Búsqueda Aproximada Directamente En Texto Comprimido

Carlos Avendaño Pérez, Claudia Feregrino Uribe, Gonzalo Navarro Badino.

Reporte Técnico No. CCC-04-007
22 de Julio de 2004

© Coordinación de Ciencias Computacionales
INAOE

Luis Enrique Erro 1
Sta. Ma. Tonantzintla,
72840, Puebla, México.

Búsqueda Aproximada Directamente En Texto Comprimido

Carlos Avendaño[1], Claudia Feregrino[1], Gonzalo Navarro[2]

[1]Coordinación de Ciencias Computacionales,
Instituto Nacional de Astrofísica, Óptica y Electrónica,
Luis Enrique Erro 1, Sta. Ma. Tonanzintla,
72840, Puebla, México
carlosap@inaoep.mx, cferegrino@inaoep.mx

[2] Departamento de Ciencias de la Computación,
Universidad de Chile,
Blanco Encalada 2120,
Santiago, Chile.
gnavarro@dcc.uchile.cl

Resumen. El problema de búsqueda aproximada de patrones directamente en texto comprimido trata de encontrar todas las coincidencias de un patrón dentro de un texto comprimido sin necesidad de descomprimirlo, teniendo en consideración que la coincidencia del patrón con el texto puede tener un número limitado de diferencias. Este problema tiene aplicaciones en recuperación de información, biología computacional y procesamiento de señales, entre otras.

En este trabajo se resuelve el problema expuesto usando texto comprimido con LZ78/LZW. La solución consiste en dividir el patrón en varios subpatrones y realizar una búsqueda exacta multipatrón con éstos. Cada que se encuentra un subpatrón se verifica la coincidencia del patrón completo usando autómatas simulados con paralelismo de bits. Los resultados muestran que esta solución es hasta 3 veces más rápida que la solución clásica de descomprimir primero el texto, ejecutar la búsqueda y volver a comprimir el texto.

Palabras clave. Búsqueda de patrones, búsqueda aproximada, compresión de texto, autómata, paralelismo de bits.

1 Introducción

La evolución tecnológica en el área de la informática y el amplio uso de la Word Wide Web, ha propiciado un considerable aumento de información textual disponible en diversos medios electrónicos tales como bibliotecas digitales, bases de datos de texto y páginas electrónicas. La búsqueda de patrones permite acceder rápidamente a zonas de interés dentro de esta gran cantidad de información.

La búsqueda de patrones dentro de un texto se define como:

> Dado un patrón $P = p_1 ... p_m$ y un texto $T = t_1 ... t_u$, ambas secuencias de caracteres sobre el alfabeto finito $\sum$, encontrar todas las coincidencias de P en T, es decir, encontrar el conjunto $\{|x|, T = xPy\}$.

Sin embargo, es común encontrar documentos que contienen palabras mal escritas, esto propiciado al capturar los documentos o a partir de software de reconocimiento óptico, entre otros, lo cual hace imposible recuperar estos documentos a menos que se cometa el mismo error en la consulta.

Una generalización del problema de búsqueda de patrones es la búsqueda aproximada de patrones [1], en la cual se tiene un patrón y un margen de error *k*, que es la máxima distancia permitida entre el patrón y sus coincidencias en el texto. Formalmente, hay que encontrar el conjunto $\{|xP'|, T = xP'y$ y $ed(P, P') \leq k\}$, donde $ed(P, P')$ es la distancia de edición entre ambas cadenas. En este trabajo se considera la distancia de Levenshtein [2], la cual se define como el menor número de inserciones, eliminaciones, o reemplazos de caracteres necesarios para hacer iguales dos palabras.

Una parte del problema de búsqueda de patrones está relacionada con la compresión de texto [3]. La compresión de texto es una opción para reducir el espacio necesario para el almacenamiento de las colecciones de texto y su transmisión por la red. Existen diversos métodos de compresión, entre ellos los de la familia Ziv-Lempel particularmente LZ78 [4] y LZW [5], los cuales son muy populares por su buena razón de compresión combinados con un tiempo eficiente de compresión y descompresión.

El problema de búsqueda de patrones sobre texto comprimido fue definido por primera vez en el trabajo de Amir y Benson [6] de la siguiente manera:

> Dado un texto $T = t_1 ... t_u$, cuyo texto comprimido correspondiente es $Z = z_1 ... z_n$, y un patrón $P = p_1 ... p_m$, encontrar todas las coincidencias de P en T, usando solamente Z.

Un excelente ejemplo donde el problema de búsqueda de patrones y la compresión de texto se combinan son las bases de datos de texto, dado que el texto debe estar comprimido para ahorrar espacio de almacenamiento y tiempo de transmisión en la red y, al mismo tiempo se deben realizar búsquedas eficientes sobre ellas. La búsqueda aproximada de patrones directamente sobre texto comprimido fue tratada por primera vez en 1992 en [6]. Desde entonces han surgido soluciones [7, 8] que en la práctica resultan más lentas que descomprimir primero el texto y hacer la búsqueda después.

En este trabajo se presenta una nueva solución a este problema, la cual consiste en dividir el patrón en en $k+1$ subpatrones y realizar una búsqueda exacta multipatrón con el conjunto de subpatrones resultantes utilizando el algoritmo de Boyer-Moore [9]. Cada que se encuentra un subpatrón se verifica la coincidencia del patrón completo permitiendo k diferencias mediante dos autómatas, uno verifica hacia la izquierda y otro hacia la derecha del subpatrón encontrado, estos autómatas se simulan mediante paralelismo de bits. Los resultados muestran que para un nivel de error moderado en $k/m < ½$ esta

solución es hasta 3 veces más rápida que los mejores algoritmos [10] y [11] para búsqueda aproximada que primero descomprimen el texto y después ejecutan la búsqueda.

2 Conceptos Básicos

En esta sección se esbozan conceptos y notaciones importantes necesarios para comprender este trabajo. Se asume que el lector tiene conocimientos básicos sobre algoritmos de búsqueda de texto y compresión de texto. Se inicia con una breve explicación sobre algoritmos de búsqueda de patrones monopatrón, multipatrón y búsqueda aproximada. Después se detallan los algoritmos de compresión LZ78 y LZW, finalizando con una breve explicación sobre paralelismo de bits.

2.1 Búsqueda de patrones.

Búsqueda monopatrón. El problema de búsqueda de patrones trata de encontrar todas las coincidencias de un patrón $P = p_1 \ldots p_m$ en el texto $T = t_1 \ldots t_u$, donde tanto P y T son secuencias de caracteres sobre un alfabeto finito Σ.

Los algoritmos de búsqueda monopatrón se pueden clasificar de acuerdo a la forma en que buscan el patrón en el texto. Todos ellos utilizan una ventana de búsqueda del tamaño del patrón, la cual se desliza de izquierda a derecha a lo largo del texto, y dentro de la cual se busca el patrón, el esquema general se muestra en la figura 1.

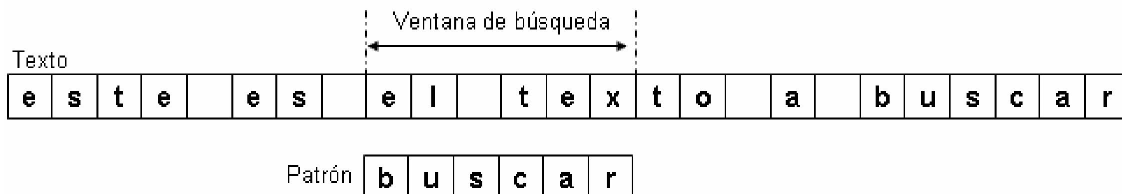


Figura 1. Búsqueda monopatrón: La búsqueda se realiza en una ventana que se desplaza a lo largo del texto. La ventana de búsqueda es del tamaño del patrón.

La diferencia entre los algoritmos radica en la forma en que la ventana se desplaza. Dada una cadena x, y, y z, se dice que x es prefijo de xy, un sufijo de yx, y un factor de yxz. En base a esto último la clasificación de los algoritmos de búsqueda de monopatrón es la siguiente:

Búsqueda por prefijo (figura 2). La búsqueda se hace hacia delante en la ventana de búsqueda, leyendo todos los caracteres del texto uno después de otro. En cada posición de la ventana, se busca el prefijo más largo de la ventana que también es un prefijo del patrón. Entre los algoritmos más conocidos que utilizan este método están el algoritmo de Knuth-Morris-Pratt (KMP) [12], Shift-And [13] y Shift-Or [14].

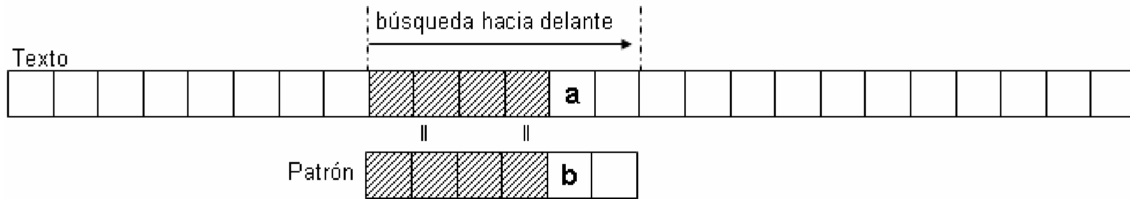


Figura 2. Búsqueda monopatrón por prefijo: Se busca el prefijo más largo del patrón en la ventana de búsqueda.

Búsqueda por sufijo (figura 3). La búsqueda se hace hacia atrás a lo largo de la ventana de búsqueda, se lee el sufijo más largo de la ventana que también es un sufijo del patrón. Este método permite evitar leer algunos caracteres del texto. El algoritmo más famoso que utiliza este método es el de Boyer-Moore[9].

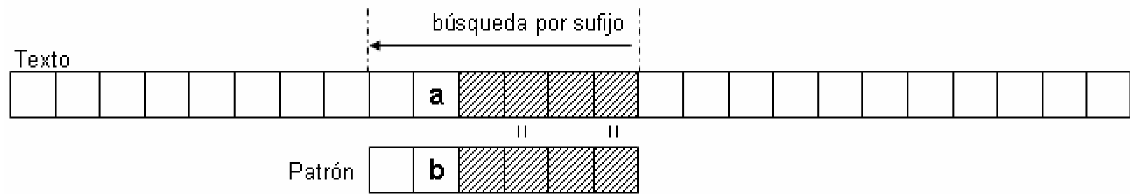


Figura 3. Búsqueda monopatrón por sufijo: Se busca un sufijo del patrón en la ventana de búsqueda.

Búsqueda por Factor (figura 4). La búsqueda se hace hacia atrás en la ventana de búsqueda, buscando el sufijo más largo de la ventana que también es un factor del patrón. La principal desventaja de este método es que requiere una forma para reconocer el conjunto de factores del patrón, lo cual es bastante complejo.

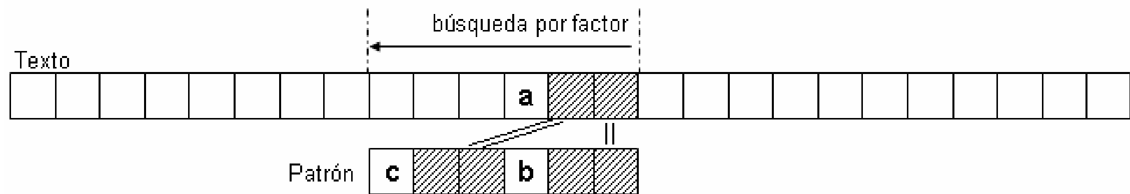


Figura 4. Búsqueda monopatrón por factor: Se busca un factor del patrón en la ventana de búsqueda.

Estas tres formas para buscar un patrón permiten a los algoritmos ser eficientes en distintos casos, dependiendo del tamaño del patrón y el tamaño del alfabeto.

Búsqueda multipatrón. El problema de búsqueda de patrones se puede extender para buscar un conjunto de patrones simultáneamente $P = \{p^1, p^2, ..., p^r\}$, donde cada p^i es una cadena $p^i = p_1^i p_2^i ... p_{m_i}^i$ sobre un conjunto de caracteres finitos $\sum$. La búsqueda se

hace sobre un texto $T = t_1...t_u$. El objetivo es encontrar todos los pares i, j tal que $t_{j-|p^i|+1}...t_j$ sea igual a p^i.

La solución más simple a este problema es realizar r búsquedas con alguno de los algoritmos clásicos de búsqueda monopatrón. Esto lleva a una complejidad en el peor de los casos de $O(ru)$. La complejidad de la fase de búsqueda se puede reducir a $O(u + nocc)$ (donde $nocc$ es el número total de coincidencias) usando algún tipo de extensión de los algoritmos de búsqueda monopatrón.

Los tres esquemas para búsqueda monopatrón se aplican para búsqueda multipatrón. Para cada esquema existen diversas extensiones posibles de acuerdo a la forma en que se manipula el conjunto de patrones y a cómo se obtienen los desplazamientos.

Búsqueda aproximada. Una generalización del problema de búsqueda de patrones es la búsqueda aproximada de patrones o también conocida como búsqueda permitiendo errores, en la cual se tiene un patrón P y un margen de error k.

Este problema tiene sentido sólo si $0 < k < m$, puesto que en otro caso cualquier subcadena de longitud m puede ser convertida a P sustituyendo los m caracteres. Cuando $k = 0$ corresponde a una búsqueda exacta monopatrón. El nivel de error $\alpha = k/m$ nos da una medida de la fracción del patrón que puede ser alterado. A continuación se describen las soluciones más rápidas para resolver este problema, particularmente cuando $\alpha <$ ½.

Programación Dinámica. La solución clásica para búsqueda aproximada de patrones se basa sobre la programación dinámica. Una matriz $E[i, j]$, representa el número mínimo de errores permitidos para hacer coincidir $p_{1...i}$ ($i \leq m$) con $t_{1...j}$ ($j \leq u$). Las entradas de la primera columna $E[0, j]$ ($0 \leq j \leq u$) se inicializan a 0, y las entradas de las primeras filas $E[i,0]$ ($0 \leq i \leq m$) se inicializan a i. Las demás entradas $E[i, j]$ ($1 \leq i \leq u, 1 \leq j \leq m$) se calculan dinámicamente columna por columna cómo muestra el siguiente seudocódigo.

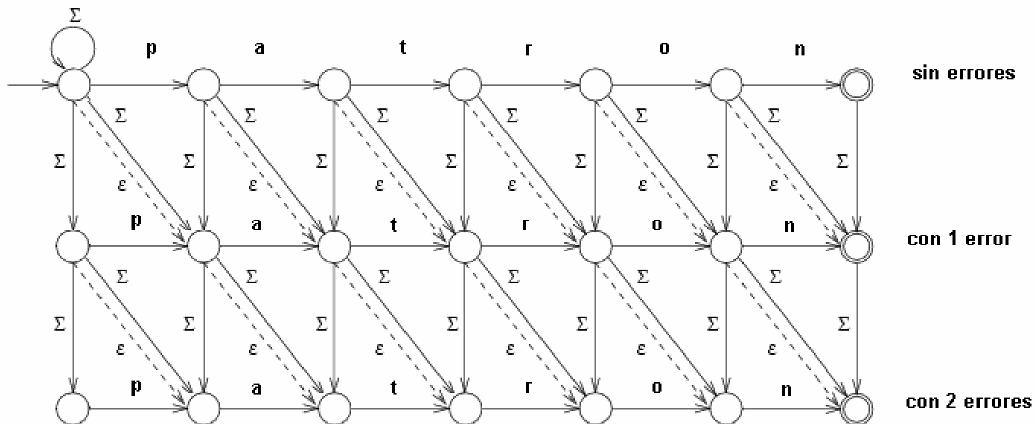
```
if  pi = tj  then
      E [i, j ] := E [ i-1, j-1]
else
      E [i, j ] := 1 + min( E [i-1, j-1], E [i-1, j ], E [i, j-1])
end
```

La figura 5 aplica el algoritmo para buscar el patrón '*incidir*' en el texto '*coincidencia*' con máximo 2 errores.

		c	o	i	n	c	i	d	e	n	c	i	a
	0	0	0	0	0	0	0	0	0	0	0	0	0
i	1	1	1	0	1	1	0	1	1	1	1	0	1
n	2	2	2	1	0	1	1	1	2	1	2	1	1
c	3	2	3	2	1	0	1	2	2	2	1	2	2
i	4	3	3	3	2	1	0	1	2	3	2	1	2
d	5	4	4	4	3	2	1	0	1	2	3	2	2
i	6	5	5	4	4	3	2	1	1	2	3	3	3
r	7	6	6	5	5	4	3	**2**	**2**	**2**	3	4	4

Figura 5. Ejemplo del algoritmo de programación dinámica para buscar "*incidir*" en el texto "*coincidencia*" permitiendo 2 errores. Los números resaltados indican las posiciones finales de las coincidencias en el texto.

Algoritmos basados en autómatas. El problema de búsqueda aproximada se puede reducir al problema de un autómata finito no determinístico (NFA). Por ejemplo, si se tienen k = 2 errores tal como se muestra en la figura 6, cada fila representa el número de errores obtenidos. La primera fila representa cero errores, la segunda 1 error y así sucesivamente. Cada una de las columnas representa la coincidencia de un prefijo del patrón.



|
Figura 6. Un Autómata finito no determinístico para búsqueda aproximada de la cadena "patrón" permitiendo 2 errores.

Cada que se lee un carácter del texto el autómata cambia de estado. Las flechas horizontales representan la coincidencia de un carácter (es decir, si el carácter del patrón y el texto coinciden, se avanza tanto en el patrón como en el texto), las flechas verticales representan inserciones en el patrón (se avanza en el texto pero no en el patrón), las flechas diagonales sólidas representan reemplazos (se avanza en el texto y en el patrón) y las flechas diagonales punteadas representan eliminaciones en el patrón (se consideran transiciones nulas, por lo tanto se avanza en el patrón pero no en el texto). El ciclo en el estado inicial permite que una coincidencia ocurra en cualquier parte del texto. El autómata indica una coincidencia cuando algún estado del extremo derecho está activo, la fila en la que se encuentre ese estado indicará el número de errores encontrados.

Algoritmos de paralelismo de bits. El paralelismo de bits se ha usado extensamente para búsqueda aproximada, teniendo sus mejores resultados para patrones cortos, que en muchos casos son los patrones de interés. La técnica de paralelismo de bits aprovecha la ventaja del paralelismo de bits intrínseco en las operaciones de bits de una palabra de computadora. Es decir, se pueden empaquetar muchos valores en una palabra de

computadora y actualizar todos éstos con una sola operación. Tomando ventaja del paralelismo de bits, el número de operaciones que un algoritmo desempeña se puede reducir por un factor de w, donde w es el número de bits de una palabra de computadora. En las arquitecturas actuales w es igual a 32 ó 64, el incremento de velocidad es muy significante en la práctica.

Algunas notaciones que se utilizan para describir los algoritmos de paralelismo de bits son las siguientes: se usa exponenciación para expresar repeticiones de bits, por ejemplo, $0^3 1$ = 0001. Una secuencia de bits $b_1 \ldots b_l$, se conoce como máscara de bits de longitud l, la cual se almacena dentro de la palabra de computadora de longitud w. Generalmente se utiliza la sintaxis del lenguaje C para las operaciones sobre los bits, es decir, “|” es una operación OR, “&” es una operación AND, “ ^ ” es una operación XOR, “ ~ “ complementa todos los bits, y “<<” (“>>”) mueve los bits a la izquierda (derecha) e ingresa ceros a partir de la derecha (izquierda), por ejemplo, $b_l b_{l-1} \ldots b_2 b_1 \ll 3 = b_{l-3} \ldots b_2 b_1 000$.

Los algoritmos de paralelismo de bits simulan los algoritmos clásicos. Algunos paralelizan el cálculo de la matriz de programación dinámica y algunos paralelizan el cálculo del NFA. La técnica más simple [13], empaqueta cada fila i del NFA en diferentes palabras de computadora R_i, cada estado está representado por un bit. Cada que se lee un carácter del texto, todas las transiciones del autómata se simulan usando operaciones de bits entre las $k+1$ máscaras de bits, las cuales tienen la misma estructura, es decir, el mismo bit está alineado a la misma posición del texto. Para actualizar los valores de R'_i en la posición del texto j teniendo los valores actuales R_i se aplica la siguiente fórmula:

$$\begin{aligned} R'_0 &\longleftarrow ((R_0 \ll 1) \mid 0^{m-1} 1) \,\&\, B[t_j] \\ R'_i &\longleftarrow ((R_i \ll 1) \,\&\, B[t_j] \mid R_{i-1} \mid (R_{i-1} \ll 1) \mid (R'_{i-1} \ll 1) \end{aligned} \quad (1)$$

Donde B es una tabla que almacena una máscara de bits $b_m \ldots b_1$ para cada carácter del patrón. La máscara en $B[c]$ tiene el j^{th} bit activo si $P_j = c$. La búsqueda se inicia con $R_i = 0^{m-i} 1^i$.

Algoritmos de filtrado. La idea de estos algoritmos se basa en el hecho de que es más fácil conocer qué posiciones del texto no pueden contener una coincidencia que conocer cuáles si. Por lo tanto, estos algoritmos descartan áreas que no pueden contener una coincidencia. La técnica de filtrado no es efectiva para niveles de error altos debido a la cantidad de verificaciones a realizar. El algoritmo más rápido para niveles de error bajo está basado en filtrado: si el patrón es dividido en $k+1$ subpatrones, alguna coincidencia aproximada del patrón debe contener al menos un subpatrón sin error, puesto que k errores no pueden cambiar los $k+1$ subpatrones. La búsqueda inicia con una búsqueda exacta multipatrón de los $k+1$ subpatrones y después se verifican las áreas que pueden contener una coincidencia.

2.2 Compresión de texto

La compresión de texto es una técnica para reducir el espacio necesario para el almacenamiento de las colecciones de texto y su transmisión por la red. Existen diversos métodos de compresión, entre los que están los de la familia Ziv-Lempel particularmente LZ78 y LZW, los cuales son muy populares por su buena razón de compresión combinados con un tiempo eficiente de compresión y descompresión. El método de compresión Ziv-Lempel reemplaza subcadenas en el texto por apuntadores a

coincidencias previas de éstas. A continuación se describen a detalle los algoritmos LZ78 y LZW.

LZ78. Este método mantiene un diccionario de cadenas encontradas anteriormente. El diccionario inicialmente está vacío y su tamaño máximo depende de la cantidad de memoria disponible. El codificador genera bloques de dos campos. El primer campo es un apuntador al diccionario, el segundo es el código de un símbolo. Cada bloque corresponde a una cadena de símbolos de entrada, y cada cadena se añade al diccionario después de que el bloque se escribe en la cadena comprimida. El diccionario contiene en la posición cero la cadena vacía. Conforme entran los símbolos y se codifican, las cadenas se añaden al diccionario en las posiciones 1,2, y así sucesivamente.

El proceso de codificación es el siguiente: el primer símbolo se lee y se convierte en una cadena de un solo símbolo. El codificador trata de encontrarla en el diccionario. Si el símbolo se encuentra en el diccionario, se lee el siguiente símbolo y se concatena con el primero para formar una cadena de dos símbolos, que el codificador trata de localizar en el diccionario. Tan pronto como esas cadenas se encuentran en el diccionario, se leen más símbolos y se concatenan a la cadena. En cierto momento la cadena no se encuentra en el diccionario, entonces el codificador la añade al diccionario y genera un bloque con la última coincidencia del diccionario en su primer campo, y el último símbolo de la cadena (el que ha provocado que la búsqueda falle) en el segundo campo.

A continuación se muestra la codificación obtenida por LZ78 para el texto “*abracadabra_*”.

	Texto codificado	Bloque obtenido
1	‘a’	(0,a)
2	‘b’	(0,b)
3	‘r’	(0,r)
4	‘ac’	(1,c)
5	‘ad’	(1,d)
6	‘ab’	(1,b)
7	‘ra’	(3,a)
8	‘_’	(0,_)

Figura 7. Codificación obtenida con el algoritmo de compresión LZ78 para el texto ‘abracadabra’.

LZW. Es una variante popular del LZ78. Este método elimina el segundo campo del bloque LZ78. Un bloque LZW consiste en un apuntador al diccionario. Como resultado, un bloque codifica siempre una cadena de más de un símbolo. El método LZW inicializa el diccionario con todos los símbolos del alfabeto. El caso más común es contar con símbolos de 8 bits, por lo cual las primeras 256 entradas del diccionario (de 0 a 255) están ocupadas antes de que entre dato alguno.

LZW trabaja de la siguiente manera: El codificador introduce los símbolos uno por uno y los acumula en la cadena *I*. Después de introducir cada símbolo y concatenarlo con *I*, se busca la cadena *I* en el diccionario y al encontrarla el proceso continúa. Cuando se añade el siguiente símbolo ‘*x*’ y se produce una falla en la búsqueda, es decir, la cadena *I* está en el diccionario pero no la cadena *I*’x’, el codificador agrega a la cadena comprimida el

apuntador de diccionario que apunta a la cadena *I*, almacena la cadena *I'x'* en la siguiente entrada del diccionario disponible, y por último inicializa la cadena *I* al símbolo '*x*'.

A continuación se muestra la codificación obtenida por LZW para el texto "*abracadabra_*".

0	NULL	99	c	262	da
1	SOH		.	263	abr
	.		.	264	ra_
	.		.		
	.	255	255		
32	SP	256	ab		
	.	257	br		
	.	258	ra		
	.	259	ac		
97	a	260	ca		
98	b	261	ad		

97(a), 98(b), 114(r), 97(a), 99(c), 97(a), 100(d), 256(ab), 258(ra) …

Figura 8. Codificación con el algoritmo de compresión LZW para el texto 'abracadabra'. Sólo se toman los números, no las cadenas entre paréntesis.

Tanto LZ78 cómo LZW logran la compresión si el apuntador toma menos espacio que la cadena reemplazada. Por lo tanto, estos métodos tienen mejores resultados si el archivo de texto en grande (mayor a 10 MB), puesto que hay más probabilidad de que las cadenas apuntadas sean de una longitud mayor.

3 Trabajo Relacionado

En este apartado se mencionan algunos trabajos que se han desarrollado en el ambiente de búsqueda de patrones en texto comprimido. Se describen brevemente las soluciones propuestas así como sus principales características, ventajas y desventajas.

Para búsqueda de patrones en texto comprimido sobresalen dos esquemas. El primero aplica métodos de compresión basados en reemplazar sólo símbolos, tal como la codificación de Fuman [15]. Mediante este esquema se han realizado trabajos que ofrecen una solución eficiente [16] y [17], pero en general la razón de compresión no es buena o su funcionalidad está limitada.

El segundo esquema considera métodos de compresión de la familia Ziv-Lempel. La búsqueda de patrones en texto comprimido con Ziv-Lempel es mucho más compleja, dado que el patrón se puede encontrar en formas diferentes a través del texto comprimido. El primer algoritmo para búsqueda exacta aparece en [18], el cual presenta un algoritmo de búsqueda de patrones para texto comprimido con LZ78 que simula una máquina KMP y determina si el patrón aparece o no en el texto en un tiempo y espacio de

$O(m^2 + n)$. Un algoritmo para LZ77 se presentó en [19], y resuelve el mismo problema en un tiempo de $O(m + n\log^2(u/n))$.

Por otra parte, en [20] se presentó una extensión de [18] que resuelve el problema de búsqueda multipatrón sobre texto comprimido con LZ78/LZW. Este algoritmo se basa en el algoritmo Aho-Corasick[21], y encuentra todas las coincidencias del patrón en un tiempo y espacio de $O(M^2 + n)$, donde M es la suma de las longitudes de los patrones.

Un esquema general para búsqueda de patrones (simples y extendidos) directamente en texto comprimido se presentó en [22], especializándose para algunos formatos en particular (LZ77, LZ78, etc.). Este esquema se basa en paralelismo de bits, con esta técnica se logra empaquetar muchos valores en los bits de una palabra de computadora de w bits y actualizar todos ellos en paralelo. Un trabajo similar se presentó en [23] para texto comprimido con LZW. Un algoritmo basado en el método de Boyer-Moore para búsqueda en texto comprimido con LZ78/LZW se presentó en [24], el cual actualmente es el más rápido para longitudes moderadas del patrón.

El problema de búsqueda aproximada sobre texto comprimido fue tratado por primera vez en [6]. Recientemente, este problema ha sido resuelto para los formatos LZW y LZ78 en un tiempo de $O(mkn + R)$ (donde R es el número de coincidencias encontradas) en el peor de los casos y $O(k^2n+R)$ en el caso promedio, utilizando técnicas de programación dinámica [7] y técnicas de paralelismo de bits [8]. Sin embargo, ambas soluciones son muy lentas. La primera solución práctica a este problema se presenta en [25]. Esta solución trabaja para texto comprimido con LZ78/LZW y se basa en dividir el patrón en k+1 subpatrones y realizar una búsqueda multipatrón de éstos, seguida de una descompresión local y una verificación directa en las áreas de texto candidatas. En este trabajo se presenta una mejora al trabajo realizado en [25], en lugar de realizar una descompresión y luego buscar el patrón en el área descomprimida, se simulan dos autómatas utilizando la técnica de paralelismo de bits que reconozcan la coincidencia con k errores a partir del subpatrón encontrado en la fase anterior, esto hace posible que la verificación sea más rápida y la búsqueda se acelere.

4 Solución propuesta: Búsqueda aproximada en texto comprimido

La mayoría de los algoritmos para búsqueda de patrones sobre texto comprimido toman las ideas de los algoritmos clásicos de búsqueda para texto sin comprimir y las adaptan para trabajar con secuencias de bloques de Ziv-Lempel en vez de una secuencia de caracteres. Las soluciones para búsqueda aproximada de patrones no son la excepción. Existen 4 diferentes esquemas para búsqueda aproximada de patrones [1], dos de las cuales son de interés en este trabajo: (1) Paralelismo de bits, y (2) Filtración. En este trabajo se adapta una técnica de filtración para trabajar sobre texto comprimido.

4.1 Dividir en *k*+1 subpatrones

El primer paso de la búsqueda consiste en dividir el patrón en k+1 subpatrones de la misma longitud, así la longitud de cada subpatrón será $\lfloor (m / (k+1)) \rfloor$. En [1,13] se establece que, bajo el modelo de inserción, eliminación y reemplazo, si el patrón es dividido en k+1 subpatrones contiguos, entonces al menos uno de los subpatrones se encontrará sin errores dentro de cualquier coincidencia con máximo k errores, esto es fácil

notarlo puesto que cada error puede alterar en el peor caso un subpatrón. La búsqueda continúa con la ejecución de una búsqueda multipatrón de los subpatrones resultantes, la cual se explica en la siguiente sección. Cada que se encuentra un subpatrón, se verifica mediante dos autómatas la coincidencia del patrón completo permitiendo *k* diferencias, uno verifica hacia la izquierda y otro hacia la derecha del subpatrón encontrado, estos autómatas se simulan mediante paralelismo de bits. A continuación se explica a detalle la etapa de búsqueda multipatrón y la etapa de verificación.

4.2 Búsqueda multipatrón Boyer-Moore.

El algoritmo multipatrón que se ejecuta en esta fase, es una adaptación del algoritmo Boyer-Moore (BM) monopatrón [24]. El algoritmo BM consiste en alinear el patrón en una ventana de texto y comparar de derecha a izquierda los caracteres de la ventana con los correspondientes al patrón. Si ocurre una desigualdad se calcula un desplazamiento seguro el cual permitirá desplazar la ventana hacia delante del texto sin riesgo de omitir alguna coincidencia. Si se alcanza el inicio de la ventana y no ocurre ninguna desigualdad, entonces se reporta una coincidencia y la ventana se desplaza.

Se han presentado algunos trabajos para adaptar esta idea a texto comprimido con Ziv-Lempel [24]. La figura 9 representa una ventana hipotética de un texto comprimido con LZ78 o con LZW. En el caso de LZ78, El cuadro oscuro representa el carácter explícito *c* del bloque *b* = (*s*,*c*), mientras que las líneas que unen a los cuadros representan los caracteres implícitos, es decir el texto que se obtiene de los bloques previos referenciados(*s*, luego el bloque referenciado por *s*, y así sucesivamente). En el caso de LZW la caja representa el primer carácter del bloque siguiente. Por cada bloque leído se almacena su último carácter, su longitud, el bloque al que referencia y algún otro dato dependiente del algoritmo.

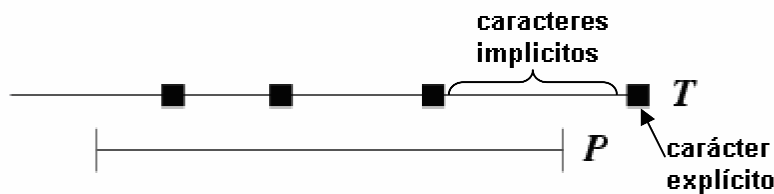


Figura 9. Ventana de un texto comprimido con LZ78 o LZW. Los cuadros oscuros representan los caracteres explícitos al final de cada bloque (o al principio del bloque siguiente en LZW) y las líneas que unen los cuadros representan los caracteres implícitos del bloque.

El algoritmo de búsqueda monopatrón BM lee desde el archivo comprimido tantos bloques como sea necesario para completar la ventana. Aplicar BM puro es costoso debido a la necesidad de acceder a los caracteres “dentro” de los bloques. Un carácter a una distancia *i* del último carácter de un bloque necesita ir *i* bloques atrás en la cadena de referenciamiento. Para evitar esto, es preferible comenzar con los caracteres explícitos dentro de la ventana. Para maximizar los desplazamientos, los caracteres se visitan de derecha izquierda. Para conocer qué desplazamiento es posible hacer al leer cada carácter del bloque se precalcula la siguiente tabla:

$$B(i,c) = \min(\{i\} \cup \{i-j,\ 1 \le j \le i \wedge P_j = c\}) \quad (2)$$

La cual da el máximo desplazamiento seguro dado que en la posición i el carácter en el texto es c.

Al encontrar el primer carácter explícito que permita un desplazamiento mayor que cero, se desplaza la ventana. Si no es así, se comienzan a considerar los caracteres implícitos. La figura 10 muestra el orden en que se consideran los bloques.

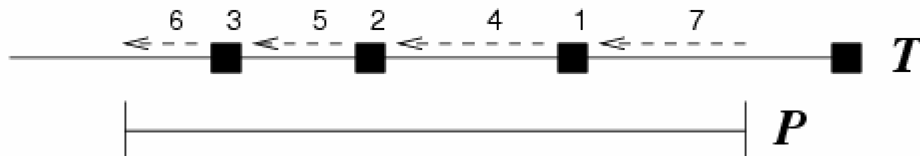


Figura 10. Orden en que se evalúan los bloques. Primero se leen los caracteres explícitos de derecha a izquierda, luego los implícitos de derecha a izquierda dejando al final el último bloque.

Si después de considerar todos los bloques no se obtiene un desplazamiento mayor que cero, se reporta una coincidencia del patrón en la posición de la ventana actual y la ventana se desplaza una posición hacia la derecha del texto.

Para la versión multipatrón, se generaliza la búsqueda de un solo patrón a la búsqueda de varios patrones, es decir, en lugar de buscar un patrón P en el texto comprimido, se buscan r patrones $P^1 \ldots P^r$ simultáneamente. Para conocer el desplazamiento posible para cada carácter leído de un bloque se redefine B de la siguiente manera:

$$B(i,c) = \min(\min(\{i\} \cup \{i-j, 1 \leq j \leq i \wedge P^k_j = c\}), 1 \leq k \leq r) \quad (3)$$

Es decir, dado que el carácter en la posición i es c, se calcula el desplazamiento máximo seguro para cada patrón y se toma el menor de ellos. Con esto se obtiene el desplazamiento máximo seguro para todos los patrones.

Una vez que se encuentra una coincidencia de un subpatrón es necesario realizar una fase de verificación para comprobar si el subpatrón encontrado corresponde a una parte de la coincidencia del patrón buscado. En caso de no ser así, la ventana se desplaza y continúa la búsqueda de otro subpatrón.

4.3 Verificación

Finalmente, hay que realizar el procedimiento de verificación. En la fase anterior se encontró uno de los $k+1$ subpatrones de P, es decir, $P = P_1 F P_2$ y F es el subpatrón que se encontró. Se precalcula un autómata de paralelismo de bits para P_1 (invertido) y otro para P_2 para cada uno de los k+1 subpatrones, y se utilizan los autómatas correspondientes al subpatron encontrado.

Se inicia el reconocimiento con P_1, pues resulta menos costoso en tiempo verificar los bloques hacia la izquierda dado que estos han sido desenrollados. Se inicia con el bloque en donde comienza el subpatrón *F*, *e*n la figura 11 el bloque *j* representa el bloque donde inicia el subpatrón, el cual está "contenido" dentro del bloque *jj. A partir del bloque j* se obtienen todos los caracteres explícitos de los bloques referenciados por *j* hasta encontrar un bloque de longitud menor o igual a 1, luego se obtienen todos los caracteres referenciados por el bloque *jj*-1, *jj*-2 y así sucesivamente, alimentando el autómata con los caracteres que se van obteniendo. El proceso se detiene cuando el autómata muere.

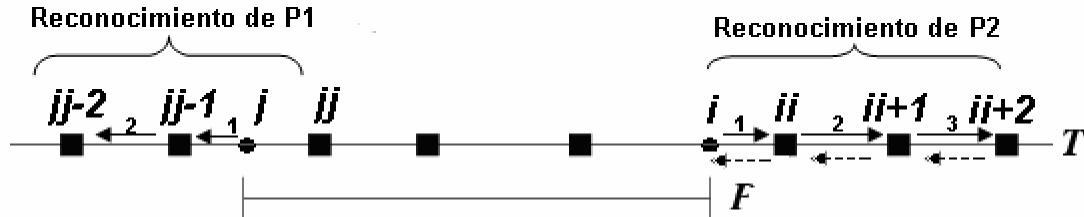


Figura 11. Orden en que se obtienen los caracteres para alimentar al autómata en el proceso de verificación.

Si el número de errores encontrados en este proceso $k' > k$, se detiene el proceso y continua la búsqueda de otro subpatrón. Si $k' \leq k$ se reconfigura el autómata P_2 para que permita $k'' = k - k'$ errores.

Para el reconocimiento de P_2 se inicia con el bloque en donde finaliza *F* . En la figura 11 este bloque esta representado por *i,* el cual está "contenido" dentro del bloque *ii*. Ahora, obtener los caracteres es más difícil, pues hay que ir hacia atrás en la cadena de referenciamiento a partir del bloque *ii* hasta encontrar el bloque *i*, almacenando en un arreglo los caracteres explícitos de los bloques que se van recorriendo. Una vez que se encontró el bloque *i*, se alimenta el autómata con los caracteres almacenados iniciando con el último carácter obtenido. Si los caracteres obtenidos no son suficientes para llegar a un estado final del autómata, se lee un nuevo bloque *ii* = *ii* +1 y se procesa de igual manera, es decir, se van almacenando los caracteres explícitos de los bloques a los que referencia *ii*, esta vez mientras que la longitud del bloque referenciado sea mayor que 1. Al llegar a este bloque se alimenta el autómata con los caracteres almacenados iniciando con el último carácter obtenido. El proceso continua hasta que el autómata muere.

Si al finalizar la verificación el número total de errores encontrados es menor o igual a *k*, se reporta una coincidencia del patrón completo.

5 Resultados experimentales

Para la ejecución de los experimentos que se realizaron en este trabajo, se utilizó una computadora exclusivamente para este propósito. Las características del hardware y software son: Procesador Intel Pentium IV a 1300 MHz, 256 MB de RAM, 80 GB de disco duro, Sistema Operativo Linux distribución RedHat 8.0. Para probar el algoritmo se comprimió un archivo de texto de 85 MB con el comando *compress* de Unix logrando una compresión del 58%. El archivo comprimido es un conjunto de títulos y/o resúmenes de

270 revistas médicas coleccionados en un periodo de 5 años (1987-1991). Los patrones se escogieron de forma aleatoria y se experimentó con longitudes del patrón de 10 hasta 30 caracteres, y con *k* igual a 0, 1, 2 y 3, ya que son los casos más comunes.

El algoritmo de búsqueda aproximada se implementó en un software llamado *alzgrep* (*lzgrep* extendido con búsqueda aproximada), los resultados que se obtuvieron se compararon con *agrep* y *nrgrep*, que son en la actualidad las mejores herramientas de búsqueda que permiten realizar búsqueda aproximada pero que deben descomprimir el archivo antes de efectuar la búsqueda. Por esta razón, a los tiempos de búsqueda que lograban se les sumó el tiempo necesario para llevar a cabo la descompresión del archivo.

La figura 12 muestra los tiempos obtenidos en la búsqueda de patrones de longitud menor a 15 y entre 15 y 30. Como se puede observar, el algoritmo propuesto en este trabajo es hasta 3 veces más rápido que el mejor software que primero descomprime el archivo y realiza la búsqueda después. Los patrones con los que se experimentó y los datos obtenidos se muestran en el anexo A.

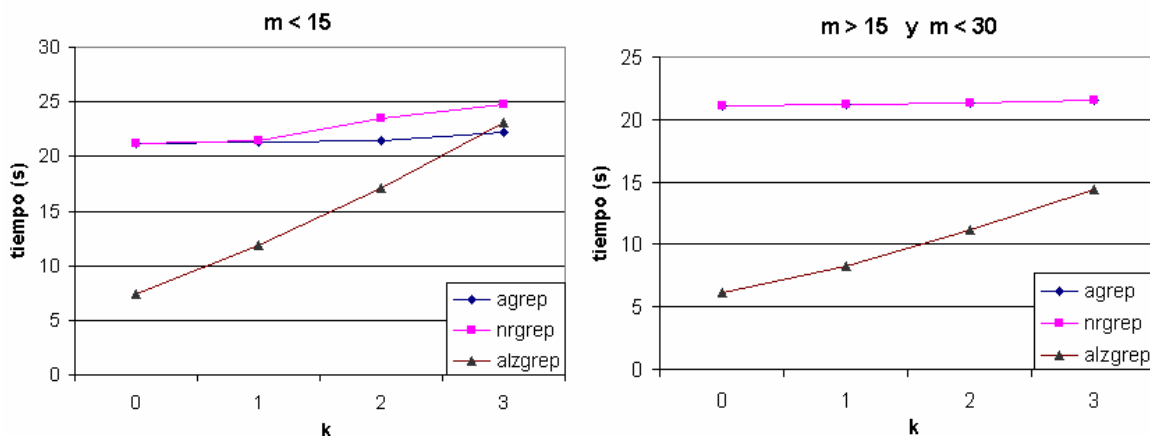


Figura 12. Tiempo de búsqueda en segundos para los diferentes algoritmos sobre un archivo de texto, para *m*<15 y *m* entre 15 y 30 y *k* = 0, 1, 2 y3.

La figura 13 muestra otra perspectiva del comportamiento del algoritmo en comparación con *agrep* y *nrgrep*. Las gráficas muestran cómo el algoritmo obtiene sus mejores tiempos para patrones largos (mayor que 15) y su eficiencia disminuye cuando el patrón es corto y k aumenta, esto debido a que los subpatrones resultantes tienen una longitud pequeña y el algoritmo encuentra coincidencias de los subpatrones en distancias cortas de texto, por lo que tiene que hacer demasiadas verificaciones.

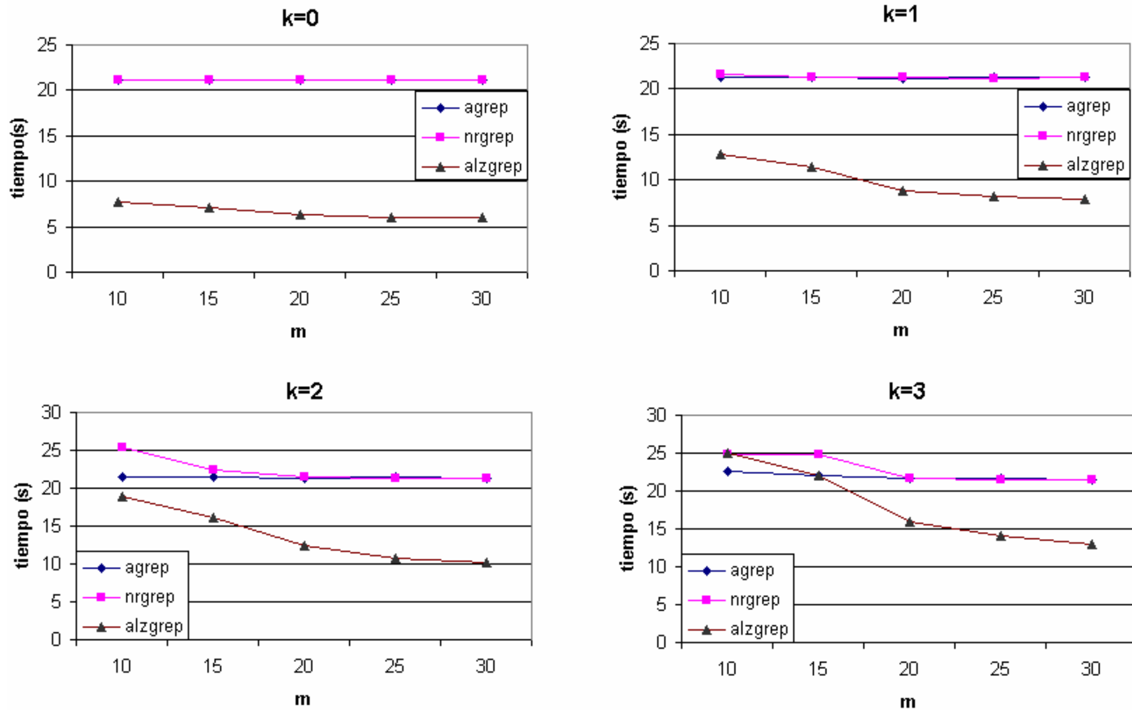


Figura 13. Tiempo de búsqueda en segundos para los diferentes algoritmos sobre un archivo de texto, para k = 0, 1, 2 y 3 y m = 10, 15, 20, 25 y 30.

6 Conclusiones

En este trabajo se ha presentado una solución al problema de búsqueda aproximada directamente sobre texto comprimido con LZ78 y LZW. Con los resultados que se obtuvieron de los experimentos realizados, se puede concluir que con el algoritmo propuesto se puede efectuar la búsqueda aproximada de patrones simples hasta 3 veces más rápido que con la mejor herramienta disponible para búsqueda aproximada, la cual primero tiene que realizar una descompresión del archivo y luego realizar la búsqueda. Para llevar a cabo los experimentos, se implementó una herramienta la cual de denominó *alzgrep*, y que posteriormente estará públicamente disponible. Contar con una herramienta de este tipo es muy importante ya que además de la ventaja que se obtiene en el tiempo necesario para ejecutar la búsqueda, se ahorra espacio para almacenar la colección, y tiempo de transmisión en la red.

Referencias

1. G. Navarro. A guide tour approximate string matching. ACM Computing Surveys, 2000.
2. V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. Problems of Information Transmission, 1:8-17, 1965.
3. T. Bell, J. Cleary, and I. Witten. Text compression. Prentice Hall, 1990.
4. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. IEEE Trans. Inf. Theory, 24:530-536, 1978.
5. T. A. Welch. A technique for high performance data compression. IEEE Computer Magazine, 17(6):8-19, June 1984.

6. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In Proc. DCC'92, pages 279-288, 1992.
7. J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching Over Ziv-Lempel compressed text. In Proc. CPM'2000, LNCS 1848, pages 195-209, 2000.
8. T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Bit-parallel approach to approximate string matching in compressed texts. In Proc. SPIRE'2000, pages 221-228. IEEE CS Press, 2000.
9. R. S. Boyer and J. S. Moore. A fast string searching algorithm. Communications of the ACM, 20(10):772, 1977.
10. S. Wu and U. Manber. Agrep- a fast approximate pattern-matching tool. In Proc. USENIX Technical Conference, pages 153-162, 1992.
11. G. Navarro, NR-grep: a fast and flexible pattern-matching tool. Software-Practice & Experience, v.31 n.13, p.1265-1312, November 10, 2001.
12. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. SIAMS Journal on Computing, 6(1):323-350, 1977.
13. S. Wu and U. Manber. Fast text searching allowing errors. Communications of the ACM, 35(10):83-91, 1992.
14. R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In N. J. Belkin and C. J. Van Rijsbergen, Editors, 12*th,* pages 168-175, 1989.
15. D. Huffman. A method for the construction of minimum-redundancy codes. Proc. Of The I. R. E., 40(9):1090-1101, 1952.
16. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. ACM TOIS, 15(2):124-136, 1997.
17. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. ACM TOIS, 18(2):113-139, 2000.
18. A. Amir, G. Benson, and M. Farach. Let Sleeping Files Lie: Pattern Matching In Z-compressed Files. J. Of Comp. And Sys. Sciences, 52(2):299-307, 1996.
19. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. Algoritmica, 20:388-404, 1998.
20. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In Proc. DCC'98, pages 103-112, 1998.
21. A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. Comm. Of the ACM, 18(6):333-340, June 1975.
22. G. Navarro and Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In Proc. CPM'99, LNCS 1645, pages 14-36, 1999.
23. T. Kida, M. Takeda, A. Shinohara, M. Miyasaki, and S. Arikawa. Shift-and approach to pattern matching in LZW compressed text. In Proc. CPM'99, LNCS 1645, pages 1-13, 1999.
24. G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In Proc. CPM'2000, LNCS, 2000.
25. G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In Proc. 11th IEEE Data Compression Conference (DCC'01), pages 459-468, 2001.

Anexo A.

En la siguiente tabla se muestran los patrones con los cuales se llevó a cabo los experimentos, así como la longitud de cada uno de ellos y el tiempo necesario para encontrar todas las coincidencias del patrón en el texto con cada uno de los algoritmos.

m	Patrón a buscar	agrep				nrgrep				alzgrep			
	k =	0	1	2	3	0	1	2	3	0	1	2	3
10	Associated	21.16	21.34	21.6	23.3	21.2	22.08	29.75	24.05	7.99	13.32	19.82	27.19
10	Hemorrhage	21.14	21.29	21.48	22.89	21.14	21.41	24.63	24.18	7.82	13.2	19.85	26.86
12	Pneumathorax	21.12	21.21	21.28	21.67	21.18	21.34	21.89	26.18	7.53	11.73	16.92	21.06
13	Resuscitation	21.13	21.46	21.8	22.21	21.14	21.46	25.07	25.62	7.31	12.23	17.84	23.07
13	Ethchlorvynol	21.11	21.26	21.25	21.86	21.11	21.26	21.47	24.92	7.08	11.19	16.23	20.75
13	Approximately	21.15	21.22	21.46	21.79	21.16	21.29	21.77	25.2	7.3	11.83	16.58	20.85
15	Cerebral anoxia	21.13	21.24	21.34	22.25	21.12	21.27	21.83	25.62	6.9	11.02	15.3	22.85
15	Cerebral artery	21.14	21.21	21.33	22.12	21.12	21.3	21.82	22.58	6.98	10.69	14.76	22.6
19	Electrocardiography	21.1	21.18	21.32	21.64	21.1	21.24	21.67	21.73	6.34	9.19	12.53	17.88
20	Immunohistochemistry	21.1	21.18	21.3	21.57	21.08	21.26	21.39	21.72	6.43	8.6	12.45	15.03
20	Globotriosylceramide	21.11	21.18	21.44	21.55	21.09	21.29	21.3	21.52	6.49	8.73	12.53	14.77
21	Possible appendicitis	21.01	21.18	21.35	21.55	21.09	21.21	21.36	21.87	6.32	8.86	12	16.34
23	Electroencephalographic	21.1	21.16	21.3	21.51	21.09	21.18	21.39	21.55	6.06	8.63	11.19	15.22
25	Immunodeficiency syndrome	21.09	21.27	21.43	21.62	21.08	21.17	21.36	21.62	5.99	7.92	10.33	13.76
26	Magnetic resonante imaging	21.1	21.28	21.47	21.67	21.09	21.37	21.25	21.55	6.06	8.29	11.34	13.78
27	Macroscopic and microscopic	21.08	21.24	21.37	21.54	21.06	21.15	21.25	21.34	5.9	7.77	10.13	13.56
28	Oromandibular reconstruction	21.09	21.24	21.37	21.54	21.07	21.31	21.29	21.41	6.07	7.84	10.52	12.64
28	Electrohydraulic ventricular	21.1	21.22	21.33	21.49	21.07	21.32	21.29	21.53	6.1	7.65	10.12	12.82
29	Possible mode of transmissión	21.1	21.23	21.38	21.52	21.07	21.34	21.26	21.42	6.09	8.12	10.37	13.31
30	Abdominal and gastrointestinal	21.09	21.25	21.36	21.49	21.08	21.27	21.25	21.52	6.01	7.86	9.88	13.44

Tiempo de la búsqueda en segundos