Análisis de Complejidad Temporal
Reto 4

Juan Camilo Colmenares Ortiz - 202011866 - j.colmenares@uniandes.edu.co
Juan Andrés Ospina Sabogal - 202021727 - ja.ospinas1@uniandes.edu.co

Máquina de Pruebas:

Máquina 2
Intel® Core ™ i5-7267U @3.1GHz
8.0 Gb
macOS 11.2.1 Big Sur

Requerimiento 1
Parámetros de prueba:
Landing point 1: Redondo Beach
Landing point 2: Vung Tau

Tiempo (ms)	Uso de Memoria (kb)
482.474	**134.146**

El requerimiento 1 fue implementado con la función req1(). Se apoya en un grafo dirigido (analyzer['connections_normal']), cuyos vértices son los id's de cada landing point. En este grafo no se tienen en cuenta los id's de los diferentes cables. La función req1() utiliza el algoritmo de Kosaraju para encontrar los componentes fuertemente conectados (clústeres) del grafo. Devuelve una tupla cuyos valores son num_clusters (el número de clusters) y estan_o_no. La variable estan_o_no guarda el resultado de la función stronglyConnected del algoritmo de Kosaraju, que determina si dos vértices están fuertemente conectados o no (mismo cluster o no). La complejidad del requerimiento es O(V+E), donde V es la cantidad de landing points (vértices) y E es la cantidad de conexiones entre landing points (edges, o arcos).

Requerimiento 2

Tiempo (ms)	Uso de Memoria (kb)
137.886	**314.632**

El requerimiento 2 fue implementado con la función req2(). Se apoya en un grafo dirigido (analyzer['connections']) y en una tabla de hash (analyzer['interconnections']). Los vértices del grafo son una cadena de la siguiente forma: "landing_point_id"+"_"+"cable+id". Esto significa que para un mismo landing_point puede haber varios vértices en el grafo. La tabla de hash relaciona el id de cada landing point con una lista de datos. El primer elemento de la lista es la cantidad de cables que conectan en dicho landing point. En la solución del requerimiento es importante notar que un mismo cable en un landing point puede conectar, aparentemente, dos veces. Esto se debe a que un mismo cable puede entrar y salir, no solo entrar. Se tuvo esto en cuenta en la creación de la tabla de hash de tal forma que, para cables que entran y salen, no se sumen valores adicionales. El grafo mencionado anteriormente solo es de utilidad para la creación de la tabla de hash. Dicho proceso es hecho durante la carga de datos. La función req2() obtiene la cantidad de cables para un landing point determinado y, si conecta más de uno, se retorna una cadena con la información requerida de dicho landing_point. La complejidad temporal de la función req2() es de orden constante O(6), pues no se realiza ningún tipo de iteración y obtener un valor de una tabla de hash dada la llave es O(1). Sin embargo, req2() se llama para todos los landing points cargados. Por lo tanto, la complejidad del requerimiento es O(N), donde N es la cantidad de landing points.

Requerimiento 3
Parámetros de prueba:
País A: Colombia
País B: Venezuela

Tiempo (ms)	**Uso de Memoria (kb)**
41966.033	**28.647**

El requerimiento 3 fue implementado con la función req3(). En las instrucciones del reto se pide que se tenga en cuenta el landing_point de cada capital para encontrar el camino mínimo. Sin embargo, esto no es posible para todos los países, pues no todas las capitales son landing_points. Un buen ejemplo de esto es Colombia: Bogotá no es una ciudad con salida al mar y, por lo tanto, no es un landing point. Como equipo de trabajo tomamos la decisión de implementar una solución mucho más costosa de tiempo pero que garantiza el camino de menor distancia posible desde cada país: revisar todos los landing points de los países en cuestión para encontrar el camino menor. Primero, la función req3() recurre a la función get_landing_points_by_country() que recibe un país y devuelve un arreglo con todos los landing points de dicho país. Se itera sobre ambos arreglos, garantizando que se revisa el camino para todos los landing points tanto del país de origen como el país de destino. Para cada iteración, se recurre a la función

get_list_of_vertices(), la cual recibe el id de un landing point y devuelve una lista con todos los vértices de dicho landing point (cada vértice es un cable diferente). Para cada par de vértices (origen y destino) se llama el algoritmo de Dijkstra. Si la distancia del camino es la menor, esta se guarda en una variable. Al final, se recorre la pila pathTo que devuelve el algoritmo para conformar una cadena con todo el recorrido.

La complejidad de todo el algoritmo depende de muchos factores. En principio, depende de la cantidad de landing points de los paises de origen y destino, de la cantidad de cables que pasen por cada landing point y de la complejidad del algoritmo de Dijkstra. En notación O, eso sería (aproximadamente): O(A*B*Va*Vb*(V+ElogV)), donde A es la cantidad de landing points del país de origen, B la cantidad de landing points del país de destino, Va y Vb la cantidad de vértices de cada landing point para ambos países (nótese que esto puede ser diferente para cada landing point, pues no todos tienen los mismos cables y, por lo tanto, hace variar aún más la complejidad del algoritmo) y V+ElogV es la complejidad del algoritmo de Dijkstra.

Si se pudiese tener en cuenta la capital de cada país como landing point, la complejidad del mismo sería mucho menor: O(Va*Vb(V+ElogV)).

Requerimiento 4

Tiempo (ms)	**Uso de Memoria (kb)**
719.593	**24.909**

El requerimiento 4 fue implementado con la función req4(). Se llamó el algoritmo de Prim para encontrar el MST (minimum spanning tree) del grafo analyzer['connections_normal'], cuyos vértices son los landing points. Este grafo no tiene en cuenta diferentes cables. Por ende, la complejidad del requerimiento es O(ElogV), siendo E la cantidad de conexiones (arcos del grafo) y V la cantidad de landing points (vertices).

Requerimiento 5
Parámetro de prueba: Fortaleza

Tiempo (ms)	**Uso de Memoria (kb)**
2.286	**8.522**

El requerimiento 5 fue implementado con la función req5(). Primero, se obtiene la lista de vertices para el grafo analyzer['connections'] del landing point solicitado por el usuario. Se

recorre la lista, y para cada vértice se obtiene una lista de arcos adyacentes con el método adjacentEdges(). Para cada uno de los arcos, se obtiene el país de destino y el peso (distancia) del mismo. Los datos obtenidos se guardan en una lista auxiliar, sin guardar países repetidos. Finalmente, se realiza mergesort en la lista auxiliar, para que esta quede en orden descendente. La complejidad temporal se puede denotar como O(V*A+Nlog(N)), donde V es el número de cables en el landing point ingresado, A es el número de conexiones de cada cable saliendo del landing point ingresado, y N es la cantidad de países con los cuales el landing point tiene una conexión directa (un solo arco).