Visualization Result for the Airport Data

Figure 3.10 and 3.11 demonstrate the visualization result for the U.S. domestic airport data and the international airport data respectively.

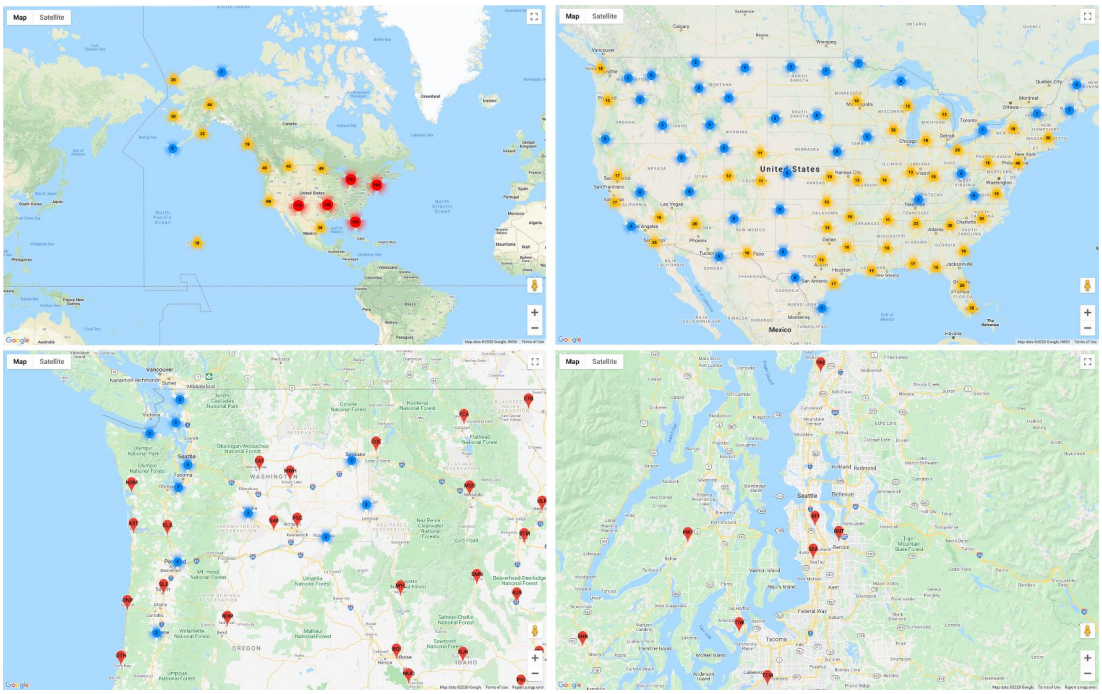

Figure 3.10: Result for domestic airports at various zoom levels

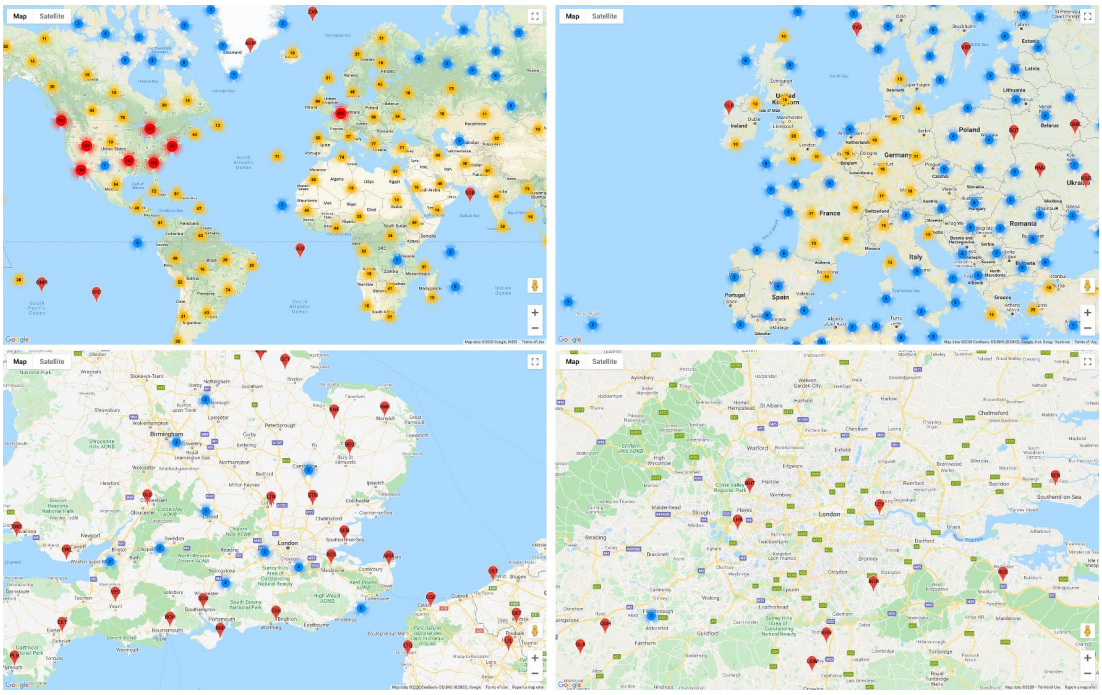

Figure 3.11: Result for international airports at various zoom levels

Chapter 4

Visualize 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).

With the modular architecture of our visualization pipeline, same `readCSVFile` function 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 one, airport 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 air traffic 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. The application can specify custom styles (such as the line's color, opacity, or weight) for those line segments by passing the `PolylineOptions` when constructing the `Polyline` object [12]. Figure 4.2 demonstrates the constructor arguments when creating a `Polyline` instance:

- `path` - to define the end points for each segment. A polyline object can contain multiple connected line segments.
- `strokeColor` - to specify the color of the line. The value is specified in hexadecimal format.
- `strokeOpacity` - to indicate the opacity of the line. The range of the opacity value is from 0.0 to 1.0 with 0 being completely transparent and 1 being completely solid.
- `strokeWeight` - to define the width of the line. The value specifies the width in pixels.

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 the IATA airport codes, not in the format of geolocation coordinates. In other words, the coordinates must be retrieved through the `airports` array by referencing the airport codes.

```
// define polyline's endPoints
endPoints = [
    { lat: 37.772, lng: -122.214 },
    { lat: 21.291, lng: -157.821 }
];

// a new polyline is created by instantiating a google.maps.Polyline object
const line = new google.maps.Polyline({
    path: endPoints,
    geodesic: true,
    strokeColor: '#78c2b7', // color in hexadecimal value
    strokeOpacity: 0.8, // opacity ranges from 0 to 1.0
    strokeWeight: 1 // weight of poly line in pixel
});

line.setMap(mapCreated);
```

Figure 4.2: Polyline options

As the pseudo code states in Figure 3.6, the airport records are stored in 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` variable's data structure is proposed to improve the efficiency in retrieving the airport 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 Airport 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 airport 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 airport data. By examining the missing airport 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 [1] or Wikipedia [24].

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 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.

```
# 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

4.3 Visualize Passengers Count

With the airline routes being efficiently displayed on the map, the next step is to visualize the traffic amount on the map. Based on the common data visualization practice and the style options offered by the Google Maps JavaScript API (Section 4.2.1), three available techniques are implemented and compared to interpret the passengers count:

1. Interpret the passengers count as the width of the `Polyline`. The width of the `Polyline` is set to be proportional to the number of passengers. A wider line represents an airline route with higher traffic volume.

   As shown in 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. Interpret the passengers count as the color of the `Polyline`. The rational behind this visualization technique is similar to the 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 blue side.

   Figure 4.5 demonstrates the visualization result in this approach. As the image shows, routes with different colors are not highly distinguishable from each other.

3. Interpret the passengers count as the opacity of the `Polyline`. Similar to the

width technique, 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 Figure 4.6, the result is both concise and visually perceivable.

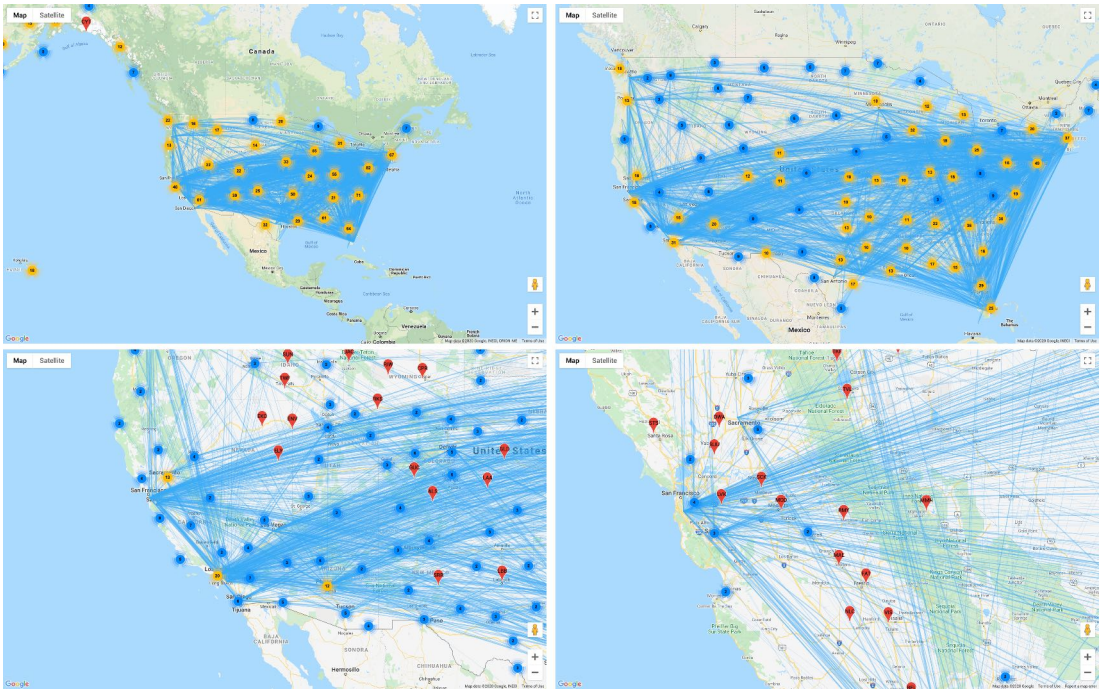

Figure 4.4: Result of width at various zoom levels

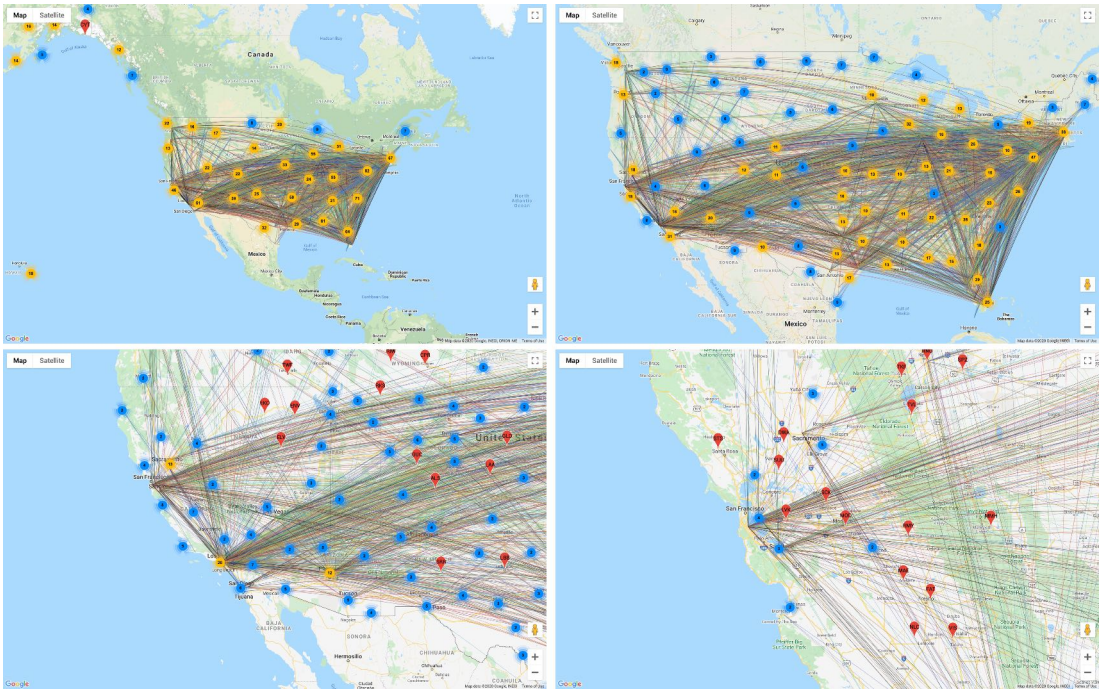

Figure 4.5: Result of color at various zoom levels

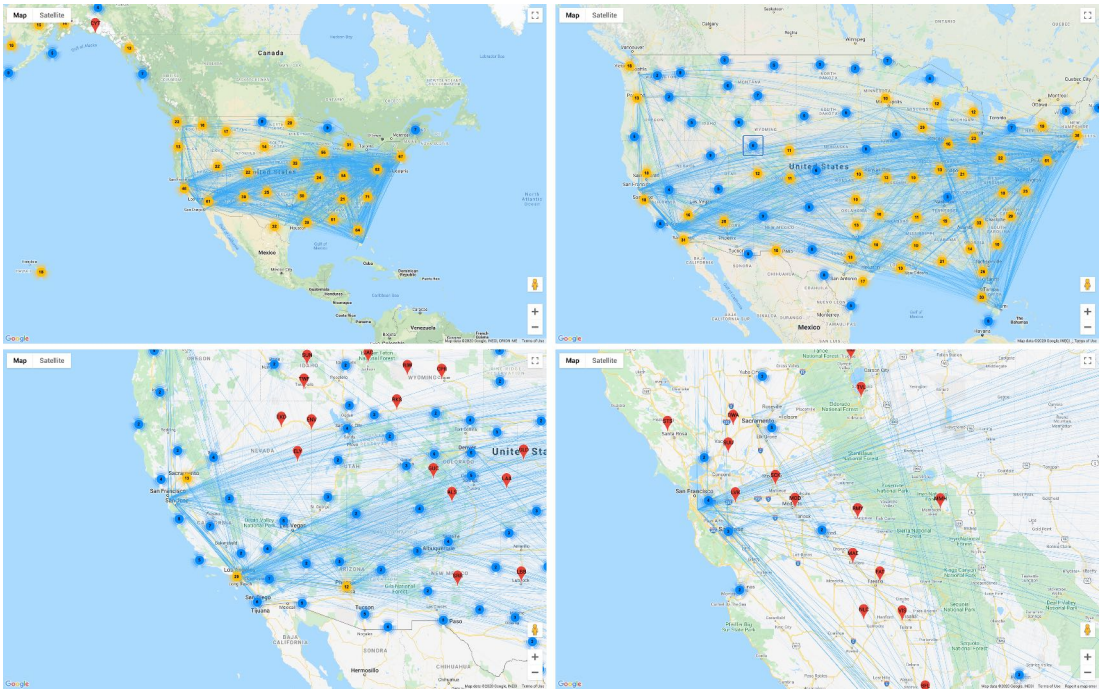

Figure 4.6: Result of opacity at various zoom levels

4.4 Resolve a Display Glitch with Polyline

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 4.7 demonstrates this display glitch.

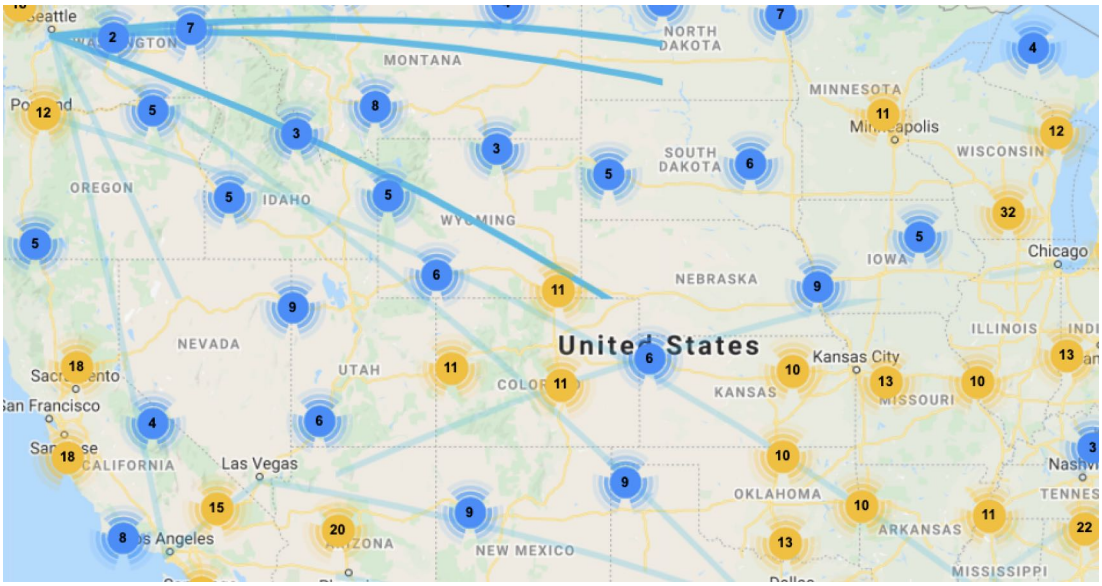

Figure 4.7: Display glitch

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

4.4.1 Customize Airport Marker Icon

Through experimentation 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%. With the size of marker icon substantially reduced, the display performance is improved.

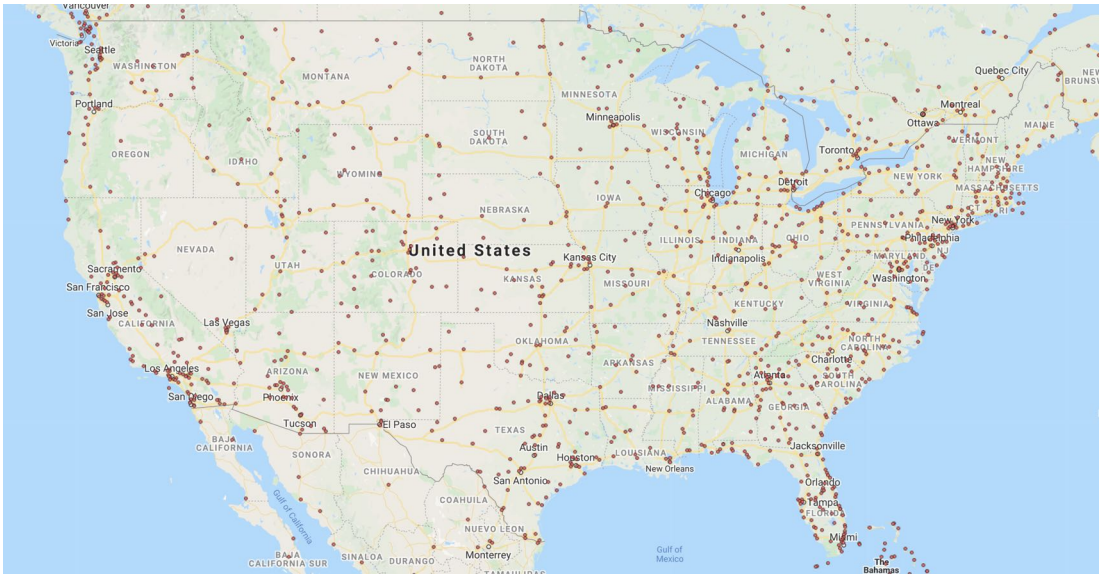

Figure 4.8: New airport marker icon

4.4.2 Add Dynamic Info Window

In addition, the airport labels also need to be removed to further improve the map's performance. Instead of displaying all the airport codes constantly on the map, the airport code should ideally only be displayed when the user needs it.

After researching online [5], I end up by creating a dynamic info window that displays the airport code when the user hovers the mouse cursor on the corresponding airport icon. This feature is implemented through the combination of HTML, CSS and JavaScript.

A new `<div>` element called `tooltip` is defined inside the HTML file created in Section 3.1.1, which will be used to display the airport code. An `id` attribute is also assigned to this element so that its style and action can be manipulated by the CSS file and JavaScript file respectively.

As Figure 4.9 indicates, various stylistic options are specified to the `tooltip` element in the CSS file. Firstly, in order to hide this info window by default, the `display` property is set to `none`. Secondly, since the info window should be displayed as an overlay with close proximity to the airport icon, I set the `position` to be `absolute` and `z-index` to be 1. With the `position` to be `absolute` the `tooltip` element can be placed at any specific position in the browser window. This position value will be provided by the JavaScript function in the next step. Finally, some typography options are added to stylize the appearance of the info window.

As the goal is to display the info window when the cursor moves on an icon and hide when the cursor leaves, we need to find a way to constantly monitors the mouse's

```
#tooltip {
    display: none;
    position: absolute;
    z-index: 1;
    background: #494949;
    color: #FFF;
    font-family: sans-serif;
    font-size: 12px;
    padding: 6px;
    opacity: 80%;
    pointer-events: none;
}
```

Figure 4.9: Custom style sheet for info window

interactions with the map. Similar to the `EventListener` interface in JavaScript, the Google Maps API provides a number of named events to handle the state changes of the objects on the map [9]. These events are designed to respond to user's certain interactions. For example, `click` is for mouse single click, `mouseover` is for mouse moves in, `mouseout` is for mouse moves out, etc. Applications interested in a certain event can register the event handler by calling the `addListener()` method and passing in a callback function. The callback function will be executed when that certain event is caught.

In our project, we register two events for each airport marker that is created, namely the `'mousemove'` event and the `'mouseout'` event. As Figure 4.10 shows,

when the user moves the mouse onto an airport icon, the `showTooltip` callback function is called to display the its corresponding airport code. When the user moves the cursor out of the airport icon, the `hideTooltip` function is executed to hide the info window.

```
1  for (const property in airports) {
2      const marker = new google.maps.Marker({
3          position: airports[property].pos,
4          icon: airportIcon,
5          map: map
6      });
7      // add eventListeners to each marker created
8      marker.addListener('mousemove', showTooltip.bind(null,
   event, airports[property].code));
9      marker.addListener('mouseout', hideTooltip);
10 }
```

Figure 4.10: Event listeners for airport marker

From a user experience perspective, the code of the airport should be displayed right next to the airport icon the mouse points to. Because of this, the mouse's position must be retrieved when an event is triggered. For a standard JavaScript event, the mouse's position can be obtained by accessing the `event.clientX` and `event.clientY` properties. However, an error is returned when I try to access the same properties of the event. After some test and research, it is found that the `event` object returned by the Google Maps API is not a standard Document Object Model (DOM) event, it is instead a custom `event` object of the Google Maps API [9] which does not contain the mouse's position.

One common approach to get the pixel position of a marker is to add an `overlay`

on the map, and compute the projection from the geolocation of a point to the pixel coordinates of that container by using the `projection.fromLatLngToDivPixel()` method. However, as this projection computation is relatively costly, it can only be used in less frequent circumstances. Considering the large number of objects in our case, a more efficient method needs to be invented to locate the position of the mouse in the browser window.

```
<body>
  <div id="map"></div>
  <div id="tooltip">Tooltip</div>
  <script>
    // global variable to store mouse position
    const MOUSEPOS = { "x": 0, "y": 0 }

    // function to update the mouse's position
      function updateMousePos(event) {
        MOUSEPOS.x = event.clientX;
        MOUSEPOS.y = event.clientY;
      }

    // create eventListener to track mouse's movement
    let mapWindow = document.getElementById("map");
    mapWindow.addEventListener("mousemove", updateMousePos);
  </script>
</body>
```

Figure 4.11: Global variable for tracking the mouse's position

To solve this issue, a standard `DOM eventListener` is created and attached to the marker's parent element, which is the `div map` element. As Figure 4.11 indicates, this `eventListener` is triggered whenever the mouse is moved on the map. Since this is

a standard `DOM eventListener`, the mouse's position properties can be accessed via the `event` object. In order to make this mouse's position values persistent, I create a global variable to store the coordinates. When the mouse is moving on the map, the callback function `updateMousePos` keeps tracking the mouse's position by constantly updating the values of the global variable.

The `showTooltip` function accepts the airport code as the argument. It replaces the content of the `tootltip` element with the airport code. Then it explicitly sets the position of the `tootltip` to be next to the position of the mouse by accessing the global variable created in the earlier step. To prevent the mouse from blocking the `tootltip`'s content, the position is offset by 5 pixels right and 5 pixels down. In the last step, the `tootltip` is displayed by changing the value of the `display` property to `block`.

```
1  function showTooltip(event, code) {
2      const tooltip = document.getElementById('tooltip');
3      tooltip.innerHTML = code;
4      tooltip.style.display = 'block';
5      tooltip.style.left = MOUSEPOS.x + 5 + 'px';
6      tooltip.style.top = MOUSEPOS.y + 5 + 'px';
7  }
8
9  function hideTooltip(event) {
10     const tooltip = document.getElementById('tooltip');
11     tooltip.style.display = 'none';
12 }
```

Figure 4.12: Callback functions

Similarly, when the mouse leaves an airport icon, the `hideTooltip` function hides the `tootltip` element by changing the value of the `display` property back to `none`.

As the final result shown in Figure 4.13, when the user hovers the mouse over an airport icon, an info window with the corresponding airport code will appear by the cursor. Once the user moves the mouse away from the airport icon, the info window will disappear automatically.

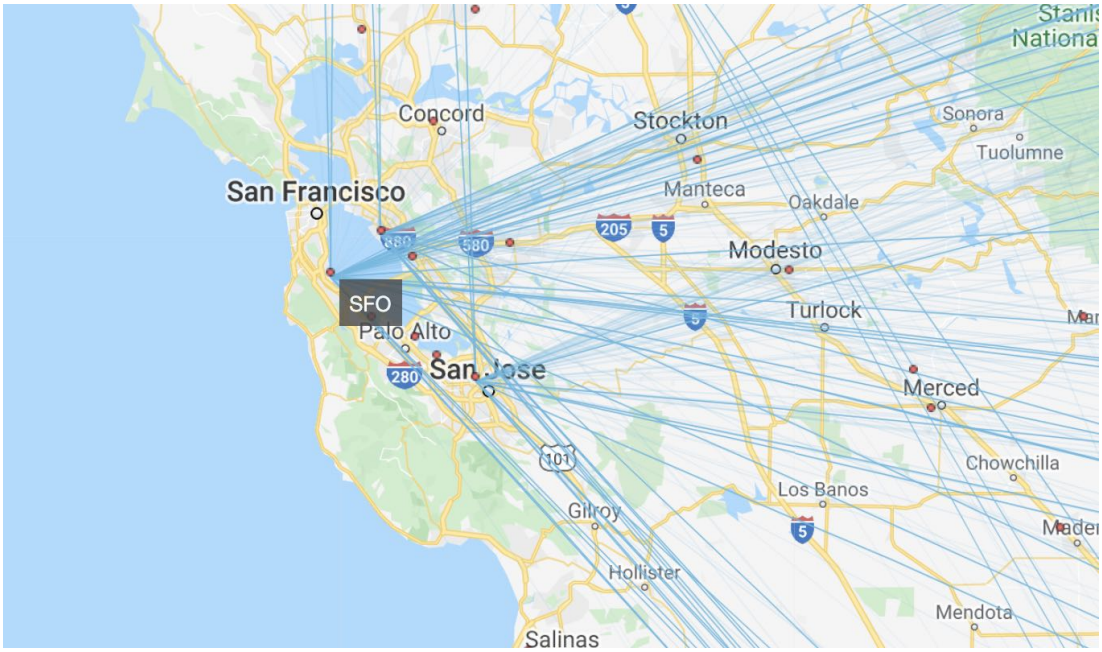

Figure 4.13: Dynamic info window

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.

4.5 Enhancements on User Experience

4.5.1 Prioritize Information Display

Google Maps by default displays a wide variety of information on the map, such as terrains, landmarks, roads, POIs, 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 4.14, 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 [10] which allows the user to interactively generate a JSON styling object in the web browser.

```
// 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 4.14: An example of style options

As the primary purpose of this visualization project is to display the 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 4.15 demonstrates the final visual style of the map. The appearance after customization is much cleaner and more recognizable comparing to the original theme.

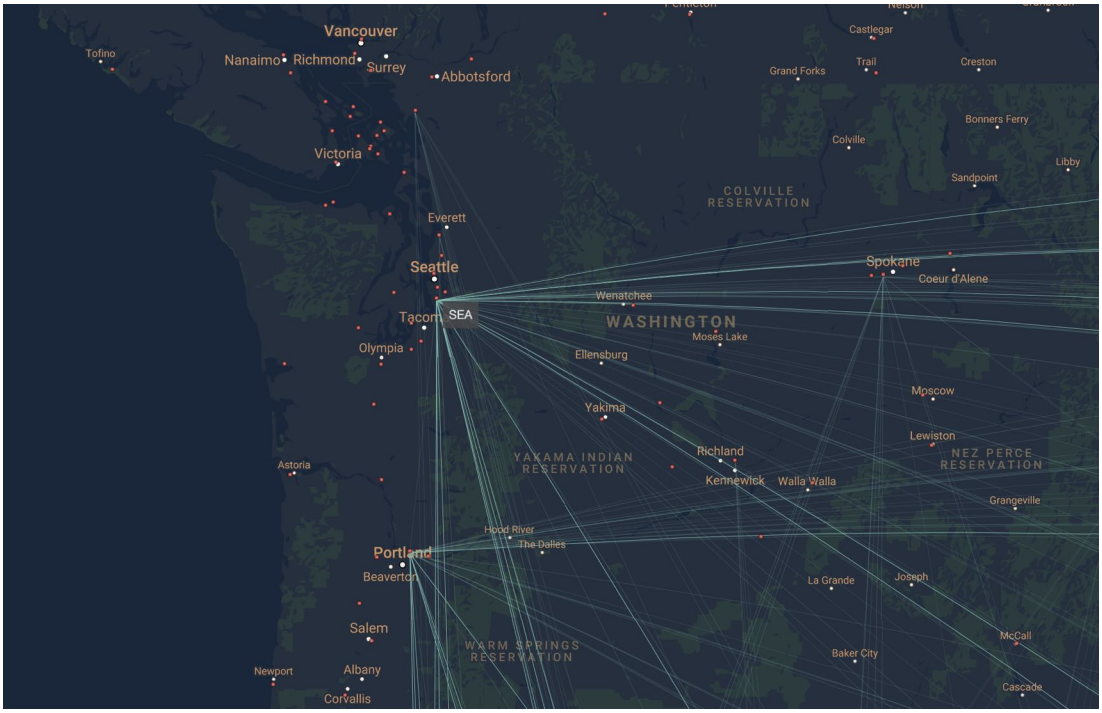

Figure 4.15: Map style after customization

4.5.2 Create a Landing Page

A landing page is created to provide the users with a single and elegant gateway to access all the visualization result in this project.

The landing page is built by HTML and `Bootstrap` [3]. `Bootstrap` is a popular open-source CSS framework that provides component-based design templates for typography, buttons, navigation, and other interface elements. `Bootstrap` provides a convenient way to keep the user interface consistent and elegant which is beneficial to fast development and prototyping.

Figure 4.16: Landing page

4.6 Visualization Result for the Edge List Data

Figure 4.17 and 4.18 illustrate the visualization result for the domestic airline network and the international airline network respectively.

Figure 4.17: Result for domestic airline network at various zoom levels

Figure 4.18: Result for international airline at various zoom levels

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 domestic and international airline network of the U.S. In order to achieve high performance on the large scale data set, a variety of optimization techniques are utilized to improve the efficiency, such as data pre-processing, caching, data structure optimization, dynamic content display, etc.

In addition, various visualization strategies are implemented and compared to interpret the routes' traffic amount, namely shape's size, color and opacity. Moreover, the application's User Interface (UI) is customized and fine-tuned to further enhance the information's density and conciseness.

The final application provides an efficient, concise and interactive environment to

visualize the complex airline network of the 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 exploring an animal's migration pattern or studying a pandemic's transmission model.

5.2 Future Work

Besides the features implemented in Section 4.5, some additional functionalities could be added to further enhance the application's user experience.

As illustrated in Section 4.4.2, an airport code can be revealed dynamically when the user hovers the cursor upon an airport icon. While the IATA code is short and precise for people who are familiar with it, its meaning is obscure to normal users. We can utilize the same info window created in Section 4.4.2 to display additional information about the airport, such an airport's full name and its location. Such information can be either gathered and stored in advance or fetched on the fly via a airport database API such as `air-port-codes` [1].

In addition, the same dynamic content window can also be used to display the information about an airline route, such as the names of the departure and landing airports, and the traffic amount. Since the mouse's position is already publicly available via the global variable created in Section 4.4.2, this feature can be implemented simply by adding certain `eventListeners` during the creating of the `Polylines`.

Moreover, for users who want to further investigate a specific airport or a certain type of routes. A filter feature can be developed which displays or hides the routes based on the criteria defined by the users. By interacting with these filter options, a user can display routes that are only connected to a particular airport or within a certain range of passengers amount, hiding uninterested information on the map.

Bibliography

[1] AIR-PORT-CODES. Free Airport Codes API Data Feed. `https://www.air-port-codes.com`, 2020 (accessed July, 2020).

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

[3] Bootstrap Team. Bootstrap - Getting started. `https://getbootstrap.com/docs/4.5/getting-started/introduction`, 2020 (accessed August, 2020).

[4] D. Brockmann and D. Helbing. The hidden geometry of complex, network-driven contagion phenomena. *science*, 342(6164):1337–1342, 2013.

[5] X.-J. Chen, G. Macrì, and S. He. WebGL2 powered geospatial visualization layers. `https://github.com/visgl/deck.gl`, 2020 (accessed July, 2020).

[6] M. T. Crotty. Visualizing more than twenty years of flight data for the raleigh-durham international airport. *Case Studies In Business, Industry And Government Statistics*, 5(1):51–57, 2014.

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

[8] GitHub Docs. About GitHub Pages. `https://docs.github.com/en/github/working-with-github-pages/about-github-pages`, 2020 (accessed July, 2020).

[9] Google Maps Platform. Events - Maps JavaScript API. `https://developers.google.com/maps/documentation/javascript/events`, 2020 (accessed July, 2020).

[10] Google Maps Platform. Styling Wizard - Maps JavaScript API. `https://mapstyle.withgoogle.com/`, 2020 (accessed July, 2020).

[11] Google Maps Platform. Marker Clustering - Maps JavaScript API. `https://developers.google.com/maps/documentation/javascript/marker-clustering`, 2020 (accessed June, 2020).

[12] Google Maps Platform. Polyline - Maps JavaScript API. `https://developers.google.com/maps/documentation/javascript/shapes#polylines`, 2020 (accessed June, 2020).

[13] Google Maps Platform. Get an API Key - Maps JavaScript API. https://developers.google.com/maps/documentation/javascript/get-api-key, 2020 (accessed May, 2020).

[14] Google Maps Platform. Overview - Maps JavaScript API. https://developers.google.com/maps/documentation/javascript/overview, 2020 (accessed May, 2020).

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

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

[17] MDN web docs. Ajax - Developer Guides. https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX, 2020 (accessed June, 2020).

[18] C. Roucolle, T. Seregina, and M. Urdanoz. Measuring the development of airline networks: Comprehensive indicators. *Transportation Research Part A: Policy and Practice*, 133:303 – 324, 2020.

[19] J. Shamoun-Baranes, A. Farnsworth, B. Aelterman, J. A. Alves, K. Azijn, G. Bernstein, S. Branco, P. Desmet, A. M. Dokter, K. Horton, et al. Innovative visualizations shed light on avian nocturnal migration. *PloS one*, 11(8):e0160106, 2016.

[20] J. Terje Bjørke, S. Nilsen, and M. Varga. Visualization of network structure by the application of hypernodes. *International Journal of Approximate Reasoning*, 51(3):275 – 293, 2010.

[21] The U.S. Department of Transportation. The Bureau of Transportation Statistics. `https://www.bts.gov`, 2020 (accessed July, 2020).

[22] Wikipedia. Comma-separated values. `https://en.wikipedia.org/wiki/Comma-separated_values`, 2020 (accessed May, 2020).

[23] Wikipedia. ECMAScript. `https://en.wikipedia.org/wiki/ECMAScript`, 2020 (accessed May, 2020).

[24] Wikipedia. IATA airport code. `https://en.wikipedia.org/wiki/IATA_airport_code`, 2020 (accessed May, 2020).