Jorge Del Barco Garza
A01284234

25 de Noviembre 2021

Reflexión Final: Programación de estructuras de datos y algoritmos fundamentales

Act 1.3

Después de haber realizado la actividad y de habernos enfrentado a diferentes retos como lo fue comprender en su totalidad los algoritmos que estamos utilizando, su complejidad, y la manera de implementarlos utilizando los vectores en C++, se llegó a un aprendizaje valioso: elegir el algoritmo correcto es importante y conocer sobre la teoría de ellos también.

Un algoritmo de ordenamiento es un programa que nos permite ordenar información de una manera específica. Para lo que hemos visto en clase, nos permite ordenar valores numéricos de forma ascendente o descendente. El motivo de utilizar estos algoritmos es ordenar los datos con los que se trabaja para habilitar un acceso a la información más rápido y posiblemente hasta seguro. Por ejemplo, si realizamos cambios a un arreglo que contiene información en el lugar equivocado, estaríamos dañando nuestra base de datos. Por lo que tener organizada información es importante sea cual sea el área o trabajo.

Aplicado al desarrollo del proyecto entregado, al inicio, se quiso utilizar el algoritmo Quicksort para ordenar por fecha los más de 16 mil elementos que se tenían que clasificar. Sin embargo, este demostró ser muy lento, pues tiene una complejidad de O(n^2). Después de seguir buscando un algoritmo más eficiente de los que hemos visto en clase nos topamos con merge sort. Este es un algoritmo recursivo que tiene una complejidad de O(n logn). Al implementarlo, la superioridad del algoritmo con este volumen de datos se vio de manera casi inmediata. Con Quicksort nunca alcanzamos a

Act 2.3

En esta actividad se juntaron distintos elementos de la programación relacionados al área de estructuras de datos y algoritmos para cumplir con la actividad. Comenzando con la implementación de listas doblemente ligadas, una estructura de datos mejor conocida por su nombre en inglés como Doubly Linked List. Este tipo de listas son un conjunto de objetos llamados nodos, que están conectados entre sí, por delante y por detrás a otros nodos a través de apuntadores en C++. Cada uno de los nodos alberga un cierto tipo de dato, desde un int o string hasta objetos complejos.

Fue sencillo implementar una referencia (apuntador) dentro de la clase nodo.h que apunte al nodo anterior, pues seguía la misma lógica que con las listas enlazadas simples. Una vez la clase nodo está lista, se comenzó a escribir la clase Double Linked List (DLL) para así poder implementar funcionalidades con nodos, y agregar características y métodos propios de la estructura de datos. Fueron todos relativamente sencillos de desarrollar. El verdadero reto comenzó cuando debíamos acoplar a nuestro problema particular la estructura de datos. Estábamos trabajando con una serie de datos de tipo “string”. Y en realidad lo que nos piden es en una DLL, almacenar todos esos datos, y así proceder a sortearlos por un valor numérico que tienen dentro las cadenas de caracteres. Para la inserción de los datos tipo string, obtenidos de la lectura de un archivo, se usó un algoritmo de inserción con complejidad O(1), reemplazando al utilizado en un inicio que era insertar cada nuevo nodo al último de la lista con complejidad O(n).

Una vez se tuvieron los datos desordenados en la DLL, procedimos a idear una forma para identificar un valor numérico entre la cadena de texto. Se diseñó una función que dado el input de texto que crea un nuevo objeto de tipo nodo, el constructor correrá esta función llamada “sortingValue()” que devolverá un valor de punto flotante que se guardará dentro del nodo que contiene el string. Esto nos permite acceder al valor que será sorteado dentro de una función y poder organizar todo dentro de la lista sin necesidad de realizar una copia a una lista vacía o algo por el estilo. El algoritmo de ordenamiento utilizado para organizar las IPs fue Insertion Sort con complejidad O(n^2).

Decidimos utilizar este algoritmo porque fue el que era más fácil de implementar nuestro programa, sin embargo, no es el más eficiente. Ese sería Merge Sort. Una vez sorteada la lista y guardados los elementos en un archivo, se pide al usuario un intervalo de IPs que quiere guardar en un nuevo archivo y utilizando el algoritmo de búsqueda secuencial con complejidad de O(n) se analiza el archivo y se depositan los logs que pidió el usuario en su .txt final.

Act 3.4

Desde mi punto de vista personal, creo que el uso de un árbol binario de búsqueda no queda para esta situación, puesto que tenemos varios valores repetidos y se rompe la propiedad de un BST. Además, hace que el acceso a estos valores repetidos sea difícil de lograr dentro del mismo BST, pues una vez ya insertados esos valores hay que tomar la decisión de ponerlos a la izquierda o la derecha. Esta decisión se tomó para nuestro caso con base a cuales datos serán insertados más adelante, si mayores o menores en promedio.

Un BST es muy eficiente para determinar organizar valores de manera rápida y eficiente. Pues la misma estructura de un BST, a la hora de hacer una inserción, organiza los elementos de una manera en la que compara qué valor es mayor o menor que otro.

Para esta situación donde se tienen valores repetidos, y había que accesar a los valores más grandes, repetidamente, se recomienda utilizar un max heap. Pues acceder a los valores más grandes del conjunto siempre tendrá una complejidad de $O(1)$ pues el elemento más grande siempre estará en la raíz del árbol, mientras que en un BST normal tiene una complejidad de $O(n)$ y su ubicación solo se determina por el valor de su número y por cómo está organizado el árbol. A pesar de que la propiedad de un BST se rompe en esta situación, la inserción sigue siendo $O(n)$, y la verificación de la altura del árbol es $O(h)$.

Para nuestro programa, decidimos optar por hacer un preprocesamiento de los datos y capturar cuantas veces se están repitiendo las IPs para almacenarlas en un vector de objetos de tipo entry. Estos objetos entry almacenan el número de accesos a la IP (int) y la IP misma en formato string. Seguido de esto, se insertan en el BST de manera secuencial como se encuentran en el vector siguiendo como mejor se puede las propiedades del mismo. Una vez el árbol está completo, se traversa el árbol en orden, comenzando con los números más grandes y guardándolos todos en otro vector. Una vez se encuentren todos los elementos del árbol en el vector, se imprimen los primeros 5 elementos de este para obtener así las primeras 5 IPs con el mayor número de accesos.

Esto se hizo de esta manera porque nos resultaba sencillo y al mismo tiempo seguíamos utilizando el BST en su totalidad para resolver el problema.

¿Cómo podemos determinar si una red está infectada o no?

En este caso se nos brinda ya una bitácora con los accesos que se hicieron a una serie de IPs, que técnicamente a esto se le conoce como logs. En este caso los logs arrojan un mensaje donde se especifica qué sucedió exactamente cuando se intentó hacerle un

ping a esa IP. Podemos determinar con la información que nos dan que, cuando el mensaje del log dice “Illegal user”, entonces ese ping realizado a esa hora en esa fecha se consideraría un intento de acceso malicioso.

Según mi experiencia, los pings a una IP son repetitivos, pues cada que se envía un paquete con datos a una dirección, se interactúa con la IP. Por lo que por la cantidad de pings que se realizan no se puede determinar si una red está infectada o no. Sin embargo, si se está intentando acceder a los distintos dispositivos dentro de la red y no se logra entrar porque se tienen las credenciales incorrectas por una cantidad de pings anormal, ahí podría considerarse que dentro de la red ya hay alguien, la red está ahora infectada, que intenta sin permiso entrar a algún dispositivo.

Act 4.3

En esta actividad, utilizamos grafos dirigidos para representar una red de computadoras interconectadas por medio del protocolo IP/TCP. Este grupo de dispositivos parece que está sufriendo un intento de ataque de tipo “brute force” desde algún dispositivo ya infectado dentro de la red. Esto se puede justificar por los mensajes de error en los logs de la red que muestran intentos fallidos de obtener acceso a otros dispositivos. El objetivo de esta actividad era encontrar al “bootmaster”, o a las posibles computadoras que intenten conectarse a la mayor cantidad de dispositivos posibles e identificarlas con una IP.

El uso de los grafos para representar redes es muy útil y es una estructura de datos en lo personal interesante para lo que es la ciberseguridad y las redes que son temas que me interesan. El hecho de ser nosotros capaces de atravesar una red de una manera ordenada y conocer los detalles de cada nodo resulta favorable. Aprendí que se pueden realizar ciertos recorridos como el Breadth-First Search o el Depth-First Search y que ambos tienen aplicaciones distintas, como podría se encontrar el camino más corto entre un nodo u otro (BFS) o para detectar si en una red existe una conexión entre ciertos nodos (DFS). Ambos ejemplos son importantes y aplicables a lo que se vio aquí que fue asegurar la integridad de una red de computadoras. Por ejemplo, si de una computadora personal de un empleado se puede llegar a un servidor operativo, ¿cuál es la ruta más corta? O ¿Existe una conexión entre ambos dispositivos a través de la red? Esto es algo que nos debemos preguntar cuando se evalúa la seguridad de una red.

Para esta actividad se eligió la representación de un grafo en el programa por medio de una lista de adyacencia. Esta se programó explícitamente con un vector de vectores de nodos (<vector<vector<Node>>>), contemplando que se iba a requerir de un acceso directo a los nodos adyacentes de cada vértice en la red, y que las operaciones de lectura y de inserción a un vector son de $O(1)$ si se realiza un *append()* o *push_back()* a la estructura.

Se optó por la lista sobre una matriz de adyacencia con el fin de optimizar el uso de memoria del programa y disminuir el nivel de complejidad computacional en espacio. Otra vía que se tomó para optimizar el programa, fue utilizar el algoritmo binary search de complejidad $O(logn)$ para acomodar las IPs en su respectivo lugar de la matríz de adyacencia y hacer un *append()* para conectar el nodo de origen con su nodo de destino y así para todas las IPs y sus nodos conectados.

Act 5.2

En esta actividad se implementó la estructura de datos *unordered_map* de la STL de C++ que funciona como una hash table. Una hash table es una estructura que asocia llaves únicas o “keys” de cierto tipo de dato, que asocian a otro tipo de dato o a un objeto, u otra estructura como fue el caso en esta actividad 5.2. Las llaves de un *unordered_map* no pueden repetirse, es decir, deben ser únicas en la instancia de la estructura. Esta es una de las características que hacen que este objeto sea especial.

Algo particular que hace que esta estructura destaque, es que no están ordenados los valores que se insertan a la hash table, y que los métodos básicos como lo son lectura, búsqueda y eliminación son *O(1).* Las hash tables reciben su nombre por la función hash con la que opera la tabla. Una función hash es un algoritmo que, dado un input, con simples operaciones aritméticas determinará una posición en la tabla que apunta a un espacio desocupado y ahí insertará, buscará, leerá o eliminará los datos. Dado un input, se espera siempre un mismo output (esto puede cambiar dependiendo del nivel de complejidad de la función hash), por lo que el uso de la llave correcta para acceder al valor esperado es de suma importancia.

Para esta actividad, se creó un *unordered_map* que reciba de key un string, que en este caso es una IP recuperada de los logs de la bitácora. El tipo de dato al que permite el acceso una llave es uno de tipo *IpAddress,* donde se guardan los datos de cada log de la bitácora como atributos. Los atributos son: fecha, hora, IP, mensaje del log y una variable que cuenta la cantidad de veces que una IP se repitió en todo el archivo de la bitácora. Recordemos que no existen llaves repetidas en un hash table como la que se creó aquí, por lo que, si un valor de llave vuelve a aparecer, se incrementa la variable contador en 1 del objeto *IpAddress*.

Reflexión Final

Durante todo el semestre se han estudiado la composición, complejidad computacional ventajas, desventajas y casos de uso de diversas estructuras de datos básicas en el lenguaje de programación C++. Como parte del contenido del curso, se realizaron diversas actividades que consistían en analizar una bitácora con registros de una red IP/TCP con el objetivo de almacenar los datos de dicho material en una estructura de datos conforme se iban revisando en clase. La actividad se realizó cinco veces utilizando una estructura de datos distinta en cada instancia. Vectores, Listas doblemente ligadas, Árboles binarios de búsqueda, Grafos, y Hash Tables son las estructuras que mayor relevancia tuvieron en mi aprendizaje desde mi punto de vista personal. Para cada una de ellas hay detrás un código fuente distinto y hecho desde cero a excepción de la tabla Hash.

Dado un mismo input y esperando el mismo resultado (guardar exitosamente la información en una estructura de datos dada) para cada actividad; la implementación a código, y la complejidad computacional de cada estructura varía enormemente. Es ahí donde un conocimiento profundo sobre las características, cualidades y desventajas de cada ADT facilitan cumplir con el objetivo que se ha propuesto y definen a un buen ingeniero de otro, pues desarrollar una solución funcional, eficiente y robusta es lo que se pide en el mundo real.

¿Cuáles son las estructuras más eficientes? Esta pregunta varía dependiendo del contexto y del problema que se quiere resolver. Para el caso del análisis de registros de red, en tiempo de desarrollo, complejidad computacional y fidelidad de la solución, las tablas hash fueron las más eficientes de todas las estructuras estudiadas. Hay tres razones por las cuales considero lo anterior.

Primero, el tiempo que tomó desarrollar una lógica conceptual utilizando la funcionalidad del hash table fue muchísimo menor al del resto de las estructuras de datos utilizadas. Esto permitió que la transición a implementación en código ocurriera antes de lo previsto.

Segundo, por lo mismo que se tenía un previo conocimiento teórico sobre cómo opera un hash table, específicamente su función hash y los métodos miembros de la clase, se decidió que era muy eficiente usar esta ADT. ¿Por qué exactamente? Las operaciones que se utilizaron para resolver la actividad tenían una complejidad de tiempo constante, es decir $O(1)$ en promedio, siendo rara vez el peor caso $O(n)$.

La tercera razón por la cual la hash table fue la más eficiente de todas las estructuras de datos estudiadas fue la fidelidad de la solución que se entregó. La hash table utilizada,

por propiedad, no podía tener valores repetidos, cosa que ahorraba trabajo pues no se debía de procesar de manera específica un input que no era válido para la ADT. Esto permitió que toda la atención del desarrollo se fuera exclusivamente a lo que se necesitaba para cumplir con la actividad, creando así una solución que no puede fallar para el tipo de input que se espera lea el programa.

¿Cuáles estructuras de datos podrías mejorar y cómo harías esta mejora? Dentro de la lista de estructuras de datos revisadas en el curso, hay algunas que por naturaleza simplemente no están hechas para procesar rápidamente información del tipo que se brindó, como es el caso de los vectores y listas doblemente ligadas. No obstante, existen otras que pueden mejorar y variar un poco su uso. Por ejemplo, los Árboles Binarios de Búsqueda que se utilizaron para el input que se dio rompía con las propiedades de la misma estructura, que no se pueden tener valores repetidos, y que se debía obtener la IP con el mayor número de accesos. Mi mejora en este caso está en utilizar un max-heap. Una variante de un árbol binario pero que tiene siempre guarda en su raíz el elemento de mayor valor de un conjunto de datos, y que dada una inserción, si algo se encuentra fuera de lugar, un algoritmo reacomodará los nodos del árbol para volver a su estado original, que el elemento de mayor valor se encuentre en la raíz y que cada padre del árbol sea de mayor valor a sus hijos. Esa sería la mejora que le haría a esa estructura de datos para el contexto que se tiene.