Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Querétaro

Análisis y diseño de algoritmos avanzados TC2038.601

Actividad Integradora 1

Presenta:

María Fernanda Moreno Gómez | A01708653
Uri Jared Gopar | A01709413
José Ricardo Rosales Castañeda | A01709449

Querétaro, Querétaro 01 de octubre, 2023

Índice

Transmisión de datos comprometidos

Contexto

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.

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?

Reto

Dados 2 archivos de transmisión (transmission1.txt y transmission2.txt), los cuales contienen caracteres de texto representando el envío de datos entre dispositivos y dados 3 archivos de código malicioso (mcode1.txt, mcode2.txt, mcode3.txt) que se pueden encontrar dentro de una transmisión, crear un programa que analice 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.

Suponiendo que el código malicioso tiene siempre código "espejeado" (palíndromos de chars), sería buena idea buscar este tipo de código en una transmisión. El programa después debe buscar si hay código "espejeado" 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 correspondientes a la posición (iniciando en 1) en donde inicia y termina el código "espejeado" más largo (palíndromo) para cada archivo de transmisión. Puede asumirse que siempre se encontrará este tipo de código.

Finalmente, el programa analiza que 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.

Puntos importantes

1. Verificar si en los archivos de transmisión se encuentra cada código malicioso (True/False)
2. Checar donde inicia y termina el código espejeado más largo (palíndromo) dentro de los archivos de transmisión
3. Mostrar la posición inicial y la posición final del primer archivo donde se encuentra el substring más largo común entre los dos archivos

Propuesta: Algoritmos utilizados

Knuth-Morris-Pratt (KMP)

KMP es un algoritmo de búsqueda de coincidencias en cadenas, el cual, en un tiempo lineal, busca la existencia de subcadenas dentro de una cadena. En este algoritmo, se crea una tabla con valores sobre el contenido del patrón para identificar donde podría existir una coincidencia, sin necesidad de analizar varias veces los caracteres de la cadena donde se está explorando, ya que únicamente analiza la cadena una sola vez.

La complejidad algorítmica de este algoritmo es O(n+m), esto debido a que depende de la cantidad de letras en la subcadena o patrón y la cantidad de letras en la cadena como tal. Esto en términos de eficiencia es algo muy bueno, pues evitamos el backtraking (retroceso) cuando un carácter del patrón no coincide con uno de la cadena. Para evitar el backtraking, podemos hacerlo si sabemos dos cosas:

1. “Si aparece o no un prefijo del patrón (p) más de una vez en la cadena (c) después de que se haya encontrado correctamente un carácter, si es así, se puede omitir al reanudar el proceso de coincidencias después de una discrepancia” (Educative, s.f.).
2. La longitud del prefijo adecuado

Es por eso que, en nuestra implementación, creamos primero un arreglo de tipo Longest Proper Prefix which also Suffix (LPS), el cual, para cada posición “i” en el patrón lps[i] almacena la longitud del prefijo propio más largo que es también un sufijo propio de p[0…1].

Luego, se va a hacer una búsqueda con el algoritmo KMP, el cual, usa LPS para no hacer comparaciones que no se necesiten entre el texto y el patrón. Cuando encuentra una discrepancia entre el patrón y la cadena en la posición “j” del patrón, KMP usa el LPS para saber cuántos caracteres del patrón puede seguir saltando sin perder coincidencias con el patrón, de modo que no se retrocede al principio de la cadena y no se hacen comparaciones de más.

Esto nos da como ventajas la reducción enorme de veces en las que se está buscando coincidencias, ya que el tiempo es lineal al pasar una única vez por la cadena y el patrón, siendo que para inputs de gran tamaño, el tiempo va a ser menor, a comparación de que si lo hacemos a fuerza bruta que compara el patrón con todas las subcadenas de la cadena, dando un tiempo de O(n x m).

Manacher: Linear Time Longest Palindromic Substring

El algoritmo de Manacher es usado para encontrar la subcadena (que sea palíndromo) más larga de una cadena. Este algoritmo es muy eficiente, ya que, si lo hacemos con fuerza bruta, tendríamos que comprobar si cada subcadena de la cadena es un palíndromo o no, teniendo que usar 3 ciclos anidados para hacer esta revisión, dando una complejidad de O(n^3), contrario a lo que sucede con Manacher.

Para implementar Manacher, tuvimos que transformar la cadena de entrada por una cadena de tipo a#b#c#d#, esto debido a que Manacher únicamente puede obtener los palíndromos de longitud impar, por lo que le agregamos “#” para trabajar con los pares.

Posteriormente, Manacher tiene un arreglo llamado “P”, el cual, almacena la longitud del palíndromo más largo con centro en cada posición “i” de la “cadenaNueva”, esto lo hace usando una técnica llamada “espejo” para evitar hacer tantas comparaciones, de modo que si “i” está dentro del palíndromo conocido (“i < R”), el valor de “P[i]“ se inicializa al mínimo de “R-i” y “P[mirr]”, donde “R” es el límite derecho del palíndromo más a la derecha explorado hasta ese momento y “mirr”es el índice espejo de “i”, es decir, “2 * C - i“, donde “C” representa el centro del palíndromo más a la derecha descubierto al momento, con esto, aprovechamos la simetría de los palíndromos para no hacer más comparaciones que no se necesiten. Si “i” está fuera del palíndromo conocido, se realiza una expansión desde “i” para encontrar un nuevo palíndromo.

Durante todo ese proceso, las variables “C” y “R” van a estar en constante movimiento, pues tienen que mantener el palíndromo más a la derecha conocido. Cuando se encuentra un palíndromo más allá del borde derecho “R”, se actualizan estas variables.

Después de recorrer la “cadenaNueva”, Manacher identifica el palíndromo más largo gracias a “P”, que recordemos que es un arreglo que ha ido almacenando la longitud del palíndromo más largo en cada posición “i” de la “cadenaNueva”, por lo que únicamente se busca el número más grande en “P” y se usa esto para determinar el inicio y el final del palíndromo más largo en la cadena original.

Por esto, la complejidad algorítmica de este algoritmo es O(n), debido a que únicamente depende el tiempo de la longitud de la cadena original, siendo muy eficiente tanto para cadenas cortas como para cadenas largas.

Longest Common Subsequence (LCS)

La subsecuencia común máxima de dos arreglos es la “subsecuencia que se encuentra tanto en el primer arreglo, como en el segundo, y es de longitud máxima” (Hernández, s.f.). Es decir, se busca la subsecuencia más larga entre dos textos o cadenas.

Primeramente, se tienen que leer ambos textos, es decir, las dos transmisiones. Se crea una matriz llamada “M” de tamaño (m+1) x (n+1), donde “m” y “n” son las longitudes de ambos textos.

Para cada posición “(i, j)” en la matriz (menos en aquellas posiciones iniciales donde es cero), se tiene que si el carácter “txt1[i]” es igual que “txt2[j]”, se incrementa el valor en “M[i][j]” en 1, esto significa que se ha encontrado una subsecuencia común de longitud “M[i][j]” hasta ese punto. Sin embargo, si no sin iguales, “M[i][j]” se establece en 0, ya que no hay una subsecuencia común.

Durante el llamado de la matriz “M”, se busca el valor máximo en la matriz “maximum” y la posición del último carácter en la subsecuencia común más larga “endIndex”.

Una vez llenada la matriz, se usa el valor “endIndex” y “maximum” para determinar la posición inicial de la LCS en “txt1”. La LCS se construye tomando los caracteres desde “txt1[endIndex - maximum]” hasta “txt1[endIndex]”.

Por lo que, si “maximum” sigue siendo 0 después del proceso, no hay LPS y se devuelve un mensaje diciendo eso, pero si hay, devuelve la LPS encontrada en ambos textos

De esta manera, podemos darle solución a nuestros 3 puntos centrales para resolver este reto, con estos 3 algoritmos presentados.

La complejidad algorítmica de LCS es O(n x m) donde n es la longitud de la primera cadena y m es la longitud de la segunda cadena, esto es debido a que se necesita llenar una matriz de tamaño “n+1 × m+1”.

Implementación y ejecución en el programa

En nuestra carpeta principal “ActInt1_Equipo_05”, tenemos una carpeta llamada “Equipo_05_main”, este controla el flujo principal del programa, donde lee los archivos de transmisión para posteriormente pasarlos a los archivos “KMP.h”, “Manacher.h” y “Substring.h” donde cada uno representa un algoritmo utilizado para resolver el reto, donde en la sección de arriba de esta sección se describió el funcionamiento de cada uno de ellos. Una vez procesadas los archivos de texto que representan la transmisión y el código malicioso, en el main.cpp se imprimen los resultados de cada parte.

Utilizamos dos archivos de transmisión llamados “transmission01.txt” y “transmission02.txt”, los cuales contienen las cadenas de texto a evaluar. También usamos tres códigos maliciosos para evaluar, los cuales son llamados “mcode01.txt”, “mcode02.txt” y “mcode03.txt”.

Al correr el programa con g++ main.cpp -o app y posteriormente ./app, nos da el siguiente output:

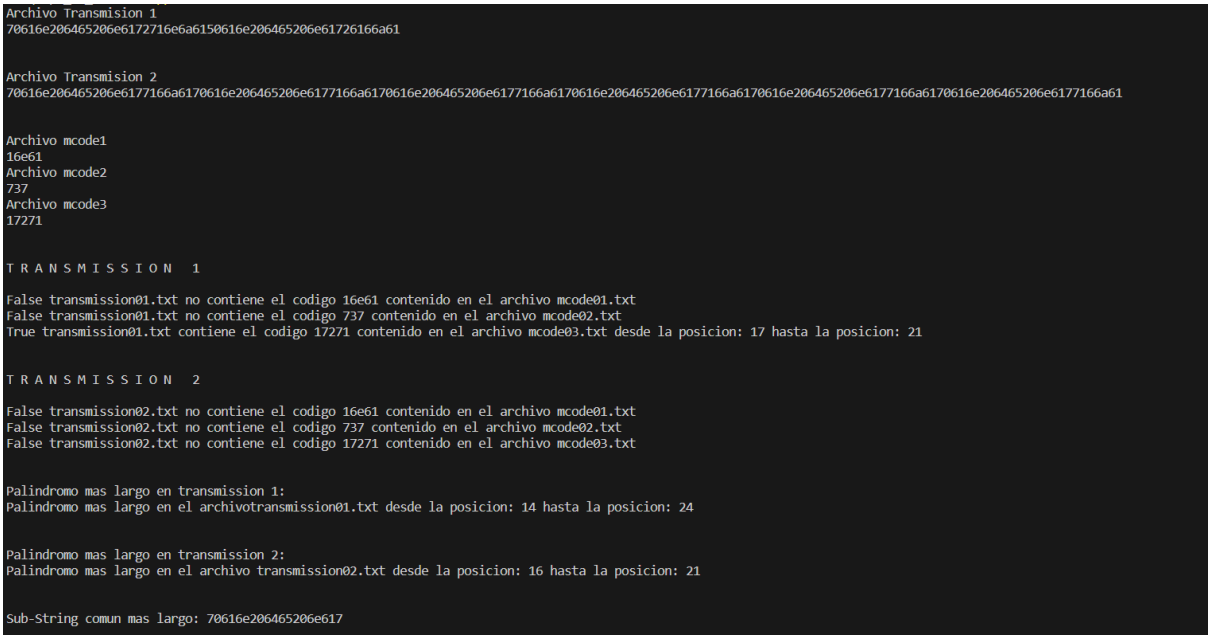

Imagen 1. Impresión de pantalla de la terminal tras haber ejecutado el programa

Como podemos observar, se imprimen primero los archivos de transmisión tal cual están en el .txt, así como los archivos del código malicioso, para posteriormente imprimir el resultado obtenido tras evaluar si el código malicioso se encuentra o no en los archivos de transmisión más la posición en la que se encuentra.

Posteriormente, se imprime el palíndromo más largo en ambos archivos de transmisión y se finaliza con la subcadena común más larga de ambos archivos.

Reflexión

Comparando los algoritmos escogidos con el tiempo de ejecución en fuerza bruta, nos dimos cuenta de que sí hay un cambio radical en tiempos, por lo que al hacer un programa que cumpla con una función no nada más se debe de tomar en cuenta el factor entrega, sino que el programa sea eficiente en tiempo y que esté bien escrito.

En nuestra investigación de algoritmos para resolver el reto, nos dimos cuenta de que la eficiencia y la precisión son de gran importancia, ya que, al haber comparado los algoritmos KMP, Manacher y LCS con otros, así como el de fuerza bruta, pudimos hacer una rápida evaluación sobre el rendimiento que estos tienen.

Primeramente, KMP es un algoritmo que se nos hizo el mejor debido a su eficiencia en la búsqueda de coincidencias en una cadena, debido a que únicamente recorre esta cadena una vez sin necesidad de volver al inicio de la cadena para buscar más coincidencias. Usando una tabla LPS, KMP puede lograr una complejidad lineal, lo cual es ideal por el tiempo de ejecución, ya que, si queremos evaluar cadenas con código malicioso, más largas (que de seguro existen en la vida real), este algoritmo hará ese trabajo en el menor tiempo posible, obteniendo un resultado más rápido en situaciones donde se requiere con rapidez, saber si el código tiene infiltrado código malicioso.

Manacher se nos presentó como la mejor opción para la búsqueda de los palíndromos más largos en una transmisión de datos, ya que, de igual manera, es lineal en esta búsqueda de subcadenas de palíndromos, lo cual es muy importante para detectar otros riesgos en la transmisión de datos que a primera vista, no se ven. Es por eso que se nos hizo muy útil para la seguridad de los datos.

Finalmente, encontramos a LCS como muy importante para verificar el qué tan parecidas son las transmisiones de los datos, debido a que al detectar la subcadena más larga entre dos transmisiones, nos podría ayudar a identificar patrones que nos permitan capturar código que se vea sospechoso y malicioso, y, al hacerlo en un tiempo O(n x m), hace que sea rápido.

Este proyecto nos sirvió a no nada más, codificar, sino investigar, informarnos en los algoritmos que mejor se adaptan a las necesidades del problema para así plantear la mejor solución. También nos abrió un panorama sobre la seguridad de los datos, hoy en día los datos están siendo algo que las personas no les damos la importancia que deberían, vemos cada vez más noticias de robos de datos en aplicaciones, en bancos, etc. Esto es algo que nos concierne a todos, y las grandes empresas a las que les confiamos nuestros datos deben de tener protocolos de identificación y bloqueo de código malicioso, por lo que este reto fue bastante interesante en ese dato al aprender un poquito más de seguridad informática y las implicaciones que se tiene la inyección de código malicioso.

Referencias

Educative answers - trusted answers to developer questions. (s/f). Educative. Recuperado el 29 de septiembre de 2023, de https://www.educative.io/answers/what-is-the-knuth-morris-pratt-algorithm

Follow, G. (2014, diciembre 16). *Manacher's Algorithm - linear time longest palindromic substring - part 1*. GeeksforGeeks. https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/

Hernandez, P. P. G. (s/f). *Subsucesión común máxima (LCS – Longest Common Subsequence)*. Com.mx. Recuperado el 29 de septiembre, de https://pier.guillen.com.mx/algorithms/08-dinamica/08.4-lcs.htm

KMP algorithm for pattern searching. (2011, abril 3). GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

KMP algorithm in detail. (s/f). Gitbook.Io. Recuperado el 1 de octubre de 2023, de https://labuladong.gitbook.io/algo-en/i.-dynamic-programming/kmpcharactermatchingalgorithmindynamicprogramming

Kumar, A. (2022, enero 28). Manachers algorithm in data structures explained. *Scaler Topics*. https://www.scaler.com/topics/data-structures/manachers-algorithm/

Longest common subsequence (LCS). (2011, junio 14). GeeksforGeeks. https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/