**RETO 4**

**Estructuras de Datos y Algoritmos**

***Estudiante A:*** *Daniel Andrés Bernal Cáceres 202020706*

***Estudiante B:*** *Juan Martín Santos Ayala 202013610*

***Correos:*** *da.bernalc1@uniandes.edu.co*

*j.santosa* *@uniandes.edu.co*

1. **Preparación del ambiente de trabajo:**

Para la preparación del ambiente de trabajo, creamos los archivos correspondientes al modelo “MVC” para independizar los cambios hechos en código y que no se vieran reflejados en el producto entregado al usuario.

* 1. **Carga de archivos**

Para cargar los archivos primero creamos un analizador para guardar la información de un archivo de forma organizada. Tal organización fue:

def newAnalyzer():

analyzer = {'landingPoint': None,

                'connections': None,

                'country': None,

                'vertex': None}

    analyzer['landingPoint'] = m.newMap(numelements=14000,

                                       maptype='PROBING',

                                      comparefunction=compareStopIds)

    analyzer['country'] = m.newMap(numelements=14000,

                                   maptype='PROBING',

                                   comparefunction=compareStopIds)

    analyzer['connections'] = gr.newGraph(datastructure='ADJ\_LIST',

                                          directed=True,

                                          size=14000,

                                      comparefunction=compareStopIds)

        return analyzer

Creamos una lista vacía con todos los vértices cargados para facilitar la consulta del tamaño total de datos cargados del archivo CSV.

    analyzer['vertex'] = lt.newList('ARRAY\_LIST')

Después de crear el catálogo lo que hacemos es agregar la información de los landing points y los arcos al grafo. Para esto, creamos una función que controla la adición de landing points, arcos y agrega los vértices a una lista.

def addLandingConnections(analyzer, service):

    origin = service['\ufefforigin']

    destination = service['destination']

    distance = service['cable\_length']

    addPoint(analyzer, origin)

    addPoint(analyzer, destination)

    op1 = origin not in analyzer['vertex']

    op2 = destination not in analyzer['vertex']

    if op1 or op2:

        lt.addLast(analyzer['vertex'], origin)

        lt.addLast(analyzer['vertex'], destination)

    addConnections(analyzer, origin, destination, distance)

    return analyzer

Agregamos los vértices al grafo.

def addPoint(analyzer, stopid):

    if not gr.containsVertex(analyzer['connections'], stopid):

        gr.insertVertex(analyzer['connections'], stopid)

    return analyzer

Agregamos los arcos al grafo.

def addConnections(analyzer, origin, destination, distance):

gr.addEdge(analyzer['connections'], origin, destination, distance)

   return analyzer

Después de crear el catálogo creamos un diccionario para guardar la información del archivo landing\_points.csv.

def landingPoints(analyzer, lp):

    country = m.newMap(numelements=5, maptype='PROBING')

    latitud = lp['latitude']

    longitud = lp['longitude']

    coordenates = (latitud, longitud)

    m.put(country, 'Name', lp['name'])

    m.put(country, 'Id', lp['id'])

    m.put(country, 'Coordenates', coordenates)

    m.put(analyzer['landingPoint'], lp['landing\_point\_id'], country)

    return analyzer

Después de crear el catálogo creamos un diccionario para guardar la información del archivo countries.csv.

def countries(analyzer, country):

    capital = m.newMap(numelements=13, maptype='PROBING')

    latitud = country['CapitalLatitude']

    longitud = country['CapitalLongitude']

    coordenates = (latitud, longitud)

    m.put(capital, 'CapitalName', country['CapitalName'])

    m.put(capital, 'Coordenates', coordenates)

    m.put(capital, 'CountryCode', country['CountryCode'])

    m.put(capital, 'ContinentName', country['ContinentName'])

    m.put(capital, 'Population', country['Population'])

    m.put(capital, 'Internet users', country['Internet users'])

    m.put(analyzer['country'], country['CountryName'], capital)

    return analyzer

* 1. **Mostrar la información al usuario**

Por último, antes de comenzar a realizar los requerimientos, imprimimos la información al usuario. Acá fue donde nos dimos cuenta de que era mejor tener todos los vértices en una lista en específico. Esto nos serviría para conocer el size del archivo a leer y en primera medida todos los datos del csv.

**La complejidad de nuestro catálogo es de O(n logn).**

1. **Requerimiento 1**

El requerimiento 1 nos exigía encontrar la cantidad de clústeres (componentes conectados) dentro de la red de cables submarinos y si dos landing points pertenecen o no al mismo clúster.

def clusterSearch(analyzer, lp1, lp2):

    scc = sc.KosarajuSCC(analyzer)

    question = sc.stronglyConnected(scc, lp1, lp2)

    return scc["components"], question

Al final de esta función obtenemos la cantidad de eventos que hay y la cantidad de artistas únicos, y en el view sólo imprimimos los que nos pide el usuario.

**La complejidad del requerimiento 1 es de O( 2n log(n) ).**

**Requerimiento 2**

El requerimiento 2 nos exigía el (los) landing point(s) que sirven como punto de interconexión a más cables en la red.

def connectionSearch(analyzer):

    vertexlist = lt.newList('SINGLE\_LINKED')

    vertetex = gr.vertices(analyzer['connections'])

    iterator = ite.newIterator(vertetex)

    while ite.hasNext(iterator):

        pos = ite.next(iterator)

        number = gr.degree(analyzer['connections'], pos)

        lt.addLast(vertexlist, (pos, number))

    finallist = sortreq2(vertexlist)

    return finallist

Al final de esta función obtenemos una lista

**La complejidad de nuestro requerimiento 2 es de O( n ( 3log(n)+ 2 ) ).**

1. **Requerimiento 3**

El requerimiento 3 nos exigía las pistas en el sistema de recomendación que podrían ayudar a los usuarios en su periodo de estudio.

def studyMusic(maps, keylo1, keyhi1, keylo2, keyhi2):

    caract1 = 'instrumentalness'

    caract2 = 'tempo'

    easylist = studyMap(maps, keylo1, keyhi1, keylo2, keyhi2, caract1, caract2)

    if easylist is None:

        return 0

    else:

        tracksMap = mp.newMap(34500,

                              maptype='PROBING',

                              loadfactor=0.5)

        iterator = ite.newIterator(easylist)

        while ite.hasNext(iterator):

            events = ite.next(iterator)

            mp.put(tracksMap, events['track\_id'], events)

        size = lt.size(tracksMap)

        if size > 5:

            tracklist = 0

        else:

            lista = mp.valueSet(tracksMap)

            tracklist = get5artists(tracksMap, lista, size)

        return size, tracklist

Al final de esta función obtenemos la cantidad de eventos de escucha que cumple la característica dada.

**La complejidad de nuestro requerimiento 3 es de O( n ( 3log(n) + 2 ) ).**

1. **Requerimiento 4**

El requerimiento 4 nos exigía saber cuantas cancones se tienen por cada género y cuantos artistas de tienen en cada género.

Función que me retorna 10 valores al azar de una lista de listas pasada por parámetro.

def get10artists(artistmap, lista, size):

    finallist = lt.newList("ARRAY\_LIST")

    eventlist = random.sample(range(size), 10)

    for i in eventlist:

        keya = lt.getElement(lista, i)

        lt.addLast(finallist, keya)

    return finallist

Al final de esta función obtenemos 10 llaves al azar y la cantidad de reproducciones totales de cada género.

**La complejidad de nuestro requerimiento 4 es de O( 9n (log(n) + n + 1 ) )**

1. **Requerimiento 5**

El requerimiento 5 nos exigía indicar el género de música más escuchado en dicho rango teniendo en cuenta todos los días disponibles e informar el promedio para cada uno de los valores.

Función que busca los eventos en un rango de tiempo.

Retorna la cantidad de eventos de escucha totales y un mapa con los eventos

def musicInTime(maps, time1, time2):

    init = seconds(time1)

    final = seconds(time2)

    if init < 0 or final < 0:

        return 0

    else:

*# thelist = lt.newList("ARRAY\_LIST")*

        themap = om.newMap(omaptype='RBT')

        eventsq = 0

*# Sacamos lo eventos en el rango deseado por el usuario*

        getusers = om.values(maps['time'], init, final)

*# Recorremos la lista de las listas de los eventos*

        iterator = ite.newIterator(getusers)

        while ite.hasNext(iterator):

            eventslist = ite.next(iterator)

*# Para encontrar el numero total de eventos en el rango*

            newiterator = ite.newIterator(eventslist['lstevent'])

            while ite.hasNext(newiterator):

                event = ite.next(newiterator)

                eventsq += 1

                entry = om.get(themap, float(event['tempo']))

                if entry is None:

                    dataentry = newDataEntry(event)

                    om.put(themap, float(event['tempo']), dataentry)

                else:

                    dataentry = me.getValue(entry)

                    addEntry(dataentry, event)

        return eventsq, themap

Al final de esta función obtenemos una lista con los valores de track\_id, cantidad de Hashtags y Vader promedio.

La complejidad de nuestro requerimiento 5 es de O( n ( 5log(n) + 2n + 10 ) ).

* Comparar la complejidad de los requerimientos implementados en el Reto No. 1 con los implementados en este reto.

**Requerimiento 1:**

En nuestro reto 1 la complejidad era de O(n logn) y en el reto 2 es de O(n logn).

**Requerimiento 2:**

En nuestro reto 1 la complejidad era de O(2n^2) y en el reto 2 es de O(n).

**Requerimiento 3:**

En nuestro reto 1 la complejidad era de O(2n^2) y en el reto 2 es de O(n).

**Requerimiento 4:**

En nuestro reto 1 la complejidad era de O(2n) y en el reto 2 es de O(n).

Estas complejidades fueron calculadas teniendo en cuenta el peor de los casos.