DOCUMENTO ANÁLISIS RETO 4

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) donde E es el número de arcos y V el número de vértices

  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 $O(N_1*N_2)$]. 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. Memoria [kB]: 5404,457.

Requerimiento 2:

- Alumnos que trabajaron en este requerimiento: Javier Cerino y Marco Zuliani.
- Análisis de complejidad:
  O(V) donde V es el número de Vértices

  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. Memoria [kB]: 19,648.

Requerimiento 3:

- Alumnos que trabajaron en este requerimiento: Javier Cerino y Marco Zuliani.
- Análisis de complejidad:
  O(V) donde V es el número de Vértices

  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. Memoria [kB]: 3740,084.

Requerimiento 4:

- Alumnos que trabajaron en este requerimiento: Javier Cerino y Marco Zuliani.
- Análisis de complejidad:
  O(E*log(V)) donde V es el número de Vértices y E el número de arcos
  Para este requerimiento comenzamos encontrando el mst del grafo con el uso de la funcion PrimMST implementada en el archivo prim.py [Complejidad O(E*log(V)]. Una vez conluido el algortimo de PrimMST, cuyo retorno es una tabla de hash que contiene los nodos con su respectiva conexión al siguiente nodo, se interpreta el resultado de este para reconstruir en cierta manera los posibles caminos y así encontrar el de mayor longitud en vértices. Para realizar esto se recorren la lista de los vértices existentes en al tabla de hash lo cual tiene una complejidad de O(V) siendo V el número total de vértices existentes pertenecientes al arból de expansión mínima. Vamos realizando un recorrido dadas las conexiones de los vértices a las cuales tenemos acceso gracias a la tabla de hash retornada por la función Prim. Realizamos un conteo desde un vértice aleatorio hasta que llegue a una conexión "None". Le asignamos al recorrido realizado la distancia hasta este vértice con conexión "None" recorriendo la lista de conexiones. En el peor de los casos esta puede asumir una complejidad de O(V) siendo V el número de vértices del MST.
  Una vez identificado la mayor rama del MST se agrega esta a un lista y se recorrera en su totalidad para imprimir el resultado, esto en el peor de los casos posibles sería O(V), comportamiento bastante raro debido a la naturaleza del grafo.
- Análisis de Tiempo:
  Media de 15 a 18 segundos por ejecución lo que nos da un tiempo de ejecución aceptable, aunque bastante peor comparado con los demás requerimientos.
- Análisis de Memoria:
  En este requerimiento el uso de memoria es bastante debido a que las únicas estructuras de datos significativas son una tabla de hash que permite identificar los elementos ya visitados y una lista cuyo contenido se va reiniciado a medida que se desarrolla el requerimiento. Memoria [kB]: 6974.944.

Requerimiento 5:

- Alumnos que trabajaron en este requerimiento: Javier Cerino y Marco Zuliani.
- Análisis de complejidad:
  O(N1*N2) donde N1 es el número de cables que salen del landing ingresado por el usuario y N2 el número de nodos a los que se llega a través de cada uno de estos cables.

    Para este requerimiento comenzamos solicitando al usuario que ingrese el nombre del landing pinta en el que se genera el fallo. Verificamos que este landing point realmente exista y obtenemos su id, el país en el que se encuentra y la distancia desde la capital a este[Complejidad O(Constante)]. Dentro de una tabla de hash creada al inicio, introducimos como llave el país y como valor la distancia del landing[Complejidad O(Constante)]. Luego, debido a como decidimos conectar los vértices, recorremos los arcos adyacentes al nodo introducido por el usuario los cuales conectan con un landing cuyo nombre es <id_landing*cable> [Complejidad O(N1)]. Posteriormente recorremos los arcos adyacentes a los nodos adyacentes mencionados en el paso anterior para conocer los landing points a los que estos están conectados. Por cada uno de los

landing points adyacentes obtenemos el país y el peso para llegar a este[Complejidad O(N2)]. Finalmente revisamos si ese país ya había sido agregado a la tabla de hash mencionada anteriormente con el fin de revisar si el peso que se encuentra de la tabla es mayor o menor al nuevo peso obtenido, en caso de ser menor el valor que ya se encontraba en la tabla se remplaza[Complejidad O(Constante)]. Por lo anterior, dentro de la tabla de hash tenemos los países con los que se conecta el landing introducido por el usuario y la menor distancia para llegar a estos. Para concluir los datos de la tabla de hash los ponemos en una lista [Complejidad O(N)] y usando mergesort sobre esta obtenemos la lista ordenada en el orden solicitado en el enunciado[Complejidad O(N*log(N))]. Por lo tanto, el tamaño de esta lista es el número de países afectados y dentro de la lista se encuentran las distancias desde el landing point indicado por el usuario hasta el landing point más cercano con el que se conecta en cada país.

- Análisis de Tiempo:

  Este requerimiento es realmente eficiente ya que se demora 1 segundo en dar el resultado.

- Análisis de Memoria:

  Este requerimiento es eficiente en memoria al inicio ya que solo utiliza las estructuras creadas con la carga de datos y una tabla de hash adicional. Ya finalizando el requerimiento su eficiencia en memoria disminuye debido a que con el objetivo de organizar los datos en orden decreciente nos fue necesario pasar los datos de la tabla de hash a una lista para usar mergesort. Memoria [kB]: 8,793.