OBSERVACIONES DEL RETO 4

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% Large	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	218.75ms	156.25ms	156.25ms	ms	0.0ms	2671.875ms
5%	328.125ms	187.5ms	171.875ms	ms	0.0ms	2906.25ms
10%	250.0ms	265.625ms	265.625ms	ms	0.0ms	3125.0ms
20%	390.625ms	515.625ms	625.0ms	ms	0.0ms	4140.625ms
30%	640.625ms	953.125ms	828.125ms	ms	15.625ms	5468.75ms
50%	1484.375ms	1968.75ms	2562.5ms	ms	15.625ms	10000.0ms
80%	4031.25ms	4453.125ms	8750.0ms	ms	15.625ms	22562.5ms
100% Large	6234.375ms	5843.75ms	13250.0ms	ms	31.25ms	36781.25ms

Análisis de complejidad por cada requerimiento.

Requerimiento 1 (Grupal): Encontrar puntos de interconexión aérea

Para lograr este requerimiento, implementamos la función *findConnections(catalog),* la cual consiste extraer el grafo dirigido y el grafo no dirigido del catálogo (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 número de vértices, porque solo está recorriendo los vértices dos veces.

Luego se creará una nueva lista -donde se almacenarán cada uno de los aeropuertos que tengan al menos una conexión- 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 sería O(NlogN)

Requerimiento 2 (Grupal): Encontrar clústeres de tráfico aéreo

Este requerimiento se resuelve con *findCluster(catalog,IATA1,IATA2)* Primero, se usan los IATA para obtener el ID de cada aeropuerto con la función *getIDbyIATA(catalog,IATA),* 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 número de clústeres o componentes fuertemente conectados del grafo. Por último, se llama a la función *scc.stronglyConnected(grafo,aeropuerto1,aeropuerto2)* que verifica si los aeropuertos dados por parámetro pertenecen al mismo clúster.

La complejidad final es O(E+V) ya que el único algoritmo que añade complejidad es el de 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 utilizarán sus coordenadas para pasarlo como parámetro a *getNear(catalog, coordenada),* 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 más cercanos de cada punto, se procederá a utilizar el algoritmo de Dijkstra para encontrar las rutas mas cortas. El algoritmo de Dijkstra tiene una complejidad de ElogV, donde E él es número de arcos y V el número de Vértices.

Es así como la complejidad quedaría N (ElogV + O(V))

Requerimiento 4 (Grupal): Utilizar las millas de viajero

Este requerimiento empieza por obtener las ciudades disponibles de acuerdo a un parámetro dado por el usuario, así como lo hace el requerimiento 3. De igual manera, este requerimiento hace uso de la función *getCityInfo(map, city).*

Luego de que el usuario elija la ciudad del listado también se hace uso de *findRoute(catalog, indice1, indice2, inicio, destino)* salvo que el indice1 y el índice2 son los mismos, así como también inicio es lo mismo a destino ya que solo se requieren los datos de una sola ciudad.

Posteriormente se llama a la función *controller.useMiles(catalog,miles,indice1,inicio)* la cual, además de extraer información de la ciudad escogida por el usuario y sus aeropuertos, llamara a *model.useMiles(catalog,miles,indice1,inicio).* Donde se aplicará el algoritmo de Prim en su versión lazy con complejidad de E*V y se extraen los datos solicitados.

La complejidad final es O(E*V) ya que únicamente se realiza uso del algoritmo Prim(Lazy) y el resto de las funciones solo realizan consultas de datos sobre Prim

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 del aeropuerto con la función *getIDbyIATA(catalog,IATA),* luego se obtienen los números de vertices y arcos de ambos grafos haciendo uso de *numEdges(catalog)* y *numVertices(catalog).*

Posteriormente, se hacen los respectivos cálculos de los nuevos valores que tendrían los grafos si el aeropuerto fuera removido del grafo, entre los cuales se resta uno al número de vértices de ambos grafos y se calcula el número restante de arcos en el grafo haciendo uso de *indegree(catalog,ID), outdegree(catalog,ID)* y *degree(catalog,ID).* Al final, llama a la función *closedAirportDF(catalog,list)* el cual se encarga de organizar los aeropuertos afectados según su ID usando MergeSort y de crear el Dataframe.

La complejidad final es O(NLog(V)) ya que solo se usa el algoritmo de MergeSort en los aeropuertos adyacentes al vértice del aeropuerto consultado, los cuales están representados por V.