UNIVERSITY OF MALAYA

WIA2005 ALGORITHM DESIGN AND ANALYSIS
GROUP PROJECT REPORT

GROUP 6

SESSION 2019/20

LECTURER: DR HAZRINA BT SOFIAN

NAME	MATRIC NO.
JEREMIAH DUDUN HENRY	17193186/1
MOHAMMAD IZWAN BIN ABDUL SALIM BIN ABDUL SALIM	17066527/1
IFFAT AHMED AFIA	17199714/1
ARIQ BIN RAZALI	17143473/1
NURUL AIMAN FARIHAH BT MUHD SHUKRI	17164719/2

Table of Contents

Introduction

For this course's group assignment project, we decided to choose **question 2.** Question 2 concerns a UK broker looking for industrial investment opportunities in the cities of Asia. He already invested in a company in Kuala Lumpur and now he plans to travel to several cities in Asia from Kuala Lumpur to expand his investment. The cities include Jakarta, Bangkok, Taipei, Hong Kong, Tokyo, Beijing and Seoul. Ben decided to focus more on the possibilities of better return of investment in cities which have a positive economy and financial growth. So, Ben needs to do some analysis of the local economy and finance situation for the last 3 months. Furthermore, he needs to optimise his travel. He will give priority to cities with possible better investment return based on the analysis of local economic and financial situations. If the next nearest city to be visited has a less better economic and financial situation than any of the other cities, Ben will visit another city first provided that the difference of distance between the 2 cities is not more than 40% and the difference of sentiment analysis between the 2 cities is not less than 2%.

Throughout this project, several algorithms were used; with some being implemented or adapted to fit different problem cases. Among them were a couple algorithms not learnt in this course. For certain sub-problems, the various problem-solving techniques and algorithm implementations taught throughout this WIA2005 Algorithm Design and Analysis were adapted and modified to meet the requirements of those problems. In addition to that, various external libraries and APIs were used in addition to the native Python libraries to help solve the various problems in this project. Workloads were delegated equally among the 5 group members with

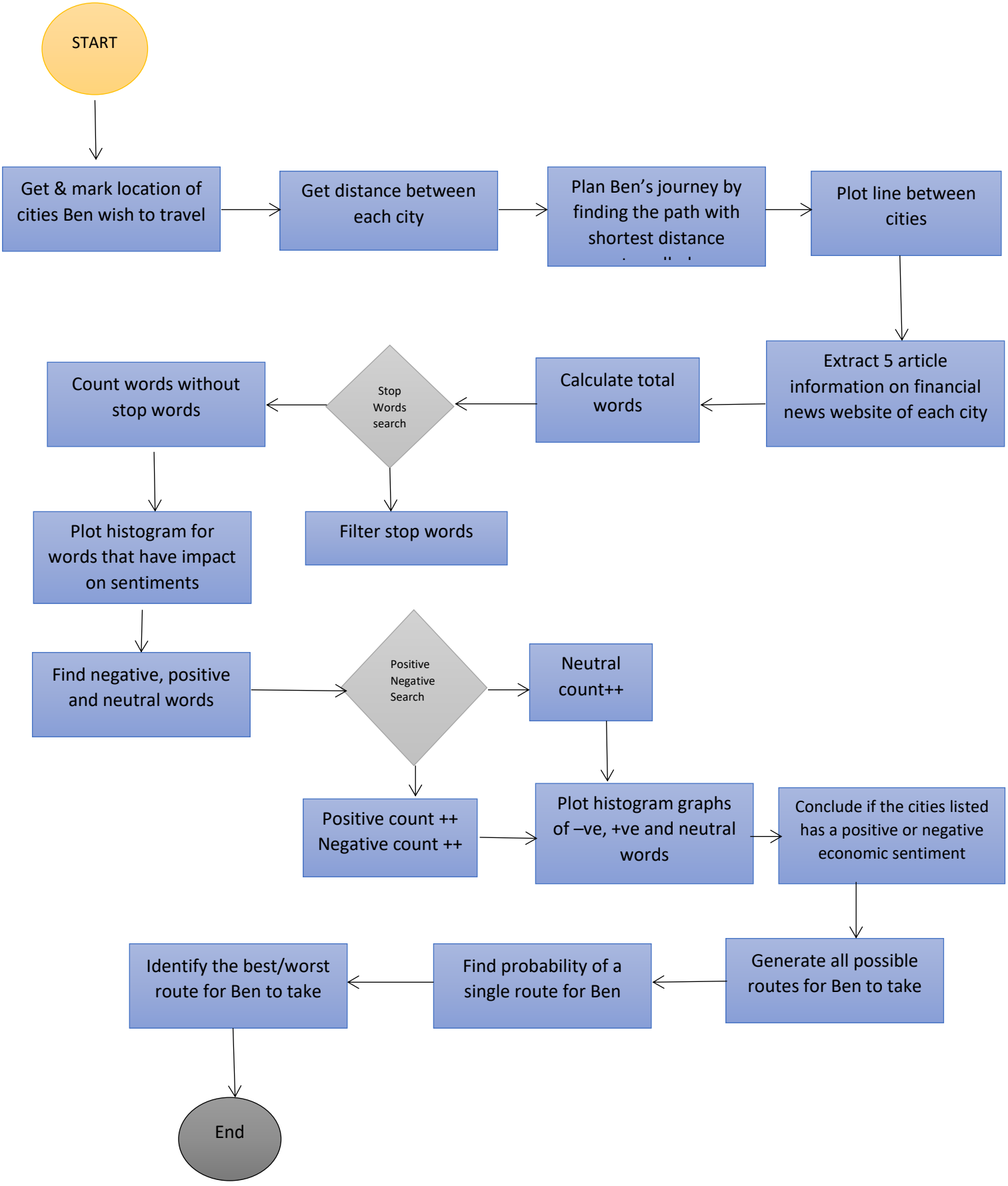
Project Description

Question two was split into 3 problems. For the first problem, the group interpreted it as the minimum Hamiltonian path problem instead of a Travelling Salesman Problem (TSP) variant. This was based by making a few assumptions- the first being that “Ben Sherman” is from the UK and has only invested in a company in Kuala Lumpur. The question states that Ben must start is journey from Kuala Lumpur but that does not mean that he has to return to Kuala Lumpur; hence why the question was not interpreted as a TSP problem/ shortest Hamiltonian cycle problem. Given that this is a Nondeterministic Polynomial complete problem, there are no polynomial-time algorithms to solve it so various algorithms and approaches were implemented to solve Problem 1.

Problem 2 simply required the extraction of words from a text-file to determine the financial situation in the respective countries Ben plans to visit. This is done by calculating the sentiment ratio of positive words to negative words from 5 articles per city. The rest of problem 2 is straightforward and does not hold much bearing on the subsequent problem. A string-matching algorithm was needed to count the number of negative, neutral, and positive words in each article before plotting the words on a histogram.

Problem 3 was a bit more confusing as we were unsure on how to interpret the probability distribution part as all the routes possible are finite and not random. Because of this, we decided to spread out all the possible routes and rank them according to how closely the routes matches with the most optimal route after calculating it based on the difference of distance between two cities and the sentiment analysis between to cities between each layover. With the number of matches spread across a frequency distribution table, the routes were then ranked from the least recommended to the most recommended.

Control Flow Chart



Problem 1

For problem 1, a few logical assumptions were made. Ben has to travel through 7 cities starting from Kuala Lumpur but does not return. This is assumed as the question states that Ben is a UK broker and that he had only invested in a company in Kuala Lumpur so it does not mean that he will return to Kuala Lumpur as he could very well be planning to head to other cities or back to the UK. Because of this very reason, this problem is interpreted to be a *Minimum Shortest Hamiltonian Path* problem and not a *Travelling Salesman Problem (TSP)* variant. It is also assumed that all cities have international airports with connecting flight to every other city since the cities in question are the capital cities of 8 countries.

Problem 1.1 Get and Mark Locations of All the Cities Ben Plans to Visit

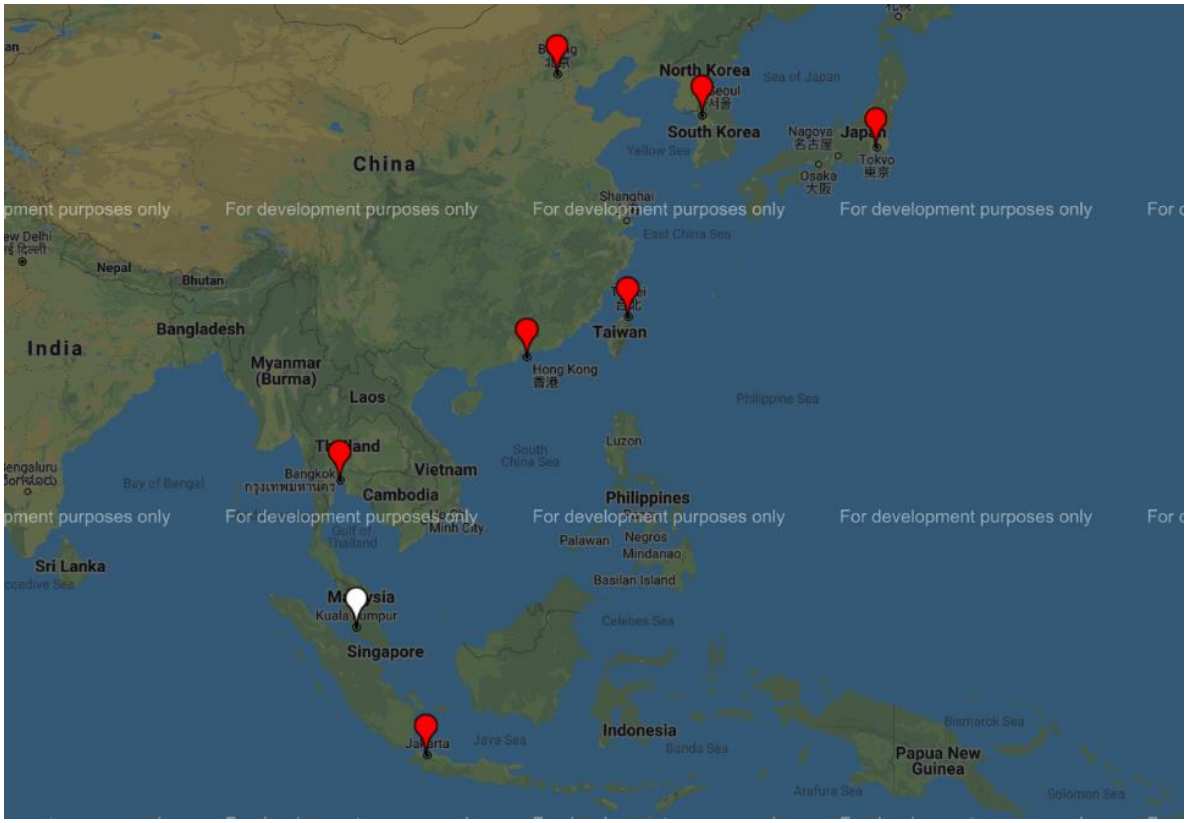
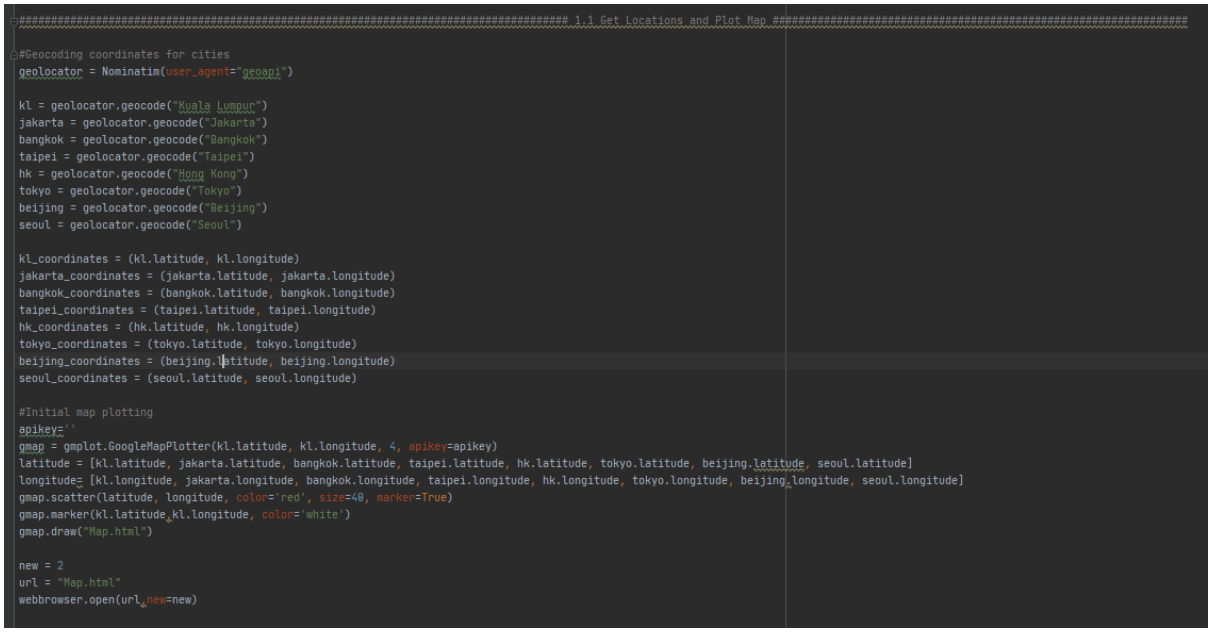


Figure 1.1 – Cities plotted on Google Maps



```
############################################################################## 1.1 Get Locations and Plot Map ##############################################################################

#Geocoding coordinates for cities
geolocator = Nominatim(user_agent="geoapi")

kl = geolocator.geocode("Kuala Lumpur")
jakarta = geolocator.geocode("Jakarta")
bangkok = geolocator.geocode("Bangkok")
taipei = geolocator.geocode("Taipei")
hk = geolocator.geocode("Hong Kong")
tokyo = geolocator.geocode("Tokyo")
beijing = geolocator.geocode("Beijing")
seoul = geolocator.geocode("Seoul")

kl_coordinates = (kl.latitude, kl.longitude)
jakarta_coordinates = (jakarta.latitude, jakarta.longitude)
bangkok_coordinates = (bangkok.latitude, bangkok.longitude)
taipei_coordinates = (taipei.latitude, taipei.longitude)
hk_coordinates = (hk.latitude, hk.longitude)
tokyo_coordinates = (tokyo.latitude, tokyo.longitude)
beijing_coordinates = (beijing.latitude, beijing.longitude)
seoul_coordinates = (seoul.latitude, seoul.longitude)

#Initial map plotting
apikey=''
gmap = gmplot.GoogleMapPlotter(kl.latitude, kl.longitude, 4, apikey=apikey)
latitude = [kl.latitude, jakarta.latitude, bangkok.latitude, taipei.latitude, hk.latitude, tokyo.latitude, beijing.latitude, seoul.latitude]
longitude= [kl.longitude, jakarta.longitude, bangkok.longitude, taipei.longitude, hk.longitude, tokyo.longitude, beijing.longitude, seoul.longitude]
gmap.scatter(latitude, longitude, color='red', size=40, marker=True)
gmap.marker(kl.latitude,kl.longitude, color='white')
gmap.draw("Map.html")

new = 2
url = "Map.html"
webbrowser.open(url,new=new)
```

Figure 1.2 – Geocoding code segment

1) The geocode of the 8 cities (Kuala Lumpur, Jakarta, Bangkok, Taipei, Hong Kong, Tokyo, Beijing, and Seoul were obtained by using the *Nominatim* import from the *geopy.geocoders* Google geocoding API.
2) The longitude and latitude of all the cities geocodes were then assigned into an array and plotted on Google Maps using the *gmplot* API.

Problem 1.2 Get the distances between each of the cities

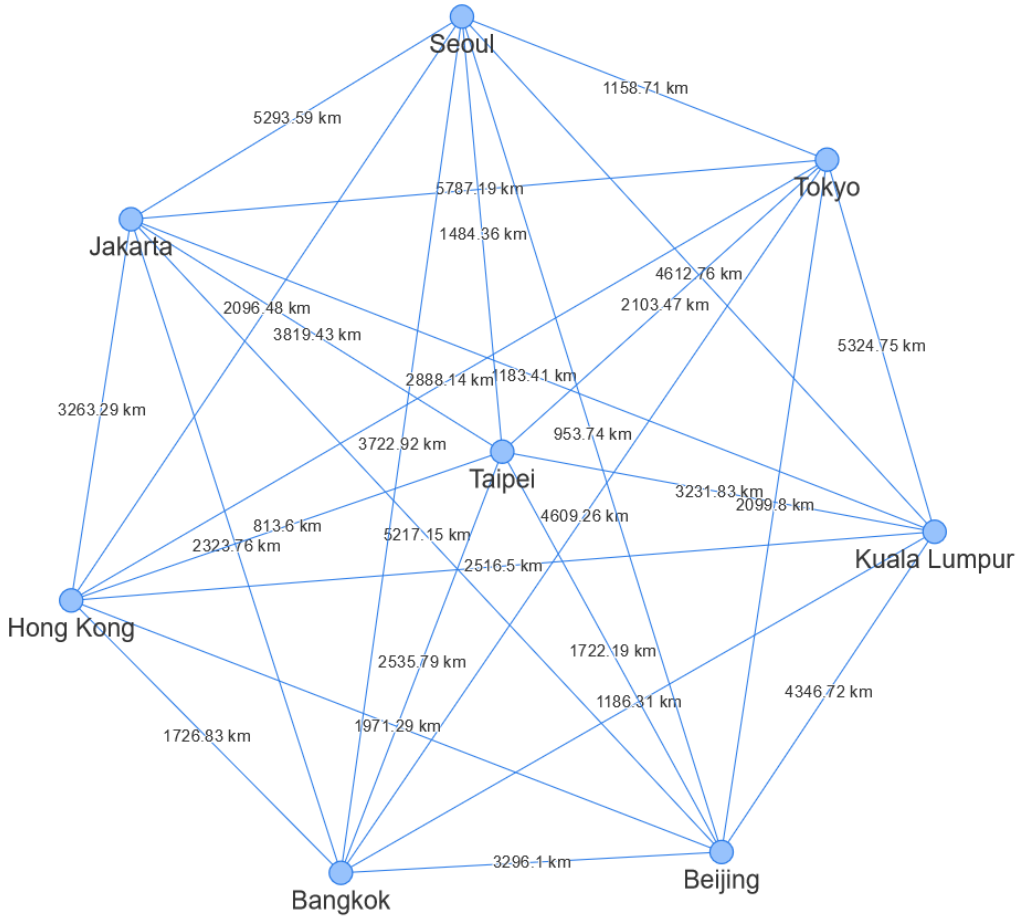
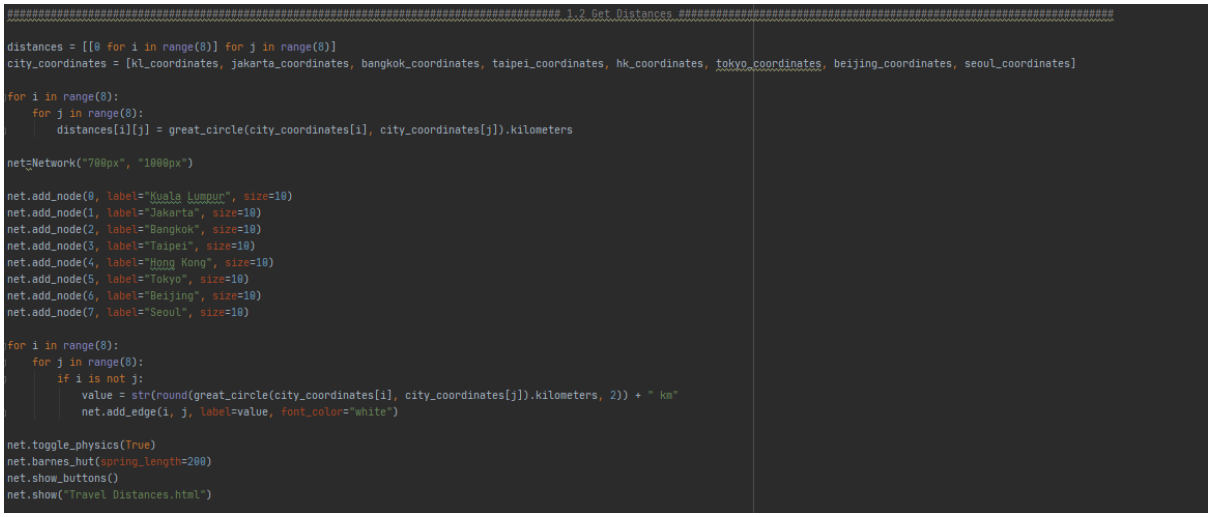


Figure 1.3 Network graph of city connections

1) The distances between all cities were visualized on a graph network. Since all cities are a connected, the respective weights and edges exist between any city to any other city.
2) The *Pyvis* library was used to visualize this graph network.



```
############################################################################## 1.2 Get Distances ##############################################################################

distances = [[0 for i in range(8)] for j in range(8)]
city_coordinates = [kl_coordinates, jakarta_coordinates, bangkok_coordinates, taipei_coordinates, hk_coordinates, tokyo_coordinates, beijing_coordinates, seoul_coordinates]

for i in range(8):
    for j in range(8):
        distances[i][j] = great_circle(city_coordinates[i], city_coordinates[j]).kilometers

net=Network("700px", "1000px")

net.add_node(0, label="Kuala Lumpur", size=10)
net.add_node(1, label="Jakarta", size=10)
net.add_node(2, label="Bangkok", size=10)
net.add_node(3, label="Taipei", size=10)
net.add_node(4, label="Hong Kong", size=10)
net.add_node(5, label="Tokyo", size=10)
net.add_node(6, label="Beijing", size=10)
net.add_node(7, label="Seoul", size=10)

for i in range(8):
    for j in range(8):
        if i is not j:
            value = str(round(great_circle(city_coordinates[i], city_coordinates[j]).kilometers, 2)) + " km"
            net.add_edge(i, j, label=value, font_color="white")

net.toggle_physics(True)
net.barnes_hut(spring_length=200)
net.show_buttons()
net.show("Travel Distances.html")
```

Figure 1.3 Distance calculation between 2 cities

3) The distances between two cities were calculated using the *great_circle()* function with the coordinates of 2 cities as the parameters.
4) This calculation is put into a for-loop that iterates for n times the number of edges in the graph to map the distances of each city to every other city
5) The distances obtained were then used as the weights for the respective connections in the network graph of city connections.

```
#############################################################
#     Backtracking algorithm to find hamiltonian paths      #
#############################################################

def hamiltonian_path(g, v, visited, path, N, pathstring):
    if len(path) == N:
        # print hamiltonian path
        global hamiltonian_count
        hamiltonian_count+=1

        pathstring.append(path[:])#append to global array
        print(path)

        return

    # Check if every edge starting from vertex v leads to a solution or not
    for w in g.adjList[v]:

        # process only unvisited vertices as hamiltonian
        # path visits each vertex exactly once
        if not visited[w]:
            visited[w] = True
            path.append(w)

            # check next vertex (w) to see if path is a valid hamiltonian
            hamiltonian_path(g, w, visited, path, N, pathstring)

            # Backtracking function
            visited[w] = False
            path.pop()
```

Figure 1.4 Backtracking algorithm

6) A backtracking algorithm was used to find all possible Hamiltonian paths (paths in a graph where every node is visited once but does not return to the node of origin) in the mapped graph. There are (n-1)! paths so since there are 8 cities, there are 7! Paths.
7) The backtracking will traverse every unvisited edge to see if a solution to a Hamiltonian path exists. If the next node to be traverse still leads to the discovery of a possible Hamiltonian path, the backtracking function will call on itself recursively to append the node to the current path before trying the next node to check for a solution.
8) If a Hamiltonian path does not exist with the traversal to a node, the path will be popped from the array and the next edge in the adjacency list will be traversed.
9) This has a time complexity of O(n^n)

10) Usually this approach is used to find Hamiltonian cycles but the function is implemented in away that it does not traverse back to the beginning node but the last node in a path.

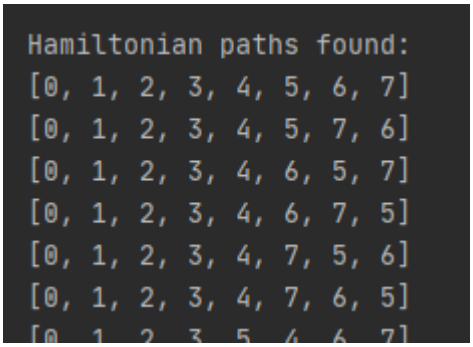


Figure 1.5 – Hamiltonian paths found after backtracking

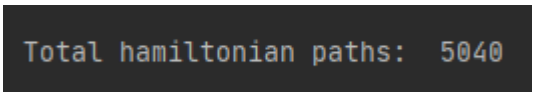


Figure 1.6- Number of Hamiltonian paths found

```
##########################################################
#bubble-calc algorithm to calculate distances of all paths #
##########################################################

def pathdistance(pathstring, city_coordinates, unsorted_distance_array):

    temp_distance = 0

    for i in range(len(pathstring)):
        current_path = pathstring[i]
        for j in range(7):
            temp_distance = round(great_circle(city_coordinates[current_path[j]], city_coordinates[current_path[j+1]]).kilometers + temp_distance, 2)

        unsorted_distance_array.append(temp_distance)
        temp_distance = 0

pathdistance(pathstring, city_coordinates, unsorted_distance_array)
```

Figure 1.7 – Bubble-calc algorithm

11) The *Bubble-calc algorithm* is a newly implemented algorithm which uses an iterative approach to calculate the total distance of each Hamiltonian path. Like bubble sort that operates on 'bubbles' of adjacent array indexes, this algorithm also utilizes the same sorting technique but instead of swapping, it calculates the distances between each city based on the corresponding node indexes (e.g. 0 = Kuala Lumpur, 1 = Jakarta, 2 = Bangkok). This will iterate for the number of Hamiltonian paths discovered. This has a time complexity of O (n^2)

Problem 1.3 Suggest a Journey for Ben to Visit Each of the Cities Once with the Least Distance Travelled

```
################################################
#      Timsort algorithm to sort path lengths  #
################################################
minrun = 32

#Implementation of insertion sort
def InsSort(arr, start, end):
    for i in range(start + 1, end + 1):
        elem = arr[i]
        j = i - 1
        while j >= start and elem < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = elem
    return arr

#Implementation of merge sort
def merge(arr, start, mid, end):
    if mid == end:
        return arr
    first = arr[start:mid + 1]
    last = arr[mid + 1:end + 1]
    len1 = mid - start + 1
    len2 = end - mid
    ind1 = 0
    ind2 = 0
    ind = start

    while ind1 < len1 and ind2 < len2:
        if first[ind1] < last[ind2]:
            arr[ind] = first[ind1]
            ind1 += 1
        else:
            arr[ind] = last[ind2]
            ind2 += 1
        ind += 1
```

Figure 1.8 – Timsort sorting algorithm

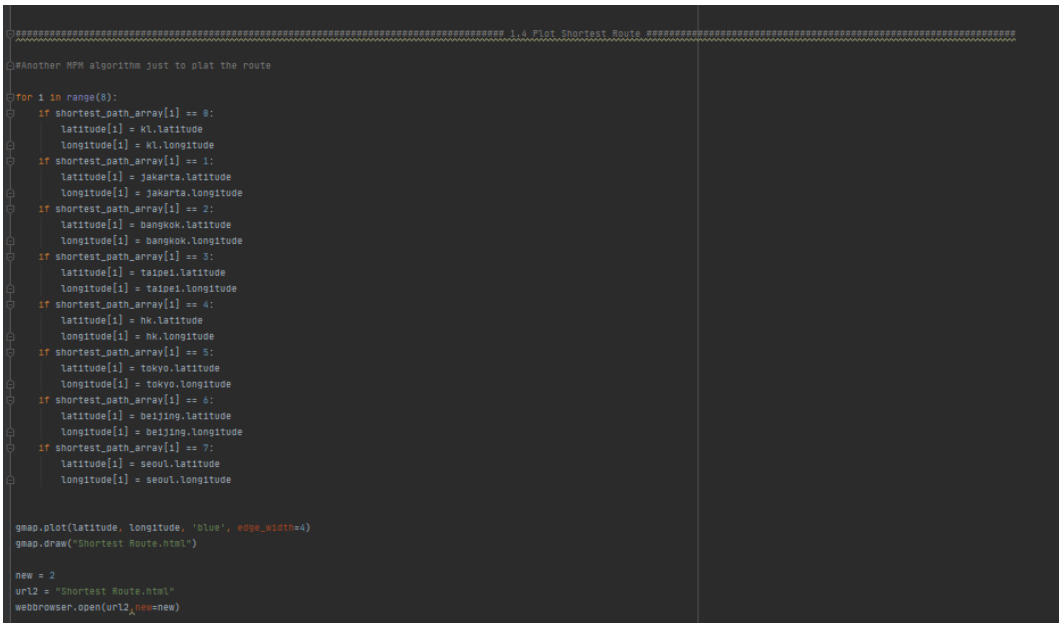
1) The Timsort algorithm was used to sort the array of distances calculated in the previous part in increasing order. Index I will later be used as the path with the shortest distance. This algorithm has a time complexity of O(n log n).

```
Unsorted path distances:
 [12798.24, 11857.15, 12086.36, 10940.3, 12211.55, 12006.58, 13959.6, 14084.79, 14314.0, 13296.45, 13372.91, 12230.17, 13783.29, 1

Sorted path distances:
 [9882.24, 10263.52, 10585.5, 10790.47, 10940.3, 11028.3, 11204.61, 11421.21, 11421.6, 11570.55, 11746.86, 11802.88, 11832.79, 118
```

Figure 1.9 – Array of distances found and sorted after using Timsort

Problem 1.4 - Plot Line Between Destinations



```
#Another MPM algorithm just to plat the route

for i in range(8):
    if shortest_path_array[i] == 0:
        latitude[i] = kl.latitude
        longitude[i] = kl.longitude
    if shortest_path_array[i] == 1:
        latitude[i] = jakarta.latitude
        longitude[i] = jakarta.longitude
    if shortest_path_array[i] == 2:
        latitude[i] = bangkok.latitude
        longitude[i] = bangkok.longitude
    if shortest_path_array[i] == 3:
        latitude[i] = taipei.latitude
        longitude[i] = taipei.longitude
    if shortest_path_array[i] == 4:
        latitude[i] = hk.latitude
        longitude[i] = hk.longitude
    if shortest_path_array[i] == 5:
        latitude[i] = tokyo.latitude
        longitude[i] = tokyo.longitude
    if shortest_path_array[i] == 6:
        latitude[i] = beijing.latitude
        longitude[i] = beijing.longitude
    if shortest_path_array[i] == 7:
        latitude[i] = seoul.latitude
        longitude[i] = seoul.longitude


gmap.plot(latitude, longitude, 'blue', edge_width=4)
gmap.draw("Shortest Route.html")

new = 2
url2 = "Shortest Route.html"
webbrowser.open(url2,new=new)
```

Figure 1.9 gmplot shortest route

1) The segment of code above assigns each node its respective latitudes and longitudes in an array by iterating through a for loop.
2) The arrays of longitudes and latitudes in the order of the shortest path is then passed through as arguments in the *gmplot.plot()* function to plot the shortest route on Google Maps.

```
The shortest travel plan:
 Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Taipei > Beijing > Seoul > Tokyo >
 Total distance:  9882.24 km
```

Figure 1.10 – Suggested travel path

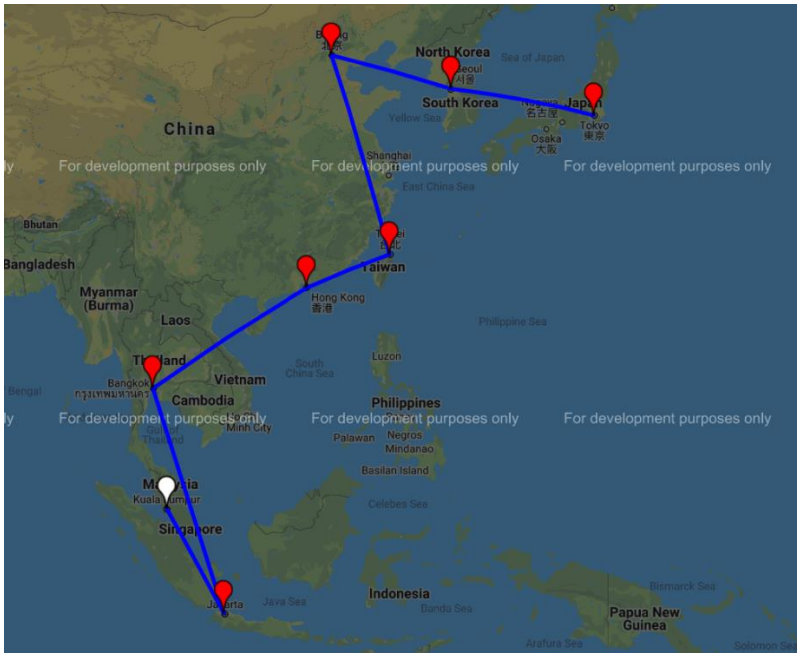


Figure 1.7 Shortest travel plan

Problem 2

Problem 2.5 Extract Information From Major Economic News Websites

From question 5, we are required to find 5 articles for 7 cities. We found all the article and using html-to-text application on textise.net to compile the articles into 7 separate text files for each cites in one folder

For each text file, we using use the built-in *open* function to read the article in text form. The open function returns a file object that contains methods and attributes to perform various operations on the file.

```
jakartaIO = open('News/jakarta.txt', 'r', encoding='utf-8-sig')
jakarta_text = jakartaIO.read().lower()
jakartaIO.close()

bangkokIO = open('News/bangkok.txt', 'r', encoding='utf-8-sig')
bangkok_text = bangkokIO.read().lower()
bangkokIO.close()

taipeiIO = open('News/taipei.txt', 'r', encoding='utf-8-sig')
taipei_text = taipeiIO.read().lower()
taipeiIO.close()

hongkongIO = open('News/hongkong.txt', 'r', encoding='utf-8-sig')
hongkong_text = hongkongIO.read().lower()
hongkongIO.close()

tokyoIO = open('News/tokyo.txt', 'r', encoding='utf-8-sig')
tokyo_text = tokyoIO.read().lower()
tokyoIO.close()

beijingIO = open('News/beijing.txt', 'r', encoding='utf-8-sig')
beijing_text = beijingIO.read().lower()
beijingIO.close()

seoulIO = open('News/seoul.txt', 'r', encoding='utf-8-sig')
seoul_text = seoulIO.read().lower()
seoulIO.close()
```

The class frequency is the function to calculate and print out the frequency of each words in the output. The words in the text file is split first and then using for loop to calculate frequency of each words. The words then are print out to show the result

```
104  # get frequency of words in a text
105  def frequency(text, city):
106      list_of_words = text.split()
107      freq = {}
108      for word in list_of_words:
109          freq[word] = freq.get(word, 0) + 1
110      keys = freq.keys()
111
112      print("Frequencies of word for " + city + "'s article:\n\n" + str(freq) + "\n")
113
114
```

Then using the driver code to run the frequency class for each city. Notice that we print out each city frequency with the city name for easy visualization of the output.

```
5   # print frequency of each word in text for every cities' article
6
7   frequency(jakarta_text, 'Jakarta')
8   frequency(bangkok_text, 'Bangkok')
9   frequency(taipei_text, 'Taipei')
0   frequency(hongkong_text, 'HongKong')
1   frequency(tokyo_text, 'Tokyo')
2   frequency(beijing_text, 'Beijing')
3   frequency(seoul_text, 'Seoul')
4
```

Here the sample output for frequency

```
Frequencies of word for Jakarta's article:

{'‘crisis': 1, 'like': 2, 'no': 3, 'other’': 1, 'will': 14, 'shrink': 6, 'indonesia’s': 11, 'economy,': 5, 'imf': 5, 'forecasts': 3, '[image]': 5, 'economy': 10, 'is': 19, 'on': 25

Frequencies of word for Bangkok's article:

{'second': 5, 'half': 5, 'to': 85, 'witness': 2, 'more': 10, 'business': 7, 'closures': 3, 'smes': 1, 'particularly': 2, 'at': 23, 'risk,': 1, 'says': 2, 'econthai': 2, '[people':

Frequencies of word for Taipei's article:

{'us': 7, 'still': 2, 'nation’s': 2, 'largest': 5, 'debtor': 3, 'in': 34, 'q1:': 1, 'central': 6, 'bank': 7, '-': 5, 'taipei': 12, 'times': 6, '[image]': 5, '*': 10, 'by': 15, 'cr

Frequencies of word for HongKong's article:

{'coronavirus:': 2, 'how': 6, 'crowdfunding': 2, 'helped': 1, 'a': 68, 'live': 4, 'music': 4, 'venue': 4, 'survive': 3, 'hong': 52, "kong's": 2, 'shutdown': 1, '|': 5, 'kong': 38,

Frequencies of word for Tokyo's article:

{'bank': 5, 'of': 45, 'japan': 19, 'likely': 4, 'to': 65, 'cut': 2, 'economic': 10, 'forecasts': 4, 'again': 2, 'on': 11, 'covid-19': 4, '|': 5, 'the': 150, 'times': 7, 'is': 19,

Frequencies of word for Beijing's article:

{'beijing': 19, 'stops': 1, 'travel': 2, 'as': 16, 'virus': 11, 'spike': 3, 'continues': 1, 'millions': 2, 'of': 69, 'people': 24, 'in': 84, 'are': 30, 'living': 4, 'under': 7, 're

Frequencies of word for Seoul's article:

{'business': 2, 'calls': 1, 'for': 18, 'regulatory': 6, 'changes': 1, 'in': 22, 'korea': 15, '[kcci]': 1, 'kcci': 1, 'south': 5, 'korean': 5, 'businesspeople': 2, 'have': 11, 'call
```

Then we need to filter stop word inside the text file so that we can show total number of stop word in each city and total word count for each city. As the question requested, we will be using Rabin Karp Algorithm to filter the word by matching each word in articles with the stop words and then remove the stop word.

We copy the stop word that we get from https://www.ranks.nl/stopwords and put it in one array that we can call later

```
stopwords = ['a', 'about', 'above', 'after', 'again', 'against', 'all', 'am', 'an', 'and',
             'any', 'are', "aren't", 'as', 'at', 'be', 'because', 'been', 'before', 'being',
             'below', 'between', 'both', 'but', 'by', "can't", 'cannot', 'could', "couldn't", 'did',
             "didn't", 'do', 'does', "doesn't", 'doing', "don't", 'down', 'during', 'each', 'few',
             'for', 'from', 'further', 'had', "hadn't", 'has', "hasn't", 'have', "haven't", 'having',
             'he', "he'd", "he'll", "he's", 'her', 'here', "here's", 'hers', 'herself', 'him',
             'himself', 'his', 'how', "how's", 'i', "i'd", "i'll", "i'm", "i've", 'if', 'in', 'into',
             'is', "isn't", 'it', "it's", 'its', 'itself', "let's", 'me', 'more', 'most', "mustn't",
             'my', 'myself', 'no', 'nor', 'not', 'of', 'off', 'on', 'once', 'only', 'or', 'other',
             'ought', 'our', 'ours', 'ourselves', 'out', 'over', 'own', 'same', "shan't", 'she', "she'd",
             "she'll", "she's", 'should', "shouldn't", 'so', 'some', 'such', 'than', 'that', "that's",
             'the', 'their', 'theirs', 'them', 'themselves', 'then', 'there', "there's", 'these', 'they',
             "they'd", "they'll", "they're", "they've", 'this', 'those', 'through', 'to', 'too', 'under',
             'until', 'up', 'very', 'was', "wasn't", 'we', "we'd", "we'll", "we're", "we've", 'were', "weren't",
             'what', "what's", 'when', "when's", 'where', "where's", 'which', 'while', 'who', "who's", 'whom', 'why',
             "why's", 'with', "won't", 'would', "wouldn't", 'you', "you'd", "you'll", "you're", "you've",
             'your', 'yours', 'yourself', 'yourselves']
```

We create word_count method to calculate the total wordcount for the article and the total number of stop word for each city articles. We using .split() function to split the article string into list for ease of usage for next part. Then in a for loop, we using Rabin Karp Algorithm to find the stop word inside the article and then remove it and increase the stop count by 1. This method will return both total number of stop word in article and total number or word in article.

```
def word_count(text):
    stop_count = 0
    list_of_words = text.split()
    for word in stopwords:
        if rabin_karp_matcher(word, text):
            stop_count = stop_count + 1
            # delete stop words
            text = text.lower().replace(word, "", 1)

    return [stop_count, len(list_of_words,)]


jakarta_stop_count, jakarta_total_words = word_count(jakarta_text)
bangkok_stop_count, bangkok_total_words = word_count(bangkok_text)
taipei_stop_count, taipei_total_words = word_count(taipei_text)
hongkong_stop_count, hongkong_total_words = word_count(hongkong_text)
tokyo_stop_count, tokyo_total_words = word_count(tokyo_text)
beijing_stop_count, beijing_total_words = word_count(beijing_text)
seoul_stop_count, seoul_total_words = word_count(seoul_text)
```

Here is the Rabin Karp Algorithm code that we use in our program. It is use to compare the stop word and the articles. The average time complexity of Rabin Karp Algorithm is O (m + n).

```
def search(pat, txt, q):
    M = len(pat)
    N = len(txt)
    pat = pat.replace(" ", "")
    p = 0  # hash value for pattern
    t = 0  # hash value for txt
    h = pow(d, M-1)

    result = False
    if M > N:
        return result
    else:
        # preprocessing
        for i in range(M):
            p = (d * p + ord(pat[i].lower())) % q
            t = (d * t + ord(txt[i].lower())) % q
        for s in range(N - M + 1):  # note the +1
            if p == t:  # check character by character
                match = True
                for i in range(M):
                    if pat[i].lower() != txt[s + i].lower():
                        match = False
                        break
                if match:
                    result = True
            if s < N - M:
                t = (t - h * ord(txt[s].lower())) % q  # remove letter s
                t = (t * d + ord(txt[s + M].lower())) % q  # add letter s+m
                t = (t + q) % q  # make sure that t >= 0
        return result
```

```
39
40  def rabin_karp_matcher(pattern, text):
41      return search(pattern, text, 2207)
42
```

Problem 2.6 Plot a Histogram Related to the Word Count

Next, we need to visualize our data from problem 2.5 by using Plotly. We decided to use histogram for this part as we can easily see which data are which without any confusion. Plotly.py is an interactive, open-source, and browser-based graphing library for Python. We are using Plotly.py version 4.8 as now it supports offline where we can directly use it from our library without needing any API key like version before.

We will first import Plotly into our project as shown below

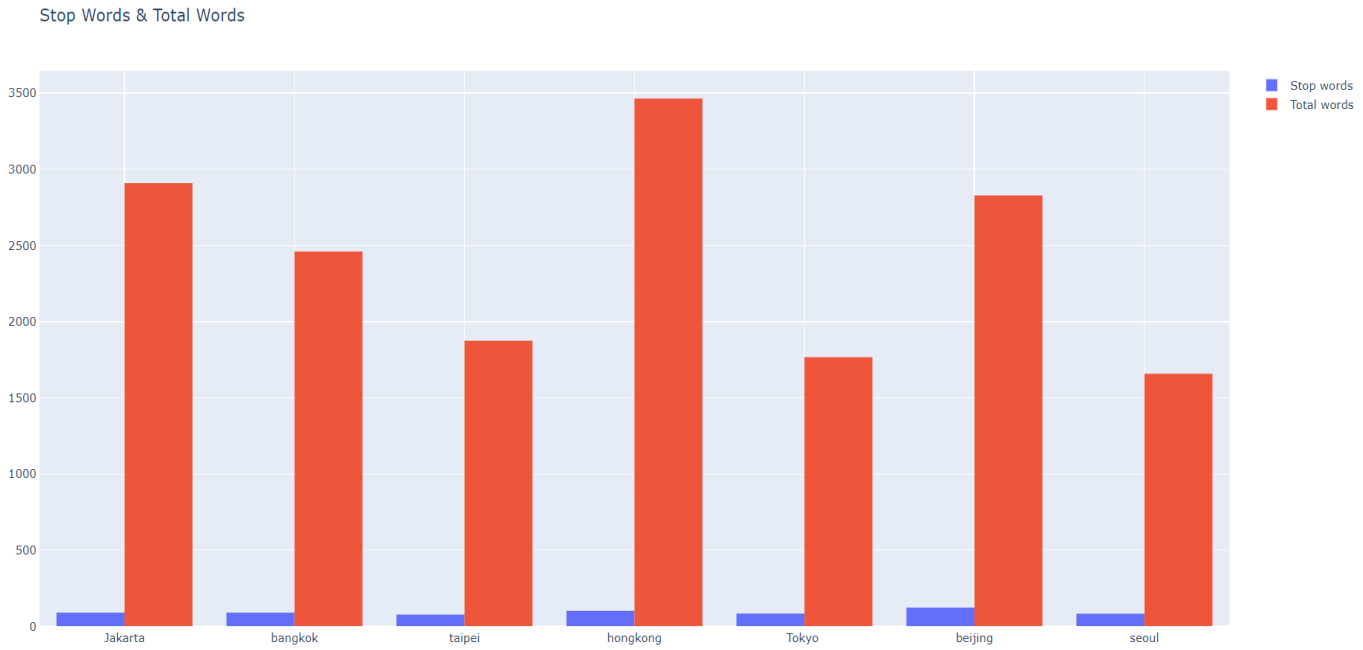
```
import plotly.graph_objects as go
```

We take the stop count and word count from previous part and us it for the histogram. As we only have 2 variables, the histogram is created as shown below and we customize the layout so we can name each variable in the histogram later.

```
x = ["Jakarta", "bangkok","taipei","hongkong","Tokyo ","beijing","seoul"]
stop_counts = [jakarta_stop_count,bangkok_stop_count,taipei_stop_count,hongkong_stop_count,tokyo_stop_count,beijing_stop_count,seoul_stop_count]
total_words = [jakarta_total_words,bangkok_total_words,taipei_total_words,hongkong_total_words,tokyo_total_words,beijing_total_words,seoul_total_words]

data = [
    go.Histogram(
        histfunc="sum",
        y=stop_counts,
        x=x,
        name="Stop words"
    ),
    go.Histogram(
        histfunc="sum",
        y=total_words,
        x=x,
        name="Total words"
    )
]
layout = go.Layout(
    title=go.layout.Title(
        text="Stop Words & Total Words",
        xref='paper',
        x=0
    )
)
fig = go.Figure(data=data, layout=layout)
fig.show()
```

Here the output that we get after we create the histogram using plotly.py



Problem 2.7 Compare Positive, Negative, and Neutral Words

On this part, we are required to compare words in webpage with negative, positive and neutral English using our chosen string-matching Algorithm.

First, we will read the positive word that we get from http://positivewordsresearch.com/list-of-positive-words/ and http://positivewordsresearch.com/list-of-negative-words/ using open() function. This way we can easily access the negative and positive word in the next part.

```
# getting the frequency of positive, negative and neutral words in a text
def wordcount(text):
    total_length = len(text.split())
    count = 0
    positive = 0
    negative = 0

    for pat in positive_text:
        pat = pat.replace(" ", "")
        if rabin_karp_matcher(pat, text):
            positive = positive + 1
            count = count + 1
    for pat in negative_text:
        pat = pat.replace(" ", "")
        if rabin_karp_matcher(pat, text):
            negative = negative + 1
            count = count + 1
    # neutral word is equal to the total words in text minus the total count
    # of words that is positive or negative
    neutral = total_length - count
    return positive, negative, neutral
```

Here is the Rabin Karp Algorithm code that we use in our program. It is use to compare the stop word and the articles. The average time complexity of Rabin Karp Algorithm is O(m + n).

We create wordcount method to calculate the total wordcount for the article and the total number of positive, negative and neutral for each city articles. We using .split() function to split the article string into list for ease of usage for next part. Then in a for loop, we using Rabin Karp Algorithm to find the stop word inside the article and then remove it and increase the count by 1. This method will return both total number of positive, negative and neutral word in article and total number or word in article.

Next we will print out the negative count, positive count and neutral count for each article.

```
print("\nJakarta's article word count")
print("Positive word: " + str(jakarta_pos) + " word(s)")
print("Negative word: " + str(jakarta_neg) + " word(s)")
print("Neutral word: " + str(jakarta_neutral) + " word(s)")

print("\nBangkok's article word count")
print("Positive word: " + str(bangkok_pos) + " word(s)")
print("Negative word: " + str(bangkok_neg) + " word(s)")
print("Neutral word: " + str(bangkok_neutral) + " word(s)")

print("\nTaipei's article word count")
print("Positive word: " + str(taipei_pos) + " word(s)")
print("Negative word: " + str(taipei_neg) + " word(s)")
print("Neutral word: " + str(taipei_neutral) + " word(s)")

print("\nHong Kong's article word count")
print("Positive word: " + str(hongkong_pos) + " word(s)")
print("Negative word: " + str(hongkong_neg) + " word(s)")
print("Neutral word: " + str(hongkong_neutral) + " word(s)")

print("\nTokyo's article word count")
print("Positive word: " + str(tokyo_pos) + " word(s)")
print("Negative word: " + str(tokyo_neg) + " word(s)")
print("Neutral word: " + str(tokyo_neutral) + " word(s)")

print("\nBeijing's article word count")
print("Positive word: " + str(beijing_pos) + " word(s)")
print("Negative word: " + str(beijing_neg) + " word(s)")
print("Neutral word: " + str(beijing_neutral) + " word(s)")

print("\nSeoul's article word count")
print("Positive word: " + str(seoul_pos) + " word(s)")
print("Negative word: " + str(seoul_neg) + " word(s)")
```

Here the output that we get from this part.

```
Jakarta's article word count
Positive word: 380 word(s)
Negative word: 124 word(s)
Neutral word: 2405 word(s)

Bangkok's article word count
Positive word: 319 word(s)
Negative word: 92 word(s)
Neutral word: 2049 word(s)

Taipei's article word count
Positive word: 260 word(s)
Negative word: 79 word(s)
Neutral word: 1536 word(s)

Hong Kong's article word count
Positive word: 412 word(s)
Negative word: 157 word(s)
Neutral word: 2895 word(s)

Tokyo's article word count
Positive word: 217 word(s)
Negative word: 83 word(s)
Neutral word: 1467 word(s)

Beijing's article word count
Positive word: 344 word(s)
Negative word: 119 word(s)
Neutral word: 2365 word(s)

Seoul's article word count
Positive word: 260 word(s)
Negative word: 91 word(s)
Neutral word: 1307 word(s)
```

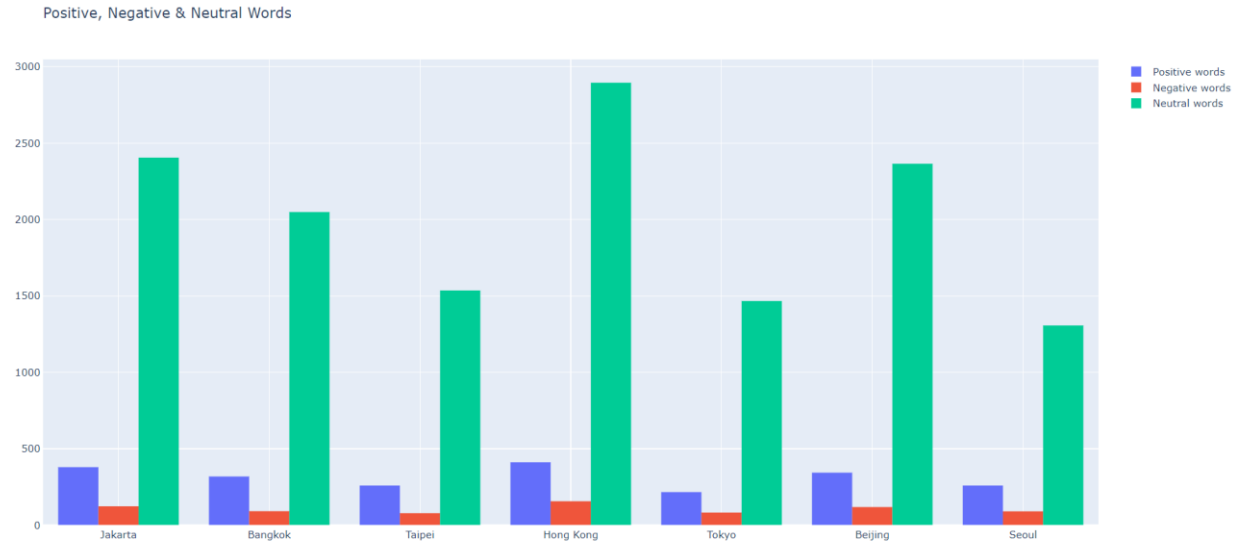
Problem 2.8 Plot a Histogram of Positive and Negative Words

The next part will produce a histogram of positive and negative word. Three arrays are created and filled with positive count, negative count and neutral count for each city. Next, the mode for the graph is chosen, which is markers and the title is also initialized. Finally, the layout of the graph is created with all the information and the graph is printed out.

```
x = ["Jakarta", "Bangkok", "Taipei", "Hong Kong", "Tokyo", "Beijing", "Seoul"]
positive_y = [jakarta_pos, bangkok_pos, taipei_pos, hongkong_pos, tokyo_pos, beijing_pos, seoul_pos]
negative_y = [jakarta_neg, bangkok_neg, taipei_neg, hongkong_neg, tokyo_neg, beijing_neg, seoul_neg]
neutral_y = [jakarta_neutral, bangkok_neutral, taipei_neutral, hongkong_neutral, tokyo_neutral,
             beijing_neutral, seoul_neutral]

data = [
    go.Histogram(
        histfunc="sum",
        y=positive_y,
        x=x,
        name="Positive words"
    ),
    go.Histogram(
        histfunc="sum",
        y=negative_y,
        x=x,
        name="Negative words"
    ),
    go.Histogram(
        histfunc="sum",
        y=neutral_y,
        x=x,
        name="Neutral words"
     )
]
layout = go.Layout(
    title=go.layout.Title(
        text="Positive, Negative & Neutral Words",
        xref='paper',
        x=0
    )
)
```

Here the output from after we plot the histogram using Plotly



Problem 2.9 Algorithmic Sentiment Conclusion

For the next part, we have to give algorithmic conclusion regarding the sentiment for each article. Thus, we calculate for each city, either the positive count for city is higher that negative count or not. If the city has higher positive count, we can conclude that the article is giving positive and have good financial situation. Also, we can conclude that the city is giving negative sentiment and have negative financial situational if it has a higher negative count. Although, if the number of positive and negative count in the article is the same, we can say the city have a neutral sentiment.

```
fig = go.Figure(data=data, layout=layout)
fig.show()
#  Sentiment & Conclusion  #
def sentiment(positive_frequency, negative_frequency, city):
    print("\n" + city.upper())
    if positive_frequency > negative_frequency:
        x = positive_frequency/negative_frequency
        print('The articles are giving positive sentiment')
        print('So the country has positive economic/financial situation of a ratio ')
        print(value(positive_frequency, negative_frequency))
    elif negative_frequency > positive_frequency:
        x = positive_frequency/negative_frequency
        print('The articles are giving negative sentiment')
        print('So the country has negative economic/financial situation of a ratio ')
        print(value(positive_frequency, negative_frequency))
    else:
        x = positive_frequency / negative_frequency
        print('The articles are giving neutral sentiment')
        print('So the country has neutral economic/financial situation of a ratio ')
        print(value(positive_frequency, negative_frequency))
```

Here the driver code to print the sentiment method

```
print("\n Concluding the cities' economic/financial situation")
sentiment(jakarta_pos, jakarta_neg, "Jakarta")
sentiment(bangkok_pos, bangkok_neg, "Bangkok")
sentiment(taipei_pos, taipei_neg, "Taipei")
sentiment(hongkong_pos, hongkong_neg, "Hong Kong")
sentiment(tokyo_pos, tokyo_neg, "Tokyo")
sentiment(beijing_pos, beijing_neg, "Beijing")
sentiment(seoul_pos, seoul_neg, "Seoul")
```

Here the output for sentiment method

```
 Concluding the cities' economic/fjnancial situation

JAKARTA
The articles are giving positive sentiment
So the country has positive economic/financial situation of a ratio
3.064516129032258

BANGKOK
The articles are giving positive sentiment
So the country has positive economic/financial situation of a ratio
3.467391304347826

TAIPEI
The articles are giving positive sentiment
So the country has positive economic/financial situation of a ratio
3.2911392405063293

HONG KONG
The articles are giving positive sentiment
So the country has positive economic/financial situation of a ratio
2.624203821656051

TOKYO
The articles are giving positive sentiment
So the country has positive economic/financial situation of a ratio
2.6144578313253013

BEIJING
The articles are giving positive sentiment
So the country has positive economic/financial situation of a ratio
2.8907563025210083
```

```
SEOUL
The articles are giving positive sentiment
So the country has positive economic/financial situation of a ratio
2.857142857142857
```

Next, we can easily make a simple method to calculate the “sentiment value” for each city. We calculate it in a way like a ratio. First, we take positive count from each city. Then we also take negative count from each city. Next, we make a formula where

$$e = \frac{Positive\ count}{Negative\ count}$$

e is equal to ratio of positive sentiment/negative sentiment. This value can be use for the next problem

```
def value(positive, negative):
    e = positive/negative
    return e
```

Here are the driver code for the sentiment ratio value.

```
jakartaValue= round(value(jakarta_pos, jakarta_neg),2)
bangkokValue= round(value(bangkok_pos, bangkok_neg),2)
taipeiValue= round(value(taipei_pos, taipei_neg),2)
hongkongValue= round(value(hongkong_pos, hongkong_neg),2)
tokyoValue= round(value(tokyo_pos, tokyo_neg),2)
beijingValue= round(value(beijing_pos, beijing_neg),2)
seoulValue= round(value(seoul_pos, seoul_neg),2)
```

Problem 3

Problem 3.10 – Path Optimization

```
############################################################ 3.10 Find The Most Optimal Route Based on Sentiment ##########################################################
city = [kl_coordinates, jakarta_coordinates, bangkok_coordinates, hk_coordinates, taipei_coordinates, beijing_coordinates, seoul_coordinates, tokyo_coordinates]

optimal = ''
optimal_array = []

sentiment_arr = data[:]

#To Find the most optimal path as a baseline
def optimalPath():
    for i in range(1, 8):

        for j in range(i + 1, 8):

            if sentiment_arr[j] > sentiment_arr[i]:
                distance_ratio = (int(great_circle(city[j], city[i - 1]).kilometers) - int(
                    great_circle(city[i], city[i - 1]).kilometers)) / int(great_circle(city[i], city[i - 1]).kilometers)

                if distance_ratio < 0.4:
                    sentiment_ratio = (sentiment_arr[j] - sentiment_arr[i]) / sentiment_arr[i]

                    if sentiment_ratio > 0.02:

                        for k in range(i, j + 1):
                            temp = sentiment_arr[k]
                            sentiment_arr[k] = sentiment_arr[j]
                            sentiment_arr[j] = temp

optimalPath()
#Another MPM algorithm just to plat the route
```

Figure 3.1 – optimalPath Algorithm Implementation

1) For problem 3, variables and arrays from solving problems 1 and 2 are used to optimize a route for ben. Firstly, the best possible path is calculated by using the optimalPath algorithm to modify the original shortest path route planned for Ben.
2) The algorithm iterates throughout an array which holds the shortest path to compare the next city set for Ben to travel to with other cities in terms of the sentiment analysis and distance ratios.
3) The algorithm checks to see if other cities if travelled to have a better sentiment ratio, a difference in distance from the original plan that is less than 40%, and a difference in sentiment ratio between the original next city and the other cities that is not less than 2%. If these criteria are fulfilled, the next city ben plans to travel to will be shifted up whilst the city that has been checked to fulfil the criteria will be put in its place.
4) The optimal path is then printed out and saved in as an array to be used later.

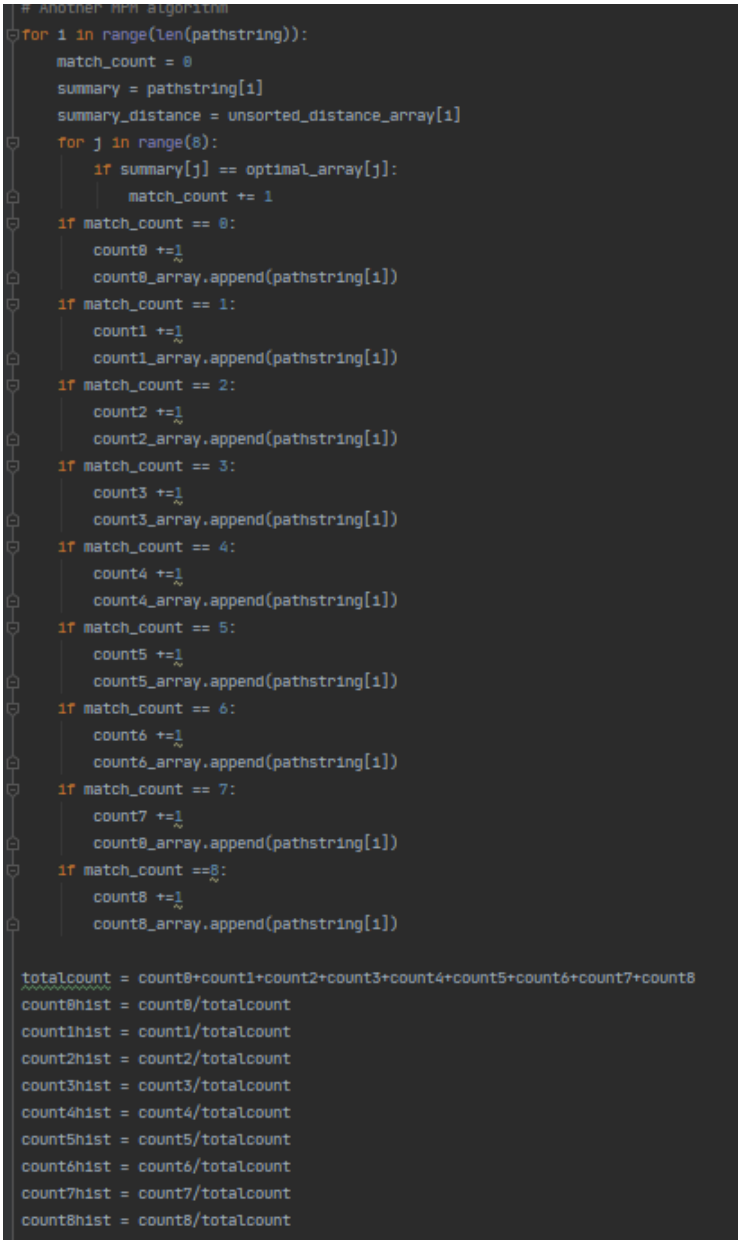
```
optimalPath()
#Another MPM algorithm just to plat the route

for i in range(8):
    if sentiment_arr[i] == data[0]:
        optimal = optimal + "Kuala Lumpur > "
        optimal_array.append(0)
    if sentiment_arr[i] == data[1]:
        optimal = optimal + "Jakarta > "
        optimal_array.append(1)
    if sentiment_arr[i] == data[2]:
        optimal = optimal + "Bangkok > "
        optimal_array.append(2)
    if sentiment_arr[i] == data[3]:
        optimal = optimal + "Taipei > "
        optimal_array.append(3)
    if sentiment_arr[i] == data[4]:
        optimal = optimal + "Hong Kong > "
        optimal_array.append(4)
    if sentiment_arr[i] == data[5]:
        optimal = optimal + "Tokyo > "
        optimal_array.append(5)
    if sentiment_arr[i] == data[6]:
        optimal = optimal + "Beijing > "
        optimal_array.append(6)
    if sentiment_arr[i] == data[7]:
        optimal = optimal + "Seoul > "
        optimal_array.append(7)

print("\nMost optimal route based on distance and sentiment: \n",optimal)
```

Figure 3.2 MPM algorithm to print optimal route

5) After storing the optimal route in an array, the MPM algorithm will iterate through the route to match each index element with the respective cities to be printed



```
# Another MPM algorithm
for i in range(len(pathstring)):
    match_count = 0
    summary = pathstring[i]
    summary_distance = unsorted_distance_array[i]
    for j in range(8):
        if summary[j] == optimal_array[j]:
            match_count += 1
    if match_count == 0:
        count0 +=1
        count0_array.append(pathstring[i])
    if match_count == 1:
        count1 +=1
        count1_array.append(pathstring[i])
    if match_count == 2:
        count2 +=1
        count2_array.append(pathstring[i])
    if match_count == 3:
        count3 +=1
        count3_array.append(pathstring[i])
    if match_count == 4:
        count4 +=1
        count4_array.append(pathstring[i])
    if match_count == 5:
        count5 +=1
        count5_array.append(pathstring[i])
    if match_count == 6:
        count6 +=1
        count6_array.append(pathstring[i])
    if match_count == 7:
        count7 +=1
        count0_array.append(pathstring[i])
    if match_count ==8:
        count8 +=1
        count8_array.append(pathstring[i])

totalcount = count0+count1+count2+count3+count4+count5+count6+count7+count8
count0hist = count0/totalcount
count1hist = count1/totalcount
count2hist = count2/totalcount
count3hist = count3/totalcount
count4hist = count4/totalcount
count5hist = count5/totalcount
count6hist = count6/totalcount
count7hist = count7/totalcount
count8hist = count8/totalcount
```

Figure 3.3 MPM algorithm to categorize routes.

6) In order to list the routes from least recommended to most recommended, all Hamiltonian paths are compared with the optimal route found. For every path, the algorithm will iterate through path array to count the number of cities in that path and order that matches with that in the optimal path. Once the entire path array has been iterated through, it will append that path into another array (e.g. count1array) according to the number of matches.

```
def arraySummary(summary):
    global optimal
    optimal = ''
    for i in range(8):
        if summary[1] == 0:
            optimal = optimal + "Kuala Lumpur > "
        if summary[1] == 1:
            optimal = optimal + "Jakarta > "
        if summary[1] == 2:
            optimal = optimal + "Bangkok > "
        if summary[1] == 3:
            optimal = optimal + "Taipei > "
        if summary[1] == 4:
            optimal = optimal + "Hong Kong > "
        if summary[1] == 5:
            optimal = optimal + "Tokyo > "
        if summary[1] == 6:
            optimal = optimal + "Beijing > "
        if summary[1] == 7:
            optimal = optimal + "Seoul > "
    print(optimal)
```

Figure 3.4 Print route function

7) The routes for each path will be printed out in yet another MPM algorithm that matches the nodes in the path array to the city. This function will be called upon in the next part.

```
print("\n#############################################################################")
print("In order of LEAST RECOMMENDED to MOST RECOMMENDED: ")
print("#############################################################################\n")

print("-----------------------------------------------------------------------------")
print("Routes with 0 matches to optimal route NON EXISTENT")
print("-----------------------------------------------------------------------------")
print()
print("-----------------------------------------------------------------------------")
print("Routes with 1 matches to optimal route ( ABSOLUTELY NOT RECOMMENDED ) :")
print("-----------------------------------------------------------------------------")
for i in range(len(count1_array)):
    arraySummary(count1_array[i])
print()
print("-----------------------------------------------------------------------------")
print("Routes with 2 matches to optimal route ( NOT RECOMMENDED ) :")
print("-----------------------------------------------------------------------------")
for i in range(len(count2_array)):
    arraySummary(count2_array[i])
print()
print("-----------------------------------------------------------------------------")
print("Routes with 3 matches to optimal route ( NOT RECOMMENDED ) :")
print("-----------------------------------------------------------------------------")
for i in range(len(count3_array)):
    arraySummary(count3_array[i])
print()
print("-----------------------------------------------------------------------------")
print("Routes with 4 matches to optimal route ( AVERAGE ) :")
print("-----------------------------------------------------------------------------")
for i in range(len(count4_array)):
    arraySummary(count4_array[i])
print()
print("-----------------------------------------------------------------------------")
print("Routes with 5 matches to optimal route ( RECOMMENDED ) :")
print("-----------------------------------------------------------------------------")
for i in range(len(count5_array)):
```

Figure 3.5 Print out recommendations

8) After appending all the paths into their respective categories, the paths will be printed in ascending order from the least recommended to the most recommended

```
--------------------------------------------------------------------------
Routes with 0 matches to optimal route NON EXISTENT
--------------------------------------------------------------------------

--------------------------------------------------------------------------
Routes with 1 matches to optimal route ( ABSOLUTELY NOT RECOMMENDED ) :
--------------------------------------------------------------------------
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Taipei > Tokyo > Seoul > Beijing >
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Taipei > Seoul > Beijing > Tokyo >
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Tokyo > Taipei > Seoul > Beijing >
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Tokyo > Seoul > Taipei > Beijing >
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Tokyo > Seoul > Beijing > Taipei >
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Beijing > Taipei > Seoul > Tokyo >
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Beijing > Tokyo > Seoul > Taipei >
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Beijing > Seoul > Taipei > Tokyo >
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Seoul > Taipei > Beijing > Tokyo >
Kuala Lumpur > Jakarta > Bangkok > Hong Kong > Seoul > Tokyo > Taipei > Beijing >
```

Figure 3.6 Example of recommendations printed

```
--------------------------------------------------------------------------
Routes with 7 matches with optimal route NON EXISTENT
--------------------------------------------------------------------------

--------------------------------------------------------------------------
Routes with COMPLETE MATCH with optimal route ( PERFECT MATCH - MOST OPTIMAL! ) :
--------------------------------------------------------------------------
Kuala Lumpur > Bangkok > Jakarta > Taipei > Hong Kong > Beijing > Tokyo > Seoul >

##########################################################################
Most optimal route based on distance and sentiment:

Kuala Lumpur > Bangkok > Jakarta > Taipei > Hong Kong > Beijing > Tokyo > Seoul >
##########################################################################
```

Figure 3.7 Output of Most Optimal route based on sentiment

```
totalcount = count0+count1+count2+count3+count4+count5+count6+count7+count8
count0hist = count0/totalcount
count1hist = count1/totalcount
count2hist = count2/totalcount
count3hist = count3/totalcount
count4hist = count4/totalcount
count5hist = count5/totalcount
count6hist = count6/totalcount
count7hist = count7/totalcount
count8hist = count8/totalcount

Optimal_Route_Match = [count0hist, count1hist, count2hist, count3hist, count4hist, count5hist, count6hist,count7hist,count8hist]

x = ["0","1","2","3","4","5","6","7","8"]

data = [
    go.Bar(

        y=Optimal_Route_Match,
        x=x,

    ),
]
layout = go.Layout(
    title=go.layout.Title(
        text="Probability Frequency Distribution",
        xref='paper',
        x=0
    )
)
fig = go.Figure(data=data, layout=layout)
fig.show()
```

Figure 3.7 Probability frequency distribution calculation

9) For the final part of the program, the probability frequency distribution was calculated by assigning values to the *counthist* variables with the formula *count/totalcount.* This shows the possible routes grouped with n number of matches with the optimal route found after running the program. The *counthist* variables will ten be used to plot a probability frequency distribution graph to visualize the distribution of routes based on the number of cities matched in the optimal route.

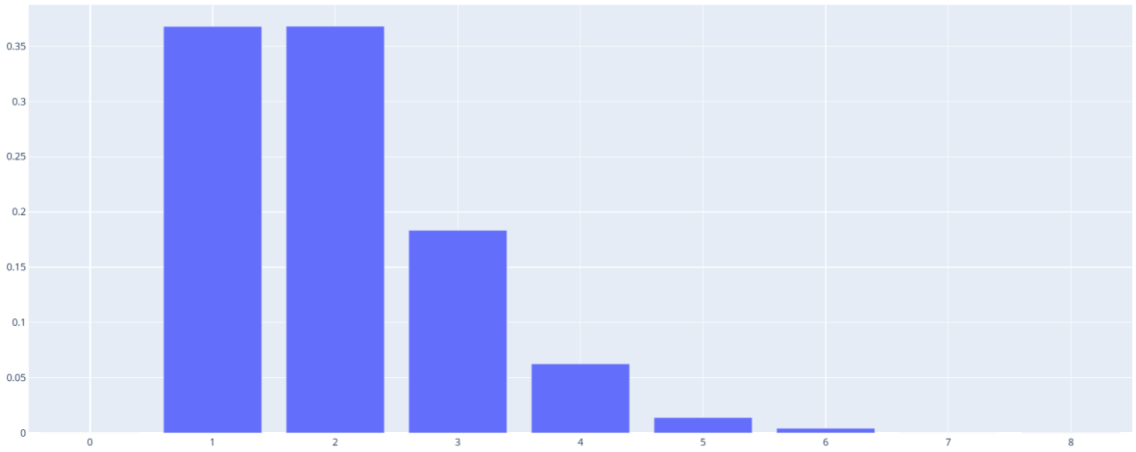


Figure 3.8 Probability frequency distribution of paths

10) Shown in the graph above is the distribution of possible routes Ben can take. This was plotted through the use of the Plotly library. The frequency of the optimal route is not seen in the graph as the probability of that route is P(X = x) = 1/5040.

Algorithms Used

1. Backtracking algorithm

   The backtracking algorithm is a general algorithm that uses a brute force approach but is more refined. It solves problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree). We implemented this algorithm in Problem 1 to find Hamiltonian paths for all the possible routes. This algorithm has a time complexity of O (n^ n) derived from O(n-1!)

2. Bubble-Calc Algorithm

   The Bubble-Calc algorithm uses an iterative approach based on the bubble sorting algorithm to calculate the distances between cities by matching indexes of a path array in adjacent pairs for every iteration until of the path. This Algorithm has an average time complexity of (n^2)

3. Tim Sort Algorithm

   Tim-sort is a sorting algorithm derived from insertion sort and merge sort. It was designed to perform in an optimal way on different kind of real-world data. The steps are:

1. Divide the array into the number of blocks known as run.
2. Consider size of run either 32 or 64(in the code, size of run is 32.)

3. Sort the individual elements of every run one by one using insertion sort.
4. Merge the sorted runs one by one using merge function of merge sort.
5. Double the size of merged sub-arrays after every iteration.

We used this approach to sort the shortest distance travelled for all possible paths that Ben can take in ascending order. This sorting algorithm has an average and worst case time complexity of O (n log n) but has a best case time complexity of O (n) when everything is almost sorted or sorted.

4.Mini Path Mapper (MPM) algorithm

These algorithms are iterative algorithm that matches arrays to the corresponding city arrays since most of the arrays contain integer elements. the MPM algorithms just have to iterate each array index to match that of the array indexes of the various cities and paths to find the corresponding values in any given array. This is used multiple times throughout the code to match array values with each other.

5. Word counter algorithm

Word counter algorithm is used to calculate the frequency of words that occur in the articles found. This algorithm is used in Problem 2.

6. Rabin-Karp pattern matching algorithm

Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function. Unlike Naive string-matching algorithm, it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison. We modified the Rabin-Karp algorithm to come up with our own algorithm to delete the stop-words and to calculate the number of positive, negative and neutral words in Problem 2. This pattern matching algorithm has a best-case and average time complexity of O (n + m) but a worst-case time complexity of O (nm).

7. String-matching algorithm

We use this algorithm when there is a need to find an input pattern in a string of characters. Usually, in case of a short string, Python programmers prefer to use the naïve approach in which, the program checks each position in the input string for the query pattern. In case it matches, it given an output with the position number. This algorithm is used to compare the negative, positive and neutral words after deleting the stop-words in Problem 2.

8. Sentiment value ratio algorithm

This algorithm that we came up with used the sentiment-analysis to generate a ratio using the positive count over negative count. This ratio is to be used in Problem 3 to find the best optimal path.

9. Optimal path algorithm

This algorithm adapts an insertion sort implementation by having a 3 nested for loop. It iterates for every city but performs a comparison between the city originally planned based solely only the distance and the cities based on sentiment and travel distance below the given percentages for the iteration of the number of cities. This algorithm has time complexity of O(n^3).

New Algorithms Used (Explanation)

Backtracking Algorithm

Backtracking uses a brute force approach to solve a problem but is vastly similar to that employed with the naïve approach. Unlike the Depth-First-Search algorithm, the backtracking algorithm does not utilize parent-child relationships and is called upon recursively to generate every possible outcome for a desired solution. It is different to the naïve approach in the sense that a solution or criteria is already specified and only needs to generate outcomes to reach the solution.

For this project, the implementation of the backtracking algorithm is used to find all possible Hamiltonian paths in a graph network. It does this by first creating an adjacency matrix based on the nodes and edges, and an empty array to append the nodes to. The algorithm will call on itself recursively based on the values in the adjacency matrix to see if the next unvisited node will lead to a solution. If it does, it will continue to do so until a Hamiltonian path (the solution we want) is found and append it to the array of paths. If the next node does not lead to a solution, the algorithm will break off there and mark that path as False so it does not need to generate an outcome for that path. This algorithm will continue to be called recursively until all desired solutions are found.

As mentioned before, based on the projects, as there are 8 cities where Ben does not have to return to the city of origin. The number of desired outcomes (Hamiltonian routes where each node is visited exactly once) should be (n-1)! In theory; that results in a total of 5040 possible paths to be discovered by this algorithm.

Bubble-Calc Algorithm

The Bubble-Calc algorithm is a new algorithm we created ourselves based on the bubble sorting technique. Instead of performing a sort however, a calculation operation is performed instead to add to the total distance of a path. This algorithm has a time complexity of O (n^2).

For example:

Note: each node corresponds to a coordinate in the program

Paths = [[0, 2, 3, 4, 5] , [0, 1, 3, 4, 5], [0, 2, 3, 5, 4]]
i=0 i=1 i=2

when i = 0, the array that will be operated on first is [0, 2, 3, 4, 5]

First iteration of j will compare the next index element with itself and calculate the total distance between the two indexes.

[(0 , 2), 3, 4, 5]
j

total distance = calculated distance between index elements + total distance

The 2nd iteration of j will compare the next index element with itself and calculate the total distance between the two indexes.

[0 , (2, 3), 4, 5]
j
total distance = calculated distance between index elements + total distance

The 3rd iteration of j will compare the next index element with itself and calculate the total distance between the two indexes.

[0 , 2, (3, 4), 5]
j

total distance = calculated distance between index elements + total distance

The algorithm will iterate till j = length of the path array and then *i* will increment to perform the operations on the next path in the *Paths* array.

Timsort Sorting Algorithm

Introduction of the algorithm

Tim-sort is a sorting algorithm derived from insertion sort and merge sort. It was designed to perform in an optimal way on different kind of real-world data.

Steps:

6. Divide the array into the number of blocks known as run.
7. Consider size of run either 32 or 64(in the code, size of run is 32.)
8. Sort the individual elements of every run one by one using insertion sort.
9. Merge the sorted runs one by one using merge function of merge sort.
10. Double the size of merged sub-arrays after every iteration.

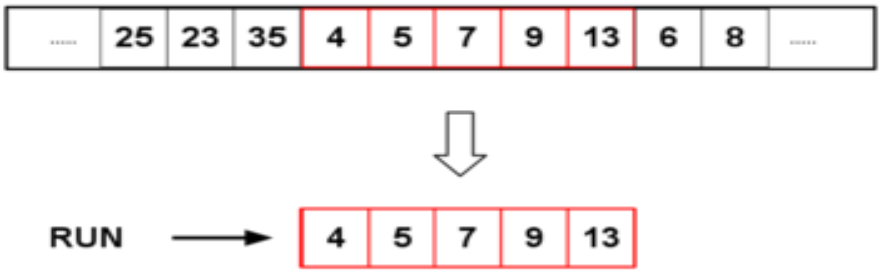


Figure: Creating minrun which is then sorted using insertion sort and merged back using merge sort

Explanation of the code

```
172
173     minrun = 32
174
175     def InsSort(arr, start, end):
176         for i in range(start + 1, end + 1):
177             elem = arr[i]
178             j = i - 1
179             while j >= start and elem < arr[j]:
180                 arr[j + 1] = arr[j]
181                 j -= 1
182             arr[j + 1] = elem
183         return arr
184
```

InsSort function is used to perform Insertion Sort on the region of the array being passed to function. start is the starting element's index number and end is the index of the last element of the region. Instead of starting from 0, the function starts from start. And similarly, it ends with end. The condition in the inner loop becomes j>=start instead of j>=0. The function returns the array after sorting the mentioned subarray in place.

```
185
186 def merge(arr, start, mid, end):
187     if mid == end:
188         return arr
189
190     first = arr[start:mid + 1]
191     last = arr[mid + 1:end + 1]
192     len1 = mid - start + 1
193     len2 = end - mid
194     ind1 = 0
195     ind2 = 0
196     ind = start
197
198     while ind1 < len1 and ind2 < len2:
199         if first[ind1] < last[ind2]:
200             arr[ind] = first[ind1]
201             ind1 += 1
202         else:
203             arr[ind] = last[ind2]
204             ind2 += 1
205         ind += 1
206
207     while ind1 < len1:
208         arr[ind] = first[ind1]
209         ind1 += 1
210         ind += 1
211
212     while ind2 < len2:
213         arr[ind] = last[ind2]
214         ind2 += 1
215         ind += 1
216
217     return arr
```

In the merge function, it merges the two given subarray indices using the merge sort algorithm. Here again, we start with index start and end with end. The `first` array should go from start to mid + 1, while the `last` array should go from mid + 1 to end + 1. Mid is basically the index value of the last element of the subarray first. The function then returns the array after merging the mentioned subarrays in place.

```
219
220 def TimSort(arr):
221     n = len(arr)
222     # Start by slicing and sorting small portions of the
223     # input array. The size of these slices is defined by
224     # your `minrun` size.
225     for start in range(0, n, minrun):
226         end = min(start + minrun - 1, n - 1)
227         arr = InsSort(arr, start, end)
228
229     curr_size = minrun
230     while curr_size < n:
231         # Compute the `mid` (where the first array ends
232         # and the second starts) and the `end` (where
233         # the second array ends)
234         for start in range(0, n, curr_size * 2):
235             mid = min(n - 1, start + curr_size - 1)
236             end = min(n - 1, mid + curr_size)
237             arr = merge(arr, start, mid, end)
238         curr_size *= 2
239     return arr
```

The Timsort function acts like the driver function that calls the above-mentioned functions with values pertaining to the logic of Timsort algorithm. It starts by slicing and sorting small portions of the input array. The size of these slices is defined by the `minrun` size. Then it computes the `mid` where the first array ends and the second starts and the end where the second array ends. Each iteration then doubles the size of the arrays until it becomes a single sorted array.

Time Complexity:

On average, the complexity of Timsort is O (n logn). The logarithmic part comes from doubling the size of the run to perform each linear merge operation. However, Timsort performs exceptionally well on already-sorted or close-to-sorted lists, leading to a best-case scenario of O(n).

optimalPath Algorithm

This algorithm is a new algorithm that was created specifically for problem 3. We drew inspiration from the insertion sort implementation but it is vastly different. By having 3 nested for-loops, it iterates for every city but performs a comparison between the city originally planned based solely only the distance and the cities based on sentiment and travel distance below the given percentages for the iteration of the number of cities. This algorithm will continue for the n length of the shortest path array and is run to modify the existing shortest path to an optimal path based on the sentiment analysis (better sentiment analysis), distance (<40% in difference with the original path), and sentiment ratio (>2%) criteria. This algorithm has time complexity of O(n^3).

Example:
checking

Shortest_path = [0 , 1 , 2 , 3 , 4 , 5]
i j k

Pointer 'I' points to the index where Ben is currently at. Pointer 'j' points to the city Ben is meant to travel to based on the on the shortest route plan from problem 1. Pointer 'k' points to the first city to check for better sentiment analysis, route distance, and sentiment analysis ratio.

If the value at k doesn't meet the criteria, k increments by 1.

Shortest_path = [0 , 1 , 2 , 3 , 4 , 5]
i j k

If the value at k still doesn't meet the criteria, k increments by 1.

Shortest_path = [0 , 1 , 2 , 3 , 4 , 5]
i j k

sorting

When k meets the criteria it will swap positions with j. This will be done until j increments to match the k

```
Shortest_path = [ 0 , 1 , 2 , 3 , 4 , 5 ]
                    i   j         k
```

```
Shortest_path = [ 0 , 4 , 2 , 3 , 1 , 5 ]
                    i   j         k
```

After swapping, j will increment by 1 until it matches j.

```
Shortest_path = [ 0 , 4 , 2 , 3 , 1 , 5 ]
                    i       j       k
```

```
Shortest_path = [ 0 , 4 , 1 , 3 , 2 , 5 ]
                    i       j       k
```

```
Shortest_path = [ 0 , 4 , 1 , 3 , 2 , 5 ]
                    i           j   k
```

```
Shortest_path = [ 0 , 4 , 1 , 2 , 3 , 5 ]
                    i           j   k
```

```
Shortest_path = [ 0 , 4 , 1 , 2 , 3 , 5 ]
                    i             j=k
```

When j finally matches with k, i will increment by 1 to run the next check of iterations. Anything to the left of i has been sorted and is in the right place. This also means Ben has flown from that previous city to the next. This will then reset pointers j and k

```
Shortest_path = [ 0 , 4 , 1 , 2 , 3 , 5 ]
                      i   j   k
```

Tools, Libraries, and APIs

Python: The language used to code this entire project. It is a powerful interpreted, high-level, general-purpose programming language with many Libraries and supported APIs that were useful in completing the tasks in this project.

Pycharm: Pycharm is an integrated development environment used in computer programming specially for the Python language. We used this to develop the solutions of all three problems and to integrate the whole project.

Geopy: Geopy is a Python client for geocoding web services. Geopy is used to locate cities to their coordinates and vice versa. Geopy is also used to find geodesic distance and the great circle distance between points. In problem 1, we used Geopy to find the coordinates of each of the 8 cities and also to calculate the great circle distance between them.

Gmplot: It is a Python library which allows user to plot data on google maps. Gmplot is used to plot each city on google maps, place markers and also to draw a map showing the shortest route.

Plotly: Plotly library allows us to visualize and analyze data. We used to Plotly in problem 2 to plot two histograms showing total word count, count of stop words and count of positive negative and neutral words.

Pyvis: The Pyvis library is used for quick generation of visual network graphs. We used to plot the network graph of all city connections.

Textise.online: It converts web pages into plain-text. We used this in problem 2 to extract text from the articles of each city to enable the program to read it easily.

Conclusion

There were many major problems encountered throughout the course of this project. Among them was not being able to discuss, plan and workout the project face to face. It was difficult to understand, debug and integrate each other's tasks through online platforms. Also, each of the members had a different schedule routine for exams and assignments so we faced difficulty in managing time and allocating duties. Moreover, all of us had just begun learning Python so this was our very first project using new tools with Python like the google API and Plotly so it took quite a time to research about the new algorithms and tools before starting the project. Regarding problem 2, the runtime was long because the positive word and negative word has very long number of string so the system was very slow to give the output. In addition, we were not sure how to interpret problem 3 and how we are expected to present the solution.

However, we learned and developed many new skills while doing the project. We learned to use and integrate different libraries and tools of Python such as Geopy, Gmplot, Plotly, Pyvis etc. We also learned many new algorithms that were not taught in class like Timsort, Backtracking, and the Boyer-Moore-Horspool String Matching Algorithm etc. Finally, we could also perceive the true meaning of teamwork and cooperation. Ultimately, through relentless hard work and combined effort of each member we were able to build a fully functional system which plans the most optimized journey for Ben.

References

Ali, A. (2016, July 01). Counting Word Frequency in a File Using Python. Retrieved from https://code.tutsplus.com/tutorials/counting-word-frequency-in-a-file-using-python--cms-25965

Hamiltonian path problem. (2020, June 30). Retrieved from https://en.wikipedia.org/wiki/Hamiltonian_path_problem

Print all Hamiltonian path present in a graph. (2020, May 23). Retrieved from https://www.techiedelight.com/print-all-hamiltonian-path-present-in-a-graph/

Hua, M., & Pei, J. (n.d.). *Probabilistic Path Queries in Road Networks: Traffic Uncertainty Aware Path Selection* (pp. 1-12). doi:http://www.openproceedings.org/2010/conf/edbt/HuaP10.pdf

Python Advanced Course Topics. (n.d.). Retrieved from https://python-course.eu/graphs_python.php

The Editors of Encyclopaedia Britannica. (2018, January 16). NP-complete problem. Retrieved from https://www.britannica.com/science/NP-complete-problem

TimSort Sorting Algorithm Implementation in Python. (2019, September 09). Retrieved from https://www.codespeedy.com/timsort-algorithm-implementation-in-python/

Travelling Salesman Problem: Set 1 (Naive and Dynamic Programming). (2018, September 06). Retrieved from https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

Tutorial¶. (n.d.). Retrieved from https://pyvis.readthedocs.io/en/latest/tutorial.html

Appendix

Appendix A – Source Code

```
import webbrowser

import gmplot
import timeit
from geopy.geocoders import Nominatim
from geopy.distance import great_circle
from pyvis.network import Network

start = timeit.default_timer()

##############################################################################
### 1.1 Get Locations and Plot Map
#################################################################

#Geocoding coordinates for cities
geolocator = Nominatim(user_agent="geoapi")

kl = geolocator.geocode("Kuala Lumpur")
jakarta = geolocator.geocode("Jakarta")
bangkok = geolocator.geocode("Bangkok")
taipei = geolocator.geocode("Taipei")
hk = geolocator.geocode("Hong Kong")
tokyo = geolocator.geocode("Tokyo")
beijing = geolocator.geocode("Beijing")
seoul = geolocator.geocode("Seoul")

kl_coordinates = (kl.latitude, kl.longitude)
jakarta_coordinates = (jakarta.latitude, jakarta.longitude)
bangkok_coordinates = (bangkok.latitude, bangkok.longitude)
taipei_coordinates = (taipei.latitude, taipei.longitude)
hk_coordinates = (hk.latitude, hk.longitude)
tokyo_coordinates = (tokyo.latitude, tokyo.longitude)
beijing_coordinates = (beijing.latitude, beijing.longitude)
seoul_coordinates = (seoul.latitude, seoul.longitude)

#Initial map plotting
apikey=''
gmap = gmplot.GoogleMapPlotter(kl.latitude, kl.longitude, 4, apikey=apikey)
latitude = [kl.latitude, jakarta.latitude, bangkok.latitude, taipei.latitude,
hk.latitude, tokyo.latitude, beijing.latitude, seoul.latitude]
longitude= [kl.longitude, jakarta.longitude, bangkok.longitude, taipei.longitude,
hk.longitude, tokyo.longitude, beijing.longitude, seoul.longitude]
gmap.scatter(latitude, longitude, color='red', size=40, marker=True)
gmap.marker(kl.latitude,kl.longitude, color='white')
gmap.draw("Map.html")

new = 2
url = "Map.html"
webbrowser.open(url,new=new)


##############################################################################
###### 1.2 Get Distances
#####################################################################

distances = [[0 for i in range(8)] for j in range(8)]
city_coordinates = [kl_coordinates, jakarta_coordinates, bangkok_coordinates,
taipei_coordinates, hk_coordinates, tokyo_coordinates, beijing_coordinates,
seoul_coordinates]

for i in range(8):
```

```
    for j in range(8):
        distances[i][j] = great_circle(city_coordinates[i],
city_coordinates[j]).kilometers

net=Network("700px", "1000px")

net.add_node(0, label="Kuala Lumpur", size=10)
net.add_node(1, label="Jakarta", size=10)
net.add_node(2, label="Bangkok", size=10)
net.add_node(3, label="Taipei", size=10)
net.add_node(4, label="Hong Kong", size=10)
net.add_node(5, label="Tokyo", size=10)
net.add_node(6, label="Beijing", size=10)
net.add_node(7, label="Seoul", size=10)

for i in range(8):
    for j in range(8):
        if i is not j:
            value = str(round(great_circle(city_coordinates[i],
city_coordinates[j]).kilometers, 2)) + " km"
            net.add_edge(i, j, label=value, font_color="white")

net.toggle_physics(True)
net.barnes_hut(spring_length=200)
net.show_buttons()
net.show("Travel Distances.html")

###############################################################################
1.3 Journey Planner
###############################################################################


class Graph:
    # Constructor
    def __init__(self, edges, N):
        # A List of Lists to represent an adjacency list
        self.adjList = [[] for _ in range(N)]

        # add edges to the undirected graph
        for (src, dest) in edges:
            self.adjList[src].append(dest)
            self.adjList[dest].append(src)

hamiltonian_count= 0 #count for number of hamiltonian paths discovered
pathstring = [] #array to store all hamiltonian paths
unsorted_distance_array = [] #array to store path distances

#############################################################
#     Backtracking algorithm to find hamiltonian paths      #
#############################################################

def hamiltonian_path(g, v, visited, path, N, pathstring):
    if len(path) == N:
        # print hamiltonian path
        global hamiltonian_count
        hamiltonian_count+=1

        pathstring.append(path[:]) #append to global array
        print(path)

        return

    # Check if every edge starting from vertex v leads to a solution or not
    for w in g.adjList[v]:

        # process only unvisited vertices as hamiltonian
        # path visits each vertex exactly once
        if not visited[w]:
```

```
            visited[w] = True
            path.append(w)

            # check next vertex (w) to see if path is a valid hamiltonian
            hamiltonian_path(g, w, visited, path, N, pathstring)

            # Backtracking function
            visited[w] = False
            path.pop()

#Driver code for the backtracking algorithm
if __name__ == '__main__':
    #edgelist assignment
    edges = [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7),
             (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7),
             (2, 3), (2, 4), (2, 5), (2, 6), (2, 7),
             (3, 4), (3, 5), (3, 6), (3, 7),
             (4, 5), (4, 6), (4, 7),
             (5, 6), (5, 7),
             (6, 7)]

    N = 8 #number of nodes

    g = Graph(edges, N) #create graph

    start = 0 #fixed starting node

    path = [start] #assigned a starting path

    # mark start node as visited
    visited = [False] * N
    visited[start] = True

    print("\nHamiltonian paths found:")
    hamiltonian_path(g, start, visited, path, N, pathstring)
    print("\nTotal hamiltonian paths: ", hamiltonian_count)

############################################################
#bubble-calc algorithm to calculate distances of all paths #
############################################################

def pathdistance(pathstring, city_coordinates, unsorted_distance_array):

    temp_distance = 0

    for i in range(len(pathstring)):
        current_path = pathstring[i]
        for j in range(7):
            temp_distance = round(great_circle(city_coordinates[current_path[j]],
city_coordinates[current_path[j+1]]).kilometers + temp_distance, 2)

        unsorted_distance_array.append(temp_distance)
        temp_distance = 0

pathdistance(pathstring, city_coordinates, unsorted_distance_array)


##################################################
#      Timsort algorithm to sort path lengths    #
##################################################
minrun = 32

#Implementation of insertion sort
def InsSort(arr, start, end):
    for i in range(start + 1, end + 1):
        elem = arr[i]
        j = i - 1
        while j >= start and elem < arr[j]:
```

```
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = elem
    return arr

#Implementation of merge sort
def merge(arr, start, mid, end):
    if mid == end:
        return arr
    first = arr[start:mid + 1]
    last = arr[mid + 1:end + 1]
    len1 = mid - start + 1
    len2 = end - mid
    ind1 = 0
    ind2 = 0
    ind = start

    while ind1 < len1 and ind2 < len2:
        if first[ind1] < last[ind2]:
            arr[ind] = first[ind1]
            ind1 += 1
        else:
            arr[ind] = last[ind2]
            ind2 += 1
        ind += 1

    while ind1 < len1:
        arr[ind] = first[ind1]
        ind1 += 1
        ind += 1

    while ind2 < len2:
        arr[ind] = last[ind2]
        ind2 += 1
        ind += 1

    return arr

# Actual Timsort implementation
def TimSort(arr):
    n = len(arr)

    for start in range(0, n, minrun):
        end = min(start + minrun - 1, n - 1)
        arr = InsSort(arr, start, end)

    curr_size = minrun
    while curr_size < n:
        for start in range(0, n, curr_size * 2):
            mid = min(n - 1, start + curr_size - 1)
            end = min(n - 1, mid + curr_size)
            arr = merge(arr, start, mid, end)
        curr_size *= 2
    return arr


# Driver code for the TimSort Algorithm
sorted_distance_array = unsorted_distance_array[:]

#Distance Array to be sorted
n = len(sorted_distance_array)

print("\nUnsorted path distances:\n", unsorted_distance_array)

TimSort(sorted_distance_array)
print("\nSorted path distances:\n", sorted_distance_array)

##############################################
```

```
#    Mini-Path-Mapper (MPM) algorithms       #
#############################################

shortest_path_array = [] #Important as the nodes for the shortest path are stored
in order here
shortest_path_string= 'Kuala Lumpur > ' #solely for output purposes

for i in range(len(sorted_distance_array)):
    if unsorted_distance_array[i] == sorted_distance_array[0]:
        shortest_path_array = pathstring[i]

for i in range(1, 8):
    if shortest_path_array[i] == 1:
        shortest_path_string = shortest_path_string + 'Jakarta > '
    elif shortest_path_array[i]== 2:
        shortest_path_string = shortest_path_string + 'Bangkok > '
    elif shortest_path_array[i]== 3:
        shortest_path_string = shortest_path_string + 'Taipei > '
    elif shortest_path_array[i]== 4:
        shortest_path_string = shortest_path_string + 'Hong Kong > '
    elif shortest_path_array[i]== 5:
        shortest_path_string = shortest_path_string + 'Tokyo > '
    elif shortest_path_array[i]== 6:
        shortest_path_string = shortest_path_string + 'Beijing > '
    elif shortest_path_array[i]== 7:
        shortest_path_string = shortest_path_string + 'Seoul > '

print("\nThe shortest travel plan:\n", shortest_path_string)
print(" Total distance: ", sorted_distance_array[0], "km")


#############################################################################
### 1.4 Plot Shortest Route
#############################################################

#Another MPM algorithm just to plat the route

for i in range(8):
    if shortest_path_array[i] == 0:
        latitude[i] = kl.latitude
        longitude[i] = kl.longitude
    if shortest_path_array[i] == 1:
        latitude[i] = jakarta.latitude
        longitude[i] = jakarta.longitude
    if shortest_path_array[i] == 2:
        latitude[i] = bangkok.latitude
        longitude[i] = bangkok.longitude
    if shortest_path_array[i] == 3:
        latitude[i] = taipei.latitude
        longitude[i] = taipei.longitude
    if shortest_path_array[i] == 4:
        latitude[i] = hk.latitude
        longitude[i] = hk.longitude
    if shortest_path_array[i] == 5:
        latitude[i] = tokyo.latitude
        longitude[i] = tokyo.longitude
    if shortest_path_array[i] == 6:
        latitude[i] = beijing.latitude
        longitude[i] = beijing.longitude
    if shortest_path_array[i] == 7:
        latitude[i] = seoul.latitude
        longitude[i] = seoul.longitude


gmap.plot(latitude, longitude, 'blue', edge_width=4)
gmap.draw("Shortest Route.html")

new = 2
```

```
url2 = "Shortest Route.html"
webbrowser.open(url2,new=new)

################################################################################
### 2.5  ###################################################################

import plotly.offline
import plotly.graph_objects as go
import timeit

# d is the number of characters in the input alphabet
d = 256


def search(pat, txt, q):
    M = len(pat)
    N = len(txt)
    pat = pat.replace(" ", "")
    p = 0  # hash value for pattern
    t = 0  # hash value for txt
    h = pow(d, M-1)

    result = False
    if M > N:
        return result
    else:
        # preprocessing
        for i in range(M):
            p = (d * p + ord(pat[i].lower())) % q
            t = (d * t + ord(txt[i].lower())) % q
        for s in range(N - M + 1):  # note the +1
            if p == t:  # check character by character
                match = True
                for i in range(M):
                    if pat[i].lower() != txt[s + i].lower():
                        match = False
                        break
                if match:
                    result = True
            if s < N - M:
                t = (t - h * ord(txt[s].lower())) % q  # remove letter s
                t = (t * d + ord(txt[s + M].lower())) % q  # add letter s+m
                t = (t + q) % q  # make sure that t >= 0
        return result


def rabin_karp_matcher(pattern, text):
    return search(pattern, text, 2207)


stopwords = ['a', 'about', 'above', 'after', 'again', 'against', 'all', 'am', 'an',
'and',
             'any', 'are', "aren't", 'as', 'at', 'be', 'because', 'been', 'before',
'being',
             'below', 'between', 'both', 'but', 'by', "can't", 'cannot', 'could',
"couldn't", 'did',
             "didn't", 'do', 'does', "doesn't", 'doing', "don't", 'down', 'during',
'each', 'few',
             'for', 'from', 'further', 'had', "hadn't", 'has', "hasn't", 'have',
"haven't", 'having',
             'he', "he'd", "he'll", "he's", 'her', 'here', "here's", 'hers',
'herself', 'him',
             'himself', 'his', 'how', "how's", 'i', "i'd", "i'll", "i'm", "i've",
'if', 'in', 'into',
             'is', "isn't", 'it', "it's", 'its', 'itself', "let's", 'me', 'more',
'most', "mustn't",
             'my', 'myself', 'no', 'nor', 'not', 'of', 'off', 'on', 'once', 'only',
'or', 'other',
```

```
            'ought', 'our', 'ours', 'ourselves', 'out', 'over', 'own', 'same',
"shan't", 'she', "she'd",
            "she'll", "she's", 'should', "shouldn't", 'so', 'some', 'such',
'than', 'that', "that's",
            'the', 'their', 'theirs', 'them', 'themselves', 'then', 'there',
"there's", 'these', 'they',
            "they'd", "they'll", "they're", "they've", 'this', 'those', 'through',
'to', 'too', 'under',
            'until', 'up', 'very', 'was', "wasn't", 'we', "we'd", "we'll",
"we're", "we've", 'were', "weren't",
            'what', "what's", 'when', "when's", 'where', "where's", 'which',
'while', 'who', "who's", 'whom', 'why',
            "why's", 'with', "won't", 'would', "wouldn't", 'you', "you'd",
"you'll", "you're", "you've",
            'your', 'yours', 'yourself', 'yourselves']



jakartaIO = open('News/jakarta.txt', 'r', encoding='utf-8-sig')
jakarta_text = jakartaIO.read().lower()
jakarta_text = jakarta_text.replace("\n", " ")
jakartaIO.close()

bangkokIO = open('News/bangkok.txt', 'r', encoding='utf-8-sig')
bangkok_text = bangkokIO.read().lower()
bangkok_text = bangkok_text.replace("\n", " ")
bangkokIO.close()

taipeiIO = open('News/taipei.txt', 'r', encoding='utf-8-sig')
taipei_text = taipeiIO.read().lower()
taipei_text = taipei_text.replace("\n", " ")
taipeiIO.close()

hongkongIO = open('News/hongkong.txt', 'r', encoding='utf-8-sig')
hongkong_text = hongkongIO.read().lower()
hongkong_text = hongkong_text.replace("\n", " ")
hongkongIO.close()

tokyoIO = open('News/tokyo.txt', 'r', encoding='utf-8-sig')
tokyo_text = tokyoIO.read().lower()
tokyo_text = tokyo_text.replace("\n", " ")
tokyoIO.close()

beijingIO = open('News/beijing.txt', 'r', encoding='utf-8-sig')
beijing_text = beijingIO.read().lower()
beijing_text = beijing_text.replace("\n", " ")
beijingIO.close()

seoulIO = open('News/seoul.txt', 'r', encoding='utf-8-sig')
seoul_text = seoulIO.read().lower()
seoul_text = seoul_text.replace("\n", " ")
seoulIO.close()


# get frequency of words in a text
def frequency(text, city):
    list_of_words = text.split()
    freq = {}
    for word in list_of_words:
        freq[word] = freq.get(word, 0) + 1
    keys = freq.keys()

    print("Frequencies of word for " + city + "'s article:\n\n" + str(freq) + "\n")


# print frequency of each word in text for every cities' article

frequency(jakarta_text, 'Jakarta')
```

```
frequency(bangkok_text, 'Bangkok')
frequency(taipei_text, 'Taipei')
frequency(hongkong_text, 'HongKong')
frequency(tokyo_text, 'Tokyo')
frequency(beijing_text, 'Beijing')
frequency(seoul_text, 'Seoul')

def word_count(text):
    stop_count = 0
    list_of_words = text.split()
    for word in stopwords:
        if rabin_karp_matcher(word, text):
            stop_count = stop_count + 1
            # delete stop words
            text = text.lower().replace(word, "", 1)

    return [stop_count, len(list_of_words,)]


jakarta_stop_count, jakarta_total_words = word_count(jakarta_text)
bangkok_stop_count, bangkok_total_words = word_count(bangkok_text)
taipei_stop_count, taipei_total_words = word_count(taipei_text)
hongkong_stop_count, hongkong_total_words = word_count(hongkong_text)
tokyo_stop_count, tokyo_total_words = word_count(tokyo_text)
beijing_stop_count, beijing_total_words = word_count(beijing_text)
seoul_stop_count, seoul_total_words = word_count(seoul_text)

#print(tokyo1_stop_count)
#print(tokyo1_total_words)
#print(tokyo1_text)



x = ["Jakarta", "bangkok","taipei","hongkong","Tokyo ","beijing","seoul"]
stop_counts =
[ jakarta_stop_count,bangkok_stop_count,taipei_stop_count,hongkong_stop_count,tokyo
_stop_count,beijing_stop_count,seoul_stop_count]
total_words =
[ jakarta_total_words,bangkok_total_words,taipei_total_words,hongkong_total_words,t
okyo_total_words,beijing_total_words,seoul_total_words]

data = [
    go.Histogram(
        histfunc="sum",
        y=stop_counts,
        x=x,
        name="Stop words"
    ),
    go.Histogram(
        histfunc="sum",
        y=total_words,
        x=x,
        name="Total words"
    )
]
layout = go.Layout(
    title=go.layout.Title(
        text="Stop Words & Total Words",
        xref='paper',
        x=0
    )
)
fig = go.Figure(data=data, layout=layout)
fig.show()
```

```
jakartaIO = open('News/jakarta.txt', 'r', encoding='utf-8-sig')
jakarta_text = jakartaIO.read().lower()

jakartaIO.close()

bangkokIO = open('News/bangkok.txt', 'r', encoding='utf-8-sig')
bangkok_text = bangkokIO.read().lower()

bangkokIO.close()

taipeiIO = open('News/taipei.txt', 'r', encoding='utf-8-sig')
taipei_text = taipeiIO.read().lower()

taipeiIO.close()

hongkongIO = open('News/hongkong.txt', 'r', encoding='utf-8-sig')
hongkong_text = hongkongIO.read().lower()

hongkongIO.close()

tokyoIO = open('News/tokyo.txt', 'r', encoding='utf-8-sig')
tokyo_text = tokyoIO.read().lower()

tokyoIO.close()

beijingIO = open('News/beijing.txt', 'r', encoding='utf-8-sig')
beijing_text = beijingIO.read().lower()

beijingIO.close()

seoulIO = open('News/seoul.txt', 'r', encoding='utf-8-sig')
seoul_text = seoulIO.read().lower()

seoulIO.close()


positive_word = open('Words/positivewords.txt', 'r', encoding='utf-8-sig')
positive_text = positive_word.read().lower().split('\n')

negative_word = open('Words/negativewords.txt', 'r', encoding='utf-8-sig')
negative_text = negative_word.read().lower().split('\n')


# getting the frequency of positive, negative and neutral words in a text
def wordcount(text):
    total_length = len(text.split())
    count = 0
    positive = 0
    negative = 0

    for pat in positive_text:
        pat = pat.replace(" ", "")
        if rabin_karp_matcher(pat, text):
            positive = positive + 1
            count = count + 1
    for pat in negative_text:
        pat = pat.replace(" ", "")
        if rabin_karp_matcher(pat, text):
            negative = negative + 1
            count = count + 1
    # neutral word is equal to the total words in text minus the total count
    # of words that is positive or negative
    neutral = total_length - count
    return positive, negative, neutral


# getting the no. of positive, negative and neutral words in the text
jakarta_pos, jakarta_neg, jakarta_neutral = wordcount(jakarta_text)
```

```
bangkok_pos, bangkok_neg, bangkok_neutral = wordcount(bangkok_text)
taipei_pos, taipei_neg, taipei_neutral = wordcount(taipei_text)
hongkong_pos, hongkong_neg, hongkong_neutral = wordcount(hongkong_text)
tokyo_pos, tokyo_neg, tokyo_neutral = wordcount(tokyo_text)
beijing_pos, beijing_neg, beijing_neutral = wordcount(beijing_text)
seoul_pos, seoul_neg, seoul_neutral = wordcount(seoul_text)


print("\nJakarta's article word count")
print("Positive word: " + str(jakarta_pos) + " word(s)")
print("Negative word: " + str(jakarta_neg) + " word(s)")
print("Neutral word: " + str(jakarta_neutral) + " word(s)")

print("\nBangkok's article word count")
print("Positive word: " + str(bangkok_pos) + " word(s)")
print("Negative word: " + str(bangkok_neg) + " word(s)")
print("Neutral word: " + str(bangkok_neutral) + " word(s)")

print("\nTaipei's article word count")
print("Positive word: " + str(taipei_pos) + " word(s)")
print("Negative word: " + str(taipei_neg) + " word(s)")
print("Neutral word: " + str(taipei_neutral) + " word(s)")

print("\nHong Kong's article word count")
print("Positive word: " + str(hongkong_pos) + " word(s)")
print("Negative word: " + str(hongkong_neg) + " word(s)")
print("Neutral word: " + str(hongkong_neutral) + " word(s)")

print("\nTokyo's article word count")
print("Positive word: " + str(tokyo_pos) + " word(s)")
print("Negative word: " + str(tokyo_neg) + " word(s)")
print("Neutral word: " + str(tokyo_neutral) + " word(s)")

print("\nBeijing's article word count")
print("Positive word: " + str(beijing_pos) + " word(s)")
print("Negative word: " + str(beijing_neg) + " word(s)")
print("Neutral word: " + str(beijing_neutral) + " word(s)")

print("\nSeoul's article word count")
print("Positive word: " + str(seoul_pos) + " word(s)")
print("Negative word: " + str(seoul_neg) + " word(s)")
print("Neutral word: " + str(seoul_neutral) + " word(s)")

################
#    Graph     #
################

x = ["Jakarta", "Bangkok", "Taipei", "Hong Kong", "Tokyo", "Beijing", "Seoul"]
positive_y = [jakarta_pos, bangkok_pos, taipei_pos, hongkong_pos, tokyo_pos,
beijing_pos, seoul_pos]
negative_y = [jakarta_neg, bangkok_neg, taipei_neg, hongkong_neg, tokyo_neg,
beijing_neg, seoul_neg]
neutral_y = [jakarta_neutral, bangkok_neutral, taipei_neutral, hongkong_neutral,
tokyo_neutral,
             beijing_neutral, seoul_neutral]

data = [
    go.Histogram(
        histfunc="sum",
        y=positive_y,
        x=x,
        name="Positive words"
    ),
    go.Histogram(
        histfunc="sum",
        y=negative_y,
        x=x,
        name="Negative words"
```

```
    ),
    go.Histogram(
        histfunc="sum",
        y=neutral_y,
        x=x,
        name="Neutral words"
     )
]
layout = go.Layout(
    title=go.layout.Title(
        text="Positive, Negative & Neutral Words",
        xref='paper',
        x=0
    )
)
fig = go.Figure(data=data, layout=layout)
fig.show()
#  Sentiment & Conclusion  #
def sentiment(positive_frequency, negative_frequency, city):
    print("\n" + city.upper())
    if positive_frequency > negative_frequency:
        x = positive_frequency/negative_frequency
        print('The articles are giving positive sentiment')
        print('So the country has positive economic/financial situation of a ratio
')
        print(value(positive_frequency, negative_frequency))
    elif negative_frequency > positive_frequency:
        x = positive_frequency/negative_frequency
        print('The articles are giving negative sentiment')
        print('So the country has negative economic/financial situation of a ratio
')
        print(value(positive_frequency, negative_frequency))
    else:
        x = positive_frequency / negative_frequency
        print('The articles are giving neutral sentiment')
        print('So the country has neutral economic/financial situation of a ratio
')
        print(value(positive_frequency, negative_frequency))

def value(positive, negative):
    e = positive/negative
    return e

print("\n Concluding the cities' economic/fjnancial situation")
sentiment(jakarta_pos, jakarta_neg, "Jakarta")
sentiment(bangkok_pos, bangkok_neg, "Bangkok")
sentiment(taipei_pos, taipei_neg, "Taipei")
sentiment(hongkong_pos, hongkong_neg, "Hong Kong")
sentiment(tokyo_pos, tokyo_neg, "Tokyo")
sentiment(beijing_pos, beijing_neg, "Beijing")
sentiment(seoul_pos, seoul_neg, "Seoul")


jakartaValue= round(value(jakarta_pos, jakarta_neg),2)
bangkokValue= round(value(bangkok_pos, bangkok_neg),2)
taipeiValue= round(value(taipei_pos, taipei_neg),2)
hongkongValue= round(value(hongkong_pos, hongkong_neg),2)
tokyoValue= round(value(tokyo_pos, tokyo_neg),2)
beijingValue= round(value(beijing_pos, beijing_neg),2)
seoulValue= round(value(seoul_pos, seoul_neg),2)


data = [0, jakartaValue,
bangkokValue,taipeiValue,hongkongValue,tokyoValue,beijingValue,seoulValue]
print(data)
```

```
########################################################################## 3.10 Find The
Most Optimal Route Based on Sentiment
##########################################################################
city = [kl_coordinates, jakarta_coordinates, bangkok_coordinates, hk_coordinates,
taipei_coordinates, beijing_coordinates, seoul_coordinates, tokyo_coordinates]

optimal = ''
optimal_array = []

sentiment_arr = data[:]

#To Find the most optimal path as a baseline
def optimalPath():
    for i in range(1, 8):

        for j in range(i + 1, 8):

            if sentiment_arr[j] > sentiment_arr[i]:
                distance_ratio = (int(great_circle(city[j], city[i -
1]).kilometers) - int(
                    great_circle(city[i], city[i - 1]).kilometers)) /
int(great_circle(city[i], city[i - 1]).kilometers)

                if distance_ratio < 0.4:
                    sentiment_ratio = (sentiment_arr[j] - sentiment_arr[i]) /
sentiment_arr[i]

                    if sentiment_ratio > 0.02:

                        for k in range(i, j + 1):
                            temp = sentiment_arr[k]
                            sentiment_arr[k] = sentiment_arr[j]
                            sentiment_arr[j] = temp

optimalPath()
#Another MPM algorithm just to plat the route

for i in range(8):
    if sentiment_arr[i] == data[0]:
        optimal = optimal + "Kuala Lumpur > "
        optimal_array.append(0)
    if sentiment_arr[i] == data[1]:
        optimal = optimal + "Jakarta > "
        optimal_array.append(1)
    if sentiment_arr[i] == data[2]:
        optimal = optimal + "Bangkok > "
        optimal_array.append(2)
    if sentiment_arr[i] == data[3]:
        optimal = optimal + "Taipei > "
        optimal_array.append(3)
    if sentiment_arr[i] == data[4]:
        optimal = optimal + "Hong Kong > "
        optimal_array.append(4)
    if sentiment_arr[i] == data[5]:
        optimal = optimal + "Tokyo > "
        optimal_array.append(5)
    if sentiment_arr[i] == data[6]:
        optimal = optimal + "Beijing > "
        optimal_array.append(6)
    if sentiment_arr[i] == data[7]:
        optimal = optimal + "Seoul > "
        optimal_array.append(7)

print("\nMost optimal route based on distance and sentiment: \n",optimal)

count0 = 0
count0_array =[]
count1 = 0
```

```
count1_array =[]
count2 = 0
count2_array =[]
count3 = 0
count3_array =[]
count4 = 0
count4_array =[]
count5 = 0
count5_array =[]
count6 = 0
count6_array =[]
count7 = 0
count7_array =[]
count8 = 0
count8_array =[]

def arraySummary(summary):
    global optimal
    optimal = ''
    for i in range(8):
        if summary[i] == 0:
            optimal = optimal + "Kuala Lumpur > "
        if summary[i] == 1:
            optimal = optimal + "Jakarta > "
        if summary[i] == 2:
            optimal = optimal + "Bangkok > "
        if summary[i] == 3:
            optimal = optimal + "Taipei > "
        if summary[i] == 4:
            optimal = optimal + "Hong Kong > "
        if summary[i] == 5:
            optimal = optimal + "Tokyo > "
        if summary[i] == 6:
            optimal = optimal + "Beijing > "
        if summary[i] == 7:
            optimal = optimal + "Seoul > "
    print(optimal)

# Another MPM algorithm
for i in range(len(pathstring)):
    match_count = 0
    summary = pathstring[i]
    summary_distance = unsorted_distance_array[i]
    for j in range(8):
        if summary[j] == optimal_array[j]:
            match_count += 1
    if match_count == 0:
        count0 +=1
        count0_array.append(pathstring[i])
    if match_count == 1:
        count1 +=1
        count1_array.append(pathstring[i])
    if match_count == 2:
        count2 +=1
        count2_array.append(pathstring[i])
    if match_count == 3:
        count3 +=1
        count3_array.append(pathstring[i])
    if match_count == 4:
        count4 +=1
        count4_array.append(pathstring[i])
    if match_count == 5:
        count5 +=1
        count5_array.append(pathstring[i])
    if match_count == 6:
        count6 +=1
        count6_array.append(pathstring[i])
    if match_count == 7:
```

```
        count7 +=1
        count0_array.append(pathstring[i])
    if match_count ==8:
        count8 +=1
        count8_array.append(pathstring[i])

totalcount = count0+count1+count2+count3+count4+count5+count6+count7+count8
count0hist = count0/totalcount
count1hist = count1/totalcount
count2hist = count2/totalcount
count3hist = count3/totalcount
count4hist = count4/totalcount
count5hist = count5/totalcount
count6hist = count6/totalcount
count7hist = count7/totalcount
count8hist = count8/totalcount

Optimal_Route_Match = [count0hist, count1hist, count2hist, count3hist, count4hist,
count5hist, count6hist,count7hist,count8hist]

x = ["0","1","2","3","4","5","6","7","8"]

data = [
    go.Bar(

        y=Optimal_Route_Match ,
        x=x,

    ),
]
layout = go.Layout(
    title=go.layout.Title(
        text="Probability Function",
        xref='paper',
        x=0
    )
)
fig = go.Figure(data=data, layout=layout)
fig.show()

print("\n##############################################################################
")
print("In order of LEAST RECOMMENDED to MOST RECOMMENDED: ")
print("##############################################################################\n
")

print("------------------------------------------------------------------------------")
print("Routes with 0 matches to optimal route NON EXISTENT")
print("------------------------------------------------------------------------------")
print()
print("------------------------------------------------------------------------------")
print("Routes with 1 matches to optimal route ( ABSOLUTELY NOT RECOMMENDED ) :")
print("------------------------------------------------------------------------------")
for i in range(len(count1_array)):
    arraySummary(count1_array[i])
print()
print("------------------------------------------------------------------------------")
print("Routes with 2 matches to optimal route ( NOT RECOMMENDED ) :")
print("------------------------------------------------------------------------------")
for i in range(len(count2_array)):
    arraySummary(count2_array[i])
print()
print("------------------------------------------------------------------------------")
print("Routes with 3 matches to optimal route ( NOT RECOMMENDED ) :")
print("------------------------------------------------------------------------------")
for i in range(len(count3_array)):
    arraySummary(count3_array[i])
print()
```

```
print("-------------------------------------------------------------------------------")
print("Routes with 4 matches to optimal route ( AVERAGE ) :")
print("-------------------------------------------------------------------------------")
for i in range(len(count4_array)):
    arraySummary(count4_array[i])
print()
print("-------------------------------------------------------------------------------")
print("Routes with 5 matches to optimal route ( RECOMMENDED ) :")
print("-------------------------------------------------------------------------------")
for i in range(len(count5_array)):
    arraySummary(count5_array[i])
print()
print("-------------------------------------------------------------------------------")
print("Routes with 6 matches to optimal route ( HIGHLY RECOMMENDED ) :")
print("-------------------------------------------------------------------------------")
for i in range(len(count6_array)):
    arraySummary(count6_array[i])
print()
print("-------------------------------------------------------------------------------")
print("Routes with 7 matches with optimal route NON EXISTENT")
print("-------------------------------------------------------------------------------")
print()
print("-------------------------------------------------------------------------------")
print("Routes with COMPLETE MATCH with optimal route ( PERFECT MATCH - MOST
OPTIMAL! ) :")
print("-------------------------------------------------------------------------------")
for i in range(len(count8_array)):
    arraySummary(count8_array[i])
print()
print("###############################################################################
#")
print("Most optimal route based on distance and sentiment: ")
print()
print(optimal)
print("###############################################################################
#")


stop = timeit.default_timer()
print("\nTotal Runtime: ", stop-start)
```

Appendix B - FILA Form 2

FILA FORM 2 – University of Malaya

FACTS	IDEAS	LEARNING ISSUES	ACTION	DATELINE
1.1 Get and mark locations of all the cities required in the question	-Utilize a geocoding API to get the coordinates of all the cities needed.		**Aiman** and **Jeremiah** coded the segments to utilize the geopy Google geocoding API where Nominatim and great_circle were imported from the library.	05/04/20
1.2 Get the distances between the 8 cities in the question (Kuala Lumpur, Jakarta, Bangkok, Taipei, Hong Kong, Tokyo, Beijing, and Seoul).	-Use a function from an imported library to get the distances between two coordinates.		**Jeremiah** created a network graph using the Pyvis import. The edge weights (travel distances) are obtained by using the great_circle function from the Geopy library.	10/04/20
1.3 Suggest a journey for Ben to visit each city once with the least distance travelled.	-Use an algorithm to solve this NP-complete minimum shortest Hamilton path problem. (note: this is not a TSP problem) -Calculate the distance and find the shortest distance travelled of all the Hamiltonian paths		**Jeremiah** implemented a backtracking algorithm to find all the possible Hamiltonian paths in the graph model of the question. After calculating the distances of all Hamiltonian paths, **Afia** implemented a Timsort algorithm to sort the paths in ascending order of distances	10/04/20

1.4 Plot a polyline for the suggested journey	-Utilize an API to plot a polyline between the suggested coordinates	**Jeremiah** used the gmplot.plot function to plot the suggested shortest journey plan on Google Maps.	20/04/20
2.5 Extract 5 information from major economics/financial news website for 7 city and analyse those article	-Using textise.net to convert the html to text for the article -create a function to calculate the and print the frequency for each word in articles -implement Rabin Karp Algorithm to create a function to compare between stop words and articles	**Izwan** find 5 articles from 7 cities provided. **Izwan** convert the html file into text file **Izwan** create a function to print the frequency of each word in article, and filter the stop words using Rabin Karp Algorithm	24/06/20
2.6 Plot a histogram using Plotly	By using data from previous part, utilize Plotly to plot a histogram of number of stop word and total number of words	**Izwan** use Plotly.py to plot a histogram to show number of word count and stop word	24/06/20
2.7 Compare words in the webpages with the positive, negative and neutral English words using a String-Matching algorithm	Implement Rabin Karp Algorithm to compare word in webpages with the positive, negative and neutral English words	**Ariq** implement Rabin Karp Algorithm to compare word in webpages with the positive, negative and neutral English words	24/06/20
2.8 Plot histogram graphs of positive and negative words found in the webpages.	By using data from previous part, utilize Plotly to plot a histogram of number of words in webpages with the positive, negative and neutral English word	**Ariq** use Plotly.py to plot histogram to show word in webpages with the positive, negative and neutral English words	24/06/20

2.9 Give an algorithmic conclusion regarding the sentiment of article from each city	-Create a function to identify the sentiment conclusion for each city -create a function to calculate the sentiment ratio value for each city	**Ariq** created both functions to identify the sentiment conclusion and sentiment ratio value of each city	24/06/20
3.10 Calculate the total probability distribution of possible routes along with the summary for the routes. Rank the routes from most recommended to least recommended	-Implement an algorithm to swap the positions of an array after comparing the sentiment ratios between cities where the indexed elements are all sorted based on the shortest distance. Calculate the probability distribution of all routes in comparison with the most optimal route	**Aiman, Afia,** and **Jeremiah** implemented and adapted the swapping techniques from insertion sort to find the most optimal path based on distance specifications and sentiment ratio of positive to negative sentiments. **Afia** and **Aiman** implemented an algorithm to group the paths into ascending order of suitability to be recommended based on the	26/05/20

		matches to the most optimal path **Aiman** calculated the probability distribution based on how closely the other paths matched the optimal path	
General: Sorting algorithms will be needed for most parts	-Implement different sorting algorithms to handle different cases in *Problem 1, Problem 2,* and *Problem 3*	**Afia** implemented and adapted most of the sorting algorithms needed.	
General: Ensure the code runs smoothly, is integrated which each member's parts and does not encounter any runtime errors	-Debug and test each part of the program separately before		