Documento de Análisis Reto 4

GRUPO 3:

- Ana Sofía Villa Benavides, 201923361, as.villa@uniandes.edu.co
- Daniela Alejandra Camacho Molano, 202110974, d.camachom@uniandes.edu.co

NRD= número de rutas en el dígrafo
NRG= número de rutas en el grafo
NAD= número de aeropuertos en el dígrafo
NAG= número de aeropuertos en el grafo
NC= número ciudades

Análisis de complejidad

Requerimiento 1

```
#Req 1
def interconexionAerea(analyzer):
    listaVerticesDirigido = gr.vertices(analyzer["digrafo conecciones"])
    minPqDirigido=mpq.newMinPQ(cmpGrado)
    for vertice in lt.iterator(listaVerticesDirigido):
        ingrado=gr.indegree(analyzer["digrafo conecciones"],vertice)
        outgrado= gr.outdegree(analyzer["digrafo conecciones"],vertice)
        gradoTotal= ingrado+outgrado
        if gradoTotal> 0:
            info=[vertice, gradoTotal,ingrado,outgrado]
            mpq.insert(minPqDirigido,info)
    listaVerticesNodirigido = gr.vertices(analyzer['grafo conecciones'])
    minPqNodirigido=mpq.newMinPQ(cmpGrado)
    for vertice in lt.iterator(listaVerticesNodirigido):
        grado=gr.degree(analyzer['grafo conecciones'],vertice)
        info=[vertice,grado]
        if grado!= 0:
            mpq.insert(minPqNodirigido,info)
    return (minPqDirigido,minPqNodirigido)
```

En este requerimiento se recorren todos los vértices del grafo dirigido ósea O(NAD), luego para cada vértice se halla su grado, esto tiene complejidad de O(1) ya que se trabaja una lista de adyacencia en la cual se saca el size de la lista de sus adyacentes. En este caso E es el número de adyacentes. Toda esta operación sería O(NAD)*O (1) = O(NAD)
Este proceso luego se repite para el grafo no dirigido teniendo una complejidad de O(NAG)
En total la función sería O(NAD)+ O(NAG)

Luego, en el view se desencola de ambas MinPQ para imprimir los resultados. Sabemos que esto esta implementado como un Heap entonces delMin sería de complejidad logNAD o logNAG, pero sabemos que O(n)>O(logn) entonces como es Big O , estas complejidades adicionales que se suman, se despreciarían.

Requerimiento 2

```
#Req 2#
def clusteresTraficoAereo(analyzer, IATA1,IATA2):
    if analyzer['components']==None:
        analyzer['components']=scc.KosarajuSCC(analyzer["digrafo conecciones"])
    #número de conectados
    conectados=scc.connectedComponents(analyzer['components'])
    #verifica si los dos aeropuertos estan en el mismo cluster
    iatasConectados=scc.stronglyConnected(analyzer['components'],IATA1,IATA2)
    return(conectados,iatasConectados)
```

En este requerimiento utilizamos en algoritmo de Kosaraju para hallar los SCC del grafo dirigido, este tiene una complejidad de O(NAD+NRD) donde NA es el número de vértices o aeropuertos y NRD es el número de rutas o arcos en este caso.
Luego se usa strongly conenected que es O (1) ya que lo que hace es acceder al componente de cada vértice dado y luego comparar para ver si es el mismo componente.

Requerimiento 3

```
def ciudadesHomonimas(analyzer,ciudad):
    pareja=m.get(analyzer['ciudades'],ciudad)
    listaCiudades=None
    if pareja != None:
        listaCiudades= me.getValue(pareja)
    return listaCiudades
def requerimiento3(analyzer,infoCiudadOrigen,infoCiudadDestino):
    origen= aeropuertoCercano(analyzer,infoCiudadOrigen)
    destino= aeropuertoCercano(analyzer,infoCiudadDestino)
    (disTerrestreOrigen,iataOrigen)=origen
    (disTerrestreDestino,iataDestino)=destino
    path=minimumCostPath(analyzer,iataOrigen,iataDestino)
    return (origen,destino,path)
```

Primero, para solucionar el problema de ciudades homónimas se toma el nombre de ciudad dado y se accede a su valor en la tabla de hash de ciudades, esto es O (1), luego este valor es una lista que se recorre para imprimirle al usuario todas las opciones, esto es O(CH) donde CH corresponde al número de ciudades homónimas que tenga la ciudad dada.
Luego se busca el aeropuerto cercano a la ciudad, para esto se va ampliando unos límites de coordenadas máximas y mínimas para buscar en recuadros que aumentan en 10km al cuadrado, esto se repite hasta encontrar un aeropuerto en el rango ósea O(D/10) donde D es la distancia del aeropuerto a la ciudad.
Con estos límites se busca los registros en esa zona geográfica utilizando un mapa ordenado de los aeropuertos usando como índice su longitud y latitud, esto tiene una complejidad de O(logNLAT)+O(logNLon), donde NLAT son las distintas latitudes posibles de los aeropuertos y NLon son las posibles longitudes distintas.
Por último, para hallar el camino de costo mínimo se utiliza el algoritmo dijsktra en el grafo dirigido, este tiene una complejidad de O(NRD*logNAD) donde NRD son los arcos o rutas y NAD son los aeropuertos o vértices.

En total el requerimiento sería O(CH) + O(D/10) + O(logNLAT)+O(logNLon)+ O(NRD*logNAD)

Requerimiento 4

En este requerimiento se utilizó el algoritmo de prim, el cual tiene una complejidad de O(NA^2). Luego, por medio de la mst se desencola todos los nodos que se encuentran en el camino mínimo y se agregan a una lista; esto tiene complejidad de O(NC), donde NC es el número de nodos que se encuentran en el camino mínimo. Luego, por medio de esta lista se accede a los elementos y por medio de esto se identifica cual es el camino más largo y se suman los pesos de cada arco que se encuentre, esto tiene complejidad de O(NC). Las siguientes operaciones que se realiza son O (1), donde compara las millas del viajero con el peso total y se determina si tiene excedente o faltante.
En total el requerimiento sería: O(NA^2) + O(NC)

```
def millasViajero(analyzer,ciudadOrigen,millas):
    grafo=analyzer["digrafo conecciones"]
    caminoMinimo=prim.PrimMST(grafo)
    minimo=caminoMinimo["mst"]
    listaNodos=lt.newList("ARRAY_LIST")
    while not q.isEmpty(minimo):
        edge=q.dequeue(minimo)
        lt.addLast(listaNodos,edge)
    search=dfs.DepthFirstSearch(analyzer["digrafo conecciones"],ciudadOrigen)
    info=None
    num=None
    costoTotal=0
    verticeInicial=None
    for ae in lt.iterator(listaNodos):
        if ae!=ciudadOrigen:
            path=djk.pathTo(search,ae)
            if num==None:
                num=lt.size(path)
                info=path
                verticeInicial=ae
            else:
                if lt.size(path)>num:
                    num=lt.size(path)
                    info=path
                    nuevo=ae
                    peso=gr.getEdge(analyzer["digrafo conecciones"],verticeInicial,nuevo)["weight"]
                    costoTotal+=peso
                    verticeInicial=nuevo
    costoTotalMi=costoTotal/1.60
    if costoTotalMi>millas:
        diferencia=costoTotalMi-millas
        cant="Faltante"
```

Requerimiento 5

Para realizar es te requerimiento primero se sacó la lista de los vértices y arcos que contiene cada grafo. La lista de vértices se utiliza para saber el número de aeropuertos en los grafos, esto sería O(NA) ya que debe extraer cada vértice y añadirlos a la lista correspondiente. Por otro lado, la lista de arcos, la cual al sacarla tendría complejidad de O(NRD) para el dígrafo y O(NRG) para el grafo; luego, se recorría cada lista correspondientemente y en ella se revisaban si en el arco revisado está presente el aeropuerto eliminado, si está presente se agregaba a una lista buscando la información por medio de la tabla de hash, esto sería O (1) debido a que la llave de la tabla es el iata de cada aeropuerto. Por ello, su complejidad es la siguiente:

-Para el dígrafo es: O(NA)+O(NRD)

-Para el grafo es: O(NA)+O(NRG)

```python
def aeropuertoCerradoDigr(analyzer,iata):
    #digrafo
    originalVerticesDigr=gr.vertices(analyzer["digrafo conecciones"])
    originalArcosDigr=gr.edges(analyzer["digrafo conecciones"])
    rutasAfectadas=0
    aeropuertosAfectados=lt.newList("ARRAY_LIST")
    for ruta in lt.iterator(originalArcosDigr):
        if ruta["vertexA"]==iata :
            rutasAfectadas=rutasAfectadas+1
            info=m.get(analyzer["aeropuertos"],ruta["vertexB"])["value"]
            lt.addLast(aeropuertosAfectados,info)
        elif ruta["vertexB"]==iata :
            rutasAfectadas=rutasAfectadas+1
            info=m.get(analyzer["aeropuertos"],ruta["vertexA"])["value"]
            lt.addLast(aeropuertosAfectados,info)
    return(originalVerticesDigr,originalArcosDigr,rutasAfectadas,aeropuertosAfectados)

def aeropuertoCerradogr(analyzer,iata):
    #grafo
    originalVerticesgr=gr.vertices(analyzer["grafo conecciones"])
    originalArcosgr=gr.edges(analyzer["grafo conecciones"])
    rutasAfectadasgr=0
    aeropuertosAfectadosgr=lt.newList("ARRAY_LIST")
    for ruta in lt.iterator(originalArcosgr):
        if ruta["vertexA"]==iata :
            rutasAfectadasgr=rutasAfectadasgr+1
            info=m.get(analyzer["aeropuertos"],ruta["vertexB"])["value"]
            lt.addLast(aeropuertosAfectadosgr,info)
        elif ruta["vertexB"]==iata :
            rutasAfectadasgr=rutasAfectadasgr+1
            info=m.get(analyzer["aeropuertos"],ruta["vertexA"])["value"]
            lt.addLast(aeropuertosAfectadosgr,info)
    return(originalVerticesgr,originalArcosgr,rutasAfectadasgr,aeropuertosAfectadosgr)
```