**OBSERVACIONES DEL RETO 3**

Juan Pablo Rodríguez Briceño Cod 202022764

Nicolas Pérez Terán Cod 202116903

**Ambientes de pruebas**

**Máquina 1** **Máquina 2**

|  |  |  |
| --- | --- | --- |
| **Procesadores** | Chip M1 | AMD Ryzen 5  3500U with Radeon  Vega Mobile Gfx  2.10 GHz |
| **Memoria RAM (GB)** | 8 GB | 12 GB (9,95  utilizables) |
| **Sistema Operativo** | MacOS BigSur | Windows 10 Home  64-bits |

**Maquina 1**

**Resultados**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Porcentaje de la**  **muestra**  **[pct]** | **Req – 1** | **Req - 2** | **Req - 3** | **Req - 4** | **Req - 5** | **Req - 6** |
| Small | ms | ms | ms | ms | ms | ms |
| 5% | ms | ms | ms | ms | ms | ms |
| 10% | ms | ms | ms | ms | ms | ms |
| 20% | ms | ms | ms | ms | ms | ms |
| 30% | ms | ms | ms | ms | ms | ms |
| 50% | ms | ms | ms | ms | ms | ms |
| 80% | ms | ms | ms | ms | ms | ms |
| 100% | ms | ms | ms | ms | ms | ms |

**Maquina 2**

**Resultados**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Porcentaje de la**  **muestra**  **[pct]** | **Req – 1** | **Req - 2** | **Req - 3** | **Req - 4** | **Req - 5** | **Req - 6** |
| Small | ms | ms | ms | ms | ms | ms |
| 5% | ms | ms | ms | ms | ms | ms |
| 10% | ms | ms | ms | ms | ms | ms |
| 20% | ms | ms | ms | ms | ms | ms |
| 30% | ms | ms | ms | ms | ms | ms |
| 50% | ms | ms | ms | ms | ms | ms |
| 80% | ms | ms | ms | ms | ms | ms |
| 100% | ms | ms | ms | ms | ms | ms |

**Análisis de complejidad por cada requerimiento.**

**Requerimiento 1 (Grupal): Encontrar puntos de interconexión aerea**

Este requerimiento se logra con dos funciones: *getDurRange(map, min, max)* y *getAllItems(lists)*. El primero tendra una complejidad de N~log(N), ya que utiliza la *om.values(map,min,max)*, el cual buscara en un arbol todos los elementos que estén en el rango [min, max] y los devolverá en forma de lista; esto teniendo que repetir ~Log(N) un total de N veces, en el peor de los casos. Luego, se llama a la funcion *getAllItems(lists)*, que será la encargada de recorrer la lista que devuelve *values(map, min, max)*, entonces tendra una complejidad de N\*N, ya que cada elemento de la lista es otra lista y obligatoriamente tiene que recorrer todos los elementos de cada una, sin embargo, nunca se devuelve.  
La complejidad final es O(N^2 + N~Log(N))

**Requerimiento 2 (Grupal): Encontrar clusteres de trafico aereo**

Este requerimiento se resuelve con *findCluster(catalog,IATA1,IATA2)* Primero, se usan los IATA para obtener el ID de cada aeropuerto, luego se va a aplicar el algoritmo Kosaraju con *scc.KosarajuSCC(graph)* con complejidad de E+V. Posteriormente, se llama la función *scc.connectedComponents(graph)* el cual obtiene el numero de clusteres o componentes fuertemente conectados. Por ultimo, se llama a la función *scc.stronglyConnected(grafo,aeropuerto1,aeropuerto2)* que verifica si los aeropuertos dados por parámetro pertenecen al mismo cluster. La complejidad final es O(E+V) ya que el único algoritmo que se usa es Kosaraju.

**Requerimiento 3 (Grupal): Encontrar la ruta mas corta entre ciudades**

Este requerimiento se resuelve con *countTime(catalog,dateMin,dateMax)*, el cual tiene una complejidad de ~Log(N) + ~Log(N) ya que trabaja con los datos que se encuentren dentro de un RBT y que también se encuentre dentro del rango de fechas que se haya establecido por parámetro. Primero, se extraen las llaves que cumplan la condición de estar en el rango de fechas usando *keys(map,dateMin,dateMax)* con complejidad de N\*N como se explico anteriormente. Por último, se llama la función *onlyMapValue(map,key)* cuya complejidad de ~Log(N) y se llama por cada llave que exista en el rango. La complejidad final es (N^2+~Log(N))

**Requerimiento 4 (Grupal): Utilizar las millas de viajero  
Nota:** La sección del catálogo utilizada en esta parte es un RBT (Ordenado a partir de la longitud), que almacena en cada llave otro RBT (Ordenado a partir de la latitud), el cual a su vez en cada llave almacena una lista con los avistamientos.

Este requerimiento se divide en dos partes: *getLonRange(catalog, min, max)* y *getLatRange(list, minLat, maxLat)*.   
Primero, *getLonRange(catalog, min, max)* posee una complejidad de O(~Nlog(N)), porque tiene que buscar los elementos de un RT en un rango [min, max] y devolverlos en una lista; los datos para ser tomados tienen una complejidad de O(~Log(N)), y se repiten N veces, en el peor de los casos.

Segundo, *getLatRange(lst, min, max)*. Que se divide en cuatro partes: la primera, tiene una complejidad de O(N), porque recorre toda la lista de mapas que se le dio por parámetro; la segunda, tiene complejidad de O(~NLog(N)) porque saca los valores de un mapa de listas, dado un rango de latitud [min, max] (Se utiliza *om.values(map, min, max)*, cuya complejidad que ya se explicó anteriormente); la tercera, tiene complejidad de N\*N, porque recorre cada elementos de la lista de listas y lo va añadiendo a una nueva lista, lo cual tiene complejidad de O(N) porque añadirá N elementos; y por último, hara un mergeSort, que tiene complejidad de O(Nlog(N)).  
La complejidad final sería de O( 2N + ~2Nlog(N) + N^2 + NlogN)

**Requerimiento 5 (Grupal): Cuantificar el efecto de un aeropuerto cerrado**

Este requerimiento se resuelve con *closedAirport(catalog,IATA)* Primero, se usan los IATA para obtener el ID de cada aeropuerto, luego se obtienen los valores originales de ambos grafos. Posteriormente, se hacen los respectivos cálculos de los nuevos valores que tendrían los grafos si el aeropuerto fuera removido del grafo. Al final, llama a la función *closedAirportDF(catalog,list)* el cual se encarga de organizar los vértices afectados según su ID usando MergeSort y de crear el Dataframe. La complejidad final es O(nLog(n)) ya que solo se usa el algoritmo de MergeSort.