E1 Actividad Integradora 1

Reflexión de la Evidencia

Sergio Gómez Guerrero
ITESM
Monterrey, Nuevo León, México
a01571538@tec.mx

Angel Angulo Prudencio
ITESM
Monterrey, Nuevo León, México
A00840313@tec.mx

Rutilo Alberto
ITESM
Monterrey, Nuevo León, México
A01384647@tec.mx

Hector Aranda Garcia
ITESM
Monterrey, Nuevo León, México
a01178284@tec.mx

RESUMEN

La meta de este escrito es explicar y reflexionar sobre la utilidad y eficiencia de los algoritmos para la identificación de patrones en textos; el contexto específico en el que se realizará esto involucra la transmisión de datos y la identificación de patrones maliciosos que pueden llegar a encontrarse en las transmisiones. En este documento se evaluará la complejidad de los algoritmos utilizados y se explicará su uso bajo el contexto de la situación problema.

CLASIFICACIÓN CCS

• D.1.0 • F.2.0

PALABRAS/CONCEPTOS CLAVE

Algoritmos, Complejidad, Búsqueda de Patrones.

1 Búsqueda de Subsecuencias

El primer objetivo en la resolución de esta situación problema radica en hallar, en caso de existir, una secuencia determinada y denominada como maliciosa en el texto que conforma una transmisión. Esto se puede lograr al emplear un algoritmo conocido como Knuth-Morris-Pratt (KMP).

KMP es un algoritmo para encontrar un patrón específico dentro de un texto eficientemente al precomputar un patrón para considerar y evitar comparaciones innecesarias. Tiene una complejidad de tiempo de $O(n + m)$ y una complejidad de espacio de $O(m)$, n siendo el tamaño del texto y m el tamaño del patrón. Lo primero que hace es computar un arreglo LPS (Longest Prefix Suffix)[1] para el patrón, posteriormente estos se utilizan en conjunto para identificar el patrón en el texto seleccionado. Se recorren simultáneamente el patrón y el texto y se comparan. Si el símbolo en la posición actual es igual al del patrón se avanza el puntero de ambos; si el símbolo es diferente se recorre el puntero del patrón a la posición indicada por el arreglo LPS.

Usando este método se lograron encontrar cadenas de texto maliciosas dentro de las transmisiones analizadas. Se evaluó cada posible patrón para cada transmisión, resultando en cuatro búsquedas con resultado positivo.

```
Transmission1.txt | mcode1.txt: true<192>
Transmission1.txt | mcode2.txt: true<204>
Transmission1.txt | mcode3.txt: false
Transmission2.txt | mcode1.txt: true<166>
Transmission2.txt | mcode2.txt: true<178>
Transmission2.txt | mcode3.txt: false
```

Fig. 1: Resultados con casos de prueba provistos.

1 El prefijo propio (no igual a la cadena) más largo que también es un sufijo de la cadena.

2 Búsqueda de Palíndromos

El objetivo principal de esta actividad es desarrollar una solución eficiente para identificar el palíndromo de mayor longitud dentro de una cadena de texto. La finalidad es determinar y reportar la posición del carácter inicial y final que delimitan dicho palíndromo, utilizando un sistema de coordenadas basado en 1. Para abordar este problema de manera óptima, se implementó el algoritmo de Manacher.
Este algoritmo nos ofrece una solución altamente eficiente con una complejidad temporal lineal O(n), superando a los enfoques de fuerza bruta que resultarían en tiempos de ejecución cuadráticos ($O(n^2)$) o cúbicos ($O(n^3)$). La genialidad del algoritmo de Manacher radica en su capacidad para resolver el principal desafío al buscar palíndromos: manejar de forma unificada tanto los casos de longitud impar (ej. "reconocer") como los de longitud par (ej. "abba"). Para realizar esto el algoritmo implementa caracteres especiales para dividir el texto de forma que puede manejar los palíndromos pares e impares. Inicia con “@” para marcar el principio del texto y evitar fallos a la hora de la expansión del palíndromo, luego inserta un carácter especial, como '#', entre cada par de caracteres originales. Esto asegura que tanto los palíndromos de longitud par (como "abba", que se transforma en "#a#b#b#a#") como los de longitud impar sean tratados de manera uniforme, ya que en la cadena transformada todos los palíndromos tienen una longitud impar. Finalmente, se añade otro carácter centinela, como '$', al final para manejar de forma segura el límite derecho de la cadena.
La optimización clave del algoritmo es reutilizar información de palíndromos ya descubiertos para evitar cálculos redundantes.

Si el punto de análisis actual i está dentro de un palíndromo previamente encontrado, el algoritmo usa la longitud de su "posición espejo" como un punto de partida inteligente. A partir de esa estimación inicial, intenta expandir el palíndromo hacia afuera. Este método de no recalcular lo ya conocido es lo que le permite alcanzar una eficiencia de tiempo lineal (O(n)).

```
Analizando archivo: Transmission1.txt
Las posiciones del palindromo mas largo: 49 92

Analizando archivo: Transmission2.txt
Las posiciones del palindromo mas largo: 63 114
```

Fig. 2: Resultados de las posiciones de los polinomios en los ejemplos dados.

3 Substring común más largo

El código crea un Automatón de Sufijos (SAM) con t1 para resolver la Subcadena Común Más Larga con t2 en tiempo lineal. Cada estado guarda len, link, transiciones next[256] y una posición final first_pos. Al insertar cada carácter de t1 con extend, se crean estados y, si hace falta, un clon para mantener las longitudes correctas. Luego se recorre t2 carácter a carácter manteniendo el estado actual v y la longitud del match l: si hay transición se avanza; si no, se retrocede por sufijo-links hasta poder continuar o reiniciar. Tras cada paso, se “canoniza” el estado para que la longitud sea válida y se actualiza el mejor candidato usando first_pos para reconstruir índices en t1. La salida son posiciones 1-based e inclusivas: start = bestEnd - bestLen + 2, end = bestEnd + 1; en empates de longitud, se elige la que empieza antes en t1. Con una complejidad de O(|t1| + |t2|) en tiempo y O(|t1|) en espacio.

4 Huffman Coding

Por último, se usará Huffman Coding para comprar los archivos de *transmission* con cada uno de los archivos de *mcode* para checar si el archivo *mcode* es sospechoso. Esto se hace a través de la creación de un min-heap, donde a cada nodo le corresponde un carácter y su frecuencia. De la frecuencia, se genera un código en binario. Luego, se analiza los caracteres del archivo *mcode.txt* para generar un string concatenado de los códigos binarios. La longitud del string codificado es comparado con la de *transmission.txt*. En nuestra implementación, la longitud de *mcode* no puede exceder 50% más del tamaño de *transmissions.txt* para ser marcado como sospechoso. Esta implementación, debido a su uso de min-heap para almacenar los códigos y la lectura de un string, tiene una complejidad de tiempo O(n log n). La complejidad de espacio es O(n), dado a que depende solamente en el tamaño del string en *transmissions.txt.* Para la lectura y comparación de strings, Huffman Coding facilita ambos procesos al convertir los caracteres en códigos binarios guardados en un árbol. Estos códigos son más fáciles de leer y, aunque no se puede demostrar con textos de prueba tan cortos, a largo plazo brindan una mejora en velocidad.

Bibliografía

[1] GeeksforGeeks. (2025, August 27). KMP algorithm for pattern searching. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/kmp-algorithm-for-pattern-searching/
[2] GeeksforGeeks. (2025, July 30). Manacher’s Algorithm. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/

[3] GeeksforGeeks. (2025, July 23). Huffman Coding | Greedy Algo-3. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/huffman-coding-greedy-algo-3/

[4] GeeksforGeeks. (2024, 30 septiembre). *Longest common substring.* GeeksforGeeks. https://www.geeksforgeeks.org/dsa/longest-common-substring-dp-29/