Estructuras de Datos y Algoritmos – Reto 4 – Grupo 13 Sección 2
Análisis de complejidad de los requerimientos

Juan Sebastián Núñez Cortes, 202021672, j.nunezc@uniandes.edu.co
Santiago Rodríguez Bernal, 202011182, s.rodriguez63@uniandes.edu.co

A continuación, se presenta un análisis de complejidad para cada requerimiento del Reto 4, considerando el código utilizado y desarrollado en este último proyecto. Para este, se representan los vértices con la letra V, y los arcos con la letra E.

Carga de datos: En la carga de datos se crean 3 grafos, 3 tablas de hash, y un árbol RBT, esto para poder garantizar el desarrollo de los requerimientos y reducir la complejidad al momento de trabajar con las pruebas. Los grafos son: un dígrafo con todas las rutas aéreas, un grafo no dirigido que incluye aeropuertos conectados con rutas tanto de ida como de vuelta, y un grafo de ciudades que conecta a cada ciudad del archivo CSV con su aeropuerto mas cercano. 2 de las tablas de hash son usadas para guardar información de los aeropuertos y las ciudades, mientras que la otra restante se emplea para poder almacenar las ciudades homónimas que el usuario distingue por consola. Finalmente, el RBT se usa para calcular los aeropuertos más cercanos a una ciudad, y de esta manera reducir la complejidad de los requerimientos que buscan saber cual es el aeropuerto más cercano a una ciudad dada.

Requerimiento 1: Para el primer requerimiento, primero se obtiene el número total de vértices en el grafo, lo cual es una consulta de orden constante. A continuación, se llama a la función “interconnection” del controlador, la cual se ocupa de encontrar los aeropuertos que sirven como puntos de interconexión aérea. Para esta única función, primero se debe obtener los vértices del grafo, y dado que el API del grafo obtiene un keySet del mapa empleado, este proceso se realiza V veces para crear una lista, por lo cual esta primera complejidad es de O(V). A continuación, se itera sobre todos los elementos de la lista de vértices. Las operaciones realizadas principalmente son consultas de orden constante que se efectúan V veces sobre la iteración de la lista, por lo cual se logra una complejidad de O(V). Adicionalmente, la lista obtenida en paso anterior debe organizarse para poder presentar los cinco aeropuertos con más interconexiones. Dado que se utiliza mergesort para realizar el ordenamiento, la complejidad es de O(VLog(V)). En este caso vuelve a hacer V, dado que en el peor caso todos los vértices tienen un grado de salida o, de entrada. Al comparar las tres grandes complejidades de este requerimiento O(V) + O(V) + O(VLog(V)), se concluye que la complejidad del primer requerimiento es de O(VLog(V)).

Requerimiento 2: Para el segundo requerimiento, se emplea la función “findSCC”. Dicha función implementa únicamente el algoritmo de Kosaraju. Este algoritmo debe primero calcular el grafo reverso del original. Dicho proceso, con V vértices y E arcos, corresponde a una complejidad de O(V^2). Esta complejidad se debe a que en el peor caso cada vértice

posee un arco a todos los demás vértices del grafo. Ahora bien, por otro lado, el algoritmo realiza un recorrido en DFO, cuya complejidad es de O(V+E). Finalmente, dado por terminado el algoritmo, se busca conocer si dos aeropuertos están fuertemente conectados. Al revisar la implementación del algoritmo dentro de la librería DISC, para conocer a que componente conectado hace parte cada aeropuerto, se realiza una consulta con la función map.get. Tomando en cuenta que un mapa las consultas realizadas tienen una complejidad de O(1.5), la complejidad de este requerimiento corresponde a O(V^2) + O(V+E) + O(1.5). En síntesis, se tiene una complejidad de O(V^2).

Requerimiento 3: Para el tercer requerimiento, el primer paso es mostrarle al usuario las ciudades homónimas de las que este indica por consola. Dado que se consulta una tabla de hash con la información ya existente, obtener las listas que corresponden a las ciudades homónimas tiene un orden de complejidad constante. Sin embargo, estas listas deben ser impresas en consola, y para cada lista, se itera la cantidad de ciudades dentro de la lista para poder presentarlas en una tabla. De manera que la complejidad es de O(n) para la primera lista, y O(m) para la segunda, siendo n y m dos cantidades diferentes, pues corresponden a las ciudades homónimas de dos lugares distintos. A continuación, se realiza el recorrido de Dijkstra con la función "dijsktraCity". Primero, se obtienen los aeropuertos mas cercanos a las dos ciudades dadas por parámetro, y dado que estos corresponden al único elemento dentro de una lista de vértices adyacentes a las ciudades, la complejidad es constante. Luego se realiza el recorrido de Dijkstra sobre el grafo, por lo cual en el peor caso se tiene una complejidad de O(E Log(V)). Finalmente, se obtiene una pila con la función "pathTo" del API del grafo. Para la creación de la pila, dado que se agrega un elemento en el tope, y considerando que la pila es una lista enlazada predeterminada, la complejidad de la operación push corresponde a O(1). No obstante, dado que push se realiza en el peor de los casos V veces, la iteración tiene una complejidad de O(V). También este es el caso para cuando se hace pop de la pila. Al revisar las complejidades obtenidas O(n), O(m), O(E Log(V)), O(V) y O(V), se concluye que la complejidad del requerimiento corresponde a O(E Log(V)).

Requerimiento 4: Para el cuarto requerimiento, al igual que en el tercero, se debe obtener una ciudad de una lista de ciudades homónimas. Nuevamente se realiza una consulta de orden constante, tanto en el mapa como en la lista de tipo arreglo. Sin embargo, considerando que en este caso solo se imprime una lista, la complejidad de esta primera parte del algoritmo corresponde a O(n). Posteriormente, se llama a la función "travelerMST" para responder a las entradas requeridas por el requerimiento. Las primeras líneas de código de esta función, después de la descripción, corresponden a un orden de complejidad constante. Luego se obtiene el MST con el algoritmo de Prim implementado en la librería DISC. Tomando en cuenta la implementación de Prim, y que se utiliza una lista de adyacencias como estructura de datos, se tiene una complejidad de O(E Log(V)). Luego, se realiza un recorrido en DFS del grafo no dirigido, por lo cual la complejidad es de O(V+E). Con los vértices visitados en dicho recorrido, se itera sobre estos para encontrar el camino mas largo de un vértice de

origen al resto. En primer lugar, esta iteración se realiza V veces en el peor caso, considerando que hay un camino del vértice de origen a todos los demás vértices del grafo. Para cada vértice, se obtiene el camino con la función "pathTo", nuevamente, en el peor caso se consideran V vértices. De este modo, se infiere que el ciclo que calcula la ruta más larga, dado un vértice de origen, tiene una complejidad de O(V^2). Finalmente, tras pasar por algunas operaciones de orden constante, se calcula el numero de aeropuertos que hacen parte del árbol de recubrimiento en el grafo. Considerando que el proceso se realiza para cada nodo, se tiene una iteración que en el peor caso equivale a O(V). En definitiva, al analizar las complejidades obtenidas O(n) + O(E Log(V)) + O(V+E) + O(V^2) + O(V), se concluye que la complejidad del requerimiento cuatro corresponde a O(V^2).

Requerimiento 5: Para el quinto requerimiento, se utiliza una única función denominada "affectedAirports". Esta función realiza, en su mayoría, operaciones de orden constante, pues son restas que calculan el efecto que tendría el cerrar un aeropuerto, y llamados a variables y estructuras que no requieren revisar todos los elementos. De las principales funciones utilizadas, se tiene "totalVertices", la cual obtiene el número de vértices de ambos grafos. Tomando en cuenta que el API devuelve el tamaño de un mapa, esta función tiene un orden de complejidad constante. Asimismo, la función "totalRoutes", que devuelve el numero total de arcos, también tiene una complejidad constante, pues consulta un valor que ya se encuentra guardado. Por último, se busca obtener los vértices adyacentes a un aeropuerto para poder calcular cuántos aeropuertos se verían afectados. Dado que la función del API añade a una lista todos los vértices adyacentes a otro, esta complejidad corresponde a O(V) en el peor caso. Dado que esta función se llama dos veces para los dos grafos (dirigido y no dirigido, con el mismo número de vértices), se tiene una complejidad de O(V) + O(V). Como se mencionó antes, el resto de las operaciones (cálculo del tamaño de una lista, restas de dos variables, entre otros) tienen un orden de complejidad constante. Así pues, la complejidad de este quinto requerimiento corresponde a O(V).

Requerimiento 6: Para el sexto requerimiento la complejidad asociada esta directamente relacionada a la API, en este caso al ser un server en vivo, que requiere manejar varias solicitudes podemos decir que la complejidad seria de O(k), pues depende meramente de cuantas solicitudes pueda dar solución la API por segundo.

Requerimiento 7: El requerimiento 6 al ser una representación gráfica del requerimiento 1 y 3, comparte su misma complejidad. Sin embargo, se es necesario agregar la complejidad que implica todo el proceso de creación del mapa y markers con la información. Para el proceso de creación se crearon listas e iteraron por lo tanto su complejidad es de O(n), y a medida que se iteraban se iba creando cada marker, donde no solo se marcaba el mapa, sino que adicionalmente cada punto en el mapa tiene la información sobre esta, por lo tanto su complejidad según lo encontrado sobre folium es O(n ^2).