E1. Actividad Integradora 1

Análisis y diseño de algoritmos avanzados
Grupo 602

Tecnológico de Monterrey

Integrantes:

Rogelio Guzman Cruzado A01639914

David Alejandro González Ibarra A01633817

Profesor:

Luis Guillermo Hernández Rojas

Introducción

Para esta primera actividad integradora nos enfocamos en usar algoritmos de manejo de strings ya que la problemática de identificar código malicioso necesita de algoritmos eficientes que puedan detectar patrones comunes en este tipo de prácticas, como palíndromos o alguna sucesión específica de caracteres que se tengan como referencia.

En esto consistió nuestro acercamiento para identificar código malicioso en los archivos de *transmission* que simulan transmisiones entre dispositivos.
Estos debían ser comparados con texto que se encontraban en archivos de texto *mcode* para saber si habían sido infectados o no.

KMP *complejidad temporal: O(m+n)*

Usamos el algoritmo de KMP para encontrar patrones que se repetían dentro de un mismo texto, pero todavía más importante para comparar dos strings de manera eficiente en lugar de hacerlo por fuerza bruta e ir comparando carácter por carácter. Este algoritmo funciona de la siguiente forma: El texto que deseamos encontrar o comparar necesitará ser procesado en un arreglo para tratar de encontrar patrones en cuanto a su prefijo y sufijo (substrings) e ir guardando de acuerdo al índice en el que se encuentre y vamos incrementando el valor cada vez que hagamos match.
Este arreglo temporal nos ayudará a la hora de compararlo con otro string para que, cuando encontremos un carácter igual este pueda tomar como referencia el arreglo temporal y no regresar hasta el principio para comparar de nuevo como lo haríamos en fuerza bruta.

Este algoritmo lo usamos para la primera parte en la que debíamos encontrar si las secuencias de los archivos mcode#.txt se encontraban en los de transimission#.txt.

La complejidad para este algoritmo es de O(m + n) en donde m es la longitud del texto y n la longitud del patrón que deseamos encontrar. Esto lo hace más eficiente que tener que checar carácter por carácter ya que tenemos un arreglo temporal que nos ayuda a

posicionarnos después de encontrar un match y saber a donde ir en caso de perder esa secuencia.

Algoritmo de Manacher *Complejidad temporal: O(n)*

Este algoritmo nos permite encontrar la posición y extensión del palíndromo más largo que se encuentre en el texto deseado. Si utilizaramos la técnica de Dynamic Programming para resolver un problema de esta índole nos dejaría con una complejidad temporal de O(n^2), mientras que un fuerza bruta nos deja con una complejidad cúbica O(n^3). El algoritmo de Manacher utiliza de forma astuta las propiedades simétricas de un palindromo para disminuir bastante la carga de computación, dejando la complejidad en O(n). El algoritmo crea un arreglo que contiene el valor computado de la posible extensión del palíndromo en esa posición. La principal expresión que representa el espejo de cualquier carácter es j' = (2*c) - j
Esto lo hicimos con el propósito de encontrar posible código malicioso "espejado", el cuál en el caso real, podría suponer un riesgo a la seguridad de un sistema.

LCS Complejidad temporal: O(n)

Para el tercer punto que consistía en encontrar la secuencia de carácteres más larga que estuviera en ambos archivos de texto de *transmission* usamos el algoritmo de Longest Common Substring.

Estel tiene un acercamiento de programación dinámica haciendo una matriz para encontrar los lugares en los que los caracteres hacen match e ir sumando 1 al número que esté arriba a la izquierda de ese lugar en la tabla. De igual manera, guardando en una variable la longitud de la secuencia más grande que encuentra y sus coordenadas en la tabla.

Para encontrar la posición inicial y final de esta secuencia, lo que hicimos fue, ya que teníamos la posición donde se terminaba, nos fuimos para atrás, recorriendo

en diagonal (arriba a la izquierda) mientras que el número no fuera 0. Cómo solo nos importa movernos en x para encontrar el índice del primer archivo, guardamos en variables las posiciones que tenía la tabla en las filas que quedamos para el principio y final de la secuencia, dándonos los índices de las letras en la palabra de donde inicia y dónde termina.

La complejidad para este algoritmo es de O (n*m) donde m y n son la longitud de los strings a comparar. Esto se debe a la tabla que hacemos, que es una matriz de m*n. para poder comparar los caracteres de manera eficiente.

En general, nos pareció una actividad retadora pero más que eso, pensamos que fue la actividad perfecta para implementar estos algoritmos sin dejar espacio para dudas, ya que tuvimos que entenderlos para saber implementarlos en nuestro código, en lugar de solo implementarlos arbitrariamente. Sin duda fue muy satisfactorio ver funcionar nuestra implementación después de analizar la problemática y planear el cómo lo íbamos a resolver. Nos quedamos con la seguridad de haber entendido estos algoritmos y el saber que ahora podremos identificar situaciones en las que necesiten ser utilizados, al igual que implementarlos por nuestra cuenta.