ANÁLISIS DEL RETO

Cesar Augusto Espinosa Gomez, 202220692, c.espinosag@uniandes.edu.co

Juan Eduardo Ballesteros, 202221829, je.ballesteros@uniandes.edu.co

Santiago Muñoz Martinez, 202221167, s.munoz234@uniandes.edu.co

Carga de Datos

La implementación de la carga de datos se divide en tres partes:

1. Cargar la información de los aeropuertos con vuelos registrados a una tabla de hash.
2. Cargar la información de los vuelos y aeropuertos a los grafos correspondientes
3. Generar los árboles binarios ordenados para organizar los aeropuertos dependiendo de su concurrencia para cada tipo de vuelo (comercial, carga o militar.

Para la primera parte de la carga, se recorre el archivo 'airports-2022' utilizando la herramienta DictReader de la librería CSV. Esto permite generar un diccionario para cada aeropuerto en el archivo, con el cual se accede al código ICAO del aeropuerto. Además, a este diccionario se le añaden tres contadores de concurrencia, uno por cada tipo de vuelo, se transforman los datos de longitud y latitud en número y se crea genera una tupla para guardar estas coordenadas (latitud, longitud) de manera conjunta. Una vez se han hecho las modificaciones necesarias al diccionario, este se inserta como valor a la tabla de hash 'airports_map', donde su llave es el código ICAO del aeropuerto. De esto se encarga la función 'add_info_airports_map'.

Una vez terminada la carga de los aeropuertos, se procede a recorrer el archivo 'flights-2022' de la misma forma que el anterior. A partir del diccionario para cada vuelo generado en el recorrido, la función 'load_connection' determina el tipo de vuelo y añade la conexión entre los dos aeropuertos en los grafos correspondientes. Para cada tipo de vuelo hay dos grafos, uno con pesos de distancia y otro con pesos de tiempo. Los pesos de distancia se calculan a través de la librería haversine y las coordenadas registradas para cada aeropuerto. La función primero revisa si los aeropuertos del vuelo ya existen en el grafo, de no exisitir los añade, para luego generar la conexión pertinente. En este proceso también se actualizan los contadores para la concurrencia de los aeropuertos, directamente en los diccionarios guardados en la tabla de hash.

Habiendo terminado la carga de las conexiones, se invoca a la función 'create_ordered_maps_airports', la cual recorre la tabla de hash con la información de los aeropuertos y, para cada aeropuerto, accede a su concurrencia comercial, de carga y militar. Con esto, la función organiza los aeropuertos (únicamente sus códigos ICAO) en tres árboles binarios ordenados, en los que las llaves son conteos de concurrencia y los valores son listas con los códigos ICAO de los aeropuertos que tienen esa concurrencia. Hay un árbol para cada uno de los tipos de vuelo.

Con esto, la carga genera las siguientes estructuras:

1. Airports_map: Tabla de hash con la información de los aeropuertos

2. Time_grap_type (directed): Un grafo cuyos pesos son tiempos de vuelo entre aeropuertos; hay tres, uno por cada tipo de vuelo (type)
3. Dist_grap_type (directed): Un grafo dirigido cuyos pesos son distancias entre aeropuertos; hay tres, uno por cada tipo de vuelo
4. Type_tree: Árbol binario ordenado para organizar los aeropuertos por su concurrencia, hay tres, uno por cada tipo de vuelo

Requerimientos 1 y 2

```python
def req_1_2(database, origin, dest, req):
    """Función que resuelve los requerimientos 1 y 2. La función accede a los grafos con pesos de distancia y
    tiempo para vuelos comerciales, revisa que las coordenadas ingresadas para origen y destino sean válidas, y
    devuelve el resultado dependiendo del requerimiento que se quiere resolver.
        1. Para el req 1, la función utiliza el algoritmo 'dfs' para determinar si existe un camino posible entre los
        puntos ingresados, accede a ese camino y genera un retorno adecuado para luego imprimir en pantalla.
        2. Para el req 2, los procedimientos son muy similares, solo se cambia al algoritmo 'bfs' para determinar el camino
        con menos escalas.

    Args:
        database (dict): Base de datos con las estructuras del modelo
        origin (tuple): Tupla de coordenadas (lat, lon) del punto de origen
        dest (tuple): Tupla de coordenadas (lat, lon) del punto de destino
        req (int): Requerimiento que se quiere resolver, '1' o '2'

    Returns:
        tuple: Una tupla con la lista de trayectos que se deben tomar, la distancia y tiempo de vuelos totales
        del camino, el número total de aeropuertos vistados, y la información de los aeropuertos de origen y destino.
    """
    #acceder a los grafos necesarios
    commercial_graph_dist = database["dist_graph_com"]
    commercial_graph_time = database["time_graph_com"]
    #acceder a la tabla de hash con la info de los aeropuertos
    airports_map = database["airports_map"]
    #determinar los aeropuertos más cercanos y sus distancias a los puntos de origen y destino.
    origin_airport, origin_distance = check_coordinates(origin, database)
    destination_airport, dest_distance = check_coordinates(dest, database)
    #si la distancia a los aeropuertos tanto del origen como del destino es menor a 30 km, sigue con el proceso.
    if origin_distance <= 30 and dest_distance <= 30:
        #establece el algoritmo de búsqueda y la estructura de búsqueda dependiendo del requerimiento que se quiere resolver.
        if req == 1:
            search_algo = dfs
            search_graph_dist = dfs.DepthFirstSearch(commercial_graph_dist, origin_airport)
        elif req == 2:
            search_algo = bfs
```

```
    if req == 1:
        search_algo = dfs
        search_graph_dist = dfs.DepthFirstSearch(commercial_graph_dist, origin_airport)
    elif req == 2:
        search_algo = bfs
        search_graph_dist = bfs.BreathFirstSearch(commercial_graph_dist, origin_airport)
    #revisar si existe un camino entre el aeropuerto de origen y el de destino
    path_exists = search_algo.hasPathTo(search_graph_dist, destination_airport)
    if path_exists == True: #hay camino
        #generar placeholders para la información que se debe retornar
        airports_list = lt.newList()
        total_dist = origin_distance+dest_distance
        total_time = 0
        #generar el camino entre origen y destino mediante el algoritmo pertinente.
        dist_path = search_algo.pathTo(search_graph_dist, destination_airport)
        #actualizar el número total de aeropuertos
        total_airports = st.size(dist_path)
        origin_airport_full = ""
        destination_airport_full = ""
        i = 1
        #recorrer el camino (solo trae los vértices)
        while i < st.size(dist_path):
            #acceder al aeropuerto de destino y al de origen
            org_airport = lt.getElement(dist_path, i+1)
            dest_airport = lt.getElement(dist_path, i)
            #acceder a los arcos entre los aeropuertos en ambos grafos
            dist_edge = gr.getEdge(commercial_graph_dist, org_airport, dest_airport)
            time_edge = gr.getEdge(commercial_graph_time, org_airport, dest_airport)
            #acceder a la distancia y al tiempo de vuelo
            flight_distance = dist_edge["weight"]
            flight_time = time_edge["weight"]
            #sumar el tiempo de vuelo y la distancia a los totales del camino
            total_dist+= flight_distance
            total_time+=flight_time
```

```
            total_dist+= flight_distance
            total_time+=flight_time

            #generar el diccionario para el camino
            airports_info= get_airports_info_req1_2(airports_map, org_airport, dest_airport,flight_distance, flight_time)

            #generar la información para el aeropuerto de origen y destino globales
            if org_airport == origin_airport:
                origin_airport_full = "{0} ({1})".format(airports_info["NOMBRE ORIGEN"], org_airport)
            if dest_airport == destination_airport:
                destination_airport_full = "{0} ({1})".format(airports_info["NOMBRE DESTINO"], dest_airport)
            #añadir el diccionario de la conexión a la lista de entrega
            lt.addFirst(airports_list, airports_info)
            i+=1
        return airports_list, total_dist, total_time, total_airports, origin_airport_full, destination_airport_full
    else:
        return -2, origin_airport, origin_distance, destination_airport, dest_distance
else:
    return -1, origin_airport, origin_distance, destination_airport, dest_distance
```

```python
def check_coordinates(coord_tuple, database):
    """Función que determina el aeropuerto registrado en la base de datos
    mas cercano a un par de coordenadas dado.

    Args:
        coord_tuple (tuple): Tupla de latitud y longitud en decimales, para la cual se quiere
        determinar el aeropuerto más cercano.
        database (dict): Base de datos que contiene las estructuras del modelo

    Returns:
        tuple: El aeropuerto de menor distancia a las coordenadas y esta distancia, en km.
    """
    #acceder a la lista de aeropuertos registrados en la base de datos.
    airports_map = database["airports_map"]
    airports_lst = mp.valueSet(airports_map)
    #placeholders para la información a guardar
    least_distance_airport = ""
    least_distance = 1000000000000
    #recorrer la lista de aeropuertos
    for airport in lt.iterator(airports_lst):
        #acceder al código y las coordenadas de cada aeropuerto
        airport_code = airport["ICAO"]
        airport_coord = airport["coord"]
        #determinar la distancia entre el aeropuerto y las coordenadas ingresadas y el aeropuerto
        #utiliza haversine
        dist = hvs(coord_tuple, airport_coord)
        #compara con la menor distancia actual y, en caso de ser menor, actualiza los datos.
        if dist <least_distance:
            least_distance_airport = airport_code
            least_distance = dist
    return least_distance_airport, least_distance
```

Descripción

El requerimiento 1 y 2 se encargan, respectivamente de determinar si hay un camino aéreo posible entre dos pares de coordenadas entregadas como origen y destino y determinar el camino más corto (con menos escalas) entre dos pares de coordenadas. Dado que ambos piden retornos similares, la diferencia entre los dos radica principalmente en el algoritmo usado para los grafos (dfs para el primero, bfs para el segundo). Por esto, se decidió utilizar una única función modular la cual, a través de un condicional, puede adaptarse para trabajar para el primer o segundo requerimiento. Esto permite ahorrar tiempo, pues dado que muchos de los métodos hubieran sido iguales para una segunda función, resultaba inoficioso desarrollarla por aparte cuando se podían hacer pequeños cambios en la función original tal que funcionara para ambos.

Sin importar el requerimiento, la función utiliza el siguiente procedimiento:

1. Utiliza la función 'check_coordinates' para verificar si las coordenadas de destino y origen están a menos de 30 km de un aeropuerto con vuelos comerciales
2. Una vez con los aeropuertos para origen y destino, genera la estructura de búsqueda a partir de estos y teniendo en cuenta el requerimiento que se quiere realizar.
3. Se verifica que haya un camino entre los dos aeropuertos
4. Si hay camino, este se guarda, se recorre y se guarda la información de los vuelos en una lista de diccionarios.
    a. En este proceso, se generan contadores para la distancia y tiempos de vuelo totales que se van actualizando con cada vuelo

Entrada	Base de datos que contiene las estructuras del modelo (database), coordenadas de origen y coordenadas de destino. Adicionalmente, se incluye un parámetro 'req' el cual nos permite indicar que requerimiento se quiere resolver. Por ello, el usuario no tiene que ver con esta entrada y es manejada desde el 'controller'
Salidas	Tupla con la lista de vuelos en el camino, la distancia y tiempos totales del camino, así como la información del aeropuerto de origen y destino. En caso de haber errores, como que las coordenadas están muy lejos de un aeropuerto o no hay un camino entre aeropuertos, se generan retornos de error que permiten gestionar la impresión en consola.
Implementado (Sí/No)	Si, implementado por Santiago Muñoz

Análisis de complejidad

Dado que se están implementando dos requerimientos con variaciones en complejidad (debidas a sus algoritmos) se harán las distinciones apropiadas cuando sea necesario.

Pasos	**Complejidad**
Acceder a los grafos y la tabla de hash necesarias	O(1), asignación simple
Revisar los aeropuertos más cercanos a las coordenadas de origen y destino ingresadas.	O(v), donde *v* corresponde al número de aeropuertos que hay en la carga. Esto pues la función 'check_coordinates' se implementó utilizando un recorrido completo, puesto que resultaba complicado generar esta busqueda de una manera más eficiente.
Generar el grafo de búsqueda a partir del grafo original	O(E+V), donde E y V son la cantidad de arcos y vértices del grafo. Esto pues, sin importar el requerimiento, pues los algoritmos de BFS y DFS deberán visitar todos los vértices y arcos en el peor caso.
Determinar si hay camino entre los aeropuertos	O(1), revisar si el aeropuerto de destino está en la lista de adyacencia del aeropuerto de origen en el grafo de búsqueda.
Reconstruir camino entre aeropuerto de origen y aeropuerto de destino y recorrerlo para acceder a la información de cada vuelo.	Requerimiento 1: O(E), esto pues, dada la naturaleza del algoritmo DFS, puede que el camino para llegar al aeropuerto de destino requiera, en el peor de los casos, pasar por todos los demás vértices. Requerimiento 2: O(1), pues BFS asegura el camino con la menor cantidad de saltos entre dos aeropuertos. Si se agregaran más datos, este camino no cambiaría, a menos de que se genere un nuevo camino más corto.

	Por ende, reconstruir este camino generalmente requerirá pocas operaciones.
TOTAL	***O(E+V),* pues resulta generar las estructuras para el recorrido resulta lo más demorado de todos los pasos planteados.**

Pruebas Requerimiento 1

- Máquina 1: Macbook Air M1 2020, 16 GB RAM, macOS Ventura 13.0
- Máquina 2: Acer Nitro Intel i5-10300H, 8GB Ram, Windows 11

Valores:

```
Latitud origen: 12.137
Longitud origen: -68.26
Latitud destino: -4.21
Longitud destino: -69.98
```

Máquina	**Tiempo [ms]**
Máquina 1	15.119
Máquina 2	28.658

Pruebas Requerimiento 2

- Máquina 1: Macbook Air M1 2020, 16 GB RAM, macOS Ventura 13.0
- Máquina 2: Acer Nitro Intel i5-10300H, 8GB Ram, Windows 11

Valores:

```
Latitud origen: 12.137
Longitud origen: -68.26
Latitud destino: -4.21
Longitud destino: -69.98
```

Máquina	**Tiempo [ms]**
Máquina 1	9.709
Máquina 2	12.302

Requerimiento 3,4,5

Funciones Auxiliares

```python
def find_most_concurrency_airport(database, req):
    if req == "req3":
        tree = database["commercial_tree"]
    elif req == "req4":
        tree = database["cargo_tree"]
    else:
        tree = database["military_tree"]

    concurrency = om.maxKey(tree)
    list_max_key = om.get(tree, concurrency)
    list = me.getValue(list_max_key)

    sorted_lst = alphabetic_sort(list)

    airport_entry = mp.get(database["airports_map"], lt.getElement(sorted_lst, 1))
    concurrency_airport = me.getValue(airport_entry)

    return concurrency, concurrency_airport

def graph_selector(database, req):
    if req == "req3":
        graph1 = database["dist_nodirected_graph_com"]
        graph2 = database["time_nodirected_graph_com"]
    elif req == "req4":
        graph1 = database["dist_nodirected_graph_cargo"]
        graph2 = database["time_nodirected_graph_cargo"]
    else:
        graph1 = database["time_nodirected_graph_mil"]
        graph2 = database["dist_nodirected_graph_mil"]

    return graph1, graph2
```

```python
def alphabetic_sort(lst):
    ordered_lst =  sort_algorithm.sort(lst, alphabetic_comparison)
    return ordered_lst
```

Requerimiento:

```python
def req_3_4_5(database, req):
    """
    Función que soluciona el requerimiento 3, 4, 5.
    Args:
        Database: el modelo con las estructuras de datos
        Req: cuál de los tres requerimientos se requiere resolver

    returns:
        Concurrency: Nivel de Concurrencia de un Aeropuerto
        Source_Airport: Diccionario del Aeropuero Base
        Total_Weight: Peso total del MST (Distancia o Tiempo)
        Other_Weight: El otro tipo de peso total del MST (Tiempo o Distancia)
        Airport_dicc_list: la lista de aeropuertos alcanzables en el mst
    """
    airports_map = database["airports_map"]
    airplanes_map = database["edges_map"]
    concurrecy, source_airport = find_most_concurrency_airport(database, req)
    ICAO_source = source_airport["ICAO"]

    selected_graph, other_graph = graph_selector(database, req)

    MST_tree = prim.PrimMST(selected_graph, ICAO_source)
    total_weight = prim.weightMST(selected_graph, MST_tree)
    cola = prim.edgesMST(selected_graph, MST_tree)
    marked_map = cola["marked"]
    edgeTo_map = cola["edgeTo"]
    vertices_list = mp.keySet(marked_map)
    total_other_weight = 0
    airport_dict_list = lt.newList("ARRAY_LIST")
    for airport in lt.iterator(vertices_list):
        if airport != ICAO_source:
            airport_info = me.getValue(mp.get(airports_map, airport))
            path_weight = 0
            path_other_weight = 0
            airplane_types = lt.newList()
            path_source = prim_path(edgeTo_map, ICAO_source, airport)
            i = 1
            while i < st.size(path_source):
                #acceder al aeropuerto de destino y al de origen
                org_airport = lt.getElement(path_source, i)
                dest_airport = lt.getElement(path_source, i+1)
                #acceder a los arcos entre los aeropuertos en ambos grafos
                edge = gr.getEdge(selected_graph, org_airport, dest_airport)
                other_edge = gr.getEdge(other_graph, org_airport, dest_airport)
                #acceder a la distancia y al tiempo de vuelo
```

```
        path_weight = 0
        path_other_weight = 0
        airplane_types = lt.newList()
        path_source = prim_path(edgeTo_map, ICAO_source, airport)
        i = 1
        while i < st.size(path_source):
            #acceder al aeropuerto de destino y al de origen
            org_airport = lt.getElement(path_source, i)
            dest_airport = lt.getElement(path_source, i+1)
            #acceder a los arcos entre los aeropuertos en ambos grafos
            edge = gr.getEdge(selected_graph, org_airport, dest_airport)
            other_edge = gr.getEdge(other_graph, org_airport, dest_airport)
            #acceder a la distancia y al tiempo de vuelo
            edge_weight = edge["weight"]
            edge_other_weight = other_edge["weight"]
            #sumar el tiempo de vuelo y la distancia a los totales del camino
            path_weight+= edge_weight
            path_other_weight+=edge_other_weight
            total_other_weight += edge_other_weight
            airplane_type_entry = mp.get(airplanes_map, (org_airport, dest_airport))
            if airplane_type_entry == None:
                airplane_type_entry = mp.get(airplanes_map, (dest_airport, org_airport))
            airplane_type = me.getValue(airplane_type_entry)
            if not lt.isPresent(airplane_types, airplane_type):
                lt.addLast(airplane_types, airplane_type)
            i+=1
        airport_dict ={}
        airport_dict["NOMBRE"] = airport_info["NOMBRE"]
        airport_dict["ICAO"] = airport
        airport_dict["CIUDAD"] = airport_info["CIUDAD"]
        airport_dict["PAIS"] = airport_info["PAIS"]
        airport_dict["weight"] = path_weight
        airport_dict["other_weight"] = path_other_weight
        airport_dict["airplanes_list"] = airplane_types
        airport_dict["coord"] = airport_info["coord"]
        lt.addLast(airport_dict_list, airport_dict)
```

Descripción

Se hizo una única función para estos tres requerimientos debido a que la lógica es la misma, y lo que varía es el grafo seleccionado para generar los recorridos. Básicamente, la función toma el grafo determinado por parámetro, encuentra el aeropuerto de mayor concurrencia a través de un árbol binario, procede a construir el MST utilizando el algoritmo Prim y luego utiliza la función '*weightMST*' para hallar uno de los dos pesos (el otro será calculado al hacer el recorrido). Ahora, la función accede a la estructura de búsqueda, y por falta de la existencia de una función determinada en disclib, se construyó una función que genera un camino entre dos vértices y entrega una cola (del Source al aeropuerto de destino), esta cola se recorre para hallar el peso faltante, los tipos de aeronaves utilizadas y, en general, para encontrar la lista de aeropuertos conectados al Source.

Entrada	No recibe parámetros de entrada
Salidas	Req 3: El aeropuerto con mayor concurrencia comercial, el tiempo en milisegundos que demora en encontrar la solucion, el

	aeropuerto más importante según la concurrencia comercial con sus datos, la suma de la distancia total de los trayectos (cada trayecto partiendo desde el aeropuerto de referencia), el número de trayectos posibles cada trayecto partiendo desde el aeropuerto de mayor importancia. Y por último de la secuencia de trayectos encontrados la información de; aeropuerto de origen, aeropuerto de destino, distancia recorrida en el trayecto y tiempo total del trayecto. Req 4: El aeropuerto con mayor concurrencia de carga, el tiempo en milisegundos que demora en encontrar la solucion, la suma de la distancia total de los trayectos, el número de trayectos posibles cada trayecto partiendo desde el aeropuerto de mayor importancia. Y por último de la secuencia de trayectos encontrados la información de; aeropuerto de origen, aeropuerto de destino, distancia recorrida en el trayecto, tiempo total del trayecto y tipo de aeronave. Req 5: El aeropuerto con mayor concurrencia de carga, el tiempo en milisegundos que demora en encontrar la solucion, la suma de la distancia total de los trayectos, el número de trayectos posibles cada trayecto partiendo desde el aeropuerto de mayor importancia. Y por último de la secuencia de trayectos encontrados la información de; aeropuerto de origen, aeropuerto de destino, distancia recorrida en el trayecto, tiempo total del trayecto y tipo de aeronave.
Implementado (Sí/No)	Sí, implementado en conjunto por todos los miembros, cada uno aportando para su requerimiento asignado.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Hallar el aeropuerto de mayor concurrencia	O(log(v)+mlog(m)), donde v es el número de aeropuertos y m es el tamaño promedio de una lista en el árbol ordenado de aeropuertos por concurrencia. Esto pues consiste en acceder a la llave máxima del árbol ordenado para cada tipo de vuelo, acceder a su lista asociada, ordenarla y acceder al primer elemento.
Seleccionar el grafo de búsqueda	Constante

Generar MST sobre el grafo requerido desde el aeropuerto de mayor concurrencia	O(Elog(V)) implementación del algoritmo de MST en DISCLib
Recorrer todos los vértices que están en el MST	O(u), donde u es el número de aeropuertos en el MST
Encontrar y recorrer el camino desde un aeropuerto hacia el aeropuerto de origen	O(V-1), pues, en el peor caso, el camino a uno de los vértices puede implicar recorrer todos los vértices. Sin embargo, esto también significa que esta complejidad solo se podría dar para uno de los vértices, mientras que para los otros sería menor o, en un caso más común, la complejidad es menor para todos los vértices. Además, esta complejidad no depende completamente del número de datos, pues puede que, aun incrementando los datos, el camino en el MST entre dos vértices se mantenga igual y, por ende, la complejidad espacial de encontrarlo no cambie.
TOTAL	***O(Elog(V)), pues lo más demorado es generar el grafo de búsqueda por MST.***

Pruebas Requerimiento 3

- Máquina 1: Macbook Air M1 2020, 16 GB RAM, macOS Ventura 13.0
- Máquina 2: Acer Nitro Intel i5-10300H, 8GB Ram, Windows 11

Máquina	**Tiempo [ms]**
Máquina 1	259.514
Máquina 2	228.71

Pruebas Requerimiento 4

- Máquina 1: Macbook Air M1 2020, 16 GB RAM, macOS Ventura 13.0
- Máquina 2: Acer Nitro Intel i5-10300H, 8GB Ram, Windows 11

Máquina	**Tiempo [ms]**
Máquina 1	124.209
Máquina 2	172.09

Pruebas Requerimiento 5

- Máquina 1: Macbook Air M1 2020, 16 GB RAM, macOS Ventura 13.0
- Máquina 2: Acer Nitro Intel i5-10300H, 8GB Ram, Windows 11

Máquina	**Tiempo [ms]**

Máquina 1	99.861
Máquina 2	81.962

Requerimiento 6

```python
def req_6(database, num):
    """Función que soluciona el requerimiento 6.
    La función accede, a través de los árboles binarios ordenados generados en la carga, a los
    'num' aeropuertos con mayor concurrencia comercial, con lo cual se conoce el de mayor concurrencia.
    Para los otros aeropuertos, la función encuentra, usando el algoritmo de Dijkstra, la secuencia de
    vuelos con la menor distancia para llegar del aeropuerto de mayor concurrencia a cada uno de los otro
    aeropuertos. Para cada una de estas secuencias, la función determina los aeropuertos visitados,
    así como los vuelos que se deben tomar.

    Args:
        database (dict): Base de datos que contiene las estructuras del modelo
        num (int): Número de aeropuertos más concurridos que se quieren revisar.

    Returns:
        tuple: Entrega una lista ADT con la información de los caminos para llegar a los 'num'
        aeropuertos más importantes, así como la información sobre el aeropuerto de mayor concurrencia.
    """
    #acceder a las estructuras necesarias (tablas de hash, grafo y árbol binario ordenado)
    airports_com_tree = database["commercial_tree"]
    commercial_dist_graph = database["dist_graph_com"]
    airports_map = database["airports_map"]
    #determinar los 'num' aeropuertos con mayor concurrencia comercial en Colombia
    most_airports = get_most_count_tree_colombia(database, airports_com_tree, num+1)
    #determinar el aeropuerto de mayor concurrencia
    max_airport = most_airports[0]
    origin_airport = max_airport["ICAO"]
    #generar la estructura de búsqueda para el grafo.
    search_graph = djk.Dijkstra(commercial_dist_graph, origin_airport)
```

```python
airports_map = database["airports_map"]
#determinar los 'num' aeropuertos con mayor concurrencia comercial en Colombia
most_airports = get_most_count_tree_colombia(database, airports_com_tree, num+1)
#determinar el aeropuerto de mayor concurrencia
max_airport = most_airports[0]
origin_airport = max_airport["ICAO"]
#generar la estructura de búsqueda para el grafo.
search_graph = djk.Dijkstra(commercial_dist_graph, origin_airport)
airports_dict_lst = lt.newList("ARRAY_LIST")
#recorrer los aeropuertos más importantes, excluyendo el de mayor concurrencia.
for i in range(1,num+1):
    #acceder a la información del aeropuerto
    airport = most_airports[i]
    destination_airport = airport["ICAO"]
    #crear listas para guardar la información, así como un diccionario para toda la información del aeropuerto
    airports_list = lt.newList()
    edges_list = lt.newList()
    path_dict = {}
    #verificar que haya camino entre el aeropuerto de mayor concurrencia y el aeropuerto actual del recorrido.
    if djk.hasPathTo(search_graph, destination_airport): #si hay camino

        #generar el camino entre los aeropuertos
        path = djk.pathTo(search_graph, destination_airport)
        #calcular la distancia entre los aeropuertos
        total_dist = djk.distTo(search_graph, destination_airport)
        #recorrer los arcos del camino para llegar al aeropuerto de destino
        for edge in lt.iterator(path):
            #declarar aeropuerto de destino y origen
            org_airport = edge["vertexA"]
            dest_airport = edge["vertexB"]
            #generar el diccionario que contiene la información del arco
            edge_info = get_airports_info_req1_2(airports_map, org_airport, dest_airport, 1, 1)
            #añadir el diccionario a la lista de arcos
            lt.addFirst(edges_list, edge_info)
            if dest_airport == destination_airport:
                #si el aeropuerto de destino actual es el de destino global, añadir su info
                #a la lista de aeropuertos. Simplifica el proceso de añadir esta información
                #porque, añadir ambos generaría duplicados.
                dest_airport_info = get_single_airport_info(airports_map, dest_airport)
                lt.addFirst(airports_list, dest_airport_info)
            #generar el diccionario con la info del aeropuerto de origen
            org_airport_info = get_single_airport_info(airports_map, org_airport)
            #añadir el diccionario del origen a la lista
            lt.addFirst(airports_list, org_airport_info)
        #actualizar el diccionario del aeropuerto con la información necesaria
        path_dict["ICAO"]=destination_airport
```

```
            edge_info = get_airports_info_req1_2(airports_map, org_airport, dest_airport, 1, 1
            #añadir el diccionario a la lista de arcos
            lt.addFirst(edges_list, edge_info)
            if dest_airport == destination_airport:
                #si el aeropuerto de destino actual es el de destino global, añadir su info
                #a la lista de aeropuertos. Simplifica el proceso de añadir esta información
                #porque, añadir ambos generaría duplicados.
                dest_airport_info = get_single_airport_info(airports_map, dest_airport)
                lt.addFirst(airports_list, dest_airport_info)
            #generar el diccionario con la info del aeropuerto de origen
            org_airport_info = get_single_airport_info(airports_map, org_airport)
            #añadir el diccionario del origen a la lista
            lt.addFirst(airports_list, org_airport_info)
        #actualizar el diccionario del aeropuerto con la información necesaria
        path_dict["ICAO"]=destination_airport
        path_dict["NOMBRE"]= airport["NOMBRE"]
        path_dict["concurrency"]=airport["com_count"]
        path_dict["airports_list"] = airports_list
        path_dict["edges_list"]=edges_list
        path_dict["total_dist"] = total_dist
        path_dict["total_airports"] = lt.size(path)+1
        #añadir el diccionario del aeropuerto a la lista que entregará el requerimiento
        lt.addLast(airports_dict_lst, path_dict)
    else: #no hay camino
        #se añade un -1 para generalizar el error e imprimir lo adecuado en consola.
        lt.addLast(airports_dict_lst, -1)
return airports_dict_lst, max_airport
```

Descripción

La función se encarga de entregar el aeropuerto con mayor concurrencia comercial, así como los caminos de menor distancia para llegar a los 'num' aeropuertos con mayor concurrencia comercial (excluyendo el de mayor). Para cada uno de estos aeropuertos se debe entregar los aeropuertos que se visitan en el camino, los vuelos que se deben tomar para llegar y la distancia total en kilómetros del camino.

Entrada	Número de aeropuertos más importantes que se quieren cubrir desde el de mayor importancia (num)
Salidas	Tupla que contiene la lista con diccionarios que tienen la información de cada uno de los aeropuertos analizados, así como el diccionario con la información del aeropuerto más concurrido comercialmente
Implementado (Sí/No)	Sí, implementado por Santiago Muñoz y Cesar Espinosa

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo. Dada la complejidad del algoritmo, se hará un desglose más detallado de los pasos, pues efectivamente hay recorridos anidados de cuatro niveles.

Pasos	**Complejidad**

1. Acceder al grafo de vuelos comerciales por distancia, la tabla de hash de los aeropuertos y el árbol binario ordenado de aeropuertos comerciales	O(1), asignación simple
2. Obtener la lista de los 'num' aeropuertos más importantes del país.	O(log(v)*(mlog(m)), donde v es el número de aeropuertos registrados con vuelos comerciales y m es el tamaño promedio de una lista en el árbol binario ordenado. Esto pues, para conseguir la lista, se utiliza la función 'get_most_count_tree_colombia' para recorrer el árbol ordenado accedido anteriormente. Esta función accede a la llave máxima del árbol (un conteo) y a su lista asociada como valor (aeropuertos con esa concurrencia), la cual ordena alfabéticamente. Luego, recorre la lista y añade sus elementos a una lista de retorno hasta que se acabe la lista o se haya cumplido el num. Si se acaba la lista y todavía faltan elementos, se pasa a la siguiente menor llave, repitiendo el proceso hasta que se cumpla el num requerido. Esto implica acceder a un elemento de un árbol (log(v), pues en el peor caso hay tantas entradas en el árbol como aeropuertos) y por cada uno ordenar su lista (mlog(m)). Sin embargo, se pudo observar que, en general, m casi siempre es 1 (difícil que dos aeropuertos tengan la misma concurrencia) , por lo que el costo tendería más a parecerse a log(v) únicamente.
3. Generar el grafo de búsqueda mediante algoritmo de Dijkstra	O(ElogV) donde V es el número de aeropuertos en el grafo y E el número de arcos, implementación del algoritmo de DISCLib.
4. Recorrer la lista de los aeropuertos más importantes	O(num), recorrido sencillo
4. 1. Revisar si existe un camino entre el aeropuerto de origen y el de destino	O(1), revisar si el aeropuerto de destino está en la lista de adyacencia del aeropuerto de origen.
4.2. Generar y recorrer el camino entre aeropuerto de origen y destino.	O(1), pues, dada una dotación inicial de datos, el camino de menor peso entre dos vértices solo podría cambiar si se añaden elementos que permitan dicha conexión con un menor costo. Esto implica que el camino en general se mantiene igual, por lo que recorrerlo y reconstruirlo no depende del

	número de elementos en carga una vez se ha hecho la estructura de búsqueda.
TOTAL	***O(ElogV), pues lo más demorado resulta ser generar el grafo de búsqueda bajo el algoritmo Dijkstra.***

Pruebas Requerimiento 6

- Máquina 1: Macbook Air M1 2020, 16 GB RAM, macOS Ventura 13.0
- Máquina 2: Acer Nitro Intel i5-10300H, 8GB Ram, Windows 11

Valores:

```
Ingrese el número de aeropuertos más importantes que desea cubrir: 10
```

Máquina	**Tiempo [ms]**
Máquina 1	24.997
Máquina 2	14.11

Requerimiento 7

```python
def req_7(database, origin, dest):
    """Función que soluciona el requerimiento 7.
    La función utiliza el algoritmo de Dijkstra para encontrar el camino con el menor tiempo de vuelo entre dos puntos
    turísticos (solo usa vuelos comerciales), primero revisando que dichos puntos estén suficientemente cerca a un aeropuerto registrado
    Tiene una lógica similar a la función que resuelve los requerimientos 1 y 2 (req_1_2), contando con modificaciones
    para manejar adecuadamente el retorno del algoritmo de Dijkstra con respecto al camino entre dos aeropuertos.

    Args:
        database (dict): Base de datos que contiene las estructuras del modelo
        origin (tuple): Tupla de coordenadas (lat, lon) del origen
        dest (tuple): Tupla de coordenadas (lat, lon) del destino

    Returns:
        tuple: Tupla con la lista de vuelos que se toman en el camino, la distancia y tiempo de vuelo total del camino,
        el número de aeropuertos visitados y la información de los aeropuertos de origen y destino globales.

    """
    #acceder a los grafos pertinentes
    commercial_graph_dist = database["dist_graph_com"]
    commercial_graph_time = database["time_graph_com"]
    #acceder a la tabla de hash con la info de los aeropuertos
    airports_map = database["airports_map"]
    #determinar aeropuertos más cercanos a las coordenadas de origen y destino
    origin_airport, origin_distance = check_coordinates(origin, database)
    destination_airport, dest_distance = check_coordinates(dest, database)

    if origin_distance <= 30 and dest_distance <= 30:#hay aeropuertos a menos de 30 km tanto del destino como del origen.
        #inicializar la estructura de búsqueda sobre el grafo con pesos de tiempo
        search_algo = djk
        search_graph_dist = search_algo.Dijkstra(commercial_graph_time, origin_airport)
        path_exists = search_algo.hasPathTo(search_graph_dist, destination_airport)
        if path_exists == True: #hay camino entre origen y destino
            #generar variables y listas para guardar la info
            airports_list = lt.newList()
            total_dist = origin_distance+dest_distance
            total_time = 0
            #generar el camino entre los dos aeropuertos, pila de disclib
            dist_path = search_algo.pathTo(search_graph_dist, destination_airport)
            #usando el tamaño de la pila, dar el total de aeropuertos.
```

```
    if path_exists == True: #hay camino entre origen y destino
        #generar variables y listas para guardar la info
        airports_list = lt.newList()
        total_dist = origin_distance+dest_distance
        total_time = 0
        #generar el camino entre los dos aeropuertos, pila de disclib
        dist_path = search_algo.pathTo(search_graph_dist, destination_airport)
        #usando el tamaño de la pila, dar el total de aeropuertos.
        total_airports = st.size(dist_path)+1
        origin_airport_full = ""
        destination_airport_full = ""

        #recorrer la lista de vuelos
        for edge in lt.iterator(dist_path):
            #declarar aeropuertos de origen y destino
            org_airport = edge["vertexA"]
            dest_airport = edge["vertexB"]
            flight_time = edge["weight"]
            #acceder a la conexión en el grafo de distancia (dijkstra se hace sobre el de tiempo)
            dist_edge = gr.getEdge(commercial_graph_dist, org_airport, dest_airport)
            #determinar distancia entre aeropuertos
            flight_distance = dist_edge["weight"]
            #sumar distancia y tiempo a los globales
            total_time += flight_time
            total_dist+= flight_distance
            #generar el diccionario con la información del vuelo entre aeropuertos.
            airports_info = get_airports_info_req1_2(airports_map, org_airport, dest_airport, flight_distance, flight_time)
            #generar la información para los aeropuertos de origen y destino globales si se cumple que el de origen/destino actual
            #es el de origen/destino global
            if org_airport == origin_airport:
                origin_airport_full = "{0} ({1})".format(airports_info["NOMBRE ORIGEN"], org_airport)
            if dest_airport == destination_airport:
                destination_airport_full = "{0} ({1})".format(airports_info["NOMBRE DESTINO"], dest_airport)
            #añadir la información del vuelo a la lista de retorno.
            lt.addFirst(airports_list, airports_info)

        return airports_list, total_dist, total_time, total_airports, origin_airport_full, destination_airport_full
    else:
        return -2, origin_airport, origin_distance, destination_airport, dest_distance
else:
    return -1, origin_airport, origin_distance, destination_airport, dest_distance
```

Descripción

La función se encarga de encontrar la secuencia de vuelos con el menor tiempo de vuelo entre dos pares coordenadas ingresadas. Para ello, se utilizan dos grafos de vuelos comerciales, uno con pesos de distancia y el otro con pesos de tiempo, siendo a este último al que se aplica el algoritmo de Dijkstra para generar la estructura de búsqueda apropiada y poder retornar la información pedida. La función primero revisa si hay aeropuertos a menos de 30 km de las coordenadas ingresadas. De ser así, genera un SPT desde el aeropuerto de origen identificado sobre el grafo de tiempo, revisa si en este hay un camino entre el origen y destino y, de ser así, accede a dicho camino. Por último, se recorre el camino, lo cual genera una lista de diccionarios con la información de los vuelos que se deben tomar, al mismo tiempo que se guarda información como tiempo y distancia total del camino.

Entrada	Base de datos que contiene las estructuras del modelo (database), coordenadas de origen y coordenadas de destino.
Salidas	Tupla con la lista de vuelos en el camino, la distancia y tiempos totales del camino, así como la información del aeropuerto de origen y destino. En caso de haber errores, como que las coordenadas están muy lejos de un aeropuerto o no hay un camino entre aeropuertos, se generan retornos de error que permiten gestionar la impresión en consola.
Implementado (Sí/No)	Si, implementado por Santiago Muñoz y Juan Eduardo Ballesteros

Análisis de complejidad

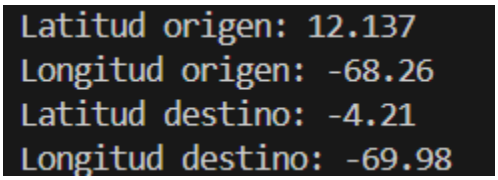
Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Acceder a los grafos y la tabla de hash necesarias	O(1), asignación simple
Revisar los aeropuertos más cercanos a las coordenadas de origen y destino ingresadas.	O(v), donde *v* corresponde al número de aeropuertos que hay en la carga. Esto pues la función 'check_coordinates' se implementó utilizando un recorrido completo, puesto que resultaba complicado generar esta busqueda de una manera más eficiente.
Generar el grafo de búsqueda a partir del grafo original	O(Elog(V)), donde E y V son la cantidad de arcos y vértices del grafo, respectivamente. Implementación del algoritmo de Dijkstra de DISCLib
Determinar si hay camino entre los aeropuertos	O(1), revisar si el aeropuerto de destino está en la lista de adyacencia del aeropuerto de origen en el grafo de búsqueda.
Reconstruir camino entre aeropuerto de origen y aeropuerto de destino.	O(1), pues, dada una dotación inicial de datos, el camino de menor peso entre dos vértices solo podría cambiar si se añaden elementos que permitan dicha conexión con un menor costo. Esto implica que el camino en general se mantiene igual, por lo que recorrerlo y reconstruirlo no depende del número de elementos en carga una vez se ha hecho la estructura de búsqueda.
Recorrer el camino entregado por el algoritmo Dijkstra para acceder a la información de cada vuelo	O(c), pues, aunque no se trata de una única operación y varía dependiendo de las entradas, se puede considerar que no depende de la cantidad de elementos en carga sino de las relaciones entre estos. Esto pues, puede que el camino menos costoso implique recorrer todos los arcos o solo uno, lo cual no depende de cuantos vértices o arcos haya en el grafo. Sin embargo, generalmente estos caminos tienden a ser cortos en escalas, lo cual aproxima su complejidad a una constante baja.
TOTAL	***O(Elog(V)),* pues generar la estructura para encontrar el camino menos costoso resulta lo más demorado de todos los pasos planteados.**

Pruebas Requerimiento 7

- Máquina 1: Macbook Air M1 2020, 16 GB RAM, macOS Ventura 13.0
- Máquina 2: Acer Nitro Intel i5-10300H, 8GB Ram, Windows 11

Valores:



Máquina	**Tiempo [ms]**
Máquina 1	24.877
Máquina 2	16.993

Requerimiento 8

Por la naturaleza del requerimiento 8, este no tiene una estructura fija, sino que, en vez, se basa en los datos que llegan de cada requerimiento para adicionar los marcadores y conexiones al mapa de folium. Por ende, resulta difícil hacer un análisis de complejidad temporal de este requerimiento. En general, para todos los requerimientos en el view se reciben los datos de coordenadas empacados junto a los datos de los aeropuertos. Estos se utilizan para generar los marcadores de cada aeropuerto y, si están conectados, se genera una línea entre las dos coordenadas.