Análisis de complejidad Reto 4-G10

Juan Camilo Falla 201922219 - [j.fallag@uniandes.edu.co](mailto:j.fallag@uniandes.edu.co)

Nicolás Klopstock 202021352 – [n.klopstock@uniandes.edu.co](mailto:n.klopstock@uniandes.edu.co)

Enlace del repositorio: <https://github.com/EDA2021-1-SEC06-G10/Reto4-G10.git>

***Análisis de complejidad:***

1. *Requerimiento 1:*

La complejidad de este requerimiento es la complejidad del algoritmo de Kosaraju para encontrar los componentes conectados del grafo principal. Esta complejidad es *O(V+E)*, donde V es el número de vértices y E es el número de arcos.

Antes de empezar con el algortimo que devuelve la información que se imprime, se necesita el formato correcto de los Landing Points. El usuario ingresa el nombre de los Landing Points, pero ese no es el formato correcto en el grafo. Eso es lo que hace la función compareLpUserLpGraph(). Convierte, por ejemplo, ‘Port Sudan’ a ‘5950-2africa'.

El requerimiento requiere que el usuario ingrese dos Landing Points, los cuales quiere saber si están en el mismo clúster. A continuación, se usa el algoritmo de Kosaraju para encontrar los componentes fuertemente conectados del grafo. Esta información la guarda como valor de una llave del catálogo principal. Luego, se usa la función connectedComponents() de DISCLIB, para saber el número de componentes fuertemente conexos del grafo. Ahora, para saber si dos vértices pertenecen al mismo clúster, se usa la función stronglyConnected() de DISCLIB, la cual retorna un booleano que indica si los dos vértices están o no fuertemente conectados. El entero mencionado anteriormente y este booleano los devuelve y e sa información es la que se imprime.

1. *Requerimiento 2:*

La complejidad es de *O(n)* en donde *n* es la cantidad de “landing points” que hay en la tabla de hash.

Este requerimiento busca mostrar el “landing point” que tiene la cantidad más grande de cables conectados. Para poder resolver este requerimiento, a partir de la carga de la información se va añadiendo a una tabla de hash los nombres de los cables que se encuentran conectados a un “landing point”. Es importante aclarar que la estructura utilizada consta de una tabla de hash cuyas llaves son los números de identificación del “landing point” y los valores son un diccionario de tres llaves en donde está una tabla de hash con todos los nombres de los cables que conectan a un “landing point” como llave y su valor es “None”. Se usa una tabla de hash ya que con esta estructura podemos asegurar que solo se encuentre una vez cada cable.

La función inicia accediendo a la tabla de hash cuyas llaves son los “landing points”. Y se recorre el contenido de cada llave. Para poder hacer esto se requiere extraer el “keySet” del mapa e ir recorriendo todas las posiciones. Se accedía a la tabla de hash con los cables y se iba comparando la cantidad de cables para poder encontrar cual era el que tenía más cables conectados. Este proceso tiene una complejidad de *O(n)* en donde *n* es la cantidad de “landing points” que hay en la tabla. Una vez se sabía qué “landing point” era el que tenía más cables, se realiza nuevamente el recorrido de la tabla para verificar si hay otros “landing points” con la misma cantidad de cables, en caso de que haya, se agrega a la lista de retorno. Dicho proceso también tiene la misma complejidad *O(n)*. Una vez se tenga(n) el (los) “landing point”(s) que tienen más cables, se imprimen los resultados.

1. *Requerimiento 3:*

La complejidad de este requerimiento es la complejidad del algoritmo de Dijkstra para encontrar los caminos más cortos desde un vértice específico a todos los vértices del grafo. Esta complejidad es de, en el peor caso, *O(E\*log(V))*, donde V es el número de vértices y E es el número de arcos.

Primero, el usuario ingresa un país del que quiere encontrar el camino más corto. Lo primero es convertir la cadena de caracteres que ingresa el usuario al formato en el que están los vértices de las capitales, esto es ‘capital\*país’. Eso es lo que hace la función encontrarCapitalDePais().

Teniendo el país en el formato deseado, se usa el algoritmo de Dijkstra para encontrar los caminos más cortos entre el vértice ingresado ‘o’ y todos los vértices del grafo. Esta información se guarda como valor de una llave en el catálogo principal.

Con esta información, primero que cualquier otro algoritmo se le pide al usuario ingresar un segundo país. Este país también queda con el formato correcto, igual que lo mencionado hace unas linas. Ahora, se usa la función pathTo() de DISCLib, la cual devuelve una pila con el camino de ‘o’ al nuevo país que acaba de ingresar el usuario. Esta pila se retorna y se imprime.

1. *Requerimiento 4:*

La complejidad de este requerimiento es *O(n\*m),* donde n es el número de vértices en el grafo grande y m es el número de vértices en el árbol.

Para este requerimiento, el usuario no tiene que ingresar ningún valor. Lo primero es encontrar el árbol de expansión mínima. Esto se hace usando el algoritmo de Prim.

El requerimiento pide diferentes informaciones:

4.1. Lo primero que pide es el total de vértices en el árbol.

4.2. Luego, pide el total de costo del árbol.

4.3. Ahora, el requerimiento pide el camino más largo (en términos de kilómetros).

4.4. Por último, el requerimiento pide el camino más corto (en términos de kilómetros).

Para 4.1: función totalVerticesMST()

Para esta parte del requerimiento, se usa el valor asociado a la llave ‘distTo’ de la estructura ‘search’ que devuelve la función de DISCLib PrimMST(). Este es una tabla de Hash, donde cada llave es un vértice y cada valor el peso de su arco adyacente dentro del árbol de expansión mínima. A este se le pide el KeySet y el size de ese KeySet. Este size es el número de vértices en el árbol.

Para 4.2: función costoTotalArcosMST()

Para esta parte del requerimiento, se usa la función de DISCLib weightMST(), la cual devuelve el costo total de ábol de expansión mínima.

Parte 4.3 y 4.4: función listaDistanciasMST():

El algoritmo para esta parte del requerimiento es la siguiente. Primero, se debe recorrer la lista de todos los vértices en el árbol. Esta lista se tiene que recorrer. La idea es coger un vértice ‘o’, relajar los arcos a partir de ese, wy ver la distancia desde ese vértice hasta el resto de los vértices en el MST usando la función de DISCLib scan(). Se tiene una variable ‘mayor’ y otra ‘menor’, las cuales van guardando la distancia mayor y la menor. Si la distancia entre el vértice ‘o’ y algún vértice es mayor a una ya encontrada anteriormente, mete en un diccionario de forma {‘conexion’: (vérticeA, vérticeB), ‘distancia’: distancia entre vérticeA y vérticeB}. Esto lo hace también para la distancia menor, revisando si la distancia es menor a una ya encontrada.

Este proceso de iteración lo debe hacer para cada vértice, viendo no comparar consigo mismo, para estar seguros de que encuentra la distancia en kilómetros más corta y la más larga.

El diccionario de la distancia mayor y el de la distancia menor se devuelven e una tupla y son los que se usan para el print.

1. *Requerimiento 5:*

La complejidad del requerimiento es O(*n log n)* donde n es la cantidad de “landing points” que se verían directamente afectados cuando falla el “landing point” que ingresa el usuario

Para el quinto requerimiento se realizaron varios procesos. En un principio, el usuario ingresa el nombre del “landing point” y el programa accede a una tabla de hash donde las llaves son los nombres y los valores son el número de identificación del “landing point” asociado a dicho nombre. Proceso cuya complejidad es constante, es decir *O(1)*.

Adicionalmente, este requerimiento también puede ser resuelto gracias a la carga. Nuevamente se usa la misma tabla de hash que es utilizada en el segundo requerimiento, pero en vez de acceder a la tabla de hash con los cables asociados, se accede a otra tabla de hash en la cual las llaves son los “landing points” que se encuentran directamente conectados al “landing point” que ha sido ingresado por parámetro. Esta tabla es recorrida y los valores son guardados en un diccionario y se añade la información a un arreglo del TAD usado en proyectos anteriores. Realizar dicho recorrido tiene un costo *O(n)* en donde *n* es la cantidad de “landing points” que se encuentran directamente conectados al que ingresa el usuario por consola. Una vez se obtiene esta lista, se hace un ordenamiento de tipo “Merge” para poder devolver los “landing points” que se verían afectados en orden descendente usando como criterio la distancia a la que se encuentra cada “landing point” al que tendría un “fallo”. Este proceso tiene una complejidad linearítmica. Es decir *O*(*n log n)*.

***Tablas:***

|  |  |  |
| --- | --- | --- |
| R1 | Tiempo (ms) | Memoria (kB) |
| 3200 | 613.99 | 3418 |
| 1600 | 352.23 | 1894.72 |
| 800 | 174.44 | 1138.9 |

***Tabla 1. Toma de datos para el requerimiento 1.***

|  |  |  |
| --- | --- | --- |
| R2 | tiempo (ms) | memoria (kB) |
| 3200 | 0.422 | 1.76 |
| 1600 | 0.269 | 0.22 |
| 800 | 0.16 | 0.22 |

***Tabla 2. Toma de datos para el requerimiento 2.***

|  |  |  |
| --- | --- | --- |
| R3 | tiempo (ms) | memoria (kB) |
| 3200 | 2019 | 2577 |
| 1600 | 902 | 1361.16 |
| 800 | 300.81 | 757.11 |

***Tabla 3. Toma de datos para el requerimiento 3.***

|  |  |  |
| --- | --- | --- |
| R5 | tiempo (ms) | memoria (kB) |
| 3200 | 1.84 | 6.5 |
| 1600 | 0.415 | 7.12 |
| 800 | 0.34 | 5.48 |

***Tabla 5. Toma de datos para el requerimiento 5.***

***Gráficas y análisis de las pruebas:***

***Grafica 1. Tiempo de ejecución requerimiento 1.***

***Grafica 1.2 Consumo de memoria para el requerimiento 1.***

***Grafica 2. Tiempo de ejecución para el requerimiento 2.***

***Grafica 2.1 consumo de memoria para el requerimiento 2.***

***Grafica 3. Tiempo de ejecución para el requerimiento 3.***

***Grafica 3.1 consumo de memoria para el requerimiento 3.***

***Grafica 5. Tiempo de ejecución para el requerimiento 5.***

***Grafica 5.1 consumo de memoria para el requerimiento 5.***

Después de haber realizado las pruebas con archivos de distintos tamaños, podemos ratificar el análisis teórico que realizamos. En las gráficas de los algoritmos usados sobre grafos, se observa un comportamiento casi lineal en la mayoría de los casos, lo cual es esperado. Esto se debe a que la mayoría de los algoritmos utilizados poseen una complejidad lineal o linearítmica, lo que significa que van a presentar gráficas bastante similares. Solo hubo una discrepancia en el requerimiento 5. No obstante esto era algo que se esperaba ya que la complejidad de este algoritmo depende del archivo utilizado y de cuántos “landing points” tengan una conexión directa con el que ingresa el usuario.