DOCUMENTO ANÁLISIS RETO 4

Requerimiento 2 - Javier Cerino Cod 202020873

Requerimiento 3 - Marco Zuliani Cod 202022412

**Integrantes del grupo:**

Marco Zuliani (código estudiante: 202022412, Usuario Uniandes: m.zuliani, Nombre de usuario Git: poloiva)

Javier Cerino (código estudiante: 202020873, Usuario Uniandes: j.cerino, Nombre de usuario Git: 2jc26)

**Requerimiento 1:**

* Alumnos que trabajaron en este requerimiento: Javier Cerino y Marco Zuliani.
* Análisis de complejidad:

O(E+V)

Para este requerimiento comenzamos solicitando al usuario el nombre de dos Landings Points de los cuales desea conocer si se encuentran en el mismo componente fuertemente conectado. Luego se hace uso del algoritmo de Kosaraju para obtener todos los componentes fuertemente conectados del grafo los cuales se guardan en la llave ‘components’ del catálogo[Complejidad O(E+V)]. Luego utilizando la función connectedComponents obtenemos el número de componentes fuertemente conectados que es el primer resultado a mostrar al usuario[Complejidad O(Constante)]. Con el fin de saber si los dos landings points que ingresó el usuario hacen parte del mismo componente fuertemente conectado primero obtenemos el id y los cables de los landing points. Esto lo hacemos usando una tabla de hash creada en la carga de datos que relaciona el nombre del landing point con algunos de sus datos[Complejidad O(constante)]. Teniendo el id de los landings y los nombres de los cables que salen de este podemos obtener los nombres de los vértices en el grafo que hacen referencia a ese landing point[Complejidad O(constante)]. Finalmente se va probando cada combinación posible <landing\_id-Cable><landing\_id-Cable> entre ambos landing points y se va ejecutando a su vez la función stronglyConnected[Complejidad O(Constante)] con estos dos vértices para averiguar si están en el mismo componente fuertemente conectado[Complejidad O(N1\*N2)]. En el momento en el que la función stronglyConnected indica que están en el mismo componente, ósea retorna True, se detiene la función sin necesidad de revisar todas las combinaciones posibles, pero en caso contrario se recorrerán todas la combinaciones y se retornará False.

Se debe resaltar que la función del algoritmo de Kosaraju y las funciones connectedComponents y stronglyConnected se encuentran implementadas en el archivo scc.py.

* Análisis de Tiempo:

El requerimiento es realmente eficiente ya que cuando se ejecuta la función por primera vez indica el resultado en 1.5 segundos y cuando no es la primera vez lo indica en 0 segundos. Esto se debe a que solo la primera vez se encuentran los componentes fuertemente conectados y las otras veces simplemente se accede a este resultado ya que queda almacenado en una llave del catálogo.

* Análisis de Memoria:

En este requerimiento el uso de memoria es muy bueno ya que solo se hace uso de una estructura de datos adicional en donde se almacenan los componentes fuertemente conectados. Esto se debe a que las otras estructuras ya se habían implementado en la carga de datos.

**Requerimiento 2:**

* Alumnos que trabajaron en este requerimiento: Javier Cerino y Marco Zuliani.
* Análisis de complejidad:

O(V)

Para este requerimiento comenzamos creando una lista vacía en la que se insertaran los vértices que tengan el mayor número de cables. Luego recorremos las llaves de la tabla de hash que contiene los landing points[Complejidad O(V)] y vamos mirando el número de cables con los que tiene conexión[Complejidad O(constante)]. En el caso en el que tenga un numero de cables mayor se cambia el mayor número de cables por este valor, la lista creada al inicio se pone como vacía y se inserta el vértice. En el caso en el que el vértice tenga el mismo número de cables que el mayor número, el vértice se agrega a la lista. Finalmente se retorna la lista con los vértices que tienen conexión con más cables y el número de cables que estos conectan.

* Análisis de Tiempo:

El requerimiento es realmente eficiente en tiempo ya indica el resultado en 62.5 milisegundos.

* Análisis de Memoria:

En este requerimiento el uso de memoria también es muy bueno ya que solo se hace uso de una estructura de datos adicional que es la lista en donde se almacenan los vértices que mayor número de cables tienen. Esto se debe a que las otras estructuras ya se habían implementado en la carga de datos.

**Requerimiento 3:**

* Alumnos que trabajaron en este requerimiento: Javier Cerino y Marco Zuliani.
* Análisis de complejidad:

O(V)

Para este requerimiento comenzamos solicitando al usuario que ingrese el nombre de los países desde los que se quiere enviar información. Debido a que en el proyecto decidimos poner el nombre del país a los landing points ubicados en la capital no necesitamos en ningún momento obtener el nombre de la capital. Luego utilizamos la implementación del algoritmo de Dijkstra para obtener los caminos de ruta mínima, en términos de distancia, partiendo desde el País1 hacia todos los demás[Complejidad O(E\*log(V))]. Posteriormente utilizamos las funciones distTo[Complejidad O(Constante)] y pathTo[Complejidad O(V)] para obtener la distancia total que hay entre los dos países y el camino a realizar para llegar desde el País1 al País2.

Se debe resaltar que la función del algoritmo de Dijkstra y las funciones distTo y pathTo se encuentran implementadas en el archivo dijkstra.py.

* Análisis de Tiempo:

El requerimiento es realmente eficiente en tiempo ya indica el resultado en 31.25 milisegundos.

* Análisis de Memoria:

En este requerimiento el uso de memoria también es muy bueno ya que solo se hace uso de una estructura de datos adicional que donde se almacenan las rutas de distancia mínima. Esto se debe a que las otras estructuras ya se habían implementado en la carga de datos.

**Requerimiento 4:**

* Alumnos que trabajaron en este requerimiento: Javier Cerino y Marco Zuliani.
* Análisis de complejidad:

\_\_\_\_

* Análisis de Tiempo:
* Análisis de Memoria:

**Requerimiento 5:**

* Alumnos que trabajaron en este requerimiento: Javier Cerino y Marco Zuliani.
* Análisis de complejidad:

\_\_\_\_\_

* Análisis de Tiempo:
* Análisis de Memoria: