ANÁLISIS DEL RETO

Estudiante 1, código 1, email 1

Estudiante 2, código 2, email 2

Estudiante 3, código 3, email 3

Carga de Datos

¿Como funciona?

```
def new_data_structs():
    data_struct={
    "airport_map": None,
    "graph_time" : None,
    "flights" : None,
    "graph_distance": None,
    "lista_comercial": None,
    "grafo_comercial_busqueda": None
    }
    data_struct["airport_map"]=mp.newMap(numelements=450)
    data_struct['graph_time']=mp.newMap(numelements=5)
    data_stucts_aux(data_struct['graph_time'],False)
    data_struct['flights']=mp.newMap(numelements=5)
    data_stucts_aux(data_struct['flights'],True)
    data_struct['graph_distance']=mp.newMap(numelements=5)
    data_stucts_aux(data_struct['graph_distance'],False)
    data_struct['lista_comercial'] = lt.newList('ARRAY_LIST')
    return data_struct
# Funciones para agregar informacion al modelo
def data_stucts_aux(data_struct,bool):
    if bool:
        mp.put(data_struct,'AVIACION_CARGA',mp.newMap(numelements=450))
        mp.put(data_struct,'MILITAR',mp.newMap(numelements=450))
        mp.put(data_struct,'AVIACION_COMERCIAL',mp.newMap(numelements=450))
    else:
        mp.put(data_struct,'AVIACION_CARGA',gr.newGraph(directed=True,size=450))
        mp.put(data_struct,'MILITAR',gr.newGraph(directed=True,size=450))
        mp.put(data_struct,'AVIACION_COMERCIAL',gr.newGraph(directed=True,size=450))
```

Crea 4 mapas. Uno es el mapa de vuelos que tiene como llave los aeropuertos y como valor los vuelos que salen de este. Otro mapa es uno que tiene como llave los aeropuertos y valores tiene todos los datos del aeropuerto incluyendo las frecuencias de los 3 tipos. Los ultimos 2 son mapas que como valores tienen carga comercial y militar. Asociados a estas llaves hay grafos de la respectiva categoria. Uno de estos grafos asocia los vertices con tiempo y el otro con distancia.

Comparación con retos pasados:

Requerimiento <<3>>

```
def req_6_intento (data_structs, n, bool=True):

    paths= lt.newList()
    grafo = data_structs['grafo_comercial_busqueda']
    lista = data_structs['lista_comercial']
    mayor = lt.getElement(lista, 0)
    if bool == True:
        for poss in range (1, n+1):
            element = lt.getElement(lista, poss+1)
            if djk.hasPathTo(grafo, element['ICAO']):
                path = djk.pathTo(grafo,element['ICAO'])
                lista_a = path_to_list(path, data_structs)
                peso = djk.distTo(grafo, element['ICAO'])
                tupla = element,lista_a, peso
                lt.addLast(paths, tupla)
    else:
        n = lt.size(lista)
        for poss in range (2, n+1):
            element = lt.getElement(lista, poss)
            if djk.hasPathTo(grafo, element['ICAO']):
                path = djk.pathTo(grafo,element['ICAO'])
                lista_a = path_to_list(path, data_structs)
                peso = djk.distTo(grafo, element['ICAO'])
                tupla = element,lista_a, peso
                lt.addLast(paths, tupla)
    return paths, mayor
```

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

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Este requerimiento se llama “intento_requerimiento_6”, no por accidente, pues es muy similar a este. En este requerimiento buscamos la red de apoyo desde el aeropuerto con mayor frecuencia comercial hasta cada aeropuerto. Lo que se hizo en este requerimiento en la carga de datos fue hacer una lista de aeropuertos comerciales ordenada por frecuencia (esto es parte de la solución del requerimiento 6) y un

grafo Dijkstra desde el aeropuerto con mayor frecuencia. Con esto en cuenta, el requerimiento 3 usa la función disTo[] para el kilometraje desde el vértice de origen hasta aeropuerto destino en la lista. Por otro lado, usa la función pathTo[] para conocer todas las paradas (escalas) que se necesitan llegar al destino. Esto se mete a una lista para después tabularla.

Entrada	No se necesitan parametros
Salidas	Lista, y areopuerto mayor
Implementado (Sí/No)	Si, lucas Ruiz

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(n)
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).

Entrada	**Tiempo (s)**
Nada	0.012
Nada	0.013
Nada	0.011

Análisis

La tabla de análisis muestra unos tiempos consistentes. Son tiempos relativamente cortos lo que sigue con nuestro análisis de complejidad de la función.

Comparación con retos pasados

Este reto es completamente diferente a los pasados. Sin embargo, podemos ver usos de los TADs como mapas y listas. El uso de árboles en este requerimiento fue nulo pues no fue necesario.

Requerimiento <<3>>

```python
def req_6_intento (data_structs, n, bool=True):

    paths= lt.newList()
    grafo = data_structs['grafo_comercial_busqueda']
    lista = data_structs['lista_comercial']
    mayor = lt.getElement(lista, 0)
    if bool == True:
        for poss in range (1, n+1):
            element = lt.getElement(lista, poss+1)
            if djk.hasPathTo(grafo, element['ICAO']):
                path = djk.pathTo(grafo,element['ICAO'])
                lista_a = path_to_list(path, data_structs)
                peso = djk.distTo(grafo, element['ICAO'])
                tupla = element,lista_a, peso
                lt.addLast(paths, tupla)
    else:
        n = lt.size(lista)
        for poss in range (2, n+1):
            element = lt.getElement(lista, poss)
            if djk.hasPathTo(grafo, element['ICAO']):
                path = djk.pathTo(grafo,element['ICAO'])
                lista_a = path_to_list(path, data_structs)
                peso = djk.distTo(grafo, element['ICAO'])
                tupla = element,lista_a, peso
                lt.addLast(paths, tupla)
    return paths, mayor
```

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

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Este requerimiento se recicla con el tres. Para poder analizarlos, ver el loop cuando el boolean es True. En la carga de datos se hace una lista ordenada de aeropuertos comerciales por la frecuencia. Aparte se encuentra el grafo Dijkstra desde el primer aeropuerto de la lista (pues sería el mayor). Ya con esto podemos usas las funciones de Dijkstra pathTo[] y disTo[] los cuales nos dan los vértices de parada y las distancias en kilómetros del recorrido. Esto se mete a una lista y se devuelve. El punto de la lista ordenada es poder recorrerla dependiendo el número de aeropuertos que pida el usuario y no perder complejidad buscándolos.

Entrada	Numero de aeropuertos que se quieren encontrar camino
Salidas	Lista, y aeropuerto mayor
Implementado (Sí/No)	Si

Análisis de complejidad

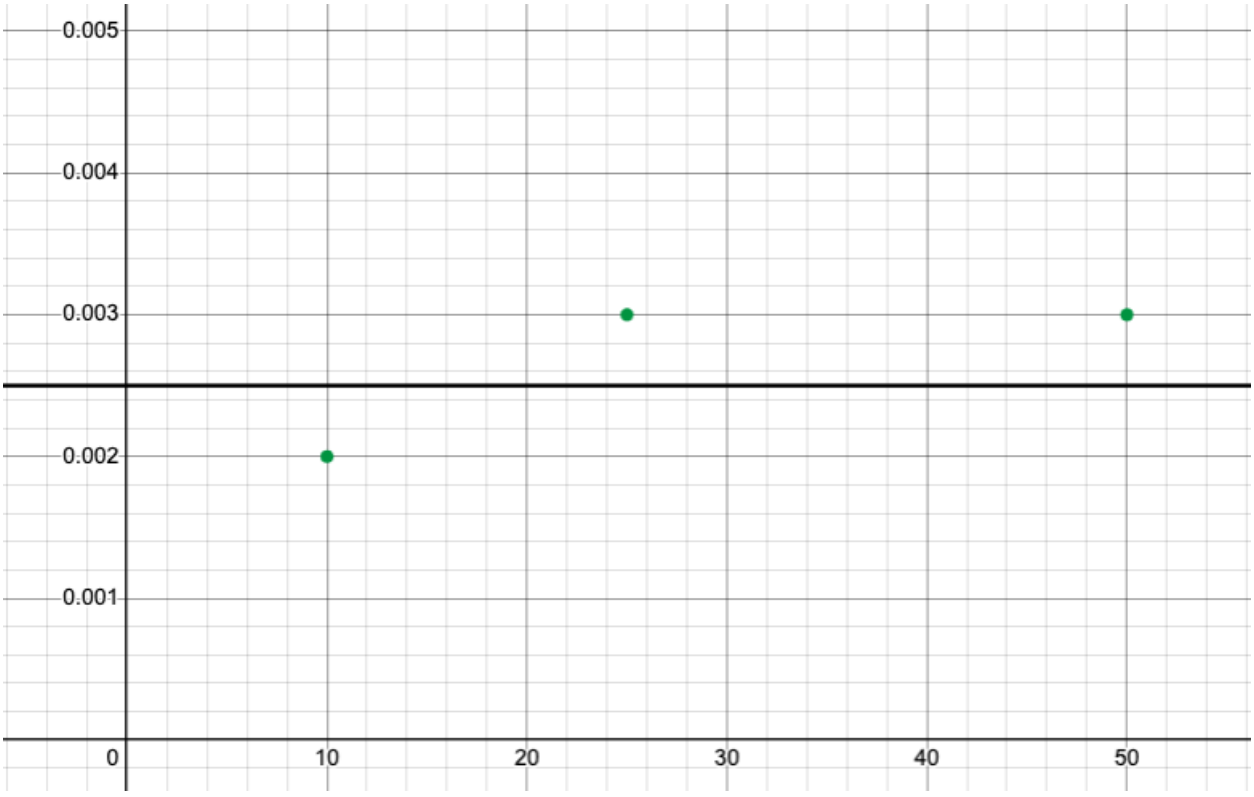
Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(n)
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).

Entrada	**Tiempo (s)**
10	0.002
25	0.003
50	0.003



Análisis

La tabla de análisis muestra unos tiempos consistentes. Son tiempos relativamente cortos lo que sigue con nuestro análisis de complejidad de la función. Por otro lado, las entradas representan un nivel bajo de aeropuertos y el algoritmo no tiene una complejidad lo suficientemente alta para que se demuestre un cambio. Puede verse mayores cambios de tiempo si se analizan miles de areopuertos. En la tabla podemos ver un pequeño cambio, pero no altera el análisis

Comparación con retos pasados

Este reto es completamente diferente a los pasados. Sin embargo, podemos ver usos de los TADs como mapas y listas. El uso de árboles en este requerimiento fue nulo pues no fue necesario.

Requerimiento <<4>> Julian Pinto

```python
def grafo_busqueda_carga(data_structs):
    lista = data_structs['lista_carga']
    mayor = lt.firstElement(lista)
    ICAO = mayor['ICAO']
    mapa = data_structs['graph_distance']
    grafo = me.getValue(mp.get(mapa,'AVIACION_CARGA'))
    data_structs['grafo_carga_busqueda'] = djk.Dijkstra(grafo, ICAO)

def lista_de_importancia_carga(data_structs, n = 50):
    grafo = gr.vertices(me.getValue(mp.get(data_structs['graph_time'],'AVIACION_CARGA')))
    numero = mp.size(data_structs['airport_map'])
    for aeropuerto in lt.iterator(grafo):
        frecuencia=lt.size(gr.adjacents(me.getValue(mp.get(data_structs['graph_time'],'AVIACION_CARGA')),aeropuerto))
        me.getValue(mp.get(data_structs['airport_map'],aeropuerto))['Frecuencia Comercial']=int(frecuencia)
    lista = maximo(data_structs['airport_map'],'Frecuencia Carga',numero)
    return lista


def req_4(data_structs, n, bool=False):
    paths= lt.newList()
    grafo = data_structs['grafo_carga_busqueda']
    lista = data_structs['lista_carga']
    mayor = lt.getElement(lista, 0)
    if bool == True:
        for poss in range (1, n+1):
            element = lt.getElement(lista, poss+1)
            if djk.hasPathTo(grafo, element['ICAO']):
                path = djk.pathTo(grafo,element['ICAO'])
                lista_a = path_to_list(path, data_structs)
                peso = djk.distTo(grafo, element['ICAO'])
                tupla = element,lista_a, peso
                lt.addLast(paths, tupla)
    else:
        n = lt.size(lista)
        for poss in range (2, n+1):
            element = lt.getElement(lista, poss)
            if djk.hasPathTo(grafo, element['ICAO']):
                path = djk.pathTo(grafo,element['ICAO'])
                lista_a = path_to_list(path, data_structs)
                peso = djk.distTo(grafo, element['ICAO'])
                tupla = element,lista_a, peso
                lt.addLast(paths, tupla)
    return paths, mayor
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Este requerimiento se basa completamente en el requerimiento 6 ya que de la misma forma que el 1 y el 7, estos dos se pueden hacer con una misma funcion. Aca opte por usar diferentes funciones ya que este requerimiento necesitaba una carga un poco diferente. Lo primero que hace es creae un grafo dijkstra de carga y con este encuentra todas las rutas posibles.

Entrada	Nada
Salidas	Lista, y aeropuerto mayor
Implementado (Sí/No)	Si Julian Pinto

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(n)
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).

Entrada	**Tiempo (s)**
Prueba 1	10 ms
Prueba 2	12 ms
Prueba 3	11 ms

Análisis

Los tiempos se muestran consistentes y esto hace sentido ya que la respuesta en los 3 casos deberia ser la misma

Comparación con retos pasados

Este reto es completamente diferente a los pasados. Sin embargo, podemos ver usos de los TADs como mapas y listas. El uso de árboles en este requerimiento fue nulo pues no fue necesario.

Requerimiento <<1 y 7>>

Ya que el requerimiento 1 y 7 eran iguales a diferencia que el requerimiento 1 no pedia el menor camino decidimos abordar ambos con un mismo modelo y ambos retornan lo mismo

```python
def req_7(data_structs,inicio,final):
    origen,distorigen=distancia(data_structs,inicio[0],inicio[1])
    destino,distdestino=distancia(data_structs,final[0],final[1])
    distorigen=float(distorigen)
    distdestino=float(distdestino)
    estruct=me.getValue(mp.get(data_structs['graph_time'],'AVIACION_COMERCIAL'))
    estruct2=me.getValue(mp.get(data_structs['graph_distance'],'AVIACION_COMERCIAL'))
    if distdestino>30 or distorigen>30:
        return (origen,destino,distorigen,distdestino)
    else:
        grafo = djk.Dijkstra(estruct,origen)
        tiempo=djk.distTo(grafo,destino)
        me.getValue(mp.get(data_structs['airport_map'],origen))
        camino=djk.pathTo(grafo,destino)
        caminoc=djk.pathTo(grafo,destino)
        dist=0
        arptoa=st.pop(caminoc)
        while not st.isEmpty(caminoc):
            arptob = st.pop(caminoc)
            dist+=gr.getEdge(estruct2,arptoa,arptob)
            arptoa=arptob
        dist+=distdestino
        dist+=distorigen
        st.push
        camino=path_to_list(camino,data_structs)
        camino=req_7aux(camino)
        tupla=camino,tiempo,dist,origen,destino,lt.size(camino),origen,destino
        return tupla
def req_7aux(lista,param=['NOMBRE','ICAO', 'CIUDAD']):
        retorno=lt.newList(datastructure='ARRAY_LIST')
        for dato in lt.iterator(lista):
            dic = {}
            for i in param:
                dic[i] = dato[i]
            lt.addLast(retorno, dic)
        return retorno
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Este requerimiento empieza validando si las distancias estan cerca a algun aeropuerto, en caso que alguna de las 2 no este a 30 km o menos el modelo no hace la busqueda y se limita a retornar los 2 aeropuertos mas cercanos con su distancia a las coordenadas. Si cumplen el requisito de distancia se ejecuta una busqyeda de dijksta en la cual busca el menor camino y lo reconstruye en una lista. Adicionalmente se va sumando la distancia a un contador para el retorno

Entrada	coordenadas
Salidas	Ruta menor de los 2 aeropuertos mas cercanos a las coordenadas con tiempo y distancia
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(n)
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).

Entrada	**Tiempo (s)**
Coordenadas invalidas	15 ms
Coordenadas validad	25 ms

Análisis

Aunque cuando las coordenadas son validas el computador se debe esforzar mas, no es mucho mas ya que este solo depende del tamaño del camino minimo de un vertice a otro que la mayoria de veces es un numero pequeño

Comparación con retos pasados

Este reto es completamente diferente a los pasados. Sin embargo, podemos ver usos de los TADs como mapas y listas. El uso de árboles en este requerimiento fue nulo pues no fue necesario.