**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** | **Carga de datos** |
| Small | 105.898 ms | 107.275 ms | 102.646 ms | ms | 2.0479 ms | 715.944 Ms |
| 5% | 117.503 ms | 125.63 ms | 106.360ms | ms | 2.233ms | 734.936 ms |
| 10% | 136.657 ms | 146.1009 ms | 119.391ms | ms | 3.6789 ms | 826.651 Ms |
| 20% | 218.16 ms | 309.928 ms | 162.246ms | ms | 4.8460 ms | 1000.981Ms |
| 30% | 317.369 ms | 427.6949 ms | 312.155ms | ms | 11.578 ms | 1191.97 Ms |
| 50% | 619.518 ms | 957.535 ms | 390.566 ms | ms | 11.58699 ms | 1973.908 Ms |
| 80% | 1305.638 ms | 1592.625 ms | 678.014 ms | ms | 23.1019 ms | 3855.9519 Ms |
| 100% | 1990.610 ms | 2234.669ms | 137.228 ms | ms | 31.6080 ms | 5603.656 Ms |

**Maquina 2**

**Resultados**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Porcentaje de la**  **muestra**  **[pct]** | **Req – 1** | **Req - 2** | **Req - 3** | **Req - 4** | **Req - 5** | **Carga de datos** |
| 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**

Para lograr este requerimiento, implementamos la función *findConnections(catalog),* la cual consiste extraer el grafo dirigido y el grafo no dirigido del catalogo (catalog[‘routes’] y catalog[‘connections]), para posteriormente recorrer cada uno de sus vértices y extraer el grado de cada uno. La complejidad de esta sección seria 2N, siendo N el numero de vertices, porque solo esta recorriendo los vértices dos veces.

Luego se creara una nueva lista -donde se almacenaran cada uno de los aeropuertos que tengan al menos una conexion- pero con la diferencia de que ahora cada uno cuenta con la información de sus grados. Luego se utilizara un mergesort en la lista comparando sus grados, y se tomaran los primeros 5 elementos para mostrarlos en pantalla. La complejidad de esta parte seria NlogN.  
  
La complejidad final seria O(NlogN)

**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**

Para este requerimiento, se inicio con un la obtención de las ciudades disponibles dado un nombre. Para esto, se utilizo una función llamada *getCityInfo(map, city)*, pasando por parámetro:

City: Nombre de una ciudad.

Map: Tabla de hash, donde las llaves son los nombres de las ciudades y los valores son listas donde está la información de cada una de las ciudades.

Esta parte tiene una complejidad de O(1) para obtener la lista de ciudades homónimas, y O(c), donde ‘c’ es el numero de ciudades homónimas de un nombre dado.

Luego, de que el usuario elija la ciudad del listado se procederá a utilizar findRoute(catalog, indice1, indice2, inicio, destino) donde los índices son las elecciones dadas por el usuario y los parámetros ‘inicio’ y ‘destino’ son las listas de las ciudades homónimas, de inicio y destino, respectivamente. Se extraerán la ciudad elegida por el usuario y se utilizaran sus coordenadas para pasarlo como parámetro a getNear(catalog, cordenada), función donde se recorrerán todos los elementos del mapa de aeropuertos, en el peor de los casos, para comparar las coordenadas y determinar cual es el mas cercano a la ciudad dada. Al final, la complejidad seria de O(N), siendo N el numero de aeropuertos que hay.

Posteriormente, cuando se hayan encontrado los aeropuertos de salida y llegada mas cercanos de cada punto, se procederá a utilizar el algoritmo de Dijsktra para encontrar las rutas mas cortas. El algoritmo de Djisktra tiene una complejidad de ElogV, donde E el es numero de arcos y V el numero de Vertices.  
  
Es asi como la complejidad quedaría N(ElogV + O(V) )

**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.