Documento de Análisis de resultados para el Reto 4

Estructuras de datos y algoritmos – 202210

Programa analizador del sistema de transporte Toronto Bikeshare

Integrantes del grupo

1. Jorge Andrés Uchima – 201914697 – ja.uchima@uniandes.edu.co
2. Faiber Alonso Hernández – 202012367 – f.hernandezt@uniandes.edu.co

1. Distribución del trabajo

Requerimientos 1,2,3 y 4: Realizados en grupo
Requerimiento 5: Faiber Alonso Hernández
Requerimiento 6: Jorge Andrés Uchima

2. Análisis de complejidad para cada requerimiento

En notación Big O.

Requerimiento 1:

(Presentado en la siguiente página)

```python
#Requerimiento 1

def topFiveStartStations(catalog):
    """
    Se encarga de realizar la búsqueda del top 5 de estaciones con más viajes de salida
    """
    topFiveList = lt.newList("ARRAY_LIST")
    finalTopFiveList = lt.newList("ARRAY_LIST")

    topFiveIDs = catalog["topFiveStationsKeys"]
    vertexInfoMap = catalog["vertexInfoMap"]

    #Devolver la información de ese top 5 de IDs a partir de buscar las IDs en la tabla
    for key in lt.iterator(topFiveIDs):
        vertexInfo = mp.get(vertexInfoMap, key)
        vertexInfoDict = me.getValue(vertexInfo)
        lt.addLast(topFiveList, vertexInfoDict)

    #Devolver una lista construida en la carga de datos, la cual contiene el ID, el día
    #top 5 estaciones de salida.
    catalogTopFive = catalog["topStationsDatesHours"]

    #------------------Fin de cálculos y procesamiento con las estructuras de datos-----

    #Construcción de la lista final que se devolverá
    counter = 1

    for vertex in lt.iterator(topFiveList):
        stationID = vertex["stationID"]
        stationName = vertex["stationName"]
        amountDepartureTrips = lt.size(vertex["connectsToList"])
        AnnualTypeTrips = vertex["Annual members"]
        CasualTypeTrips = vertex["Casual members"]

        secondDict = lt.getElement(catalogTopFive, counter)
        topDate = secondDict["Top Date"]
        topHour = secondDict["Top Hour"]

        topHour = str(topHour)+":00 a "+str(topHour+1)+":00"

        itemDict = {"ID de la estación": stationID,
                    "Nombre de la estación":stationName,
                    "Cantidad de viajes que salieron":amountDepartureTrips,
                    "Viajes hechos por usuarios Anuales": AnnualTypeTrips,
                    "Viajes hechos por usuarios Casuales": CasualTypeTrips,
                    "Fecha en la que salieron más viajes": topDate,
                    "Hora a la que salieron más viajes": topHour}

        lt.addLast(finalTopFiveList, itemDict)
        counter+=1

    return finalTopFiveList
```

Análisis de complejidad:

Una parte muy importante del cálculo para determinar el top 5 de estaciones con más viajes de salida se realizó durante la carga de datos, pues, este requerimiento no pide parámetros específicos que limite la respuesta.

Al ejecutar el requerimiento, se inicia un ciclo que se repite solo 5 veces y cada vez se busca un elemento en una tabla de hash para devolver la información de un vértice, esto es **O(1.5) – O(2.5)** pues el mecanismo de solución de colisiones usado es Linear Probing.

La información de cada vértice del top 5 es guardada en una lista. Luego, se pide la información de la fecha y la hora en las cuales más viajes iniciaron para cada vértice del top, esta información se pide a una lista que también había sido guardada en la carga de datos, esta operación también tiene una complejidad **O(1.5) – O(2.5).**

Conclusión: Este requerimiento pudo realizarse con una complejidad temporal en el peor caso de **O(K),** donde K es un número constante, influenciado principalmente por una serie de operaciones “get” a 2 tablas de hash ya construidas.

Requerimiento 2

```
def searchPaths(catalog, startName, duration, minS, maxS):

    nameMap = catalog['stationNameMap']
    graph = catalog['graph']
    couple = mp.get(nameMap, startName)
    startId = me.getValue(couple)

    lstAdj = gr.adjacents(graph, startId)
    lstCaminos=lt.newList('ARRAY_LIST')
    contRutas =0

    for adj in lt.iterator(lstAdj):
        dura = gr.getEdge(graph,startId,adj)['weight']

        if dura<duration:
            lstRuta=lt.newList('ARRAY_LIST')
            lt.addLast(lstRuta,adj)
            lstNew = continuePath(graph,adj,dura,lstRuta,duration,minS,maxS)

            if contRutas<maxS and lstNew != 'nada':
                lt.addLast(lstCaminos,lstNew)
                contRutas+=1
            elif contRutas== maxS:
                lstPeso, lstParadas =tiempoyNumParadas(graph, lstCaminos)
                return lstCaminos,lstPeso, lstParadas

            if adj == lt.lastElement(lstAdj):
                lstPeso, lstParadas = tiempoyNumParadas(graph, lstCaminos)
                return lstCaminos, lstPeso, lstParadas
```

```
def continuePath(graph,adjV,peso,lstRuta,duration,minS,maxS):
    lstAdj = gr.adjacents(graph,adjV)

    for adj in lt.iterator(lstAdj):
        dura = gr.getEdge(graph,adjV,adj)['weight']
        peso+=dura
        if peso<duration:
            lt.addLast(lstRuta,adj)
            continuePath(graph,adj,peso,lstRuta,duration,minS,maxS)

        if peso<duration and lt.size(lstRuta)>=minS:
            return lstRuta

        if peso>duration and lt.size(lstRuta)<minS:
            return 'nada'
```

```python
def tiempoyNumParadas(graph, lstCaminos):

    lstPeso = lt.newList('ARRAY_LIST')
    lstParadas = lt.newList('ARRAY_LIST')

    for camino in lt.iterator(lstCaminos):
        peso = 0
        for cont in range(1,lt.size(camino)+1,1):

            if cont<=lt.size(camino)-1:
                ini = lt.getElement(camino,cont)
                fin = lt.getElement(camino,cont+1)
                weight = gr.getEdge(graph, ini,fin)
                if weight != None:
                    weight = weight['weight']
                    peso+=weight

        lt.addLast(lstPeso,peso)
        lt.addLast(lstParadas,lt.size(camino))

    return lstPeso, lstParadas
```

Análisis de complejidad:

Para este requerimiento se realizó una búsqueda de caminos a partir de una estación de inicio que el usuario ingresa. Para realizar esta búsqueda se implementó una especie de algoritmo DFS en donde se pide la lista de adyacencia y se va en profundidad por un camino solo si este no sobrepasa los limites de tiempo y estaciones que ingresa el usuario. Como resultado se podría decir que tiene la complejidad de un algoritmo DFS O(V+E) aunque en realidad en promedio su complejidad será menor ya que en general los caminos que necesita encontrar son cortos comparados con recorrer todo el grafo y adicionalmente cuando ya posee todos los caminos que necesita el código para y deja de buscar nuevos caminos. Adicionalmente al final se realiza un recorrido por toda la lista de caminos para poder mostrar al usuario el tiempo total que demora cada camino por lo que esta operación tiene una complejidad O(A*B) en donde A es el tamaño de la lista que guarda los caminos y B es el tamaño de la lista con cada vértice de un solo camino. Cabe resaltar que los tamaños A y B son mínimos a comparación con el número de vertices y arcos V y E.

Conclusión: Este requerimiento pudo realizarse con una complejidad temporal en el peor caso de **O(V+E),** ya que el peor caso sería encontrar un camino en donde se tuvieran que recorrer todos los vértices para cumplir los requerimientos del usuario.

Requerimiento 3

```python
#Requerimiento 3: Componentes fuertemente conectados
def getSCCfromGraph(catalog):
    """
    Calcula los componentes conectados del grafo.
    Se utiliza el algoritmo de Kosaraju.
    """
    #En la carga de datos, ya se llamó a la función de Kosaraju para identificar estos componentes fuertemente cone
    #Además, se clasificaron en una tabla de hash.
    componentsMap = catalog["componentsMap"]
    vertexInfoMap = catalog["vertexInfoMap"]

    sccNumber = catalog["numberOfSCC"]

    #Ahora, queda iterar sobre cada lista de ids de cada componente fuertemente conectado y devolver la información
    #donde más viajes inician y terminan, llamando a una función que busque ese id de estación en el mapa de inform
    #del catálogo.

    #Diccionario de respuesta
    infoDict = {"Total de componente fuertemente conectados":sccNumber,
                "Información por componente":lt.newList("ARRAY_LIST")}

    #Llaves del mapa
    componentNumbers = mp.keySet(componentsMap)

    #Devolución de las listas de ids de cada componente
    for comp in lt.iterator(componentNumbers):
        kv = mp.get(componentsMap, comp)
        k = comp
        idList = me.getValue(kv)
        #Contadores de viajes para hallar las estaciones más visitadas
        maxDeparture = 0
        maxArrive = 0
        topDeparture = None
        topArrive = None
        topDepartureID = None
        topArriveID = None
```

```
    #Iteración sobre esa lista para hallar las estaciones donde más viajes inician y terminan
    for id in lt.iterator(idList):
        findVertex = mp.get(vertexInfoMap, id)
        vertexInfo = me.getValue(findVertex)
        departures = lt.size(vertexInfo["connectsToList"])
        arrivals = lt.size(vertexInfo["connectedToList"])

        if departures > maxDeparture:
            maxDeparture = departures
            topDepartureID = id
            topDeparture = vertexInfo["stationName"]

        if arrivals > maxArrive:
            maxArrive = arrivals
            topArriveID = id
            topArrive = vertexInfo["stationName"]

    compDict = {"Número de componente":k,
                "Estación de la que más salen viajes":(topDepartureID, topDeparture),
                "Estación a la que más llegan viajes":(topArriveID, topArrive),
                "Número de viajes de salida":departures,
                "Número de viajes que llegan":arrivals}

    lt.addLast(infoDict["Información por componente"], compDict)

return infoDict
```

Análisis de complejidad: Al igual que el requerimiento 1, en la carga de datos se adelantó gran parte del cálculo.

La ejecución y preprocesamiento del algoritmo de Kosaraju para encontrar los SCC del grafo construido fueron realizados completamente en la carga de datos, en ese caso, la complejidad es de **O(V+E)** en el peor caso (complejidad del algoritmo de Kosaraju).

Lo primero que se realiza en la ejecución del requerimiento, es obtener el número de componentes fuertemente conectados hallados con el algoritmo de Kosaraju, esto tiene una complejidad O(1) pues es solo obtener el valor de una variable del catálogo.

Posterior a esto, se obtiene el mapa resultado del algoritmo de Kosaraju para dividir los distintos componentes y hallar la información solicitada de cada uno. Para cada componente, se obtiene la información del número de viajes de salida y que llegan para compararlos entre ellos y obtener los mayores. La complejidad de obtener la información de los vértices de los componentes fuertemente conectados es de **O(1.5) - O(2.5)** pues estos se encuentran en una tabla de hash (mecanismo Linear Probing).

Conclusión: A la hora de ejecutar el requerimiento, las operaciones de mayor complejidad son: el KeySet() de un mapa y el get() sobre mapas también, los cuales tienen complejidades constantes. Sin embargo, se realiza la operación get() de forma cíclica, dependiendo de la cantidad de elementos que hay en el KeySet()

realizado sobre un mapa. Por tanto, en el peor caso, se debe repetir el ciclo N veces, donde N es el número de elementos fuertemente conectados del grafo.

Complejidad final: O(N), donde **N** es la cantidad de componentes fuertemente conectados hallados por el algoritmo de Kosaraju.

Requerimiento 4

```
def searchMinCostPath(catalog,nameO, nameD):

    nameMap = catalog['stationNameMap']
    graph = catalog['graph']
    couple1 = mp.get(nameMap, nameO)
    couple2 = mp.get(nameMap, nameD)
    startId = me.getValue(couple1)
    endId = me.getValue(couple2)

    result_djk = djk.Dijkstra(graph, startId)
    path = djk.pathTo(result_djk, endId)

    seAcabo = False
    pesoTot = 0
    lstProm = lt.newList('ARRAY_LIST')
    lstNom = lt.newList('ARRAY_LIST')
    idNom = startId
    lt.addLast(lstNom,idNom)
    while seAcabo == False:
                if stack.size(path)>0:
                    fin1 = stack.pop(path)
                    ver1 = fin1['vertexB']
                    peso = fin1['weight']
                    pesoTot += peso
                    lt.addLast(lstProm, peso)

                    idNom1 = ver1
                    lt.addLast(lstNom,idNom1)
                elif stack.size(path)==0:
                    seAcabo = True

    return lstNom, lstProm, pesoTot
```

Análisis de complejidad: En este caso, primero se recuperan los id de las estaciones ingresadas por el usuario. Estos id se recuperan a través de una tabla de hash por lo que su complejidad seria apenas de O(K) donde K es una constante teniendo en cuenta que se maneja la tabla de Hash con un factor de carga de 2 para Separte Chaining. Luego se implementa el algoritmo de Dijkstra para encontrar el camino de costo mínimo entre la estación de origen y la de destino ingresadas por el usuario. Este algoritmo tiene una complejidad O(E log(V)) y retorna una cola con los vértices que forman el camino. Finalmente, en el código se recorre toda la cola para recuperar nombres y pesos de los vértices. Esta operación tiene una complejidad de O(H) donde H es el tamaño de la cola.

Conclusión: A la hora de ejecutar el requerimiento, la operación de mayor complejidad es realizar algoritmo de Dijkstra por lo que el requerimiento maneja una complejidad de **O(E log(V)).**

Requerimiento 5

```python
#Requerimiento 5
def routesReport(catalog, iDate, fDate):
    """
    Función que se encarga de encontrar información sobre la dinámica de transporte
    de los usuarios ANUALES dentro del rango de fechas dado, para presentarse como reporte.
    """
    #Obtención de la información del catálogo
    datesRBT = catalog["datesRBT"]
    vertexInfoMap = catalog["vertexInfoMap"]

    iDate = datetime.strptime(iDate, '%Y-%m-%d')
    fDate = datetime.strptime(fDate, '%Y-%m-%d')

    #Creación de contadores
    tripsCounter = 0
    tripsDuraton = 0

    maxDepartureTrips = 0
    maxArrivalTrips = 0
    topDepartureStation = None
    topArrivalStation = None

    listOfVertex = om.values(datesRBT, iDate, fDate) #Array list

    #Creación de listas array para guardar el conteo de horas de llegada y salida.
    endHoursList = lt.newList("ARRAY_LIST")
    begHoursList = lt.newList("ARRAY_LIST")

    for i in range(0, 24):
        lt.addLast(endHoursList, {"Hora":i,"counter":0})
        lt.addLast(begHoursList, {"Hora":i,"counter":0})

    #Iteración sobre la lista de fechas y sus estaciones relacionadas
```

```
#Iteración sobre la lista de fechas y sus estaciones relacionadas
for dateDict in lt.iterator(listOfVertex):
    #TripsNumber and TripsDuration
    tripsNumber = dateDict["tripsNumber"]
    tripDuration = dateDict["tripsDuration"]
    tripsCounter += tripsNumber
    tripsDuraton += tripDuration

    #Estaciones de donde más salen y a donde más llegan
    dictList = dateDict["dictList"]

    for infoDict in lt.iterator(dictList):
        stationID = infoDict["startStation"]
        #print(infoDict)
        stationName = infoDict["startStation"]

        #Búsqueda de la información de esa estación en el mapa de infoVertex.
        vertexInfo = om.get(vertexInfoMap, stationID)
        vertexInfo = me.getValue(vertexInfo) #Aquí ya se tiene el diccionario con la info del vértice

        departuresNumber = lt.size(vertexInfo["connectsToList"])
        arrivalsNumber = lt.size(vertexInfo["connectedToList"])

        #Comparación con máximos
        if departuresNumber > maxDepartureTrips:
            maxDepartureTrips = departuresNumber
            topDepartureStation = stationName

        if arrivalsNumber > maxArrivalTrips:
            maxArrivalTrips = arrivalsNumber
            topArrivalStation = stationName
```

```
        #Ahora, se debe hallar la hora en la que más salen y llegan viajes en ese rango de fechas
        endHoursMap = dateDict["endHoursMap"]
        begHoursMap = dateDict["begHoursMap"]
        keys = mp.keySet(endHoursMap)
        keys2 = mp.keySet(begHoursMap)

        for endHour in lt.iterator(keys):
            getCounter = mp.get(endHoursMap, endHour)
            getCounter = me.getValue(getCounter)
            getCounter = getCounter["counter"]
            #Contador de la lista
            iList = lt.getElement(endHoursList, endHour)
            iList["counter"] += getCounter

        for begHour in lt.iterator(keys2):
            getCounter2 = mp.get(begHoursMap, begHour)
            getCounter2 = me.getValue(getCounter2)
            getCounter2 = getCounter2["counter"]
            #Contador de la lista
            iList2 = lt.getElement(begHoursList, begHour)
            iList2["counter"] += getCounter2

    print("\nMax num viajes iniciados:",maxDepartureTrips)
    print("Max num viajes que llegan:",maxArrivalTrips)

    #Encontrar los rangos de horas mayor
    maxBegHourC = 0
    maxBegHour = None
    maxEndHourC = 0
    maxEndHour = None

    for hour in lt.iterator(endHoursList):
        localCounter = hour["counter"]
        hourInt = hour["Hora"]
```

```
        if localCounter > maxEndHourC:
            maxEndHourC = localCounter
            maxEndHour = hourInt

    for hour2 in lt.iterator(begHoursList):
        localCounter2 = hour2["counter"]
        hourInt2 = hour2["Hora"]

        if localCounter2 > maxBegHourC:
            maxBegHourC = localCounter2
            maxBegHour = hourInt2

    maxBegHour = str(maxBegHour)+":00 - "+str(maxBegHour+1)+":00"
    maxEndHour = str(maxEndHour)+":00 - "+str(maxEndHour+1)+":00"


    #Creación de un diccionario de respuesta
    routesInfo = {"Cantidad de viajes realizados":tripsCounter,
                  "Duración de todos los viajes en segundos":tripsDuraton,
                  "Estación de la que más salen viajes":topDepartureStation,
                  "Estación a la que más llegan viajes":topArrivalStation,
                  "Hora en la que más viajes salen":maxBegHour,
                  "Hora en la que más viajes llegan":maxEndHour}

    return routesInfo
```

Análisis de complejidad: En este caso, se construyó en la carga de datos un árbol RBT para guardar las distintas fechas en las que se realizaron viajes junto a información pertinente.

En la ejecución del requerimiento, lo primero que se realiza es la obtención de la información de los viajes iniciados en el rango de tiempo comprendido entre la fecha inicial y la final. Esta operación (om.values()) tiene complejidad en el peor caso de **O(M)** siendo M la cantidad total de diferentes fechas en el RBT de fechas.

Luego, se comienza a iterar sobre la lista de vértices (estaciones de bicicletas) en las cuales hubo viajes de llegada y salida durante ese rango de fechas. En esta iteración, se busca la información de cada estación en la tabla de hash construida **(O(1.5) – O(2.5))** y se comparan valores como el número de viajes de entrada y salida para hallar las estaciones más concurridas en esas fechas.

Sobre la anterior iteración, se va sumando a dos variables los valores de número de viajes realizados y la duración de estos en segundos, la cual es información ya guardada para cada viaje en la carga de datos. En esta iteración, también se añaden la cantidad de viajes que salieron y entraron en cierto rango de horas para luego hallar las horas pico de salida y entrada de viajes en los días dados por parámetro.

Conclusión: Las operaciones más costosas de este requerimiento son: Primero, obtener todos los valores del árbol Rojo-Negro de las fechas **(O(M) peor caso)**, y después, la realización de un ciclo dentro de otro ciclo. El primer ciclo se podría repetir tantas veces como fechas diferentes hayan en el RBT de fechas, por lo que

sería O(M), el segundo ciclo anidado podría terminar analizando todos los vértices del grafo en caso de que se tomen todas las fechas disponibles en el archivo, por tanto este tiene complejidad O(N) en el peor caso.

Complejidad resultante: O(M) + **O(M*N)**, donde M es la cantidad de fechas diferentes en las que hubo viajes y N es la cantidad de estaciones diferentes existentes en la red de estaciones de la ciudad de Toronto.

Requerimiento 6

```
def getBikeInfo(catalog,nameBike):

    bikeMap = catalog['bikeMap']
    valueKey = mp.get(bikeMap, nameBike)
    info = me.getValue(valueKey)

    totTrip = info['TotalTrips']
    hours = info['TotalTime']
    numOrigen = info['InfoOrigen']['numA']
    numDestino = info['InfoDestino']['numD']
    cont = 1
    big = 0
    for cont in range(1,lt.size(numOrigen)+1,1):
        present = lt.getElement(numOrigen,cont)

        if present > big:
            big=present

    posO = lt.isPresent(info['InfoOrigen']['numA'],big)
    nameO = lt.getElement(info['InfoOrigen']['lstA'],posO)
    cont = 1
    big = 0
    for cont in range(1,lt.size(numDestino)+1,1):
        present = lt.getElement(numDestino,cont)

        if present > big:
            big=present

    posD = lt.isPresent(info['InfoDestino']['numD'],big)
    nameD = lt.getElement(info['InfoDestino']['lstD'],posD)

    return totTrip, hours, nameO, nameD
```

Análisis de complejidad: Para este requerimiento, primero se recupera la información del id de la bicicleta ingresa por el usuario. Esta información se encuentra en una tabla de hash por lo que su complejidad seria apenas de O(K) donde K es una constante, teniendo en cuenta que se maneja la tabla de Hash con un factor de carga de 2 para Separte Chaining. Luego a partir de la información recuperada del mapa, se realizan dos ciclos, el primero para encontrar la estación de origen donde más se ha utilizado esta bicicleta y el segundo para encontrar la estación de destino donde más se ha dejado esta bicicleta. Esta operación tiene una complejidad de O(A+B) donde A y B son respectivamente los tamaños de la lista de datos de origen y datos de destino.

Conclusión: Las operaciones más costosas de este requerimiento son recorrer las dos listas, de origen y de destino, por lo que la complejidad del requerimiento es de **O(A+B)**.