Análisis de complejidad

Nicolás Díaz Montaña – 202021006 – n.diaz9@uniandes.edu.co

Daniel R Alonso A - 201419873 – dr.alonso10@uniandes.edu.co

Resumen Ejecutivo:

Requerimiento	Complejidad
1	O(n log(n)) (n pequeño)
2	O(v + e)
3	O(n^2)
4	O((v+e) log v)
5	O(n) (n pequeño)

Requerimiento1:

La función principal de este requerimiento es la función Requerimiento1():

```
def Requerimiento1(analyzer):
    grafo = analyzer['red']
    aeropuertos = gr.vertices(grafo)
    conexiones = lt.newList()
    for aeropuerto in lt.iterator(aeropuertos):
        num = gr.degree(grafo,aeropuerto)
        if num > 200:
            lt.addLast(conexiones,num)
    ms.sort(conexiones,ordenAscendenteD)
    c = lt.lastElement(conexiones)
    mas_conectados = lt.newList()
    for aeropuerto in lt.iterator(aeropuertos):
        n = gr.degree(grafo,aeropuerto)
        if int(n) == int(c):
            lt.addLast(mas_conectados,aeropuerto)
    mapa = analyzer['aeropuertos']
    resultado = lt.newList('ARRAY_LIST')
    for ide in lt.iterator(mas_conectados):
        pareja = mp.get(mapa,ide)
        valor = me.getValue(pareja)
        lt.addLast(resultado,valor)
    lt.addLast(resultado,c)
    return resultado
```

Hay dos cosas por analizar: los ciclos y el mergesort. En el caso de los ciclos no se hace nada complejo dentro de ellos por lo que su complejidad es O(n). En el caso del mergesort su complejidad es O(n log(n)) pero este n es pequeño pues se trata de una lista recortada a los datos con más de 200 conexiones. Luego, en general, podemos decir que el requerimiento 1 tiene complejidad O(n log(n)) para n pequeño.

Requerimiento 2:

```
def Requerimiento2(analyzer, Aero_1, Aero_2):
    verificación, total = controller.Requerimiento2(analyzer, Aero_1, Aero_2)
    print("-" * 50)
    print("Total de clusteres presentes: " + str(total))
    print("-" * 50)
    if verificación is True:
        print("Los aeropuertos " + Aero_1 + " y " + Aero_2 + " SI estan Conectados.")
    else:
        print("Los aeropuertos " + Aero_1 + " y " + Aero_2 + " NO estan Conectados.")
```

Primero se le pide al usuario los IATAS de los aeropuertos de los cuales se le quiere saber si están en un clúster o no. Luego se llama a la función Requerimiento 2, la cual llama a clusterAirports() en el model que recibe el analyzer y los datos dados por el usuario.

```
def clusterAirports(analyzer, IATA1, IATA2):
    """
    Informa si dos aeropuertos estan fuertemente conectados.
    """
    analyzer["components"] = scc.KosarajuSCC(analyzer["red"])
    verificación = scc.stronglyConnected(analyzer["components"], IATA1, IATA2)
    total = scc.connectedComponents(analyzer["components"])
    return verificación, total
```

En esta función se encuentran 3 funciones importantes que tiene que ver con scc.py. La primera es KusarajuSCC() que se encarga de implementar el algoritmo de Kusaraju en el grafo designado y encontrar sus componentes fuertemente conectado, lo cual tiene una complejidad de O(v + e). La segunda es stronglyConnected() que se encarga de ver si dos vértices esta fuertemente conectados y tiene complejidad O(1). Finalmente, la tercera es connectedComponents() cuenta cuantos elementos fuertemente conectados tiene un grafo, lo cual también tiene complejidad de O(1).

En conclusión, el requerimiento 2 tiene una complejidad temporal O(e + v).

Requerimiento3:

La función principal de este requerimiento es la función Requerimiento3(). Esta función es larga y hace varias cosas. Consideramos que la parte más relevante (con respecto a la complejidad) es la siguiente:

```
# ver si existe ruta entre el aeropuerto de origen y el aeropuerto de destino mas cercanos a las ciudades
# De ser necesario, iterar cambiando los aeropuertos por aeropuertos menos cercanos hasta que tal ruta exista
k = 0
while k < 1:
    grafo = analyzer['red']
    IATA_o = aero_o['IATA']
    IATA_d = aero_d['IATA']
    search = djk.Dijkstra(grafo,IATA_o)
    existspath = djk.hasPathTo(search,IATA_d)
    if existspath:
        camino = djk.pathTo(search,IATA_d)
        latitud_o = float(aero_o['Latitude'])
        longitud_o = float(aero_o['Longitude'])
        latitud_d = float(aero_d['Latitude'])
        longitud_d = float(aero_d['Longitude'])
        dt_o = haversine(o_lng,o_lat,longitud_o,latitud_o)
        dt_d = haversine(d_lng,d_lat,longitud_d,latitud_d)
        k += 1
    else:
        o1 = lt.removeFirst(distancias_o)
        o2 = lt.firstElement(distancias_o)
        d1 =lt.removeFirst(distancias_d)
        d2 = lt.firstElement(distancias_d)
        if o2 <= d2:
            for a_o in lt.iterator(aeropuertos_origen):
                latitud_o = float(a_o['Latitude'])
                longitud_o = float(a_o['Longitude'])
                d_o = haversine(o_lng,o_lat,longitud_o,latitud_o)
                if d_o == o2:
                    aero_o = a_o
                    break
        if d2 < o2:
            for a_d in lt.iterator(aeropuertos_destino):
                latitud_d = float(a_d['Latitude'])
                longitud_d = float(a_d['Longitude'])
                d_d = haversine(d_lng,d_lat,longitud_d,latitud_d)
                if d_d == d2:
                    aero_d = a_d
                    break
```

En esta parte se mira si hay camino entre los aeropuertos más cercanos a las ciudades y se halla el camino más corto. Si no existe tal camino se cambian los aeropuertos a examinar por aeropuertos cada vez más lejanos (dentro de los aeropuertos cercanos) hasta que exista tal camino. La iteración entonces, en el peor caso, depende del tamaño de la lista de aeropuertos cercanos, la cual no es tan grande. Aún así, se ejecuta el algoritmo Dijkstra cada vez, el cual tiene complejidad O(m + k log(k)). En los demás ciclos no se hace nada realmente complejo. La complejidad es entonces O(nm + nk log(k)), para n pequeño (tamaño listas aeropuertos cercanos), m, k grandes (m número de edges, k número de vértices, en el grafo dirigido con todos los aeropuertos y rutas). Podemos decir, por tanto, que esta parte tiene complejidad aproximada O(n^2).

Así, en general, el requerimiento 3 tiene complejidad O(n^2).

<u>Requerimiento 4:</u>

```
def lifeMiles(analyzer, origen):
    blue = analyzer["blue"]
```

```
ciudades = analyzer['ciudades']
cities_origin = mp.get(ciudades, origen)["value"]
if lt.size(cities_origin) > 1:
    ID_base = ciudades_homonimas(cities_origin)
    ciudad_base = mp.get(analyzer["ciudades_id"], ID_base)["value"]
else:
    ID_base = lt.firstElement(cities_origin)["id"]
    ciudad_base = mp.get(analyzer["ciudades_id"], ID_base)["value"]

aero_base = aeropuertos_cercanos(analyzer, ciudad_base)["IATA"]

search = prim.PrimMST(blue) #O((v+e) log v)
prim.edgesMST(blue, search)
path = search['mst']
nodos = lt.newList()
large = 0
final = None
peso = prim.weightMST(blue, search)

while not q.isEmpty(path):
    edge = q.dequeue(path)
    lt.addLast(nodos, edge["vertexB"])

cantidad = lt.size(nodos)

h = dfs.DepthFirstSearch(blue, aero_base)# O(v)
for nodo in lt.iterator(nodos):
    if dfs.hasPathTo(h, nodo):
        p = dfs.pathTo(h, nodo)
        if st.size(p) > large:
            large = st.size(p)
            final = p
return final, peso, cantidad, aero_base
```

Lo que tiene más complejidad en este requerimiento es la función lifeMIles(), que se separa entres partes. La primera parte es conseguir el aeropuerto más cercano que como ya se menciono antes tiene una complejidad dependiendo de cuantos aeropuertos tiene ciudades homónimas. La segunda es crear el MST usando prim, la cual tiene una complejidad temporal de O((v+e) log v). Por ultimo se tiene que realizar un dfs a partir del aeropuerto de origen, lo cual resulta siendo o O(v). Pero luego se raliza un ciclo while para poder conseguir la rama más larga usando el grafo creado del dfs y los nodos obtenidos del MST, el cual puede llegar a ser O(n). Aún asi, este requerimiento quedaría con algoritmo de complejidad de O((v+e) log v).

Requerimiento5:

La función principal de este requerimiento es la función Requerimiento5():

```
def Requerimiento5(analyzer,aeropuerto):
    grafo = analyzer['red']
    iatas = gr.adjacents(grafo,aeropuerto)
    aeropuertos = analyzer['aeropuertos']
    resultado = lt.newList()
    lt.addLast(resultado,lt.size(iatas))
    lista = lt.newList()
    for iata in lt.iterator(iatas):
        pareja = mp.get(aeropuertos,iata)
        aeropuerto = me.getValue(pareja)
        lt.addLast(lista,aeropuerto)
    lt.addLast(resultado,lista)
    return resultado
```

Lo más complejo que se hace es un ciclo con complejidad O(n), donde n es el tamaño de la lista de adyacentes a un vértice en el grafo. Dentro del ciclo no se hace nada complejo. Así, el requerimiento 5 tiene complejidad O(n) para n pequeño.