ANÁLISIS DEL RETO 4

Andrea Davila, 202024214, a.davilab2
Emma Mc Douall, 202315167,e.mcd
Mateo Cardenas, 202325960,ma.cardenasl1

Requerimiento <<1>>

Descripción

Este requerimiento retorna un posible camino entre dos puntos turísticos (solo se utilizan vuelos comerciales). Lo primero que hace es usar el input (las coordenadas del punto de origen y destino) para ejecutar *get_distancia_aerop_30km(),*función que retorna el aeropuerto más cercano en un radio de 30km con base en una latitud y longitud.

Luego, ya sabiendo el aeropuerto de partida y el de llegada, la función encuentra un posible camino dentro del grafo *"comercial_dist".* Esto lo hace haciendo un recorrido DFS desde el aeropuerto de partida y luego, usando el DFS, encontrando el camino hasta el aeropuerto de llegada. Este camino se guarda en una pila, la cual contiene el ICAO de los aeropuertos (vértices) que forman la ruta.

Estos ICAO se utilizan para obtener la información de cada aeropuerto del mapa *"map_aeropuertos".* Esta información se va añadiendo a la lista de resultados que al final va a ser retornada. Además, con estos ICAO se obtienen los vuelos (arcos) guardados en los grafos *"comercial_dist"* y *"comercial_time".* Con esto, se obtiene la duración de cada vuelo y la distancia que recorre, y esta información se va sumando a la distancia y tiempo total. Al final, la función retorna la lista con las escalas del trayecto (y su información), el tiempo total, la distancia total (distancia de los vuelos + distancia de los puntos a los aeropuertos), el número de escalas (el tamaño de la cola) y el tiempo de ejecución de la función (se obtiene en el controlador).

Entrada	Estructura de datos del modelo Punto de origen (latitud y longitud) (float) Punto de destino (latitud y longitud) (float)
Salidas	Tiempo de ejecución del requerimiento (float) La distancia total entre el punto de origen y el de destino (float) Tiempo del trayecto (int) El número de aeropuertos que se visitan en el camino (int) Lista con las escalas del trayecto y su información (ADT List)
Implementado (Sí/No)	Implementado por Emma Mc Douall

Análisis de complejidad

Pasos	Complejidad
`no_airports=0` `result=lt.newList("ARRAY_LIST")` `dist_vuelos=0`	O(1)

time_total=0	
o_encontrado,o_ICAO,o_dist=get_distancia_aerop_30km(data_structs,o_latitud,o_longitud) d_encontrado,d_ICAO,d_dist=get_distancia_aerop_30km(data_structs,d_latitud,d_longitud)	O(V) V=aerop
if o_encontrado and d_encontrado:	O(1)
caminos=dfs.DepthFirstSearch(data_structs["comercial_dist"],o_ICAO)	O(E+V) E=vuelos comerciales.
if dfs.hasPathTo(caminos,d_ICAO):	O(1)
path=dfs.pathTo(caminos,d_ICAO)	O(path) Peor caso path=E
no_airports= st.size(path)	O(1)
while st.size(path) > 0: aerop=st.pop(path) airport=me.getValue(mp.get(data_structs["map_aeropuertos"],aerop)) lt.addLast(result,airport) if st.top(path):	O(path)
dist=gr.getEdge(data_structs["comercial_dist"],aerop,st.top(path)) ["weight"]	O(V*path) Peor caso= V*E
dist_vuelos+=dist	O(path)
time=gr.getEdge(data_structs["comercial_time"],aerop,st.top(path))["weight"]	O(V*path) Peor caso= V*E
time_total+=time	O(path)
else: lt.addLast(result,"Busqueda fallida. No existe una ruta entre los dos destinos.Aeropuertos encontrados:") lt.addLast(result, str(o_ICAO)+", "+str(round(o_dist,2))+" km del punto de ORIGEN") lt.addLast(result,str(d_ICAO)+", "+str(round(d_dist,2))+" km del punto de DESTINO")	O(1)
else: lt.addLast(result,"Busqueda fallida.") lt.addLast(result,"Aeropuerto más cercano al ORIGEN: "+str(o_ICAO)+", "+str(round(o_dist,2))+" km")	O(1)

`lt.addLast(result,"Aeropuerto más cercano al DESTINO: "+str(d_ICAO)+", "+str(round(d_dist,2))+" km")`	
`return result,no_airports,time_total,dist_vuelos,o_dist,d_dist`	O(1)
TOTAL	***O(V*path)*** ***Peor caso O(V*E)***

Pruebas Realizadas

Procesadores	**AMD Ryzen 3 2200U with Radeon Vega Mobile Gfx 2.50 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Pro

Datos utilizados en la prueba: flights-2022.csv, airports-2022.csv.

Entrada	Tiempo (milisec)
ORIGEN (4.601992771389502, -74.06610470441926) DESTINO(10.507688799813222, -75.4706488665794)	22.94 (camino 16)
ORIGEN (10.507688799813222, -75.4706488665794) DESTINO(4.601992771389502, -74.06610470441926)	15.35 (camino 10)
ORIGEN (20.549885, -103.303295) DESTINO(6.177103, -75.434270)	23.44 (camino 17)
ORIGEN (6.177103, -75.434270) DESTINO(20.549885, -103.303295)	15.84 (ruta no encontrada)

Análisis

Después de haber extraído el camino del DFS usando *pathTo()*, la función itera este camino en un *while loop*, haciendo que la complejidad de todo lo que se encuentre dentro de este sea el tamaño del camino (*el size()* de la pila). En el peor de los casos, este camino podría llegar a ser del tamaño de todos los arcos del grafo (E), es decir podía ser todos los vuelos comerciales. Esto ocurriría si los arcos del grafo todos formaran un gran ciclo que comienza en el aeropuerto (vértice) de origen y termina en el aeropuerto (vértice) de destino. Analizando los datos, nos damos cuenta de que esto no es posible y por eso la complejidad es dada como path (largo del camino) y no E.

Dentro del *while loop*, se utiliza la operación *getEdge(graph, vertexA,vertexB)* dos veces para encontrar el peso (distancia y tiempo) de los vuelos del camino. Lo que hace esta operación es obtener la lista de vértices adyacentes del *vertexA*, la itera hasta encontrar el *vertexB* y retorna el arco entre ambos. En el peor de los casos, esta lista de adyacencia puede ser de tamaño V-1 (si no hay autorreferencia). Por lo tanto, en el peor caso, *getEdge()* tiene una complejidad de O(V).

Entonces, es por esto que la complejidad de la función en el peor caso es O(V*path) y en el peor, peor, y en este caso improbable, de los casos, O(V*E).

Requerimiento <<2>>

Descripción

Este requerimiento retorna el camino con menor número de escalas entre dos puntos turísticos (solo se utilizan vuelos comerciales). Lo primero que hace es usar el input (las coordenadas del punto de origen y destino) para ejecutar *get_distancia_aerop_30km(),* función que retorna el aeropuerto más cercano en un radio de 30km con base en una latitud y longitud.

Luego, ya sabiendo el aeropuerto de partida y el de llegada, la función encuentra el camino con menor número de escalas (menor número de arcos) dentro del grafo *"comercial_dist".* Esto lo hace haciendo un recorrido BFS desde el aeropuerto de partida y luego, usando el BFS, encontrando el camino hasta el aeropuerto de llegada. Este camino se guarda en una pila, la cual contiene el ICAO de los aeropuertos (vértices) que forman la ruta.

Estos ICAO se utilizan para obtener la información de cada aeropuerto del mapa *"map_aeropuertos"*. Esta información se va añadiendo a la lista de resultados que al final va a ser retornada. Además, con estos ICAO se obtienen los vuelos (arcos) guardados en los grafos *"comercial_dist"* y *"comercial_time"*. Con esto, se obtiene la duración de cada vuelo y la distancia que recorre, y esta información se va sumando a la distancia y tiempo total. Al final, la función retorna la lista con las escalas del trayecto (y su información), el tiempo total, la distancia total (distancia de los vuelos + distancia de los puntos a los aeropuertos), el número de escalas (el tamaño de la cola) y el tiempo de ejecución de la función (se obtiene en el controlador).

Entrada	Estructura de datos del modelo Punto de origen (latitud y longitud) (float) Punto de destino (latitud y longitud) (float)
Salidas	Tiempo de ejecución del requerimiento (float) La distancia total entre el punto de origen y el de destino (float) Tiempo del trayecto (int) El número de aeropuertos que se visitan en el camino (int) Lista con las escalas del trayecto y su información (ADT List)
Implementado (Sí/No)	Implementado por Emma Mc Douall

Análisis de complejidad

Pasos	**Complejidad**
`no_airports=0` `result=lt.newList("ARRAY_LIST")` `dist_vuelos=0` `time_total=0`	O(1)
`o_encontrado,o_ICAO,o_dist=get_distancia_aerop_30km(data_structs,o_latitud,o_longitud)` `d_encontrado,d_ICAO,d_dist=get_distancia_aerop_30km(data_structs,d_latitud,d_longitud)`	O(V) V=aerop

`if o_encontrado and d_encontrado:`	O(1)
`    caminos=bfs.BreathFirstSearch(data_structs["comercial_dist"],o_ICAO)`	O(E+V) E=vuelos comerciales.
`    if bfs.hasPathTo(caminos,d_ICAO):`	O(1)
`        path=bfs.pathTo(caminos,d_ICAO)`	O(path) Peor caso path=E
`        no_airports= st.size(path)`	O(1)
`        while st.size(path) > 0:` `                aerop=st.pop(path)` `                airport=me.getValue(mp.get(data_structs["map_aeropuertos"],aerop))` `                lt.addLast(result,airport)` `                if st.top(path):`	O(path)
`                    dist=gr.getEdge(data_structs["comercial_dist"],aerop,st.top(path))     ["weight"]`	O(V*path) Peor caso= V*E
`                    dist_vuelos+=dist`	O(path)
`                    time=gr.getEdge(data_structs["comercial_time"],aerop,st.top(path))["weight"]`	O(V*path) Peor caso= V*E
`                    time_total+=time`	O(path)
`    else:` `        lt.addLast(result,"Busqueda fallida. No existe una ruta entre los dos destinos.Aeropuertos encontrados:")` `        lt.addLast(result, str(o_ICAO)+", "+str(round(o_dist,2))+" km del punto de ORIGEN")` `        lt.addLast(result,str(d_ICAO)+", "+str(round(d_dist,2))+" km del punto de DESTINO")`	O(1)
`else:` `    lt.addLast(result,"Busqueda fallida.")` `    lt.addLast(result,"Aeropuerto más cercano al ORIGEN: "+str(o_ICAO)+", "+str(round(o_dist,2))+" km")` `    lt.addLast(result,"Aeropuerto más cercano al DESTINO: "+str(d_ICAO)+", "+str(round(d_dist,2))+" km")`	O(1)
`return result,no_airports,time_total,dist_vuelos,o_dist,d_dist`	O(1)
TOTAL	***O(V*path)***

	Peor caso O(V*E)

Pruebas Realizadas

Procesadores	**AMD Ryzen 3 2200U with Radeon Vega Mobile Gfx 2.50 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Pro

Datos utilizados en la prueba: flights-2022.csv, airports-2022.csv.

Entrada	**Tiempo (milisec)**
ORIGEN (4.601992771389502, -74.06610470441926) DESTINO(10.507688799813222, -75.4706488665794)	28.15 (camino 2)
ORIGEN (10.507688799813222, -75.4706488665794) DESTINO(4.601992771389502, -74.06610470441926)	21.74 (camino 2)
ORIGEN (20.549885, -103.303295) DESTINO(6.177103, -75.434270)	30.75 (camino 2)
ORIGEN (6.177103, -75.434270) DESTINO(20.549885, -103.303295)	31.99 no hay camino

Análisis

Después de haber extraído el camino del BFS usando *pathTo()*, la función itera este camino en un *while loop*, haciendo que la complejidad de todo lo que se encuentre dentro de este sea el tamaño del camino (*el size()* de la pila). En el peor de los casos, este camino podría llegar a ser del tamaño de todos los arcos del grafo (E), es decir podía ser todos los vuelos comerciales. Esto ocurriría si los arcos del grafo todos formaran un gran ciclo que comienza en el aeropuerto (vértice) de origen y termina en el aeropuerto (vértice) de destino. Analizando los datos, nos damos cuenta de que esto no es posible y por eso la complejidad es dada como path (largo del camino) y no E.

Dentro del *while loop*, se utiliza la operación *getEdge(graph, vertexA,vertexB)* dos veces para encontrar el peso (distancia y tiempo) de los vuelos del camino. Lo que hace esta operación es obtener la lista de vértices adyacentes del *vertexA*, la itera hasta encontrar el *vertexB* y retorna el arco entre ambos. En el peor de los casos, esta lista de adyacencia puede ser de tamaño V-1 (si no hay autorreferencia). Por lo tanto, en el peor caso, *getEdge()* tiene una complejidad de O(V).

Entonces, es por esto que la complejidad de la función en el peor caso es O(V*path) y en el peor, peor, y en este caso improbable, de los casos, O(V*E).

Requerimiento <<3>>

Descripción

Determinar la red de trayectos comerciales de cobertura máxima desde el aeropuerto con mayor concurrencia. Se abordó realizando un for loop que iterara entre los aeropuertos seguido de un if para filtrar las entradas que se quería revisar. Se uso el algoritmo dijkstra para crear los caminos mas cortos y encontrar los de menor costo para cierto origen fijo y varios destinos. Dentro del for loop se fueron actualizando los contadores y las listas pertinentes para poder obtener los resultados del requerimiento.

Entrada	No tiene parámetros de entrada
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: o Aeropuerto de origen (identificador ICAO, nombre, ciudad, país) o Aeropuerto de destino (identificador ICAO, nombre, ciudad, país) o Distancia recorrida en el trayecto o Tiempo del trayecto
Implementado (Sí/No)	Si, Andrea Davila

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
`aeropuertos_base = mp.keySet(data_structs["map_aeropuertos"])`	O(V)
`aerop_mayor_concurrencia = lt.getElement(data_structs["conc_comercial"], 1)`	O(1)
`dijkstra_mayor_conc = djk.Dijkstra(data_structs["comercial_dist"], aerop_mayor_concurrencia["aerop"]["ICAO"])`	O(ElogV)
`dist_trayectos = 0` `trayectos_posibles = 0` `encontrados = lt.newList("ARRAY_LIST")`	O(1)
`for aeropuerto in lt.iterator(aeropuertos_base):` `if djk.hasPathTo(dijkstra_mayor_conc, aeropuerto) and aeropuerto is not aerop_mayor_concurrencia["aerop"]["ICAO"]:`	O(V*E') E' siendo muy pequeña en comparación con V la mayoría de los casos porque es el camino.

`` camino_ar = djk.pathTo(dijkstra_mayor_conc, aeropuerto) # retorna una pila de arcos distancia = 0 tiempo = 0 tamanio_pila = st.size(camino_ar) for i in range(tamanio_pila): edge = st.pop(camino_ar) distancia += edge["weight"] tiempo += gr.getEdge(data_structs["comercial_time"], edge["vertexA"], edge["vertexB"])["weight"] trayectos_posibles += 1 dist_trayectos += distancia lt.addLast(encontrados, {"origen": aerop_mayor_concurrencia["aerop"], "destino": me.getValue(mp.get(data_structs["map_aeropuertos"], aeropuerto)), "distancia": distancia, "tiempo": tiempo}) ```	
TOTAL	O(V*E') E siendo el camino

Pruebas Realizadas

El computador se encontraba conectado al cargador y se tenía abierto Google Chrome con este documento de Análisis y WhatsApp, la aplicación de Windows.

Procesadores	Intel(R) Core (TM) i7-10750H CPU @ 2.60GHz 2.59 GHz
Memoria RAM	16 GB
Sistema Operativo	Windows 11

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entrada	**Tiempo (s)**

NA	15.664900004863739
NA	22.324579700361924
NA	18.973902539189356

Análisis

El requerimiento 3 tiene complejidad de *O(ElogV). Esta última complejidad se debe a que se usa el algoritmo Dijkstra para encontrar el camino más corto.* Los resultados siempre fueron consistentes. No se logró comprobar en grande escala la complejidad del requerimiento ya que no existían archivos csv con diferentes cargas de información ni entradas para el requerimiento.

Requerimiento <<4>>

Descripción

Este requerimiento busca encontrar la red de rutas de transporte que, partiendo desde el aeropuerto con mayor importancia de carga, cubra la mayor cantidad de aeropuertos con la menor distancia recorrida entre trayectos. Lo primero que hace la función es identificar el aeropuerto con mayor importancia, es decir el que tenga mayor concurrencia de vuelos de carga. Esto lo hace obteniendo el primer elemento de la lista "*conc_carga",* que guarda contiene todos los aeropuertos ordenados por orden de concurrencia.

Ya teniendo el aeropuerto de mayor importancia (Bogotá con concurrencia de 67), el algoritmo recorre el grafo con Dijkstra y usando Bogotá como su punto de origen. Luego, para cada aeropuerto, la función busca si encontró un camino dentro del recorrido. Si lo encontró, se obtiene el camino, que consiste en una pila con todos los vuelos (siendo estos los arcos en la forma *{"vertexA","vertexB" y "weight"}.*

Con la información del arco se puede construir las llaves del mapa *"map_vuelos",* el cual contiene la información de cada vuelo y del cual se puede extraer el tipo de aeronave y su tiempo. El tiempo se va sumando a el total de tiempo del vuelo, al igual que la distancia *("weight"* de cada arco) se va sumando al total de distancia del vuelo y al total de distancia general. El tipo de aeronave se va guardando en una lista.

Al final, para cada camino encontrado hay un diccionario donde se guarda su aeropuerto de origen y destino (y la info de ambos, obtenida con su ICAO de *"map_aeropuertos"*), su tiempo, su distancia y los tipos de aeronave utilizados durante el trayecto. Esto a su vez, se va guardando en la lista de resultados. Adicionalmente, se retorna el aeropuerto de mayor importancia (y su info, obtenida con su ICAO de *"map_aeropuertos"),* el número de trayectos encontrados (un contador), la distancia total de todos los trayectos, y el tiempo de ejecución de la función (se obtiene en el controlador).

Entrada	Estructura de datos del modelo.
Salidas	Tiempo de ejecución del requerimiento (float) Aeropuerto más importante según la concurrencia de carga (Dict con la info) Distancia total de los trayectos sumada (float) Número total de trayectos posibles (int) Lista con secuencia de trayectos encontrados (ADT List)
Implementado (Sí/No)	Implementado por Emma Mc Douall

Análisis de complejidad

Pasos	**Complejidad**
`importante= lt.getElement(data_structs["conc_carga"],1)` `dist_total=0` `no_trayectos=0` `results=lt.newList("ARRAY_LIST"`	O(1)
`caminos=djk.Dijkstra(data_structs["carga_dist"], importante["aerop"]["ICAO"])`	O(ElogV) E=vuelos carga

`lst_aerop=mp.keySet(data_structs["map_aeropuertos"])`	O(V) V=aerop
`for icao in lt.iterator(lst_aerop):` `  if (icao !=importante["aerop"]["ICAO"]) and (djk.hasPathTo(caminos,icao)):` `origen=importante["aerop"]` `destino=me.getValue(mp.get(data_structs["map_aeropuertos"],icao))` `tipo_avion=lt.newList("ARRAY_LIST")` `dist_viaje=0` `time_viaje=0`	O(V)
`path=djk.pathTo(caminos,icao)`	O(V*path) Peor caso path=E
`while st.size(path) > 0:` `aerop=st.pop(path)` `#aerop es el arco {vertexA,vertexB,weight}` `key="{0};{1};{2}".format(aerop["vertexA"],aerop["vertexB"],"AVIACION_CARGA")` `avion=me.getValue(mp.get(data_structs["map_vuelos"],key))` `lt.addLast(tipo_avion,avion["TIPO_AERONAVE"])` `dist_viaje+=aerop["weight"]`	O(V*path) Peor caso path=E
`time=gr.getEdge(data_structs["carga_time"],aerop["vertexA"],aerop["vertexB"])["weight"]`	O(V^2*path) Peor caso path=E
`time_viaje+=time`	O(V*path) Peor caso path=E
`dist_total+=dist_viaje` `no_trayectos+=1` `camino={"origen":origen,"destino":destino,"tipo_avion":tipo_avion,"distancia":dist_viaje,"tiempo":time_viaje}` `lt.addLast(results,camino)`	O(V)
`results=sort_results_req4(results)`	O(VlogV)
`return results,importante,dist_total,no_trayectos`	O(1)
TOTAL	***O(V^2*path)*** ***Peor caso*** ***O(V^2*E)***

Pruebas Realizadas

Procesadores	**AMD Ryzen 3 2200U with Radeon Vega Mobile Gfx 2.50 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Pro

Datos utilizados en la prueba: flights-2022.csv, airports-2022.csv.

Entrada	**Tiempo (ms)**
Intento 1	53.67
Intento 2	52.31
Intento 3	59.28
Intento 4	47.67

Análisis

Después de ejecutar Dijkstra, la función necesita encontrar los caminos (si los hay) hacia todos los aeropuertos. De *"map_areopuertos"* se extrae la lista de llaves (los ICAOS) y se itera en un *for loop* que, dado que los aeropuertos son los vértices del grafo, tiene complejidad O(V).

Dentro de este *for loop,* la función encuentra el camino del aeropuerto de mayor importancia a cada aeropuerto. Este camino a su vez se itera en un *while loop* que, en el peor de los casos podría tener complejidad O(E) (el *size()* de la pila este camino podría llegar a ser del tamaño de todos los arcos del grafo (E), es decir todos los vuelos de carga). Sin embargo, si se analizan los datos, nos damos cuenta de que esto no es posible y por eso la complejidad es dada como path (largo del camino) y no E.

Finalmente, dentro del *while loop* se utiliza la operación *getEdge(graph, vertexA,vertexB)* para encontrar el peso (tiempo) de los vuelos del camino. Lo que hace esta operación es obtener la lista de vértices adyacentes del *vertexA*, la itera hasta encontrar el *vertexB* y retorna el arco entre ambos. En el peor de los casos, esta lista de adyacencia puede ser de tamaño V-1 (si no hay autorreferencia). Por lo tanto, en el peor caso, *getEdge()* tiene una complejidad de O(V).

Entonces, una operación que toma O(V) dentro de un *while loop* de complejidad O(path), dentro de un *for loop* de complejidad O(V), tiene una complejidad total de $O(V^2*path)$. Y en el absoluto peor e improbable de los casos, donde path fuera igual a E, sería $O(V^2*E)$.

Requerimiento <<5>>

Descripción

Este requerimiento retorna el aeropuerto con mayor concurrencia militar y la información de los trayectos desde este aeropuerto.

Este requisito usa una lista que contiene el ICAO de cada aeropuerto y su concurrencia, dos mapas de aeropuertos militares en los que el peso representan el tiempo de vuelo y sus distancias respectivamente y dos mapas conteniendo la información de los vuelos y los aeropuertos.

Con estos datos se obtiene el aeropuerto más concurrido con la lista de aeropuertos y su concurrencia. Luego se recorre con Dijkstra el grafo de aeropuertos militares con tiempo como su peso desde el aeropuerto más concurrido. Con el recorrido Dijkstra se obtiene la información de los arcos y vértices, la cual se asocia con el peso del grafo de distancias, el mapa de aeropuertos y el mapa de vuelos. Toda la información es luego guardad en una lista para su posterior impresión.

Entrada	---
Salidas	Tiempo que toma el algoritmo Información del aeropuerto más importante Distancia total de los trayectos sumada Número total de trayectos posibles Lista con la información de cada trayecto
Implementado (Sí/No)	Implementado por Mateo Cárdenas

Análisis de complejidad

Pasos	**Complejidad**
`def req_5(data_structs):` `concurrency_list = data_structs['conc_militar']` `    flights = data_structs['map_vuelos']` `    airports = data_structs['map_aeropuertos']` `    top_airports =`	O(1)
`  most_important = lt.getElement(concurrency_list, 1)` `    most_important_info = mp.get(airports,` `most_important['aerop']['ICAO'])` `    most_important_info["Conc"] = most_important["conc"]` `    graph = data_structs['militar_time']` `    graph2 = data_structs['militar_dist']`	O(1)
`search = djk.Dijkstra(graph,most_important['aerop']["ICAO"])`	O(ElogV) (E y V de militar_time)
`    total_distance = 0` `    total_routes = 0` `    route_list = lt.newList('SINGLE_LINKED')` `    vertices = gr.vertices(graph)`	O(1)
`for vertex in lt.iterator(vertices):` `        if djk.hasPathTo(search, vertex):` `            path = djk.pathTo(search, vertex)` `            path_distance = 0`	O(n) (n de vertices)
`while not lt.isEmpty(path):` `edge = lt.removeFirst(path)`	$O(n^2)$

`` if edge['vertexA'] == most_important['aerop']["ICAO"]: total_routes += 1 path_distance += gr.getEdge(graph2,edge['vertexA'],edge['vertexB'])['weight'] edge_id = f"{edge['vertexA']};{edge['vertexB']};MILITAR" flight_data = mp.get(flights, edge_id) origin_airport = mp.get(airports, edge['vertexA'])['value'] destination_airport = mp.get(airports, edge['vertexB'])['value'] lt.addLast(route_list, { 'origin': edge['vertexA'], 'origin_name': origin_airport['NOMBRE'], 'origin_city': origin_airport['CIUDAD'], 'origin_country': origin_airport['PAIS'], 'destination': edge['vertexB'], 'destination_name': destination_airport['NOMBRE'], 'destination_city': destination_airport['CIUDAD'], 'destination_country': destination_airport['PAIS'], 'weight': gr.getEdge(graph2,edge['vertexA'],edge['vertexB']), 'tiempo': flight_data['value']['TIEMPO_VUELO'], 'aircraft': flight_data['value']['TIPO_AERONAVE'] }) total_distance += path_distance return most_important['aerop']["ICAO"], most_important_info, total_distance, total_routes, route_list ```	(n de vertices)
TOTAL	***O(ElogV)***

Pruebas Realizadas

Procesadores	**AMD Ryzen 5 5500U with Radeon Graphics 2.10 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Pro

Entrada	**Tiempo (milisec)**

Intento 1	17.69
Intento 2	18.52
Intento 3	18.32
Intento 4	18.63

Graficas

Análisis

Este requerimiento tiene complejidad O(ElogV) dada por el algoritmo de Dijkstra que se usa para recorrer el grafo de vuelos militares con peso de tiempo. E siendo la cantidad de arcos presentes en el grafo y V la cantidad de vértices. Esto es dado a que el grafo es denso, pues si fuera disperso, el $O(n^2)$ dada por el while dentro del for que recorre cada vértice sería mayor que O(ElogV) marcando así la complejidad del algoritmo. Dado que el resto del algoritmo es relativamente sencillo en esto recae la complejidad del algoritmo.

La función obtiene los resultados obteniendo primero el aeropuerto con mayor concurrencia militar. Luego se usa el Dijkstra sobre “militar_time” el cual es un grafo de vuelos militares. Con esto se usa ese recorrido para obtener toda la información de cada trayecto, la cual es puesta en una lista para que su impresión en view.

Requerimiento <<6>>

Descripción

Este requerimiento retorna el aeropuerto comercial más concurrido y los caminos desde dicho aeropuerto a M aeropuertos.

Este requerimiento usa un mapa con la información de los aeropuertos, una lista conteniendo cada aeropuerto y su concurrencia y dos grafos de vuelos comerciales cuyos pesos son el tiempo de vuelo y su distancia respectivamente.

Con la lista se obtiene el aeropuerto con más concurrencia junto a los M aeropuertos a los que se van a ver los trayectos. Luego se recorre con Dijkstra el mapa de vuelos comerciales con peso de tiempo desde el aeropuerto más concurrido. Desde aquí se iteran los M aeropuertos y se obtienen sus caminos, de los cuales se obtiene la información y se pone en una lista para su posterior impresión.

Entrada	Los M aeropuertos a cubrir
Salidas	Tiempo que toma el algoritmo Información del aeropuerto comercial más concurrido Lista con información por camino
Implementado (Sí/No)	Implementado por Mateo Cárdenas

Análisis de complejidad

Pasos	Complejidad
def req_6(data_structs,M): airports = data_structs['map_aeropuertos'] concurrency_list = data_structs['conc_comercial'] top_airports = lt.newList(datastructure='SINGLE_LINKED')	O(1)
for i in range(int(M)+1): lt.addLast(top_airports, concurrency_list['elements'][i])	O(M)
most_important = lt.getElement(top_airports, 1) most_important_info = mp.get(airports, most_important['aerop']['ICAO']) most_important_info["Conc"] = most_important["conc"] graph = data_structs['comercial_time'] graph2 = data_structs['comercial_dist']	O(1)
search = djk.Dijkstra(graph,most_important['aerop']['ICAO'])	O(ElogV)
paths_info = lt.newList('SINGLE_LINKED') for item in lt.iterator(top_airports): airport_code = item['aerop']['ICAO'] airport_info = mp.get(airports, airport_code)['value'] if airport_code != most_important['aerop']['ICAO']: if djk.hasPathTo(search, airport_code): path = djk.pathTo(search, airport_code) if path: path_distance = 0 airports_in_path = lt.newList('SINGLE_LINKED') flights_in_path = lt.newList('SINGLE_LINKED')	O(n) (n de top_airports)
while not lt.isEmpty(path): edge = lt.removeFirst(path) path_distance += gr.getEdge(graph2,edge['vertexA'],edge['vertexB'])['weight'] origin_airport = mp.get(airports, edge['vertexA'])['value'] destination_airport = mp.get(airports, edge['vertexB'])['value'] lt.addLast(airports_in_path, origin_airport) lt.addLast(airports_in_path, destination_airport) lt.addLast(flights_in_path, {'source': edge['vertexA'], 'destination': edge['vertexB']}) lt.addLast(paths_info, { 'airport_code': airport_code, 'airport_info': airport_info, 'total_airports': lt.size(airports_in_path), 'airports': airports_in_path,	$O(n^2)$ (n de top airports)

```
                    'flights': flights_in_path,
                    'distance': path_distance
                })

    return most_important_info, paths_info
```

TOTAL	***O(ElogV)***

Pruebas Realizadas

Procesadores	**AMD Ryzen 5 5500U with Radeon Graphics 2.10 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Pro

Input utilizado en la prueba: M = 20

Entrada	**Tiempo (milisec)**
Intento 1	11.54
Intento 2	12.03
Intento 3	12.29
Intento 4	8.19

Graficas

Análisis

Este requerimiento tiene de complejidad O(ElogV) dada por el algoritmo Dijkstra que se usa para obtener los caminos del grafo de aeropuertos comerciales con peso de tiempo a otros aeropuertos. E siendo los arcos del grafo de vuelos comerciales y V los vértices. Esto es dado a que el grafo es denso, y similar al requerimiento 5, O(ElogV) es mayor que $O(n^2)$. También hay que notar que en $O(n^2)$ n es de los aeropuertos de “top_airports” una lista que almacena los aeropuertos que ya han sido filtrados por M, mientras O(ElogV) toma el grafo entero.

Requerimiento <<7>>

Descripción

Obtener el camino más corto en tiempo para llegar entre dos puntos turísticos. Se abordó usando el algoritmo Dijkstra para poder obtener los caminos mas cortos de el mapa de vuelos comerciales filtrados por tiempo desde un origen especificado por el usuario y despues se usó para obtener el camino mas corto hacia cierto destino fijado por el usuario. Dentro del while se actualizan los contadores y las listas de aeropuertos que pertenecen al camino para poder integrar esto con la soluciono del requerimiento que se muestra en pantalla.

Entrada	• Punto de origen (una localización geográfica con latitud y longitud). • Punto de destino (una localización geográfica con latitud y longitud).
Salidas	• Tiempo de ejecución del requerimiento. • Tiempo y distancia total del camino. La distancia total 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. • La secuencia de aeropuertos que componen el camino encontrado. Para esta secuencia debe indicar: o Aeropuerto de origen (identificador ICAO, nombre, ciudad, país). o Secuencia de aeropuertos intermedios (identificador ICAO, nombre, ciudad, país) o Aeropuerto de destino (identificador ICAO, nombre, ciudad, país). o Tiempo del trayecto. o Distancia del trayecto
Implementado (Sí/No)	Si, por Andrea Dávila

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
`distancia = 0` `    tiempo = 0` `    num_aerop_camino = 0` `    camino_final = lt.newList("ARRAY_LIST")` `    lista_aeropuertos = data_structs['map_aeropuertos']` `    error_str = ""` `    success = True`	O(1)
`origen_tof, origen_icao, origen_dist = get_distancia_aerop_30km(data_structs, origen_lat, origen_lon)` `    destino_tof, destino_icao, destino_dist = get_distancia_aerop_30km(data_structs, destino_lat, destino_lon)`	O(V)

`if (origen_tof and destino_tof):` `        aeropuerto_origen = me.getValue(mp.get(data_structs["map_aeropuertos"], origen_icao))` `        aeropuerto_destino = me.getValue(mp.get(data_structs["map_aeropuertos"], destino_icao))`	O(1)
`caminos = djk.Dijkstra(data_structs['comercial_time'], origen_icao)` `        if djk.hasPathTo(caminos, destino_icao):` `            print("TIENE PATH")` `            pathto = djk.pathTo(caminos, destino_icao)`	O(ElogV)
`while st.size(pathto) > 0:` `                edge = st.pop(pathto)` `                tiempo += edge["weight"]` `                distancia += gr.getEdge(data_structs["comercial_dist"], edge["vertexA"], edge["vertexB"])["weight"]` `                num_aerop_camino += 1` `                aerop_i = mp.get(lista_aeropuertos, edge['vertexA'])['value']` `                aeropuerto_info = {"icao": aerop_i["ICAO"], "nombre" : aerop_i["NOMBRE"], "ciudad": aerop_i["CIUDAD"], "pais": aerop_i["PAIS"]}` `                lt.addLast(camino_final, aeropuerto_info)`	O(E')
`else:` `        error_str, x1 ,x2,x3,x4,x5 = req_1(data_structs, origen_lat, origen_lon, destino_lat, destino_lon)` `        success = False`	O(V*E')
TOTAL	**Si no se encuentran las coordenadas: O(V*E') *E's siendo el camino*** ***SI se encuentran las coordenadas: O(ElogV)***

Pruebas Realizadas

El computador se encontraba conectado al cargador y se tenía abierto Google Chrome con este documento de Análisis y WhatsApp, la aplicación de Windows.

Procesadores	Intel(R) Core (TM) i7-10750H CPU @ 2.60GHz 2.59 GHz
Memoria RAM	16 GB
Sistema Operativo	Windows 11

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entrada	**Tiempo (s)**
4.601992771389502, -74.06610470441926, 10.507688799813222, -75.4706488665794	12.713600009679794
4.601992771389502, -74.06610470441926, -4.19355, -69.9432	14.821200013160706
4.601992771389502, -74.06610470441926, 4.452780000000001, -75.7664	17.32150000333786

Análisis

El requerimiento 7 tiene dos complejidades posibles depende si entran en el primer if o no. Si no se encuentran las coordenadas: O(V*E') *E's siendo el camino (esto es si entra en el if) SI se encuentran las coordenadas: O(ElogV). Esta última complejidad se debe a que se usa el algoritmo Dijkstra para encontrar el camino más corto. La primera, se debe a que se llama la función del requerimiento 1 para procesar que no se pudieron encontrar y arrojar los datos pertinentes a la consola.* Los resultados siempre fueron consistentes. No se logró comprobar en grande escala la complejidad del requerimiento ya que no existían archivos csv con diferentes cargas de información para el requerimiento.

Estructuras de Datos