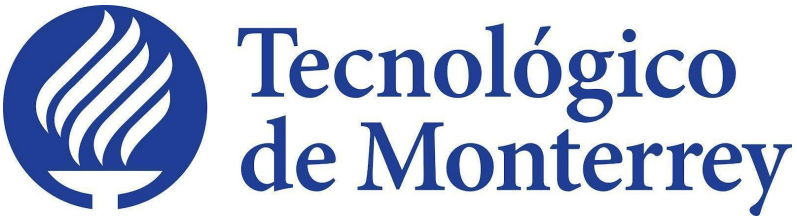

Análisis y diseño de algoritmos avanzados

TC2038.602

Actividad Integradora 1

25 de noviembre 2022

Diego Araque - A01026037
Marco Torres - A01025334
Fernando Valdeón - A01745186
Uriel Aguilar - A01781698

Descripción de la solución implementada

Para la solución de esta actividad integradora se utilizó el algoritmo kmp, el cual nos permite hacer string matching y de esa forma encontrar los pedazos de código malicioso en el archivo de transmisión.

El kmp se encarga de recorrer cada carácter del string de transmisión y comparar el carácter actual con el correspondiente del patrón que estamos buscando. Si estos son iguales hay que revisar si se llegó al final del patrón en el que estamos buscando, si es el caso, se agrega el índice en el que inicia el patrón en un arreglo (esto contiene todas las posiciones iniciales en las que se ubica el patrón analizado en dicha transmisión), se reinicia el índice del patrón y se vuelve a buscar a partir de donde se terminó en la iteración anterior, si no es el último carácter se vuelve a correr la iteración pero avanzamos un paso tanto en el patrón como en la transmisión. De otro modo si ya hemos hecho match de distintos caracteres y el próximo no coincide, reiniciamos el índice del patrón y nos quedamos en la misma posición de la transmisión (nota: aquí no se aumenta el contador de la transmisión, por lo que es posible iterar más de k veces, k siendo la longitud de la transmisión). Como paso final, si el primer carácter del patrón no coincide, avanzamos 1 carácter en la transmisión y repetimos el proceso. Nunca se reinicia el índice de la transmisión, esto es lo que permite reducir la complejidad computacional y lo que diferencia a este algoritmo de una solución de fuerza bruta.

En el párrafo anterior se menciona reiniciar el índice en el que nos encontramos en nuestro patrón. Este se realiza con respecto a otro arreglo que contiene los prefijos más largos de cada carácter. Sin este arreglo, la implementación de este algoritmo es impensable.

Para encontrar los prefijos más largos recorremos cada carácter del patrón y vamos comparándolo con un carácter que se encuentra en la posición del tamaño del prefijo de carácter anterior. Dicho esto, el primer carácter siempre será 0 porque no tiene un prefijo, los demás dependen de la longitud del prefijo del carácter anterior.

Al correr nuestro algoritmo necesitaremos de una transmisión y un patrón de texto que se busque en este. El patrón debe ser más pequeño que la transmisión, si no es el caso, lógicamente nunca se encontrará el patrón. Como resultado recibiremos un archivo que diga si se encontró el archivo o no, cuantas veces se encontró el patrón y en qué posiciones se encuentra.

A continuación presentamos un ejemplo mas grafico de como funciona el algoritmo:

transmission=aacaabca
pattern=aab
array=[0]
locArray=[]

Generar arreglo de prefijos

Iteración 1
i=1
length=0

pattern= **aa**b
Coinciden por lo que agregamos 1 a nuestro arreglo
array=[0,1]

Iteración 2
i=2
length=1

pattern= a**ab**
No coinciden y length no es 0, por lo que la reiniciamos
length=arr[length-1]
length=0

Iteración 2
i=2
length=0

pattern= **aa**b
No coinciden y length es 0, por lo que aumentamos i en 1 y agregamos 0 a nuestro arreglo de prefijos
arr = [0,1,0]

No hay mas iteraciones porque i es mas grande que la longitud del patron

Implementación del kmp

n=3
m=8
i=0
j=0

Iteración 1
i=0
j=0
transmission= **a**acaabca
pattern=**a**ab

Coinciden por lo que aumentamos i y j en 1.
revisamos si ya estamos en el final del patrón, como no es el caso iteramos otra vez

Iteración 2
i=1
j=1
transmission= a**a**caabca
pattern=a**a**b

Coinciden por lo que aumentamos i y j en 1.
revisamos si ya estamos en el final del patrón, como no es el caso iteramos otra vez

Iteración 3
i=2
j=2
transmission= aa**c**aabca
pattern=aa**b**

No coinciden y como no estamos en el primer caracter reiniciamos j con respecto a nuestro arreglo de prefijos.
j = array[j-1]
j= 1

Iteración 4

i=2
j=1
transmission= aa**c**aabca
pattern=a**a**b

No coinciden y como no estamos en el primer caracter reiniciamos j con respecto a nuestro arreglo de prefijos.
j = array[j-1]
j= 0

Iteración 5

i=2
j=0
transmission= aa**c**aabca
pattern=**a**ab

No coinciden y como estamos en el primer caracter avanzamos un en i.
i+=1

Iteración 6

i=3
j=0
transmission= aac**a**abca
pattern=**a**ab

Coinciden por lo que aumentamos i y j en 1.
revisamos si ya estamos en el final del patrón, como no es el caso iteramos otra vez

Iteración 7

i=4
j=1
transmission= aaca**a**bca
pattern=a**a**b

Coinciden por lo que aumentamos i y j en 1.
revisamos si ya estamos en el final del patrón, como no es el caso iteramos otra vez

Iteración 8

i=5
j=2
transmission= aacaa**b**ca
pattern=aa**b**

Coinciden por lo que aumentamos i y j en 1.
Revisamos si ya estamos en el final del patrón, como si es el caso agregamos el indice en el que empieza a nuestro array de resultados.
locaArray.append[i-j] => locArray=[3]
Al igual reiniciaremos nuestra j:
j = array[j-1]
j=0

Iteración 9

i=6
j=0
transmission= aacaab**c**a
pattern=**a**ab

No coinciden y como estamos en el primer caracter avanzamos un en i.
i+=1

Iteración 10

i=7
j=0
transmission= aacaabc**a**
pattern=**a**ab

Coinciden por lo que aumentamos i y j en 1.
revisamos si ya estamos en el final del patrón, como no es el caso iteramos otra vez

No hay mas iteraciones porque i es mas grande que la longitud de la transmisión.
Se encontro el string solo una vez en la posición 3 de la transmisión

Complejidad Computacional

Para calcular la complejidad de nuestro algoritmo tenemos que tomar dos cosas en cuenta, primero que todo la longitud del pedazo de código que se está buscando en la transmisión, debido a que lo recorreremos para generar un arreglo con las longitudes de los prefijos de cada posición del string.

Además de esto necesitamos la longitud del mensaje transmitido, debido a que recorreremos este en su totalidad, pero se diferencia de un algoritmo de fuerza bruta gracias al uso de este arreglo de prefijos. Esto porque si fuera de fuerza bruta podríamos recorrer los primero 5 caracteres y si el 6 no coincide, empezamos de nuevo el proceso de analizar la transmisión en el segundo carácter. El arreglo de prefijos nos ayuda a reducir la redundancia, y nos puede decir si avanzamos al carácter 2 es necesario o si lo podemos omitir y empezar en otro.

Dicho esto para la función que obtiene los prefijos tiene una complejidad computacional hay que tomar en consideración el tamaño del patrón que vamos a recorrer en su totalidad, a este tamaño lo denominaremos como *n*. Solo hay una condición en la cual no aumentaremos el contador al recorrer el string, esta condición es cuando la longitud del prefijo anterior no es 0. Para que esto suceda tienen que haber coincidido mínimo un string anteriormente (en donde si estaríamos aumentando este contador), dicho esto se puede deducir que en el peor de los casos se recorre *n* veces más de las estipuladas, dejando la complejidad de nuestro algoritmo en ***O(n+n)***, que es lo mismo que tener una complejidad computacional de ***O(n)***.

La función kmp sigue la misma metodología, pero entenderlo se hace un poco más confuso debido a que utiliza dos contadores. Uno para el patrón y otro para la transmisión. En este caso recorreremos toda la transmisión, al igual que lo hicimos para encontrar los prefijos. Por lo que la longitud de esta será muy importante y la denominaremos *k.* Solo hay una condición en la cual el contador de la transmisión no aumenta, que es cuando no estamos en el primer carácter del patrón y los caracteres no coinciden. En este caso iniciaremos nuestro contador *j* del patrón usando nuestro arreglo de prefijos. De igual manera que para encontrar los prefijos, no se puede entrar en esta condición si no han coincidido caracteres anteriormente, por lo que a esta condición en el pero de los casos se entra *k* veces. La complejidad de esta función sería ***O(k+k)***, que es lo mismo que tener una complejidad computacional de ***O(k)***.

Después de haber dividido nuestro problema en 2 partes, para calcular la complejidad de manera más simple, se puede decir que nuestra solución tiene una complejidad computacional de ***O(n + k)***. Siendo *n* la longitud del patrón que se está buscando y *k* la longitud de la transmisión en la cual estamos buscando el patrón.

Tres aplicaciones distintas de los algoritmos usados

1. Una de las aplicaciones es precisamente la de encontrar código u algún otro texto que sea malicioso en algún archivo. Se seguirá el mismo proceso que se usó en la situación problema y se podrían añadir nuevos parámetros para acelerar el proceso.

Como por ejemplo hacer una paralelización, para poder buscar distintos códigos en el mismo archivo. Como lo tenemos pensado, es que se corra la misma función para cada pedazo de código que se está buscando, en vez de hacerlo de manera secuencial y cuando se termine el análisis de uno empezar con el otro.

2. En nuestras computadoras existe una función que se convoca con ctrl+f o cmd+f. Esta nos permite buscar un texto específico en un documento, página web, etc. A pesar de que pensamos que pueden existir algoritmos más rápidos, no se nos hace descabellada la idea de usar este para matchear strings y así encontrarlos más rápidos en cualquier archivo, página web, etc.
3. Por último pensamos en un caso de uso bastante parecido al de encontrar código malicioso, pero para buscar palabras prohibidas en un chat room. Antes de que se mande el mensaje, se hace esta revisión, y si no contiene ninguna de las palabras prohibidas se enviará, si contiene prohibir el mensaje y obligar a cambiarlo. Para esta solución sentimos que también sería de gran ayuda hacer una paralelización para no arruinar la experiencia de usuario.