OBSERVACIONES & ANÁLISIS – RETO 4

Req2 - María Alejandra Estrada García Cod. 202021060

Req3 - Santiago Martínez Novoa Cod. 202112020

Análisis de complejidad

Requerimiento 1 (Encontrar puntos de intersección aérea):

- En el requerimiento 1 se busca encontrar los puntos de intersección aérea. Esto se hace mediante el grafo dirigido de vuelos entre aeropuertos, primero se sacan los vértices de todo el grafo y luego por cada uno de los aeropuertos presentes se calculan los vuelos que salen y entran por cada uno de los vértices, se hace una suma de los dos y ese es el valor que se guarda, para que un aeropuerto sea considerado interconectado debe tener por lo menos un vuelo que salga y uno que entre a este. Luego de comprobar si cumple esa condición se decide si se suma o no como aeropuerto interconectado. Finalmente, la lista reducida con todos los aeropuertos y sus respectivas interconexiones se organiza de mayor a menor y se retorna lo encontrado. La función tiene dos complejidades notables, la del ordenamiento que es de O(VlogV) y la de la iteración por cada vértice y por cada aeropuerto O(V+E). Por ser la segunda complejidad linear y la primera linearítmica, la final será de O(VlogV)

```
#
#Req 1
def getRoutesbyAirport(analyzer):
    rock = analyzer['connections_d']
    list_airports = lt.newList(datastructure='ARRAY_LIST')
    airports = gr.vertices(rock)
    interconectados = 0
    for airport in lt.iterator(airports):
        indegree = gr.indegree(rock, airport)
        outdegree = gr.outdegree(rock, airport)
        suma = indegree+outdegree
        info = om.get(analyzer['map_airports'],airport)['value']
        dic = {'IATA': airport, 'num_connections': suma, 'info': info}
        if indegree > 0 and outdegree > 0:
            interconectados += 1
        lt.addLast(list_airports, dic)

    ms.sort(list_airports,compareReq1)
    return list_airports, interconectados
```

Complejidad temporal final:

Requerimiento 2 (Encontrar clústeres de tráfico aéreo):

- En el segundo requerimiento se buscó encontrar los clústeres de tráfico aéreo. Para este requerimiento se utilizó el algoritmo de Kosaraju. Con la función KosarajuSCC, el cual implementa el algoritmo Kosaraju para encontrar los componentes conectados de un grafo dirigido. Por el cual, utilizando el grafo dirigido se buscaron si dos aeropuertos están conectados (en el mismo clúster), y por otro lado el número total de componentes fuertemente conectados en el grafo dirigido (número total de clústeres presentes en la red de transporte aéreo). Para ello se utilizaron las funciones connectedComponents(...) y stronglyConnected(...). Finalmente, el requerimiento 2 tiene una complejidad estimada en O(V+E) para el algoritmo inicial de Kosaraju (scc) y luego de O(1.5) aproximadamente, esto porque se busca en la estructura que crea Kosaraju (map) si los dos vértices dados se encuentran en el mismo cluster.

```
#Req 2
def getConnectionsByIATA(analyzer, IATA1, IATA2):
    grafo_d = analyzer['connections_d']
    analyzer['components'] = scc.KosarajuSCC(grafo_d)
    res1 = scc.connectedComponents(analyzer['components']) #Verificar si dos aeropuertos están en el mismo clúster
    res2 = scc.stronglyConnected(analyzer['components'], IATA1, IATA2) #Número total de cúlsteres presentes en el grafo dirigido
    return res1, res2
```

Complejidad temporal final: O(V + E).

Requerimiento 3 (Encontrar la ruta más corta entre ciudades):

- En el requerimiento 3 se busca la ruta más corta entre ciudades, los cuales llegan como parámetro. Para esto se usa el algoritmo de Dijkstra, y posteriormente se revisa si del primer aeropuerto hay una ruta (path) para el segundo y por medio de la función pathTo(...) el cual retorna la ruta de un aeropuerto a otro, luego la función distTo(...), el cual retorna el costo de llegar a un aeropuerto al otro (vértices). Finalmente, tiene una complejidad estimada en O(V^2), ya que el teorema dice que realiza esta cantidad de operaciones (sumas y comparaciones) para determinar la longitud del camino mas corto de dos vértices en un grafo. Por otro lado, al hacer uso de pathTo() se le suma una complejidad aproximada de O(1.5) ya que se busca en la estructura creada (map) el camino que se requiere.

```
def camino(analyzer, a1, a2):
    analyzer['camino'] = djk.Dijkstra(analyzer['connections_d'], a1)

    if djk.hasPathTo(analyzer['camino'], a2):
        path = djk.pathTo(analyzer['camino'], a2)
        total = djk.distTo(analyzer['camino'], a2)

        return path, total
#
```

Complejidad temporal final: O(V^2)

Requerimiento 4 (Utilizar las millas de viajero):

- En el requerimiento 4 se busca “utilizar las millas de un viajero para realizar un viaje redondo que cubra la mayor cantidad de ciudades que él pueda visitar”. El requerimiento 4 tiene una complejidad estimada de O(ElogV) + O(V+E) que resulta de

la suma del uso del algoritmo Prim(Eager) y DFS respectivamente. Esta función empieza realizando el Prim Mst, encontrando todos los caminos posibles desde un vértice, luego se elige con la ayuda del DFS el camino más largo entre todos los caminos posibles, luego de haber encontrado la ruta más larga se calcula el peso total de ese viaje para así poder restarle las millas y retornar el tamaño del camino más largo, su peso, el camino con todos sus aeropuertos, la distancia máxima y el residuo al restarle las millas. Por ende, su complejidad final sería O(ElogV).

```python
#Req 4
def Lifemiles(analyzer,c_origen, millas):
    distance_km = 1.6*millas
    mst = prim.PrimMST(analyzer['connections_nd'])

    peso = prim.weightMST(analyzer['connections_nd'], mst)
    mst = (prim.edgesMST(analyzer['connections_nd'], mst))['mst']

    for i in lt.iterator(mst):
        addPointConneMst(analyzer, i['vertexA'], i['vertexB'], i['weight'])


    mstAnalyzer = analyzer['mst']
    vert = gr.vertices(mstAnalyzer)
    num = lt.size(vert)
    primero = c_origen
    mayor = 0
    camino = None
    dijta = dfs.DepthFirstSearch(analyzer['mst'], primero)

    for v in lt.iterator(vert):
        if dfs.hasPathTo(dijta, v) == True:
            ruta = dfs.pathTo(dijta, v)
            x = lt.size(ruta)
            if x > mayor:
                mayor = x
                camino = ruta
    print(camino)
    distancia = 0
    previo = ""
    for item in lt.iterator(camino):
        if previo != "":
            print(previo,item)
            info = gr.getEdge(analyzer["connections_nd"], previo, item)
            a_sum = info['weight']
            previo = item
            distancia += float(a_sum)
        else:
            previo = item


    residuo = float(distancia) - float(distance_km)

    return num, peso, camino, distancia, residuo
```

Complejidad temporal final: O(ElogV).

Requerimiento 5 (Cuantificar el efecto de un aeropuerto cerrado):

- En el requerimiento 5 se busca “cuantificar el efecto de un aeropuerto cerrado”, para esto se buscaron los adyacentes del aeropuerto cerrado, por medio de la función adjacents(…), el cual retorna una lista con todos los vértices adyacentes del aeropuerto. Posteriormente se saca por medio de la función size(...) la cantidad de aeropuertos que son afectados. Finalmente, la complejidad de este requerimiento es de O(1).

```python
def removeA(analyzer, cIATA):
    lt_adjacents = gr.adjacents(analyzer['connections_d'], cIATA)
    size = lt.size(lt_adjacents)
    return lt_adjacents, size
```

Complejidad temporal final: O(1)