Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

Programación de estructuras de datos y algoritmos fundamentales – TC1031

Act 6.2 - Reflexión Final de Actividades Integradoras de la Unidad de Formación TC1031 (Evidencia Competencia)

Grupo 14

Adrián Alejandro Salgado Martínez – A00828843

2 de diciembre de 2020

Tabla de contenidos

Introducción

A lo largo del curso, se llevaron a cabo diferentes actividades con el objetivo de practicar sobre los temas vistos en clase, aplicándolos para resolver ciertas situaciones problema y para el desarrollo de las distintas subcompetencias que competen al curso.

Siendo así, este documento tiene la función de servir como compendio de todas las reflexiones que se realizaron junto a cada actividad integradora, ordenadas de manera cronológica según se desempeñaron en el curso.

I. Algoritmos fundamentales

Los algoritmos de búsqueda y ordenamiento, tales como los que fueron utilizados en el desarrollo de dicha situación problema, son de suma importancia ya que se presentan como una de las bases fundamentales del área de la computación. A través de estos, es posible realizar la manipulación de datos para lo cual es tan útil el uso de la informática, puesto a que nos permite estructurar y desglosar grandes cantidades de información que de hacerse de manera manual sería no solo impráctico, sino también inviable.

Asimismo, un aspecto crucial que diferencia a los algoritmos que existen es su eficiencia, pues a pesar de los múltiples avances que han habido en la computación la realidad es que existen limitaciones de tiempo y de potencia, combinado con el hecho que en ocasiones como en esta situación problema la cantidad de datos con los que se trabaja es inmensa, lo que significa que elegir o diseñar e implementar un buen algoritmo de búsqueda o de ordenamiento trae un gran impacto en la administración de los recursos limitados a nuestra disposición.

Es por esto mismo que para esta actividad se decidió utilizar los siguientes algoritmos de búsqueda y ordenamiento disponibles en la librería estándar de C++:

- `std::sort()` - Este algoritmo de ordenamiento, disponible en el encabezado `<algorithm>`, organiza los elementos en orden ascendente. Su especificación indica que tiene una complejidad promedio de O(n log n). La forma en la que está implementado es por medio del algoritmo Introsort, el cual es una implementación híbrida que combina el uso de algoritmos increíblemente eficientes como lo son Quicksort, Heapsort e Insertionsort, aprovechando sus fortalezas y compensando sus debilidades, obteniendo así una alta eficiencia promedio en comparación a si se utilizara solo uno de ellos.
- `std::lower_bound()` y `std::upper_bound()` - Estos algoritmos de búsqueda, también disponibles en el encabezado `<algorithm>`, encuentran el apuntador a un

elemento que es el límite inferior o el anterior al superior comparado a un valor dado en un vector ordenado. Su especificación indica que tienen una complejidad de O(log n). La manera en la que están implementados, como es evidente con su complejidad, es a través de búsqueda binaria modificada.

La ventaja que tuvo usar estos algoritmos internos, en lugar de implementar uno por nuestra cuenta, es que están diseñados para ser increíblemente eficientes y seguros. Sin embargo, una desventaja importante es que existe mucha limitación respecto a cómo podemos modificarlos para necesidades específicas, aunque no haya sido tan impactante gracias a la sobrecarga de operadores.

II. Estructuras de datos lineales

Las listas doblemente encadenadas, tal como la que fue utilizada para el desarrollo de esta actividad, representan una de las estructuras de datos más importantes para el almacenamiento de información computacional. Lo que las diferencia de una estructura de datos más simple como es un arreglo es que estas están compuestas de distintos “nodos”, donde además de el tipo de dato almacenado también se encuentran apuntadores a las direcciones de memoria del siguiente nodo, del nodo anterior (solo en las encadenadas dobles), o nullptr si es que se trata del primer o el último nodo. De esta manera se logra conseguir que agregar un elemento a la lista se pueda llevar a cabo de manera constante cuando se tiene acceso a los nodos relevantes, como es el caso del inicio o del final de la lista, y la única limitación para la extensión máxima de esta es la cantidad de memoria total disponible en la computadora.

Sin embargo, una de las más grandes limitaciones de este tipo de estructuras de datos es que obtener acceso a un elemento en un índice determinado no se puede llevar a cabo de manera trivial, como es el caso con un vector o un arreglo, puesto a que se tiene que visitar cada uno de los nodos necesarios para saber cual es el siguiente, ya que la organización en la memoria de estos no es contigua y no se puede llevar a cabo de manera aleatoria, esto significa que para listas considerablemente largas leer y escribir en un índice medio se vuelve realmente costoso, especialmente si no se utilizan algoritmos eficientemente diseñados con estas limitaciones. Es por esto mismo que para esta actividad se decidió implementar una clase iterador además de la clase de la lista, consiguiendo así utilizar los siguientes algoritmos de búsqueda y ordenamiento:

- `DLinkedList<T>::sort()` - El algoritmo que se implementó para ordenar la lista está prácticamente compuesto en su totalidad por el algoritmo utilizado para la lista de la implementación de GCC de STL. Este, como lo dicta la especificación de C++ para cualquier sort de una lista, tiene una complejidad de O(n log n), lo cual es bastante inusual para un contenedor de dicho tipo. Se utiliza en lugar de `std::sort()` ya que este último requiere de iteradores de acceso aleatorio para mantener esta eficiencia, en contraste con los iteradores bidireccionales implementados.

- `std::lower_bound()` - Este algoritmo de búsqueda, disponible en el encabezado `<algorithm>`, encuentra el iterador correspondiente al primer elemento que no es menor a un valor dado en un contenedor ordenado. Su especificación indica que normalmente tiene una complejidad de O(log n), no obstante, al tratarse de iteradores bidireccionales y no aleatorios significa que este tendrá realmente complejidad de O(n).

Se concluye entonces que para esta situación problema el uso de una lista doblemente encadenada es bastante útil por la gran cantidad de datos con los que se trabajan, que permiten inserción y eliminación constante, sin embargo no se piensa que es la mejor que pudo haber utilizada puesto a las limitantes al ordenarla y realizar búsquedas.

III. Estructuras de datos jerárquicas

Los Binary Search Trees (BST), también conocidos como árboles de búsqueda binaria, son un tipo de estructura de datos que poseen características estructurales y funcionalidad que es de gran utilidad para algunas situaciones donde se requiere de una búsqueda eficiente dentro de sus datos, puesto a que por definición por cada nodo del árbol existe un subárbol donde todos los elementos a la izquierda son menores y a la derecha son mayores, logrando así una más rápida búsqueda al descartar la mitad de los datos en cada iteración para un árbol bien balanceado, es decir, uno donde la diferencia de altura de los subárboles no sea mayor a 1.

Con esto en mente, para darle solución a la situación problema planteada que consiste en encontrar las IPs con un mayor número de accesos por medio de un archivo de texto, se implementó una clase contenedora con el número de accesos y la IP, además de un BST y un iterador constante (para solo leer la información) y reverso (para recorrer de mayor a menor) donde los principales algoritmos utilizados fueron los siguientes:

- `BST<Registro>::add(Registro)` - Este método del árbol es el encargado de añadir al mismo el nuevo elemento que se desea guardar. Esto lo hace comparando al dato nuevo que se desea insertar con los distintos nodos, y haciendo el recorrido necesario hasta poder insertarlo como nuevo nodo hoja. Dependiendo del balance del árbol esta

operación se puede llevar de manera lineal como se haría con una `Linked List` (todos a la derecha o todos a la izquierda) con complejidad de `O(n)`, y en el caso de un árbol balanceado sería de `O(log n)`, por lo que se dice que en realidad la complejidad dependerá de la altura del árbol, en otras palabras será de `O(h)`.

- `BST<Registro>::ConstReverseIterator::operator++()` - Este método del iterador realmente es el más importante del mismo, puesto a que es el que se encarga de partir del nodo al que pertenece el iterador y encontrar el siguiente nodo siguiendo una secuencia transversal 'InOrder' conversa. La manera en la que se realiza esto es que se determina si es que el nodo al que pertenece el iterador tiene un hijo izquierdo, si es así se busca su predecesor y si no se mueve hacia arriba en el árbol hasta encontrar el siguiente en este orden. Para conseguir esto último, una de las maneras posibles de implementarlo es por medio de un stack que mantenga el recorrido desde la raíz hasta el nodo actual, sin embargo, se decidió que para simplificar las cosas cada nodo tendría además un atributo que apunte al nodo padre del mismo. Finalmente, al igual que la operación pasada este depende completamente de si el árbol se encuentra balanceado o no, por lo que tiene una complejidad de `O(h)`.

A partir de estos algoritmos, así como otras funciones de ayuda como lo son `rbegin()` y `rend()`, se logró obtener entonces estas IPs con más accesos, que en el contexto de la problemática planteada nos permitiría determinar la posibilidad de que un sistema esté infectado, pues al observar de qué IPs provienen el mayor número de accesos fallidos en un corto tiempo se puede determinar cuál es el sistema con más probabilidad de estar infectado y actuando como un 'bot' en una 'botnet', asimismo al observar que una IP tiene un bajo número de accesos fallidos cercanos se puede concluir que es más probable que en realidad se trate de simplemente de una persona o de la mala configuración del servidor.

Finalmente, se concluye que en este caso particular el uso de un BST tuvo importantes ventajas. Principalmente que este inherentemente mantiene una estructura ordenada por lo que no es necesario recorrer todos las entradas de nuevo para ordenarlas, si no que al irlas agregando estas ya estarán en orden. No obstante, se reconoce que el BST que se implementó es el más sencillo que puede haber, careciendo de la habilidad de otro tipo de BSTs más especializados de balancearse a sí mismo, que a cambio de un poco más de complejidad computacional al insertar o buscar elementos traen consigo la ventaja de garantizar este balance. Asimismo, se observa que a pesar de la facilidad del ordenamiento de los datos, realmente acceder de manera secuencial a los mismos es considerablemente más complicado que con las estructuras de datos con las que se trabajó anteriormente, por lo que este tipo de datos son de mejor uso para búsquedas irregulares de elementos no secuenciales.

IV. Estructuras de datos no lineales

Los grafos, dirigidos o no dirigidos, son una estructura de datos no lineales de increíble importancia para modelar y estudiar el mundo real, puesto a que su manera de representar relaciones entre información “de muchos a muchos” es de utilidad para un gran número de situaciones, desde el mapeado de ubicaciones geográficas hasta la planeación de redes informáticas para la interconexión de computadoras. Asimismo, el uso de distintos algoritmos para navegarlos y caracterizarlos, basados en su mayoría por recorridos por profundidad (DFS) o por anchura (BFS), posibilitan la comprensión y análisis sistemático de estas complejas redes y conexiones. De acuerdo a el propósito de la búsqueda o recorrido es el tipo que resulta más conveniente y eficiente, puesto a que en el caso de que se espera encontrar una solución cercana al inicio o que solo se requiera recorrer parcialmente el grafo, un acercamiento por BFS es usualmente más recomendable, en contraste a si se requiere recorrer todo el árbol usualmente DFS es más apropiado al ser más eficiente con la memoria utilizada. No obstante, todo esto es altamente dependiente de la estructura del grafo, la anchura, la profundidad, etc.

Con un contexto similar al último caso mencionado anteriormente, para la situación problema presente se planteó el uso de un grafo dirigido, optando por almacenar los accesos desde una IP hacia otra en una lista de adyacencias. Además de esto, se envolvió este grafo en su propia clase para poder trabajar con una interfaz más abstracta y reusable, basándose para almacenar esta información en el `unordered_map` de la librería estándar de C++. De este modo, las tres importantes secciones en la ejecución de la aplicación fueron las siguientes:

- Creación de vertices del grafo: En esta primera parte, se leyeron los datos de las IPs que se almacenarían como nodos o vertices del grafo, utilizando el `unordered_map::operator[]` e inicializando vectores vacíos para almacenar las adyacencias. Esto significó que esta operación es de complejidad `O(1)` por cada nodo que se inicializa, y `O(n)` para todos los nodos.

- Creación de los arcos del grafo: Posteriormente, se cargó la información de las conexiones por intentos entre IPs, de manera similar a la anterior para el acceso a las adyacencias pero en esta ocasión utilizando `unordered_map::at` para asegurar que los nodos existan en el grafo, y al también ser de complejidad `O(1)` el añadir elementos al vector de adyacencias significa que la complejidad por arco sería constante y para todos los arcos lineal.

- Búsqueda del mayor outdegree: Finalmente, para encontrar todos los nodos que en el contexto de la problemática se titulan como “bot masters”, se utilizó un mapa con los mismos nodos que la lista de adyacencia, pero en lugar del vector de nodos representaría el número de vecinos o “outdegrees”. Con esto, primero se realizó un recorrido para saber cuál era el número mayor de outdegrees en el grafo y siguientemente se realizó otro para mostrar en pantalla todos los coincidentes. A pesar de estas dos vueltas, se dice que la complejidad de esta operación es `O(n)`.

Es de esta manera que gracias al uso de un grafo se pudo encontrar una representación y solución a esta problemática, que no hubiese con la utilización de otras estructuras de datos.

V. Estructuras de datos asociativas

Las tablas de hash son una estructuras de datos bastante utilizada, cuyo principal propósito es almacenar y asociar un conjunto de datos a un conjunto de “llaves” con los que se pueden accesar, que debido al principio matemático y computacional que la fundamenta tiene propiedades que resultan ser tremendamente útiles para ciertas situaciones donde sus cualidades se aprovechen para reducir la complejidad computacional de un programa.

Entre los motivos por los cuales se suele usar una tabla de hash se encuentra que tanto el verificar si una llave existe en la tabla, como encontrar (para poder leerlo o modificarlo) el valor al que “mapea” esta llave, tiende a tener una complejidad constante. Esto se logra de manera tan eficiente gracias a la técnica que le da su nombre, conocida como “hashing”, donde para un espacio de memoria o un arreglo para asignar la ubicación de memoria donde residirá tanto la llave como el dato asociado se transforma matemáticamente la llave de tal manera que se distribuyan de la forma más uniforme posible, evitando las llamadas “colisiones” donde varias el hash de diferentes llaves lleva a la misma ubicación. A pesar de esto, el evitar por completo todo tipo de colisitones para todos los tipos de llaves es estadistica y prácticamente imposible, se deben de implementar técnicas para resolverlas, sea por dirección abierta, por encadenamiento, etc.

No obstante, algunas de las desventajas que presentan las hash tables son consecuencia de los mismos principios que les da sus atributos positivos, pues el hecho de utilizar una función de hashing no óptima para los tipos de datos con los que se trabaja, es decir, una que no distribuya de manera muy uniforme, significa que las colisiones aumentan considerablemente, y combinado a esto una técnica donde la resolución de colisiones no sea la más eficiente, como es el caso de la prueba lineal en la técnica de dirección abierta, significa que la complejidad de realizar las operaciones de pertenencia y búsqueda irán poco a poco a complejidad lineal. Asimismo, para minimizar la probabilidad usualmente es útil el uso de números primos, que introducen otro tipo de complejidad computacional para encontrarlos, y además de esto que son algo menos eficientes respecto al espacio pues para minimizar las colisiones deben existir espacios de memoria sin utilizar gracias al principio del palomar.

Concretamente, para esta solución problema se utilizó el `unordered_map` disponible en la librería estándar, el cual está implementado por medio de una tabla de hash. Sin embargo, a diferencia de los conjuntos que implementamos en clase donde solamente se guardaban los datos en un arreglo o en un vector, este está implementado por medio de “buckets”, o

específicamente por medio de un arreglo de listas encadenadas de nodos, motivo por el cual es evidente que se puede decir que las colisiones son resueltas por encadenamiento. Entonces, al establecer nuestro `unordered_map` donde las llaves fueran strings y los valores fueran vectores, al utilizar el `unordered_map::operator[]` y `vector::push_back`, ambos con una complejidad `O(1)`, se logró tener una complejidad lineal general, y sobre todo una simplicidad en el código del programa honestamente sorprendente gracias a todo lo que tiene por ofrecer la librería estándar de C++.

Conclusión

A pesar de no ser muy sencillo comparar las soluciones que se dieron debido a su gran diferencia en función principal para la que se crearon, se debe de decir que se considera que las soluciones que fueron más eficientes dada la situación que querían resolver son la primera, que hacía uso de los algoritmos de búsqueda y ordenamiento de la librería estándar en un vector, que era bastante eficiente tanto en la memoria como en la complejidad de ordenarlo y buscar, y la última, que hacía uso de un mapa no ordenado de la librería estandar para guardar un resumen de accesos, que tenía una muy buena complejidad logrando un orden lineal para todo el programa y en una sola pasada al archivo.

En contraste a esto, entre las soluciones que podrían haberse mejorado se encuentran la del ábol binario de búsqueda, para el cual se pudo haber implementado algún método de balanceo para reducir la complejidad total, y el grafo no dirigido, que pese a ser relativamente eficiente respecto a la complejidad, no era particularmente bueno en el uso de memoria ya que existía bastante información como strings duplicada, que pudo haber sido reemplazada por enteros asociados al mismo string utilizando una hash table.

Finalmente, la práctica de las actividades que se llevaron a cabo fue de gran utilidad para conocer las fortalezas y debilidades de cada una de las diferentes estructuras de datos que se utilizaron y sus algoritmos asociados. Asimismo, durante la codificación de los programas se pudo practicar la identificación de qué variables o procedimientos podrían tener un efecto inesperado en el flujo regular del programa y el impacto que tiene aumentar la complejidad computacional en la ejecución del programa.