**Análisis de rendimiento Reto 4**

**Carga**

Para la carga utilizamos un dígrafo donde los vértices son tuplas conformadas por el código del landing point o la capital y por el nombre del cable que representa a dicho vértice. Los pesos de las aristas son las distancias (en kilómetros) entre cada uno de los vértices. La distancia que utilizamos entre landing points fue sacada del connections.csv de “cable\_length”. La distancia que utilizamos entre landing points y capitales fue determinada utilizando la librería haversine. Además del grafo utilizamos varias tablas de hash para hacer los procesos de, cambio de país a capital, nombre de landing point a código del landing point, vértice a país donde se encuentra el vértice y viceversa, fueran más rápidos. Esta carga está compuesta por varias funciones, pero la mayoría de estas tienen una complejidad lineal debido a que hacen un recorrido de las líneas de un csv o hacen un recorrido de elementos existentes en el analyzer. El tiempo de carga obtenido fue de 75917.19ms en promedio y con un consumo de memoria de 23828.39kb.

**Requerimiento 1**

Para el requerimiento 1 se implementó el algoritmo Kosaraju utilizando la función  scc.KosarajuSCC(). Este algoritmo nos ayuda a encontrar los componentes conectados del grafo y tiene una complejidad de O(V+E) donde V=vertices y E=edges o aristas. Como este algoritmo es la base del requerimiento entonces la complejidad en general del requerimiento uno es lineal (O(V+E)). El tiempo obtenido para este requerimiento fue de 5432.27ms en promedio y el uso de memoria de 133.55kb.

**Requerimiento 2**

En el requerimiento 2 se obtiene una lista con todos los vértices del grafo principal y se hace un recorrido sobre esta lo cual significa que la complejidad de este es de O(V). El tiempo de ejecución fue de 1267.44ms en promedio y el uso de memoria de 5.05kb.

**Requerimiento 3**

El requerimiento 3 requería conseguir el camino más corto entre dos vértices del grafo entonces utilizamos el algoritmo de Dijkstra haciendo uso de la función djk.Dijkstra() que tiene una complejidad temporal de O(ElogV). El tiempo de ejecución fue de 2735.52ms en promedio y el uso de memoria de 31.35kb.

**Requerimiento 4**

Para el requerimiento 4 utilizamos el algoritmo Prim para conseguir el MST del grafo original. Para esto utilizamos la función prim.PrimMST() que tiene una complejidad de O(ElogE). El tiempo de ejecución fue de 1837.35ms en promedio y el uso de memoria de 21.44kb.

**Requerimiento 5**

En el requerimiento 5 se hace un recorrido de todos los vértices del grafo lo que significa una complejidad de O(V) y dentro de este recorrido se hace un recorrido de los vértices adyacentes a un vértice especifico O(a). Además, al final se utiliza un merge sort que tiene una complejidad de O(NlogN) para organizar de forma descendente la lista final. Sin embargo, como en todos los casos los vértices adyacentes y la lista final son tan pequeños en relación a la cantidad de vértices recorrida al inicio se podría decir que la complejidad temporal del requerimiento es lineal O(V). El tiempo de ejecución fue de 2585.53ms en promedio y el uso de memoria de 18.67kb.

**Requerimiento 6**

La complejidad de este requerimiento es de O(V) debido a que se hace un recorrido a los vértices del grafo y los demás recorridos que se hacen son fuera de este recorrido e insignificantemente cortos. El tiempo de ejecución fue de 1182.89ms en promedio y el uso de memoria de 6.5kb.

**Pruebas de tiempo y memoria**

