Jaime Eduardo López Castro - A00833173

Reflexión:

La estructura de datos en la que se basó esta entrega fue en las HashTables, en nuestro caso específico, el uso de una HashTable de dirección abierta con prueba cuadrática, lo que significa que no utiliza memoria externa para tratar con las colisiones. Gracias a la función Hash implementada, añadir elementos tiene una complejidad de O(1), así como también buscarlos después, algo que no habíamos conseguido desde que empezó el curso y una gran mejora por sobre cualquier otra estructura de datos. Esta estructura funciona gracias a una llave numérica, que es la que entra a la función Hash, gracias a que utilizamos la IP numérica como llave, podemos almacenar en el nodo de la tabla el resto de datos de la IP, como su representación en string, conexiones de entrada y salida, entre otras cosas. En cuanto al tamaño inicial de esta estructura, decidimos utilizar un valor de 26759, que es un número primo que representa un poco más del doble de elementos que contamos en nuestra bitácora, nos dimos cuenta que utilizando este método logramos reducir las colisiones a 2,445; a diferencia de cuando utilizamos los espacios justos que obtuvimos 307,230 colisiones.

La complejidad final de nuestro programa fue de O(n * V) (el número de líneas en el txt por el número de vértices del grafo); esto es gracias a que leer el txt y crear un grafo en base a él tiene esta complejidad, gracias a que tenemos que crear un vector de listas enlazadas para representar nuestra lista de adyacencia, posteriormente tenemos que crear una lista enlazada vacía y añadirla a nuestro vector de adyacencias por cada nodo, terminando con leer cada uno de los enlaces y ponerlo en el nuestra lista de adyacencias.

Justamente gracias a que añadir elementos a nuestra HashTable tiene un tiempo en promedio constante, poblar nuestra tabla tiene una complejidad de O(n), ya que tenemos que poner elemento por elemento. Ya una vez cargada nuestra tabla en cuando empieza la magia, buscar elementos y obtener sus datos es en promedio O(1), lo único que nos toma tiempo es imprimir las conexiones de esta ip específicamente de forma ascendente, ya que para esto creamos un Heap, lo llenamos de todas las conexiones de salida de nuestra ip y realizamos un HeapSort para imprimirlas en la pantalla, tomándonos un tiempo de O(nlogn).

Intentar realizar esta evidencia con alguna otra estructura que no fuera una HashTable tendría consecuencias en su tiempo de ejecución, un vector por ejemplo tardaría un tiempo O(n) en buscar un elemento si están desordenados y si quisiéramos ordenarlos para realiza un algoritmo más eficiente, este no sería mejor a O(nlogn). Una estructura del estilo de un BST es más rápida que la búsqueda lineal, pero sigue sin ser constante, por lo que utilizar una HashTable es sin duda nuestra mejor opción aquí.

Fuentes:

- Hash table runtime complexity (insert, search and delete). (2012, 9 febrero). Stack Overflow. https://stackoverflow.com/questions/9214353/hash-table-runtime-complexity-insert-search-and-delete#:%7E:text=Hash%20tables%20suffer%20from%20O,each%20element%20to%20the%20table%5D.

- GeeksforGeeks. (2022e, julio 17). Hashing | Set 1 (Introduction). https://www.geeksforgeeks.org/hashing-set-1-introduction/

- GeeksforGeeks. (s. f.). Hashing Data Structure. https://www.geeksforgeeks.org/hashing-data-structure/