ANÁLISIS DEL RETO

1. YEFRAN CESPEDES, Y.CESPEDES@UNIANDES.EDU.CO, 202316255.

2. Jose Carvajal, jd.carvajalg1@uniandes.edu.co, 202317192.

3. Luis Vega, ld.vegam1@uniandes.edu.co, 202311012.

Requerimiento <<0>> CARGA DE DATOS

```
data_structs["airports_map"] = mp.newMap(maptype="PROBING")

data_structs["carga_airports_distance"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=False)

data_structs["carga_airports_time"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=False)

data_structs["militar_airports_distance"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=False)

data_structs["militar_airports_time"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=False)

data_structs["comercial_airports_distance"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=False)

data_structs["comercial_airports_time"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=False)

data_structs["carga_airports_distance_directed"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=True)

data_structs["carga_airports_time_directed"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=True)

data_structs["militar_airports_distance_directed"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=True)

data_structs["militar_airports_time_directed"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=True)

data_structs["comercial_airports_distance_directed"] = gr.newGraph(datastructure="ADJ_LIST",
                                            directed=True)
```

```
data_structs["comercial_airports_time_directed"] = gr.newGraph(datastructure="ADJ_LIST",
                                                               directed=True)

data_structs["airports_comercial_map"] = om.newMap()


data_structs["airports_militar_map"] = om.newMap()


data_structs["airports_carga_map"] = om.newMap()

return data_structs
```

Descripción

Para la carga de datos se implementaron 12 grafos de estos se dividen en dos tipos 6 dirigidos y otros 6 no dirigidos, por cada 6 de estos se dividen en 2, grafos donde sus pesos son el tiempo de trayecto y los otros donde sus pesos son la distancia recorrida, por ultimo se clasifican por tipo de vuelo, estando 3, comerciales, militares y de carga. Luego también se implementa un mapa donde su llave es el código ICAO de cada aeropuerto y el valor es el diccionario con la información de cada uno de estos. Finalmente se crean tres diccionarios ordenados y clasificados por el tipo de vuelo donde su llave es el número de concurrencia y el valor es una lista con los códigos ICAO de cada aeropuerto.

Breve descripción de como abordaron la implementación del requerimiento

Entrada	El modelo de la estructura de datos.
Salidas	La estructura de datos con sus datos correspondientes
Implementado (Sí/No)	Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Cargar los aeropuertos 	O(n) donde n es el número de aeropuertos en el archivo
Cargar los vuelos	O(m) donde m es el número de vuelos en el archivo

```
def add_data_airports(data_struct,data):
    """
    Agrega cada aeropuerto a un mapa donde la llave es el codigo ICAO y el valor es el diccionario con la información
    """
    airport_code = data["ICAO"]

    map_airport = data_struct["airports_map"]

    mp.put(map_airport,airport_code,data)

    gr.insertVertex(data_struct["carga_airports_distance_directed"],airport_code)
    gr.insertVertex(data_struct["carga_airports_time_directed"],airport_code)
    gr.insertVertex(data_struct["militar_airports_distance_directed"],airport_code)
    gr.insertVertex(data_struct["militar_airports_time_directed"],airport_code)
    gr.insertVertex(data_struct["comercial_airports_distance_directed"],airport_code)
    gr.insertVertex(data_struct["comercial_airports_time_directed"],airport_code)
    gr.insertVertex(data_struct["carga_airports_distance"],airport_code)
    gr.insertVertex(data_struct["carga_airports_time"],airport_code)
    gr.insertVertex(data_struct["militar_airports_distance"],airport_code)
    gr.insertVertex(data_struct["militar_airports_time"],airport_code)
    gr.insertVertex(data_struct["comercial_airports_distance"],airport_code)
    gr.insertVertex(data_struct["comercial_airports_time"],airport_code)
```

```
def add_data_fligths(data_structs, data):
    """
    Función para agregar nuevos elementos a la lista
    """
    type_flight = data["TIPO_VUELO"]

    if type_flight == "AVIACION_CARGA":

        origen = data["ORIGEN"]
        destino = data["DESTINO"]
        entry1 = mp.get(data_structs["airports_map"],origen)
        data1 = me.getValue(entry1)
        entry2 = mp.get(data_structs["airports_map"],destino)
        data2 = me.getValue(entry2)
        distance = haversine_data(data1,data2)
        time = float(data["TIEMPO_VUELO"])
        aeronave = data["TIPO_AERONAVE"]
        gr.addEdge(data_structs["carga_airports_distance_directed"],origen,destino,distance)
        gr.addEdge(data_structs["carga_airports_time_directed"],origen,destino,time)
        gr.addEdge(data_structs["carga_airports_distance"],origen,destino,distance)
        gr.addEdge(data_structs["carga_airports_time"],origen,destino,(time,aeronave))


    elif type_flight == "MILITAR":

        origen = data["ORIGEN"]
        destino = data["DESTINO"]
        entry1 = mp.get(data_structs["airports_map"],origen)
        data1 = me.getValue(entry1)
        entry2 = mp.get(data_structs["airports_map"],destino)
        data2 = me.getValue(entry2)
        distance = haversine_data(data1,data2)
        time = float(data["TIEMPO_VUELO"])
        aeronave= data["TIPO_AERONAVE"]
        gr.addEdge(data_structs["militar_airports_distance_directed"],origen,destino,distance)
        gr.addEdge(data_structs["militar_airports_time_directed"],origen,destino,time)
        gr.addEdge(data_structs["militar_airports_distance"],origen,destino,(distance,aeronave))
        gr.addEdge(data_structs["militar_airports_time"],origen,destino,time)

    elif type_flight== "AVIACION_COMERCIAL":

        origen = data["ORIGEN"]
        destino = data["DESTINO"]
        entry1 = mp.get(data_structs["airports_map"],origen)
        data1 = me.getValue(entry1)
        entry2 = mp.get(data_structs["airports_map"],destino)
        data2 = me.getValue(entry2)
        distance = haversine_data(data1,data2)
        time = float(data["TIEMPO_VUELO"])
        aeronave= data["TIPO_AERONAVE"]
        gr.addEdge(data_structs["comercial_airports_distance_directed"],origen,destino,distance)
        gr.addEdge(data_structs["comercial_airports_time_directed"],origen,destino,time)
        gr.addEdge(data_structs["comercial_airports_distance"],origen,destino,distance)
        gr.addEdge(data_structs["comercial_airports_time"],origen,destino,(time,aeronave))
```

Carga de datos a los mapas ordenados de concurrencia	O(m) donde m es el número de vuelos en el archivo

```
def add_concurrence(data_structs,data):

    type_fligth = data["TIPO_VUELO"]
    origen = data["ORIGEN"]
    destino = data["DESTINO"]

    if type_fligth== "AVIACION_COMERCIAL":

        mapa = data_structs["airports_comercial_map"]
        grafo = data_structs["comercial_airports_distance_directed"]
        indegree_origen = gr.indegree(grafo,origen)
        outgree_origen = gr.outdegree(grafo,origen)
        concurrence_origen= indegree_origen+outgree_origen

        indegree_destino = gr.indegree(grafo,destino)
        outgree_destino =  gr.outdegree(grafo,destino)
        concurrence_destino = indegree_destino+ outgree_destino

        exist_concurrence_origen = om.contains(mapa,concurrence_origen)
        exist_concurrence_destino =om.contains(mapa,concurrence_destino)

        if exist_concurrence_origen:
            entry = om.get(mapa,concurrence_origen)
            mapa_airports = me.getValue(entry)
            exist_origen = mp.contains(mapa_airports,origen)
            if exist_origen:
                pass
            else:
                mp.put(mapa_airports,origen,0)
        else:
            info_concurrence = mp.newMap(maptype="PROBING",loadfactor=1)
            om.put(mapa,concurrence_origen,info_concurrence)

        if exist_concurrence_destino:
            entry = om.get(mapa,concurrence_destino)
            mapa_airports = me.getValue(entry)
            exist_destino = mp.contains(mapa_airports,destino)
            if exist_destino:
                pass
            else:
                mp.put(mapa_airports,destino,0)
        else:
            info_concurrence = mp.newMap(maptype="PROBING")
            om.put(mapa,concurrence_destino,info_concurrence)
```

```
if type_fligth == "AVIACION_CARGA":

    grafo = data_structs["carga_airports_distance_directed"]
    mapa = data_structs["airports_carga_map"]
    indegree_origen = gr.indegree(grafo,origen)
    outgree_origen = gr.outdegree(grafo,origen)
    concurrence_origen= indegree_origen+outgree_origen

    indegree_destino = gr.indegree(grafo,destino)
    outgree_destino =  gr.outdegree(grafo,destino)
    concurrence_destino = indegree_destino+ outgree_destino


    exist_concurrence_origen = om.contains(mapa,concurrence_origen)
    exist_concurrence_destino =om.contains(mapa,concurrence_destino)

    if exist_concurrence_origen:
        entry = om.get(mapa,concurrence_origen)
        mapa_airports = me.getValue(entry)
        exist_origen = mp.contains(mapa_airports,origen)
        if exist_origen:
            pass
        else:
            mp.put(mapa_airports,origen,0)
    else:
        info_concurrence = mp.newMap(maptype="PROBING")
        om.put(mapa,concurrence_origen,info_concurrence)

    if exist_concurrence_destino:
        entry = om.get(mapa,concurrence_destino)
        mapa_airports = me.getValue(entry)
        exist_destino = mp.contains(mapa_airports,destino)
        if exist_destino:
            pass
        else:
            mp.put(mapa_airports,destino,0)
    else:
        info_concurrence = mp.newMap(maptype="PROBING")
        om.put(mapa,concurrence_destino,info_concurrence)


if type_fligth == "MILITAR":

    grafo = data_structs["militar_airports_distance_directed"]
    mapa = data_structs["airports_militar_map"]

    indegree_origen = gr.indegree(grafo,origen)
    outgree_origen = gr.outdegree(grafo,origen)
    concurrence_origen= indegree_origen+outgree_origen

    indegree_destino = gr.indegree(grafo,destino)
    outgree_destino =  gr.outdegree(grafo,destino)
    concurrence_destino = indegree_destino+ outgree_destino

    exist_concurrence_origen = om.contains(mapa,concurrence_origen)
    exist_concurrence_destino =om.contains(mapa,concurrence_destino)

    if exist_concurrence_origen:
        entry = om.get(mapa,concurrence_origen)
        mapa_airports = me.getValue(entry)
        exist_origen = mp.contains(mapa_airports,origen)
        if exist_origen:
            pass
        else:
            mp.put(mapa_airports,origen,0)
    else:
        info_concurrence = mp.newMap(maptype="PROBING")
        om.put(mapa,concurrence_origen,info_concurrence)

    if exist_concurrence_destino:
        entry = om.get(mapa,concurrence_destino)
        mapa_airports = me.getValue(entry)
        exist_destino = mp.contains(mapa_airports,destino)
        if exist_destino:
            pass
        else:
            mp.put(mapa_airports,destino,0)
    else:
        info_concurrence = mp.newMap(maptype="PROBING")
        om.put(mapa,concurrence_destino,info_concurrence)
```

Comparaciones, insertar, obtener valores, obtener grados, crear estructuras de datos, agregar arcos	O(1)
TOTAL	***O(m+n) donde m es el número de vuelos y n es el número de aeropuertos.***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	**Tiempo (s)**
None	494.961000001058
None	415.16049999929965
None	447.76410000026226

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entrada	**Tiempo (s)**
None	494.961000001058
None	415.16049999929965
None	447.76410000026226

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el análisis de complejidad.

Estas estructuras se implementaron con el fin de hacer los requerimientos y de poder hacerlos en una complejidad bastante acorde a las exigencias, se duplica el número de grafos al necesitar para los requerimientos individuales el uso del algoritmo Prim-MST que solo funciona correctamente en grafos no dirigidos, pero los demás requerimientos se necesita hacer uso de grafos dirigidos ya que nada asegura que si un aeropuerto A tiene vuelo a un aeropuerto B, el aeropuerto B tenga vuelo al aeropuerto A, entonces al realizar la carga con grafos dirigidos aseguramos datos verídicos, ahora para obtener aeropuertos según la concurrencia se hace uso de mapas ordenados ya que su complejidad de acceder a estos datos es constante, y al estar ordenados según la concurrencia permite obtener los de mayores complejidades y menores. Por último, un mapa con la información de todos los aeropuertos es ideal para poder acceder en cualquier momento a la información de un aeropuerto en cualquier requerimiento.

Requerimiento <<1>>

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_1(data_structs,origen_latitud,origen_longitud,destino_latitud,destino_longitud,method):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    distance_origen,airport_origen,distance_destino,airport_destino = get_nearby_airports(data_structs,(origen_latitud,origen_longitud),(destino_latitud,destino_longitud))

    entry = mp.get(data_structs["airports_map"],airport_origen)
    data_origen = me.getValue(entry)
    entry = mp.get(data_structs["airports_map"],airport_destino)
    data_destino = me.getValue(entry)

    respuesta = lt.newList()
    if distance_origen<=30 and distance_destino <= 30:
        tiempo_total =0
        distancia_total = distance_origen+distance_destino
        if method == 1 :

            data_structs["search"] = bfs.BreathFirstSearch(data_structs["comercial_airports_distance_directed"],airport_origen)
            exist_camino = bfs.hasPathTo(data_structs["search"],airport_destino)
            if exist_camino:

                ruta = bfs.pathTo(data_structs["search"],airport_destino)
                total_airports = st.size(ruta)
                for x in range(0,total_airports):

                    airport=st.pop(ruta)
                    entry = mp.get(data_structs["airports_map"],airport)
                    data_airport= me.getValue(entry)
                    if st.size(ruta)>0:

                        siguiente = st.top(ruta)
                        arco_distance = gr.getEdge(data_structs["comercial_airports_distance_directed"],airport,siguiente)
                        distance = float(arco_distance["weight"])
                        distancia_total += distance
                        arco_time =gr.getEdge(data_structs["comercial_airports_time_directed"],airport,siguiente)
                        time = float(arco_time["weight"])
                        tiempo_total += time
                    lt.addLast(respuesta,data_airport)
```

```python
        if method == 2:

            data_structs["search"] = dfs.DepthFirstSearch(data_structs["comercial_airports_distance_directed"],airport_origen)
            exist_camino = dfs.hasPathTo(data_structs["search"],airport_destino)

            if exist_camino:

                ruta = dfs.pathTo(data_structs["search"],airport_destino)
                total_airports = st.size(ruta)

                for x in range(0,total_airports):

                    airport=st.pop(ruta)
                    entry = mp.get(data_structs["airports_map"],airport)
                    data_airport= me.getValue(entry)
                    if st.size(ruta)>0:

                        siguiente = st.top(ruta)
                        arco_distance = gr.getEdge(data_structs["comercial_airports_distance_directed"],airport,siguiente)
                        distance = float(arco_distance["weight"])
                        distancia_total += distance
                        arco_time =gr.getEdge(data_structs["comercial_airports_time_directed"],airport,siguiente)
                        time = float(arco_time["weight"])
                        tiempo_total += time

                    lt.addLast(respuesta,data_airport)

        return distancia_total,total_airports,respuesta,tiempo_total
```

```python
    else:

        print("El aeropuerto más cercano al origen es: ",airport_origen, " con una distancia a tu ubicacion de :",distance_origen, " donde su latitud es: ",data_origen["LATITUD"]," y su longitud es: ",data_origen["LONGITUD"])
        print("El aeropuerto más cercano al destino es: ",airport_destino, " con una distancia a tu ubicacion de :",distance_destino," donde su latitud es: ",data_destino["LATITUD"]," y su longitud es: ",data_destino["LONGITUD"])

        return None
```

Descripción

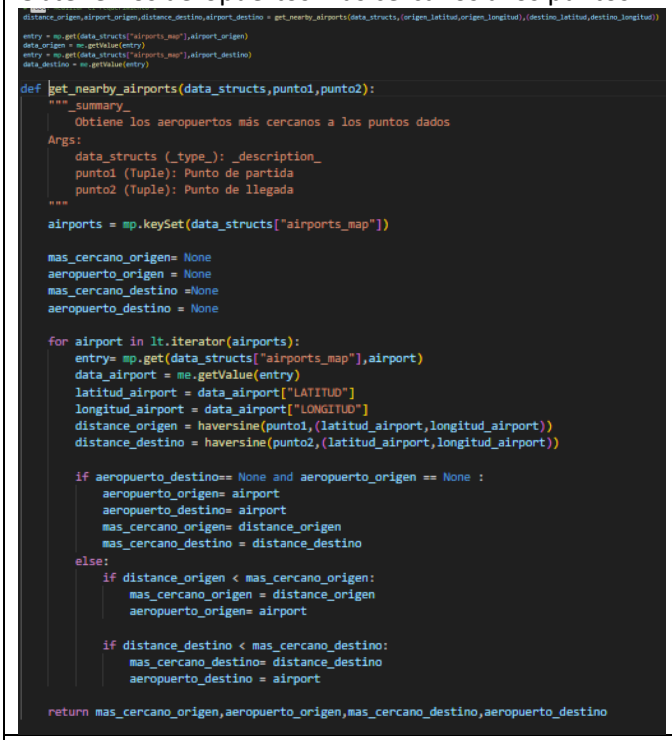
Para este requerimiento se implemento como adicional que el usuario pudiese elegir entre dos algoritmos para calcular la ruta entre el punto de origen al punto de destino, los algoritmos son dfs y bfs, entonces primero se calcula el dfs o bfs sobre el algoritmo de inicio y luego se pregunta si existe ruta para el punto de destino, si la hay se calcula esta ruta, esto devuelve una pila con arcos del trayecto para los cuales se comienza a calcular los pesos obteniéndolos de los grafo que se creo para aeropuertos

comerciales donde su peso es la distancia y se van almacenando en una lista donde se retorna para ser impreso en forma de tabla.

Entrada	Longitud y latitud tanto del punto de origen como del punto de destino
Salidas	Si hay una ruta, se retorna la ruta de origen a la de destino sino se notifica que aeropuerto hay más cercano a los puntos o que no hay ruta
Implementado (Sí/No)	Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Obtener los aeropuertos más cercanos a los puntos 	O(n) donde n es el número de aeropuertos a recorrer
Algoritmo bfs y dfs 	O(V+E) donde V es el número de vértices y E el número de arcos en el grafo

```
distance_origen,airport_origen,distance_destino,airport_destino = get_nearby_airports(data_structs,(origen_latitud,origen_longitud),(destino_latitud,destino_longitud))

entry = mp.get(data_structs["airports_map"],airport_origen)
data_origen = me.getValue(entry)
entry = mp.get(data_structs["airports_map"],airport_destino)
data_destino = me.getValue(entry)

def get_nearby_airports(data_structs,punto1,punto2):
    """_summary_
        Obtiene los aeropuertos más cercanos a los puntos dados
    Args:
        data_structs (_type_): _description_
        punto1 (Tuple): Punto de partida
        punto2 (Tuple): Punto de llegada
    """
    airports = mp.keySet(data_structs["airports_map"])

    mas_cercano_origen= None
    aeropuerto_origen = None
    mas_cercano_destino =None
    aeropuerto_destino = None

    for airport in lt.iterator(airports):
        entry= mp.get(data_structs["airports_map"],airport)
        data_airport = me.getValue(entry)
        latitud_airport = data_airport["LATITUD"]
        longitud_airport = data_airport["LONGITUD"]
        distance_origen = haversine(punto1,(latitud_airport,longitud_airport))
        distance_destino = haversine(punto2,(latitud_airport,longitud_airport))

        if aeropuerto_destino== None and aeropuerto_origen == None :
            aeropuerto_origen= airport
            aeropuerto_destino= airport
            mas_cercano_origen= distance_origen
            mas_cercano_destino = distance_destino
        else:
            if distance_origen < mas_cercano_origen:
                mas_cercano_origen = distance_origen
                aeropuerto_origen= airport

            if distance_destino < mas_cercano_destino:
                mas_cercano_destino= distance_destino
                aeropuerto_destino = airport

    return mas_cercano_origen,aeropuerto_origen,mas_cercano_destino,aeropuerto_destino
```

```
if distance_origen<=30 and distance_destino <= 30:
    tiempo_total =0
    distancia_total = distance_origen+distance_destino
    if method == 1 :

        data_structs["search"] = bfs.BreathFirstSearch(data_structs["comercial_airports_distance_directed"],airport_origen)
        exist_camino = bfs.hasPathTo(data_structs["search"],airport_destino)
        if exist_camino:

            ruta = bfs.pathTo(data_structs["search"],airport_destino)
            total_airports = st.size(ruta)
            for x in range(0,total_airports):

                airport=st.pop(ruta)
                entry = mp.get(data_structs["airports_map"],airport)
                data_airport= me.getValue(entry)
                if st.size(ruta)>0:

                    siguiente = st.top(ruta)
                    arco_distance = gr.getEdge(data_structs["comercial_airports_distance_directed"],airport,siguiente)
                    distance = float(arco_distance["weight"])
                    distancia_total += distance
                    arco_time =gr.getEdge(data_structs["comercial_airports_time_directed"],airport,siguiente)
                    time = float(arco_time["weight"])
                    tiempo_total += time
                lt.addLast(respuesta,data_airport)
```

	
Agregar los aeropuertos a la lista de respuesta 	O(k) donde k es el número de aeropuertos en el trayecto desde el punto de origen y el punto de destino
Comparaciones, insertar, obtener valores, obtener grados, crear estructuras de datos, agregar arcos	O(1)
TOTAL	***O(k+E+N)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	**Tiempo (s)**
4.601992771389502, -74.0661047044192,-10.507688799813222,- -75.4706488665794	1138.0

Análisis

El requerimiento es bastante rápido y cumple con las expectativas de su implementación ya que como tal en el caso promedio se comportará de manera lineal donde el causante de esta complejidad es el algoritmo que el usuario elija.

Requerimiento <<2>>

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_2(data_structs,origen_latitud,origen_longitud,destino_latitud,destino_longitud):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    distance_origen,airport_origen,distance_destino,airport_destino = get_nearby_airports(data_structs,(origen_latitud,origen_longitud),(destino_latitud,destino_longitud))

    entry = mp.get(data_structs["airports_map"],airport_origen)
    data_origen = me.getValue(entry)
    entry = mp.get(data_structs["airports_map"],airport_destino)
    data_destino = me.getValue(entry)

    respuesta = lt.newList()
    if distance_origen<=30 and distance_destino <= 30:
        tiempo_total =0
        distancia_total = distance_origen+distance_destino
        data_structs["search"] = bfs.BreathFirstSearch(data_structs["comercial_airports_distance_directed"],airport_origen)
        exist_camino = bfs.hasPathTo(data_structs["search"],airport_destino)
        if exist_camino:

                ruta = bfs.pathTo(data_structs["search"],airport_destino)
                total_airports = st.size(ruta)
                for x in range(0,total_airports):

                    airport=st.pop(ruta)
                    entry = mp.get(data_structs["airports_map"],airport)
                    data_airport= me.getValue(entry)
                    if st.size(ruta)>0:

                        siguiente = st.top(ruta)
                        arco_distance = gr.getEdge(data_structs["comercial_airports_distance_directed"],airport,siguiente)
                        distance = float(arco_distance["weight"])
                        distancia_total += distance
                        arco_time =gr.getEdge(data_structs["comercial_airports_time_directed"],airport,siguiente)
                        time = float(arco_time["weight"])
                        tiempo_total += time
                    lt.addLast(respuesta,data_airport)
```

Descripción

Para este algoritmo primero se calculan los aeropuertos más cercanos a los puntos dados por el usuario, luego si los hay, entonces se aplica el algoritmo Bfs y sobre este se trabaja con el fin de retornar la ruta que el usuario pidió entre el punto de origen al punto de salida

Entrada	Longitud y latitud tanto del punto de origen como el punto de destino
Salidas	Ruta entre el punto de origen al punto de destino, distancia y tiempo total del trayecto y el número total de aeropuertos
Implementado (Sí/No)	Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Obtener los aeropuertos más cercanos `distance_origen,airport_origen,distance_destino,airport_destino = get_nearby_airports(data_structs,(origen_latitud,origen_longitud),(destino_latitud,destino_longitud))`	O(n) donde n es el número de aeropuertos

Algoritmo BFS
`respuesta = lt.newList()`
`if distance_origen<=30 and distance_destino <= 30:`
`  tiempo_total =0`
`  distancia_total = distancia_origen+distancia_destino`
`  data_structs["search"] = bfs.BreathFirstSearch(data_structs["comercial_airports_distance_directed"],airport_origen)`
`  exist_camino = bfs.hasPathTo(data_structs["search"],airport_destino)`
	O(V+E) donde V es el número de vértices y E el número de arcos

Calcular sobre la ruta encontrada
`for x in range(0,total_airports):`
`  airport=st.pop(ruta)`
`  entry = mp.get(data_structs["airports_map"],airport)`
`  data_airport= me.getValue(entry)`
`  if st.size(ruta)>0:`
`  siguiente = st.top(ruta)`
`  arco_distance = gr.getEdge(data_structs["comercial_airports_distance_directed"],airport,siguiente)`
`  distance = float(arco_distance["weight"])`
`  distancia_total += distance`
`  arco_time =gr.getEdge(data_structs["comercial_airports_time_directed"],airport,siguiente)`
`  time = float(arco_time["weight"])`
`  tiempo_total += time`
`  lt.addLast(respuesta,data_airport)`
	O(k) donde k es la cantidad de aeropuertos en la ruta encontrada

TOTAL	***O(n+K+E)***

Análisis

Se aplica el algoritmo BFS con el fin de conseguir la ruta con menores escalas sobre el grafo de vuelos comerciales entre un punto de destino y uno de llegada, la complejidad del requerimiento es bastante apropiada y cumple con las expectativas de implementación ya que su complejidad al final predomina cobre la complejidad que toma el uso de el algoritmo BFS y el obtener los aeropuertos más cercanos al punto, sobre este último se hizo una investigación y se encontró una forma de obtenerlo en una complejidad mejor, y es implementando dos mapas ordenados donde sus llaves de ordenamiento sean la latitud y longitud de los aeropuertos sobre estos se puede calcular el Haversive inverso y obtener los aeropuertos del punto de origen y destino en un mejor tiempo, sin embargo por problemas de tiempo ya que se estaba finalizando el semestre, el resultado de investigación se quedo en la teoría.

Requerimiento <<3>>

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_3(data_structs):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    data_aeropuerto_mayor = mayor_concurrencia(data_structs,data_structs["airports_comercial_map"],1)
    name_aeropuerto_mayor = lt.getElement(data_aeropuerto_mayor,0)["ICAO"]

    search = prim.PrimMST(data_structs["comercial_airports_distance"],name_aeropuerto_mayor)

    distancia_total = prim.weightMST(data_structs["comercial_airports_distance"],search)

    total_trayectos = lt.size(search["mst"])

    respuesta = lt.newList(datastructure="ARRAY_LIST")

    for ruta in lt.iterator(search["mst"]):

        origen = e.either(ruta)
        entry = mp.get(data_structs["airports_map"],origen)
        data_origen = me.getValue(entry)
        destino = e.other(ruta,origen)
        entry = mp.get(data_structs["airports_map"],destino)
        data_destino = me.getValue(entry)
        distancia = e.weight(ruta)
        tiempo,aeronave = e.weight(gr.getEdge(data_structs["comercial_airports_time"],origen,destino))

        data = new_data(data_origen,data_destino,distancia,tiempo,aeronave)

        lt.addLast(respuesta,data)

    return respuesta,data_aeropuerto_mayor,total_trayectos,distancia_total
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Para este requerimiento primero se obtiene el aeropuerto de mayor concurrencia comercial y sobre este se calcula el MST-Prim , luego se obtiene el peso del mst el cual se pide en el requerimiento y luego se itera sobre el mst para ir agregando lo que se pide entre los aeropuertos de origen y destino.

Entrada	None.
Salidas	Lista con los datos a imprimir, el dato del aeropuerto con mayor concurrencia comercial, el total de trayectos que en teoría es el número de arcos más 1, y la distancia total del mst
Implementado (Sí/No)	José Carvajal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
`data_aeropuerto_mayor = mayor_concurrencia(data_structs,data_structs["airports_comercial_map"],1)` `name_aeropuerto_mayor = lt.getElement(data_aeropuerto_mayor,0)["ICAO"]`	O(1)

`search = prim.PrimMST(data_structs["comercial_airports_distance"],name_aeropuerto_mayor)`
 	
`distancia_total = prim.weightMST(data_structs["comercial_airports_distance"],search)`
 	
`total_trayectos = lt.size(search["mst"])`
	O(ElogV) donde E es el número de arcos y V el número de vértices

`for ruta in lt.iterator(search["mst"]):`
 	
`origen = e.either(ruta)`
`entry = mp.get(data_structs["airports_map"],origen)`
`data_origen = me.getValue(entry)`
`destino = e.other(ruta,origen)`
`entry = mp.get(data_structs["airports_map"],destino)`
`data_destino = me.getValue(entry)`
`distancia = e.weight(ruta)`
`tiempo,aeronave = e.weight(gr.getEdge(data_structs["comercial_airports_time"],origen,destino))`
 	
`data = new_data(data_origen,data_destino,distancia,tiempo,aeronave)`
 	
`lt.addLast(respuesta,data)`
 	
`return respuesta,data_aeropuerto_mayor,total_trayectos,distancia_total`
	O(n) donde n es el número de arcos del MST

TOTAL	***O(n+ElogV)***

Análisis

Para este requerimiento se hizo uso del algoritmo MST el cual asegura el grafo de peso mínimo en grafos no dirigidos por ende se hizo necesario agregarle a la estructura de datos la creación de un grafo no dirigido de tipo comercial con peso distancia y otro para peso tiempo, luego al calcular el MST se hace la iteración sobre los arcos que retorna para poder agregar y calcular la información que se pide.

Requerimiento <<4>>

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_4(data_structs):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    data_aeropuerto_mayor = mayor_concurrencia(data_structs,data_structs["airports_carga_map"],1)
    name_aeropuerto_mayor = lt.getElement(data_aeropuerto_mayor,0)["ICAO"]

    search = prim.PrimMST(data_structs["carga_airports_distance"],name_aeropuerto_mayor)

    distancia_total = prim.weightMST(data_structs["carga_airports_distance"],search)

    total_trayectos = lt.size(search["mst"])

    respuesta = lt.newList(datastructure="ARRAY_LIST")

    for ruta in lt.iterator(search["mst"]):

        origen = e.either(ruta)
        entry = mp.get(data_structs["airports_map"],origen)
        data_origen = me.getValue(entry)
        destino = e.other(ruta,origen)
        entry = mp.get(data_structs["airports_map"],destino)
        data_destino = me.getValue(entry)
        distancia = e.weight(ruta)
        tiempo,aeronave = e.weight(gr.getEdge(data_structs["carga_airports_time"],origen,destino))

        data = new_data(data_origen,data_destino,distancia,tiempo,aeronave)

        lt.addLast(respuesta,data)
    return respuesta,data_aeropuerto_mayor,total_trayectos,distancia_total
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Para este requerimiento primero se obtiene el aeropuerto de mayor concurrencia de carga y sobre este se calcula el MST-Prim , luego se obtiene el peso del mst el cual se pide en el requerimiento y luego se itera sobre el mst para ir agregando lo que se pide entre los aeropuertos de origen y destino.

Entrada	None.
Salidas	Lista con los datos a imprimir, el dato del aeropuerto con mayor concurrencia carga, el total de trayectos que en teoría es el número de arcos más 1, y la distancia total del MST
Implementado (Sí/No)	Yefran Cespedes

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
`data_aeropuerto_mayor = mayor_concurrencia(data_structs,data_structs["airports_carga_map"],1)` `name_aeropuerto_mayor = lt.getElement(data_aeropuerto_mayor,0)["ICAO"]`	O(1)

`search = prim.PrimMST(data_structs["carga_airports_distance"],name_aeropuerto_mayor)`
`distancia_total = prim.weightMST(data_structs["carga_airports_distance"],search)`
`total_trayectos = lt.size(search["mst"])`
`respuesta = lt.newList(datastructure="ARRAY_LIST")`
	O(ElogV) donde E es el número de arcos y V el número de vértices

	O(n) donde n es el número de arcos del MST
TOTAL	***O(n+ElogV)***

Análisis

Para este requerimiento se hizo uso del algoritmo MST el cual asegura el grafo de peso mínimo en grafos no dirigidos por ende se hizo necesario agregarle a la estructura de datos la creación de un grafo no dirigido de tipo carga con peso distancia y otro para peso tiempo, luego al calcular el MST se hace la iteración sobre los arcos que retorna para poder agregar y calcular la información que se pide, el requerimiento se hace en un tiempo ideal, el cual depende en mayor medida de la complejidad del algoritmo MST-PRIM y el número de arcos retornados en este requerimiento.

Requerimiento <<5>>

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_5(data_structs):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    data_aeropuerto_mayor = mayor_concurrencia(data_structs,data_structs["airports_militar_map"],1)
    name_aeropuerto_mayor = lt.getElement(data_aeropuerto_mayor,0)["ICAO"]

    search = prim.PrimMST(data_structs["militar_airports_time"],name_aeropuerto_mayor)

    data_rutas = prim.edgesMST(data_structs["militar_airports_time"],search)

    total_trayectos = lt.size(search["mst"])

    respuesta = lt.newList(datastructure="ARRAY_LIST")
    distancia_total =0

    for ruta in lt.iterator(search["mst"]):

        origen = e.either(ruta)
        entry = mp.get(data_structs["airports_map"],origen)
        data_origen = me.getValue(entry)
        destino = e.other(ruta,origen)
        entry = mp.get(data_structs["airports_map"],destino)
        data_destino = me.getValue(entry)
        tiempo = e.weight(ruta)
        distancia,aeronave = e.weight(gr.getEdge(data_structs["militar_airports_distance"],origen,destino))
        distancia_total+=distancia

        data = new_data(data_origen,data_destino,distancia,tiempo,aeronave)

        lt.addLast(respuesta,data)

    return respuesta,data_aeropuerto_mayor,total_trayectos,distancia_total
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Para este requerimiento primero se calculo un MST sobre el grafo de tiempo partiendo desde el aeropuerto de mayor concurrencia militar, luego se calcula un EDGES-MST con el fin de que retorne todos los arcos en forma de pila y poder trabajar con el resultado del MST, luego se itera sobre el resultado del MST para ir obteniendo los aeropuertos de origen y destino e ir agregando la información que se está pidiendo en el requerimiento a una lista que luego se va a retornar con fin de imprimirla en el view.py

Entrada	None
Salidas	Lista de los vuelos del MST, datos del aeropuerto de mayor concurrencia militar, total de trayectos y distancia final
Implementado (Sí/No)	Luis Vega

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

`data_aeropuerto_mayor = mayor_concurrencia(data_structs,data_structs["airports_militar_map"],1)` `name_aeropuerto_mayor = lt.getElement(data_aeropuerto_mayor,0)["ICAO"]`	O(1)
P `search = prim.PrimMST(data_structs["militar_airports_time"],name_aeropuerto_mayor)`	O(ElogV) donde E son los arcos y V es el número de vértices
`data_rutas = prim.edgesMST(data_structs["militar_airports_time"],search)`	O(E)

`for ruta in lt.iterator(search["mst"]):`
`origen = e.either(ruta)`
`entry = mp.get(data_structs["airports_map"],origen)`
`data_origen = me.getValue(entry)`
`destino = e.other(ruta,origen)`
`entry = mp.get(data_structs["airports_map"],destino)`
`data_destino = me.getValue(entry)`
`tiempo = e.weight(ruta)`
`distancia,aeronave = e.weight(gr.getEdge(data_structs["militar_airports_distance"],origen,destino))`
`distancia_total+=distancia`
`data = new_data(data_origen,data_destino,distancia,tiempo,aeronave)`
`lt.addLast(respuesta,data)`
	O(n) donde n es el número de arcos en el mst

TOTAL	***O(...)***

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el análisis de complejidad.

Se hace uso del MST en un grafo de tiempo porque asegura el grafo de tiempo mínimo respecto al aeropuerto de mayor concurrencia Militar.

Requerimiento <<6>>

Plantilla para el documentar y analizar cada uno de los requerimientos.

```
def req_6(data_structs,num):
    """
    Función que soluciona el requerimiento 6
    """
    num+=1
    # TODO: Realizar el requerimiento 6

    airports = mayor_concurrencia(data_structs,data_structs["airports_comercial_map"],num)
    data_airport_mayor = lt.getElement(airports,1)
    lt.deleteElement(airports,1)
    key_airport_mayor = data_airport_mayor["ICAO"]
    name_airport_mayor = data_airport_mayor["NOMBRE"]
    search = djk.Dijkstra(data_structs["comercial_airports_distance_directed"],key_airport_mayor)
    cont =2
    respuesta = lt.newList()
    for airport in lt.iterator(airports):
        key_airport = airport["ICAO"]
        name_airport = airport["NOMBRE"]
        exist_ruta = djk.hasPathTo(search,key_airport)
        if  exist_ruta:
            distancia_total = djk.distTo(search,key_airport)
            ruta = djk.pathTo(search,key_airport)
            data = new_data_req6(data_structs,ruta,distancia_total,key_airport_mayor)

            lt.addLast(respuesta,data)

        else:
            print("No existe una ruta entre el aeropuerto de mayor importancia: ",name_airport_mayor, " y el ",cont," de mayor importancia ",name_airport)

        cont+=1

    return respuesta,data_airport_mayor
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Para este requerimiento primero se obtiene los n aeropuertos que el usuario desea cubrir desde el aeropuerto de mayor concurrencia, luego se calcula Dijkstra sobre el aeropuerto de mayor importancia comercial y finalmente se calcula la ruta sobre los n aeropuertos a cubrir desde el aeropuerto

Entrada	Estructura de datos y el número de aeropuertos de mayor concurrencia a cubrir.
Salidas	Lista con los datos a imprimir, datos del aeropuerto de mayor concurrencia comercial desde el cual se parte.
Implementado (Sí/No)	Grupal.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
`airports = mayor_concurrencia(data_structs,data_structs["airports_comercial_map"],num)` `data_airport_mayor = lt.getElement(airports,1)`	O(1)

`search = djk.Dijkstra(data_structs["comercial_airports_distance_directed"],key_airport_mayor)`
`cont =2`
`respuesta = lt.newList()`
	O((V+E)logV) donde v son los números de los vertives y E el número de arcos

`for airport in lt.iterator(airports):`
`  key_airport = airport["ICAO"]`
`  name_airport = airport["NOMBRE"]`
`  exist_ruta = djk.hasPathTo(search,key_airport)`
`  if exist_ruta:`
`  distancia_total = djk.distTo(search,key_airport)`
`  ruta = djk.pathTo(search,key_airport)`
`  data = new_data_req0(data_structs,ruta,distancia_total,key_airport_mayor)`
`  lt.addLast(respuesta,data)`
`  else:`
`  print("No existe una ruta entre el aeropuerto de mayor importancia: ",name_airport_mayor, " y el ",cont," de mayor importancia ",name_airport)`
`  cont+=1`
	O(n) donde n es el número de aeropuertos a cubrir

TOTAL	***O(n+(v+e)loge)***

Análisis

Se hace uso de Dijkstra con el fin de obtener el camino más cercano de un vértice fuente a los demás vértices, luego se itera sobre los aeropuertos de mayor concurrencia a cubrir, entonces se pregunta si existe una ruta desde ese aeropuerto y si la hay se devuelve la información.

Requerimiento <<7>>

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_7(data_structs,origen_latitud,origen_longitud,destino_latitud,destino_longitud):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    distance_origen,airport_origen,distance_destino,airport_destino = get_nearby_airports(data_structs,(origen_latitud,origen_longitud),(destino_latitud,destino_longitud))

    entry = mp.get(data_structs["airports_map"],airport_origen)
    data_origen = me.getValue(entry)
    entry = mp.get(data_structs["airports_map"],airport_destino)
    data_destino = me.getValue(entry)

    respuesta = lt.newList()
    if distance_origen<=30 and distance_destino <= 30:
        tiempo_total =0
        distancia_total = distance_origen+distance_destino
        data_structs["search"] = djk.Dijkstra(data_structs["comercial_airports_time_directed"],airport_origen)
        exist_camino = djk.hasPathTo(data_structs["search"],airport_destino)
        if exist_camino:

            ruta = djk.pathTo(data_structs["search"],airport_destino)
            total_airports = st.size(ruta)+1
            while (not st.isEmpty(ruta)):
                airport=st.pop(ruta)
                key_airportA = airport["vertexA"]
                entry_A = mp.get(data_structs["airports_map"],key_airportA)
                airportA= me.getValue(entry_A)
                time_A_B = airport["weight"]
                key_airportB = airport["vertexB"]
                entry_B = mp.get(data_structs["airports_map"],key_airportB)
                airportB= me.getValue(entry_B)
                arco_distance = gr.getEdge(data_structs["comercial_airports_distance_directed"],key_airportA,key_airportB)
                distance_A_B= e.weight(arco_distance)
                tiempo_total+=time_A_B
                distancia_total+=distance_A_B
                data_fligth = new_data_req7(airportA,airportB,time_A_B,distance_A_B)
                lt.addLast(respuesta,data_fligth)

        return distancia_total,total_airports,respuesta,tiempo_total
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Primero se obtiene los aeropuertos de origen y destino más cercanos a los puntos dados, si las distancia son menores a 30 km los puntos del aeropuerto entonces se va calcular Dijkstra sobre el aeropuerto de origen y el grafo dirigido de tiempo, luego se pregunta si existe un camino entre este el aeropuerto de origen y el de destino, si hay una ruta, se obtiene y luego se itera sobre los arcos que retorna paht_to, con el fin de agregar la información a una lista e imprimirla en el view.py.

Entrada	Latitud y longitud tanto del punto inicial, como el ponto de origen
Salidas	Distancia total, total de aeropuertos, el tiempo total del trayecto y finalmente la lista con la información de los aeropuertos en el trayecto
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

`distance_origen,airport_origen,distance_destino,airport_destino = get_nearby_airports(data_structs,(origen_latitud,origen_longitud),(destino_latitud,destino_longitud))`
`entry = mp.get(data_structs["airports_map"],airport_origen)`
`data_origen = me.getValue(entry)`
`entry = mp.get(data_structs["airports_map"],airport_destino)`
`data_destino = me.getValue(entry)`
	O(n) donde n es el número de aeropuertos

`data_structs["search"] = djk.Dijkstra(data_structs["comercial_airports_time_directed"],airport_origen)` `exist_camino = djk.hasPathTo(data_structs["search"],airport_destino)`	O((E+V)logV) donde E es el número de arcos y V el número de vertices

	O(t) donde t es el número de arcos del trayecto
TOTAL	***O(n+((E+V)LogV)+t)***

Análisis

El requerimiento es bastante eficiente y depende en gran medida del algoritmo de Dijkstra el cual se implementa con el fin de obtener los caminos desde mi vértice fuente a mis demás vértices en este caso en el menor tiempo posible.

Requerimiento <<8>>

Plantilla para el documentar y analizar cada uno de los requerimientos.

```
elif (i == 1 or i==2) and mp.size(mapa) != None:
        entry= mp.get(mapa,i)
        if entry != None:
            airports = me.getValue(entry)
            req = folium.FeatureGroup(("requerimiento "+str(i))).add_to(mi_mapa)
            color = random.choice(colores)
            for i in range(1,lt.size(airports)):
                key_airport_o = lt.getElement(airports,i)["ICAO"]
                key_airport_d = lt.getElement(airports,i+1)["ICAO"]
                entry_origen = mp.get(data["airports_map"],key_airport_o)
                data_origen = me.getValue(entry_origen)
                entry_destino = mp.get(data["airports_map"],key_airport_d)
                data_destino = me.getValue(entry_destino)
                latitud_O=Float(data_origen["LATITUD"].replace(",","."))
                longitud_O = float(data_origen["LONGITUD"].replace(",","."))

                latitud_d=Float(data_destino["LATITUD"].replace(",","."))
                longitud_d = float(data_destino["LONGITUD"].replace(",","."))

                info_origen = str(data_origen["NOMBRE"] + " "+ data_origen["CIUDAD"]+ " " + data_origen["PAIS"])
                folium.Marker(location=[latitud_O,longitud_O],tooltip=key_airport_o,popup=info_origen,icon=folium.Icon(color=color),).add_to(req)

                info_destino = str(data_destino["NOMBRE"] + " "+ data_destino["CIUDAD"]+ " " + data_destino["PAIS"])
                folium.Marker(location=[latitud_d,longitud_d],tooltip=key_airport_d,popup=info_destino,icon=folium.Icon(color=color),).add_to(req)

                coordenadas = [ (latitud_O,longitud_O),(latitud_d,longitud_d)]

                folium.PolyLine(coordenadas).add_to(req)

elif (i == 6 ) and mp.size(mapa) != None:
        entry= mp.get(mapa,i)
        if entry != None:
            airports = me.getValue(entry)
            req = folium.FeatureGroup(("requerimiento "+str(i))).add_to(mi_mapa)
            color = random.choice(colores)
            for airport in lt.iterator(airports):

                for arco in lt.iterator(airport["ruta"]):
                    key_airport_o = arco["vertexA"]
                    key_airport_d = arco["vertexB"]
                    entry_origen = mp.get(data["airports_map"],key_airport_o)
                    data_origen = me.getValue(entry_origen)
                    entry_destino = mp.get(data["airports_map"],key_airport_d)
                    data_destino = me.getValue(entry_destino)
                    latitud_O=Float(data_origen["LATITUD"].replace(",","."))
                    longitud_O = float(data_origen["LONGITUD"].replace(",","."))

                    latitud_d=Float(data_destino["LATITUD"].replace(",","."))
                    longitud_d = float(data_destino["LONGITUD"].replace(",","."))

                    info_origen = str(data_origen["NOMBRE"] + " "+ data_origen["CIUDAD"]+ " " + data_origen["PAIS"])
                    folium.Marker(location=[latitud_O,longitud_O],tooltip=key_airport_o,popup=info_origen,icon=folium.Icon(color=color),).add_to(req)

                    info_destino = str(data_destino["NOMBRE"] + " "+ data_destino["CIUDAD"]+ " " + data_destino["PAIS"])
                    folium.Marker(location=[latitud_d,longitud_d],tooltip=key_airport_d,popup=info_destino,icon=folium.Icon(color=color),).add_to(req)

                    coordenadas = [ (latitud_O,longitud_O),(latitud_d,longitud_d)]

                    folium.PolyLine(coordenadas).add_to(req)

else:
        print("No hay datos según lo reportado por el requerimiento "+str(i))
```

Descripción

Para este requerimiento se creo un mapa que almacenará las respuestas que retornaba los otros requerimientos, los requerimientos 1 y 2 tiene una estructura de respuesta similar, luego 3,4,5 y 7 igual, ya por ultimo 6 tiene otra respuesta diferente, por ese se crean tres casos para cada uno de estos grupos, pero en general, lo que se hace es escoger un color para el requerimiento, luego crear una capa del requerimiento, esto con el fin de poder dividir los datos por requerimientos en el mapa, y luego se itera sobre las respuestas, agregando marcadores por cada aeropuerto en la respuesta y vectores en forma de línea que une los aeropuertos según el vuelo como esté organizado en la respuesta del requerimiento, esto con el fin de representar mejor la idea de lo que se está imprimiendo.

Entrada	Mapa con las respuestas de todos los requerimientos y la estructura de datos
Salidas	Mapa en forma de html
Implementado (Sí/No)	Si

Pasos	Complejidad
Iteración por cada respuesta de los requerimientos	(b+c+d+e+f+g+h) donde b,c,d,e,f,g,h son los datos de los requerimientos 1 al 7
Acceder, obtener, eliminar, insertar	O(1)
TOTAL	***O(***b+c+d+e+f+g+h***)***

Análisis

Usa bastante memoria el uso de Folium para la creación del mapa, por ende se planteo imprimir por cada requerimiento si se pasa de 100 datos bajarlo a 100 ya que el colocar tantos marcadores y vectores a forma de línea es bastante pesado al momento de abrir el archivo sin embargo, al momento de ejecutarlo es espacio temporal es ideal y constante y no demorado.

DIAGRAMA DE DATOS