Universidad de los Andes
Facultad de Ingeniería

Departamento de Ingeniería de Sistemas y Computación
Estructuras de Datos y Algoritmos
ISIS-1225

ABET Engineering Accreditation Commission

ANÁLISIS DEL RETO

Lucas Valbuena León Cod 202311538
Juan José Cortés Villamil Cod 202325148
Alisson Moreno Cod 202120330

Carga de datos

Descripción

Entrada	Control (modelo de estructura de datos vacío)
Salidas	Control (modelo de estructura de datos lleno)
Implementado (Sí/No)	Sí se implementó y lo hizo Juan Jose Cortes Villamil.

```
def add_data(data_structs,part, data):
    """
    Función para agregar nuevos elementos a la lista
    """
    #TODO: Crear la función para agregar elementos a una lista
    if part == "airport":
        data["concurrencia_comercial"] = 0
        data["concurrencia_carga"] = 0
        data["concurrencia_militar"] = 0


        existsAirport = om.contains(data_structs["airports"],
data["ICAO"])
        if not existsAirport:
#Com,Car,Mil
```

```
            om.put(data_structs["airports"], data["ICAO"], data)



        gr.insertVertex(data_structs["comercialDistancia"], data["ICAO"])
        gr.insertVertex(data_structs["cargaDistancia"], data["ICAO"])
        gr.insertVertex(data_structs["militarDistancia"], data["ICAO"])
        gr.insertVertex(data_structs["comercialTiempo"], data["ICAO"])
        gr.insertVertex(data_structs["cargaTiempo"], data["ICAO"])
        gr.insertVertex(data_structs["militarTiempo"], data["ICAO"])


    if part == "flight":
        if not mp.contains(data_structs["flights"],
f"{data["ORIGEN"]}_{data["DESTINO"]}_{data["TIPO_VUELO"]}"):
            mp.put(data_structs["flights"],
f"{data["ORIGEN"]}_{data["DESTINO"]}_{data["TIPO_VUELO"]}", data)



        origen =
me.getValue(mp.get(data_structs["airports"],data["ORIGEN"]))
        destino =
me.getValue(mp.get(data_structs["airports"],data["DESTINO"]))
        lat1 = float(origen["LATITUD"])
        lat2 = float(destino["LATITUD"])
        long1 = float(origen["LONGITUD"])
        long2 = float(destino["LONGITUD"])
        distance = coversine(lat1,lat2, long1, long2)


        if data["TIPO_VUELO"] == "AVIACION_CARGA":
```

```
me.getValue(mp.get(data_structs["airports"],data["ORIGEN"]))["concurrencia
_carga"] += 1

me.getValue(mp.get(data_structs["airports"],data["DESTINO"]))["concurrenci
a_carga"] += 1
            """
            tupla =
me.getValue(mp.get(data_structs["airports"],data["ORIGEN"]))
            mp.put(data_structs["airports"],data["ORIGEN"], (tupla[0],
tupla[1] ,tupla[2] + 1, tupla[3]))
            tupla =
me.getValue(mp.get(data_structs["airports"],data["DESTINO"]))
            mp.put(data_structs["airports"],data["DESTINO"], (tupla[0],
tupla[1] , tupla[2] + 1, tupla[3]))
            """


            gr.addEdge(data_structs["cargaDistancia"],data["ORIGEN"],
data["DESTINO"], distance)
            gr.addEdge(data_structs["cargaTiempo"],data["ORIGEN"],
data["DESTINO"], float(data["TIEMPO_VUELO"]))


        elif data["TIPO_VUELO"] == "AVIACION_COMERCIAL":

me.getValue(mp.get(data_structs["airports"],data["ORIGEN"]))["concurrencia
_comercial"] += 1

me.getValue(mp.get(data_structs["airports"],data["DESTINO"]))["concurrenci
a_comercial"] += 1
            """
```

```
            tupla =
me.getValue(mp.get(data_structs["airports"],data["ORIGEN"]))

            mp.put(data_structs["airports"],data["ORIGEN"], (tupla[0],
tupla[1]+1 ,tupla[2], tupla[3]))

            tupla =
me.getValue(mp.get(data_structs["airports"],data["DESTINO"]))

            mp.put(data_structs["airports"],data["DESTINO"], (tupla[0],
tupla[1]+1, tupla[2], tupla[3]))

            """



            gr.addEdge(data_structs["comercialDistancia"],data["ORIGEN"],
data["DESTINO"], distance)

            gr.addEdge(data_structs["comercialTiempo"],data["ORIGEN"],
data["DESTINO"], float(data["TIEMPO_VUELO"]))



        else:



me.getValue(mp.get(data_structs["airports"],data["ORIGEN"]))["concurrencia
_militar"] += 1


me.getValue(mp.get(data_structs["airports"],data["DESTINO"]))["concurrenci
a_militar"] += 1

            """

            tupla =
me.getValue(mp.get(data_structs["airports"],data["ORIGEN"]))

            mp.put(data_structs["airports"],data["ORIGEN"], (tupla[0],
tupla[1] ,tupla[2], tupla[3]+1))

            tupla =
me.getValue(mp.get(data_structs["airports"],data["DESTINO"]))
```

```
            mp.put(data_structs["airports"],data["DESTINO"], (tupla[0],
tupla[1], tupla[2], tupla[3]+1))
            """


            gr.addEdge(data_structs["militarDistancia"],data["ORIGEN"],
data["DESTINO"], distance)
            gr.addEdge(data_structs["militarTiempo"],data["ORIGEN"],
data["DESTINO"], float(data["TIEMPO_VUELO"]))
```

La carga de datos tiene varios factores a considerar, por ello primero hay que dejar claro que estructuras se crearon durante la carga de datos:

```
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
    catalog= {"airports": None,
              "airports_importance":None,
              "flights" : None
              }


    catalog['airports'] = mp.newMap(numelements=108, maptype="CHAINING",
loadfactor=4)
```

```
    catalog["airports_importance"] = mp.newMap(numelements=6,
maptype="PROBING", loadfactor=0.5)

    catalog["flights"] = mp.newMap(numelements=755, maptype="CHAINING",
loadfactor=4)




    catalog["comercialDistancia"] = gr.newGraph("ADJ_LIST", directed=True,
size=428)

    catalog["cargaDistancia"] = gr.newGraph("ADJ_LIST", directed=True,
size=428)

    catalog["militarDistancia"] = gr.newGraph("ADJ_LIST", directed=True,
size=428)

    catalog["comercialTiempo"] = gr.newGraph("ADJ_LIST", directed=True,
size=428)

    catalog["cargaTiempo"] = gr.newGraph("ADJ_LIST", directed=True,
size=428)

    catalog["militarTiempo"] = gr.newGraph("ADJ_LIST", directed=True,
size=428)
```

Como se podrá observar en un primer momento se inicializan tres mapas, uno de los aeropuertos indexado por el código ICAO y como valor la información del aeropuerto. Luego hay dos mapas, 'airports_importance' en un principio lo planteamos en aras de conseguir los aeropuertos de mayor concurrencia para los requerimientos individuales, sin embargo al final no fue necesario. El mapa de 'flights' si lo usamos y estaba indexado por llaves de la forma 'origen_destino_tipo_vuelo' y como valor la información del vuelo. A continuación comenzamos la implementación de seis grafos. En realidad estamos hablando de tres grupos de dos grafos, cada grupo contenía un grafo con arcos con costo asociado de distancia y otro grafo con arcos con costo asociado temporal. Cada grupo de dos corresponde a los tipos de vuelo militar, comercial y de carga.

En lo concerniente a añadir datos como tal hay un parámetro de la función 'add_data' que le dice al algoritmo si se está cargando vuelos o aeropuertos. En el caso de que estén cargando los aeropuertos, se inicializan tres llaves en el diccionario de cada aeropuerto con cero; las llaves 'concurrencia_comercial', 'concurrencia_carga' y concurrencia_militar'. Acto seguido, se consulta si el aeropuerto está en la tabla de hash, si no, se pone. Luego se añade el ICAO de el aeropuerto de la iteración como vértice en todos los grafos. Una vez todos los vértices están en todos los grafos solo queda cargar los vuelos. En un primer momento ponemos el vuelo en el mapa 'flights'. En seguida, se consigue el código ICAO del origen y del

destino del vuelo de la iteración. Con la función haversine que creamos para la carga calculamos la distancia entre el origen y el destino. Con esta información se crea el arco consultando con un condicional si el tipo de vuelo es comercial, de carga o militar para clasificar cada arcos según su grafo correspondiente. Cabe aclarar que en cada iteración se modifica por uno el campo de concurrencia del aeropuerto y se crean dos vértices, uno con el costo del tiempo de vuelo y otro con la distancia.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Iterar por las entradas del CSV	O(N)
`existsAirport = om.contains(data_structs["airports"], data["ICAO"])`	O(N/M)
`Si no existe: om.put(data_structs["airports"], data["ICAO"], data)`	O(N/M)
`gr.insertVertex()`	O(1)
`mp.contains(data_structs["flights"], f"{data["ORIGEN"]}_{data["DESTINO"]}_{data["TIPO_VUELO"]}")`	O(N/M)
Si no existe: `mp.put(data_structs["flights"], f"{data["ORIGEN"]}_{data["DESTINO"]}_{data["TIPO_VUELO"]}", data)`	O(N/M)
`me.getValue(mp.get(data_structs["airports"],data["ORIGEN"])) me.getValue(mp.get(data_structs["airports"],data["DESTINO"]))`	O(N)
`coversine(lat1,lat2, long1, long2)`	O(1) *coversine es la función de haversine en el código*
condicional para verificar si el vuelo es de carga, militar o comercial	***O(1)***
`me.getValue(mp.get(data_structs["airports"],data["ORIGEN"]))["concurrencia_carga/militar/comercial"] += 1 me.getValue(mp.get(data_structs["airports"],data["DESTINO"]))["concurrencia_carga/militar/comercial"] += 1`	***O(N)***

`gr.addEdge()`	*O(1)*
TOTAL	*O(N)*

Pruebas Realizadas

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

Procesadores	**AMD Ryzen 7 5300U with Radeon Vega Mobile Gfx 2.30 GHz**
Memoria RAM	**16,0 GB**
Sistema Operativo	**Windows 11**

Entrada	Tiempo (ms)
airports-2022/flights-2022	1124.445

Análisis

Si la comparamos con cargas de datos pasadas, probablemente esta sea la más rápida. Claro está que el volumen de datos es diferente a los anteriores retos. Sin embargo, esta carga ayudó en gran medida a mejorar nuestro entendimiento sobre los grafos ya que son guías de consulta contrario a lo que pensábamos en un principio (creíamos que cada vértice contenía la información del aeropuerto). Difícilmente una carga de este tipo va a volverse más compleja de lo lineal, porque en los peores casos se recorren siempre N elementos. Para que la carga se vuelva de un orden más complejo que el lineal tendría que considerarse la idea de estructuras anidadas pero probablemente dificultará mucho la manipulación sin realmente ser necesario hacerlo.

Requerimiento <<1 y 2>>

```
def req_1(data_structs, latitud_origen, longitud_origen, latitud_destino,
longitud_destino):

    """

    Función que soluciona el requerimiento 1

    """

    # TODO: Realizar el requerimiento 1
```

```
    distance1 = None
    origen = None
    distance2 = None
    destino = None

    for airport in lt.iterator(mp.valueSet(data_structs["airports"])):
        distance_i = coversine(latitud_origen, float(airport["LATITUD"]),
longitud_origen, float(airport["LONGITUD"]))
        distance_ii = coversine(latitud_destino,
float(airport["LATITUD"]), longitud_destino, float(airport["LONGITUD"]))

        if distance1 == None:
            distance1 = distance_i
            origen = airport["ICAO"]

        if distance_i < distance1:
            distance1 = distance_i
            origen = airport["ICAO"]

        if distance2 == None:
            distance2 = distance_ii
            destino = airport["ICAO"]

        if distance_ii < distance2:
            distance2 = distance_ii
            destino = airport["ICAO"]
```

```
    if distance1 > 30:
        return False, distance1, distance2, origen, destino, False

    if distance2 > 30:
        return False, distance1, distance2, origen, destino, False


    search = bfs.BreathFirstSearch(data_structs["comercialDistancia"],
origen)
    vertex = bfs.bfsVertex(search, data_structs["comercialDistancia"],
origen)

    camino = bfs.pathTo(search, destino)
    if camino == None:
        return False, False, False, False, False, False

    vertices = lt.newList("ARRAY_LIST")
    for i in range(st.size(camino)):
        tope = st.pop(camino)
        lt.addLast(vertices, tope)

    conteo = 1
    kilometros_camino = 0
    trayectory_time = 0
    for airport in lt.iterator(vertices):
        if conteo < lt.size(vertices):
```

```
            edgeD = gr.getEdge(data_structs["comercialDistancia"],
airport, lt.getElement(vertices, conteo+1))

            edgeT = gr.getEdge(data_structs["comercialTiempo"], airport,
lt.getElement(vertices, conteo+1))




            kilometros_camino += float(edgeD["weight"])

            trayectory_time += float(edgeT["weight"])

        conteo +=1



    trayectory_sequence = lt.newList("ARRAY_LIST")

    for airport in lt.iterator(vertices):

        lt.addLast(trayectory_sequence,
me.getValue(mp.get(data_structs["airports"], airport)))


    variable = 0

    return kilometros_camino, distance1, distance2, lt.size(vertices),
trayectory_time, trayectory_sequence
```

Descripción

Primero utilizamos la función *coversine* para aplicar la fórmula haversine de encontrar la distancia entre dos puntos mientras se itera por los aeropuertos para saber el aeropuerto más cercano en el origen y en el destino. Si la distancia entre las coordenadas y el aeropuerto es mayor a 30 kilómetros devolverá False. Después se realiza la función search con el algoritmo BFS, que garantiza el camino de longitud mínima entre el origen y el destino. Con la función de search se realiza la función PathTo entre un origen (que viene en el search) y un destino. Ya que la función PathTo devuelve un stack, iteramos sobre el PathTo para organizarlo en un arreglo. Después iteramos sobre este arreglo para sacar el valor de los arcos y sumarlos, de este modo podemos obtener la distancia y el tiempo total del trayecto. Volvemos a iterar sobre el mismo arreglo para meter la información de cada aeropuerto buscando por su código ICAO en data_structs["airports"].

Entrada	Coordenadas de origen y coordenadas de destino.
Salidas	Distancia total entre el origen y el destino, número de aeropuertos visitados y el trayecto encontrado.
Implementado (Sí/No)	Si se implementó y lo hicieron Juan Jose Cortes Villamil y Lucas Valbuena Leon.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: iterar sobre el value set de aeropuertos	O(N)
Paso 2: función search	O(V+E)
Paso 3: función PathTo	O(V+E)
Paso 4: iterar sobre el camino	O(V+E)
Paso 5: función getEdge	O(V)
TOTAL	***O(N)***

Pruebas Realizadas

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

Procesadores	**AMD Ryzen 7 5300U with Radeon Vega Mobile Gfx 2.30 GHz**
Memoria RAM	**16,0 GB**
Sistema Operativo	**Windows 11**

Entrada	**Tiempo (ms)**
airports-2022/flights-2022	54.729

Análisis

El código que busca el camino de menor longitud entre un punto A y un punto B tiene una eficiencia gracias al recorrido BFS. La eficiencia del código también depende en gran parte de las funciones adyacentes a BFS como la función `pathTo`. Pudimos analizar que los enunciados del requerimiento 1 y el requerimiento 2 podrán funcionar bajo la misma lógica de código, ya que el requerimiento 1 no especifica que se tiene que devolver el menor camino o cualquier camino, entonces una implementación con el algoritmo BFS no es equivocada; en el requerimiento 2 si era indispensable utilizar el algoritmo BFS.

Requerimiento <<3>>

Descripción

Entrada	control[“model”] con las estructuras de datos
Salidas	Todos los trayectos alcanzables desde el aeropuerto de mayor importancia comercial, paths (trayectos pero construidos con los ICAO), aeropuerto de mayor concurrencia de carga, distancia total de los trayectos sumada y cantidad de trayectos posibles
Implementado (Sí/No)	Sí se implementó y lo hizo Alisson Moreno

```
def req_3(data_structs):

    """

    Función que soluciona el requerimiento 3

    """

    # TODO: Realizar el requerimiento 3

    lista_bono= lt.newList("ARRAY_LIST")

    grafo= data_structs["comercialDistancia"]

    mapa=data_structs["airports"]

    valor=mp.valueSet(mapa)

    ICAO,mayor=mayor_concurrencia(valor)

    busqueda= prim.PrimMST(grafo,ICAO)

    costo_total= prim.weightMST(grafo,busqueda)



    #else:

    contador=0

        #respuesta="noooooo"
```

```
    for aeropuerto in lt.iterator(mp.keySet(data_structs["airports"])):
        lista_caminos= lt.newList("ARRAY_LIST")
        if hasPathToMST(busqueda, aeropuerto):
            camino = pathToMST(busqueda, aeropuerto,ICAO)
            suma=0
            if camino != False:
                contador += 1
                pathlen = st.size(camino)
                print('El camino desde ',ICAO," hasta", aeropuerto,"es de
longitud:", str(pathlen), "\n")
                while (not st.isEmpty(camino)):
                    stop = st.pop(camino)
                    lt.addLast(lista_caminos,stop)
                lista_aero=[]
                for i in lt.iterator(lista_caminos):


                    ICAO_origen= i["vertexA"]
                    info_origen= me.getValue(mp.get(mapa,ICAO_origen))
                    ICAO_destino= i["vertexB"]
                    info_destino= me.getValue(mp.get(mapa,ICAO_destino))
                    i["Nombre_origen"]=info_origen["NOMBRE"]
                    i["Pais_origen"]= info_origen["PAIS"]
                    i["Ciudad_origen"]= info_origen["CIUDAD"]
                    i["Pais_destino"]= info_destino["PAIS"]
                    i["Nombre_destino"]= info_destino["NOMBRE"]
                    i["Ciudad_destino"]= info_destino["CIUDAD"]
                    suma += i["weight"]
```

```
                    lista_aero.append(ICAO_origen)
                lista_aero.append(aeropuerto)
                        #print(info_origen,info_destino)
                lt.addLast(lista_bono,lista_aero)
                print("la distancia total del camino es: ",suma,"km" "\n")
                imprimir=[]
                headers= ["Origen", "Nombre Origen","Pais O","Ciudad
O","Destino", "Nombre Destino","Pais D","Ciudad D","distancia",]


llaves=["vertexA","Nombre_origen","Pais_origen","Ciudad_origen","vertexB",
"Nombre_destino","Pais_destino","Ciudad_destino","weight"]
                    #elemento1=lt.lastElement(lista_imp)
                for j in lt.iterator(lista_caminos):
                    provisional=[]
                    for k in llaves:
                        provisional.append(j[k])
                    imprimir.append(provisional)
                print(tabulate(imprimir,headers=headers),"\n")


            else:
                print('No hay camino')




    return contador,costo_total, lista_bono
```

```
def mayor_concurrencia(lista):
    mayor=0
    ICAO= None
    for aeropuerto in lt.iterator(lista):
        if aeropuerto["concurrencia_comercial"]> mayor:
            mayor= aeropuerto["concurrencia_comercial"]
            ICAO = aeropuerto["ICAO"]
        if aeropuerto["concurrencia_comercial"] != 0 and
aeropuerto["concurrencia_comercial"] == mayor:
            if aeropuerto["ICAO"] > ICAO:
                mayor= aeropuerto["concurrencia_comercial"]
                ICAO = aeropuerto["ICAO"]
    return ICAO, mayor
```

Para este requerimiento, en primer lugar se obtuvo el aeropuerto con mayor concurrencia comercial haciendo uso de la función mayor_concurrencia() a la que entra por parámetro una lista con la información de todos los aeropuertos del archivo. Una vez hecho esto, se utilizó el algoritmo prim, para encontrar el árbol de recubrimiento de costo mínimo desde el aeropuerto con mayor concurrencia comercial, aquí se utilizo la funcion weightMST() para encontrar el costo total del árbol y se inicializo un contador de trayectos posibles. Luego, se itero por cada aeropuerto en el mapa “airports”, se inicializo una lista y se comprobó con la función haspathtoMST() que existiera un camino en el search(resultado del prim) hasta el aeropuerto actual, en este caso, se encuentra la ruta con la funcion pathToMST Y si el camino no es False se aumenta el contador en 1. Aquí, se empieza a emplear la pila, y se obtiene cada trayecto necesario mientras se agrega a una lista. Finalmente, se imprime toda la información relacionada con los trayectos y los aeropuertos involucrados en cada ruta.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad

Paso 1: función mayor_concurrencia	O(N)
Paso 2: función search MST	O(E log V)
Paso 3: iterar sobre aeropuertos	O(M)
Paso 3: función PathTo	O(E log V)
Paso 4: iterar sobre el camino	O(V)
Paso 5: iterar lista_caminos	O(V)
Paso 6: Imprimir	O(V)
TOTAL	***O(M*ElogV)***

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).

Procesadores	**AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz**
Memoria RAM	**16.00 GB (7.88 GB usable)**
Sistema Operativo	**Windows 10**

Entrada	**Tiempo (ms)**
airports-2022/flights-2022	162.5431

Análisis

La complejidad del req_3 se debe a la combinación de iterar sobre todos los aeropuertos y asi mismo encontrar caminos en el Árbol de Expansión Mínima (MST). En el peor de los casos, donde la naturaleza del grafo es densa y en cada camino se relacionan todos los vértices, se obtendría como resultado una complejidad temporal demasiado alta. Sin embargo, para el caso promedio, sugiere que, la iteración sobre cada uno de los caminos de los trayectos es menos costosa,por lo que, el tiempo de ejecución es significativamente bajo.

Requerimiento <<4>>

Descripción

Entrada	control[“model”] con las estructuras de datos
Salidas	Todos los trayectos alcanzables desde el aeropuerto de mayor importancia de carga, paths (trayectos pero construidos con los

	ICAO), aeropuerto de mayor concurrencia de carga, distancia total de los trayectos sumada, peso del MST, cantidad de trayectos posibles
Implementado (Sí/No)	Sí se implementó y lo hizo Juan José Cortés Villamil

```
def req_4(data_structs):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    airports = gr.vertices(data_structs["cargaDistancia"])

    maximo = None
    airportMC = None
    iguales =lt.newList("ARRAY_LIST")
    for airport in lt.iterator(airports):
        if maximo == None:
            maximo = gr.indegree(data_structs["cargaDistancia"], airport)
+ gr.outdegree(data_structs["cargaDistancia"], airport)
            airportMC = airport

        maximo_i = gr.indegree(data_structs["cargaDistancia"], airport) +
gr.outdegree(data_structs["cargaDistancia"], airport)

        if maximo_i > maximo:
            iguales = lt.newList("ARRAY_LIST")
            maximo = maximo_i
            airportMC = airport
```

```
            lt.addLast(iguales, airport)

        elif maximo_i == maximo:
            lt.addLast(iguales, airport)

    if lt.size(iguales) > 1:
        merg.sort(iguales, sort_critAlfabeticamente)
        airportMC = lt.getElement(iguales, 1)
    trayectos = lt.newList("ARRAY_LIST")
    searchi = prim.PrimMST(data_structs["cargaTiempo"], airportMC)
    searchii = prim.PrimMST(data_structs["cargaDistancia"], airportMC)

    for airport in lt.iterator(mp.keySet(data_structs["airports"])):
        if hasPathToMST(searchii, airport):
            path = pathToMST(searchii, airport, airportMC)
            if path != False:
                lt.addLast(trayectos, path)

    paths = lt.newList("ARRAY_LIST")
    for path in lt.iterator(trayectos):
        lstProvisional = lt.newList("ARRAY_LIST")
        lstProvisional2 = lt.newList("ARRAY_LIST")
        conteo = 1

        for arco in lt.iterator(path):
            if conteo == 1:
```

```
                lt.addLast(lstProvisional, arco["vertexB"])
                lt.addLast(lstProvisional, arco["vertexA"])
            else:
                lt.addLast(lstProvisional, arco["vertexA"])

            conteo += 1

        for i in range(lt.size(lstProvisional), 0, -1):
            lt.addLast(lstProvisional2, lt.getElement(lstProvisional,i))

        if not lt.isEmpty(lstProvisional2):
            lt.addLast(paths, lstProvisional2)

    distancia_total_trayectos = 0
    trayectos = lt.newList("ARRAY_LIST")

    for path in lt.iterator(paths):

        trayectory_time = 0
        kilometros_camino = 0
        tipo_aeronave = lt.newList("ARRAY_LIST")
        conteo = 1
        for airport in lt.iterator(path):

            if conteo < lt.size(path):
```

```
                flight_info = me.getValue(mp.get(data_structs["flights"],
f"{airport}_{lt.getElement(path, conteo+1)}_AVIACION_CARGA"))

                edgeD = gr.getEdge(data_structs["cargaDistancia"],
airport, lt.getElement(path, conteo+1))

                edgeT = gr.getEdge(data_structs["cargaTiempo"], airport,
lt.getElement(path, conteo+1))

                lt.addLast(tipo_aeronave, flight_info["TIPO_AERONAVE"])


                kilometros_camino += float(edgeD["weight"])

                trayectory_time += float(edgeT["weight"])

            conteo += 1

        lt.addLast(path, str(kilometros_camino))



        distancia_total_trayectos += kilometros_camino


        trayecto = (lt.getElement(path, 1), lt.getElement(path,
lt.size(path)-1), trayectory_time, kilometros_camino, tipo_aeronave)


        lt.addLast(trayectos, trayecto)


    merg.sort(trayectos, sort_crit_trayectos)

    merg.sort(paths, sort_crit_paths)

    return trayectos, paths, airportMC, distancia_total_trayectos,
prim.weightMST(data_structs["cargaDistancia"], searchii), lt.size(paths)
```

El requerimiento 4 no tenía como tal parámetros de entrada. Es un requerimiento que hace la consulta de los trayectos o la red de trayectos posibles tomando como vértice inicial el aeropuerto con mayor concurrencia de carga. Entonces, primero se hace la consulta del aeropuerto de mayor concurrencia de

carga, por ello se itera por los aeropuertos, se suma el indegree y outdegree del aeropuerto en los grafos de carga, y se va haciendo una comparación con un máximo, si es mayor cambia el valor de máximo y otra variable que guarda el nombre del aeropuerto. Si se da el caso de que es igual al máximo, se guarda en una lista el código ICAO. En este caso hay dos con la misma concurrencia máxima entonces se ordena la lista de iguales con un sort_crit alfabéticamente. Una vez, se tiene el aeropuerto de mayor concurrencia se hace Prim tomando como vértice de partida ese aeropuerto. A la hora de implementar este requerimiento nos dimos cuenta de una problemática y es que manejamos grafos dirigidos que no funcionan correctamente con prim. Además DISClib no tenía la función para construir los caminos, por lo que la creamos nosotros como se ve a continuación:

```
def pathToMST(search, destino, source):

    puntoB = mp.get(search["edgeTo"], destino)

    if puntoB == None:

        return False

    else:


        nombreB = me.getValue(puntoB)["vertexA"]

        camino = st.newStack()

        notfound = True

        while notfound == True:

            if puntoB == None:

                if nombreB == source:

                    notfound = False

                else:

                    notfound = None

            elif puntoB != None:

                st.push(camino, me.getValue(puntoB))

                puntoB = mp.get(search["edgeTo"],
me.getValue(puntoB)["vertexA"])

                if puntoB != None:

                    nombreB = me.getValue(puntoB)["vertexA"]
```

```
        if notfound == None:
            return False
        else:
            return camino



def hasPathToMST(search, destino):
    if mp.get(search["edgeTo"], destino):
        return True
    else:
        return False
```

Con esta función había que manipular el edgTo de la estructura search que nos da el algoritmo de Prim de DISClib, pero nuevamente, como es un grafo dirigido, en edgeTo habían arcos que apuntaban a 'vértices de inicio' que no eran realmente el aeropuerto de mayor concurrencia. Para solucionar esto, hubo que guardar el ICAO del aeropuerto de la iteración anterior en el edgeTo para asegurarse de que fuese el aeropuerto de mayor concurrencia, de no ser así se ignoraba el camino de la iteración. Una vez lidiamos con este problema, se itera por los aeropuertos y se consulta si hay un camino hasta ese aeropuerto, si sí, se consulta el camino y se añade a una lista. Con todos los trayectos, se itera por los trayectos posibles y se va consultando el costo asociado en términos de distancia y de tiempo para cada uno. Finalmente se devuelven los trayectos ordenados de mayor a menor distancia y se hace lo mismo con otra lista de trayectos que eventualmente servirá únicamente para la descripción de cada trayecto.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
`airports = gr.vertices(data_structs["cargaDistancia"])`	O(1)
Iterar por todos los aeropuertos (vertices)	O(N)
gr.indegree(grafo, aeropuerto)	O(1)
gr.outdegree(grafo, aeropuerto)	O(1)
lt.addLast()	O(1)
lt.size()	O(1)

Si el tamaño de 'iguales' es mayor que uno, merge sort `merg.sort(iguales, sort_critAlfabeticamente)`	O(N log N)
lt.getElement(iguales, 1)	O(1)
`searchii = prim.PrimMST(data_structs["cargaDistancia"], airportMC)`	O(E log V)
Iterar por los aeropuertos	O(N)
`if hasPathToMST(searchii, airport):`	O(1)
`path = pathToMST(searchii, airport, airportMC)`	O(N)
`lt.addLast(trayectos, path)`	O(1)
lt.newList()	O(1)
iterar por los caminos obtenidos	O(N)
creación de listas provisionales lt.newList()	O(1)
iterar por los arcos en cada camino	O(N)
lt.addLast() añadir los vértices de los arcos a una lista provisional	O(1)
Como los arcos estaban en una pila, invierto la lista en otra lista provisional	O(N)
lt.addLast(paths, lstProvisional2)	O(1)
iterar por los caminos de la lista 'paths'	O(N-1)
`flight_info = me.getValue(mp.get(data_structs["flights"], f"{airport}_{lt.getElement(path, conteo+1)}_AVIACION_CARGA"))`	O(N/M)
`edgeD = gr.getEdge(data_structs["cargaDistancia/Tiempo"], airport, lt.getElement(path, conteo+1))`	O(N)+O(N/M)
lt.addLast() para añadir las aeronaves dentro de la lista que se va a imprimir con tabulate	O(1)
`lt.addLast(path, str(kilometros_camino))` Para poder tener un indicador con el cual sortear las descripciones de los trayectos	O(1)
construcción de la tupla que constituye el trayecto a imprimir lt.getElement() y lt.size()	O(N/M)+O(1)
merge sort de 'trayectos' y 'paths'(descripción trayectos')	O(N logN)

TOTAL	*O(N+N!)*

Pruebas Realizadas

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

Procesadores	**Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz**
Memoria RAM	**8.00 GB (7.88 GB usable)**
Sistema Operativo	**Windows 10**

Entrada	Tiempo (ms)
airports-2022/flights-2022	156.73509991168976

Análisis

La complejidad de este algoritmo se debe al ciclo anidado en el que por cada camino itero por todos los aeropuertos del camino. De este modo, el peor caso es un grafo 'lineal' de modo que un camino tenga todos los vértices N. Pero entonces el siguiente camino solo podría tener N-1 aeropuertos, y el siguiente N-3 y así, configurando de esta forma la necesidad de iterar por N-1 caminos en realidad, con N! iteraciones anidadas. Nuevamente, por el volumen de los datos, el tiempo de ejecución es muy rápido esto nos da un indicio de que no estamos ante el peor caso de complejidad por la distribución de los datos, y que el volumen de datos local es aún menor teniendo en cuenta que el enfoque es solamente los vuelos de carga.

Requerimiento <<5>>

```
def req_5(data_structs):

    """

    Función que soluciona el requerimiento 5

    """

    # TODO: Realizar el requerimiento 5

    mayor = 0

    nombre_mayor = ""
```

```
    trayectos_posibles = 0



    for airport in lt.iterator(mp.valueSet(data_structs["airports"])):
        if int(airport["concurrencia_militar"]) > int(mayor):
            mayor = airport["concurrencia_militar"]
            nombre_mayor = airport["ICAO"]



    search = prim.PrimMST(data_structs["militarTiempo"], nombre_mayor)
    #suma de kilometros por arcos
    kilometros_totales = prim.weightMST(data_structs["militarDistancia"],
search)



    caminos = lt.newList("ARRAY_LIST")
    nombres_aeropuertos = lt.newList("ARRAY_LIST")
    trayectory_sequence = lt.newList("ARRAY_LIST")

    total_weights = lt.newList("ARRAY_LIST")



    for airport_search in
lt.iterator(mp.keySet(data_structs["airports"])):

        #ES UN STACK
        camino = pathToMST(search, airport_search, nombre_mayor )
```

```
        #Guarda el camino desde el aeropuerto mayor a el aeropuerto de la
iteracion
        if camino !=False:
            camino_actual = lt.newList("ARRAY_LIST")
            size_total = 0

            while not st.isEmpty(camino):
                tope = st.pop(camino)
                vertice_anadir =
me.getValue(mp.get(data_structs["airports"], tope['vertexB']))
                lt.addLast(camino_actual, vertice_anadir)
                size_total += tope['weight']



            lt.addFirst(camino_actual,
me.getValue(mp.get(data_structs["airports"], nombre_mayor)))
            lt.addLast(caminos, camino_actual)
            lt.addLast(total_weights, size_total)
            trayectos_posibles +=1









        """
```

```
        for camino in lt.iterator(caminos):
            trayectory = lt.newList("ARRAY_LIST")
            for airport in lt.iterator(camino):
                airport_info =
me.getValue(mp.get(data_structs["airports"], airport))
                lt.addLast(trayectory, airport_info)

            lt.addLast(trayectory_sequence, trayectory)

        """

    nombre_mayor = me.getValue(mp.get(data_structs["airports"],
nombre_mayor))

    return nombre_mayor, kilometros_totales, nombres_aeropuertos,
trayectos_posibles, caminos, total_weights
```

Descripción

En el requerimiento 5 primero se saca el aeropuerto con mayor concurrencia militar utilizando el valueset de los aeropuertos. Como necesitamos la optimización global del grafo mediante la solución de menor costo pero a la vez con mayor recubrimiento de vértices utilizamos MST prim, entonces ejecutamos search. Cómo es posible sacar el costo total del grafo se ejecuta la función weight que corresponde a los kilómetros totales que hay en el grafo de distancia. Después por cada aeropuerto se saca el camino desde el aeropuerto de mayor concurrencia militar hasta el aeropuerto por el que estamos iterando, si el camino existe utilizamos la función popo porque el camino es un stack y lo guardamos en un arreglo; lo guardamos de una vez con toda la información del aeropuerto.

Entrada	La estructura de datos.
Salidas	Todos los trayectos alcanzables desde el aeropuerto de mayor importancia militar.
Implementado (Sí/No)	Si se implementa y lo hizo Lucas Valbuena Leon.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: iterar sobre el value set de aeropuertos	O(N)
Paso 2: función search MST	O(E log V)
Paso 3: función PathTo	O(E log V)
Paso 4: iterar sobre el camino	O(V)
Paso 5: función getEdge	O(V)
TOTAL	***O(N)***

Pruebas Realizadas

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

Procesadores	**AMD Ryzen 7 5300U with Radeon Vega Mobile Gfx 2.30 GHz**
Memoria RAM	**16,0 GB**
Sistema Operativo	**Windows 11**

Entrada	Tiempo (ms)
airports-2022/flights-2022	159.844

Análisis

Logramos la implementación de una función PathTo en MST a pesar de que manejaramos un grafo dirigido y poder optimizar la solución y así obtener un árbol de expansión mínima con el recubrimiento necesario, cosa que no hubiéramos alcanzado si lo hubiéramos hecho con dijkstra. Aunque el grafo no fuera bidireccional se puso una condición en el recorrido de pathto para que parara cuando el edgeto de un nodo fuera None y no fuera el vértice origen.

Requerimiento <<6>>

Descripción

Entrada	control[“model”] con las estructuras de datos, M

Salidas	Caminos de costo mínimo hasta los M aeropuertos desde el aeropuerto de mayor concurrencia comercial
Implementado (Sí/No)	Sí se implementó y lo hizo Alisson Moreno

```
def req_6(data_structs,M):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    lista_bono=lt.newList("ARRAY_LIST")
    grafo= data_structs["comercialDistancia"]
    mapa=data_structs["airports"]
    lista= mp.valueSet(mapa)
    mayor,ICAO,lista_final= mayor_concurrencia_req6(lista)
    info_mayor= mp.get(mapa,ICAO)
    valor=me.getValue(info_mayor)
    #lista_caminos=lt.newList("ARRAY_LIST")
    ordenada= merg.sort(lista_final,sort_crit_req6)
    lista_aeropuertos= lt.newList("ARRAY_LIST")
    while lt.size(lista_aeropuertos)< M and not lt.isEmpty(ordenada):
        elemento = lt.removeFirst(ordenada)
        lt.addLast(lista_aeropuertos, elemento)

    if lt.size(lista_aeropuertos) <M:
        print("Hay menoooos:(")
    #MST= prim.PrimMST(grafo,ICAO)
    algoritmo=djk.Dijkstra(grafo, ICAO)
    for element in lt.iterator(lista_aeropuertos):
```

```
        lista_imp=lt.newList("ARRAY_LIST")
        vertice= element["ICAO"]
        if djk.hasPathTo(algoritmo,vertice):
            camino= djk.pathTo(algoritmo,vertice)
            suma= 0
            if camino is not None:
                pathlen = st.size(camino)
                print('El camino desde ',ICAO," hasta", vertice,"es de
longitud:", str(pathlen), "\n")
                while (not st.isEmpty(camino)):
                    stop = st.pop(camino)
                    lt.addLast(lista_imp,stop)
                lista_aero=[]
                for i in lt.iterator(lista_imp):
                    ICAO_origen= i["vertexA"]
                    info_origen= me.getValue(mp.get(mapa,ICAO_origen))
                    ICAO_destino= i["vertexB"]
                    info_destino= me.getValue(mp.get(mapa,ICAO_destino))
                    i["Nombre_origen"]=info_origen["NOMBRE"]
                    i["Pais_origen"]= info_origen["PAIS"]
                    i["Ciudad_origen"]= info_origen["CIUDAD"]
                    i["Pais_destino"]= info_destino["PAIS"]
                    i["Nombre_destino"]= info_destino["NOMBRE"]
                    i["Ciudad_destino"]= info_destino["CIUDAD"]
                    suma += i["weight"]
                    lista_aero.append(ICAO_origen)
                lista_aero.append(vertice)
```

```
                        #print(info_origen,info_destino)
                lt.addLast(lista_bono,lista_aero)
                        #print(info_origen,info_destino)
                print("la distancia total del camino es: ",suma,"km" "\n")
                imprimir=[]
                headers= ["Origen", "Nombre Origen","Pais O","Ciudad
O","Destino", "Nombre Destino","Pais D","Ciudad D","distancia",]


llaves=["vertexA","Nombre_origen","Pais_origen","Ciudad_origen","vertexB",
"Nombre_destino","Pais_destino","Ciudad_destino","weight"]
                    #elemento1=lt.lastElement(lista_imp)
                for j in lt.iterator(lista_imp):
                    provisional=[]
                    for k in llaves:
                        provisional.append(j[k])
                    imprimir.append(provisional)
                print(tabulate(imprimir,headers=headers),"\n")


            else:
                print('No hay camino')
            #print(camino,"\n")



    return valor,suma,lista_bono
def mayor_concurrencia_req6(lista):
    mayor=0
```

```
    ICAO= None
    lista_nueva= lt.newList("ARRAY_LIST")
    for aeropuerto in lt.iterator(lista):
        if aeropuerto["PAIS"] == "Colombia" and
aeropuerto["concurrencia_comercial"] != 0:
            lt.addLast(lista_nueva,aeropuerto)


        if aeropuerto["PAIS"] == "Colombia" and
aeropuerto["concurrencia_comercial"]> mayor:
            mayor= aeropuerto["concurrencia_comercial"]
            ICAO = aeropuerto["ICAO"]
        if aeropuerto["PAIS"] == "Colombia"and
aeropuerto["concurrencia_comercial"] != 0 and
aeropuerto["concurrencia_comercial"] == mayor:
            if aeropuerto["ICAO"] > ICAO:
                mayor= aeropuerto["concurrencia_comercial"]
                ICAO = aeropuerto["ICAO"]
    return mayor,ICAO,lista_nueva
```

En este requerimiento, era necesario encontrar los caminos de costo mínimo hasta los M aeropuertos de importancia comercial, para lo que, en primer lugar se encontro el vértice inicial con la función mayor_concurrencia_req6, la cual también agregaba cada aeropuerto a una lista de tipo array y se obtuvo la información relacionada con un mp.get() del mapa “airports” .Aquí, se ordeno la lista con merge sort, donde el criterio de ordenamiento es de acuerdo a la concurrencia comercial y después alfabéticamente por el código ICAO, esto se hizo con el fin de obtener los M aeropuertos más importantes. Con esto en mente, se ejecutó el algoritmo dijkstra con el grafo “comercialDistancia” y el aeropuerto mas importante comercialmente, luego, se utilizo la funcion pathTo, para encontrar el camino hacia todos los aeropuertos interes, los cuales posteriormente fueron organizados para poder imprimirlos.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad

Paso 0: Inicialización de variables	O(1)
Paso 1: función mayor_concurrencia	O(N)
Paso 2: función dijkstra	O(E log V)
Paso 3: Merge sort	O(NlogN)
Paso 3: while lt.size(lista_aeropuertos) < M and not lt.isEmpty(ordenada):	O(M)
Paso 4: iterar sobre aeropuertos	O(M)
Paso 5: función PathTo	O(E log V)
Paso 6: iterar sobre el camino	O(V)
Paso 7: iterar lista_caminos	O(V)
Paso 8: Imprimir	O(V)
TOTAL	***O(MN*ElogV)***

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).

M=5

Procesadores	**AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz**
Memoria RAM	**16.00 GB (7.88 GB usable)**
Sistema Operativo	**Windows 10**

Entrada	Tiempo (ms)
airports-2022/flights-2022	95.33

Análisis

La complejidad del req_6 está dada principalmente por la ejecución del algoritmo dijkstra, el cual en términos de eficiencia temporal, le confiere al algoritmo una baja complejidad, por lo que es posible aumentar el tamaño de M, sin que esto resulte en tiempos de ejecución demasiado altos.Ademas fue posible observar cómo el uso de pilas en la función path to, facilita el entendimiento de los caminos que componen un trayecto.

Requerimiento <<7>>

```
def req_7(data_structs, latitud_origen, longitud_origen, latitud_destino,
longitud_destino):

    """
```

```
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    distance1 = None
    origen = None
    distance2 = None
    destino = None

    for airport in lt.iterator(mp.valueSet(data_structs["airports"])):
        distance_i = coversine(latitud_origen, float(airport["LATITUD"]),
longitud_origen, float(airport["LONGITUD"]))
        distance_ii = coversine(latitud_destino,
float(airport["LATITUD"]), longitud_destino, float(airport["LONGITUD"]))

        if distance1 == None:
            distance1 = distance_i
            origen = airport["ICAO"]

        if distance_i < distance1:
            distance1 = distance_i
            origen = airport["ICAO"]

        if distance2 == None:
            distance2 = distance_ii
            destino = airport["ICAO"]

        if distance_ii < distance2:
```

```
            distance2 = distance_ii
            destino = airport["ICAO"]



    if distance1 > 30:
        return False, distance1, distance2, origen, destino


    if distance2 > 30:
        return False, distance1, distance2, origen, destino


    search = djk.Dijkstra(data_structs["comercialTiempo"], origen)



    if djk.hasPathTo(search, destino):
        camino = djk.pathTo(search, destino)
        conteo = 1
        lista_provisional = lt.newList("ARRAY_LIST")
        path = lt.newList("ARRAY_LIST")

        trayectory_time = 0
        path_distance = 0

        for edge in lt.iterator(camino):

            if conteo == 1:
                lt.addLast(lista_provisional, edge["vertexB"])
```

```
                lt.addLast(lista_provisional, edge["vertexA"])
            else:
                lt.addLast(lista_provisional, edge["vertexA"])
            conteo +=1
        for i in range(lt.size(lista_provisional), 0, -1):
            lt.addLast(path, lt.getElement(lista_provisional, i))


        airports_sequence = lt.newList("ARRAY_LIST")
        conteo = 1
        for airport in lt.iterator(path):
            if conteo < lt.size(path):


                #flight_info = me.getValue(mp.get(data_structs["flights"],
f"{airport}_{lt.getElement(path, conteo+1)}_AVIACION_COMERCIAL"))


                edgeD = gr.getEdge(data_structs["comercialDistancia"],
airport, lt.getElement(path, conteo+1))
                edgeT = gr.getEdge(data_structs["comercialTiempo"],
airport, lt.getElement(path, conteo+1))


                edgeTbefore = gr.getEdge(data_structs["comercialTiempo"],
lt.getElement(path, conteo-1), lt.getElement(path, conteo))
                #lt.addLast(airports_sequence,
(me.getValue(mp.get(data_structs["airports"],airport)),edgeT["weight"]))




                path_distance += float(edgeD["weight"])
                trayectory_time += float(edgeT["weight"])
```

```
            if conteo != 1 and conteo != lt.size(path):
                lt.addLast(airports_sequence,
(me.getValue(mp.get(data_structs["airports"],airport)),edgeTbefore["weight
"]))
            elif conteo == 1:
                lt.addLast(airports_sequence,
(me.getValue(mp.get(data_structs["airports"],airport)), 0))
            else:
                lt.addLast(airports_sequence,
(me.getValue(mp.get(data_structs["airports"],airport)), edgeT["weight"]))
            conteo += 1


        return trayectory_time, path_distance, path_distance + distance1
+distance2, lt.size(path), airports_sequence




    else:
        return False, False, False, False, False
```

Descripción

En este requerimiento primero se hace la consulta del rango de 30 kilómetros de las coordenadas entregadas por el usuario. Para ello se itera por todos los aeropuertos y usando la fórmula haversine (coversine en el código) se va guardando en una variable el aeropuerto que tenga la distancia mínima con el aeropuerto entregado por el usuario y en otra variable la distancia entre el punto entregado por el usuario y ese mismo mínimo. Si el valor de esa última variable no es superior a los 30km entonces sigue el requerimiento, en su defecto se cancela la consulta y se le notifica al usuario. Luego se ejecuta el algoritmo dijkstra tomando como vértice de partida el aeropuerto encontrado en la consulta previa para ver si hay un camino hacia el destino que entró por parámetro, si verificamos que está el camino con la función hasPath() se ejecuta pathTo() para obtener el camino y se manipulan los arcos que contiene la pila del camino. Cada arco dentro de esta pila tiene elementos que describen arcos tales que Vertex A - Vertex B y el costo asociado. Se manipulan estos vértices para ir guardando el camino invertido porque es una pila en una lista, luego se itera sobre esa misma lista para ordenar secuencialmente, es decir en el

orden esperado. Finalmente iteramos sobre la lista final (el camino) y se va añadiendo el peso en términos de costo y de tiempo de los arcos involucrados. Por esta razón no tomamos los pesos del camino que devuelve pathTo(), pues solo contemplaba los costos de distancia.

Entrada	Coordenadas de origen y coordenadas de destino.
Salidas	El camino de menor costo entre dos puntos turísticos proporcionados por el usuario.
Implementado (Sí/No)	Sí se implementó y lo hizo Juan Jose Cortes Villamil.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
iterar sobre el value set de aeropuertos	O(N)
función coversine (haversine)	O(1)
función search dijkstra	O(E log V)
hasPathTo()	O(1)
función PathTo	O(E log V)
iterar sobre el camino	O(V)
función getEdge()	O(V)
TOTAL	***O(N)***

Pruebas Realizadas

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

Procesadores	**AMD Ryzen 7 5300U with Radeon Vega Mobile Gfx 2.30 GHz**
Memoria RAM	**16,0 GB**
Sistema Operativo	**Windows 11**

Entrada	**Tiempo (ms)**
airports-2022/flights-2022	37.844

Análisis

La implementación con el algoritmo de dijkstra en este requerimiento genera también mucha eficiencia en términos de complejidad temporal. Si se aumentara el tamaño de los datos o se generarán más operaciones sobre listas la complejidad se vería afectada, ya que al utilizar addLast garantizamos una

complejidad mínima. Aunque Dijkstra optimiza la búsqueda de rutas, el manejo eficiente de las estructuras de datos también influye en la eficiencia global del código.

Requerimiento <<8>>

```
def print_req_8(control):
    """
        Función que imprime la solución del Requerimiento 8 en consola
    """
    # TODO: Imprimir el resultado del requerimiento 8
    print("_"*100)
    print("Por favor digite la opción de requerimiento desea visualizar graficamente: ")
    for i in range(1,8):
        print(f"[{i}] Requerimiento {i}")
    answer= int(input("Por favor digite la opción del requerimiento que desea visualizar: "))
    print("_"*100)
    if answer == 1:
        print("\nPOR FAVOR ENTREGUE LA SIGUIENTE INFORMACIÓN PARA LA CONSULTA:\n")
        latitud_origen = input("Por favor digite la latitud del punto geográfico de origen... ")
        longitud_origen = input("Por favor digite la longitud del punto geográfico de origen... ")


        latitud_destino = input("Por favor digite la latitud del punto geográfico de destino... ")
        longitud_destino = input("Por favor digite la longitud del punto geográfico de destino... ")
```

```
        kilometros_camino, distance1, distance2, cantidad_paradas ,
trayectory_time, trayectory_sequence, time,  trayectories_times =
controller.req_1(control, latitud_origen, longitud_origen,
latitud_destino, longitud_destino)

        conteo = 1

        m = folium.Map(location = (lt.getElement(trayectory_sequence,
1)["LATITUD"],lt.getElement(trayectory_sequence, 1)["LONGITUD"]))

        coords = []

        for airport in lt.iterator(trayectory_sequence):

            if conteo < lt.size(trayectory_sequence):

                start = [float(airport["LATITUD"]),
float(airport["LONGITUD"])]

                end = [float(lt.getElement(trayectory_sequence,
conteo+1)["LATITUD"]), float(lt.getElement(trayectory_sequence,
conteo+1)["LONGITUD"])]



                folium.Marker(

                location=start,

                tooltip="Click me!",

                popup=f"Título: {airport['NOMBRE']}",

                icon=folium.Icon(color="green"),

                ).add_to(m)


                folium.Marker(

                location=end,

                tooltip="Click me!",
```

```
                popup=f"Título: {lt.getElement(trayectory_sequence,
conteo+1)['NOMBRE']}",
                icon=folium.Icon(color="green"),
                ).add_to(m)

                folium.PolyLine(
                locations=[start,end],
                color='blue',
                weight=5,
                opacity=0.7,
                dash_array='10',
                arrowheads={'fill': True, 'size': 5, 'frequency': 'end'}
                ).add_to(m)
                conteo += 1

        #m.save("map_with_arrow.html")

        m.show_in_browser()
```

...

```
elif answer == 4:
        trayectos, paths, airportMC, distancia_total_trayectos, pesoMST=
controller.req_4(control)
        mapa = False
        m = None
        for path in lt.iterator(paths):
            lt.deleteElement(path, lt.size(path))
            conteo = 1
```

```
            for icao in lt.iterator(path):
                if conteo < lt.size(path):
                    airport =
me.getValue(mp.get(control["model"]["airports"], icao))
                    nextAirport =
me.getValue(mp.get(control["model"]["airports"], lt.getElement(path,
conteo+1)))
                    if not mapa:
                        m = folium.Map(location =
[airport["LATITUD"],airport["LONGITUD"]])
                        mapa = True

                    start = [float(airport["LATITUD"]),
float(airport["LONGITUD"])]
                    end = [float(nextAirport["LATITUD"]),
float(nextAirport["LONGITUD"])]



                    folium.Marker(
                    location=start,
                    tooltip="Click me!",
                    popup=f"Título: {airport['NOMBRE']}",
                    icon=folium.Icon(color="green"),
                    ).add_to(m)

                    folium.Marker(
                    location=end,
                    tooltip="Click me!",
```

```
                    popup=f"Título: {nextAirport['NOMBRE']}",
                    icon=folium.Icon(color="green"),
                    ).add_to(m)

                    folium.PolyLine(
                    locations=[start,end],
                    color='blue',
                    weight=5,
                    opacity=0.7,
                    dash_array='10',
                    arrowheads={'fill': True, 'size': 5, 'frequency':
'end'}
                    ).add_to(m)
                    conteo += 1
        #m.save("map_with_arrow.html")

        m.show_in_browser()
```

...

Nota: No pongo todo el código porque no es necesario, la implementación del bono con un requerimiento se replica en los demás, es decir hay dos arquetipos de implementación del requerimiento.

Descripción

Para desarrollar este requerimiento nos basamos en el reto pasado, y consultamos otras fuentes para incluir líneas entre los marcadores de Folium. En realidad no tuvo mayor complejidad, sin embargo, es importante destacar que la implementación requirió que los requerimientos retornan listas de caminos, cabe aclarar que unos requerimientos retornaban una lista que describe un solo camino y otros una lista de listas, donde cada lista interna es un camino. Por ello, arriba copie dos ejemplos y no todo el código pues el resto se replica en el resto del código. Entonces en el primer caso (ejemplo: requerimiento 1 y 2) que retornaba una lista describiendo un único camino había que recorrer por los elementos o 'paradas' de la lista que o bien eran códigos ICAO o los diccionarios de los aeropuertos de una vez. Entonces se

marcaban el elemento de la iteración y el que seguía en la lista del camino, y se creaba una elemento dinámico de Folium llamado Polyline que permitió conectar ambas marcas. En el segundo caso, en que los requerimientos retornan una lista de listas describiendo caminos, se hizo un ciclo anidado, el primer ciclo recorriendo los caminos y el segundo los aeropuertos dentro de cada camino. A continuación se hizo lo mismo que en el caso anterior (marcar el elemento de la iteración y el siguiente en la lista para conectarlos con un elemento dinámico de Folium). Es importante aclarar que para marcar el siguiente en la lista se creó una variable 'conteo' que permite llevar una noción de en qué iteración se está y de ese modo manipular la lista de a pares.

Entrada	Requerimiento a visualizar gráficamente (Otros parámetros de entrada dependen exclusivamente de cada requerimiento)
Salidas	Resultados del requerimiento x graficados en un mapa que permite visualizar los diferentes caminos.
Implementado (Sí/No)	Sí se implementó y lo hizo Juan Jose Cortes Villamil.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Iterar para presentar un menú de opciones de requerimiento al usuario	**O(7)**
Caso 1 (El requerimiento entrega una lista describiendo un único camino)	
Requerimiento X	O(complejidad temporal del requerimiento escogido)
folium.map()	O(1)
iterar por los aeropuertos del trayecto	O(N)
lt.getElement() para obtener el aeropuerto de la siguiente posición en la lista de aeropuertos	O(1)
folium.Marker()	O(1)
folium.Marker()	O(1)
folium.PolyLine()	O(1)
m.show_in_browser()	O(1)
Caso 2 (El requerimiento entrega una lista de listas donde cada lista describe un camino)	
Requerimiento X	O(complejidad temporal del requerimiento escogido)
iterar por los caminos encontrados con el requerimiento	O(N-1)
iterar por los aeropuertos de cada camino	O(N!)
(Si la lista de cada camino son los códigos ICAO) me.getValue(mp.get()) para obtener el diccionario del aeropuerto de la iteración y del siguiente en la lista para tratar los aeropuertos en pares	O(N/M)

`if not mapa:`
`    m = folium.Map(location = [airport["LATITUD"],airport["LONGITUD"]])`
`    mapa = True`
	O(1) **Si no se ha creado el mapa se inicializa con las coordenadas del primer aeropuerto sobre el que este iterando**

folium.Marker()	O(1)
folium.Marker()	O(1)
folium.PolyLine()	O(1)
m.show_in_browser()	O(1)
TOTAL	***O(N+N!) en el peor caso*** ***Es posible que se den peores casos de complejidad, depende de la suma de complejidades con las de los requerimientos individuales.***

Pruebas Realizadas

Procesadores	**Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz**
Memoria RAM	**8.00 GB (7.88 GB usable)**
Sistema Operativo	**Windows 10**

Entrada	Tiempo (ms)
airports-2022/flights-2022	37.844

Análisis

Las complejidades a las que nos podemos enfrentar en general son O(N logN), O(N) y O(N+N!) sumarle O(N + N!) del requerimiento 8 por si solo puede ser considerado una constante de modo que el orden global no cambiaría, el tiempo de ejecución real sí.

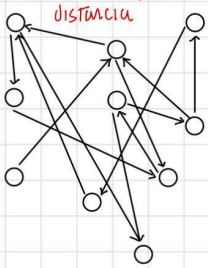
Análisis General reto 4:

En general todos los grafos que usamos en este reto son dispersos, probablemente porque todos tienen los mismos vértices independiente de si se van a usar o no. El menos disperso de los grafos es el militar, claro está porque tiene menos arcos. Ahora, si tomáramos por ejemplo el requerimiento 4 y el 5 que recorren todo el grafo que les corresponde si bien es cierto hay una diferencia de tiempo minúscula, ésta favorece al requerimiento 4. Esto nos da un vistazo a entender que la diferencia mínima en densidad entre los grafos no implica realmente una diferencia en términos temporales. Claro está que el algoritmo de grafos que se use en cada requerimiento va a tener una complejidad diferente, y se comportará de diferentes formas con respecto a la densidad del grafo en el que se aplique. Por cómo está hecha la carga de datos bajo ningún concepto sería buena idea usar matrices de adyacencia pues siempre el número de vértices va a ser el mismo para todos los grafos. Entonces los grafos siempre van a ser dispersos, obviamente desde la teoría aplicada al caso habrá excepciones pero por el tipo de datos realmente es inviable que por ejemplo la mayoría de vuelos sean comercial de modo que haya un grafo extremadamente denso y el resto dispersos (no tendría sentido hacer consultas sobre los grafos extremadamente dispersos).

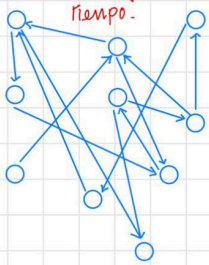
Diagrama estructuras de datos utilizadas:

Aviación Tipo militar

Grafo dirigido distancia

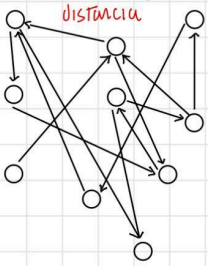


Grafo dirigido tiempo.

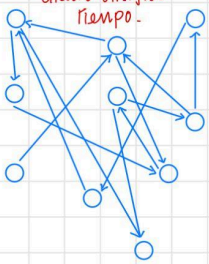


Aviación Tipo comercial.

Grafo dirigido distancia

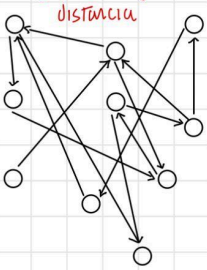


Grafo dirigido tiempo.

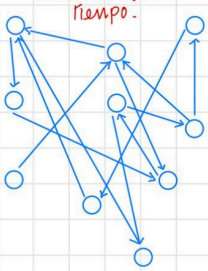


Aviación Tipo Curyu.

Grafo dirigido distancia



Grafo dirigido tiempo.



mapus aeropuertos

mapa de vuelos indexados.