Tecnológico de Monterrey

Análisis y diseño de algoritmos avanzados

Actividad Integradora 1

Profesores:

Ramona Fuentes Valdéz

Integrantes - Squad 3

~

Alan Rodrigo Castillo Sánchez | A01708668

Alan Patricio González Bernal | A01067546

Fecha de entrega:

1 oct 2023

Índice

Situación problema 1

Transmisiones de datos comprometidas

Cuando se transmite información de un dispositivo a otro, se transmite una serie sucesiva de bits, que llevan una cabecera, datos y cola. Existe mucha gente mal intencionada, que puede interceptar estas transmisiones, modificar estas partes del envío, y enviarlas al destinatario, incrustando sus propios scripts o pequeños programas que pueden tomar cierto control del dispositivo que recibe la información

Suponiendo que conocemos secuencias de bits de código malintencionado:

- ¿Serías capaz de identificarlo dentro del flujo de bits de una transmisión?
- ¿Podremos identificar si el inicio de los datos se encuentra más adelante en el flujo de bits?
- Si tuviéramos dos transmisiones de información y sospechamos que en ambas han sido intervenidas y que traen el mismo código malicioso, ¿podríamos dar propuestas del código mal intencionado ?

¿Qué tenemos que hacer?

En equipos de máximo 3 personas, escribe un programa en Python que lea 5 archivos de texto (de nombre fijo, no se piden al usuario) que contienen exclusivamente caracteres del 0 al 9, caracteres entre A y F y saltos de línea.

Los archivos de transmisión contienen caracteres de texto que representan el envío de datos de un dispositivo a otro.	transmission1.txt transmission2.txt
Los archivos mcodex.txt representan código malicioso que se puede encontrar dentro de una transmisión.	mcode1.txt mcode2.txt mcode3.txt

Parte 1

El programa debe analizar si el contenido de los archivos mcode1.txt, mcode2.txt y mcode3.txt están contenidos en los archivos transmission1.txt y transmission2.txt y desplegar un true o false si es que las secuencias de chars están contenidas o no. En caso de ser true, muestra true, seguido de exactamente un espacio, seguido de la posición en el archivo de transmisiónX.txt donde inicia el código de mcodeY.txt

Parte 2

Suponiendo que el código malicioso tiene siempre código "espejado" (palíndromos de chars), sería buena idea buscar este tipo de código en una transmisión. El programa después debe buscar si hay código "espejado" dentro de los archivos de transmisión. (palíndromo a nivel chars, no meterse a nivel bits). El programa muestra en una sola línea dos enteros separados por un espacio correspondiente a la posición (iniciando en 1) en donde inicia y termina el código "espejado" más largo (palíndromo) para cada archivo de transmisión. Puede asumirse que siempre se encontrará este tipo de código.

Parte 3

Finalmente el programa analiza qué tan similares son los archivos de transmisión, y debe mostrar la posición inicial y la posición final (iniciando en 1) del primer archivo en donde se encuentra el substring más largo común entre ambos archivos de transmisión.

Propuestas de solución

Parte 1

Para la solución de la parte número 1 básicamente decidimos utilizar el algoritmo *Naive* o algoritmo de comparación de cadenas ingenuo, que utiliza un enfoque conocido como de fuerza bruta ya que es el algoritmo más “sencillo” aplicable pero de gran valor para solucionar este problema. El algoritmo se encarga de buscar patrones o similitudes entre distintas cadenas de texto o *strings* de la siguiente manera:

1. Dados dos strings, comienza comparando el primer carácter de la cadena de patrón con el primer carácter de la cadena de texto.
2. Si los caracteres coinciden, pasa al siguiente carácter en ambas cadenas y continúa comparando. Si no coinciden, avanza un carácter en la cadena de texto y vuelve a comparar el primer carácter de la cadena de patrón con el nuevo carácter de la cadena de texto.
3. Este proceso se repite hasta que se encuentre una coincidencia completa de la cadena de patrón en la cadena de texto o hasta que se llegue al final de la cadena de texto sin encontrar una coincidencia.
4. Si se encuentra una coincidencia completa, se considera que se ha encontrado una similitud o una subcadena igual en la cadena de texto, y se registra la posición en la que comenzó esta coincidencia.
5. El algoritmo puede continuar buscando más ocurrencias de la cadena de patrón en la cadena de texto, repitiendo el proceso desde el paso 1, pero esta vez comenzando la búsqueda desde la posición siguiente en la cadena de texto después de la última coincidencia encontrada.

El algoritmo de comparación de cadenas ingenuo si bien es sencillo de implementar (razón por el cuál fue elegido), puede ser ineficiente en términos de tiempo de ejecución, especialmente cuando se tienen cadenas largas. En el peor de los casos, su complejidad

temporal puede ser O(N*M), donde N es la longitud de la cadena de texto y M es la longitud de la cadena de patrón.

Parte 2

Para esta parte nos decidimos por hacer un algoritmo igualmente con un método de fuerza bruta “Algoritmo de búsqueda de palíndromos por fuerza bruta”, que se encarga de buscar subcadenas que se consideren *palindrómicas*, es decir que se leen igual de derecha a izquierda que viceversa, de la siguiente manera:

1. Comienza observando todas las subcadenas posibles en la cadena de texto, una por una.
2. Para cada subcadena, verifica si es un palíndromo. Lo hace comparando los caracteres en los extremos de la subcadena y comprobando si coinciden.
3. Si encuentra un palíndromo en la subcadena actual y es más largo que el palíndromo más largo encontrado hasta ahora, registra la longitud del nuevo palíndromo y su posición de inicio.
4. Continúa este proceso para todas las subcadenas posibles en la cadena de texto.
5. Al final, devuelve la posición de inicio y fin del palíndromo más largo encontrado en la cadena de texto.

Tomamos este algoritmo como la solución a esta segunda parte del problema debido a que muy a pesar de su mala eficiencia con su complejidad temporal de O(n^2), es un algoritmo relativamente sencillo de entender y de implementar. Esto puede ser beneficioso en situaciones donde la claridad del código es una prioridad, como en la enseñanza, la documentación o en equipos de desarrollo con habilidades variadas

Parte 3

En esta parte utilizamos un algoritmo que se encarga de conocer las subcadenas de texto iguales entre ambas transmisiones más largas que se conoce como *Longest Common Substring* (Subcadena Común Más Larga) y que básicamente funciona de la siguiente manera:

1. Se obtienen las longitudes de las dos cadenas de texto str1 y str2 y se almacenan en las variables m y n, respectivamente.
2. Se crea una matriz LCSuff (Longest Common Suffix) de dimensiones (m+1) x (n+1) para almacenar los resultados de los subproblemas. Cada celda (i, j) de esta matriz representa la longitud del substring común más largo que termina en str1[i-1] y str2[j-1].
3. Se inicializa la variable result a 0 para almacenar la longitud del substring común más largo encontrado hasta ahora.
4. Se inicializa la variable end a 0 para almacenar el índice de la última fila en la matriz LCSuff donde se encontró el substring común más largo.
5. Se utiliza un bucle anidado con dos bucles for para recorrer ambas cadenas de texto y llenar la matriz LCSuff de acuerdo con las siguientes reglas:

    a. Si i o j es 0 (es decir, estamos en la primera fila o columna de la matriz), LCSuff[i][j] se establece en 0 porque no puede haber un substring común en ese punto.
    b. Si los caracteres str1[i-1] y str2[j-1] son iguales, entonces LCSuff[i][j] se establece en LCSuff[i-1][j-1] + 1, lo que significa que se extiende el substring común más largo encontrado hasta ese punto.
    c. Si los caracteres no son iguales, LCSuff[i][j] se establece en 0 porque no hay un substring común en ese punto.
    d. Durante el proceso de llenado de la matriz LCSuff, se actualiza la variable result si se encuentra un nuevo substring común más largo. La variable end se actualiza para almacenar la fila donde se encuentra el nuevo substring común más largo.
6. Luego, se crea una cadena vacía para almacenar el substring común más largo.
7. Se utiliza un bucle while para reconstruir el substring común más largo a partir de la matriz LCSuff. Se agrega un carácter a la cadena res desde str1 y se decrementa end y result hasta que result sea 0.
8. Finalmente, la función devuelve el substring común más largo encontrado en las cadenas str1 y str2, que se encuentra en la variable res.

La decisión de utilizar este algoritmo se debe a que este algoritmo en concreto encuentra el substring común más largo de manera precisa y completa y no se limita a encontrar una coincidencia cercana, sino que identifica la parte más larga que es común a ambas cadenas. Y si bien su complejidad de O(n *m) no consigue ser la más eficaz si es relativamente eficiente, especialmente cuando se comparan cadenas largas y es más eficiente que algunas otras técnicas de búsqueda exhaustiva, como la comparación de todas las subcadenas.

Conclusión final

Al analizar los algoritmos vistos en clase y sus funcionalidades (sobre todo en los utilizados en la resolución de esta actividad integradora) pudimos notar la importancia y sobre todo la sencillez de los mismos. Que a pesar de todos poder resolver un problema en común, en más o menos pasos, todos son sumamente útiles y vigentes hoy en día, la actividad nos permitió comprender cómo puede funcionar un programa de detección de malware y un poco más sobre cómo operan. Sin duda este tipo de actividades nos permiten aumentar y seguir desarrollando mis habilidades que en conjunto con las vistas en materias pasadas, me permitirán sobresalir en el ámbito laboral