Documento de análisis del Reto 4

Estudiante 1- Maria Alejandra Moreno Bustillo Cod 202021603
Estudiante 2 - Juliana Delgadillo Cheyne Cod 202020986

I. Carga de datos

Para la carga de los datos se hizo uso de muchas de las estructuras de datos vistas a lo largo de este semestre. Para establecer las relaciones necesarias de los aeropuertos con las respectivas rutas que cada uno cubre se hizo uso de un mapa dirigido y un mapa no dirigido. En el mapa dirigido ('AirportRoutesD') se definieron como vértices los aeropuertos mediante su código IATA y los arcos definían todas las rutas existentes que cubría cada aeropuerto, teniendo en cuenta se creaba únicamente una vez cada ruta posible, es decir, establecimos un condicional que nos permitía que en el caso de que exista una misma ruta recorrida por diferentes aerolíneas, el arco se crearía una única vez. Con este grafo podríamos llevar a cabo los algoritmos necesarios para cumplir con algunos requerimientos y que únicamente funcionan con grafos dirigidos, por ejemplo, para realizar el requerimiento 2 en el cual hay que obtener los componentes fuertemente conectados del grafo es necesario hacer uso del algoritmo de kosaraju y este sirve únicamente en grafos dirigidos. Por otro lado, creamos otro grafo con la misma estructura, pero no dirigido, en el cual los arcos solo se creaban si había un vuelo tanto de ida como de vuelta entre ese par de aeropuertos. Por otro lado, creamos un mapa auxiliar llamado 'AirportIATAS' en el cual se relacionan los códigos IATAS de cada aeropuerto con su respectiva información, de esta forma, cuando se hagan algoritmos en los grafos, con los códigos IATAS en los vértices podremos obtener la información de los aeropuertos con esos códigos. Luego, para la solución del requerimiento 3, había que hacer uso de las ciudades y con el fin de poder encontrar rápidamente el aeropuerto más cercano a cada ciudad, en la carga de datos se creó un mapa llamado 'Cities-Airport' en el cual se tienen como llaves los nombres de cada ciudad y asociado se encuentra el aeropuerto más cercano a esta, de esta forma podemos determinar en la carga de datos los aeropuertos más cercanos y sea más fácil y rápido de obtener para el requerimiento que lo necesite. Por otro lado, tenemos otro mapa con función auxiliar que relaciona las ciudades con su respectivo número ID y otro mapa que mediante relaciona el ID con la información de cada ciudad, de ese modo se puede obtener la información asociada a una ciudad en específico. Por último, se crearon dos listas que contienen todos los aeropuertos ordenados de mayor a menor según el grado que tienen en el grafo, esto se hizo con el fin de que se pudieran optimizar los tiempos de respuesta del requerimiento 1 que necesita de esta información y ya estando esta cargada desde la carga de datos, los tiempos de ejecución serán mucho menores.

II. Requerimiento 1 (Grupal) – Encontrar puntos de interconexión aérea

En este requerimiento se deseaba encontrar todos los aeropuertos que eran punto de interconexión en rutas aéreas. Esta información era solicitada para la red de cada grafo es decir el dirigido y el no dirigido. Para cumplir con este objetivo se crearon dos listas de tipo ARRAY_LIST (una por grafo) en las que se almacenaron diccionarios que contienen información de cada aeropuerto encontrado. Para el grafo dirigido, se creó una función que inicia llamando a dicho grafo y guardando en una lista todos sus vértices, luego se realiza un bucle for en el cual se itera por cada vértice encontrado. Dentro de dicho ciclo, se calcula el número de arcos que llegan y salen de un vértice por medio de las funciones indegree y outderee correspondientemente, luego se suman estos dos valores para así obtener el número total de arcos por vértice

que corresponde también al número de interconexiones. A continuación, se hace un llamado a la función relateIATA, en la cual se relaciona el valor del vértice con un código IATA existente en el mapa AirportIATAS, como resultado se logra obtener toda la información contenida en los archivos sobre un aeropuerto en específico. Finalmente, se crea por vértice un diccionario que contiene nombre del aeropuerto, IATA, ciudad, país y número total de interconexiones, el cual es añadido con cada iteración a la lista AirpotsInterconnected. Una vez creada la lista se ejecuta sobre esta una función de ordenamiento sort en la que se hace una comparación del número de interconexiones de cada aeropuerto y según esto se organizan de mayor a menor. El mismo proceso descrito anteriormente es utilizado con el grafo no dirigido, la única diferencia es que en esta función no se realiza la suma de arcos de llegada y salida para determinar el número total de interconexiones, puesto que, por ser un grafo no dirigido, todas las conexiones a los vértices tienen ambos sentidos. Por esta razón, para calcular el número de arcos de cada vértice en el grafo no dirigido basta con encontrar el número de arcos asociados al vértice por medio de la función degree. Dado que las dos listas fueron creadas en la carga de datos, en la función de consulta para el requerimiento 1 solo es necesario llamar cada arreglo el cual ya contiene toda la información completa y ordenada que se necesita para cumplir con el requerimiento, por esta razón la complejidad calculada para el requerimiento es de O(1).

Creación de listas en carga de datos:

```
def addInterconnections(analyzer):
    graph = analyzer['AirportRoutesD']
    vertex_list = gr.vertices(graph)
    #info_list = analyzer["AirpotsInterconnected"]

    for vertex in lt.iterator(vertex_list):
        info = relateIATA(analyzer, vertex)
        arocos_llegada = gr.indegree(graph, vertex)
        arcos_salida = gr.outdegree(graph, vertex)
        total_arcos = arocos_llegada + arcos_salida
        datos = {"Aeropueto": info["Name"],
                 "IATA": vertex,
                 "Ciudad": info["City"],
                 "Pais": info["Country"],
                 "TotalConnections": total_arcos,
                 }
        lt.addLast(analyzer["AirpotsInterconnected"], datos)

    sortInterconnected(analyzer["AirpotsInterconnected"])


def addInterconnectionsND(analyzer):
    graphND = analyzer['AirportRoutesND']
    vertex_listND = gr.vertices(graphND)
    #info_listND = ["AirpotsInterconnectedND"]

    for vertex in lt.iterator(vertex_listND):
        info = relateIATA(analyzer, vertex)
        total_arcosND = gr.degree(graphND,vertex)
        datos = {"Aeropueto": info["Name"],
                 "IATA": vertex,
                 "Ciudad": info["City"],
                 "Pais": info["Country"],
                 "TotalConnections": total_arcosND,
                 }
        lt.addLast(analyzer["AirpotsInterconnectedND"], datos)

    sortInterconnected(analyzer["AirpotsInterconnectedND"])
```

Función de Consulta:

```
def getInterconnections(analyzer):

    info_list = analyzer['AirpotsInterconnected']
    info_listND = analyzer['AirpotsInterconnectedND']

    return info_list, info_listND
```

III. Requerimiento 2 (Grupal) – Encontrar clústeres de tráfico aéreo

En el requerimiento 2 era solicitado encontrar clústeres de tráfico aéreo e indicar si dos aeropuertos pertenecen al mismo cluster. Estos clusters hacen referencia a componentes fuertemente conectados en el grafo. Para encontrar estos componentes se utilizó el algoritmo de kosaraju en el grafo dirigido y luego de encontrar la cantidad de componentes fuertemente conectados, se utilizaron los dos aeropuertos ingresados y mediante la función scc.stronglyConnected se verificó que los dos aeropuertos se encontraran en el mismo componente conectado a partir de la respuesta obtenida del algoritmo de kosaraju. Para optimizar los tiempos de respuesta de este algoritmo, este se ejecuta en la carga de datos y así cuando se tuviera que ejecutar el requerimiento solo fuera necesario encontrar la cantidad de clusters y si los dos aeropuertos se encontraban juntos o no, reduciendo así significativamente los tiempos de ejecución. La complejidad final de este algoritmo sería principalmente la que le corresponde a la ejecución del algoritmo de kosaraju, pues el resto de requerimientos se ejecutan rápidamente en un orden muy cercano a O(1), y la complejidad asociada al algoritmo de kosaraju es de O(V+E), por ende, la complejidad final del algoritmo implementado para la solución de este requerimiento es de O(V+E).

```
def getcluster(analyzer):

    cluster = scc.KosarajuSCC(analyzer['AirportRoutesD'])

    return cluster

def getClusterNum(cluster):

    cluster_num = scc.connectedComponents(cluster)

    return cluster_num

def getTraficClustersCon(cluster, IATA1,IATA2):

    airports_connected = scc.stronglyConnected(cluster, IATA1, IATA2)

    return airports_connected
```

IV. Requerimiento 3 (Grupal) – Encontrar la ruta más corta entre ciudades

Para el requerimiento 3, se buscaba establecer la ruta mínima, en términos de distancia, para viajar entre dos ciudades las cuales son ingresadas por el usuario. Para cumplir con este requerimiento fue necesario implementar cuatro funciones principalmente que permitirán encontrar la ruta más corta entre dos puntos. En primer lugar, se pide al usuario que ingrese la ciudad origen y a partir de ello se utiliza la función getCities la cual relaciona el nombre de la ciudad con su respectivo ID, esto se hace encontrando la pareja llave-valor en el mapa Cities-ID y extrayendo de su valor la información contenida en ID (O(1.5)). En segunda instancia, se crea la función ClosestairportCity que llama al mapa de ciudades y extrae el valor de la pareja que contiene el ID de la ciudad establecido en la anterior función, esto se lleva a cabo con ayuda de las funciónes get y getValue las cuales tienen una complejidad temporal de O(1.5) . Luego, se accede a

dicho valor y se retorna únicamente la información correspondiente a AirportClosest, lo cual corresponde al aeropuerto de origen de la ruta que se desea encontrar. En segunda instancia, se crea la función DijkstraAirport, cuyo único propósito es implementar el algoritmo dijkstra sobre el grafo dirigido AirportRoutesD y un vértice que en este caso corresponde a el aeropuerto de origen encontrado anteriormente, de este algoritmo implementado en la librería DISCLIB se obtiene una complejidad temporal de O(ElogV) en el peor caso. A continuación, se pide al usuario que ingrese la ciudad de destino y con dicho input se ejecutan nuevamente las funciones getCities para determinar el ID de dicha ciudad y ClosestairportCity para encontrar el aeropuerto de destino de la ruta que se está analizando. Finalmente, se crea una función llamada getShortestRoute la cual recibe dos paramentos; la estructura generada al implementar el algoritmo dijkstra a partir del aeropuerto origen y el aeropuerto de destino. En esta función, se utiliza hasPathTo para determinar si existe un camino entre el componente inicial y el vértice de destino y si y solo si el resultado es True, se crea una pila con el camino encontrado (pathTo) y se calcula el consto total de dicho camino (distTo), con esta información ya es posible cumplir con el requerimiento. Teniendo él cuenta el proceso que se llevó a cabo, es posible calcular una complejidad temporal de O(1.5) + O(ElogV), entre estas dos complejidades principales la mayor corresponde a O(ElogV) y esta sería la complejidad temporal en notación Big O.

```python
def getCities(analyzer, name):

    entry = m.get(analyzer['Cities-ID'], name)
    value = me.getValue(entry)

    return value['ID']
```

```python
def ClosestairportCity(analyzer,city_id):

    city_closest_map = analyzer['Cities-Airport']
    airportentry = m.get(city_closest_map,city_id)
    airportvalue = me.getValue(airportentry)
    airportIATA = airportvalue['AirportClosest']

    return airportIATA


def DijkstraAirport(analyzer, airport):

    shortest_routes = djk.Dijkstra(analyzer['AirportRoutesD'], airport)

    return shortest_routes


def getShortestRoute(dijkstra, airport2):

    if djk.hasPathTo(dijkstra,airport2):

        dijk_route = djk.pathTo(dijkstra,airport2)

        dist_total = djk.distTo(dijkstra, airport2)

        return dijk_route,dist_total
```

V. Requerimiento 4 (Grupal) – Utilizar las millas del viajero

Para este requerimiento se espera encontrar un viaje redondo que cubra la mayor cantidad de ciudades posible para utilizar las millas de viajero. Para cumplirlo, se creó a partir del grafo no dirigido un MST haciendo uso del algoritmo prim y un árbol DFS utilizando la función DepthFirstSearch. El MST, nos permitió extraer todos los aeropuertos identificados dentro de este indicados como vertexA y vertexB, los cuales fueron almacenados en un arreglo. Por otro lado, el árbol DFS que tiene como raíz el aeropuerto ingresado por el usuario, nos permitió ejecutar pathTo con el fin de encontrar el camino entre la raíz y cada uno de los vértices que se encontraron en el MST. Una vez con esta información, se notó que las rutas encontradas se superponían entre sí, por lo cual fue necesario unificar aquellas rutas que generaran un camino redondo y eliminar caminos repetidos, para así identificar la rama más extensa que abarcará la máxima cantidad de ciudades posible a partir de la ciudad origen ingresada como input. Luego, se llevó a cabo un formateo del camino encontrado, guardando en tuplas el origen y destino para cada arco de la ruta, lo que permite observar los nodos por los que se viaja de una forma más sencilla y amigable. Finalmente, se hizo el cálculo total de los pesos de cada uno de los arcos que hacen parte de la rama más larga y se multiplico por dos, puesto que era necesario considerar el viaje de ida y vuelta; y se analizó cuantas millas le hacían falta o le sobraban a la persona, de acuerdo a la cantidad de millas de viajero que ingreso. En cuanto a la complejidad para cumplir con lo requerido, es necesario tener en cuenta el uso del algoritmo prim (O(ElogV)), las ejecuciones sobre el árbol DFS (O(V+E))) y los bucles for que fueron utilizados para guardar y organizar toda la información recolectada (O(k*j*z*w) + O(2w))); cabe mencionar que los recorridos de los bucles corresponden a interacciones delimitadas lo cual hace que estos no sean más extensos de los estrictamente necesario. Teniendo en cuanta lo anterior la mayor complejidad corresponde a O (k*j*z*w) + O(2w), por lo cual en notación BigO esta sería la complejidad temporal del requerimiento 4.

```python
def planViajero(analyzer, origen, distancia):

    lista_nodos = lt.newList('ARRAY_LIST')
    posibles_caminos = lt.newList('ARRAY_LIST')#####
    largestBranch = lt.newList('ARRAY_LIST')

    graphND = analyzer['AirportRoutesND']
    mst = prim.PrimMST(graphND)
    tree = mst["mst"]

    weight = prim.weightMST(graphND, mst)

    ruta = dfs.DepthFirstSearch(graphND, origen)
    for i in lt.iterator(tree):
        nodo = i['vertexA']
        nodo2 = i['vertexB']

        #lista de nodos
        exist = lt.isPresent(lista_nodos,nodo)
        exist2 = lt.isPresent(lista_nodos,nodo2)
        if (exist == 0):
            lt.addLast(lista_nodos, nodo)
        if (exist2 == 0):
            lt.addLast(lista_nodos, nodo2)
```

```python
        #encontrar camino
        path = dfs.pathTo(ruta, nodo)
        path2 = dfs.pathTo(ruta, nodo2)
        if path and path2:
            posibles_caminos = lt.newList('ARRAY_LIST')
            lt.addLast(posibles_caminos,path)
            lt.addLast(posibles_caminos,path2)

        for i in lt.iterator(posibles_caminos):
            union_caminos = lt.newList('ARRAY_LIST')
            #print(i)
            for iata in lt.iterator(i):
                #hacer pareja
                posicion = (lt.isPresent(i, iata))+1
                if posicion <= lt.size(i):
                    elemento = lt.getElement(i,posicion)
                    pareja_iata = (iata, elemento)
                    lt.addLast(union_caminos,pareja_iata)
            for m in (union_caminos["elements"]):
                exist = lt.isPresent(largestBranch,m)
                if (exist == 0):
                    lt.addLast(largestBranch, m)
```

```
#distancia total
lista_pesos = lt.newList('ARRAY_LIST')
total_pesos = 0
for element in lt.iterator(largestBranch):
    edge = gra.getEdge(graphND, element[0], element[1])
    peso = edge['weight']
    lt.addLast(lista_pesos,peso)
for peso in lt.iterator(lista_pesos):
    total_pesos = total_pesos + peso

distancia_total = total_pesos*2
print(distancia_total)

nodesConnected = lt.size(lista_nodos)
largestBranch = largestBranch["elements"]
milles = ((distancia * 1.6) - distancia_total)/1.6

return nodesConnected, weight, largestBranch, milles, distancia_total
```

VI. Requerimiento 5 (Grupal) – Cuantificar el efecto de un aeropuerto cerrado

En el requerimiento 5 se deseaba determinar el impacto que tendría que un aeropuerto no esté en funcionamiento, dicho aeropuerto es ingresado como input por el usuario por medio del código IATA. Para cumplir con lo solicitado se creó una función en la cual se accede al grafo dirigido y se encuentran dentro de este todos los vértices adyacentes al IATA indicado por el usuario. Lo anterior se logra ejecutando adjacents lo cual retorna una lista con todas las adyacencias identificadas. Luego se determina el tamaño de esta lista (size), lo cual corresponde al número de aeropuertos afectados. Con la lista de adyacencias y su respectivo tamaño, ya es posible dar respuesta al requerimiento. De acuerdo con lo anterior, se calcula una complejidad de O (1) o O (1.5) pues la lista de adyacencias se piensa como una tabla de hash y esa es la complejidad temporal de obtener el valor asociado a la llave que es el IATA del aeropuerto en consulta. Por último, algo adicional que es necesario realizar en el print del requerimiento es realizar un for loop de los primeros y últimos 3 aeropuertos afectados y dentro de estos se realiza un m.get para poder obtener la información relacionada con cada uno de los aeropuertos afectados, lo cual le otorga otra vez una complejidad de O (1). Para concluir, la complejidad temporal de este requerimiento es constante, es decir, no depende de la cantidad de datos que se estén procesando y sería de O (1).

```
def getAffectedAirports(analyzer, IATA):

    adj = gr.adjacents(analyzer['AirportRoutesD'],IATA)
    size = lt.size(adj)

    return  adj, size
```

```
if tamano > 6:
    first3 = lt.subList(list, 1, 3)
    last3 = lt.subList(list, tamano-2 , 3)

    for airport in lt.iterator(first3):
        entry = m.get(analyzer['AirportIATAS'], airport)
        value = me.getValue(entry)
        print('Nombre: ' + value['Name'] + ', Ciudad: ' + value['City'] + ', IATA: ' + value['IATA'])

    for airport in lt.iterator(last3):
        entry = m.get(analyzer['AirportIATAS'], airport)
        value = me.getValue(entry)

        print('Nombre: ' + value['Name'] + ', Ciudad: ' + value['City'] + ', IATA: ' + value['IATA'])

else:

    for airport in lt.iterator(lista):
        entry = m.get(analyzer['AirportIATAS'], airport)
        value = me.getValue(entry)

        print('Nombre: ' + value['Name'] + ', Ciudad: ' + value['City'] + ', IATA: ' + value['IATA'])
```

VII. Requerimiento 6 (Grupal) - Comparar con servicio WEB externo (Bono)

Para este requerimiento se hizo uso de un servidor web externo el cual se le pasaba de parámetro las latitudes y longitudes de la ciudad a buscar y en un radio especificado, este servidor encontraba en orden los aeropuertos más relevantes y que estuvieran dentro del radio especificado. La operación que este realizaba era de aproximadamente O(1), pues los tiempos de respuesta no dependen de la cantidad de datos cargados, estos por lo general eran constantes. Con esta información, el usuario debía ingresar el código IATA del aeropuerto que estuviera con mayor relevancia en el listado obtenido y se repetía el mismo proceso para el aeropuerto de destino. Con esto, se realizó el mismo procedimiento que en el requerimiento 3 en el cual se hacia el algoritmo de Dijkstra a partir del aeropuerto de origen y con este se buscaba el camino más corto hacia el aeropuerto de destino. Los datos obtenidos son diferentes, puesto a que el criterio de selección de los aeropuertos más cercanos a una ciudad es diferente y ahora se eligen en base a la relevancia de estos. Por último, la complejidad final de este requerimiento sería la misma que del requerimiento 3, es decir, O(ElogV).

```
elif int(inputs[0]) == 8:

    'Requerimiento 6: Comparar con servicio WEB externo'

    getAccessToken.accestoken()

    city_origin = input('Ingrese la ciudad de origen que desea: ')

    ciudades_o = controller.getCities(analyzer, city_origin)

    if lt.size(ciudades_o) > 1:

        print('Se encontraron los siguientes códigos de ciudades con el mismo nombre que usted seleccionó: ')

        for ciudad in lt.iterator(ciudades_o):

            print(ciudad['city']+', '+ ciudad['country'] + ', ' + ciudad['lat'] + ', ' + ciudad['lng'] + ', ' + ciudad['id'])

        ciudad_o_codigo = input('De las anteriores ciudades, seleccione el código de la que quiere como ciudad de origen: ')

    else:

        ciudad_o_codigo = ciudades_o['elements'][0]['id']

    origen_latsylons = controller.Req6City(ciudad_o_codigo, analyzer)

    token = input('Ingrese el access token: ')

    queryAPI.Req6ClosestAirport( token,origen_latsylons[0], origen_latsylons[1])

    iata_O = input('Ingrese el código IATA del aeropuerto más cercano y más relevante: ')

    dijkstra_airport_o = controller.DijkstraAirport(analyzer, iata_O)

    city_destiny = input('Ingrese ciudad de destino que desea: ')

    ciudades_d = controller.getCities(analyzer, city_destiny)
```

```
    if lt.size(ciudades_d) > 1:

        print('Se encontraron los siguientes códigos de ciudades con el mismo nombre que usted seleccionó: ')
        for ciudad in lt.iterator(ciudades_d):

            print(ciudad['city']+', '+ ciudad['country'] + ', ' + ciudad['lat'] + ', ' + ciudad['lng'] + ', ' + ciudad['id'])

        ciudad_d_codigo = input('De las anteriores ciudades, seleccione el código de la que quiere como ciudad de destino: ')
    else:

        ciudad_d_codigo = ciudades_d['elements'][0]['id']

    destination_latsylons = controller.Req6City(ciudad_d_codigo, analyzer)

    print(destination_latsylons)

    queryAPI.Req6ClosestAirport(token,destination_latsylons[0], destination_latsylons[1])

    iata_D = input('Ingrese el código IATA del aeropuerto más cercano y más relevante: ')

    respuesta = controller.getShortestRoute(dijkstra_airport_o, iata_D)

    printReq3(analyzer, respuesta[0],respuesta[1], iata_O, iata_D)
```

```
def Req6City(citycode, analyzer):

    entryinfo = m.get(analyzer['CitiesMapInfo'], citycode)

    value = me.getValue(entryinfo)

    lat = value['lat']
    lon = value['lng']

    return lat, lon
```

```
def Req6ClosestAirport(token, lat, lon):
  # https://developers.amadeus.com/self-service/category/air/api-doc/airport-nearest-relevant/api-reference

  access_token = token #TODO
  headers = {"Authorization": "Bearer " + access_token}
  params = {
    "latitude": lat,
    "longitude": lon,
    "radius": 500
  }

  r = requests.get('https://test.api.amadeus.com/v1/reference-data/locations/airports', headers=headers, params=params)

  print(r.text)     #Solo para imprimir
  #print(r.json()) #Para procesar

  return r.json()
```

```
def DijkstraAirport(analyzer, airport):

    shortest_routes = djk.Dijkstra(analyzer['AirportRoutesD'], airport)

    return shortest_routes


def getShortestRoute(dijkstra, airport2):

    if djk.hasPathTo(dijkstra,airport2):

        dijk_route = djk.pathTo(dijkstra,airport2)

        dist_total = djk.distTo(dijkstra, airport2)

        return dijk_route,dist_total
```