Reto 4

Grupo 6 - Sección 6

Carlos Arturo Holguín Cardenás, 202012385, c.holguinc@uniandes.edu.co

Daniel Hernández Pineda, 202013995, d.hernandez24@uniandes.edu.co

Requerimiento 1

Preámbulo

Entiéndase: **E** = número de rutas en el dígrafo; **V** = número de aeropuertos en el dígrafo; **N** = número de vértices interconectados.

Análisis de complejidad

El requerimiento se compone de tres funciones:

1. REQ1(): Esta es la función principal del requerimiento, en primer lugar, se hace uso del método gr. vértices (), el cual posee una complejidad de **V**. Posteriormente, se hace un ciclo de complejidad **V,** dentro del cual se hace uso de los métodos outdegree() e indegree(), lo cual representa una complejidad de orden constante. Dentro del mismo ciclo, se hace uso de la función DatatreeREQ1(). Finalmente se, hace un recorrido inorden al árbol el cual posee una complejidad de **N** (V en el peor caso).
2. DataREQ1(): Esta función se encarga de crear una lista con los datos necesario para resolver el requerimiento, gran parte de estas operaciones son de orden constante, por tanto, no afectan trascendentalmente la complejidad del requerimiento.
3. DatatreeREQ1(): Esta función se encarga de agregar los datos a un árbol rojo-Negro, con complejidad **log(E)**. El resto de las operaciones son de orden constante, por tanto, no afectan trascendentalmente la complejidad del requerimiento.

Finalmente, se concluye que el requerimiento 1 tiene una complejidad de **O(V*log(E)).** En la práctica, se comportará más como **O(V)** únicamente, dado que el árbol que se crea en la tercera función tendrá pocos elementos.

Requerimiento 2

Preámbulo

Entiéndase: **E** = número de rutas en el dígrafo; **V** = número de aeropuertos en el dígrafo.

Análisis de complejidad

Para este requerimiento, simplemente se implementó el algoritmo de Kosaraju para obtener las componentes fuertemente conectadas en el dígrafo. Específicamente, se implementó el método scc.KosarajuSCC(), el cual realiza la búsqueda con complejidad **E + V**, y se implementó el método

scc.stronglyConnected(), el cual verifica si dos vértices del dígrafo están fuertemente conectados. Este último tiene orden de crecimiento constante, puesto que accede directamente a la componente a la que pertenece cada vértice por medio de la búsqueda ya realizada, y las compara.

De esta manera, el Requerimiento 2 tiene una complejidad de **O(E + V)**

Requerimiento 3

Preámbulo

Entiéndase: **E** = número de rutas en el dígrafo, **V** = número de aeropuertos en el dígrafo; **a** = cantidad máxima de longitudes; **b** = cantidad máxima de latitudes; **x** = máximo de veces que habrá ampliar el rango de búsqueda (si bien no se puede saber con precisión, dada la naturaleza del sistema de coordenadas será significativamente menor que E y V).

Para este requerimiento, se creó un árbol auxiliar en el que se organizan las coordenadas de los aeropuertos, así como en el Reto 3 del curso.

Análisis de complejidad

El requerimiento se compone de 4 funciones:

1. homonymsREQ3(): Esta función se encarga de mostrar las ciudades con el mismo nombre cuando haya ambigüedad en las entradas dadas por el usuario. El orden de crecimiento de esta función no se tiene en cuenta para el requerimiento, puesto que se entiende como un paso previo a su ejecución.
2. calculateRangeREQ3(): Esta función realiza únicamente operaciones aritméticas, pues solo se encarga de expandir el rango de búsqueda de aeropuertos. Así, su orden de crecimiento es constante.
3. findNearestAirportREQ3(): El primer ciclo que realiza esta función se ejecuta, a lo sumo, **x** veces. Dentro de este ciclo, se implementa el método values() del TAD de árboles, el cual realiza log(2a) operaciones en el peor caso. Luego, se implementa otro ciclo de **a** recorridos, dentro del cual se ejecuta otro values() con log(2b) recorridos y un último ciclo de **b** recorridos. En total, la complejidad de esta función es **O(x*(log(2a) + a*(log(2b) + b))**, que se simplifica a **O(x*a*b)**. Nótese que a,b,c siempre serán significativamente menores que E y V.
4. REQ3(): Esta función ejecuta el algoritmo Dijkstra, el cual tiene complejidad de **E*log(V)**. Luego, se invoca la función pathTo() de dijkstra, la cual verifica el camino del vértice de origen a otro vértice en la búsqueda anteriormente realizada. Este último tiene complejidad de **V**, pues en la implementación del TAD se verifica cada vértice encontrado en la búsqueda hasta dar con el vértice de origen. En total, la complejidad de esta función es de **O(E*log(V)).**

Finalmente, comparando las complejidades anteriormente calculadas, se concluye que el Requerimiento 3 tiene complejidad de **O(E*log(V))**.

Requerimiento 4

Preámbulo

Entiéndase: **E** = número de rutas en el dígrafo; **V** = número de aeropuertos en el dígrafo.

Análisis de complejidad

Las operaciones que se realizan en este requerimiento son las siguientes:

1. Método PrimMST: complejidad E*log(V) según lo enseñado en clase.
2. Método weightMST: complejidad E + V, dado que, primero, invoca la función edgesMST, la cual realiza un ciclo de V recorridos, y segundo, realiza un ciclo de E recorridos.
3. Se realiza un ciclo de E recorridos al MST para encontrar los aeropuertos que lo componen.
4. Se implementa el método keySet() a un mapa de tamaño E, lo cual representa una complejidad de E.

La complejidad sería, en total, O(E*log(V) + 3E + V), lo cual se resume a **E*log(V)**

Requerimiento 5

Preámbulo

Entiéndase: **E1** = número de rutas en el dígrafo; **E2** = número de rutas en el grafo no conectado; **V** = número de aeropuertos.

En la carga de datos, se creó el grafo “ReversedMainGraphReq5”, el cual, como su nombre lo dice, es el grafo inverso al dígrafo (las rutas llevan sentido contrario).

Análisis de complejidad

Para este requerimiento, inicialmente se invoca dos veces el método adjacents() de los grafos, el cual tiene complejidad **E** puesto que, en el caso de un grafo completamente conectado, se evalúan todos sus ejes como adyacentes al vértice en cuestión. Luego, se implementan los métodos indegree() y outdegree(), ambos con orden de crecimiento constante según su implementación en el TAD. Más adelante, se invoca la función getUnrepeatedREQ5(), la cual revisa todos los aeropuertos conectados (con rutas hacia y desde) al aeropuerto que se desea eliminar, y devuelve una lista con el total de aeropuertos afectados sin repeticiones. En esta función, se realizan **2V** recorridos en el peor caso. Por lo tanto, la complejidad del Requerimiento 5, hasta aquí, sería de **O(E + V).**

Lo último que se realiza en este requerimiento es eliminar el vértice del dígrafo y del grafo no conectado. Este se compone de dos funciones:

1. removeEdgeREQ5(): esta función elimina un arco del grafo. Primero, implementa un ciclo de **E** recorridos en el peor caso, evaluando cada arco que sale de determinado vértice. Luego, dentro de este ciclo, se implementa el método deleteElement() del TAD de lista, el cual, en el peor caso, también tiene complejidad de **E**. Por lo tanto, esta función tendría complejidad de **E^2**. No obstante, dada la naturaleza de nuestros datos, el orden de crecimiento de esta

función será mucho menor dado que los aeropuertos tienen, a lo sumo, un número máximo de arcos mucho menor a **E.**

2. removeVertexREQ5(): esta función remove() del TAD de mapa, el cual tiene orden de crecimiento constante dada su implementación, y luego un ciclo de **V** recorridos en el peor caso. Finalmente, implementa otro ciclo de **V** recorridos, dentro del cual se invoca dos veces la función removeEdgeREQ5(), por lo que se tendría una complejidad de **O (V*E^2).**

En resumen, la complejidad del Requerimiento 5 será de **O(V*E^2)** en el peor caso. No obstante, en la práctica, el requerimiento funcionará más rápido dada la naturaleza de los datos.