Mejorando un Algoritmo para Búsqueda Aproximada

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,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 en texto comprimido trata de encontrar todas las coincidencias de un patrón en un texto comprimido, sin necesidad de descomprimirlo y teniendo en consideración que la coincidencia del patrón con el texto puede tener un número limitado de diferencias. Este problema tiene diversas aplicaciones en recuperación de información, biología computacional y procesamiento de señales, entre otras. Una de las mejores soluciones a este problema es realizar una búsqueda multipatrón de un conjunto de piezas del patrón, seguida de una descompresión local y una verificación directa en las áreas descomprimidas. En este trabajo se presenta una mejora a esta solución en la parte de verificación, en lugar de realizar una descompresión y buscar el patrón se construyen autómatas de paralelismo de bits que reconozcan el patrón. Con esto se logra realizar todo el proceso de búsqueda sin necesidad de descomprimir el archivo de texto en ningún momento además de obtener tiempos competitivos con los mejores algoritmos clásicos de búsqueda aproximada disponibles actualmente.

Palabras clave. 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 World Wide Web, ha propiciado un considerable aumento de información textual disponible en diversos medios, tales como bibliotecas digitales y bases de datos de texto. La búsqueda de patrones permite acceder rápidamente a zonas de interés dentro de gran cantidad de información. La búsqueda de patrones en 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, propiciado al capturar los documentos o a partir de software de reconocimiento óptico, por lo que es 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], la cual tiene aplicaciones en recuperación de información, biología computacional y procesamiento de señales, por nombrar algunas. La búsqueda aproximada de patrones se define como: Dado un texto T de longitud u, un patrón P de longitud m, y un número máximo de errores permitidos k, encontrar todas las coincidencias de P cuya máxima distancia de edición al patrón es k. La distancia de edición entre dos cadenas x y y es el número mínimo de operaciones de edición necesarias para transformar x en y. Las operaciones de edición permitidas son inserción, eliminación o reemplazos de un carácter.

Una parte del problema de búsqueda de patrones está relacionada con la compresión de texto [2], la cual 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 [3] y LZW [4], los cuales son muy populares por su buena razón de compresión combinado con un tiempo eficiente de compresión y descompresión.

El problema de búsqueda de patrones en texto comprimido se define 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.

Una de las mejores soluciones al problema de búsqueda aproximada en texto comprimido se propuso en [5], y consiste en dividir el patrón en k+1 subpatrones y realizar una búsqueda exacta multipatrón con el conjunto de subpatrones resultantes utilizando el algoritmo de Boyer-Moore [6], cada vez que se encuentra un subpatrón se verifica la existencia del patrón completo descomprimiendo un área de texto alrededor del subpatrón y ejecutando un algoritmo clásico para búsqueda aproximada en el área de texto descomprimida. En este artículo se presenta una mejora a esta solución en la parte de verificación, en lugar de realizar una descompresión y buscar el patrón, se construyen dos autómatas de paralelismo de bits para que reconozcan el patrón completo, uno reconoce hacia la izquierda del subpatrón encontrado y otro hacia la derecha, de tal forma que la suma de los errores encontrados en cada uno de los autómatas sea menor o igual a k. Con esto se logra realizar todo el proceso de búsqueda sin necesidad de descomprimir el archivo de texto en ningún momento además de obtener tiempos de búsqueda competitivos con los mejores algoritmos clásicos de búsqueda aproximada que primero descomprimen el archivo y después realizan la búsqueda.

La notación que se utilizará durante las siguientes secciones son: Una cadena x es una secuencia de caracteres sobre un alfabeto finito $\sum$ de tamaño σ. La longitud de la

cadena se expresa como $|x|$. Una subcadena se expresará como $x_{i...j}$, lo que significa tomar del i^{th} carácter al j^{th} carácter de x. Las letras P y T representaran al patrón y al texto, de longitud m y u respectivamente, Z representa el texto comprimido de longitud n.

Algunas notaciones que se utilizarán 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. Se utiliza la sintaxis del lenguaje de programación 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, $b_l b_{l-1} \ldots b_2 b_1 << 3 = b_{l-3} \ldots b_2 b_1 000$.

2 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 resolver el problema de búsqueda aproximada de patrones en texto sobresalen cuatro esquemas. La primera de ellas es la programación dinámica, la cual consiste en calcular una matriz $E[i, j]$ que representa el número mínimo de errores permitidos k para hacer coincidir $p_{1...i}$ ($i \leq m$) con $t_{j'...j}$ ($j' \leq j \leq u$). El algoritmo trabaja de la siguiente forma: 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. Las posiciones de texto en donde $E[m, j] \leq k$ se reportan como coincidencias del patrón.

Otro esquema es modelar la búsqueda aproximada con un autómata finito no determinístico (AFN). Por ejemplo, si se tienen $k = 2$ errores tal como se muestra en la figura 1, 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.

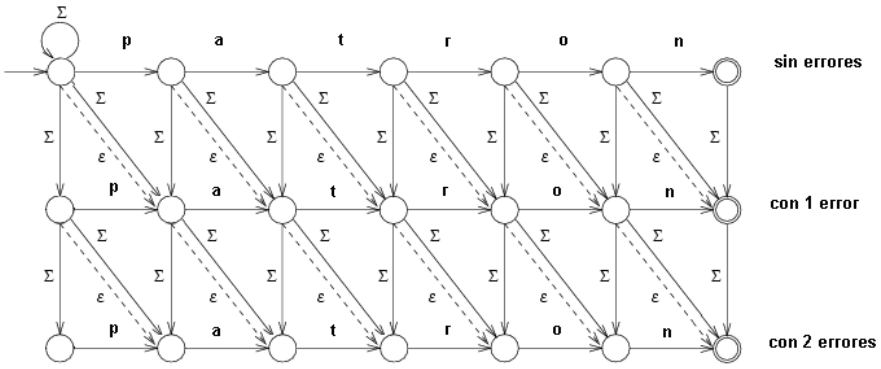

Fig. 1. Autómata finito no determinístico para búsqueda aproximada de la cadena "patron" permitiendo 2 errores

Cada vez que se lee un carácter del texto el autómata cambia de estado. Las flechas horizontales, verticales, diagonales sólidas y diagonales punteadas, representan la coincidencia, inserción, reemplazo o eliminación de un carácter respectivamente. El ciclo en el estado inicial permite que una coincidencia ocurra en cualquier parte del texto. El autómata indica una coincidencia cuando algún estado final está activo, la fila en la que se encuentre ese estado indicará el número de errores encontrados.

Otra técnica que se utiliza para búsqueda aproximada es la de paralelismo de bits. Generalmente 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 AFN. La técnica más simple [7] enfocada a paralelizar el cálculo del AFN, empaqueta cada fila *i* del AFN en diferentes palabras de computadora R_i, donde cada estado está representado por un bit. Cada vez 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 propuesta en [8]:

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

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 $R_i = 0^{m-i}\,1^i$. En la fórmula, R'_i expresa las flechas horizontales, verticales, diagonales sólidas y diagonales punteadas de la figura 1 respectivamente.

El cuarto esquema son los algoritmos de filtrado, los cuales se basan en el hecho de que es más fácil conocer qué posiciones del texto no pueden contener una

coincidencia que conocer cuáles sí. Por lo tanto, estos algoritmos descartan áreas que no pueden contener una coincidencia.

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 Huffman[9]. Bajo este esquema se han realizado trabajos que ofrecen una solución eficiente [10] y [11], pero en general la razón de compresión no es buena o su funcionalidad está limitada (sólo a búsquedas en lenguaje natural o búsquedas de patrones simples).

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 [12], el cual trabaja con texto comprimido con LZ78 y sólo 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 [13], y resuelve el mismo problema en un tiempo de $O(m + n\log^2(u/n))$.

Por otra parte, en [14] se presentó una extensión de [12] a búsqueda multipatrón sobre texto comprimido con LZ78/LZW. Este algoritmo se basa en el algoritmo Aho-Corasick[15], 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 [16], 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 [17] 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 [18], el cual actualmente es el más rápido para longitudes moderadas del patrón.

El problema de búsqueda aproximada sobre texto comprimido fue propuesto por primera vez en [19]. En [20] este problema se resolvió 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. Otra técnica que se utiliza para búsqueda aproximada es la de paralelismo de bits, en [21] se utilizó esta técnica lograron un tiempo en el peor de los casos de $O(nmk^3/w)$. Sin embargo, las soluciones presentadas tanto en [20] como en [21] son muy lentas, por lo que en este trabajo se presenta otra solución utilizando la simulación de un autómata con paralelismo de bits, mejorando el proceso de búsqueda.

En [5] se presentó la primera solución práctica al problema de búsqueda aproximada utilizando algoritmos de filtrado. 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. Sin embargo, la técnica de filtrado no es efectiva para niveles de error altos debido a la cantidad de verificaciones a realizar. En este trabajo se presenta una mejora al trabajo realizado en [5], en la cual en lugar de realizar una descompresión y luego buscar el patrón en

el área descomprimida, se simulan dos autómatas similares al de la figura 1 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.

3. Algoritmos de compresión LZ78 y LZW

Los algoritmos de compresión de la familia Ziv-Lempel reemplazan subcadenas en el texto por apuntadores a coincidencias previas de éstas. Destacan los algoritmos LZ78 y LZW los cuales se usan en este trabajo y se explican a detalle.

El algoritmo de compresión LZ78 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.

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. Tanto LZ78 como LZW logran la compresión si el apuntador toma menos espacio que la cadena reemplazada.

4 Mejorando el algoritmo para 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. De los esquemas expuestos anteriormente para búsqueda aproximada, tres son de interés en este trabajo: 1) Filtración, 2) basados en autómatas y 3) paralelismo de bits. En este trabajo se adaptan estos tres esquemas para trabajar sobre texto comprimido. El algoritmo se divide en tres fases: 1) dividir el patrón en k+1 subpatrones, 2) realizar una búsqueda multipatrón 3) verificar la coincidencia del patrón completo. Las dos primeras partes se retoman de lo propuesto en [22], y se presenta una mejor solución en la parte de verificación. A continuación se explica a detalle cada una de estas fases.

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 $m / (k+1)$. En [1] y [22] 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 vez 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 usando paralelismo de bits. A continuación se explican 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 [6]. 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 [18]. La figura 2 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 el último carácter, su longitud, el bloque al que referencia y algún otro dato dependiente del algoritmo.

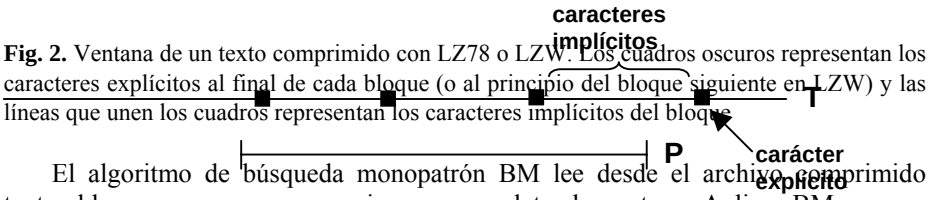

Fig. 2. Ventana de un texto comprimido con LZ78 o LZW. Los cuadros oscuros representan los caracteres explícitos al final de cada bloque (o al principio del bloque siguiente en LZW) y las líneas que unen los cuadros representan los caracteres implícitos del bloque.

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 a izquierda. Para conocer qué desplazamiento es posible hacer al leer cada carácter del bloque, se precalcula la siguiente tabla:

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

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 3 muestra el orden en que se consideran los bloques.

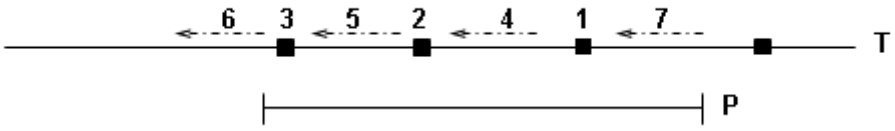

Fig. 3. Orden en que se evalúan los bloques. Primero se leen los caracteres explícitos de derecha a izquierda, luego los implícitos de derecha a izquierda dejando al final el último bloque

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:

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

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 subpatrón encontrado.

Se inicia el reconocimiento con P_1, pues resulta menos costoso en tiempo verificar los bloques hacia la izquierda dado que los caracteres se obtienen en el orden adecuado para alimentar al autómata. EL autómata si simula por medio de la formula 1 expresa anteriormente. Se inicia con el bloque en donde comienza el subpatrón *F*. En la figura 4 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, o cuando el número de errores encontrados en mayor que *k*.

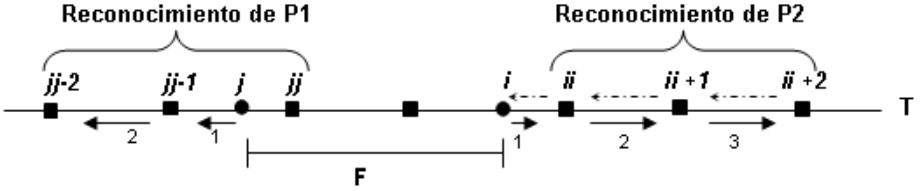

Fig. 4. Orden en que se obtienen los caracteres para alimentar al autómata en el proceso de verificación

Si el número de errores encontrados en este proceso *k*' es mayor que *k*, se continúa la búsqueda de otro subpatrón. Si $k' \le k$ 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 4 este bloque está 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 continúa hasta que el autómata muere ó se encuentra una coincidencia del patrón.

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 con las características del hardware y software siguientes: 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 10 MB con el comando *compress* de Unix logrando una compresión del 58%. El archivo comprimido es un conjunto parcial de títulos y/o resúmenes de 270 revistas médicas coleccionados en un periodo de 5 años (1987-1991)[1]. Los patrones se escogieron de forma aleatoria y se experimentó con longitudes del patrón de 15 hasta 30 caracteres, y con *k* igual a 1, 2, 3 y 4, ya que son los casos más comunes.

El algoritmo de búsqueda aproximada se implementó en un software al que se llamó *alzgrep* (*lzgrep* extendido con búsqueda aproximada), los resultados que se obtuvieron se compararon contra el algoritmo original al que denominamos PP-BM, y contra *agrep* [8] y *nrgrep* [22], 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. La figura 5 muestra los tiempos obtenidos por los algoritmos de búsqueda aproximada.

[1] La colección se puede obtener visitando el siguiente enlace http://trec.nist.gov/

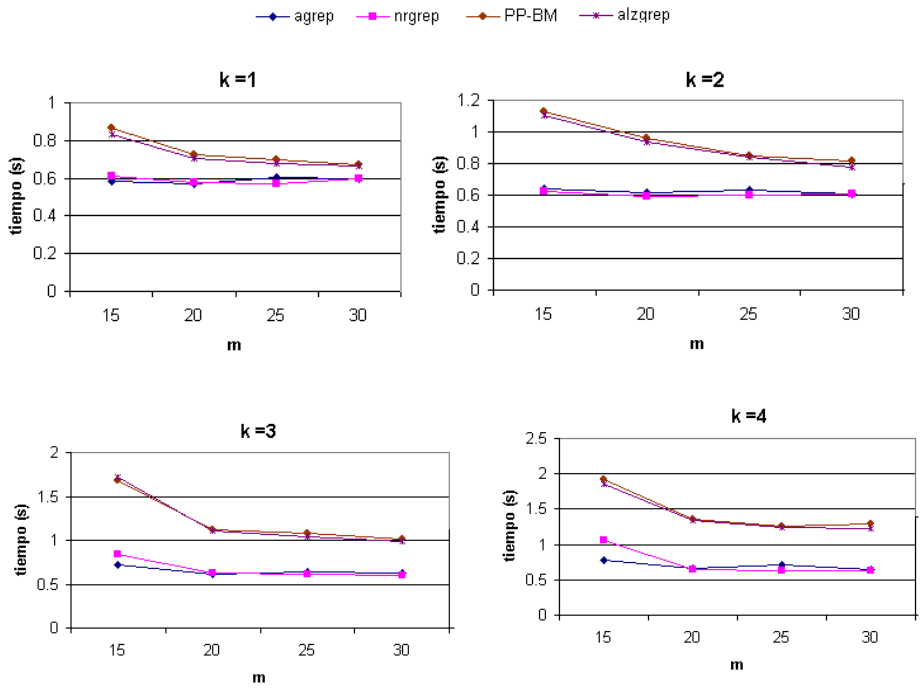

Fig. 5. Tiempo de búsqueda en segundos para los diferentes algoritmos sobre un archivo de texto, para $k = 0, 1, 2$ y 3 y $m = 10, 15, 20, 25$ y 30

Las gráficas muestran cómo el algoritmo obtiene sus mejores tiempos para patrones largos (mayor que 20) y su eficiencia disminuye cuando el patrón es corto y k aumenta. Esto se debe 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.

6 Conclusiones

En este trabajo se ha presentado una mejora a una solución al problema de búsqueda aproximada directa en 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 en texto comprimido con tiempos competitivos a los realizados por los mejores algoritmos para búsqueda aproximada actualmente disponibles. Además que no es necesario descomprimir el archivo de texto en ningún momento del proceso de búsqueda, lo que permite realizar la búsqueda sobre el texto comprimido sin ninguna herramienta adicional de descompresión. Para llevar a cabo los experimentos, se implementó una

herramienta la cual de denominó *alzgrep*, y que posteriormente estará públicamente disponible.

Referencias

1. G. Navarro. A guided tour approximate string matching. ACM Computing Surveys, 2000.
2. T. Bell, J. Cleary, and I. Witten. Text compression. Prentice Hall, 1990.
3. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. IEEE Trans. Inf. Theory, 24:530-536, 1978.
4. T. A. Welch. A technique for high performance data compression. IEEE Computer Magazine, 17(6):8-19, June 1984.
5 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.
6. R. S. Boyer and J. S. Moore. A fast string searching algorithm. Communications of the ACM, 20(10):772, 1977.
7. S. Wu and U. Manber. Fast text searching allowing errors. Communications of the ACM, 35(10):83-91, 1992.
8. S. Wu and U. Manber. agrep- a fast approximate pattern-matching tool. In Proc. USENIX Technical Conference, pages 153-162, 1992.
9. D. Huffman. A method for the construction of minimum-redundancy codes. Proc. Of The I. R. E., 40(9):1090-1101, 1952.
10. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. ACM TOIS, 15(2):124-136, 1997.
11. 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.
12. 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.
13. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. Algoritmica, 20:388-404, 1998.
14. 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.
15. A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. Comm. Of the ACM, 18(6):333-340, June 1975.
16. 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.
17. 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.
18. G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In Proc. CPM'2000, LNCS, 2000.
19. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In Proc. DCC'92, pages 279-288, 1992.
20. 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.
21. 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.

22. G. Navarro, NR-grep: a fast and flexible pattern-matching tool. Software-Practice & Experience, 31(13). p. 1265-1312, 2001.