Análisis de Complejidad

Sergio David López Becerra 202110260

Sergio Montoya 202112171

1.

```
def more_edges(grafo):
    lstvert = gp.vertices(grafo)
    airports = lt.size(lstvert)
    top_5 = m.newMap()
    n = 0
    while n < 5:
        maxvert = None
        maxdeg = 0
        for vert in lt.iterator(lstvert):
            degree = gp.outdegree(grafo,vert) * gp.indegree(grafo,vert)
            if(degree > maxdeg) and not m.contains(top_5,vert):
                maxvert = vert
                maxdeg = degree
        m.put(top_5,maxvert,maxdeg)
        n+=1
    return top_5, airports
```

Nota: Este requerimiento tiene el doble de complejidad que se enuncia aquí dado que se hace para 2 grafos. Sin embargo, no aparece en la foto pues esta función se la relegamos a el controler.

Primero conseguimos la lista de vértices del grafo que nos pasaron O(1)

Luego le sacamos el número de vértices (esto para responder la cantidad de aeropuertos interconectados (por que pertenecen al grafo) O(1)

Luego creamos un nuevo mapa que se llame top_5 O (1)

Ahora recorremos 5 veces la lista de vértices buscando justamente los 5 mayores.

Esto quiere decir que de ahora en adelante la complejidad se multiplica por 5

Hacemos un for que por ende hará que todo cuando menos tenga una complejidad de O(5V) dado que se repite n veces y esto se hace 5 veces.

Luego calculamos el número de conexiones indegree y outdegree y las multiplicamos. Esto lo hacemos pues el enunciado era poco claro al respecto de lo que buscábamos (cosa que se repitió bastante a lo largo de todo el reto) y por ende consideramos que al referirse con interconectados hacía referencia a la cantidad de rutas por las que pasaba en el medio y no como fin de camino o inicio de camino. Y por ende la cantidad de posibilidades era de indegree x outdegreee.

Ahora comprobamos cual es mayor si el que ya teníamos o este O(5V)

Y si resulta que es mayor re definimos O < O(5V)

Por último, agregamos lo que resulto ser el mayor a top 5 y repetimos O(5)

Por ende, la complejidad de este Requerimiento es O(5n)

2.

```
def clusters(catalogo,airport1,airport2):
    SCC = scc.KosarajuSCC(catalogo["Vector"])
    print(SCC)
    numero_SCC = SCC["components"]
    Existe = "NO"
    if m.contains(SCC["idscc"],airport1) and m.contains(SCC["idscc"],airport2):
        airport_1 = me.getValue(m.get(SCC["idscc"],airport1))
        airport_2 = me.getValue(m.get(SCC["idscc"],airport2))
        if airport_1 == airport_2:
            Existe = "SI"
    return numero_SCC, Existe
```

Primero sacamos un algoritmo de kosaraju O(3(V+E))

Luego consultamos cuantos componentes tiene O(1)

Y por último, revisamos en idscc si ambos tienen el mismo resultado. Esto para poder identificar si pertenecen al mismo SCC O(v)

3.

```
def near_route(catalogo,ciudad1,ciudad2):
    ciudad_1 = me.getValue(m.get(catalogo["cities"],ciudad1))
    ciudad_2 = me.getValue(m.get(catalogo["cities"],ciudad2))
    if lt.size(ciudad_1["aeropuertos"]) != 0:
        aeropuerto_min_ciudad1 = None
        distancia_min_ciudad1 = 1*10**100
        for aeropuerto in lt.iterator(ciudad_1["aeropuertos"]):
            airport = me.getValue(m.get(catalogo["airports"],aeropuerto))
            ciudad = (float(me.getValue(m.get(catalogo["cities"],ciudad1))["lat"]),float(me.getValue(m.get(c
            airport_lt = (float(airport["Latitude"]),float(airport["Longitude"]))
            distancia = hv(ciudad,airport_lt)
            if distancia < distancia_min_ciudad1:
                distancia_min_ciudad1 = distancia
                aeropuerto_min_ciudad1 = aeropuerto
    else:
        print("Pa que quieres hacer eso?1")
        aeropuerto_min_ciudad1 = None
        distancia_min_ciudad1 = 1*10**100
        for aeropuerto in lt.iterator(m.keySet(catalogo["airports"])):
            airport = me.getValue(m.get(catalogo["airports"],aeropuerto))
            ciudad = (float(me.getValue(m.get(catalogo["cities"],ciudad1))["lat"]),float(me.getValue(m.get(c
            airport_lt = (float(airport["Latitude"]),float(airport["Longitude"]))
            distancia = hv(ciudad,airport_lt)
            if distancia < distancia_min_ciudad1:
                distancia_min_ciudad1 = distancia
                aeropuerto_min_ciudad1 = aeropuerto
```

Nota: En este requerimiento la primera parte era distinguir los nombres, esto lo vamos a ignorar pues no resulta ser realmente útil. Y la segunda parte en donde encontramos el aeropuerto se repite dos veces, pero dado que es lo mismo analizaremos solamente esta primera parte.

Primero sacamos del catálogo de ciudades las dos que nos interesan. Ahora, nosotros en la carga de datos creamos una sublista que se llama aeropuertos (aprovechándonos de que los aeropuertos tenían un indicador en donde decían a qué ciudad pertenecen) si resulta que esta lista tiene al menos un aeropuerto entonces recorremos esta lista y empaquetamos en una tupla latitud y longitud de los aeropuertos y latitud y longitud de la ciudad para pasársela a una función de la librería harvesine la cual (como nos indicaron) calcula la distancia entre la ciudad y el aeropuerto. Nos quedamos en el menor posible cuando lo comparamos.

En caso de que esta lista este vacía, lo que hacemos es básicamente lo mismo, pero considerando absolutamente todos los aeropuertos lo cual incrementa la complejidad bastante.

```
                    aeropuerto_min_ciudad2 = aeropuerto
    rutas = djk.Dijkstra(catalogo["Vector"],aeropuerto_min_ciudad1)
    camino_min = djk.pathTo(rutas,aeropuerto_min_ciudad2)
```

Por último, (y más complejo) simplemente hacemos un algoritmo de Dijkstra O(E*2*Log(v))

Y utilizamos una de sus propias funciones pathTo para hallar el camino que nos sirve dentro del propio grafo que nos devolvió Dijkstra

4.

```
def more_cities(catalogo,mill):

    prim = pm.PrimMST(catalogo["AntiVector"])
    mst = pm.edgesMST(catalogo["AntiVector"],prim)
    mst = mst["mst"]
    actual_mst = lt.newList(datastructure="ARRAY_LIST")
    weight = 0
    for current in lt.iterator(mst):
        lt.addLast(actual_mst,current)
        weight += current["weight"]
    max_weight = 0
    max_elements = 0
    for element in lt.iterator(mst):
        way, km = serch_continue(mst,element)
        if lt.size(way) > max_weight:
            max_weight = lt.size(way)
            max_elements = way

    print(max_elements)
    millas = (km*1.6)/2-mill
    return actual_mst, km, millas, max_elements

def serch_continue(mst,element):
    way = lt.newList()
    lt.addLast(way,element["vertexA"])
    searching = element["vertexB"]
    km = element["weight"]
    for element2 in lt.iterator(mst):
        if searching == element2["vertexA"] and lt.isPresent(way,searching) == 0:
            lt.addLast(way,searching)
            searching = element2["vertexB"]
            km += element2["weight"]
    lt.addLast(way,searching)
    return way, km
```

Ahora, lo primero que hacemos es buscar un mst con el algoritmo de prim O(E*2*Log(v)). En este nos ocurría una cosa rarísima y es que prim[“mst”] Estaba vacío, por ende, buscamos y vimos que, si le pasábamos edgeMST este se llenaba, luego pasábamos esa lista a una nueva variable para que no nos ocupara el espacio de todo el catálogo que devuelve prim y de ahí buscamos en mst la vía más larga. La manera en la que hacemos esto es buscando en vértices A los vértices B para saber que hay una continuidad. Es decir, imaginemos que estamos en la vía LED – DXB pues ahora busquemos una vía en DXB que nos lleve a otro lugar y así continuamos hasta que lleguemos a una que no nos lleve a ninguna parte. Ahí nos detenemos (en este caso la complejidad seria de O(E**2)

Por ende, su dificultad es O(E**2) dado que esta es mayor incluso a la del algoritmo de prim.

5.

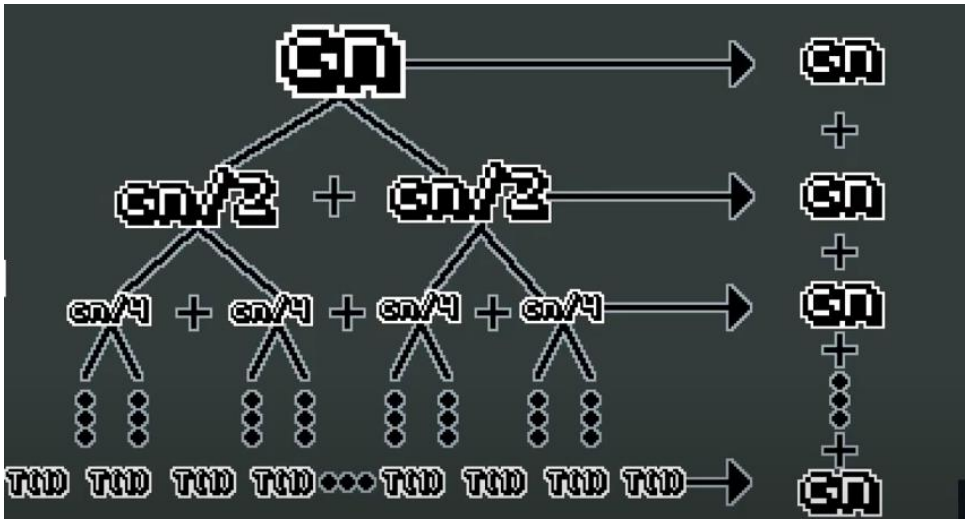
```python
def airport_closed(catalogo,iata):
    grafo = catalogo["Vector"]
    adyacentes = gp.adjacents(grafo,iata)
    salidas = lt.newList()
    for vertex in lt.iterator(adyacentes):
        if gp.getEdge(grafo, vertex, iata)!=None:
            lt.addLast(salidas,vertex)
    first_vertex = lt.getElement(salidas,1)
    salidas_finales = recursive_airport(grafo,salidas,first_vertex)

    return salidas_finales

def recursive_airport(grafo,salidas,iata):
    adyacentes = gp.adjacents(grafo,iata)
    segunda = lt.newList()
    for vertex in lt.iterator(adyacentes):
        if gp.getEdge(grafo, vertex, iata)!=None:
            lt.addLast(segunda,vertex)
    if segunda != None:
        for aereo_2 in lt.iterator(segunda):
            if lt.isPresent(salidas,aereo_2) == 0:
                lt.addLast(salidas,aereo_2)
                recursive_airport(grafo,salidas,aereo_2)
    return salidas
```

Para este requerimiento se hace uso de dos funciones. En la primera únicamente se sacan los vértices adyacentes de un aeropuerto seleccionado por el usuario y se busca cuáles de ellos son de entrada. Para esto hacemos un for, con lo que la primera función tiene una complejidad de O(E).

Para la segunda función se revisa comienza con el mismo procedimiento que la primera a excepción de que ahora revisaremos las entradas iniciales y las entradas de las entradas, esto haciendo un uso de dos for. Notemos que la complejidad inicial se dividirá en dos partes importantes, y estos a su vez se dividirán de igual forma. Con lo que podemos hacer un diagrama de la forma:



Notemos entonces que recorrer todas las recursividades de la función es equivalente a un recorrido de un árbol balanceado, con lo que podemos definir en últimas la complejidad temporal igual a O(logE).

REFERENCIAS:

La imagen usada para explicar la complejidad del Req-5 fue tomada del vídeo, [Big O para algoritmos Recursivos | Análisis de Algoritmos]

.