ANÁLISIS DEL RETO

Mattia Riccardi, 202321259, m.riccardi@uniandes.edu.co

Miguel Santiago Roa Vallejo, 202322288, ms.roa@uniandes.edu.co

Laura Sofia Sarmiento, 202113056, l.sarmientog@uniandes.edu.co

Para el análisis del tiempo de ejecución y memoria utilizada, se acortó a criterio el documento CSV de la siguiente forma:

Airports 100%: 428 elementos Flights 100%: 3021

Airports 70%: 299 elementos Flights 70%: 2115

Airports 50%: 214 elementos Flights 50%: 1510

Airports 30%: 128 elementos Flights 30%: 906

Airports 10%: 43 elementos Flights 10%: 302

Carga de Datos

Descripción

Implementado (Sí/No)	Si: Miguel Roa.

En el view:

```
def load_data(control):
    """
    Carga los datos
    """
    #TODO: Realizar la carga de datos
    flights, airports = controller.load_data(control)
    return flights, airports
```

Descripción

Entrada	Control: va a ser la instancia del controlador para poder cargar los datos a partir de una estructura que se encuentra en el model llamada new_data_structs
Salidas	Flights, airports Flights es la cantidad de vuelos existentes, y airports es la cantidad de aeropuertos cargados

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad** **M: aeropuertos** **T: vuelos**
Controller.load_data()	Como tal la complejidad de este algoritmo depende de la función load_data() del controller. Como se explica posteriormente, su complejidad es: Temporal: O(m) + O(t) ya que una sola vez va a recorrer la información del csv y la irá agregando. Espacial: O(m) + O(t) Note que en realidad son varios O(m) (ya que cada estructura de datos es distinta (por ejemplo, una tabla de Hash o un grafo), sin embargo, todas guardan siempre la misma cantidad de información que en este caso se va a denotar m y t).
TOTAL	***Espacial:*** ***O(m) + O(t)*** ***Temporal:*** ***O(m) + O(t)***

En el controller:

```python
def load_data(control):
    """
    Carga los datos del reto
    """
    # TODO: Realizar la carga de datos
    data_structs = control["model"]

    airports = load_airports(data_structs)
    flights = load_flights(data_structs)
    cantidad_aeropuertos = flights[0]
    cantidad_vuelos = flights[1]

    return cantidad_aeropuertos, cantidad_vuelos

def load_flights(data_structs):
    name_file = cf.data_dir + "fligths-2022.csv"
    input_file = csv.DictReader(open(name_file, encoding="utf-8"), delimiter=";")

    #conteo_vuelos = 0
    for vuelos in input_file:
        model.add_flight(data_structs, vuelos)
        #conteo_vuelos += 1
    return model.graf_size(data_structs)

def load_airports(data_structs):
    name_file = cf.data_dir + "airports-2022.csv"
    input_file = csv.DictReader(open(name_file, encoding="utf-8"), delimiter=";")

    #conteo_aeropuertos = 0
    for aeropuertos in input_file:
        model.add_airport(data_structs, aeropuertos)
        #conteo_aeropuertos += 1
    return None

def mostrar_info(control):
    l1, l2, l3 = model.mostrar_info(control)
    return l1, l2, l3
```

Descripción

Entrada	Control: va a ser la instancia del controlador para poder cargar los datos a partir de una estructura que se encuentra en el model llamada new_data_structs
Salidas	Cantidad_aeropuertos, cantidad_vuelos

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad** **M: aeropuertos** **T: vuelos**
Load_airports()	Temporal: O(n) Espacial: O(v) Note que load_airports hace una iteración por todos los elementos del CSV, por eso sería O(n)

Load_flights()	Temporal: O(n) Espacial: O(E) Note que load_flights hace una iteración por todos los elementos del CSV, por eso sería O(n)
TOTAL	***Espacial:*** ***O(E + V)*** ***Temporal:*** ***O(N)***

En el model:

```python
def new_data_structs():
    """
    Inicializa las estructuras de datos del modelo. Las crea de
    manera vacía para posteriormente almacenar la información.
    """
    #TODO: Inicializar las estructuras de datos
    data_structs = {
        "flights": None, #va a ser una HashTable, donde las llaves van a ser un string donde está el ICAO-ICAO, siendo el primer ICAO el origen y el segundo ICAO el destino.
        #como valor va a tener un array_list con toda la información de los vuelos que tienen dicho trayecto.
        "airports_by_distance": None, #va a ser el grafo más general, donde los vértices serán los aeropuertos, y los arcos los trayectos que hay entre los aeropuertos.
        #Su peso va a ser la distancia en kilómetros
        "airpots_by_time": None, #va a ser el grafo más general, donde los vértices serán los aeropuertos, y los arcos los trayectos que hay entre los aeropuertos.
        #Su peso va a ser el tiempo del trayecto

        #los que están abajo es lo mismo que los de arriba, solo que únicamente van a estar los vuelos de cada tipo en particular. Note que en todos van a estar los mismos vértices.
        #es decir, que el tamaño de los grafos va a ser el mismo, lo único que va a cambiar son los arcos.
        "comercial_by_distance": None,
        "comercial_by_time": None,
        "militar_by_distance": None,
        "militar_by_time": None,
        "charge_by_distance": None,
        "charge_by_time": None,

        "airports": None, #este es el array con la info de todos los aeropuertos
        "airports_id": None,

        "airports_mayor_ocurrencia": None

    }

    data_structs["flights"] = mp.newMap(numelements= 4000, maptype="PROBING", loadfactor=0.6)
    data_structs["airports_by_distance"] = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=2000)
    data_structs["airports_by_time"] = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=2000)
    data_structs["comercial_by_distance"] = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=2000)
    data_structs["comercial_by_time"] = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=2000)
    data_structs["militar_by_distance"] = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=2000)
    data_structs["militar_by_time"] = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=2000)
    data_structs["charge_by_distance"] = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=2000)
    data_structs["charge_by_time"] = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=2000)
    data_structs["airports"] = lt.newList(datastructure="ARRAY_LIST")
    data_structs["airports_id"] = mp.newMap(numelements= 4000, maptype="PROBING", loadfactor=0.6)
    data_structs["airports_mayor_ocurrencia"] = mp.newMap(numelements= 2000, maptype="PROBING", loadfactor=0.6)

    return data_structs
```

Descripción

Entrada	None
Salidas	Data_structs

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad** **M: aeropuertos** **T: vuelos**
Mp.newMap() para los vuelos, los id de los aeropuertos, y los aeropuertos con mayor concurrencia	Espacial: O(m)
Gr.newGraph) par almacenar los aeropuertos que tienen como arcos los vuelos entre dichos lugares	Espacial: O(E + V), donde E sería la cantidad de vuelos, y V la cantidad de aeropuertos.
Lt.newList()	Espacial: O(n) ya que en el ARRAY_LIST se va a guardar los ICAO de todos los aeropuertos para que se puedan recorrer.
TOTAL	***Espacial:*** ***O(E + V) + O(n)***

```
def add_airport(data_structs, data):
    airport_id = data["ICAO"]
    gr.insertVertex(data_structs["airports_by_distance"], airport_id)
    gr.insertVertex(data_structs["airports_by_time"], airport_id)
    gr.insertVertex(data_structs["comercial_by_distance"], airport_id)
    gr.insertVertex(data_structs["comercial_by_time"], airport_id)
    gr.insertVertex(data_structs["militar_by_distance"], airport_id)
    gr.insertVertex(data_structs["militar_by_time"], airport_id)
    gr.insertVertex(data_structs["charge_by_distance"], airport_id)
    gr.insertVertex(data_structs["charge_by_time"], airport_id)
    lt.addLast(data_structs["airports"], data)
    mp.put(data_structs["airports_id"], airport_id, data)
```

Descripción

La función se encarga de añadir como vértices a todos los aeropuerto existentes. De igual forma, añade en un array todos los aeropuertos para posteriormente poder recorrerlos, y usa un mp.put para guardar en un mapa la información de los aeropuertos, según su id (es decir, según el ICAO).

Entrada	Data_structs, data, Donde data_structs es la gran estructura que tiene todo guardado, y data es cada línea del csv que se lee utilizando la librería csv.dictReader.
Salidas	None

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad** **M: aeropuertos** **T: vuelos**
Gr.insertVertex()	Espacial: O(V) Temporal: O(1)
Lt.addLast()	Espacial: O(m) Temporal: O(1)
Mp.put()	Espacial: O(n) Temporal: O(1)
TOTAL	***Espacial:*** ***O(V) + O(n)***

```python
def add_flight(data_structs, data):
    origen = data["ORIGEN"]
    destino = data["DESTINO"]
    distancia = calculo_distancia_haversine(data_structs, origen, destino)

    key = origen + "-" + destino
    entry = mp.get(data_structs["flights"], key)
    if entry is None:
        lista = lt.newList(datastructure="ARRAY_LIST")
        lt.addLast(lista, data)
        mp.put(data_structs["flights"], key, lista)
    else:
        lista_anterior = me.getValue(entry)
        lt.addLast(lista_anterior, data)
        mp.put(data_structs["flights"], key, lista_anterior)

    #este también va a agregar los arcos:

    gr.addEdge(data_structs["airports_by_distance"], origen, destino, distancia)
    gr.addEdge(data_structs["airports_by_time"], origen, destino, data["TIEMPO_VUELO"])

    #en caso de hacerlo con tuplas:
    #gr.addEdge(data_structs["airports_by_distance"], origen, destino, (distancia, data["TIEMPO_VUELO"]))

    if data["TIPO_VUELO"] == "AVIACION_CARGA":
        gr.addEdge(data_structs["charge_by_distance"], origen, destino, distancia)
        gr.addEdge(data_structs["charge_by_time"], origen, destino, data["TIEMPO_VUELO"])

        #gr.addEdge(data_structs["charge_by_distance"], origen, destino, (distancia, data["TIEMPO_VUELO"]))

    elif data["TIPO_VUELO"] == "MILITAR":
        gr.addEdge(data_structs["militar_by_distance"], origen, destino, distancia)
        gr.addEdge(data_structs["militar_by_time"], origen, destino, data["TIEMPO_VUELO"])

        #gr.addEdge(data_structs["militar_by_distance"], origen, destino, (distancia, data["TIEMPO_VUELO"]))

    elif data["TIPO_VUELO"] == "AVIACION_COMERCIAL":
        gr.addEdge(data_structs["comercial_by_distance"], origen, destino, distancia)
        gr.addEdge(data_structs["comercial_by_time"], origen, destino, data["TIEMPO_VUELO"])

        #gr.addEdge(data_structs["comercial_by_distance"], origen, destino, (distancia, data["TIEMPO_VUELO"]))
```

Descripción

La función se encarga de añadir como arcos a los vuelos que hay entre 2 aeropuertos

Entrada	Data_structs, data, Donde data_structs es la gran estructura que tiene todo guardado, y data es cada línea del csv que se lee utilizando la librería csv.dictReader.
Salidas	None

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad** **M: aeropuertos** **T: vuelos**

Gr.addEdge()	Espacial: O(E) Temporal: O(1)
Lt.addLast()	Espacial: O(m) Temporal: O(1)
Mp.put()	Espacial: O(n) Temporal: O(1)
TOTAL	***Espacial:*** ***O(E) + O(n)***

```python
def calculo_distancia_haversine(data_structs, origen, destino):
    """
    Esta función es para calcular la distancia entre un aeropuerto de
    origen, y una de destino.
    """
    aeropuertos_id = data_structs["airports_id"]
    info_1 = me.getValue(mp.get(aeropuertos_id, origen))
    info_2 = me.getValue(mp.get(aeropuertos_id, destino))

    lat_1 = math.radians(float(info_1["LATITUD"].replace(",", ".")))
    lat_2 = math.radians(float(info_2["LATITUD"].replace(",", ".")))
    lon_1 = math.radians(float(info_1["LONGITUD"].replace(",", ".")))
    lon_2 = math.radians(float(info_2["LONGITUD"].replace(",", ".")))

    primer_termino = math.sin((lat_2 - lat_1) / 2)**2 + math.cos(lat_1) * math.cos(lat_2) * math.sin((lon_2 - lon_1) / 2)**2
    segundo_termino = 2 * math.atan2(math.sqrt(primer_termino), math.sqrt(1 - primer_termino))

    return 6371.0 * segundo_termino

def calculo_distancia_haversine_posiciones(lat_1, lat_2, lon_1, lon_2):
    lat_1 = float(lat_1.replace(",", "."))
    #lat_2 = float(lat_2.replace(",", "."))
    lon_1 = float(lon_1.replace(",", "."))
    #lon_2 = float(lon_2.replace(",", "."))
    primer_termino = math.sin((lat_2 - lat_1) / 2)**2 + math.cos(lat_1) * math.cos(lat_2) * math.sin((lon_2 - lon_1) / 2)**2
    segundo_termino = 2 * math.atan2(math.sqrt(primer_termino), math.sqrt(1 - primer_termino))

    return 6371.0 * segundo_termino
```

Descripción

La función se encarga de añadir como arcos a los vuelos que hay entre 2 aeropuertos

Entrada	Data_structs, origen, destino Donde data_structs es la gran estructura que tiene todo guardado, y origen y destino es el ICAO de los aeropuertos
Salidas	Distancia_haversine

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad** **M: aeropuertos** **T: vuelos**

Mp.get()	Espacial: O(1) Temporal: O(1)
Math.radians()	Espacial: O(1) Temporal: O(1)
Math.sin() y math.cos()	Espacial: O(1) Temporal: O(1)
TOTAL	***Espacial:*** ***O(1)***

```python
def mostrar_info(control):
    aeropuertos = control["model"]["airports"]
    militares = control["model"]["militar_by_distance"]
    carga = control["model"]["charge_by_distance"]
    comercial = control["model"]["comercial_by_distance"]

    cantidad_militares = lt.newList("ARRAY_LIST")
    cantidad_carga = lt.newList("ARRAY_LIST")
    cantidad_comercial = lt.newList("ARRAY_LIST")

    for aero in lt.iterator(aeropuertos):
        id_a = aero["ICAO"]
        can_militares = float(gr.indegree(militares, id_a) + gr.outdegree(militares, id_a))
        can_carga = float(gr.indegree(carga, id_a) + gr.outdegree(carga, id_a))
        can_comercial = float(gr.indegree(comercial, id_a) + gr.outdegree(comercial, id_a))

        if can_militares != 0:
            lt.addLast(cantidad_militares, {"nombre": id_a, "cantidad": can_militares})
        if can_carga != 0:
            lt.addLast(cantidad_carga, {"nombre": id_a, "cantidad": can_carga})
        if can_comercial != 0:
            lt.addLast(cantidad_comercial, {"nombre": id_a, "cantidad": can_comercial})

    sa.sort(cantidad_militares, cmp_mayor_concurrencia)
    sa.sort(cantidad_comercial, cmp_mayor_concurrencia)
    sa.sort(cantidad_carga, cmp_mayor_concurrencia)

    añadir = control["model"]["airports_mayor_ocurrencia"]
    mp.put(añadir, "comercial", cantidad_comercial)
    mp.put(añadir, "carga", cantidad_carga)
    mp.put(añadir, "militar", cantidad_militares)


    return cantidad_militares, cantidad_comercial, cantidad_carga
```

Descripción

La función se encarga de mostrar la información en pantalla. De igual forma, se encarga de guardar en un map los aeropuertos con mayor concurrencia de carga, militar y comercial

Entrada	control
Salidas	Cantidad_militares, cantidad_comercial, cantidad_carga

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad** **M: aeropuertos** **T: vuelos**
Lt.addLast()	Espacial: O(n) Temporal: O(1)
Lt.newList()	Espacial: O(m) Temporal: O(1)
Mp.put()	Espacial: O(m) Temporal: O(1)
shellSort()	Temporal: O(n^1.25) En el peor caso: O(n^(3/2)) Espacial: O(1)
TOTAL	Temporal: O(n^1.25) En el peor caso: O(n^(3/2)) Espacial: O(1)

Pruebas Realizadas CARGA DE DATOS

Las pruebas realizadas fueron hechas en una máquina con las siguientes especificaciones:

Procesadores	**AMD Ryzen 3 3250U with Radeon Graphics 2.60 GHz**
Memoria RAM	8 GB
Sistema Operativo	Sistema operativo de 64 bits, procesador basado en x64 Windows 11

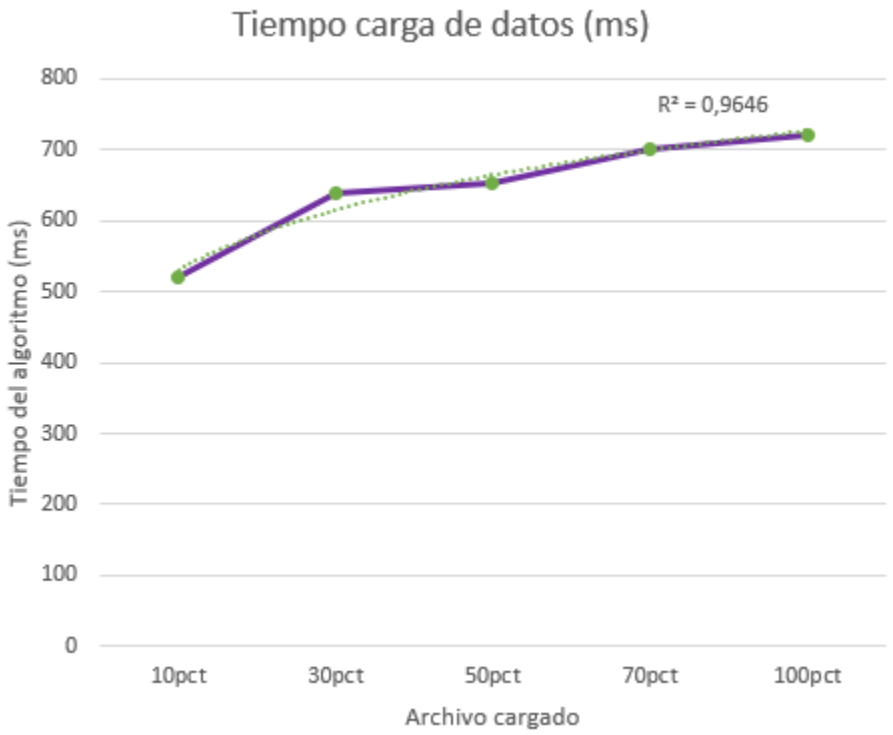
Tiempos:

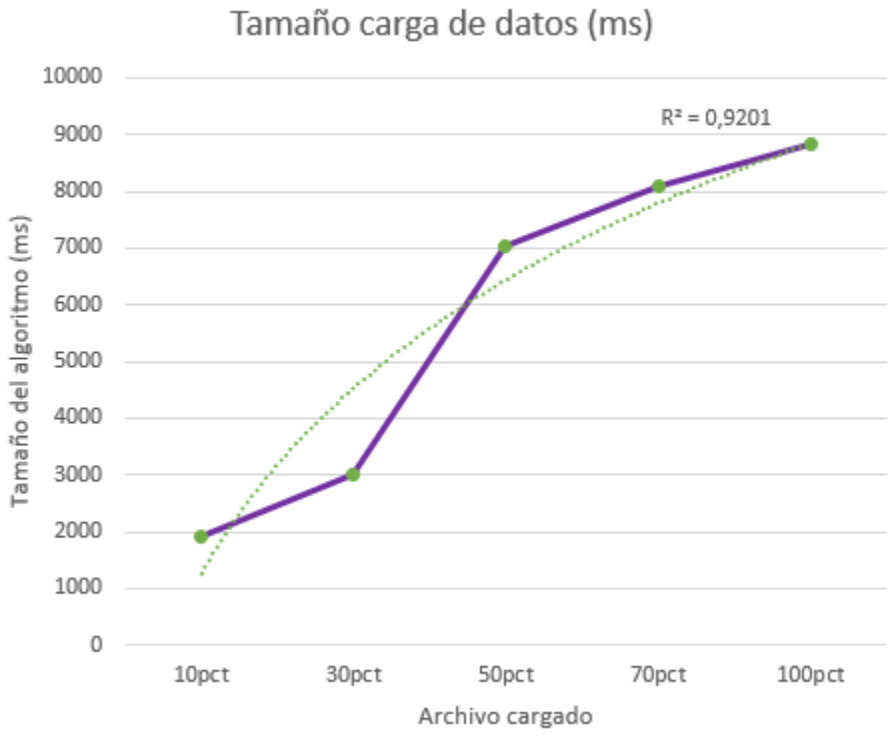
Entrada	**Tiempo (ms)**
10 pct	520,18
30 pct	640,03
50 pct	653,95
70pct	702,21
100pct	721,53

Memoria:

Entrada	**Memoria (Kbs)**
10 pct	1902,67
30 pct	3024,92
50 pct	7019,23
70pct	8109,32
100pct	8828,17

Graficas


Tiempo carga de datos (ms)
R² = 0,9646
Tiempo del algoritmo (ms)
800
700
600
500
400
300
200
100
0
10pct
30pct
50pct
70pct
100pct
Archivo cargado


Tamaño carga de datos (ms)
R² = 0,9201
Tamaño del algoritmo (ms)
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
10pct
30pct
50pct
70pct
100pct
Archivo cargado

Análisis

En primera instancia, puede observarse que a nivel temporal, la carga de datos se comporta como una función lineal. Esto puede deberse a que, justamente, entre más aumenta la cantidad de archivos, más tiempo se tarda al tener que recorrer más líneas. Note que justamente, la función que recorre el csv tiene complejidad O(n).

Por otro lado, el tamaño de la carga de datos tiene un gran salto cundo pasa de 30 a 50%. Esto puede deberse a que, justamente, entre más archivos se leen, el tamaño de los grafos crece y, recuerde, que en un grafo conforme crece el tamaño de los archivos, aumenta la cantidad de vértices V y árcos E. Note curiosamente que la complejidad del tamaño ocupado se parece mucho a un logaritmo, lo cual puede recordar a la complejidad de gran parte de los algoritmos que se utilizan (como Log (E+ V)). En conclusión, puede observarse que conforme crece el tamaño de los archivos, ambas complejidades aumentan con valores óptimos-esperados.

Requerimiento <<1>>

```
def req_1(data_structs, latitud_origen, longitud_origen, latitud_destino, longitud_destino):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    aeropuertos_comerciales_distance = data_structs["comercial_by_distance"]
    #print(aeropuertos_comerciales_distance)
    aeropuertos_comerciales_time = data_structs["comercial_by_time"]
    aeropuertos = data_structs["airports"]

    #print(aeropuertoss)

    cantidad_aeropuerto_visitados = 0
    lista_aeropuertos = lt.newList("ARRAY_LIST")
    tiempo_trayecto = 0
    tiempo_entre_trayectos = lt.newList("ARRAY_LIST")
    distancia_trayecto = 0

    if(encontrar_minimo(latitud_origen, longitud_origen, aeropuertos, True) == None or encontrar_minimo(latitud_destino, longitud_destino, aeropuertos, True) == None):
        ae_origen = None
        ae_destino = None
    else:
        ae_origen = encontrar_minimo(latitud_origen, longitud_origen, aeropuertos, True)[0]
        ae_destino = encontrar_minimo(latitud_destino, longitud_destino, aeropuertos, True)[0]
    #print(ae_destino)
    #print(ae_origen)
    if ae_origen == None or ae_destino == None:
        ae_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, aeropuertos, False)[0]
        distancia_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, aeropuertos, False)[1]
        return  ae_mas_cercano, distancia_mas_cercano, None, None, None, None, False
    else:
        search = bfs.BreathFirstSearch(aeropuertos_comerciales_distance, ae_origen)
        camino = bfs.pathTo(search, ae_destino)
        if camino != None:
            cantidad_aeropuerto_visitados = camino["size"]
            #print(camino) #que putasssss
            for x in lt.iterator(camino):
                lt.addFirst(lista_aeropuertos, x)
            #print(gr.getEdge(aeropuertos_comerciales_distance, "SKBO", "SKOT"))
            #print(lista_aeropuertos)
            for i in range(1, lt.size(lista_aeropuertos)):
                vertice_a = lt.getElement(lista_aeropuertos, i)
                vertice_b =  lt.getElement(lista_aeropuertos, i + 1)
                #print(vertice_a)
                #print(vertice_b)
                arco_distancia = gr.getEdge(aeropuertos_comerciales_distance, vertice_a, vertice_b)
                distancia = float(arco_distancia["weight"])
```

```
            distancia_trayecto += distancia
            arco_tiempo = gr.getEdge(aeropuertos_comerciales_time, vertice_a, vertice_b)
            tiempo = float(arco_tiempo["weight"])
            lt.addLast(tiempo_entre_trayectos, {vertice_a + "-" + vertice_b : tiempo})
    else:
        ae_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, aeropuertos, False)[0]
        distancia_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, aeropuertos, False)[1]
        return ae_mas_cercano, distancia_mas_cercano, None, None, None, None, None
    #print(tiempo_trayecto)
    #print(distancia_trayecto)

    #notese que lista_aeropuertos tiene todos los vértices (incluyendo el orien y destino, entonces tengo que quitarle el primero y el último)
    bono_req1(lista_aeropuertos, data_structs)
    return ae_origen, ae_destino, lista_aeropuertos, cantidad_aeropuerto_visitados, tiempo_entre_trayectos, distancia_trayecto, True
```

Descripción

Este requerimiento se encarga de determinar si hay un lugar turístico (comercial) entre 2 coordenadas propuestas por el usuario. De esta forma, el usuario va a digitar una coordenada: en caso de encontrar un aeropuerto cercano a menos de 30km de las coordenadas propuestas, la función se ejecutará. En caso en que encuentre un aeropuerto a menos de 30km, pero que no tenga una ruta al aeropuerto de destino, va a indicar que efectivamente, hay un aeropuerto cercano pero que no hay un camino hasta dicho lugar de interés.

Entrada	`data_structs, latitud_origen, longitud_origen, latitud_destino, longitud_destino` donde data_structs es la estructura donde se hizo la carga de datos y que tiene los distintos grafos almacenados. Latitud_origen y longitud_origen son las coordenadas propuestas por el usuario Latitud_destino y longitud_destino son las coordenadas dadas por el usuario a las que se quieren llegar.
Salidas	`ae_origen, ae_destino, lista_aeropuertos, cantidad_aeropuerto_visitados, tiempo_entre_trayectos, distancia_trayecto, True` ae_origen: es el aeropuerto más cercano a 30km ae_destino: es el aeropuerto destino que permite llegar al destino lista_aeropuertos: incluye todos los aeropuertos (incluso los intermedios-escala) para llegar al destino cantidad_aeropuerto_visitado: son la cantidad de aeropuertos por los cuales hay que pasar para llegar al destino tiempo_entre_trayectos: es una lista de diccionarios con el tiempo para cubrir 2 trayectos distancia_trayecto: es la distancia total del trayecto. Bool: es True en caso en que haya un camino; es False en caso de que no haya un camino; es None en caso en que no haya ni siquiera encontrado un aeropuerto cerca

Implementado (Sí/No)	Si: Miguel Roa.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Lt.newList() para crear y almacenar la lista de aeropuertos para concretar el trayecto, y almacenar el tiempo entre trayectos	Temporal: O(1) Espacial: O(m)
Encontrar_minimo() es una función creada de forma adicional, para encontrar el aeropuerto más cercano y que se encuentre a menos de 30km: este algoritmo tien distintos algoritmos, entre los cuales lt.newList(), lt.iterator(), lt.addLast() y un shellSort(). De esta forma, el que tiene mayor complejidad es el shellSort().	Temporal: O(n^1.25) En el peor caso: O(n^(3/2)) Espacial: O(1) Note que en este caso, la n depende del tamaño del grafo a recorrer.
Bfs.pathTo()	
Lt.iterator()	Temporal: O(n) Espacial: O(1) Note que en este caso, la n depende del número de aeropuertos que permiten cubrir la ruta propuesta.
Lt.getElement()	Temporal: O(1) ya que es un arrayList y se puede acceder directamente a la posición. Espacial: O(1)
Gr.getEdge()	O(1)
Lt.addLast()	Temporal: O(1) Espacial: O(1).
TOTAL	**Temporal: O(n^1.25)** **En el peor caso: O(n^(3/2))**

Pruebas Realizadas

Las pruebas realizadas fueron hechas en una máquina con las siguientes especificaciones:

Procesadores	**AMD Ryzen 3 3250U with Radeon Graphics 2.60 GHz**
Memoria RAM	8 GB
Sistema Operativo	Sistema operativo de 64 bits, procesador basado en x64 Windows 11

Tiempos:

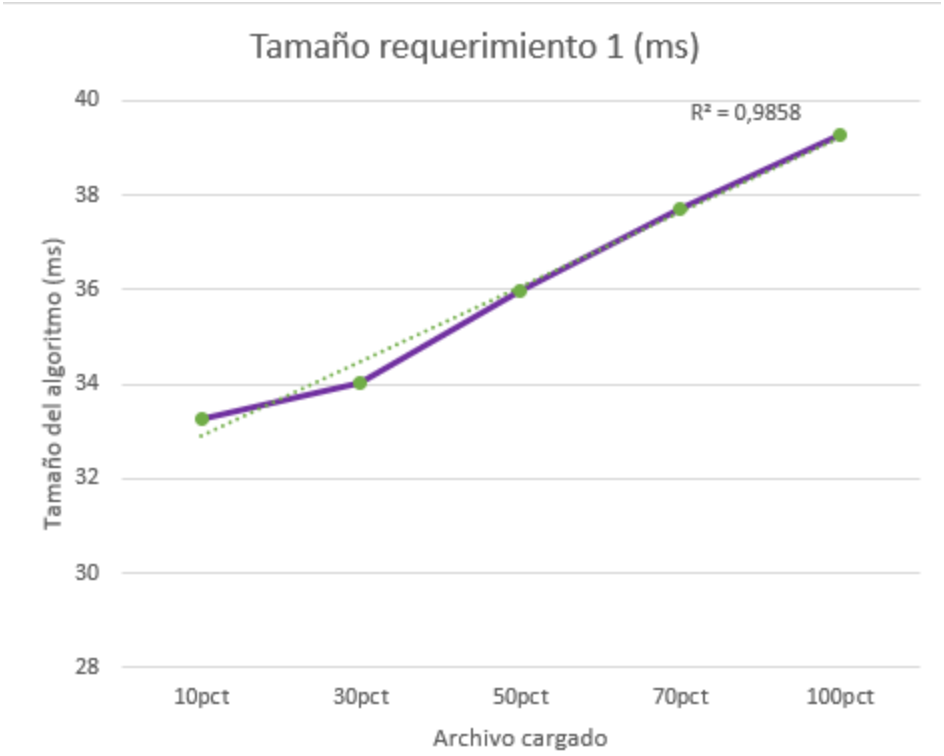
Entrada	**Tiempo (ms)**

10 pct	32,78
30 pct	34,29
50 pct	33,82
70pct	45,02
100pct	46,98

Memoria:

Entrada	**Memoria (Kbs)**
10 pct	33,25
30 pct	34,01
50 pct	35,99
70pct	37,72
100pct	39,29

Graficas



Análisis

Puede observarse que el tiempo de la función no es tan constante (no es regular), pues se pueden observar picos. Sin embargo, puede resumirse a una función lineal O(n). Esto significaría que efectivamente, el shellSort que se encarga de encontrar la cantidad mínima está en su mejor de los casos. Además, esto podría significar que, justamente, como el shellSort está trabajando con una cantidad menor de datos (ya que se encarga únicamente de organizar una lista según la concurrencia de cada aeropuerto), mientras que los algoritmos del requerimiento 1 trabajan con más datos.

Por otro lado, se observa que el tamaño ocupado por el requerimiento crece demasiado poco y que, además es lineal, lo cual indicaría que gran parte de la memoria se va en array_list o en complejidades lineales.

Requerimiento <<2>>

Model

App > model.py > req_2

```python
def req_2(data_structs, latitud_origen, longitud_origen, latitud_destino, longitud_destino):
    Funcion que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    gr_aeropuertos_comerciales_distance = data_structs["comercial_by_distance"]
    gr_aeropuertos_comerciales_time = data_structs["comercial_by_time"]
    lst_aeropuertos = data_structs["airports"]

    cantidad_aeropuerto_visitados = 0
    lista_aeropuertos = lt.newList("ARRAY_LIST")
    tiempo_entre_trayectos = lt.newList("ARRAY_LIST")
    distancia_trayecto = 0
    tiempo_trayecto_total = 0

    if(encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, True) == None or encontrar_minimo(latitud_destino, longitud_destino, lst_aeropuertos, True) == None):
        ae_origen = None
        ae_destino = None
    else:
        ae_origen = encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, True)[0]
        ae_destino = encontrar_minimo(latitud_destino, longitud_destino, lst_aeropuertos, True)[0]

    if ae_origen == None or ae_destino == None:
        ae_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, False)[0]
        distancia_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, False)[1]
        return  ae_mas_cercano, distancia_mas_cercano, None, None, None, None, False, None
    else:
        search = bfs.BreathFirstSearch(gr_aeropuertos_comerciales_distance, ae_origen)
        camino = bfs.pathTo(search, ae_destino)
        if camino != None:
            cantidad_aeropuerto_visitados = camino["size"]
            for x in lt.iterator(camino):
                lt.addFirst(lista_aeropuertos, x)
            for i in range(1, lt.size(lista_aeropuertos)):
                vertice_a = lt.getElement(lista_aeropuertos, i)
                vertice_b =  lt.getElement(lista_aeropuertos, i + 1)
                #print(vertice_a)
                #print(vertice_b)
                arco_distancia = gr.getEdge(gr_aeropuertos_comerciales_distance, vertice_a, vertice_b)
                distancia = float(arco_distancia["weight"])
                distancia_trayecto += distancia
                arco_tiempo = gr.getEdge(gr_aeropuertos_comerciales_time, vertice_a, vertice_b)
                tiempo = float(arco_tiempo["weight"])
                lt.addLast(tiempo_entre_trayectos, {vertice_a + "-" + vertice_b : tiempo})
                tiempo_trayecto_total += tiempo
        else:
            ae_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, False)[0]
            distancia_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, False)[1]
            return ae_mas_cercano, distancia_mas_cercano, None, None, None, None, None, None
```

Controller

```python
def req_2(control, latitud_origen, longitud_origen, latitud_destino, longitud_destino):
    """
    Retorna el resultado del requerimiento 2
    """
    # TODO: Modificar el requerimiento 2
    data_structs = control["model"]
    tiempo_inicial = get_time()
    ae_origen, ae_destino, lista_aeropuertos, cantidad_aeropuerto_visitados, tiempo_trayecto, distancia_trayecto, checking, tiempo_total_trayecto = model.req_2(data_structs, latitud_origen, longitud_origen, latitud_destino, longitud_destino)
    tiempo_final = get_time()
    tiempo_total = delta_time(tiempo_inicial, tiempo_final)
    return tiempo_total, ae_origen, ae_destino, lista_aeropuertos, cantidad_aeropuerto_visitados, tiempo_trayecto, distancia_trayecto, checking, tiempo_total_trayecto

```

View

```python
def print_req_2(control):
    """
        Función que imprime la solución del Requerimiento 2 en consola
    """
    # TODO: Imprimir el resultado del requerimiento 2
    latitud_origen = input("Digite la latitud del punto de origen: ")
    longitud_origen = input("Digite la longitud del punto de origen: ")
    latitud_destino = input("Digite la latitud del punto de destino: ")
    longitud_destino = input("Digite la longitud del punto de destino: ")
    memoria = input("Quiere ver la memoria almacenada? Diga True o False")
    tiempo_total, ae_origen, ae_destino, lista_aeropuertos, cantidad_aeropuerto_visitados, tiempo_entre_trayectos, distancia_trayecto, checking, tiempo_total_trayecto = controller.req_2(control, latitud_origen, longitud_origen, latitud_destino, longitud_destino)
    tiempo_total = round(float(tiempo_total), 2)
    print("El tiempo total de ejecución fue de: " + str(tiempo_total) + " ms.")
    if checking == False:
        print("")
        print("No se encontró un aeropuerto cercano a menos de 30km desde la ubicación ofrecida.")
        ae_destino = round(float(ae_destino), 2)
        print("Sin embargo, el aeropuerto más cercano fue " + str(ae_origen) + str(" a una distancia de ") + str(ae_destino))
    elif checking == None:
        print("")
        print("Se encontró un aeropuerto a una distancia menor de 30km.")
        print("Sin embargo, no hay un camino que lleve desde dicho diccionario hasta la posición de destino. ")
        round(float(ae_destino), 2)
        print("El aeropuerto más cercano fue " + str(ae_origen) + str(" a una distancia de ") + str(ae_destino))
    elif checking == True:
        print("")
        distancia_trayecto = round(float(distancia_trayecto), 2)
        print("La distancia total para ir de ambos lugares es: " + str(distancia_trayecto) + "km. ")
        print("El tiempo total del trayecto es de: " + str(tiempo_total_trayecto) + " minutos")
        print("La cantidad de aeropuertos visitados fue de: " + str(cantidad_aeropuerto_visitados))
        mostrar_req2(control, lista_aeropuertos, ae_origen, ae_destino)

        print("\nTiempo entre trayectos: \n")
        for z in lt.iterator(tiempo_entre_trayectos):
            print(z)

```

```python
def mostrar_req2(control, lst, ae_origen, ae_destino):
    aeropuertos_id = control["model"]["airports_id"]
    lista = []
    resultado = []

    #para el de origen
    print("\n Aeropuerto de origen: \n")
    icao = me.getValue(mp.get(aeropuertos_id, ae_origen))["ICAO"]
    nombre = me.getValue(mp.get(aeropuertos_id, ae_origen))["NOMBRE"]
    ciudad = me.getValue(mp.get(aeropuertos_id, ae_origen))["CIUDAD"]
    pais = me.getValue(mp.get(aeropuertos_id, ae_origen))["PAIS"]
    resultado = [icao, nombre, ciudad, pais]
    print(tabulate([resultado], headers=["ICAO", "NOMBRE", "Ciudad", "PAIS"], tablefmt="rounded_grid"))

    print("\n Aeropuertos intermedios: ")
    lista = []
    resultado = []
    for x in lt.iterator(lst):
        if ((x != ae_origen) and (x != ae_destino)):
            icao = me.getValue(mp.get(aeropuertos_id, x))["ICAO"]
            nombre = me.getValue(mp.get(aeropuertos_id, x))["NOMBRE"]
            ciudad = me.getValue(mp.get(aeropuertos_id, x))["CIUDAD"]
            pais = me.getValue(mp.get(aeropuertos_id, x))["PAIS"]
            resultado = [icao, nombre, ciudad, pais]
            lista.append(resultado)

    print(tabulate(lista, headers=["ICAO", "NOMBRE", "Ciudad", "PAIS"], tablefmt="rounded_grid"))

    print("\n Aeropuerto de destino: \n")
    lista = []
    resultado = []
    icao = me.getValue(mp.get(aeropuertos_id, ae_destino))["ICAO"]
    nombre = me.getValue(mp.get(aeropuertos_id, ae_destino))["NOMBRE"]
    ciudad = me.getValue(mp.get(aeropuertos_id, ae_destino))["CIUDAD"]
    pais = me.getValue(mp.get(aeropuertos_id, ae_destino))["PAIS"]
    resultado = [icao, nombre, ciudad, pais]
    print(tabulate([resultado], headers=["ICAO", "NOMBRE", "Ciudad", "PAIS"], tablefmt="rounded_grid"))

```

Descripción

El Requerimiento No. 2 tiene como objetivo encontrar el itinerario con el menor número de escalas entre dos puntos turísticos, especificados por latitud y longitud, utilizando aeropuertos comerciales que se encuentren a menos de 30 km de estos puntos. El proceso implica aproximar los puntos ingresados al aeropuerto más cercano utilizando la distancia Haversine y, si no se encuentran aeropuertos dentro del rango, se indica el aeropuerto más cercano y su distancia. La respuesta debe incluir el tiempo de ejecución del algoritmo, la distancia total del trayecto (incluyendo los segmentos desde los puntos hasta los aeropuertos y entre los aeropuertos), el número de aeropuertos visitados y una secuencia detallada del trayecto con información de los aeropuertos y el tiempo del trayecto.

Entrada	• Punto de origen (una localización geográfica con latitud y longitud). • Punto de destino (una localización geográfica con latitud y longitud).Latitud_origen y longitud_origen son las coordenadas propuestas por el usuario

Salidas	• El tiempo que se demora algoritmo en encontrar la solución (en milisegundos). • La distancia total que tomará el camino entre el punto de origen y el de destino (incluye la distancia del origen al aeropuerto origen, la distancia del trayecto entre aeropuertos y la distancia del aeropuerto destino al destino). • El número de aeropuertos que se visitan en el camino encontrado. • Del camino encontrado presente en la secuencia del trayecto completo con la siguiente información: ○ Aeropuerto de origen (identificador ICAO, nombre, ciudad, país) ○ Secuencia de aeropuertos intermedios (identificador ICAO, nombre, ciudad, país) ○ Aeropuerto de destino (identificador ICAO, nombre, ciudad, país) ○ Tiempo del trayecto.
Implementado (Sí/No)	Si: Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Lt.newList() para crear y almacenar la lista de aeropuertos para concretar el trayecto, y almacenar el tiempo entre trayectos	Temporal: O(1) Espacial: O(m)
Encontrar_minimo() es una función creada de forma adicional, para encontrar el aeropuerto más cercano y que se encuentre a menos de 30km: este algoritmo tien distintos algoritmos, entre los cuales lt.newList(), lt.iterator(), lt.addLast() y un shellSort(). De esta forma, el que tiene mayor complejidad es el shellSort().	Temporal: O(n^1.25) En el peor caso: O(n^(3/2)) Espacial: O(1) Note que en este caso, la n depende del tamaño del grafo a recorrer.
Bfs.pathTo()	
Lt.iterator()	Temporal: O(n) Espacial: O(1) Note que en este caso, la n depende del número de aeropuertos que permiten cubrir la ruta propuesta.
Lt.getElement()	Temporal: O(1) ya que es un arrayList y se puede acceder directamente a la posición. Espacial: O(1)
Gr.getEdge()	O(1)
Lt.addLast()	Temporal: O(1)

	Espacial: O(1).
TOTAL	**Temporal: O(n^1.25)** **En el peor caso: O(n^(3/2))**

Pruebas Realizadas

Las pruebas realizadas fueron hechas en una máquina con las siguientes especificaciones:

Procesadores	Apple M2 8 núcleos (4 de rendimiento 3.5 GHz y 4 de eficiencia 2.2 GHz)
Memoria RAM	16 GB
Sistema Operativo	macOS Sonoma Versión 14.0

Tiempos:

Entrada	**Tiempo (ms)**
10 pct	4.39
30 pct	4.46
50 pct	5.14
70pct	5.24
100pct	5.42

Memoria:

Entrada	**Memoria (Kbs)**
10 pct	4.16
30 pct	4.98
50 pct	5.14
70pct	5.18
100pct	5.28

Análisis

El análisis de complejidad del Requerimiento No. 2, que busca encontrar el itinerario con el menor número de escalas entre dos puntos turísticos especificados por latitud y longitud, utilizando aeropuertos comerciales a menos de 30 km de estos puntos, se centra en diversos pasos clave del algoritmo. La creación y almacenamiento de listas de aeropuertos y tiempos de trayecto tienen una complejidad temporal de O(1) y espacial de O(m). La función adicional Encontrar_minimo(), que encuentra el aeropuerto más cercano, incluye algoritmos como lt.newList(), lt.iterator(), lt.addLast(), y shellSort(), siendo este último el más complejo con una temporalidad de O(n^1.25) en promedio y O(n^(3/2)) en el peor caso, con una espacialidad de O(1). La búsqueda de rutas (Bfs.pathTo()) y la iteración (lt.iterator()) tienen una complejidad temporal de O(n) y espacial de O(1), dependiendo del número de aeropuertos. Acciones como lt.getElement() y Gr.getEdge() son de complejidad O(1). En conjunto, el algoritmo tiene una complejidad temporal total de O(n^1.25) en promedio y O(n^(3/2)) en el peor caso, con pruebas realizadas en una máquina Apple M2 que mostraron tiempos de ejecución y uso de memoria eficientes.

Requerimiento <<3>>

Model

App > model.py > req_3

```python
def req_3(data_structs):
    # TODO: Realizar el requerimiento 3
    ocurrencia = data_structs["airports_mayor_ocurrencia"]
    grafo_comercial_distancia = data_structs["comercial_by_distance"]
    grafo_comercial_tiempos = data_structs["comercial_by_time"]

    flights = data_structs["flights"]

    lst_comercial = me.getValue(mp.get(ocurrencia, "comercial"))
    primer_elemento = lt.firstElement(lst_comercial)["nombre"]
    cantidad_ocurrencias = lt.firstElement(lst_comercial)["cantidad"]

    search = prim.PrimMST(graph=grafo_comercial_distancia, origin=primer_elemento)
    table_edge_to = search["edgeTo"] #es una hashTable
    lista_aeropuertos_a_los_que_hay_camino = mp.keySet(table_edge_to)
    lst = lt.newList("ARRAY_LIST")

    for x in lt.iterator(lista_aeropuertos_a_los_que_hay_camino):
        lt.addLast(lst, x)

    nuevo_grafo_a_partir_del_mst = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=1000)
    for y in lt.iterator(lst):
        value = me.getValue(mp.get(table_edge_to, y))
        key_a = value["vertexA"]
        key_b = value["vertexB"]
        weight_asc = value["weight"]
        if gr.containsVertex(nuevo_grafo_a_partir_del_mst, key_a) == False:
            gr.insertVertex(nuevo_grafo_a_partir_del_mst, key_a)
        if gr.containsVertex(nuevo_grafo_a_partir_del_mst, key_b) == False:
            gr.insertVertex(nuevo_grafo_a_partir_del_mst, key_b)

        if gr.getEdge(grafo_comercial_tiempos, key_a, key_b):
            gr.addEdge(nuevo_grafo_a_partir_del_mst, key_a, key_b, weight_asc)

    #Recorrido desde el aereopuerto_de_mayor concurrencia hasta todas sus posibles conexiones

    search_2 = bfs.BreathFirstSearch(nuevo_grafo_a_partir_del_mst, primer_elemento)
    respuesta = lt.newList("ARRAY_LIST")
    for destiny in lt.iterator(lst):
        if destiny != primer_elemento:
            distancia = 0
            distancia_trayecto = 0
            tiempo = 0
            tiempo_trayecto = 0
            camino = bfs.pathTo(search_2, destiny)
            lst_aero = lt.newList("ARRAY_LIST")
            aeronaves_str = ""
            check_aeronaves = mp.newMap(maptype="PROBING")
            for y in lt.iterator(camino):
                lt.addFirst(lst_aero, y)
            for i in range(1, lt.size(lst_aero)):
                vertice_a = lt.getElement(lst_aero, i)
                vertice_b =  lt.getElement(lst_aero, i + 1)
                arco_distancia = gr.getEdge(nuevo_grafo_a_partir_del_mst, vertice_a, vertice_b)
                distancia = float(arco_distancia["weight"])

                distancia_trayecto += distancia
                arco_tiempo = gr.getEdge(grafo_comercial_tiempos, vertice_a, vertice_b)
                tiempo = float(arco_tiempo["weight"])
                tiempo_trayecto += tiempo

                key_flight = str(vertice_a) + "-" + str(vertice_b)
                aeronave = me.getValue(mp.get(flights, key_flight))["elements"][0]["TIPO_AERONAVE"]

                if mp.contains(check_aeronaves, aeronave) == False:
                    aeronaves_str = aeronaves_str + " - " + aeronave
                    mp.put(check_aeronaves, aeronave, aeronave)

            lt.addLast(respuesta,
                    {"aeropuerto origen": primer_elemento,
                        "aeropuerto destino": destiny,
                        "distancia recorrida": distancia_trayecto,
                        "tiempo trayecto": tiempo_trayecto,
                        "aeronaves": aeronaves_str})
    bono_req3(respuesta, data_structs)
    cantidad_tr = lt.size(respuesta)
    return primer_elemento, cantidad_ocurrencias, cantidad_tr, respuesta
```

Controller

```python
def req_3(control):
    """
    Retorna el resultado del requerimiento 3
    """
    # TODO: Modificar el requerimiento 3
    data_structs = control["model"]
    tiempo_inicial = get_time()
    aeropuerto_mayor_ocurrencias, cantidad_ocurrencias, cantidad_trayectos, respuesta = model.req_3(data_structs)
    tiempo_final = get_time()
    tiempo_total = delta_time(tiempo_inicial, tiempo_final)
    return tiempo_total, aeropuerto_mayor_ocurrencias, cantidad_ocurrencias, cantidad_trayectos, respuesta
```

View

```python
def print_req_3(control):
    """
        Función que imprime la solución del Requerimiento 3 en consola
    """
    # TODO: Imprimir el resultado del requerimiento 3
    aeropuertos_id = control["model"]["airports_id"]
    memoria = input("Quiere ver la memoria almacenada? Diga True o False")
    tiempo_total, aeropuerto_mayor_ocurrencias, cantidad_ocurrencias, cantidad_trayectos, respuesta = controller.req_3(control)
    tiempo_total = round(float(tiempo_total), 2)
    print("El tiempo total de ejecución fue de: " + str(tiempo_total) + " ms.")

    print("El aeropuerto más importante según la concurrencia comercial fue: ")

    icao = me.getValue(mp.get(aeropuertos_id, aeropuerto_mayor_ocurrencias))["ICAO"]
    nombre = me.getValue(mp.get(aeropuertos_id, aeropuerto_mayor_ocurrencias))["NOMBRE"]
    ciudad = me.getValue(mp.get(aeropuertos_id, aeropuerto_mayor_ocurrencias))["CIUDAD"]
    pais = me.getValue(mp.get(aeropuertos_id, aeropuerto_mayor_ocurrencias))["PAIS"]
    resultado = [icao, nombre, ciudad, pais, cantidad_ocurrencias]
    print(tabulate([resultado], headers=["ICAO", "NOMBRE", "Ciudad", "PAIS", "CANTIDAD OCURRENCIAS"], tablefmt="rounded_grid"))

    print("El número total de trayectos es: " + str(cantidad_trayectos))

    print("\n A continuación, la secuencia de trayectos encontrados: \n")

    mostrar_req3(control["model"], respuesta)

def mostrar_req3(data_structs, lst):
    aeropuertos_id = data_structs["airports_id"]

    lista = []
    resultado = []
    distancia_recorrida_total = 0
    for x in lt.iterator(lst):
        aeropuerto_origen = x["aeropuerto origen"]
        nombre_origen = me.getValue(mp.get(aeropuertos_id, aeropuerto_origen))["NOMBRE"]
        pais_origen = me.getValue(mp.get(aeropuertos_id, aeropuerto_origen))["PAIS"]
        ciudad_origen = me.getValue(mp.get(aeropuertos_id, aeropuerto_origen))["CIUDAD"]
        string_origen = aeropuerto_origen + " - " + nombre_origen + " - " + pais_origen + " - " + ciudad_origen
        aeropuerto_destino = x["aeropuerto destino"]
        nombre_destino = me.getValue(mp.get(aeropuertos_id, aeropuerto_destino))["NOMBRE"]
        pais_destino = me.getValue(mp.get(aeropuertos_id, aeropuerto_destino))["PAIS"]
        ciudad_destino = me.getValue(mp.get(aeropuertos_id, aeropuerto_destino))["CIUDAD"]
        string_destino = aeropuerto_destino + " - " + nombre_destino + " - " + pais_destino + " - " + ciudad_destino


        distancia_recorrida = round(float(x["distancia recorrida"]), 2)
        distancia_recorrida_total += float(x["distancia recorrida"])
        tiempo_trayecto = x["tiempo trayecto"]
        aeronaves = x["aeronaves"]

        resultado = [string_origen, string_destino, distancia_recorrida, tiempo_trayecto]
        lista.append(resultado)

    print(tabulate(lista, headers=["Origen - Nombre Origen - Pais Origen - Ciudad Origen", "Destino - Nombre Destino - Pais destino - Ciudad Destino", "Distancia Recorrida", "Duración"], tablefmt="rounded_grid"))
    print("")
    print("La suma de las distancias de los trayectos es: " +str(round(distancia_recorrida_total, 3)))
```

Descripción

El Requerimiento No. 3 tiene como objetivo determinar una red de trayectos comerciales desde el aeropuerto con mayor concurrencia comercial para maximizar la cobertura de aeropuertos con la menor distancia recorrida. Para identificar la importancia de un aeropuerto, se considera el número total de vuelos comerciales que llegan y salen de él. En caso de empate en la concurrencia, se utiliza el código ICAO para la comparación alfabética. El requerimiento no recibe parámetros de entrada y debe identificar automáticamente el aeropuerto más concurrido. La respuesta debe incluir el tiempo de ejecución del algoritmo, la identificación del aeropuerto más importante, la suma de la distancia total de los trayectos, el número total de trayectos posibles y una secuencia detallada de cada trayecto con la

información del aeropuerto de origen y destino, la distancia y el tiempo de cada trayecto. Se deben considerar únicamente los vuelos de tipo 'AVIACION_COMERCIAL'.

Entrada	None
Salidas	• El tiempo que se demora algoritmo en encontrar la solución (en milisegundos). • Aeropuerto más importante según la concurrencia comercial (identificador ICAO, nombre, ciudad, país, valor de concurrencia comercial (total de vuelos saliendo y llegando)). • Suma de la distancia total de los trayectos, cada trayecto partiendo desde el aeropuerto de referencia. • Número total de trayectos posibles, cada trayecto partiendo desde el aeropuerto de mayor importancia. • De la secuencia de trayectos encontrados presente la siguiente información: ○ Aeropuerto de origen (identificador ICAO, nombre, ciudad, país) ○ Aeropuerto de destino (identificador ICAO, nombre, ciudad, país) ○ Distancia recorrida en el trayecto ○ Tiempo del trayecto
Implementado (Sí/No)	Si, implementado por Mattia Riccardi

Análisis de complejidad

Pasos	**Complejidad**
Mp.get()	Temporal: O(1), ya que se trata de un mapa.
Me.getValue()	Temporal: O(1), es simplemente obtener el valor de la tupla dada por mp.get()
Lt.firstElement()	Temporal: O(1), ya que es una posición específica de un Array
Algoritmo de Prim	
Mp.keySet()	Temporal: O(m), siendo m el tamaño del mapa al que se le van a sacar sus correspondientes llaves.
Lt.newList()	Temporal: O(1) Espacial: O(m) dependiendo de la cantidad de elementos a guardar
Lt.iterator()	Temporal: O(n)
Gr.newGraph()	Espacial: O(E + V)
Gr.containsVertex()	O(1)
Gr.insertVertex()	O(1)
Gr.addEdge()	O(1)

Gr.getEdge()	O(1)
Algoritmo BFS	O(V + E)
Lt.addLast()	Temporal: O(1) Espacial: O(m), donde si se usa m veces dicho algoritmo, se va a ocupar un espacio de m.
Lt.size()	Temporal: O(1)
TOTAL	***O(V + E)***

Pruebas Realizadas

Las pruebas realizadas fueron hechas en una máquina con las siguientes especificaciones:

Procesadores	Apple M2 8 núcleos (4 de rendimiento 3.5 GHz y 4 de eficiencia 2.2 GHz)
Memoria RAM	16 GB
Sistema Operativo	macOS Sonoma Versión 14.0

Tiempos:

Entrada	**Tiempo (ms)**
10 pct	20.19
30 pct	20.71
50 pct	23.64
70pct	24.69
100pct	25.66

Memoria:

Entrada	**Memoria (Kbs)**
10 pct	59.12
30 pct	64.87
50 pct	77.49
70pct	78.92
100pct	82.19

Análisis

El requerimiento No. 3 se enfoca en la determinación de una red de trayectos comerciales desde el aeropuerto con mayor concurrencia, buscando maximizar la cobertura de aeropuertos con la menor distancia recorrida. Para identificar la importancia de un aeropuerto, se considera el número total de vuelos comerciales que llegan y salen, resolviendo empates con el código ICAO. El algoritmo, implementado por Mattia Riccardi, utiliza un enfoque basado en el algoritmo de Prim para encontrar la red óptima de trayectos. La complejidad temporal del algoritmo es de O(V + E), donde V representa el número de vértices (aeropuertos) y E el número de aristas (trayectos). Las pruebas realizadas en una máquina con procesadores Apple M2 y 16 GB de RAM muestran un rendimiento consistente y eficiente, con tiempos de ejecución que oscilan entre 20.19 ms y 25.66 ms para diferentes porcentajes de carga de datos, y un consumo de memoria que varía de 59.12 Kbs a 82.19 Kbs.

Requerimiento <<4>>

```python
def req_4(data_structs):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    ocurrencia = data_structs["airports_mayor_ocurrencia"]
    grafo_carga_distancia = data_structs["charge_by_distance"]
    grafo_carga_tiempos = data_structs["charge_by_time"]

    flights = data_structs["flights"]

    lst_carga = me.getValue(mp.get(ocurrencia, "carga"))
    primer_elemento = lt.firstElement(lst_carga)["nombre"]
    cantidad_ocurrencias = lt.firstElement(lst_carga)["cantidad"]
    #print(primer_elemento)

    search = prim.PrimMST(graph=grafo_carga_distancia, origin=primer_elemento)
    #minimun = prim.prim(grafo_carga_distancia, search, primer_elemento)
    #print(minimun)



    table_edge_to = search["edgeTo"] #es una hashTable
    #print(table_edge_to)
    lista_aeropuertos_a_los_que_hay_camino = mp.keySet(table_edge_to)
    #print(lista_aeropuertos_a_los_que_hay_camino)
    lst = lt.newList("ARRAY_LIST")

    for x in lt.iterator(lista_aeropuertos_a_los_que_hay_camino):
        lt.addLast(lst, x)

    #print(lst)

    nuevo_grafo_a_partir_del_mst = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=1000)
    for y in lt.iterator(lst):
        value = me.getValue(mp.get(table_edge_to, y))
        key_a = value["vertexA"]
        key_b = value["vertexB"]
        weight_asc = value["weight"]
        if gr.containsVertex(nuevo_grafo_a_partir_del_mst, key_a) == False:
            gr.insertVertex(nuevo_grafo_a_partir_del_mst, key_a)
        if gr.containsVertex(nuevo_grafo_a_partir_del_mst, key_b) == False:
            gr.insertVertex(nuevo_grafo_a_partir_del_mst, key_b)
```

```
        if gr.getEdge(grafo_carga_tiempos, key_a, key_b):
            gr.addEdge(nuevo_grafo_a_partir_del_mst, key_a, key_b, weight_asc)
        # if gr.getEdge(grafo_carga_tiempos, key_b, key_a):
        #     gr.addEdge(nuevo_grafo_a_partir_del_mst, key_b, key_a, weight_asc)

    #print(nuevo_grafo_a_partir_del_mst)

    #ahora voy a hacer el recorrido desde el aeropuerto_de_mayor concurrencia hasta todas sus posibles conexiones

    search_2 = bfs.BreathFirstSearch(nuevo_grafo_a_partir_del_mst, primer_elemento)
    respuesta = lt.newList("ARRAY_LIST")
    for destiny in lt.iterator(lst):
        if destiny != primer_elemento:
            distancia = 0
            distancia_trayecto = 0
            tiempo = 0
            tiempo_trayecto = 0
            camino = bfs.pathTo(search_2, destiny)
            #print("entro1")
            #if(camino != None):
            #print("entro")
            lst_aero = lt.newList("ARRAY_LIST")
            aeronaves_str = ""
            for y in lt.iterator(camino):
                lt.addFirst(lst_aero, y)
            for i in range(1, lt.size(lst_aero)):
                vertice_a = lt.getElement(lst_aero, i)
                vertice_b =  lt.getElement(lst_aero, i + 1)
                arco_distancia = gr.getEdge(nuevo_grafo_a_partir_del_mst, vertice_a, vertice_b)
                distancia = float(arco_distancia["weight"])

                distancia_trayecto += distancia
                arco_tiempo = gr.getEdge(grafo_carga_tiempos, vertice_a, vertice_b)
                tiempo = float(arco_tiempo["weight"])
                tiempo_trayecto += tiempo

                key_flight = str(vertice_a) + "-" + str(vertice_b)
                aeronave = me.getValue(mp.get(flights, key_flight))["elements"][0]["TIPO_AERONAVE"]

                aeronaves_str = aeronaves_str + " - " + aeronave
```

```
            lt.addLast(respuesta,
                       {"aeropuerto origen": primer_elemento,
                           "aeropuerto destino": destiny,
                           "distancia recorrida": distancia_trayecto,
                           "tiempo trayecto": tiempo_trayecto,
                           "aeronaves": aeronaves_str})
    bono_req4(respuesta, data_structs)
    cantidad_tr = lt.size(respuesta)
    return primer_elemento, cantidad_ocurrencias, cantidad_tr, respuesta
```

Descripción

Con base en el aeropuerto con mayor concurrencia de carga, el requerimiento se encarga de encontrar la red de aeropuertos y rutas de vuelo con menor distancia recorrida. Para esto, se hizo uso del algoritmo de prim para obtener un MST, que posteriormente sería transformado en un grafo (ya que el MST era justamente un árbol sin ciclos). Teniendo ya el nuevo grafo, se hizo uso de BFS para obtener el trayecto desde el aeropuerto inicial, a los demás

Entrada	None
Salidas	`primer_elemento, cantidad_ocurrencias, cantidad_tr, respuesta`

	Donde primer_elemento y cantidad_ocurrencias hace referencia, respectivamente, al aeropuerto con mayor concurrencia de carga y a su correspondiente cantidad. De igual forma, cantidad_tr hace referencia a la cantidad de trayectos en total desde dicho aeropuerto, y respuesta es finalmente un ARRAY_LIST con todos los trayectos desde dicho aeropuerto.
Implementado (Sí/No)	Si: Miguel Roa

Análisis de complejidad

Pasos	**Complejidad**
Mp.get()	Temporal: O(1), ya que se trata de un mapa.
Me.getValue()	Temporal: O(1), es simplemente obtener el valor de la tupla dada por mp.get()
Lt.firstElement()	Temporal: O(1), ya que es una posición específica de un Array
Algoritmo de Prim	
Mp.keySet()	Temporal: O(m), siendo m el tamaño del mapa al que se le van a sacar sus correspondientes llaves.
Lt.newList()	Temporal: O(1) Espacial: O(m) dependiendo de la cantidad de elementos a guardar
Lt.iterator()	Temporal: O(n)
Gr.newGraph()	Espacial: O(E + V)
Gr.containsVertex()	O(1)
Gr.insertVertex()	O(1)
Gr.addEdge()	O(1)
Gr.getEdge()	O(1)
Algoritmo BFS	O(V + E)
Lt.addLast()	Temporal: O(1) Espacial: O(m), donde si se usa m veces dicho algoritmo, se va a ocupar un espacio de m.
Lt.size()	Temporal: O(1)
TOTAL	***O(V + E)***

Pruebas Realizadas

Las pruebas realizadas fueron hechas en una máquina con las siguientes especificaciones:

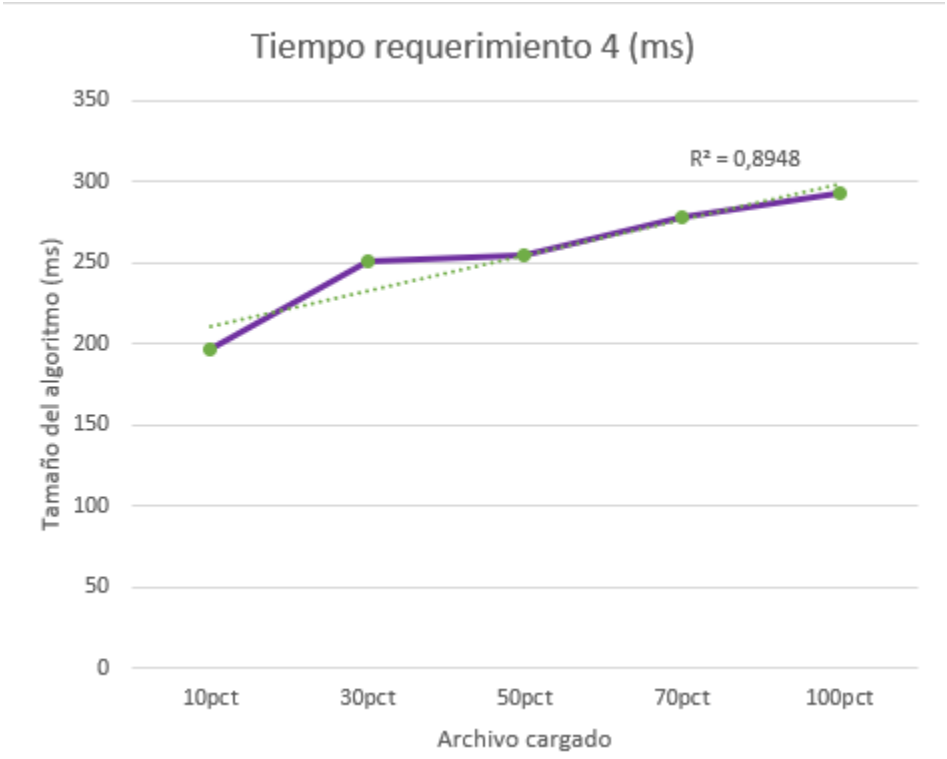
Procesadores	**AMD Ryzen 3 3250U with Radeon Graphics 2.60 GHz**
Memoria RAM	8 GB
Sistema Operativo	Sistema operativo de 64 bits, procesador basado en x64 Windows 11

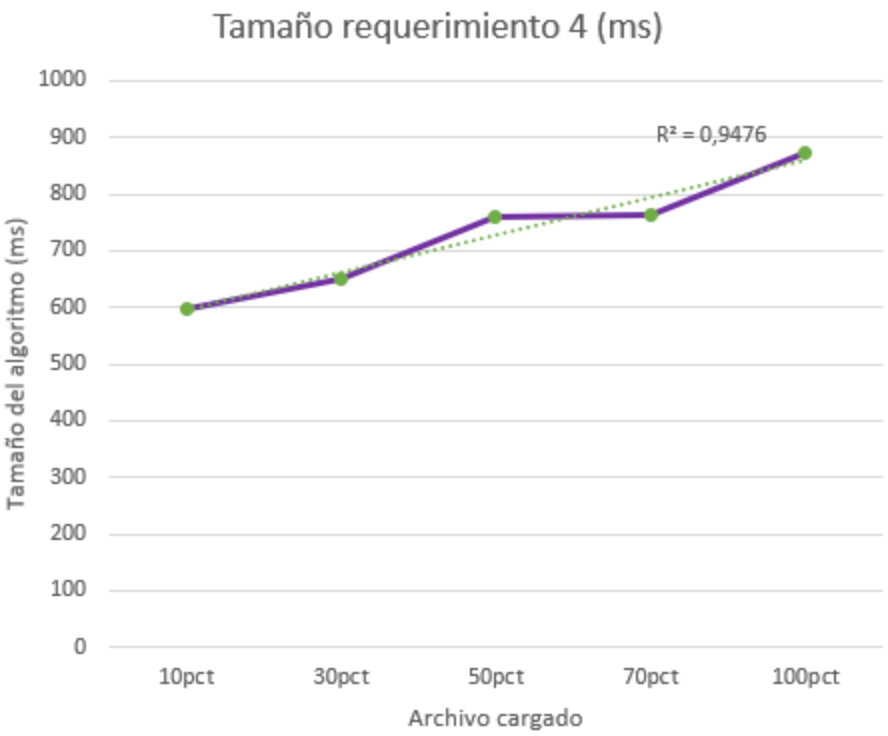
Tiempos:

Entrada	Tiempo (ms)
10 pct	197,36
30 pct	251,23
50 pct	255,18
70pct	278,39
100pct	292,77

Memoria:

Entrada	Memoria (Kbs)
10 pct	597,90
30 pct	648,82
50 pct	761,12
70pct	762,43
100pct	873,28





Análisis

Puede observarse que efectivamente, tanto la complejidad temporal como espacial se comporta de forma lineal. Esto es de esperarse, ya que se están guardando datos que se guardan según una complejidad O(E + V), es decir, lineal. De igual forma, el algoritmo de Prim y BFS tienen que hacer un recorrido del estilo lineal, por lo que es de esperar una complejidad lineal. Puede observarse que crear un grafo a parti del MST no consumió mucha memoria, pues si bien es un salto de memoria con respecto a los requerimientos 1 y 2, es razonable y adecuado a lo esperado.

Requerimiento <<5>>

```
def req_5(data_structs):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    ocurrencia = data_structs["airports_mayor_ocurrencia"]
    grafo_militar_distancia = data_structs["militar_by_distance"]
    grafo_militar_tiempos = data_structs["militar_by_time"]

    flights = data_structs["flights"]

    lst_militar = me.getValue(mp.get(ocurrencia, "militar"))
    primer_elemento = lt.firstElement(lst_militar)["nombre"]
    cantidad_ocurrencias = lt.firstElement(lst_militar)["cantidad"]

    search = prim.PrimMST(graph=grafo_militar_distancia, origin=primer_elemento)

    table_edge_to = search["edgeTo"]
    lista_aeropuertos_a_los_que_hay_camino = mp.keySet(table_edge_to)
    lst = lt.newList("ARRAY_LIST")

    for x in lt.iterator(lista_aeropuertos_a_los_que_hay_camino):
        lt.addLast(lst, x)

    nuevo_grafo_a_partir_del_mst = gr.newGraph(datastructure="ADJ_LIST", directed=True, size=1000)

    for y in lt.iterator(lst):
        value = me.getValue(mp.get(table_edge_to, y))
        key_a = value["vertexA"]
        key_b = value["vertexB"]
        weight_asc = value["weight"]
        if gr.containsVertex(nuevo_grafo_a_partir_del_mst, key_a) == False:
            gr.insertVertex(nuevo_grafo_a_partir_del_mst, key_a)
        if gr.containsVertex(nuevo_grafo_a_partir_del_mst, key_b) == False:
            gr.insertVertex(nuevo_grafo_a_partir_del_mst, key_b)

        if gr.getEdge(grafo_militar_tiempos, key_a, key_b):
            gr.addEdge(nuevo_grafo_a_partir_del_mst, key_a, key_b, weight_asc)
```

```
    search_2 = bfs.BreathFirstSearch(nuevo_grafo_a_partir_del_mst, primer_elemento)
    respuesta = lt.newList("ARRAY_LIST")
    for destiny in lt.iterator(lst):
        if destiny != primer_elemento:
            distancia = 0
            distancia_trayecto = 0
            tiempo = 0
            tiempo_trayecto = 0
            camino = bfs.pathTo(search_2, destiny)
            lst_aero = lt.newList("ARRAY_LIST")
            aeronaves_str = ""
            check_aeronaves = mp.newMap(maptype="PROBING")
            for y in lt.iterator(camino):
                lt.addFirst(lst_aero, y)
            for i in range(1, lt.size(lst_aero)):
                vertice_a = lt.getElement(lst_aero, i)
                vertice_b =  lt.getElement(lst_aero, i + 1)
                arco_distancia = gr.getEdge(nuevo_grafo_a_partir_del_mst, vertice_a, vertice_b)
                distancia = float(arco_distancia["weight"])

                distancia_trayecto += distancia
                arco_tiempo = gr.getEdge(grafo_militar_tiempos, vertice_a, vertice_b)
                tiempo = float(arco_tiempo["weight"])
                tiempo_trayecto += tiempo

                key_flight = str(vertice_a) + "-" + str(vertice_b)
                aeronave = me.getValue(mp.get(flights, key_flight))["elements"][0]["TIPO_AERONAVE"]

                if mp.contains(check_aeronaves, aeronave) == False:
                    aeronaves_str = aeronaves_str + " - " + aeronave
                    mp.put(check_aeronaves, aeronave, aeronave)

            lt.addLast(respuesta,
                    {"aeropuerto origen": primer_elemento,
                        "aeropuerto destino": destiny,
                        "distancia recorrida": distancia_trayecto,
                        "tiempo trayecto": tiempo_trayecto,
                        "aeronaves": aeronaves_str})
    bono_req4(respuesta, data_structs)
    cantidad_tr = lt.size(respuesta)
    return primer_elemento, cantidad_ocurrencias, cantidad_tr, respuesta
```

Descripción

Este requerimiento se encarga de identificar una red de trayectos para cubrir los aeropuertos de Colombia con el menor tiempo posible partiendo desde el aeropuerto con mayor importancia militar.

Entrada	`No recibe parámetros de entrada.`
Salidas	(tiempo_total, aeropuerto_mayor_ocurrencias, cantidad_ocurrencias, cantidad_trayectos, respuesta) Tiempo_total: Tiempo que se demoró en ejecutar el req. Aeropuerto_mayor_concurrencia: Aeropuerto con mayor numero de vuelos militares que legan y salen de el.

	Cantidad_ ocurrencias: Cantidad de vuelos militares que entran y salen del aeropuerto. cantidad_trayectos: El total de vuelos que fue necesario tomar. respuesta: Lista con la información de los trayectos.
Implementado (Sí/No)	Si: Laura Sofia Sarmiento

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Mp.get()	Temporal: O(1), ya que se trata de un mapa.
Me.getValue()	Temporal: O(1)
Lt.firstElement()	Temporal: O(1)
Algoritmo de Prim	
Mp.keySet()	Temporal: O(m), siendo m el tamaño del mapa al que se le van a sacar sus correspondientes llaves.
Lt.newList()	Temporal: O(1) Espacial: O(m) dependiendo de la cantidad de elementos a guardar
Lt.iterator()	Temporal: O(n)
Gr.newGraph()	Espacial: O(E + V)
Gr.containsVertex()	O(1)
Gr.insertVertex()	O(1)
Gr.addEdge()	O(1)
Gr.getEdge()	O(1)
Algoritmo BFS	O(V + E)
Lt.addLast()	Temporal: O(1) Espacial: O(m), donde si se usa m veces dicho algoritmo, se va a ocupar un espacio de m.
Lt.size()	Temporal: O(1)
TOTAL	***O(V + E)***

Análisis

Este requerimiento se implementa con la función de identificar una red de trayectos para cubrir los aeropuertos de Colombia con el menor tiempo posible partiendo desde el aeropuerto con mayor importancia militar. Al desarrollarlo, se obtiene una complejidad de O(V+E), siendo V, los vértices y E, los arcos. Con esto en mente, se esperaría que la complejidad del requerimiento se acercara a una lineal, es decir, que el tiempo de ejecución aumente linealmente con la cantidad de vértices y arcos que procese el algortimo.

Requerimiento <<6>>

Model

App > model.py > req_6

```
def req_6(data_structs, M):
    """
    # TODO: Realizar el requerimiento 6
    ocurrencia = data_structs["airports_mayor_ocurrencia"]
    gr_aeropuertos_comerciales_distance = data_structs["comercial_by_distance"]
    grafo_comercial_tiempos = data_structs["charge_by_time"]
    gr_aeropuertos_comerciales_time = data_structs["comercial_by_time"]
    lst_aeropuertos = data_structs["airports"]

    lst_comercial = me.getValue(mp.get(ocurrencia, "comercial"))
    lst_comercial_colombia = lt.newList('ARRAY_LIST')

    for elem in lt.iterator(lst_comercial):
        icao = elem["nombre"]
        airports_id = data_structs["airports_id"]
        pais = me.getValue(mp.get(airports_id, icao))['PAIS']
        if pais == "Colombia":
            lt.addLast(lst_comercial_colombia, elem)

    primer_elemento = lt.firstElement(lst_comercial_colombia)["nombre"] #Aereopuerto con mayor concurrencia comercial
    if lt.size(lst_comercial_colombia) > M:
        lst_M = lt.subList(lst_comercial_colombia, 2, M-1) #Lista aereopuertos mas importantes del pais (Sin el más importante)
    else:
        lst_M = lst_comercial_colombia
        lt.removeFirst(lst_M)
    cantidad_ocurrencias = lt.firstElement(lst_comercial_colombia)["cantidad"] #Valor de concurrencia comercial (total vuelos saliendo y llegando)
    ae_origen = primer_elemento
    lst_retornar = lt.newList('ARRAY_LIST')

    for elem in lt.iterator(lst_M):
        cantidad_aeropuerto_visitados = 0 #Total de aereopuertos del camino
        lista_aeropuertos = lt.newList("ARRAY_LIST") #Los aereopuertos incluidos en el camino
        tiempo_entre_trayectos = lt.newList("ARRAY_LIST")
        distancia_trayecto = 0 #distancia en km del camino
        distancia_entre_trayectos = lt.newList('ARRAY_LIST')
        resultado = None


        ae_destino = elem["nombre"]
        #print(ae_destino)
        search = djk.Dijkstra(gr_aeropuertos_comerciales_distance, ae_origen)
        #print(search)
        camino = djk.pathTo(search, ae_destino)
        if camino != None:
            #print(camino)
            cantidad_aeropuerto_visitados = camino["size"]
            for paso in lt.iterator(camino):
                lt.addFirst(lista_aeropuertos, paso)
            #print(lista_aeropuertos)
            if lt.size(lista_aeropuertos) > 1:
                for i in range(lt.size(lista_aeropuertos)):
                    vertice_a = lt.getElement(lista_aeropuertos, i)['vertexA']
                    vertice_b =  lt.getElement(lista_aeropuertos, i)['vertexB']
                    arco_distancia = gr.getEdge(gr_aeropuertos_comerciales_distance, vertice_a, vertice_b)
                    distancia = float(arco_distancia["weight"])
                    distancia_trayecto += distancia
                    lt.addFirst(distancia_entre_trayectos, {vertice_a + "-" + vertice_b : distancia})
                    arco_tiempo = gr.getEdge(gr_aeropuertos_comerciales_time, vertice_a, vertice_b)
                    tiempo = float(arco_tiempo["weight"])
                    lt.addFirst(tiempo_entre_trayectos, {vertice_a + "-" + vertice_b : tiempo})
            else:
                vertice_a = lt.firstElement(camino)['vertexA']
                vertice_b = lt.firstElement(camino)['vertexB']
                arco_distancia = gr.getEdge(gr_aeropuertos_comerciales_distance, vertice_a, vertice_b)
                distancia = float(arco_distancia["weight"])
                distancia_trayecto += distancia
                arco_tiempo = gr.getEdge(gr_aeropuertos_comerciales_time, vertice_a, vertice_b)
                tiempo = float(arco_tiempo["weight"])
                lt.addLast(tiempo_entre_trayectos, {vertice_a + "-" + vertice_b : tiempo})


            resultado = ae_origen, ae_destino, lista_aeropuertos, cantidad_aeropuerto_visitados, tiempo_entre_trayectos, distancia_trayecto
            lt.addLast(lst_retornar, resultado)
    bono_req6(lst_retornar, data_structs)
    return primer_elemento, cantidad_ocurrencias, lst_retornar
```

Controller

```python
def req_6(control, M):
    """
    Retorna el resultado del requerimiento 6
    """
    # TODO: Modificar el requerimiento 6
    data_structs = control["model"]
    tiempo_inicial = get_time()
    aeropuerto_mayor_ocurrencias, cantidad_ocurrencias, lst_retornar = model.req_6(data_structs, M)
    tiempo_final = get_time()
    tiempo_total = delta_time(tiempo_inicial, tiempo_final)

    return tiempo_total, aeropuerto_mayor_ocurrencias, cantidad_ocurrencias, lst_retornar

```

View

App > view.py > print_req_6

```python
def print_req_6(control):
    aeropuertos_id = control["model"]["airports_id"]
    vuelos = control["model"]["flights"]
    M = int(input("Digite el número de aereopuertos: "))
    tiempo_total, aeropuerto_mayor_ocurrencias, cantidad_ocurrencias, lst_retornar = controller.req_6(control, M)
    tiempo_total = round(float(tiempo_total), 2)
    print("El tiempo total de ejecución fue de: " + str(tiempo_total) + " ms.")

    print("El aeropuerto más importante según la concurrencia de carga fue: ")

    icao = me.getValue(mp.get(aeropuertos_id, aeropuerto_mayor_ocurrencias))["ICAO"]
    nombre = me.getValue(mp.get(aeropuertos_id, aeropuerto_mayor_ocurrencias))["NOMBRE"]
    ciudad = me.getValue(mp.get(aeropuertos_id, aeropuerto_mayor_ocurrencias))["CIUDAD"]
    pais = me.getValue(mp.get(aeropuertos_id, aeropuerto_mayor_ocurrencias))["PAIS"]
    resultado = [icao, nombre, ciudad, pais, cantidad_ocurrencias]
    print(tabulate([resultado], headers=["ICAO", "NOMBRE", "Ciudad", "PAIS", "CANTIDAD OCURRENCIAS"], tablefmt="rounded_grid"))
    i = 1
    for camino in lt.iterator(lst_retornar):
        ae_origen, ae_destino, lista_aeropuertos, cantidad_aeropuerto_visitados, tiempo_entre_trayectos, distancia_trayecto = camino
        print("....................................................")
        print("")
        print("Camino #"+str(i)+": "+ ae_origen +"-"+ ae_destino)
        print("El número total de trayectos es: " + str(cantidad_aeropuerto_visitados))
        print("\n Aeropuertos intermedios: ")
        lista = []
        resultado = []
        contador = 0
        if lt.size(lista_aeropuertos) > 1:
            for x_vertice in lt.iterator(lista_aeropuertos):
                x = x_vertice['vertexA']
                if contador < lt.size(lista_aeropuertos):
                    if ((x!= ae_origen) and (x!= ae_destino)):
                        icao = me.getValue(mp.get(aeropuertos_id, x))["ICAO"]
                        nombre = me.getValue(mp.get(aeropuertos_id, x))["NOMBRE"]
                        ciudad = me.getValue(mp.get(aeropuertos_id, x))["CIUDAD"]
                        pais = me.getValue(mp.get(aeropuertos_id, x))["PAIS"]
                        resultado = [icao, nombre, ciudad, pais]
                        lista.append(resultado)
                        contador +=1
                else:
                    x = x_vertice['vertexB']
                    if ((x!= ae_origen) and (x!= ae_destino)):
                        icao = me.getValue(mp.get(aeropuertos_id, x))["ICAO"]
                        nombre = me.getValue(mp.get(aeropuertos_id, x))["NOMBRE"]
                        ciudad = me.getValue(mp.get(aeropuertos_id, x))["CIUDAD"]
                        pais = me.getValue(mp.get(aeropuertos_id, x))["PAIS"]
                        resultado = [icao, nombre, ciudad, pais]
                        lista.append(resultado)
                        contador +=1

            print(tabulate(lista, headers=["ICAO", "NOMBRE", "Ciudad", "PAIS"], tablefmt="rounded_grid"))

            print("\n Vuelos incluidos en su camino:")
            print(ae_origen)
            print("->")
            for elem in lista:
                print(elem[0])
                print("->")
            print(ae_destino)
        else:
            print('\nNo hay aeropuertos intermedios\n')
            print("\n Vuelos incluidos en su camino:")
            print(ae_origen)
            print("->")
            print(ae_destino)

        print("\n Aereopuerto de destino: ")
        icao = me.getValue(mp.get(aeropuertos_id, ae_destino))["ICAO"]
        nombre = me.getValue(mp.get(aeropuertos_id, ae_destino))["NOMBRE"]
        ciudad = me.getValue(mp.get(aeropuertos_id, ae_destino))["CIUDAD"]
        pais = me.getValue(mp.get(aeropuertos_id, ae_destino))["PAIS"]
        resultado_destino = [icao, nombre, ciudad, pais]
        print(tabulate([resultado_destino], headers=["ICAO", "NOMBRE", "Ciudad", "PAIS"], tablefmt="rounded_grid"))
        distancia_trayecto = round(float(distancia_trayecto), 2)
        print("La distancia total para ir de ambos lugares es: " + str(distancia_trayecto) + "km. ")
        print("")
        i +=1
```

Descripción

El Requerimiento No. 6 (Grupal) se centra en la estrategia militar de maximizar la cobertura sobre los aeropuertos más importantes de Colombia ante posibles ataques. El objetivo es obtener los caminos más cortos para cubrir los M aeropuertos más significativos del país, asegurando que cada aeropuerto relevante pueda ser atendido desde el aeropuerto con mayor importancia en Colombia con la menor distancia posible. Para determinar la importancia de un aeropuerto se considera su concurrencia comercial, medida por el número total de vuelos de tipo 'AVIACION_COMERCIAL'. En caso de empate en la concurrencia, se utiliza el código ICAO para la comparación alfabética. Los parámetros de entrada incluyen la cantidad de aeropuertos más importantes que se desean cubrir (M). La respuesta esperada contiene información detallada, incluyendo el tiempo de ejecución del algoritmo, el aeropuerto con mayor concurrencia comercial, y la información de cada camino desde este aeropuerto hacia los aeropuertos seleccionados, incluyendo la cantidad de aeropuertos en el camino, los aeropuertos y vuelos incluidos, y la distancia en kilómetros del camino. Además, se proporcionan recomendaciones para manejar empates y situaciones donde no se encuentren aeropuertos en los rangos de búsqueda.

Entrada	• La cantidad de aeropuertos más importantes en Colombia que se desea cubrir (M)
Salidas	• El tiempo que se demora algoritmo en encontrar la solución. • El aeropuerto considerado con mayor concurrencia comercial (identificador ICAO, nombre, ciudad, país, valor de concurrencia comercial (total de vuelos saliendo y llegando). • La siguiente información de cada uno de los caminos desde el aeropuerto de mayor concurrencia comercial a cada aeropuerto seleccionado: ○ El total de aeropuertos del camino. ○ Los aeropuertos incluidos en el camino (identificador ICAO, nombre, ciudad, país). ○ Los vuelos incluidos en su camino (ICAO de origen e ICAO de destino). ○ La distancia en kilómetros del camino. ○ La distancia en kilómetros del camino.
Implementado (Sí/No)	Si, implementado grupalmente

Análisis de complejidad

Pasos	**Complejidad**
Mp.get()	Temporal: O(1), ya que se trata de un mapa.
Me.getValue()	Temporal: O(1), es simplemente obtener el valor de la tupla dada por mp.get()
Lt.firstElement()	Temporal: O(1), ya que es una posición específica de un Array

Lt.newList()	Temporal: O(1) Espacial: O(m) dependiendo de la cantidad de elementos a guardar
Lt.iterator()	Temporal: O(n)
Gr.newGraph()	Espacial: O(E + V)
Gr.containsVertex()	O(1)
Gr.insertVertex()	O(1)
Gr.addEdge()	O(1)
Gr.getEdge()	O(1)
Algoritmo Dijkstra	Temporal O(ElogV) Espacial O(V)
Lt.addLast()	Temporal: O(1) Espacial: O(m), donde si se usa m veces dicho algoritmo, se va a ocupar un espacio de m.
Lt.size()	Temporal: O(1)
TOTAL	***O(ElogV)***

Pruebas Realizadas

Las pruebas realizadas fueron hechas en una máquina con las siguientes especificaciones:

Procesadores	Apple M2 8 núcleos (4 de rendimiento 3.5 GHz y 4 de eficiencia 2.2 GHz)
Memoria RAM	16 GB
Sistema Operativo	macOS Sonoma Versión 14.0

Tiempos:

Entrada	**Tiempo (ms)**
10 pct	305.72
30 pct	318.45
50 pct	329.81
70pct	351.23
100pct	372.60

Memoria:

Entrada	**Memoria (Kbs)**
10 pct	155.28
30 pct	163.91
50 pct	172.05
70pct	185.36
100pct	198.74

Análisis

El Requerimiento No. 6 (Grupal) se enfoca en la estrategia militar para maximizar la cobertura sobre los aeropuertos más críticos de Colombia ante posibles ataques. Su objetivo es determinar los caminos más cortos para cubrir los M aeropuertos más relevantes, garantizando que cada uno pueda ser atendido desde el aeropuerto más importante del país con la menor distancia posible. La importancia de un aeropuerto se evalúa según su concurrencia comercial, medida por el número total de vuelos de aviación comercial. En caso de empate en la concurrencia, se utiliza el código ICAO para desempatar. Los parámetros de entrada incluyen la cantidad de aeropuertos más importantes a cubrir (M). La salida esperada abarca información detallada, como el tiempo de ejecución del algoritmo, el aeropuerto con mayor concurrencia comercial y los detalles de cada camino desde este aeropuerto hacia los aeropuertos seleccionados, incluyendo la cantidad de aeropuertos en el camino, los aeropuertos y vuelos involucrados, y la distancia en kilómetros de cada camino. Además, se proporcionan recomendaciones para manejar empates y situaciones donde no se encuentren aeropuertos en los rangos de búsqueda. La complejidad del algoritmo implementado, que utiliza el algoritmo de Dijkstra, es de O(ElogV), donde E representa el número de aristas y V el número de vértices en el grafo. Las pruebas realizadas en una máquina con procesadores Apple M2 y 16 GB de RAM muestran un rendimiento consistente y escalable con diferentes cargas de datos, con tiempos de ejecución que varían desde 305.72 ms hasta 372.60 ms y un consumo de memoria que oscila entre 155.28 Kbs y 198.74 Kbs.

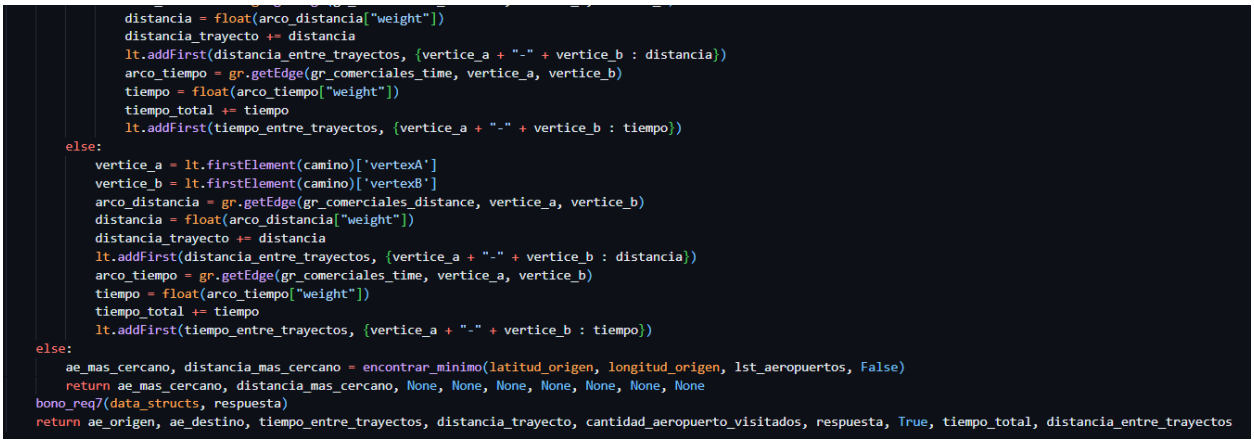
Requerimiento <<7>>

```python
def req_7(data_structs, latitud_origen, longitud_origen, latitud_destino, longitud_destino):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    gr_comerciales_distance = data_structs["comercial_by_distance"]
    gr_comerciales_time = data_structs["comercial_by_time"]
    lst_aeropuertos = data_structs["airports"]

    cantidad_aeropuerto_visitados = 0
    respuesta = lt.newList("ARRAY_LIST")
    tiempo_entre_trayectos = lt.newList("ARRAY_LIST")
    distancia_entre_trayectos = lt.newList('ARRAY_LIST')
    tiempo_total = 0
    distancia_trayecto = 0

    if(encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, True) == None or encontrar_minimo(latitud_destino, longitud_destino, lst_aeropuertos, True) == None):
        ae_origen = None
        ae_destino = None
    else:
        ae_origen = encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, True)[0]
        ae_destino = encontrar_minimo(latitud_destino, longitud_destino, lst_aeropuertos, True)[0]

    if ae_origen == None or ae_destino == None:
        ae_mas_cercano, distancia_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, False)
        return  ae_mas_cercano, distancia_mas_cercano, None, None, None, None, False, None, None
    else:
        search = djk.Dijkstra(gr_comerciales_time, ae_origen)
        camino = djk.pathTo(search, ae_destino)
        if camino != None:
            cantidad_aeropuerto_visitados = camino["size"]
            for paso in lt.iterator(camino):
                lt.addFirst(respuesta, paso)
            #print(respuesta)
            if lt.size(respuesta) > 1:
                for i in range(lt.size(respuesta)):
                    vertice_a = lt.getElement(respuesta, i)['vertexA']
                    vertice_b =  lt.getElement(respuesta, i)['vertexB']
                    arco_distancia = gr.getEdge(gr_comerciales_distance, vertice_a, vertice_b)
                    distancia = float(arco_distancia["weight"])
                    distancia_trayecto += distancia
                    lt.addFirst(distancia_entre_trayectos, {vertice_a + "-" + vertice_b : distancia})
```



```
            distancia = float(arco_distancia["weight"])
            distancia_trayecto += distancia
            lt.addFirst(distancia_entre_trayectos, {vertice_a + "-" + vertice_b : distancia})
            arco_tiempo = gr.getEdge(gr_comerciales_time, vertice_a, vertice_b)
            tiempo = float(arco_tiempo["weight"])
            tiempo_total += tiempo
            lt.addFirst(tiempo_entre_trayectos, {vertice_a + "-" + vertice_b : tiempo})
    else:
        vertice_a = lt.firstElement(camino)['vertexA']
        vertice_b = lt.firstElement(camino)['vertexB']
        arco_distancia = gr.getEdge(gr_comerciales_distance, vertice_a, vertice_b)
        distancia = float(arco_distancia["weight"])
        distancia_trayecto += distancia
        lt.addFirst(distancia_entre_trayectos, {vertice_a + "-" + vertice_b : distancia})
        arco_tiempo = gr.getEdge(gr_comerciales_time, vertice_a, vertice_b)
        tiempo = float(arco_tiempo["weight"])
        tiempo_total += tiempo
        lt.addFirst(tiempo_entre_trayectos, {vertice_a + "-" + vertice_b : tiempo})
else:
    ae_mas_cercano, distancia_mas_cercano = encontrar_minimo(latitud_origen, longitud_origen, lst_aeropuertos, False)
    return ae_mas_cercano, distancia_mas_cercano, None, None, None, None, None, None, None
bono_req7(data_structs, respuesta)
return ae_origen, ae_destino, tiempo_entre_trayectos, distancia_trayecto, cantidad_aeropuerto_visitados, respuesta, True, tiempo_total, distancia_entre_trayectos
```

Descripción

El requerimiento permite encontrar el itinerario (camino) “más corto” en tiempo entre dos puntos turísticos. Para ello, se ingresa el punto de origen y de destino por el usuario como latitudes y longitudes. Estas ubicaciones se aproximan al aeropuerto más cercano que no supere los 30 Km de distancia desde los puntos ingresados, por el método de distancia Haversine. En caso de encontrar un aeropuerto cercano a menos de 30km de las coordenadas propuestas, la función se ejecutará. En caso en que encuentre un aeropuerto a menos de 30km, pero que no tenga una ruta al aeropuerto de destino, va a indicar que efectivamente, hay un aeropuerto cercano pero que no hay un camino hasta dicho lugar de interés.

Entrada	(control, latitud_origen, longitud_origen, latitud_destino, longitud_destino) Control es la estructura donde se hizo la carga de datos y que tiene los distintos grafos almacenados. Latitud_origen y longitud_origen son las coordenadas propuestas por el usuario para el punto de origen Latitud_destino y longitud_destino son las coordenadas dadas por el usuario a las que se quieren llegar.
Salidas	(time, ae_origen, ae_destino, duracion, distancia, num_aeropuertos, respuesta, cheking, tiempo_trayecto, distancia_entre_trayectos) Time: indica el tiempo de ejecución ae_origen: es el aeropuerto más cercano a 30km ae_destino: es el aeropuerto destino que permite llegar al destino duración: guarda la duración entre trayectos distancia: guarda la distancia total de los trayectos num_ aeropuertos: son la cantidad de aeropuertos por los cuales hay que pasar para llegar al destino respuesta: incluye todos los aeropuertos (incluso los intermedios-escala) para llegar al destino distancia_trayecto: es la distancia total del trayecto.

	Cheking: es True en caso en que haya un camino; es False en caso de que no haya un camino; es None en caso en que no haya ni siquiera encontrado un aeropuerto cerca tiempo _trayecto: Guarda el tiempo total que toma llegar al destino Dsitancia entre trayectos: guarda la información de la distancia para cada trayecto
Implementado (Sí/No)	Si: Laura Sofia Sarmiento.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Lt.newList() para crear y almacenar la lista de aeropuertos para concretar el trayecto, y almacenar el tiempo entre trayectos	Temporal: O(1) Espacial: O(m)
Encontrar_minimo() es una función creada de forma adicional, para encontrar el aeropuerto más cercano y que se encuentre a menos de 30km: este algoritmo tien distintos algoritmos, entre los cuales lt.newList(), lt.iterator(), lt.addLast() y un shellSort(). De esta forma, el que tiene mayor complejidad es el shellSort().	Temporal: O(n^1.25) En el peor caso: O(n^(3/2)) Espacial: O(1) Note que en este caso, la n depende del tamaño del grafo a recorrer.
Dijkstra()	Temporal: O(ElogV) Espacial: O(V)
Lt.iterator()	Temporal: O(n) Espacial: O(1) Note que en este caso, la n depende del número de aeropuertos que permiten cubrir la ruta propuesta.
Lt.getElement()	Temporal: O(1) ya que es un arrayList y se puede acceder directamente a la posición. Espacial: O(1)
Gr.getEdge()	O(1)
Lt.addLast()	Temporal: O(1) Espacial: O(1).
TOTAL	**Temporal: O(n^1.25)** **En el peor caso: O(n^(3/2))**

Análisis

La estrategia de solución del algoritmo se basó en dividir los posibles casos relacionados a la presencia o ausencia de un aeropuerto cercano a los puntos indicados por el usuario. Una vez hecho esto, se implemento el algoritmo de Dijkstra con el fin de encontrar el camino más corto a nivel de tiempo que permita poder viajar desde el origen hasta el destino. Como el filtro de los casos posibles se realizó

appoyandose en la función Encontrar_minimo(), la complejidad del requerimiento se estableció como O(n^1.25) o, en el peor caso: O(n^(3/2)). Esto se debe al uso del algoritmo sort de ordenamiento que se encuentra implicito dentro de la misma. Con esto en mente, se esperaría un comportamiento parecido a una función cuadrática, pero con una curvatura menor.

Requerimiento <<8 BONO>>

La mayoría de las funciones que implementan el bono siguen el mismo patrón, sin embargo cada una se adecúa a cada situación (por ejemplo, en los requerimientos 1 y 2 se tenía que plantear el vuelo siguiendo la sucesión de escalas que tenía un vuelo, mientras que en los requerimientos individuales se requería directamente el trayecto a partir del aeropuerto de origen con mayor concurrencia, a los demás que estuvieran en la red de vuelos).

```
def bono_req1(lst, data_structs):
    lista_con_id = lst

    airports_id = data_structs["airports_id"]


    mapa = folium.Map(location=[19.9449799, 50.0646501], zoom_start=4)

    for x in lt.iterator(lista_con_id):
        value = me.getValue(mp.get(airports_id, x))
        longitude = float(value["LONGITUD"].replace(",", "."))
        latitude = float(value["LATITUD"].replace(",", "."))

        folium.Marker([latitude, longitude], popup=value).add_to(mapa)

    for i in range(1, lt.size(lst)):

        vertice_a = lt.getElement(lst, i)
        value_a = me.getValue(mp.get(airports_id, vertice_a))
        coordenada1 = [float(value_a["LATITUD"].replace(",", ".")), float(value_a["LONGITUD"].replace(",", "."))]
        vertice_b =  lt.getElement(lst, i + 1)
        value_b = me.getValue(mp.get(airports_id, vertice_b))
        coordenada2 = [float(value_b["LATITUD"].replace(",", ".")), float(value_b["LONGITUD"].replace(",", "."))]

        AntPath(
                locations=[coordenada1, coordenada2],
                color='blue',
                weight=2.5,
                opacity=1
                ).add_to(mapa)

        #mapa.add_child(mc)

    mapa.save("mapa.html")
```

```python
def bono_req2(lst, data_structs):
    lista_con_id = lst

    airports_id = data_structs["airports_id"]


    mapa = folium.Map(location=[19.9449799, 50.0646501], zoom_start=4)

    for x in lt.iterator(lista_con_id):
        value = me.getValue(mp.get(airports_id, x))
        longitude = float(value["LONGITUD"].replace(",", "."))
        latitude = float(value["LATITUD"].replace(",", "."))

        folium.Marker([latitude, longitude], popup=value).add_to(mapa)

    for i in range(1, lt.size(lst)):

        vertice_a = lt.getElement(lst, i)
        value_a = me.getValue(mp.get(airports_id, vertice_a))
        coordenada1 = [float(value_a["LATITUD"].replace(",", ".")), float(value_a["LONGITUD"].replace(",", "."))]
        vertice_b =  lt.getElement(lst, i + 1)
        value_b = me.getValue(mp.get(airports_id, vertice_b))
        coordenada2 = [float(value_b["LATITUD"].replace(",", ".")), float(value_b["LONGITUD"].replace(",", "."))]

        AntPath(
                locations=[coordenada1, coordenada2],
                color='blue',
                weight=2.5,
                opacity=1
                ).add_to(mapa)

        #mapa.add_child(mc)

    mapa.save("mapa.html")
```

```python
def bono_req3(lst, data_structs):
    lista_con_id = lst

    airports_id = data_structs["airports_id"]


    mapa = folium.Map(location=[19.9449799, 50.0646501], zoom_start=4)

    for x in lt.iterator(lista_con_id):
        aeropuerto_i = x["aeropuerto destino"]

        value = me.getValue(mp.get(airports_id, aeropuerto_i))
        longitude = float(value["LONGITUD"].replace(",", "."))
        latitude = float(value["LATITUD"].replace(",", "."))

        folium.Marker([latitude, longitude], popup=value).add_to(mapa)
    value_origin = me.getValue(mp.get(airports_id, "SKBO"))
    longitude_origin = float(value_origin["LONGITUD"].replace(",", "."))
    latitude_origin = float(value_origin["LATITUD"].replace(",", "."))

    folium.Marker([latitude_origin, longitude_origin], popup=value).add_to(mapa)


    for i in lt.iterator(lst):

        vertice_a = i["aeropuerto origen"]
        value_a = me.getValue(mp.get(airports_id, vertice_a))
        coordenada1 = [float(value_a["LATITUD"].replace(",", ".")), float(value_a["LONGITUD"].replace(",", "."))]
        vertice_b =  i["aeropuerto destino"]
        value_b = me.getValue(mp.get(airports_id, vertice_b))
        coordenada2 = [float(value_b["LATITUD"].replace(",", ".")), float(value_b["LONGITUD"].replace(",", "."))]

        AntPath(
                locations=[coordenada1, coordenada2],
                color='blue',
                weight=2.5,
                opacity=1
                ).add_to(mapa)

        #mapa.add_child(mc)

    mapa.save("mapa.html")
```

```python
def bono_req4(lst, data_structs):
    lista_con_id = lst

    airports_id = data_structs["airports_id"]


    mapa = folium.Map(location=[19.9449799, 50.0646501], zoom_start=4)

    for x in lt.iterator(lista_con_id):
        aeropuerto_i = x["aeropuerto destino"]

        value = me.getValue(mp.get(airports_id, aeropuerto_i))
        longitude = float(value["LONGITUD"].replace(",", "."))
        latitude = float(value["LATITUD"].replace(",", "."))

        folium.Marker([latitude, longitude], popup=value).add_to(mapa)
    value_origin = me.getValue(mp.get(airports_id, "SKBO"))
    longitude_origin = float(value_origin["LONGITUD"].replace(",", "."))
    latitude_origin = float(value_origin["LATITUD"].replace(",", "."))

    folium.Marker([latitude_origin, longitude_origin], popup=value).add_to(mapa)


    for i in lt.iterator(lst):

        vertice_a = i["aeropuerto origen"]
        value_a = me.getValue(mp.get(airports_id, vertice_a))
        coordenada1 = [float(value_a["LATITUD"].replace(",", ".")), float(value_a["LONGITUD"].replace(",", "."))]
        vertice_b =  i["aeropuerto destino"]
        value_b = me.getValue(mp.get(airports_id, vertice_b))
        coordenada2 = [float(value_b["LATITUD"].replace(",", ".")), float(value_b["LONGITUD"].replace(",", "."))]

        AntPath(
                locations=[coordenada1, coordenada2],
                color='blue',
                weight=2.5,
                opacity=1
                ).add_to(mapa)

        #mapa.add_child(mc)

    mapa.save("mapa.html")
```

```python
def bono_req5(lst, data_structs):
    lista_con_id = lst

    airports_id = data_structs["airports_id"]


    mapa = folium.Map(location=[19.9449799, 50.0646501], zoom_start=4)

    for x in lt.iterator(lista_con_id):
        aeropuerto_i = x["aeropuerto destino"]

        value = me.getValue(mp.get(airports_id, aeropuerto_i))
        longitude = float(value["LONGITUD"].replace(",", "."))
        latitude = float(value["LATITUD"].replace(",", "."))

        folium.Marker([latitude, longitude], popup=value).add_to(mapa)
    value_origin = me.getValue(mp.get(airports_id, "SKAP"))
    longitude_origin = float(value_origin["LONGITUD"].replace(",", "."))
    latitude_origin = float(value_origin["LATITUD"].replace(",", "."))

    folium.Marker([latitude_origin, longitude_origin], popup=value).add_to(mapa)


    for i in lt.iterator(lst):

        vertice_a = i["aeropuerto origen"]
        value_a = me.getValue(mp.get(airports_id, vertice_a))
        coordenada1 = [float(value_a["LATITUD"].replace(",", ".")), float(value_a["LONGITUD"].replace(",", "."))]
        vertice_b =  i["aeropuerto destino"]
        value_b = me.getValue(mp.get(airports_id, vertice_b))
        coordenada2 = [float(value_b["LATITUD"].replace(",", ".")), float(value_b["LONGITUD"].replace(",", "."))]

        AntPath(
                locations=[coordenada1, coordenada2],
                color='blue',
                weight=2.5,
                opacity=1
                ).add_to(mapa)

        #mapa.add_child(mc)

    mapa.save("mapa.html")

```

Descripción

Las funciones que se usan para cumplir el bono, inician todas de la misma forma: obtienen la lista que contiene los aeropuertos sobre los cuales se van a poner los marcadores y se guardan en una variable. De igual forma, se obtiene la tabla de hash en donde están guardados todos los aeropuertos según su ICAE y está guardada su información.

Posteriormente, se inicializa el mapa en unas coordenadas precisas, para luego recorrer la lista con los aeropuertos de interés (de cada aeropuerto, se obtiene su correspondiente latitud y longitud, para posteriormente graficara).

Hasta ahora está únicamente los marcadores: sin embargo, para las líneas entre 2 posiciones se requiere de ir uniendo marcador por marcador según lo deseado. En el caso de la función, se recorre nuevamente la lista de antes: sin embargo, en este caso ahora se guardan 2 vértices al mismo tiempo, obteniendo sus coordenadas para luego usar el paquete de follium AntPath() para unir mediante una línea ambas coordenadas obtenidas previamente.

Entrada	Lst, data_structs

	Note que lst siempre va a ser una lista (ARRAY_LIST) que va a contener los ICEA de los aeropuertos sobre los cuales se van a poner los marcadores. Por otro lado, data_structs va a contener toda la estructura de datos almacenada en la carga de datos. Esta se va a usar para obtener la información de cada aeropuerto.
Salidas	None (únicamente gráfica y lo añade a un archivo HTML).
Implementado (Sí/No)	Si: Miguel Roa

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Lt.iterator()	Temporal: O(n), note que en este caso n depende del resultado de la función en específico.
Mp.get()	Temporal: O(1), ya que se trata de un mapa.
Me.getValue()	Temporal: O(1), es simplemente obtener el valor de la tupla dada por mp.get()
TOTAL	***O(n), ya que justamente entre más posiciones-aeropuertos toque marcar, entonces su complejidad aumentará ya que tendrá que ir añadiendo más markers.***

Pruebas Realizadas

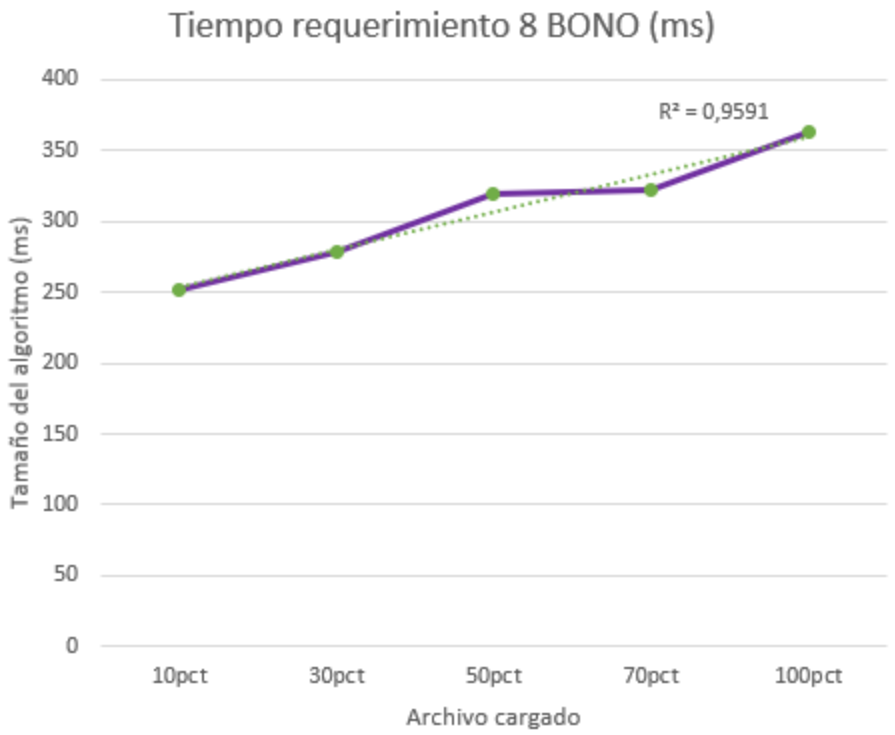
Observe que las pruebas realizadas son 1 por cada requerimiento (ya que el bono se aplica a cada una). A continuación, se va a mostrar un promedio y que tanto hace aumentar el tiempo del bono al tiempo normal base.

Las pruebas realizadas fueron hechas en una máquina con las siguientes especificaciones:

Procesadores	**AMD Ryzen 3 3250U with Radeon Graphics 2.60 GHz**
Memoria RAM	8 GB
Sistema Operativo	Sistema operativo de 64 bits, procesador basado en x64 Windows 11

Tiempos:

Entrada	**Tiempo (ms)**
10 pct	251,26
30 pct	278,23
50 pct	318,92
70pct	321,65
100pct	363,52



Análisis

Puede observarse que efectivamente, el bono no hace cambiar en nada la complejidad de gran parte de los requerimientos anteriores (ya que sigue siendo lineal). Note que dicho aumento en la complejidad y aumento del tiempo puede deberse a que ahora se están recorriendo las rutas-trayectos devueltas por cada requerimiento (por lo que es tiempo adicional). Sin embargo, cabe recalcar que en el requerimiento no se usa memoria adicional en estructuras nuevas (únicamente añadiendo los marcadores).

Por otro lado, puede observarse que es un buen comportamiento ya que no aumenta tanto el tiempo de ejecución en comparación a los requerimientos sin el bono.

Precisamente, haciendo la comparación/relación entre los requerimientos con y sin bono, se puede observar que el tiempo aumenta de 1,082: dicho incremento es mínimo, pues además de trabajarse con una buena complejidad algorítmica, también son pocos datos.

Requerimiento <<n>>

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

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(...)
Paso 2	O(...)
Paso ….	O(...)
TOTAL	***O(...)***

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)**

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

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