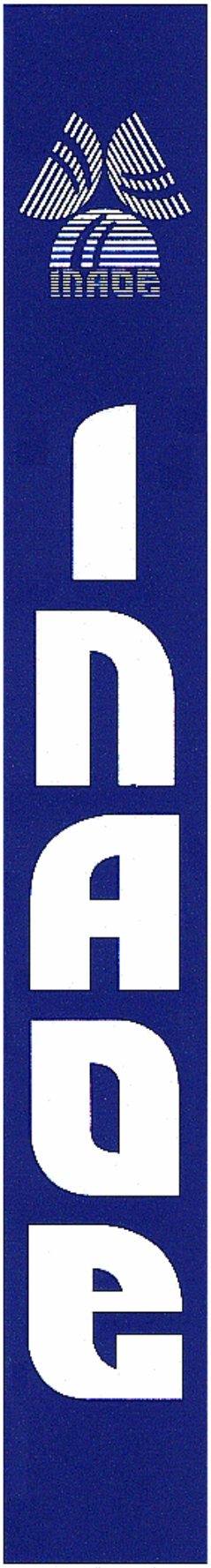


Diseño e Implementación de un Sistema de Búsqueda en una Colección de Texto Comprimido

Por

Carlos Avendaño Pérez

Tesis sometida como requisito parcial para obtener el grado de **Maestro en Ciencias** en la especialidad de **Ciencias Computacionales** en el Instituto Nacional de Astrofísica Óptica y electrónica.

Asesor:

Dra. Claudia Feregrino Uribe, Coordinación de Ciencias Computacionales. Instituto Nacional de Astrofísica, Óptica y Electrónica.

Co-Asesor:

Dr. Gonzalo Navarro Badino, Departamento de Ciencias de la Computación, Universidad de Chile.

Tonantzintla, Puebla. 2004

Resumen

En este trabajo de tesis se trata el problema de búsqueda de patrones en texto comprimido con LZW. Particularmente se abordaran dos tipos de búsqueda: búsqueda aproximada de patrones simples y búsqueda exacta de expresiones regulares.

El primer algoritmo que se expone es el de búsqueda aproximada de patrones simples. El algoritmo consiste de una combinación de técnicas de búsqueda aproximada de patrones que se adaptan para trabajar con archivos comprimidos con LZW. El algoritmo que se expone es una mejora al trabajo realizado por G. Navarro et al. donde utilizan algoritmos del tipo Boyer-Moore para realizar la búsqueda.

El segundo algoritmo que se expone es el de búsqueda de expresiones regulares. El algoritmo consiste en simular un autómata finito no determinístico por medio de la técnica de paralelismo de bits. Al igual que en la búsqueda anterior se retoman algoritmos del tipo Boyer-Moore para realizar la búsqueda.

Para comprobar la eficiencia práctica de los algoritmos expuestos, lo cual pocos trabajos logran, se implementó una herramienta de búsqueda tipo *grep*. Esta herramienta se denominó *eralzgrep* y posteriormente estará disponible públicamente. Esta herramienta es la primera en su tipo, pues hasta antes de este trabajo no existía ninguna herramienta que realizara estos dos tipos de búsqueda con la capacidad de ejecutar todo el proceso de búsqueda sin descomprimir el texto.

Agradecimientos

Dedicatoria

Contenido

Lista de Figuras

Lista de tablas

Capítulo 1

Introducción

El problema de búsqueda de patrones trata de recuperar segmentos de texto con alguna propiedad específica dentro de una colección grande de documentos. Este tipo de búsqueda es útil para aplicaciones como biología computacional, recuperación de información y procesamiento de señales, entre otras. Cada sistema de búsqueda permite la especificación de diversos tipos de patrones, los cuales van desde un rango muy simple (por ejemplo palabras) a más complejos (tal como expresiones regulares). En general, entre más extenso es el conjunto de patrones permitidos, más amplia es la consulta que el usuario puede formular y más compleja es la implementación de la búsqueda.

1.1 Planteamiento del problema

El problema de búsqueda de patrones tiene referencias desde los años sesenta [1], [2] y [3], desde esos años hasta ahora se han implementado diversas herramientas de búsqueda [4] y [5], las cuales han logrado un desempeño aceptable. Sin embargo, debido a la evolución tecnológica en el área de la informática y el amplio uso de bibliotecas digitales, sistemas de automatización de oficinas, bases de datos de documentos y la World Wide Web, se ha propiciado un considerable aumento de

información textual disponible en línea, la cual crece día con día de manera considerable. Por lo tanto es necesario mantener comprimida la información para ahorrar espacio de almacenamiento. Por ende, los sistemas de búsqueda de patrones tienen que realizar la búsqueda en texto comprimido y además encontrar la información relevante para el usuario en un tiempo aceptable, hecho que hasta ahora sólo algunos sistemas han logrado realizar parcialmente [6].

En este trabajo de tesis se trata el problema de búsqueda secuencial en texto comprimido. La búsqueda secuencial se refiere al hecho de que el texto no ha sido preprocesado previamente para llevar a cabo la búsqueda, lo cual se presenta frecuentemente cuando no existen las condiciones ya sea de tiempo o espacio para llevar a cabo un preprocesamiento previo del texto (por ejemplo, indexación). Particularmente se abordaran dos tipos de búsqueda: búsqueda exacta de expresiones regulares y búsqueda aproximada de patrones simples.

La búsqueda de expresiones regulares permite especificar un conjunto más amplio de patrones de búsqueda, permitiéndole al usuario mayor flexibilidad al momento de ingresar el patrón de búsqueda. El problema de búsqueda directa de expresiones regulares en texto comprimido se puede definir de la siguiente manera:

> Dado un patrón P que es una expresión regular, un texto $T = t_1...t_u$, que es una secuencia de caracteres sobre el alfabeto finito $\sum$ y cuyo texto comprimido correspondiente es $Z = z_1 \ldots z_n$, encontrar todas las subcadenas de T que correspondan al lenguaje generado por P utilizando solamente Z.

Por otro lado, la búsqueda aproximada de patrones simples recupera todas las palabras en el texto que sean similares a un patrón dado. El concepto de similaridad puede ser definido de varias maneras. El concepto general es que el patrón o el texto

pueden tener errores y la consulta deberá recuperar los documentos que contengan la palabra dada y sus variantes erróneas. El modelo más utilizado para la similaridad entre palabras es la distancia de edición que se explica posteriormente. Al igual que en el caso anterior, la búsqueda se debe efectuar sobre texto comprimido y se define como:

> Dado un patrón $P = p_1 \ldots p_m$ y un texto $T = t_1 \ldots t_u$, ambas secuencias de caracteres sobre el alfabeto finito $\sum$ cuyo texto comprimido correspondiente es $Z = z_1 \ldots z_n$, y un número k de errores permitidos, encontrar el conjunto $\{|xP'|, T = xP'y$ y $ed(P,P') \leq k\}$, donde $ed(P,P')$ es la distancia de edición.

Un excelente ejemplo donde el problema de búsqueda de patrones y la compresión de texto se combinan son las bases de datos de texto, aquí el texto debe estar comprimido para ahorrar espacio de almacenamiento y tiempo de transmisión en la red y, al mismo tiempo se debe poder realizar búsquedas eficientes sobre ellas.

Para comprimir las colecciones de texto se ha elegido el método de compresión LZW, el cual es uno de los más populares por su buena razón de compresión combinado con un tiempo eficiente de compresión y descompresión.

1.2 Motivación

Debido al tamaño de las colecciones de texto de hoy en día y a su heterogeneidad, controlar la calidad de éstas es prácticamente imposible. Por lo mismo, es común encontrar documentos que contienen palabras mal escritas, propiciado por errores ortográficos, al momento de mecanografiar o a partir de

software de reconocimiento óptico, lo cual hace que sea imposible recuperar estos documentos a menos que se cometa el mismo error en la consulta.

El problema de corregir palabras mal escritas en textos es quizá la aplicación potencial más antigua para la búsqueda aproximada de patrones. La recuperación de información (RI) es una de las áreas que más demanda este tipo de búsqueda. La RI trata con encontrar información relevante en una colección de texto grande y la búsqueda de patrones es una de sus herramientas básicas.

En otras aplicaciones, como por ejemplo la biología computacional, el problema radica en que las secuencias de ADN y proteínas pueden verse como textos largos sobre un alfabeto en específico (por ejemplo, {A, C, G, T} en ADN). Estas secuencias representan el código genético de los seres vivos. Buscar secuencias específicas sobre estos textos es una operación fundamental para problemas tales como determinar características en las cadenas de ADN o determinar qué tan diferentes son dos secuencias genéticas.

Al principio, este problema se modeló para búsqueda exacta de patrones. Sin embargo, la búsqueda exacta fue poco útil para esta aplicación, puesto que los patrones rara vez aparecen exactamente en el texto. Las secuencias genéticas de dos miembros de la misma especie no son idénticas, sino muy similares. Algunas referencias para aplicaciones de búsqueda aproximada de patrones para biología computacional son [7], [8] y [9].

Otra motivación viene del área de procesamiento de señales. Uno de los principales problemas trata con el reconocimiento de voz, donde el problema general es determinar el mensaje textual que se está transmitiendo dada una señal de audio. El problema es complejo, dado que ciertas porciones de la señal puede comprimirse en el tiempo, otras partes pueden no pronunciarse, etc. Una coincidencia perfecta es

prácticamente imposible. Algunas referencias donde se aplica la búsqueda aproximada de patrones con procesamiento de señales son [10], [11] y [12].

En la actualidad sólo algunas herramientas tienen la capacidad de efectuar búsqueda aproximada y búsqueda de expresiones regulares con tiempos de búsqueda aceptables. Sin embargo, no existe hasta ahora, herramienta alguna capaz de efectuar estos dos tipos de búsqueda en forma directa en texto comprimido. Por lo que cuando una colección está comprimida, las herramientas actuales tienen que realizar un proceso de descompresión para después realizar la búsqueda.

1.3 Objetivo

El objetivo de esta tesis es analizar, proponer o mejorar técnicas de búsqueda aproximada de patrones simples y búsqueda de expresiones regulares que realicen una búsqueda directa en texto comprimido con LZW [13]. Además de construir una herramienta de búsqueda tipo *grep* basada en las mejores alternativas y que sería la primera herramienta capaz de realizar estos dos tipos de búsqueda sin tener que descomprimir el texto para después realizar la búsqueda.

1.4 Solución propuesta

La solución que se propone para resolver el problema de búsqueda aproximada consiste de tres etapas. La primera consiste en dividir el patrón en $k+1$ subpatrones con el objeto de asegurar que al menos uno de los subpatrones se puede encontrar en forma exacta. La segunda etapa consiste entonces en realizar una búsqueda exacta multipatrón con el conjunto de subpatrones resultantes utilizando el algoritmo de Boyer-Moore [14]. La tercera etapa parte del hecho de que se ha

encontrado un subpatrón. Por lo tanto es necesario hacer una verificación para comprobar si se ha encontrado una coincidencia del patrón completo permitiendo k diferencias, esto se realizará mediante dos autómatas, uno verificará hacia la izquierda y otro hacia la derecha del subpatrón encontrado, estos autómatas se simularán mediante la técnica de paralelismo de bits.

Para búsqueda de expresiones regulares la solución propuesta consiste en calcular el largo mínimo de una cadena que sea una coincidencia con la expresión regular. Después se obtendrán todos los prefijos de ese largo en el lenguaje generado por la expresión regular y se realizará una búsqueda multipatrón de los prefijos. Cada vez que se encuentra una coincidencia, se verifica si hay una coincidencia de la expresión regular por medio de un autómata simulado por paralelismo de bits.

1.5 Organización de la tesis

Este trabajo se divide en 5 capítulos. El capítulo 1 contiene la Introducción (este capítulo). El capítulo 2 incluye definiciones y conceptos básicos necesarios para la comprensión de los capítulos subsecuentes y describe las principales técnicas para búsqueda aproximada y búsqueda de expresiones regulares. El capítulo 3 describe los principales trabajos que se han desarrollado en el ambiente de búsqueda de patrones en texto comprimido. El capítulo 4 contiene los algoritmos propuestos para cada tipo de búsqueda, las simulaciones realizadas y resultados que se obtuvieron. En el capítulo 5 se exponen las conclusiones de este trabajo de tesis.

Capítulo 2

Conceptos Básicos

En este capítulo se presenta la notación que se utiliza en este trabajo así como un número de definiciones necesarias para comprender los capítulos siguientes. También se da una breve descripción del algoritmo de compresión LZW y se mencionan a detalle las técnicas más populares para búsqueda de patrones en texto sin comprimir.

2.1 Notación

La notación utilizada en esta tesis sigue la simbología que se encuentra generalmente en los trabajos relacionados con el fin de facilitar la compresión del trabajo. A continuación se presenta una lista de las variables utilizadas en los capítulos siguientes:

T: Cadena de caracteres sin comprimir (texto no comprimido).
Z: Cadena de caracteres comprimida (texto comprimido).
P: Patrón a buscar en el texto comprimido.
u: Longitud de la cadena de caracteres *T*.

n: Longitud de la cadena de caracteres *Z*.

m: Longitud del patrón *P*.

a, b, c: Caracteres de texto.

x, y, z: Subcadenas de texto.

k: Errores permitidos entre el patrón y una subcadena de *T*.

$\sum$: Alfabeto.

σ: Tamaño del alfabeto.

ER: Expresión regular.

r: Longitud de la expresión regular.

Para los algoritmos de paralelismo de bits que se mencionan en capítulos siguientes se utiliza la siguiente notación: 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 almacenan en una 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, $b_l b_{l-1} \ldots b_2 b_1 << 3 = b_{l-3} \ldots . b_2 b_1 000$.

2.2 Definiciones básicas

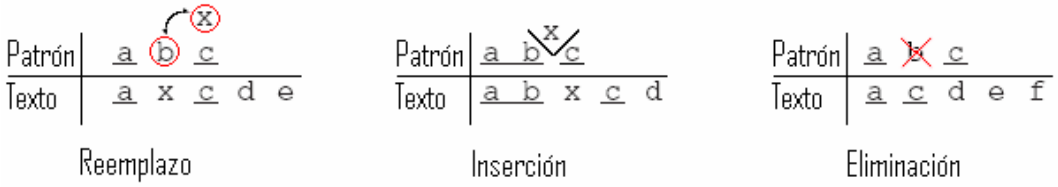
Definición 1. (*Alfabeto* $\sum$): Es un conjunto finito y no vacío de símbolos. Algunas veces se utilizará el término carácter en lugar de símbolo.

Definición 2. (*Patrón*): Dado un alfabeto $\sum$ de tamaño σ, un patrón es una secuencia finita de elementos de $\sum$. Existen diversos tipos de patrones, los más utilizados son:

- *Simples*. El patrón es una secuencia de caracteres que es una palabra en el texto, este es el patrón básico.
- *Prefijos*. Es una cadena que forma el inicio de una palabra en el texto.
- *Sufijos*. Es una cadena que forma la terminación de una palabra en el texto.
- *Subcadenas (factor)*: Es una cadena la cual aparece en una palabra del texto.
- *Rangos*: Un par de cadenas que coinciden con alguna palabra que se encuentra entre ellas en un orden lexicográfico.
- *Permitiendo errores*: Una palabra con un margen de error. La coincidencia del patrón con el texto puede tener un número de errores menor o igual a una distancia de edición especificada.
- *Expresiones regulares*: Es un patrón general construido por simples cadenas y los siguientes operadores: Unión (|), Concatenación (·) y Repetición (*).

Definición 3. (Coincidencia del *patrón*): Son los segmentos que satisfacen las especificaciones del patrón.

Definición 4. (*Distancia de edición*): Se define como el menor número de inserciones, eliminaciones, o reemplazos de caracteres necesarios para hacer iguales dos palabras.



Definición 5. (Lenguaje): Un lenguaje es un conjunto finito o infinito de cadenas. En particular, el lenguaje ∑* denota el conjunto de todas las cadenas sobre el alfabeto ∑. A la cadena de longitud cero se le llama cadena vacía, generalmente se denota como ε.

Definición 6. (E*xpresión regular*): Una expresión regular (*ER*) es una cadena sobre el conjunto de símbolos $\sum$ U {ε, |, · , *, (,) } y se define recursivamente como la cadena vacía ε, un carácter $\alpha \in \sum^*$; y (ER_1), $(ER_1 \cdot ER_2)$ $(ER_1 \mid ER_2)$, y (ER_1^*), donde ER_1 y ER_2 son expresiones regulares.

Definición 7. (Longitud de una *ER*): La longitud de una expresión regular es el número total de elementos de $\sum$ contenidos en la expresión regular, omitiendo los operadores (*, |, ·).

Definición 8. (*Autómata*): Es un grafo donde las flechas son etiquetadas con elementos de $\sum$ u {ε}. Sólo hay un estado inicial y cero o más estados finales. Un autómata reconoce una cadena $x \in \sum^*$ si hay una ruta del estado inicial a un estado final, tal que la concatenación de la etiquetas de las flechas recorridas sea *x*. El lenguaje reconocido por un autómata es el conjunto de cadenas reconocidas.

Definición 9. (*Estado activo*): Dada una cadena *x*, un estado *i* de un autómata está activo después de leer *x*, si *x* es la ruta del estado inicial al estado *i*.

Definición 10. (*Autómata Finito Determinístico*): Un autómata es un autómata finito determinístico (AFD) si sólo un estado puede estar activo, para una cadena dada *x*. Se representa como una quíntupla $(Q, \sum, \delta, q_0, F)$ donde:

Q: Conjunto finito de estados.

$\sum$: Alfabeto de entrada finito.

$q_0 \in Q$: Estado inicial.

$F \subseteq Q$: Conjunto de estados finales o de aceptación.

δ: $Q \times \sum \rightarrow Q$ Función de transición de un [estado, símbolo] a un único estado.

Definición 11. (*Autómata Finito No Determinístico*): Un autómata es finito no determinístico (AFN) si más de un estado puede estar activo, para una cadena dada *x*. Se representa como una quíntupla $(Q, \sum, \delta, q_0, F)$ donde:

Q: Conjunto finito de estados.

$\sum$: Alfabeto de entrada finito.

$q_0 \in Q$: Estado inicial.

$F \subseteq Q$: Conjunto de estados finales o de aceptación.

$\delta \subseteq Q \times \sum \times Q$: Función de un transición de un [estado, símbolo] a varios estados posibles.

2.3 Método de compresión LZW

En general la idea de los métodos de compresión Ziv-Lempel es reemplazar subcadenas en el texto por un apuntador a coincidencias previas de éstas. Entre las variantes más conocidas se encuentra a LZW, este método fue implementado por Terry Welch [13] y se ha convertido en uno de los más populares debido a su buena razón de compresión combinado con un tiempo eficiente de compresión y descompresión. El programa *Compress* de Unix utiliza este método. 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 '*a*' y se produce una falla en la búsqueda, es decir, la cadena *I* está en el diccionario pero no la cadena *I'a*', el codificador agrega a la cadena comprimida el apuntador de diccionario que apunta a la cadena *I*, almacena la cadena *I'a*' en la siguiente entrada del diccionario disponible, y por último inicializa la cadena *I* al símbolo '*a*'. En la Tabla 2-1 se muestra el diccionario generado y la codificación obtenida por LZW para el texto "*abracadabra_*".

0	NULL	99	c	262	da	**97**(a),
1	SOH		.	263	abr	**98**(b)
	.		.	264	ra_	**114**(r)
	.		.			**97**(a)
	.	255	255			**99**(c)
32	SP	256	ab			**97**(a)
	.	257	br			**100**(d)
	.	258	ra			**256**(ab)
	.	259	ac			**258**(ra)...
97	a	260	ca			
98	b	261	ad			

Tabla 2-1 Codificación del texto 'abracadabra' con el algoritmo de compresión LZW

2.4 Técnicas de búsqueda

Existen diversas técnicas para búsqueda en texto, sin embargo, sólo se describirán aquellas que han demostrado un buen desempeño en la práctica o que han sido valiosas teóricamente para el surgimiento de nuevas técnicas.

El algoritmo de búsqueda de patrones más simple es el algoritmo de fuerza bruta [15]. Este recorre el texto carácter por carácter y trata de encontrar una coincidencia con el patrón en todas las posiciones del texto. El problema con este algoritmo es la necesidad de retroceder. Por ejemplo, si el texto es '*abaababa*' y el patrón '*abab*', el algoritmo trata de encontrar una coincidencia a partir de la primera posición del texto. El algoritmo falla en la posición cuatro (letra '*a*') del texto y procede a encontrar una coincidencia a partir de la segunda posición del texto y así sucesivamente. En este caso el algoritmo toma cuatro comparaciones sólo para avanzar un carácter en el texto. La Figura 2.1 ilustra el ejemplo:

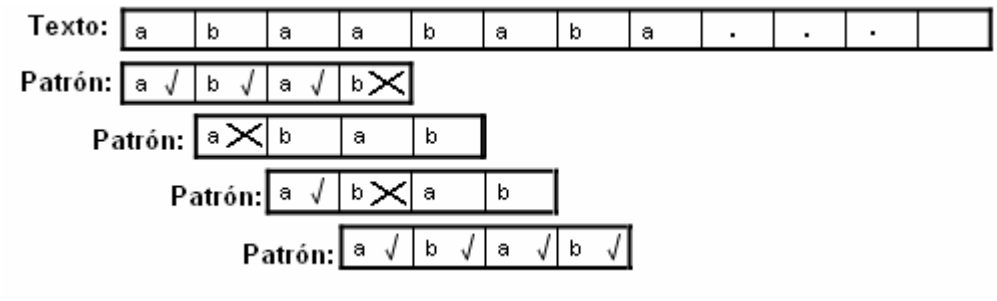


Figura 2.1 Algoritmo de fuerza bruta para búsqueda de patrones en texto

Si *m* es la longitud del patrón y *u* la longitud del texto, la complejidad en tiempo del algoritmo es de O(*mu*) en el peor caso. Otros algoritmos que siguen la misma idea son los de Knuth-Morris-Pratt [16] y Shift-Or [17], este último tiene la flexibilidad de poder extenderse para resolver búsqueda de patrones más complejos.

Otra técnica de búsqueda es por sufijo y consiste en buscar el patrón dentro de una ventana que se desplaza a lo largo del texto. La búsqueda se hace de derecha a izquierda a lo largo de la ventana de búsqueda, se lee el sufijo más largo de la ventana, que también es un sufijo del patrón. Esta técnica permite evitar leer algunos

caracteres del texto. El algoritmo más famoso que utiliza esta técnica es el de Boyer-Moore[14]. La Figura 2.2 ilustra el proceso de búsqueda de patrones por sufijo.

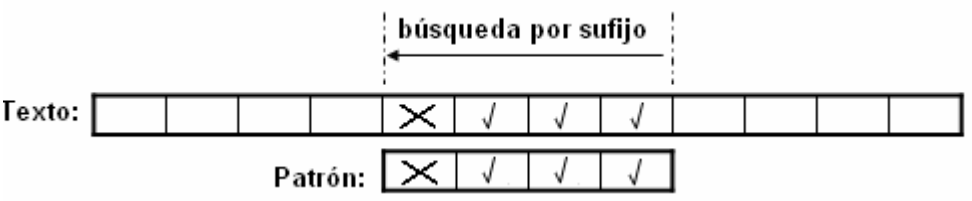


Figura 2.2 Búsqueda de patrones por sufijo

2.4.1 Búsqueda aproximada

El problema de búsqueda de patrones se vuelve más complejo cuando se requiere buscar patrones en donde ocurren alteraciones con cierta frecuencia, es decir, la búsqueda no es exacta. A este problema se le conoce como búsqueda aproximada o búsqueda permitiendo errores y su objetivo es encontrar las posiciones del texto donde el patrón ocurre con cierto número de errores (ver definición 4). A continuación se describen las principales técnicas para búsqueda aproximada.

Programación Dinámica.

La solución clásica para búsqueda aproximada de patrones se basa en la programación dinámica [18]. Esta técnica consiste en calcular una matriz $E[i,j]$ que representa el número mínimo de errores permitidos k para hacer coincidir $p_{1\ldots i}$ ($i \leq m$) con $t_{j'\ldots j}$ ($j' \leq j \leq u$). El algoritmo trabaja de la siguiente forma: las entradas de la primera fila $E[0,j]$ ($0 \leq j \leq u$) se inicializan a 0, y las entradas de la primera columna $E[i,0]$ ($0 \leq i \leq m$) se inicializan a i. Las demás entradas $E[i, j]$ ($1 \leq i \leq m$, $1 \leq j \leq u$) se calculan dinámicamente columna por columna como muestra el siguiente seudocódigo:

```
1 If p_i = t_j Then
2    E[i,j] ← E[i-1,j-1]
3 Else
4    E[i,j] ← 1 + min(E[i-1,j-1],E[i-1,j],E[i,j-1])
5 EndIf
```

Figura 2.3 Seudocódigo del algoritmo de programación dinámica para búsqueda de patrones en texto

Las posiciones de texto donde $E[m, j] \leq k$ se reportan como coincidencias del patrón. La Tabla 2-2 muestra los valores que genera el algoritmo al buscar el patrón '*incidir*' en el texto '*coincidencia*' con máximo 2 errores. Los números en color indican las posiciones finales de las coincidencias en el texto.

		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

Tabla 2-2 Ejemplo del algoritmo de programación dinámica para buscar 'incidir' en el texto 'coincidencia' permitiendo 2 errores

Basados en autómatas.

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 2.4, 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.

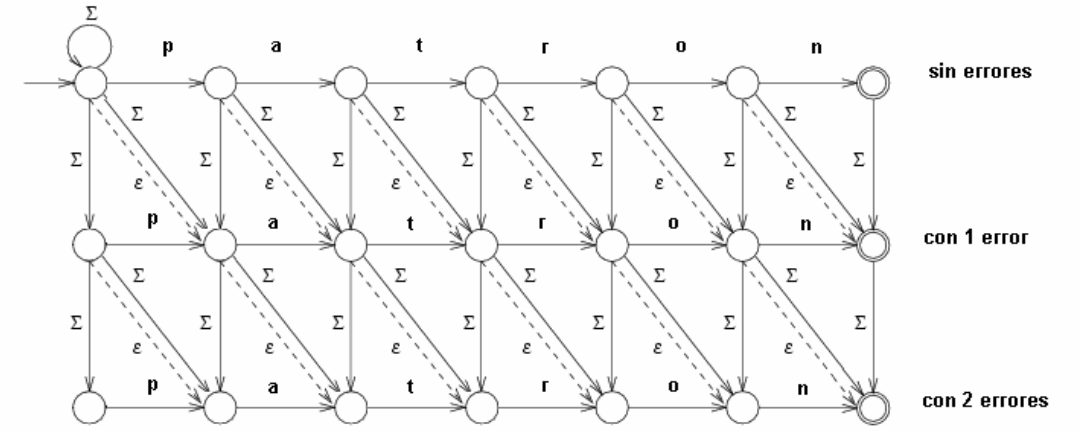


Figura 2.4 Autómata finito no determinístico para búsqueda aproximada de la cadena 'patrón' permitiendo 2 errores

Cada vez 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 final está activo, la fila en la que se encuentre ese estado indicará el número de errores encontrados.

Algoritmos de paralelismo de bits.

Otra técnica que se utiliza para búsqueda aproximada es la de paralelismo de bits, teniendo sus mejores resultados para patrones cortos, que en muchos casos son los patrones de interés. 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 [19] 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 [4]:

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

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 2.4 respectivamente.

Filtrado.

Finalmente se tienen a los algoritmos de filtrado, los cuales se basan en el hecho de que algunas porciones del patrón aparecen sin error aún en una coincidencia aproximada. La técnica de filtrado no es efectiva para niveles de error altos debido a la cantidad de verificaciones que es necesario realizar.

2.4.2 Búsqueda de Expresiones Regulares

Una las características más importantes de los sistemas de búsqueda de patrones se refiere a la flexibilidad que debe ofrecer al momento de especificar el

patrón a buscar. Las expresiones regulares (*ER*) son una forma extremadamente poderosa para especificar un conjunto de patrones de búsqueda.

El problema de búsqueda de una expresión regular en un texto *T* es encontrar todos los factores de *T* que coincidan con las cadenas que pertenecen al lenguaje generado por la *ER*.

Existen diferentes esquemas para realizar la búsqueda de expresiones regulares. El esquema clásico analiza la expresión regular por medio de un árbol de análisis al que posteriormente transforma en un autómata finito no determinístico. Existen diversas maneras de realizar este último paso, sin embargo, hay dos que sobresalen entre las demás. La primera es la construcción de Thompson [20], y la segunda es la construcción de Glushkov [21]. Estas dos construcciones se explican a detalle más adelante.

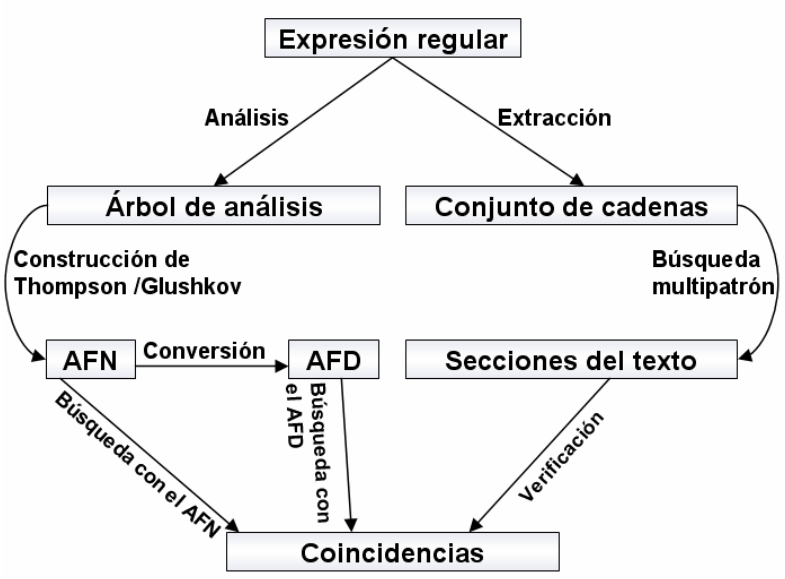


Figura 2.5 Esquema clásico para búsqueda de expresiones regulares en texto

Aunque se puede hacer la búsqueda directamente con el AFN de diferentes formas, el proceso es bastante lento. El algoritmo consiste de mantener una lista de estados activos y actualizar la lista cada vez que se lee un nuevo carácter del texto. La búsqueda toma un tiempo en el peor de los casos de $O(mu)$.

Otro esquema es convertir el AFN en un Autómata Finito Determinístico (AFD), el cual permite efectuar la búsqueda en un tiempo de $O(u)$ desempeñando una transición directa por carácter de texto. Sin embargo, la construcción de tal autómata toma un tiempo y espacio de $O(2^m)$ en el peor de los casos.

Un tercer esquema es filtrar el texto utilizando una búsqueda multipatrón y encontrar vecindades donde pueda haber una coincidencia, y verificar localmente utilizando una de las estrategias previas. Estas estrategias se pueden combinar, sin embargo, el uso de paralelismo de bits puede acelerar algunas partes del proceso de búsqueda. Recientemente el uso de paralelismo de bits ha sido propuesto para simular el autómata finito no determinístico y evitar así la construcción del autómata finito determinístico. Esta técnica consiste en representar por medio de un bit si un estado del autómata está activo o no.

Un aspecto importante es que la mayoría de las construcciones de autómatas utilizan una representación de árbol de la *ER* para realizar los cálculos. Las hojas del árbol se etiquetan con los caracteres que pertenecen al alfabeto de la *ER* y también con el símbolo ε en caso de que haya. Los nodos internos se etiquetan con los operadores. Los nodos que se etiquetan con "|" o "·" tienen dos subexpresiones hijas ER_1 y ER_2. Los nodos etiquetados con "*", "?", o "+" tienen sólo una subexpresión hija ER_1. La representación generalmente no es única, puesto que algunos operadores son conmutativos y/o asociativos. La Figura 2.6 muestra el árbol generado para la expresión regular "(aba|ba)a | a((ba)*)ab" .

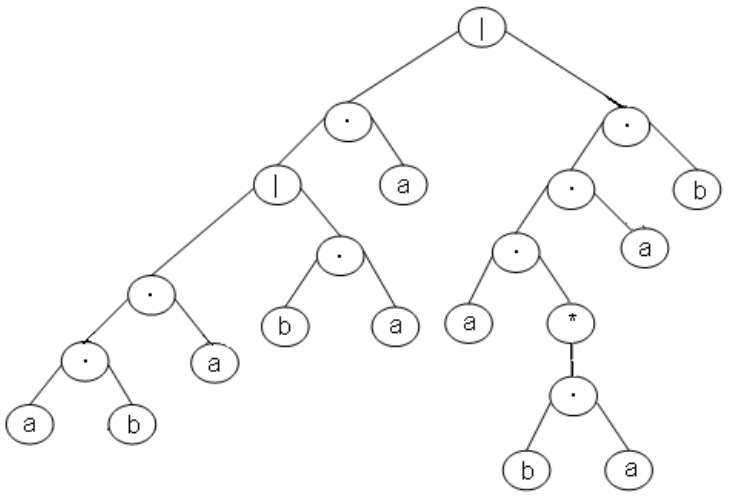


Figura 2.6 Árbol de análisis para la expresión regular (aba|ba)a|a((ba)*)ab

En el árbol, el símbolo "·" (v_l, v_r) significa la concatenación de v_l con v_r. De forma similar el símbolo "|" (v_l, v_r) es la unión de v_l con v_r. El símbolo * simboliza un nodo de un solo hijo.

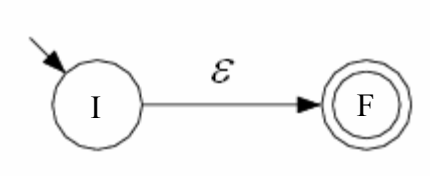
Construcción del AFN

Como se mencionó anteriormente, los esquemas más utilizados y prácticos para construir un AFN son la construcción de Thompson y la construcción de Glushkov. La construcción de Thompson es simple y genera un AFN que es lineal en el número de estados (máximo $2m$) y de transiciones (máximo $4m$). Sin embargo el autómata tiene transiciones nulas (transiciones-ε), es decir, transiciones que pueden pasar de un estado a otro sin leer ningún carácter de texto o al leer la cadena vacía ε. Por otro lado, la construcción de Glushkov genera un AFN que tiene exactamente m+1 estados pero un número de transiciones que es $O(m^2)$ en el peor de los casos. Sin embargo, esta construcción no produce transiciones-ε.

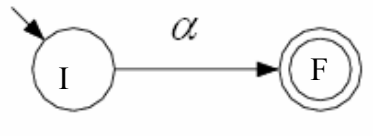
Autómata de Thompson

La construcción de Thompson [20] es una trascripción directa de la representación de árbol de la expresión regular, utilizando transiciones-ε para simplificar su trascripción. Se asocia la construcción de un autómata específico *Th*(*v*) para cada tipo de nodo y hoja del árbol. Estas son:

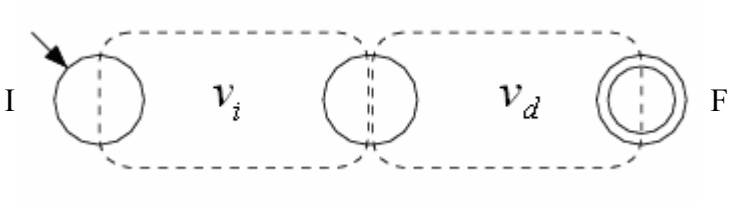
i) La construcción para la cadena vacía consiste de dos nodos unidos por una transición-ε.



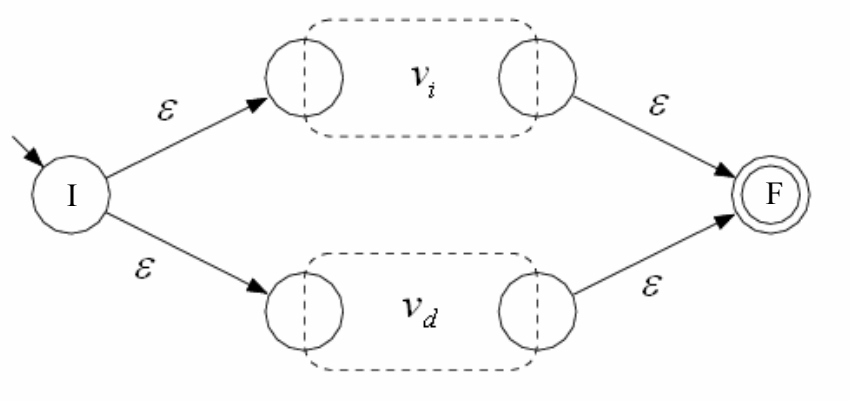
ii) Para un carácter α la construcción es similar, excepto que la transición se etiqueta con el carácter en vez de etiquetarse con el carácter nulo.



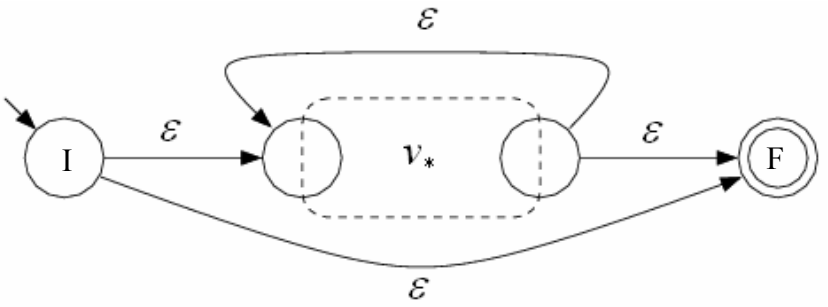
iii) En la construcción para la concatenación de dos subexpresiones regulares v_i y v_d el estado de inicio de v_i se convierte en el estado de inicio del AFN compuesto, y el estado de aceptación de v_d se convierte en el estado de aceptación del AFN compuesto. El estado de aceptación de v_i se fusiona con el estado de inicio de v_d; es decir todas las transiciones desde el estado de inicio de v_d se convierten en transiciones desde el estado de aceptación de v_i.



iv) La construcción para la unión de dos subexpresiones regulares v_i y v_d se realiza colocando transiciones etiquetadas con caracteres ε que conducen al inicio de cada subexpresión a partir de un nuevo estado inicial y que relaciona el final de cada subexpresión a un nuevo nodo final por medio de transiciones etiquetadas con caracteres ε.



v) La construcción para un nodo '*' usa la misma idea que la construcción anterior. Dado que la subexpresión v_* puede aparecer cero o más veces, se crea una transición-ε de retroceso del estado final de v_* al estado inicial de v_*. Por otro lado, la estrella también significa que v_* puede ser ignorada, por lo que se crean dos nuevos nodos, uno será el estado inicial *I* y el otro estado final *F*, estos dos se unen por medio de una transición-ε. Otras dos transiciones-ε unen el estado *I* con el estado inicial de v_* y el estado final de v_* con el estado *F*.



El algoritmo completo de Thompson consiste en desempeñar un recorrido de abajo hacia arriba en el árbol e ir manteniendo la construcción del autómata de la raíz como el autómata de Thompson de la expresión completa. El seudocódigo del algoritmo para construir el autómata de Thompson se muestra en la Figura 2.7.

```
alg_Thompson(v)
  If v = [|](v_i,v_d) OR v = [·](v_i,v_d) Then
      Th(v_i) ← alg_Thompson(v_i)
      Th(v_d) ← alg_Thompson(v_d)
  EndIf
  Else
      If v = [*](v_*) Then
      Th(v_*)← alg_Thompson (v_*)
  EndElse
  /*  Al finalizar la parte recursiva, se construye el
      autómata para el nodo actual */
 If v =(ε) Then Return construcción (i)
 If v =(α) Then Return construcción (ii)
 If v =[·](v_i,v_d) Then
      Return construcción (iii) de Th(v_l) y Th(v_r)
 If v = [|](v_i,v_d) Then
      Return construcción (iv) de Th(v_l) y Th(v_r)
 If v = [*](v_*) Then
      Return construcción (v) de Th(v_*)


Thompson (ER)
      v_ER ← construccion_arbol (ER)
      Th(v_ER) ← alg_Thompson (v_ER)
```

Figura 2.7 Seudocódigo del algoritmo de Thompson. El autómata se construye recursivamente utilizando el árbol de análisis de la expresión regular

Propiedades del autómata de Thompson

La construcción de cada uno de los nodos del árbol agrega máximo dos estados y cuatro transiciones al autómata actual. Por lo tanto, al final de la construcción, el total de número de estados y transiciones se limita a 2*m* y 4*m*, respectivamente. Aunque se pueden calcular límites más ajustados, lo importante es notar que el número de estados y transiciones es lineal en *m*. Además, cada uno de los nodos del AFN tiene máximo dos arcos de entrada y dos arcos de salida, y el AFN completo tiene sólo un estado inicial y un estado final.

Otra propiedad interesante es que todas las flechas que no están etiquetadas por ε van de los estados numerados por *i* a estados numerados por *i*+1 siempre que se procesen los caracteres de la expresión regular de izquierda a derecha. La construcción de Thompson para la expresión regular "(aba|ba)a | a((ba)*)ab" se muestra en la Figura 2.8.

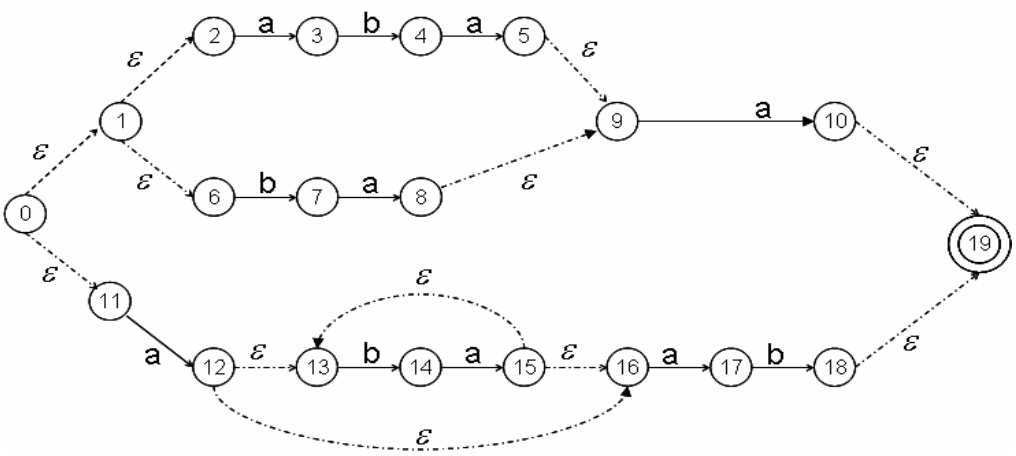


Figura 2.8 Autómata de Thompson para la expresión regular (aba|ba)a | a((ba)*)ab

Autómata de Glushkov

La construcción de Glushkov [21] ha sido muy popularizada por Berry y Sethi en [22]. Para clarificar la construcción del AFN de Glushkov se ejemplificará el proceso. Se inicia marcando las posiciones de los caracteres del ∑ en la *ER*, contando sólo los caracteres. Por ejemplo, la expresión regular (aba|ba)a | a((ba)*) ab se marca como $(a_1b_2a_3 \mid b_4a_5)a_6 \mid a_7((b_8a_9)^*)\ a_{10}b_{11}$. La expresión regular marcada se denota como *ER'* y su lenguaje donde cada uno de los caracteres incluye su índice, se denota como *L(ER')*. En el ejemplo, $L((a_1b_2a_3 \mid b_4a_5)a_6 \mid a_7((b_8a_9)^*)\ a_{10}b_{11}) = \{a_1b_2a_3a_6, b_4a_5a_6, a_7a_{10}b_{11}, a_7b_8a_9a_{10}b_{11}, a_7b_8a_9b_8a_9a_{10}b_{11}\ldots\}$. *Pos*(*ER'*) = {1…*m*} es el conjunto de posiciones de *ER'* y ∑' el alfabeto de caracteres marcados. El autómata de Glushkov se construye primero sobre la expresión marcada *RE'* y este autómata reconoce *L*(*ER'*). A partir de esto se deduce el autómata de Glushkov que reconoce *L*(*ER*) eliminando los índices de todos los caracteres.

El autómata resultante es el conjunto de posiciones (estados) mas un estado inicial 0. Así se generan *m*+1 estados etiquetados a partir de 0 a *m*. Cuando se lee un nuevo carácter α, se necesita conocer cuales son las posiciones que se puede alcanzar a partir del estado *j* con el carácter α. El algoritmo de Glushkov calcula todas las posiciones (estados) accesibles a partir de una posición (estado) *j*.

A continuación se presentan cuatro nuevas definiciones para explicar en profundidad el algoritmo. Se denota por α_y el carácter indexado de *ER*' que está en la posición *y*.

Definición 12: *PrimeraPos (ER')* = $\{ x \in Pos\ (ER'),\ \exists a \in \Sigma'^{*}, \alpha_x a \in L(ER')\}$

El conjunto *PrimeraPos*(*ER'*) representa el conjunto de posiciones iniciales del *L*(*RE'*), es decir, donde puede iniciar la *ER*. En el ejemplo *PrimeraPos* $((a_1b_2a_3 \mid b_4a_5)a_6 \mid a_7((b_8a_9)^*)\ a_{10}b_{11}) = \{1,4,7\}$

Definición 13: *UltimaPos* (*ER'*) = { $x \in Pos\ (ER'),\ \exists a \in \Sigma'^{*},\ a\alpha_x \in L(ER')$}

El conjunto *UltimaPos*(*ER'*) representa el conjunto de posiciones finales de *L*(ER'), es decir, las posiciones donde una cadena leída puede ser reconocida. En el ejemplo *UltimaPos* $((a_1b_2a_3 \mid b_4a_5)a_6 \mid a_7((b_8a_9)^*)\ a_{10}b_{11}) = \{6,11\}$

Definición 14: *SiguientePos* (*ER'*, *x*) = { $y \in Pos\ (ER'), \exists a,b \in \Sigma'^{*}, a\alpha_x\alpha_y b \in L(ER')$}

El conjunto *SiguientePos* (*ER'*, *x*) representa todas las posiciones en *Pos*(*ER'*) que son accesibles a partir de *x*. Si se considera la posición 9 en el ejemplo, el conjunto de posiciones accesibles es: *SiguientePos* $((a_1b_2a_3 \mid b_4a_5)a_6 \mid a_7((b_8a_9)^*)\ a_{10}b_{11}, 9) = \{ 10,8\}$

También se necesita una función extra $Anulable_{ER}$ que indique si la cadena vacía ε está en *L*(*ER*).

Definición 15: La función $Anulable_{ER}$ se define recursivamente. Su valor es {ε} si ε pertenece al *L*(*ER*) y 0 en otro caso.

$$
\begin{aligned}
Anulable_{ER} &= \{\varepsilon\} \\
Anulable_{a\in\Sigma} &= \varnothing \\
Anulable_{ER1 \mid ER2} &= Anulable_{ER1} \cup Anulable_{ER2} \\
Anulable_{ER1 \cdot ER2} &= Anulable_{ER1} \cap Anulable_{ER2} \\
Anulable_{ER^*} &= \{\varepsilon\}
\end{aligned}
$$

El autómata determinístico de Glushkov *GL'* que reconoce el lenguaje *L*(*RE'*) se construye de la siguiente manera:

$$GL' = (S, \sum, I, F, \delta')$$

Donde:

i) S es el conjunto de estados, $S = \{ 0, 1, \ldots , m\}$, es decir, el conjunto de posiciones de *Pos* (*ER'*) y el estado inicial es $I = 0$.

ii) F es el conjunto de estados finales, F = *UltimaPos* (*ER'*) U ($Anulable_{ER} \cdot \{0\}$). Informalmente, un estado (posición) i es final si está en *UltimaPos* (*ER'*). El estado inicial 0 también es final si la cadena vacía ε pertenece a $L(ER')$, en tal caso $Anulable_{ER} = \{\varepsilon\}$ y por lo tanto $Anulable_{ER} \cdot \{0\} = \{0\}$. Sino, $Anulable_{ER} \cdot \{0\} = \varnothing$

iii) δ' es la función de transición del autómata, y se define por

$$Vx \in Pos\ (ER')\ \forall y \in SiguientePos\ (ER', x),\quad \delta'(x, \alpha_y) = y$$

Las transiciones del estado inicial se definen por:

$$Vy \in PrimeraPos(ER'),\ \delta'(0, \alpha_y) = y$$

El autómata de Glushkov para la expresión marcada $(a_1b_2a_3 \mid b_4a_5)a_6 \mid a_7((b_8a_9)^*)\ a_{10}b_{11}$ se muestra en la Figura 2.9

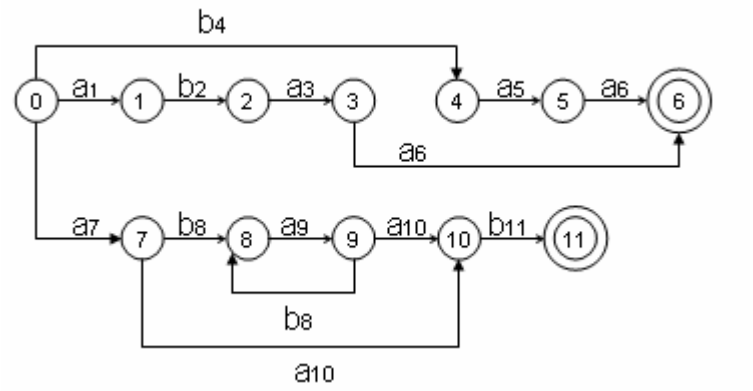


Figura 2.9 Autómata de Glushkov para la expresión marcada $(a_1b_2a_3 \mid b_4a_5)a_6 \mid a_7((b_8a_9)^*)\ a_{10}b_{11}$

Para obtener el autómata de Glushkov de la *ER* original, simplemente se eliminan los índices del autómata marcado. Con este paso, el autómata generalmente se vuelve no determinístico. El nuevo autómata reconoce el lenguaje *L*(*ER*). El autómata de Glushkov para el ejemplo (aba | ba)a | a((ba)*)ab se muestra en la Figura 2.10

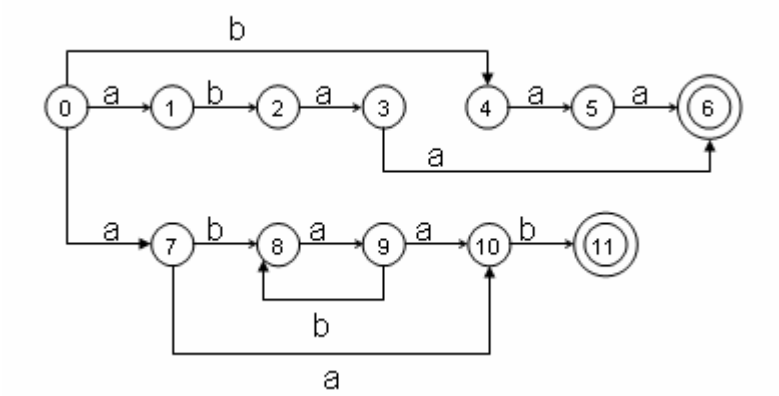


Figura 2.10 Autómata de Glushkov construido para la expresión regular (aba | ba)a | a((ba)*)ab. El autómata se deriva del autómata marcado eliminando únicamente los índices

El algoritmo de Glushkov se basa en la representación de árbol T_{ER} de la *ER*. Cada uno de los nodos *v* de este árbol representa una subexpresion ER_v de *ER*. Las siguientes variables se asocian a *v*:

PrimeraPos(*v*): Lista de las posiciones que representan el conjunto *PrimeraPos* (*ER'*)
UltimaPos(*v*): Lista de las posiciones que representan el conjunto de *UltimaPos* (ER')
$Anulable_v$: Se establece a {ε} si $L(ER_v)$ contiene la cadena vacía ε, y a 0 en otro caso.

Estas variables se calculan para cada nodo en orden postfijo, es decir, se calculan para cada uno de los hijos de *v* y sólo después para *v*. Si *v* representa "|" o "·" los hijos de *v* se denotan como v_l y v_r, y si v representa "*" su único hijo se denota como *v**.

El conjunto *SiguientePos*(*x*) es una variable Global. Para cada nodo *v* *SiguientePos*(*x*) se actualiza acorde a las posiciones en la subexpresion ER'_v. El algoritmo para calcular las variables se muestra en la Figura 2.11. Para cada nodo del árbol *v* se calculan los valores de *PrimeraPos*(*v*), *UltimaPos*(*v*), *SiguientePos*(*v*), y $Anulable_v$. Los nodos se visitan en orden posfijo. Los valores de el nodo v_{ER} se calculan a partir de los valores obtenidos de sus hijos. La posición de cada carácter se calcula conforme se recorre el árbol.

```
variables_Glushkov (v_ER , Ipos)
      /* se calcula recursivamente el primer hijo en orden posfijo */
 If v = [|]  (v_i , v_d) OR v =  [·]  (v_i , v_d) Then
       Ipos← variables_Glushkov (v_i, lpos)
       Ipos ← variables_Glushkov (v_d, lpos)
 EndIf
 Else
     If v = [*] (v*) Then Ipos ← variables_Glushkov (v*,lpos)
 EndElse


      /* Al finalizar la parte recursiva se calculan los valores para
el nodo actual */
If v = (ε) Then
       PrimeraPos(v) ← ø, UltimaPos(v) ← ø, Anulable_v ←{ε}

If v = (α) ,  α ∈ Σ Then
        lpos ← Ipos + 1
        PrimeraPos(v) ← {lpos}, UltimaPos(v) ← {lpos}, Anulable_v ← ø,
       SiguientePos (lpos) ← 0
EndIf
If v = [|] (v_i , v_d) Then
       PrimeraPos(v) ← PrimeraPos(v_i) U PrimeraPos(v_d)
       UltimaPos(v)  ← UltimaPos(v_i) U UltimaPos(v_d)
       Anulable_v ← Anulablev_i  U  Anulablev_d
EndIf
If   v = [·] (v_i , v_d )Then
      PrimeraPos(v) ← PrimeraPos(v_i) U (Anulablev_i  · Anulablev_d)),
      UltimaPos(v)  ← (Anulablev_d · UltimaPos(v_i)) U UltimaPos(v_d),
      Anulable_v ← Anulablev_i  ∩ Anulablev_d
      For x ∈ UltimaPos(v_i) Do SiguientePos(x) ← SiguientePos(x) U
      PrimeraPos(v_d)
Endif
If v = [*] (v*) Then
```

```
      PrimeraPos(v) ← PrimeraPos(v*), PrimeraPos(v) ← PrimeraPos(v*),
      Anulable_v ← {ε}
      For x ∈ UltimaPos(v*) Do SiguientePos(x) ← SiguientePos(x) U
      PrimeraPos(v*)
EndIf

Return lpos
```

Figura 2.11 Algoritmo para calcular las variables de Glushkov

El algoritmo completo de Glushkov consiste en transformar la *ER* en un árbol v_{RE}, calcular las variables de éste con el algoritmo de la Figura 2.11 y luego construir el autómata de Glushkov a partir del nodo raíz v_{RE} del árbol. El seudocódigo del algoritmo completo se muestra en la Figura 2.12.

```
Gluhskov(ER)
      v_ER ← construccion_arbol(ER)
      m ← variables_Glushkov(v_ER )
      δ = ∅;
      For i ∈ 0…m Do crear el estado i
      For x ∈ First(v_ER) Do δ ← δ U {(0, α_x, x)}
      For i ∈ 0…m Do
            For x ∈ SiguientePos(i) Do
                  δ ← δ U {(i, α_x, x)}
      EndFor
      For x ∈ UltimaPos(v_ER) U (Anulablev_ER · {0}) Do
            marcar x como terminal
```

Figura 2.12 Algoritmo de Glushkov. En general el autómata es no determinístico y sus funciones de transición se denotan como δ

Propiedades del autómata de Glushkov.

El autómata de Glushkov no contiene transiciones-ε. Además todas las flechas que conducen a un estado dado y se etiquetan por el mismo carácter.

Esquemas clásicos para la búsqueda de expresiones regulares.

Se consideran los dos extremos, es decir, la simulación de un AFN puro y un AFD puro. Thompson propuso en [20] un algoritmo de búsqueda basado en la simulación directa de su AFN. Este algoritmo no es competitivo hoy en día, pero es la base para algoritmos más competitivos. El algoritmo almacena el conjunto de estados activos y para cada nuevo carácter de texto leído, revisa que nuevos estados se pueden agregar al conjunto de estados activos. A partir de los nuevos estados activos el algoritmo sigue todas las transiciones-ε hasta que se obtienen todos los estados alcanzables.

Así, bajo la construcción de Thompson cada estado tiene O(1) transiciones de salida y puede haber O(m) estados activos. Producir el nuevo conjunto de estados activos toma un tiempo de O(m) bajo una representación adecuada del conjunto de estados, por ejemplo un vector de bits. La propagación por transiciones-ε también toma un tiempo de O(m).

Como se mencionó anteriormente, otra técnica de búsqueda consiste en convertir la expresión regular en un AFD y luego realizar la búsqueda en el texto utilizando el AFD. Empleando esta técnica se han implementado algoritmos que ejecutan la búsqueda con un tiempo de O(u). La solución más simple es construir primero un AFN con alguna de las técnicas explicadas en las secciones previas (Thompson o Glushkov) y luego convertir el AFN en un AFD. Con el AFN un conjunto de estados pueden estar activos después de leer algún carácter de texto. Sin embargo, un AFD tiene exactamente un estado activo a la vez. Por lo tanto, el AFD se construye sobre el conjunto de estados del AFN, el número de estados generados en el peor de los casos es de O(2^m) lo cual es exponencial. Por lo que esta técnica sólo es conveniente para expresiones regulares pequeñas.

Capítulo 3

Trabajo Relacionado

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

3.1 Búsqueda en texto comprimido

El problema de búsqueda de patrones en texto comprimido fue planteado por primera vez por Amir y Benson [23], en un trabajo en el cual presentan un método de búsqueda en archivos comprimidos con LZ77. Desde entonces se ha desarrollado muchos trabajos sobre búsqueda de patrones en textos comprimidos. Sin embargo, la mayoría de éstos son estudios teóricos, sin llevar a cabo la realización de la implementación de los algoritmos para verificar su eficiencia práctica.

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 [24]. Mediante este esquema se han realizado

trabajos que ofrecen una buena solución al problema, pero en general la razón de compresión no es buena o su funcionalidad está limitada para archivos que contienen lenguaje natural y que sean de un tamaño superior a los 10 MB.

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. A continuación se mencionan los trabajos más importantes que se han desarrollado mediante estos dos esquemas.

Bajo el primer esquema de búsqueda, el primer algoritmo de compresión que permite búsqueda directa en texto comprimido fue el propuesto por Manber en [25]. El principal propósito del algoritmo era acelerar el proceso de búsqueda de patrones directamente en texto comprimido, dado que la compresión que se logra con este algoritmo de compresión no es buena. El algoritmo de compresión consiste en representar pares de caracteres que aparecen con gran frecuencia en el texto por medio de un único byte. La búsqueda se realiza mediante cualquier algoritmo de búsqueda de patrones, sólo se requiere generar pares de caracteres del patrón y buscarlos en el texto comprimido para encontrar las coincidencias del patrón.

Sin embargo, para el primer esquema de búsqueda, la principal tendencia es utilizar el método de compresión de Huffman basado en palabras, es decir, las palabras del texto se consideran como los símbolos que componen el texto. La compresión del archivo de texto se hace utilizando la versión semi-estática del método de compresión, es decir, se requiere dos pasos para lograr la compresión. El primer paso consiste en obtener las frecuencias de cada palabra en el texto y el segundo paso consiste en codificarlas, la idea es asignar códigos pequeños a palabras frecuentes.

En Turpin y Moffat [26] se presentó un método de búsqueda que trabaja bajo esta idea. Este método realiza la búsqueda directa en texto comprimido solamente si el patrón está compuesto por una sola palabra y se requiere una búsqueda exacta. En otro caso, Turpin y Moffat proponen que el texto se descomprima durante la búsqueda.

En [27] se presentó una variante para el método de compresión de Huffman basado en palabras, la cual consiste en codificar el alfabeto utilizando bytes en vez de bits. Aunque al utilizar bytes se presenta una pequeña pérdida en la razón de compresión, la ventaja que se adquiere es la capacidad de poder realizar una gran gama de búsquedas como de frases, expresiones regulares, búsqueda exacta y aproximada. Para generar los códigos se construye un árbol donde cada hoja representa una palabra y la ruta del nodo raíz a la hoja representa su código. Para llevar a cabo la búsqueda en el texto comprimido se busca el código del patrón en el árbol y después se ejecuta cualquier algoritmo de búsqueda de patrones para encontrar una coincidencia del patrón. El problema de esta representación es que para cada archivo comprimido se tiene que almacenar el árbol que contiene los códigos asignados a cada palabra, lo que requiere gran cantidad de espacio de almacenamiento. Además sólo es práctico para textos que contienen lenguaje natural.

Para el esquema que maneja archivos comprimidos con Ziv-Lempel, el primer algoritmo para búsqueda exacta aparece en [28], el cual presenta un algoritmo de búsqueda de patrones para texto comprimido con LZ78 que simula un autómata propuesto por Knuth, Morris y Pratt (KMP) en [16]. Este algoritmo sólo tiene la capacidad de determinar si el patrón aparece o no en el texto en un tiempo y espacio de $O(m^2 + n)$. Un algoritmo que resuelve el mismo problema pero que trabaja con archivos comprimidos con LZ77 se presentó en [29], y resuelve problema en un tiempo de $O(m + n\log^2(u/n))$.

Por otra parte, en [30] se presentó una extensión de [28] que resuelve el problema de búsqueda multipatrón sobre texto comprimido con LZ78/LZW. Este algoritmo se basa en el algoritmo Aho-Corasick [31], y encuentra todas las coincidencias del patrón en un tiempo y espacio de $O(M^2 + n)$, donde M es la suma de la longitud de cada uno de los patrones.

Un esquema general para búsqueda de patrones (simples y extendidos) directamente en texto comprimido se presentó en [32], especializándose para algunos formatos en particular (LZ77, LZ78). 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 [33] 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 [34], el cual actualmente es el más rápido en la práctica cuando la longitud del patrón es menor a 30 caracteres.

3.2 Búsqueda aproximada en texto comprimido

El problema de búsqueda aproximada sobre texto comprimido fue tratado por primera vez en [23]. 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 [35] y técnicas de paralelismo de bits [36].

Sin embargo, ambas soluciones son muy lentas. La primera solución práctica a este problema se presenta en [37]. 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 directamente en las áreas de texto candidatas.

En este trabajo se presenta una mejora al trabajo realizado en [37], 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.

3.3 Búsqueda de expresiones regulares en texto comprimido

Para búsqueda de expresiones regulares en texto comprimido hay pocos trabajos que mencionar. El trabajo más importante hasta ahora para búsqueda de expresiones regulares en texto comprimido es el que se presentó en [38], el cual trabaja con archivos comprimidos con métodos de la familia Ziv-Lempel, específicamente las variantes LZ78 y LZW. El algoritmo toma un tiempo en el peor de los casos de $O(2^m+mn+Rm\log m)$ donde R es el número de coincidencias encontradas. El algoritmo de búsqueda consiste en adaptar el esquema de búsqueda que trabaja con un AFD simulado por medio de paralelismo de bits, para que trabaje con secuencias de bloques, en vez de secuencias de caracteres.

El algoritmo de búsqueda de expresiones regulares en texto comprimido que se presenta en este trabajo, consiste en una idea similar al presentado en [38]. Sin embargo, en este trabajo también se combina algoritmos multipatrón que permitan evitar leer caracteres durante el proceso de búsqueda con el objetivo de acelerar el mismo.

3.4 Herramientas de búsqueda en texto sin comprimir

En esta sección se describen brevemente dos herramientas de búsqueda que trabajan sobre texto sin comprimir. Estas herramientas son las más rápidas que existen hoy en día. La razón por las que se mencionan es por que posteriormente se utilizan para comparar el desempeño de éstas con los algoritmos de búsqueda que se proponen en esta tesis.

La primera herramienta se denomina *agrep* [4], la cual fue desarrollada en 1992 por Sun Wu y Udi Manber en la Universidad de Arizona. *Agrep* es un software de búsqueda capaz de realizar búsqueda exacta y aproximada de cadenas simples, extendidas y expresiones regulares, también búsqueda exacta de múltiples cadenas.

Agrep no utiliza un algoritmo común para llevar a cabo el proceso de búsqueda, selecciona el mejor algoritmo dependiendo del tipo de búsqueda que se efectúe. Para búsqueda de cadenas simples donde su longitud no exceda de 400 caracteres utiliza al algoritmo Horspool [39], el cual es una variante del algoritmo Boyer-Moore. Para cadenas más largas utiliza una técnica similar, pero utilizando pares de caracteres en vez de caracteres sencillos.

Para búsqueda de expresiones regulares utiliza una extensión del algoritmo Shift-And [19], simulando por medio de paralelismo de bits el autómata de Thompson. La búsqueda aproximada la maneja de dos formas. Para búsqueda de cadenas simples con niveles de error bajos, *agrep* utiliza un algoritmo de filtrado, el cual divide el patrón en varias piezas y realiza una búsqueda multipatrón de ellas realizando una verificación alrededor de cada pieza encontrada. En otro caso la búsqueda la maneja por medio de paralelismo de bits simulando un AFN.

Otra herramienta muy popular es *nrgrep*. Esta herramienta fue desarrollada en el 2000 por Gonzalo Navarro en la Universidad de Chile. Funcionalmente es similar a *agrep*, sin embargo, no tiene la capacidad de llevar a cabo búsquedas multipatrón. *Nrgrep* se basa completamente en el algoritmo Backward Nondeterministic Dawg Matching (BNDM) [5]. Nrgrep es un software muy flexible y rápido para búsqueda de patrones complejos y expresiones regulares, exacta o aproximada.

En general *agrep* muestra un mejor desempeño que *nrgrep* para búsqueda en archivos de texto. Sin embargo, cuando se busca sobre archivos que contienen secuencias de ADN, el desempeño de *nrgrep* es superior al de *agrep*.

Capítulo 4

Algoritmos Propuestos

En esta sección presentamos los algoritmos que se implementaron para la búsqueda de patrones simples y de expresiones regulares ambas en texto comprimido con LZW. Al final de cada sección se presentan los resultados que se obtuvieron en las pruebas realizadas para cada uno de los algoritmos implementados.

4.1 Metodología de prueba

Para la ejecución de los experimentos que se realizaron en este trabajo, se utilizó una computadora con las características de 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.
- Compilador *gcc* versión 3.2

Para medir el desempeño de los algoritmos implementados se comprimieron archivos de texto de diferentes tamaños con el comando *compress* de Unix logrando

una compresión del 58%. Los archivos comprimidos son un conjunto parcial de títulos y/o resúmenes de 270 revistas médicas coleccionados en un periodo de 5 años (1987-1991)[1].

En las pruebas realizadas con el algoritmo de búsqueda aproximada, los patrones se escogieron de forma aleatoria del archivo de texto y se experimentó con longitudes del patrón de 15 hasta 80 caracteres, y con k igual a 1, 2 y 3 que son los casos más comunes. Para el algoritmo de búsqueda de expresiones regulares, se ejecutaron pruebas con expresiones regulares con diferentes tipos de operadores. En cada experimento se grafica la suma de los tiempos en segundos de CPU de usuario y sistema, tomando el promedio para cada largo del patrón.

4.2 Algoritmo de búsqueda aproximada en texto comprimido

La mayoría de los algoritmos para búsqueda de patrones en 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 lugar de una secuencia de caracteres. Las soluciones para búsqueda aproximada de patrones no son la excepción. De los esquemas expuestos en el capítulo 2 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. A continuación se explica a detalle cada una de estas fases.

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

4.2.1 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 de *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. Los argumentos de entrada son el patrón *p*, la longitud del patrón *m*, y el número de errores permitidos *k*. Primero se obtiene cuál es la longitud máxima que puede tener cada subpatrón. Después se obtienen todos los subpatrones con una longitud igual a la calculada anteriormente.

```
Dividir_patron (p = p1…pm, m, k)
    wlen ← m/k+1 /*calcular longitud máxima de cada subpatrón*/
    len ← 0;
    for i ϵ 0…m Do
       /* obtener subpatrón */
        str ← plen … plen+wlen
        almacenar_subpatron(str)
        len ← len + wlen
    EndFor
```

Figura 4.1 Seudocódigo del algoritmo para dividir el patrón en *k*+1 subpatrones

Los subpatrones se almacenan en una estructura de datos conocida como *trie*. Un ejemplo de un *trie* para las cadenas "patrón", "parse", "buscar" y "begin" se muestra en la Figura 4.2.

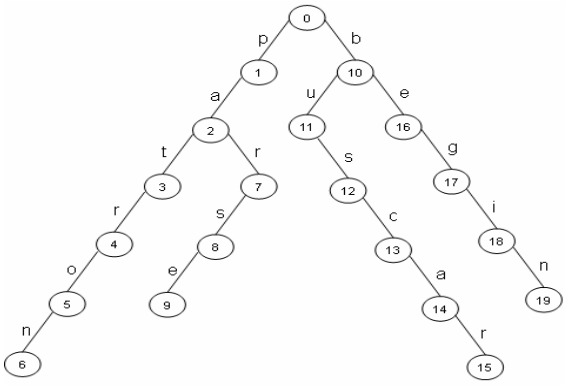


Figura 4.2 Trie generado para las cadenas 'patrón', 'parse', 'buscar', y 'begin'

En nuestra implementación, el *trie* se representó con una colección de arreglos (una matriz), en donde cada renglón corresponde a un estado y cada columna a un símbolo de entrada. La razón es que una búsqueda en un *trie* representado de esta forma es sumamente rápida.

Una vez que se ha dividido el patrón en k+1 subpatrones, se continúa la búsqueda con la ejecución de una búsqueda multipatrón. Este proceso se explica a continuación.

4.2.2 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 4.3 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.

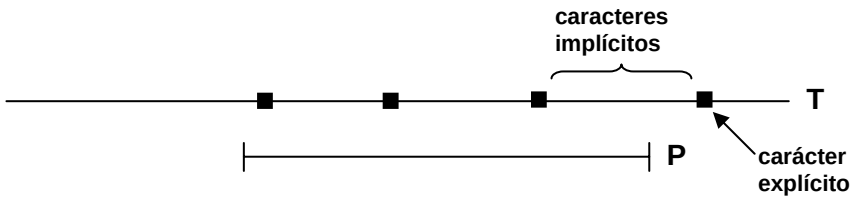


Figura 4.3 Ventana de un texto comprimido con LZ78 o LZW

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 sin omitir ninguna coincidencia, se precalcula la siguiente tabla:

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

Lo que 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 con ningún bloque explicito se logra un desplazamiento, se comienzan a considerar los caracteres implícitos. El orden en que son visitados cada uno de los bloques se define en la Figura 4.4. El último bloque se visita hasta el final dado que es costoso alcanzar los caracteres relevantes. Cada que se abre un bloque, se obtiene un carácter y por cada carácter se obtiene un posible desplazamiento. Si el desplazamiento obtenido es mayor que cero inmediatamente se avanza la ventana y se reejecuta el proceso completo con la nueva ventana.

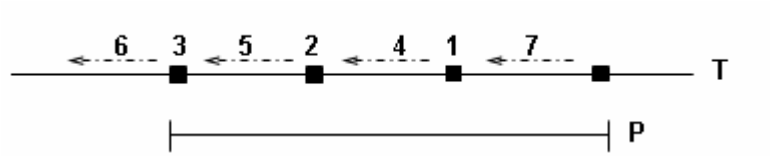


Figura 4.4 Orden en que se evalúan los bloques. Primero se leen los caracteres explícitos de derecha a izquierda, luego los implícitos de derecha a izquierda dejando al final el último bloque

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. Todas las ventanas se alinean al último bloque leído tal como lo muestra la Figura 4.5.

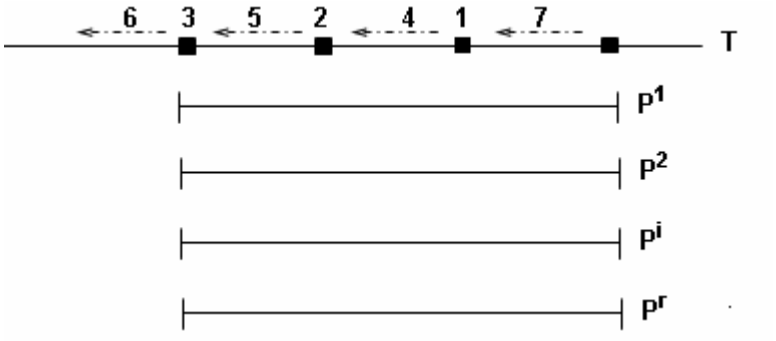


Figura 4.5 Ventana de búsqueda multipatrón. Los patrones se alinean con el último bloque de la ventana de búsqueda

Para conocer el desplazamiento posible para cada carácter leído de un bloque se redefine *B* de la siguiente manera:

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

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.

A diferencia de la versión monopatrón BM, con la versión multipatrón BM al terminar de revisar todos los bloques de la ventana sin obtener un desplazamiento positivo no es posible reportar una coincidencia de un subpatrón en la posición de la ventana, sino que es necesario verificar que el texto de la ventana coincide con algún subpatrón. Este proceso se realiza utilizando el *trie* generado en la fase anterior dando como cadena de entrada los caracteres del texto de la ventana. El seudocódigo completo para el proceso de búsqueda multipatrón efectuado en esta fase se muestra en el Apéndice 1.

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 completo. En caso de no ser así, la ventana se desplaza y continúa la búsqueda de otro subpatrón.

4.2.3 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. Utilizando P se precalculan los autómatas de paralelismo de bits de P_1 (invertido) y de P_2 para cada uno de los $k+1$ subpatrones, y se utilizan los autómatas correspondientes al subpatrón encontrado.

La Figura 4.6 muestra qué segmentos de P se utilizan para construir P_1 y P_2 para cada uno de los subpatrones. Como se observa, P_1 y P_2 son máscaras de bits donde cada bit representa un estado. Se inicia el reconocimiento con P_1, pues es menos costoso en tiempo verificar los bloques hacia la izquierda, dado que los caracteres se obtienen en el orden adecuado para alimentar al autómata. Para simular el autómata se utilizó el algoritmo propuesto en [19], donde el autómata se simula por medio de la fórmula (1) especificada en el capítulo 2.

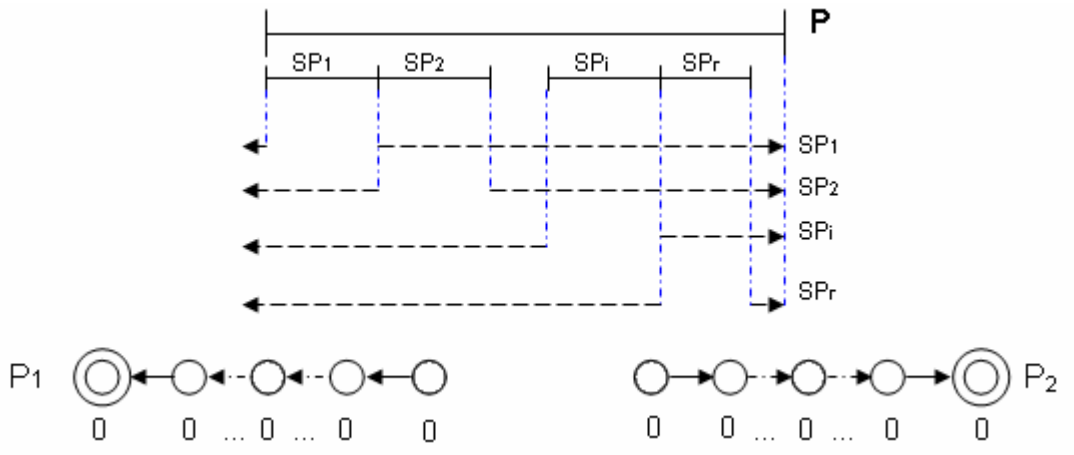


Figura 4.6 Representación de los autómatas P_1 y P_2 para cada uno de los subpatrones generados

Para alimentar el autómata P_1 se inicia con el bloque en donde comienza el subpatrón *F*. En la Figura 4.7 el bloque *j* representa el bloque donde inicia el subpatrón (el cual está "contenido" dentro del bloque *jj*). A partir del bloque *j* se van obteniendo todos los caracteres de los bloques referenciados por *j* hasta encontrar un bloque de longitud menor o igual a uno.

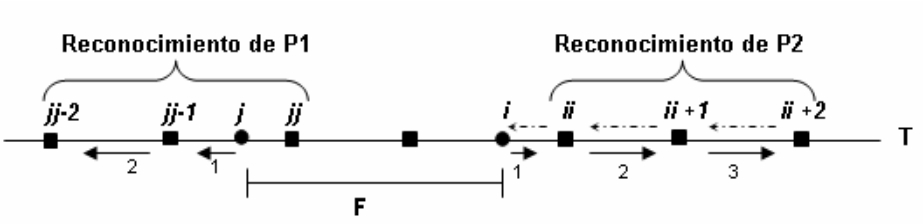


Figura 4.7 Orden en que se obtienen los caracteres para alimentar al autómata en el proceso de verificación

Al mismo tiempo se va alimentando el autómata con los caracteres obtenidos de cada bloque. Si con los caracteres obtenidos el autómata no llega a un estado final y el número de errores encontrados hasta ese momento es menor o igual que *k*, se prosigue a obtener 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 es mayor que *k*. El seudocódigo para reconocer P_1 se muestra en la Figura 4.8

```
While (true) Do
   While bj(length) > 1 Do
      a ← bj(c) /*se obtiene el carácter explicito de bj */
      (e,k') ← alimentar_automata (a)
      if (k' > k ) then return 0 /* falló la verificación */
      if (e es un estado final) then
```

```
        return 1 /* se reconocio P1 */
      bj ← bj(s) /* se obtiene el bloque referenciado por bj*/
   EndWhile
   bj ← bjj
EndWhile
```

Figura 4.8 Seudocódigo para reconocer P_1

Si el número de errores encontrados en este proceso k' es mayor que k, se continúa la búsqueda de otro subpatrón. En caso de que $k' \leq k$ 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 4.7 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 primero al bloque ii y a partir de éste ir hacia atrás en la cadena de referenciamiento hasta encontrar el bloque i, almacenando en un arreglo los caracteres 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 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. El seudocódigo del algoritmo para el reconocimiento de P_2 se muestra en la Figura 4.9.

```
temp ← bii
h ← 0
While temp ≠ bi Do
   A[h] ← temp(c) /*almacena en un arreglo los caracteres explícitos*/
   temp ← temp(s) /* se obtiene el bloque referenciado por temp */
```

```
   h ← h + 1
EndWhile
While (true) Do
   While h ≠ 0 Do
      (e,k'') ← alimentar_automata (A[h])
      if (k'' > k-k' ) then return 0 /*falló la verificación */
      if (e es un estado final) then
         return 1 /* se reconocio P2 */
      h ← h-1
   EndWhile
   bii ← bii + 1
   temp = bii
   While temp(length) > 1 Do
      A[h]← temp(c)
      temp ← temp(s)
      h ← h + 1
   EndWhile
EndWhile
```

Figura 4.9 Seudocódigo para reconocer P_2

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

4.2.4 Resultados experimentales

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.

El primer caso de prueba se hizo tomando el caso más común, es decir, la longitud del patrón está entre 15 y 30 caracteres y sólo hay un error en la coincidencia del patrón con el texto. El archivo que se utilizó para esta prueba tenía un tamaño de 20 MB en su forma no comprimida y de 8 MB en su forma comprimida. Los resultados que se obtuvieron se muestran en la Figura 4.10.

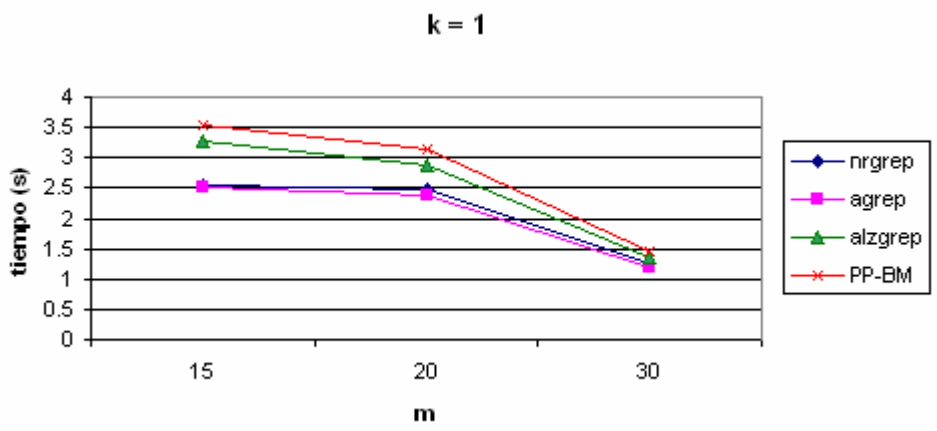


Figura 4.10 Tiempo de búsqueda en segundos para los diferentes algoritmos sobre un archivo de texto, para $k = 1$ y $m = 15, 20, 30$

Aunque los resultados obtenidos en esta prueba muestran que *alzgrep* tiene un mejor desempeño que el algoritmo original (*PP-BM*), no supera a las dos herramientas que primero descomprimen el texto y luego realizan la búsqueda. Sin embargo, se observa que el desempeño de *alzgrep* tiende a mejorar conforme la longitud del patrón aumenta.

Tomando en consideración lo último, se realizó una prueba en donde se incrementa la longitud del patrón en rangos de 10 caracteres hasta alcanzar una longitud de 80, manteniendo el mismo número de errores permitidos en la coincidencia. Los resultados que se obtuvieron se muestran en la Figura 4.11. No se

graficaron los resultados obtenidos para *PP-BM*, dado que *alzgrep* demostró ser superior en la prueba anterior y sólo es de interés verificar si puede superar a *nrgrep* y *agrep*.

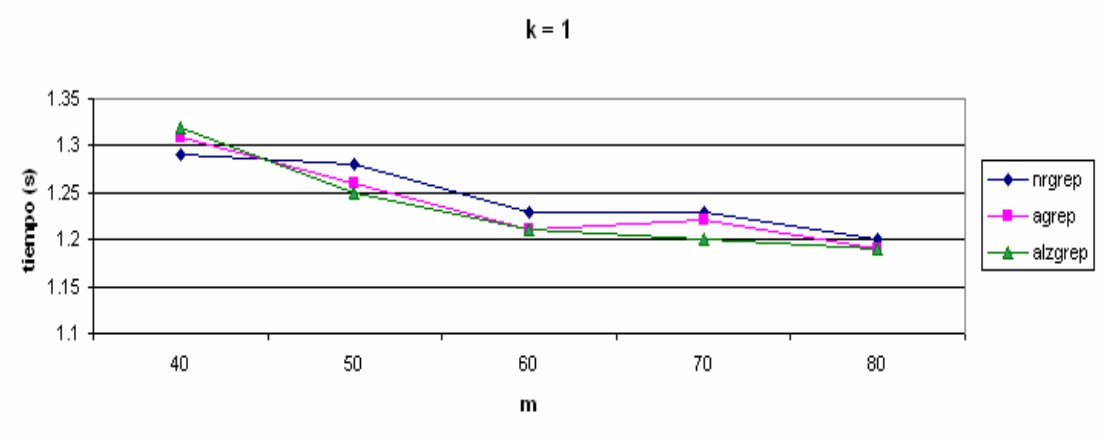
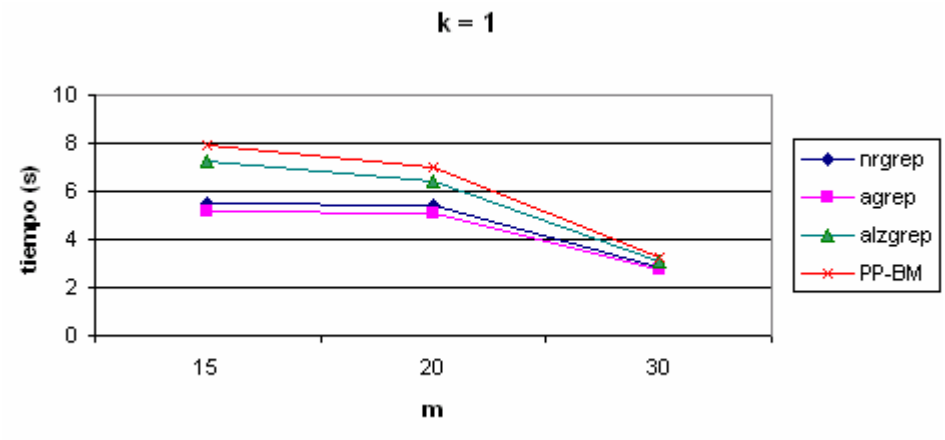


Figura 4.11 Tiempo de búsqueda en segundos para los diferentes algoritmos sobre un archivo de texto, para *k* = 1 y *m* = 40, 50, 60, 70 y 80

En esta prueba los resultados muestran que *alzgrep* tiene un mejor desempeño que *nrgrep* y *agrep* cuando la longitud del patrón es mayor a 40 caracteres. Dado que la diferencia en el tiempo de búsqueda es muy pequeña entre los algoritmos, se decidió hacer pruebas aumentando el tamaño del archivo comprimido y conocer si el desempeño de *alzgrep* mejoraba o empeoraba con respecto a los demás algoritmos. La prueba se realizó con archivos de texto que tenían un tamaño de 40 MB y 80 MB en su forma no comprimida y de 15 MB y 35 MB en se forma comprimida respectivamente.

Los resultados obtenidos con el archivo de 15 MB en su forma comprimida se muestran en la Figura 4.12. Al igual que en las pruebas anteriores, el número de errores permitidos en la coincidencia del patrón con el texto es de uno.



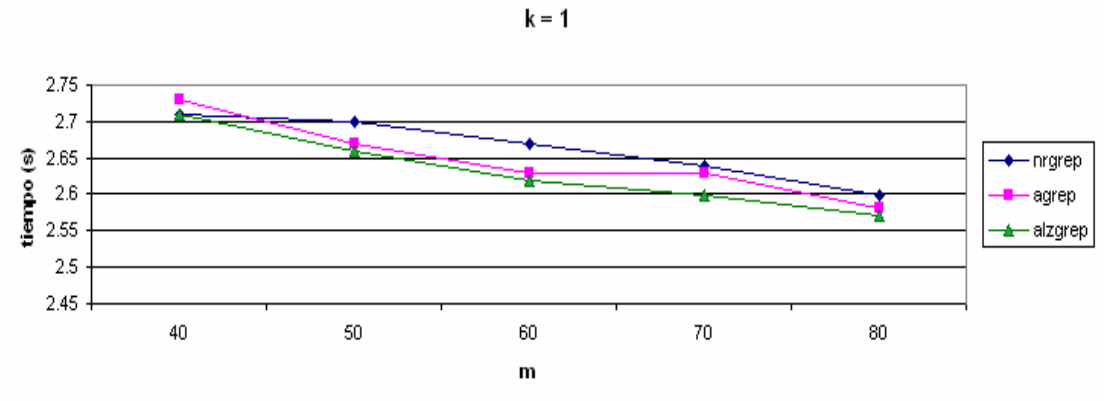


Figura 4.12 Tiempo de búsqueda de los algoritmos sobre un archivo de texto comprimido de 15 MB, para $k = 1$ y $m = 15, 20, 30, 40, 50\ 60, 70$ y 80

Los resultados obtenidos para el archivo de 35 MB en su forma comprimida se muestran en la Figura 4.13.

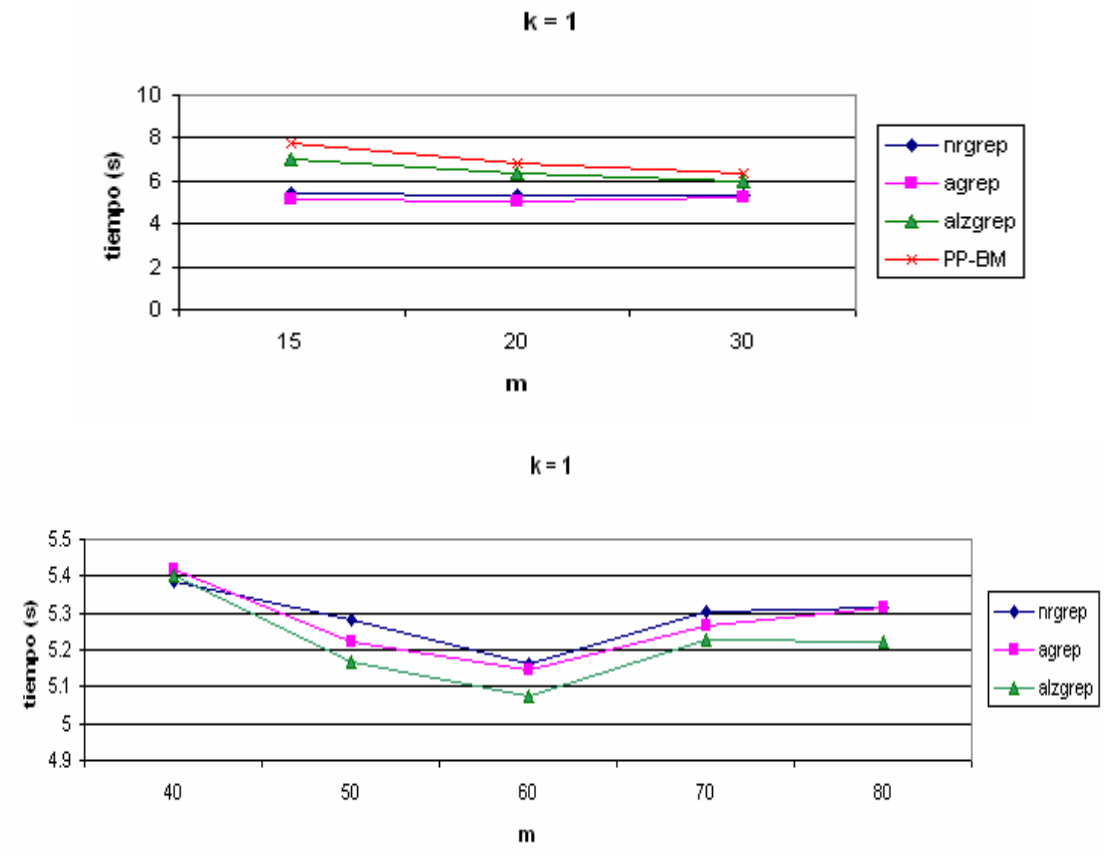


Figura 4.13 Tiempo de búsqueda de los algoritmos sobre un archivo de texto comprimido de 35 MB, para *k* = 1 y *m* = 15, 20, 30, 40, 50 60, 70 y 80

Como se puede observar en las gráficas, el desempeño de *alzgrep* se mantiene, superando siempre a *nrgrep* y *agrep* cuando la longitud del patrón es mayor a 40 caracteres.

Tomando en consideración que *alzgrep* funciona mejor para longitudes largas del patrón, se realizaron pruebas con archivos que contenían secuencias de ADN, dado que es más común buscar un patrón largo en un archivo de este tipo que en un

archivo de texto. El archivo que se utilizó tenía un tamaño de 98 MB en su forma no comprimida y de 28 MB en su forma comprimida. Los resultados que se obtuvieron se muestran en la Figura 4.14. Únicamente se realizó la prueba con largos del patrón de 60 a 80 caracteres puesto que hasta el momento es donde mejor desempeño ha mostrado *alzgrep*.

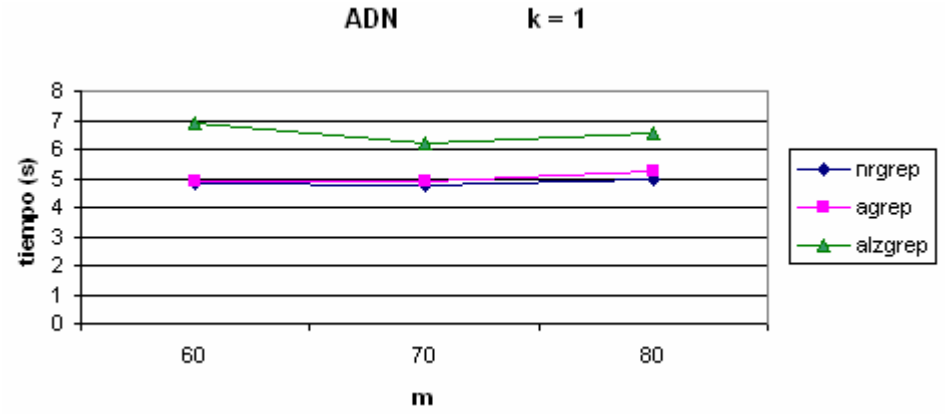


Figura 4.14 Tiempo de búsqueda de los algoritmos utilizando un archivo que contiene ADN para *k* = 1 y *m* = 60, 70 y 80

Los resultados obtenidos muestran que el desempeño de *alzgrep* disminuye para archivos que contienen secuencias de ADN en comparación al desempeño mostrado para archivos de texto. La razón se debe a que la compresión que se logra para un archivo que contiene secuencias de ADN es mayor a la que se logra con archivos de texto, por lo tanto, al efectuar la búsqueda, la posibilidad de que haya una coincidencia dentro de un bloque se incrementa, lo que hace que se realicen un mayor número de verificaciones.

Hasta el momento *alzgrep* ha mostrado un mejor desempeño que *nrgrep* y *agrep* cuando la longitud del patrón es mayor a 40 caracteres y sólo se permite un error en la coincidencia del patrón con el texto. En base a esto, se realizó la siguiente prueba, que fue aumentar el número de errores permitidos. Los resultados obtenidos

se muestran en la Figura 4.15. El archivo de texto que se utilizó para esta prueba fue el de 35 MB en su forma comprimida.

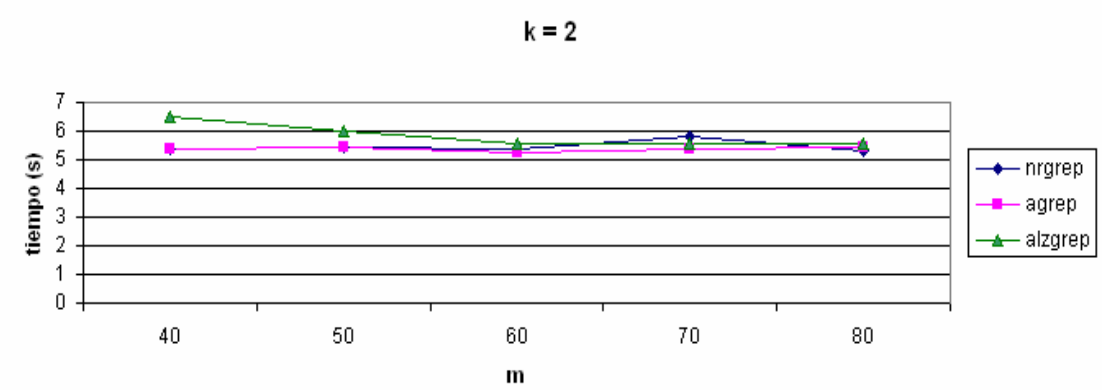


Figura 4.15 Tiempo de búsqueda de los algoritmos sobre un archivo de texto, para *k* = 2 y *m* = 40, 50, 60, 70 y 80

Se puede observar como el desempeño de *alzgrep* es similar al desempeño de *nrgrep* y *agrep* cuando el largo del patrón es mayor a 60 caracteres. Sin embargo, cuando el largo del patrón es inferior a 60 caracteres, nuevamente *alzgrep* es superado por *nrgrep* y *agrep*.

Ahora se incrementa a tres el número de errores permitidos en la coincidencia del patrón con el texto. La prueba se realiza con largos del patrón de 60, 70 y 80 caracteres. Los resultados obtenidos se muestran en la Figura 4.16. En esta prueba *alzgrep* no logra supera en ningún caso a los tiempos logrados por *nrgrep* y *alzgrep*.

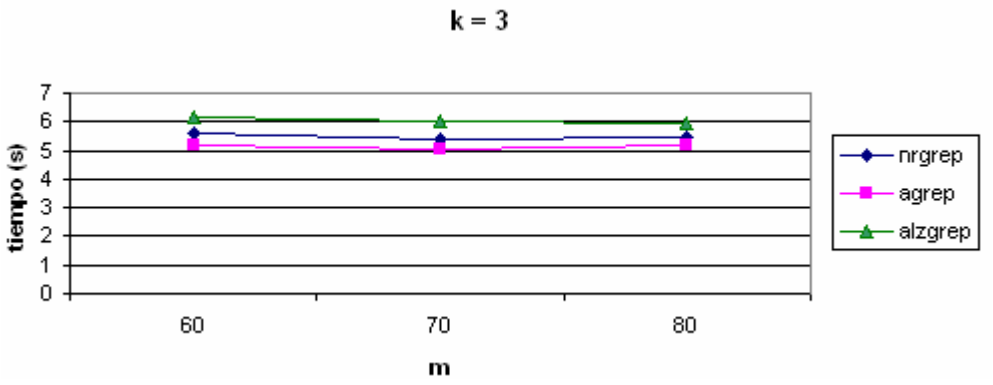


Figura 4.16 Tiempo de búsqueda de los algoritmos sobre un archivo de texto, para *k* = 3 y *m* = 60, 70 y 80

4.3 Algoritmo de búsqueda de expresiones regulares en texto comprimido

El algoritmo para búsqueda de expresiones regulares en texto comprimido con LZW consiste en obtener todos los prefijos de una longitud igual al largo de la cadena más corta que pertenezca al lenguaje generado por la expresión regular. Después de obtener todos los prefijos se realiza una búsqueda multipatrón de éstos y cada vez que se encuentra un prefijo se verifica si es una coincidencia de alguna cadena del lenguaje generado por la expresión regular por medio de un autómata finito determinístico, el cual se simula por medio de paralelismo de bits. La Figura 4.17 ilustra el proceso general de búsqueda. A continuación se explican a detalle cada una de sus etapas.

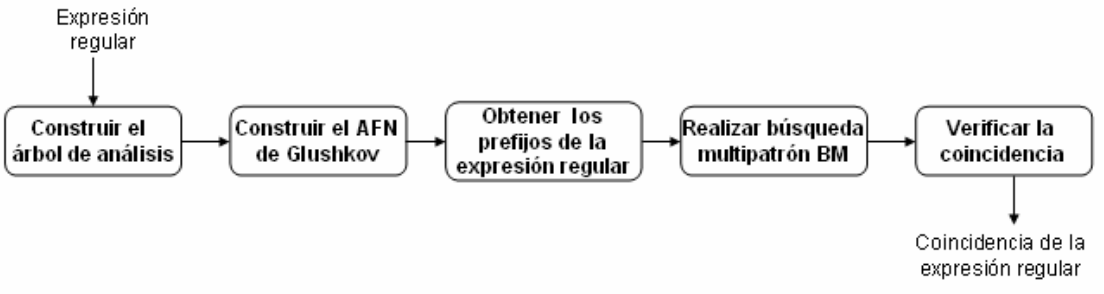


Figura 4.17 Proceso general de búsqueda de expresiones regulares en texto comprimido

4.3.1 Construir el árbol de análisis

Como se muestra en la Figura 4.16, el primer paso del algoritmo es construir un árbol de análisis a partir de la expresión regular. La construcción del árbol tiene diversos propósitos, como validar que la expresión regular esta escrita correctamente ó para posteriormente construir el AFN de Glushkov. En la Figura 4.18 se muestra el

seudocódigo que se utilizó para construir el árbol de análisis a partir de una expresión regular.

```
construccion_arbol (p = p1p2…pm, l, pos)
      v ← ø
      While p1 ≠ $ Do
            If p1 ϵ ∑ OR p1 = ε Then
                  vd ← Crear nodo con p1
                  vd(pos) ← pos   pos ← pos+1
                  If v ≠ ø Then v ← [·](v, vd)
                  Else v ← vd
                  l ← l + 1
            Else If p1 = '|' Then
                  (vd, l, pos) ← construccion_arbol(p, l+1,pos)
            v ← [|](v,vd)
            Else If p1 = '*' Then
                  v ← [*](v)
                  l ← l + 1
            Else If P1 = '(' Then
                  (vd,l, pos) ← construccion_arbol(p, l+1,pos)
                  l ← l + 1
                  If v ≠ ø Then v ← [·](v,vd)
                  Else v ← vd
            Else If p1 = ')' Then
                  Return (v, l, pos)
            EndIf
       EndWhile
      Return(v, l, pos)
```

Figura 4.18 Seudocódigo del algoritmo de construcción de un árbol de análisis a partir de una expresión regular

El seudocódigo asume que la expresión regular termina con el carácter especial '$', también se asume que el operador de concatenación '·' está implícito. El seudocódigo mostrado sólo es una guía para construir el árbol, pues en la implementación real maneja un número mayor de operadores. Además que al mismo tiempo se valida que la expresión regular esté bien escrita.

En vez de explicar en profundidad cómo trabaja el seudocódigo, lo cual puede resultar confuso por la recursividad que se emplea, se ejemplificarán los primeros

pasos sobre cómo obtener el árbol de análisis mostrado en la sección 2.4.2 para la expresión regular (aba|ba)a | a((ba)*)ab. Sólo se van mostrando los valores que van cambiando en el transcurso de la construcción del árbol.

Entrada: p = (aba|ba)a | a((ba)*)ab l=1 pos = 1
v = ǿ
Se lee '**(**'
Llamada recursiva l = 2

---

v = ǿ
Se lee '**a**'

$v_r \leftarrow$ ⓐ1 $v_r(pos) = 1$ pos =2

$v^2 \leftarrow v_r$ l =3

Como $p_L \neq$ \$ se ingresa nuevamente al ciclo while

Se lee '**b**'

$v_r \leftarrow$ ⓑ2 $v_r(pos) = 2$ pos =3

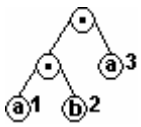
$v \leftarrow$ 

l =4

Se itera nuevamente

Se lee '**a**'

$v_r \leftarrow$ ⓐ3 $v_r(pos) = 3$ pos =4

$v \leftarrow$ 

l =5

El proceso para construir el árbol continúa de forma similar
⋮

Finalmente, al término de todas las llamadas recursivas el árbol que se genera es igual al que se muestra en la Figura 2.6. Cada nodo hoja tiene una variable *pos* la cual indica la posición del carácter en la expresión regular sin tomar en cuenta los operadores. Esta variable nos servirá posteriormente para generar las máscaras de bits que nos permitan simular el AFN de Glushkov.

4.3.2 Construir el AFN de Glushkov

Una vez generado el árbol de análisis para la expresión regular, se continúa con el cálculo de las variables de Glushkov las cuales se requieren para simular el AFN de Glushkov, para posteriormente generar el AFD, todo esto por medio de paralelismo de bits, es decir, se utilizan máscaras de bits donde el i^{th} bit es 1 para representar un estado activo y 0 en el otro caso.

La razón por la que se seleccionó la construcción de Glushkov en vez de la de Thompson es que para simular el AFD por medio de paralelismo de bits se necesita construir y almacenar una tabla de tamaño $2^{|Q|}$ donde Q es el número de estados del AFN y la construcción del autómata de Thompson genera más estados que la de Glushkov.

Retomando lo que se expuso en el capítulo 2, para construir el autómata de Glushkov se calculan las variables *PrimeraPos*, *UltimaPos* y *SiguientePos*. A partir de estas variables el AFN (Q , $\sum$, I, F, δ) se representa de la siguiente forma: Q = {0…*l-1*} (donde l es la longitud de la expresión regular), I = *Primerapos,* F = *UltimaPos*, δ = *SiguientePos*. Para calcular estas variables representadas por medio de paralelismo de bits, a cada nodo del árbol se le asocian las variables (máscaras de bits) *PrimeraPos* y *UltimaPos* y se calcula el conjunto *SiguientePos*, el cual representa las transiciones del AFN. Tomando nuevamente la expresión regular (aba|ba)a | a((ba)*)ab, el primer paso consiste en marcar la expresión regular (ver

sección 2.4.2), es decir, asignarle un índice a cada letra de la expresión regular de acuerdo a su orden de aparición. La expresión regular marcada es $(a_1b_2a_3|b_4a_5)a_6$ | $a_7()b_8a_9)$*=$a_{10}b_{11}$. Este proceso se representa en el árbol de análisis utilizando una máscara de bits *maskpos* para cada nodo del árbol. La Figura 4.19 muestra el seudocódigo para calcular los valores en cada nodo del árbol.

```
establecer_pos(v)
   If v(α) Then
      v(maskpos) ← 1 << v(pos)
   If v(*) Then
      establecer_pos(v1)
      v(maskpos) ← vi(maskpos).
   EndIf
   If v(|) or v(·) Then
      establecer_pos(vi)
      establecer_pos(vd)
       v(maskpos) ← vi(maskpos) OR vd(maskpos)
   EndIf
```

Figura 4.19 Seudocódigo para representar la expresión regular marcada por medio de paralelismo de bits

Al ejecutar el algoritmo de la Figura 4.19, la variable *maskpos* de los nodos hojas tendrán un solo bit en 1 en la posición que señale la variable *pos* del nodo respectivo, todos los demás bits estarán en cero.

Después se calculan las variables *PrimeraPos* y *UltimaPos* las cuales representan el comienzo o terminación de una cadena aceptada por la expresión regular respectivamente. Recordemos que para la expresión regular $(a_1b_2a_3|b_4a_5)a_6$ | $a_7(b_8a_9)$*$a_{10}b_{11}$, *PrimeraPos* = {1,4,7} y *UltimaPos* = {6,11}. La Figura 4.20 muestra el seudocódigo para calcular las variables *PrimeraPos* y *UltimaPos* mediante paralelismo de bits.

```
PrimeraUltima(v)
   If v(α)  Then
      v(PrimeraPos) ← 1 << v(pos)
   If v(*)  Then
     PrimeraUltima(v_i)
     v(PrimeraPos) ← v_i(PrimeraPos)
     v(UltimaPosPos) ← v_i(UltimaPos)
   EndIf
   If v(|) Then
     PrimeraUltima(v_l)
     PrimeraUltima(v_r)
     v(PrimeraPos) ← v_i(PrimeraPos) | v_d(PrimeraPos)
     v(ultimaPos) ←  v_i(UltimaPos) | v_d(UltimaPos)
   EndIf

   If v(·) Then
     PrimeraUltima(v_l)
     PrimeraUltima(v_r)
     v(PrimeraPos) ← v_i(PrimeraPos)
     v(UltimaPos) ←  v_d(UltimaPos)
   EndIf
```

Figura 4.20 Seudocódigo para calcular las variables *PrimeraPos* y *UltimaPos*

Los valores obtenidos para las variables *PrimeraPos* y *UltimaPos* del nodo raíz son las que finalmente representan el comienzo o terminación una cadena aceptada por la expresión regular respectivamente.

El árbol de la Figura 4.21 muestra el resultado de cada una de las variables que se obtuvieron para los nodos hojas y el nodo raíz para la expresión regular $(a_1b_2a_3|b_4a_5)a_6|a_7((b_8a_9)^*)a_{10}b_{11}$. Note que *maskpos* = *PrimeraPos* = *UltimaPos* para los nodos hojas.

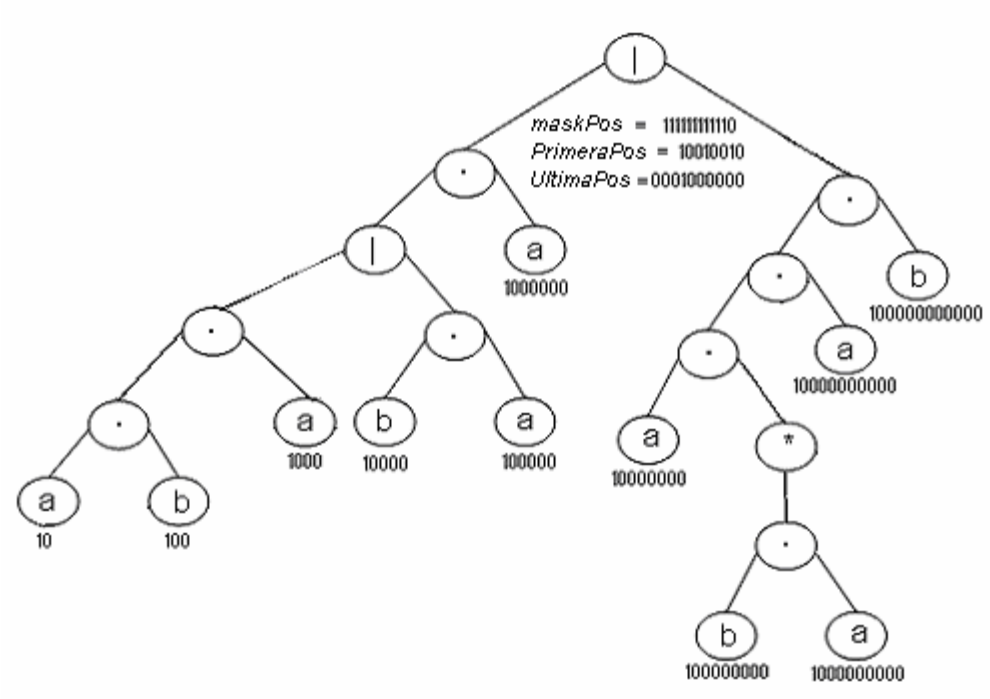


Figura 4.21 Valores de las hojas y el nodo raíz para las variables *maskpos*, *PrimeraPos* y *UltimaPos* para la expresión regular $(a_1b_2a_3|b_4a_5)a_6|a_7((b_8a_9)^*)a_{10}b_{11}$

Como se puede observar, *PrimeraPos* tiene un 1 en los bits con posición en {1, 4 y 7} contando de derecha a izquierda y siendo 0 la primera posición, que indican los estados iniciales de la expresión regular, y *UltimaPos* tiene un 1 en la posición {8 y 11}, que indican los estados finales de la expresión regular.

Ahora se calculan las transiciones del AFN, es decir, se calculan los valores de la variable *SiguientePos*. La Figura 4.22 muestra el seudocódigo que se utilizó para calcular los valores de la variable *SiguientePos*.

```
Siguiente_trans (v, pos)
     /* calcula el conjunto SiguientePos para la posición dada
     pos, es decir, el conjunto de estados que pueden seguir
     de la posición pos*/
  If v(α) Then return 0
  If v(*)Then
    If( v_i(UltimaPos) en la posición pos igual a 1) Then
```

```
      return(Siguiente_trans(v_i,pos) | v_i(PrimeraPos))
      Else return Siguiente_trans(v_i,pos)
   EndIf
EndIf

If v(|)Then
   If( v_i(maskPos) en la posición pos igual a 1) Then
      If( v_d(maskPos) en la posición pos igual a 1) Then
         tmp1 ← Siguiente_trans(v_i,pos)
         tmp2 ← Siguiente_trans(v_d,pos)
         tmp1 ← tmp1 | tmp2
         return tmp1
      EndIf
      Else
        return Siguiente_trans(v_i,pos)
   EndIf
   Else
      If (v_d(maskPos) en la posición pos igual a 1) Then
         return Siguiente_trans( v_d,pos)
      Else return 0
   EndElse
EndIf

If v(·) Then
   If( v_i(UltimaPos) en la posición pos igual a 1) Then
      tmp1 ← v_d(PrimeraPos)
   Else tmp1 ← 0
   If( v_i(maskPos) en la posición pos igual a 1) Then
      If( v_d(maskPos) en la posición pos igual a 1) Then
         tmp2 ← Siguiente_trans(v_i,pos)
         tmp1 ← tmp1 | tmp2
         tmp2 ← Siguiente_trans(v_d,pos)
         tmp1 ← tmp1 | tmp2
         return tmp1
      EndIf
      Else
         tmp2 ← Siguiente_trans(v_i,pos)
         tmp1 ← tmp1 | tmp2
        return tmp1
      EndElse
   EndIf
   Else
      If( v_d(maskPos) en la posición pos igual a 1) Then
         tmp2 ← Siguiente_trans(v_d,pos)
         tmp1 ← tmp1 | tmp2
         return tmp1
      EndIf
      Else return tmp1
   EndElse
```

```
   EndIf

Trans_AFN(v,L)
   for i ∈ 0…L Do
      If (i==0) Then SiguientePos[i] ← PrimeraPos
      Else   SiguientePos[i] ← Siguiente_Trans(v,i)
```

Figura 4.22 Seudocódigo para calcular las transiciones del AFN. Los valores se almacenan en la variable *SiguientePos*.

La variable *SiguientePos* se representa por medio de un conjunto de arreglos (una matriz). La Tabla 4-1 muestra los valores de la variable *SiguientePos* a partir de la expresión regular $(a_1b_2a_3|b_4a_5)a_6|a_7((b_8a_9)^*)a_{10}b_{11}$. La fila 0 representa la variable *PrimeraPos*, es decir, los estados con los que puede iniciar una coincidencia de la expresión regular. Las demás filas representan el conjunto de estados siguientes que se pueden alcanzar a partir del estado actual.

	11	10	9	8	7	6	5	4	3	2	1	0
0					1	0	0	1	0	0	1	0
1										1	0	0
2									1	0	0	0
3						1	0	0	0	0	0	0
4							1	0	0	0	0	0
5						1	0	0	0	0	0	0
6												0
7		1	0	1	0	0	0	0	0	0	0	0
8			1	0	0	0	0	0	0	0	0	0
9		1	0	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	0	0
11												0

Tabla 4-1 Transiciones del AFN para la expresión regular $(a_1b_2a_3|b_4a_5)a_6|a_7((b_8a_9)^*)a_{10}b_{11}$. Estos valores se almacenan en la variable *SiguientePos*

Aunque se puede realizar la búsqueda en texto comprimido utilizando el AFN (la variable *SiguientePos*), una forma más eficiente es convertir el AFN en un AFD. La búsqueda es más eficiente dado que el AFD sólo tiene un estado activo a la vez.

Para convertir el AFN a un AFD utilizando paralelismo de bits se almacena el vector de estados activos e inactivos en una máscara de bits. Así en vez de examinar los estados activos uno por uno, se utiliza una máscara de bits para indexar una tabla que dado el carácter actual proporciona el nuevo conjunto de estados activos (otra máscara de bits). Esto se puede considerar como una simulación de un AFN por medio de paralelismo de bits o como una implementación de un AFD.

Las transiciones del AFD se construyen a partir de los valores de la variable *SiguientePos* utilizando el algoritmo que se muestra en la Figura 4.23. La idea es representar por medio de un único estado el conjunto de estados activos del AFN. Las transiciones del AFD se almacenan en la variable *Td*, la cual contiene en la posición 1 los valores de la variable *PrimeraPos*. El tamaño de *Td* es 2^l donde *l* es la longitud de la expresión regular.

```
Construir T_AFD(SiguientePos,m)
   For i ∈ 0…m Do
      For j ∈ 0…2^i-1 Do
           Td[2^i+j] ← Td[j] | SiguientePos[i]
      EndFor
   EndFor
   Return Td
```

Figura 4.23 Seudocódigo para generar el AFD a partir de las transiciones del AFN

4.3.3 Obtener los prefijos de la expresión regular

La idea es generar todos los prefijos diferentes de longitud *lmin* para todas las cadenas que pertenecen al lenguaje generado por la expresión regular, donde *lmin* es la longitud minina de todas las cadenas que pertenece al lenguaje generado por la expresión regular. Para poder aplicar el algoritmo, la longitud mínima debe ser mayor a cero, en caso de que no sea así, la búsqueda se realizará descomprimiendo el texto para después utilizar un algoritmo clásico de búsqueda de expresiones regulares.

Para la expresión regular (aba|ba)a|a((ba)*)ab, *lmin* = 3 y los prefijos son {aba,baa,aab}. La Figura 4.24 muestra un seudocódigo del algoritmo para calcular *lmin*. El algoritmo recursivo consiste en visitar cada nodo de abajo hacia arriba. Para el caso donde un nodo sea una hoja se retorna un incremento a la variable *lmin*. Para el caso donde el nodo *v* representa al operador '|' se toma la longitud mínima de v_i (hijo izquierdo del nodo) y v_d (hijo derecho del nodo). Si el nodo representa al operador '·' se suman las longitudes de los nodos hijos. En caso de que el nodo represente nulidad (por ejemplo el operado '*') se le asigna un cero a la variable *lmin*.

```
Lmin(v)
      If v = [|](v_i, v_s) Then Return min(Lmin(v_i), Lmin(v_d))
      If v = [·](v_i, v_d) Then Return Lmin(v_i) + Lmin(v_d)
      If v = [*](v_*) Then Return 0
      If v = (α), α ∈ ∑ Then Return 1
      If v = (ε) Then Return 0
```

Figura 4.24 Seudocódigo del algoritmo para calcular la longitud mínima de una cadena que sea coincidencia con una expresión regular

Una vez calculado *lmin*, se obtienen los prefijos de la expresión regular de esa longitud. La Figura 4.25 muestra el seudocódigo para el algoritmo que obtiene estos prefijos. Una vez que todos los prefijos se han obtenido, el algoritmo de búsqueda de

expresiones regulares en texto comprimido ejecuta una búsqueda multipatrón Boyer-Moore con el conjunto de prefijos generados.

```
prefijos(lmin, pos, δ, s, Trie)
      For (s,σ,s') ∈ δ Do
            next ← Crear_nodo(Trie)
            next ← σ
            If pos > lmin Then
               Return (Trie)
            Else
                  Prefijos(lmin, pos+1, δ, s', next)
      EndFor
```

Figura 4.25 Seudocódigo para calcular todos los prefijos de longitud *lmin* de una expresión regular

4.3.4 Búsqueda multipatrón Boyer-Moore

El algoritmo multipatrón que se ejecuta en esta fase sigue la misma idea que el explicado en la sección 4.2. Es por eso que omitimos explicarlo nuevamente. Una vez que se ha localizado un prefijo, para que una cadena sea una coincidencia de la expresión regular debe iniciar con el prefijo encontrado, por lo tanto es necesario un proceso de verificación para comprobar si realmente hay una coincidencia.

4.3.5 Verificación

El proceso de verificación es sumamente sencillo, la idea es ir obteniendo los caracteres de los bloques e ir alimentado al AFD construido en la fase previa. Si se alcanza un estado final, se reporta una coincidencia con la expresión regular. Se hace uso de una tabla $B[\alpha]$, la cual representa que estados se pueden alcanzar por medio del carácter α a partir de cualquier estado. Los valores de esta tabla para la expresión regular (aba|ba)a|a((ba)*)ab, se muestran en la Tabla 4-2, se puede observar que únicamente los caracteres ‘a’ y ‘b’ tienen bits activos, esto dado que la expresión

regular sólo contiene estos caracteres. Para cualquier otro carácter α todos los bits estarán en cero.

	11	10	9	8	7	6	5	4	3	2	1	0
a	0	1	1	0	1	1	1	0	1	0	1	0
b	1	0	0	0	0	0	1	0	1	0	0	0
α	0	0	0	0	0	0	0	0	0	0	0	0

Tabla 4-2 Valores generados para la tabla *B*[α]. Esta tabla representa que estados se pueden alcanzar por el carácter α a partir de cualquier estado

De esta forma, la verificación inicia en la posición 1 de T_d el cual representa el estado inicial del AFD. Mediante una operación OR con *B*[α], podemos saber cuál es el siguiente estado activo (el cual es un conjunto de estados del AFN). Si este es un estado final (es decir, alguno de los estados del AFN es final), se reporta una coincidencia de la expresión regular. El seudocódigo para el algoritmo de verificación se muestra en la Figura 4.26.

```
temp ← b_ii
h ← 0
While temp ≠ b_i Do
   A[h]← temp(c) /*almacena en un arreglo los caracteres explícitos*/
   temp ← temp(s) /* se obtiene el bloque referenciado por temp */
   h ← h + 1
EndWhile

D ← 0^m1
While (true) Do
   D ← Td[D] & B[A[h]];
   If (D = 0) Then  /*fallo la verificación */
   If ((D & F_n) ≠ 0^(m+1)) Then Reportar una coincidencia de la ER
   If h = 0 Then
      b_ii ← b_ii + 1
      temp = b_ii
      While temp(length) > 1 Do
         A[h]← temp(c)
         temp ← temp(s)
         h ← h + 1
```

```
      EndWhile
   EndIf
EndWhile
```

Figura 4.26 Seudocódigo del algoritmo para la verificación de una coincidencia de una expresión regular en texto comprimido con LZW

Como se puede observar, el proceso de verificación termina de dos formas, ya sea que con el carácter leído no se avance a un nuevo estado, o que se encuentre un estado final, en tal caso se reporta una coincidencia del patrón y se continúa con la búsqueda de otro prefijo.

4.3.6 Resultados experimentales

La implementación del algoritmo de búsqueda de expresiones regulares en texto comprimido se integró al sistema *alzgrep* y se denominó *eralzgrep* (*alzgrep* extendido con búsqueda de expresiones regulares). Los patrones que se utilizaron en las pruebas son expresiones regulares que contienen diferentes operadores y que en ocasiones expresan la misma búsqueda, con el objeto de conocer con que operadores se obtiene un mejor desempeño o de que manera se afecta al algoritmo de búsqueda.

Los resultados que se obtienen para este tipo de búsqueda son más complicados de interpretar que el caso de búsqueda aproximada, pues depende de diversos factores como son el número de operadores, la forma en que se especifica el patrón a buscar, la longitud mínima de una cadena que sea coincidencia con la expresión regular, etc.

La tabla 4-3 muestra los patrones que se utilizaron en las pruebas así como los tiempos de búsqueda que se obtuvieron. Los resultados obtenidos se compararon contra los tiempos de búsqueda de las herramientas *agrep* y *nrgrep* las cuales

permiten realizar búsqueda de expresiones regulares en texto sin comprimir y contra el único software disponible de búsqueda directa en texto comprimido, el cual hemos denominado *PB* (por la técnica de paralelismo de bits que utiliza).

Patrón	# de caracteres	*min-len*	Tiempo (s)			
			nrgrep	*agrep*	*PB*	*eralz-grep*
1. American|Mexican	15	7	5.04	5.61	14.17	6.44
2. American|Mexican|Canadian	23	7	5.07	5.89	17.37	9.63
3. Amer[a-z]*can|Me[a-z]*can	14	5	5.41	6.00	18.09	10.91
4. Ame(i|(r|i)*)can	9	6	5.27	5.34	14.54	6.70
5. A(mer|i)+can	8	8	4.98	5.22	15.05	8.66

Tabla 4-3 Resultados obtenidos con el algoritmo propuesto de búsqueda de expresiones regulares en texto comprimido

Los resultados obtenidos muestran que el algoritmo que se implementó logra tiempos de búsqueda muy cercanos a los que logran *nrgrep* y *agrep*, sin embargo, no los supera en ningún caso de búsqueda. Por otra parte, supera ampliamente en todos los casos al algoritmo que realiza la búsqueda directa en texto comprimido.

En la Tabla 4-3 los patrones 3, 4 y 5 expresan la misma búsqueda, sin embargo el tiempo que ocupa *eralzgrep* para ejecutar la búsqueda varia significativamente. Esto se debe al número de prefijos que genera lo cual depende en su mayor parte de los operadores que se utilicen. Como se puede observar, la búsqueda es más lenta cuando se utilizan los operadores que especifican un rango de caracteres (corchetes), lo que sugiere que sólo se utilicen en caso de que la búsqueda no se pueda expresar de forma diferente.

4.4 Resumen de resultados

En esta sección se expusieron dos algoritmos para búsqueda de patrones en texto comprimido. El primer algoritmo resuelve el problema de búsqueda de patrones simples en texto comprimido y el segundo resuelve el problema de búsqueda de expresiones regulares en texto comprimido. Ambos algoritmos se integraron en un sistema de búsqueda el cual es el primero en su tipo, pues lleva a cabo todo el proceso de búsqueda sin descomprimir el texto en ningún momento.

Los resultados obtenidos para búsqueda aproximada de patrones simples en texto comprimido demuestran que el algoritmo que se implementó obtiene mejores resultados que las mejores herramientas de búsqueda clásicas disponibles actualmente, las cuales primero descomprimen el texto y después realizan la búsqueda. El algoritmo obtiene sus mejores tiempos para patrones largos (mayor que 40) 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.

Los resultados obtenidos para el caso de búsqueda de expresiones regulares en texto comprimido no superaron a los que se obtienen con las mejores herramientas que permiten búsqueda de expresiones regulares que primero descomprimen el texto y después realizan la búsqueda. Sin embargo, se obtiene el primer sistema que realiza todo el proceso de búsqueda de expresiones regulares sin descomprimir el texto con tiempos de búsqueda similares a las herramientas contra las que fue comparado. Por lo cual este trabajo es un buen punto de partida para trabajos futuros.

Capítulo 5

Conclusiones y trabajo futuro

En este capítulo se detallan las conclusiones a las que se llegó después de realizar la tesis. También se mencionan los trabajos futuros que pueden ser estudiados para lograr una mayor eficiencia en los algoritmos de búsqueda de expresiones y búsqueda aproximada en texto comprimido y aumentar la flexibilidad del sistema de búsqueda que se desarrolló.

5.1 Conclusiones

Para alcanzar los objetivos que se plantearon al iniciar este trabajo de tesis, se hizo un estudio y análisis de las diferentes técnicas de búsqueda de patrones disponibles actualmente tanto para texto sin comprimir como para texto comprimido. También se estudiaron los diferentes métodos de compresión, en particular los de la familia Ziv-Lempel. A partir de esto, se propusieron dos algoritmos los cuales combinan diferentes técnicas de búsqueda con la finalidad de acelerar cada parte del proceso de búsqueda y obtener un algoritmo competitivo.

Con los resultados que se obtuvieron de los experimentos realizados, se puede concluir que los algoritmos propuestos son una excelente opción para ser utilizados en diversas aplicaciones prácticas, puesto que se puede efectuar la búsqueda con tiempos competitivos a los realizados por los mejores algoritmos para búsqueda aproximada y de expresiones regulares actualmente disponibles y mantener las colecciones de texto comprimidas, pues 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.

También se implementó una herramienta de búsqueda para comprobar la eficiencia práctica de los algoritmos expuestos, lo cual pocos trabajos logran. Esta herramienta se denominó *eralzgrep* y posteriormente estará disponible públicamente. Esta herramienta la primera en su tipo, pues hasta antes de este trabajo no existía ninguna herramienta que realizara estos dos tipos de búsqueda con la capacidad de ejecutar todo el proceso de búsqueda sin descomprimir el texto.

5.2 Trabajo Futuro

Con relación a los trabajos futuros, existen muchos caminos para explorar. Uno de ellos se refiere al algoritmo de búsqueda aproximada y a la distancia de edición que maneja. Sería interesante trabajar con funciones de distancia más complejas que la distancia de edición. El objetivo es que se permitan las transposiciones de caracteres, el cual es un error que se comete muy frecuentemente al escribir un texto. Con la distancia de edición utilizada en este trabajo, son necesarios dos errores para simular la transposición.

Algo más complejo sería explorar nuevas técnicas de búsqueda multipatrón que se puedan adaptar para que trabajen en texto comprimido con LZW, y de esta forma acelerar aun más el proceso de búsqueda. Pues la parte de búsqueda que consume más tiempo es precisamente la búsqueda multipatrón, la cual se utiliza tanto para búsqueda aproximada como para búsqueda de expresiones regulares.

Finalmente, se puede trabajar para extender el software de búsqueda que se implementó en este trabajo, de forma que se pueda efectuar una búsqueda mucho más compleja como es la búsqueda aproximada de expresiones regulares en texto comprimido. Con esto, se obtendría una herramienta de búsqueda bastante completa, flexible y eficiente.

Referencias

[1] C. Blair. A program for correcting spelling errors. *Information and Control*, 3:60-67, 1960

[2] F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. of the ACM*, 7(3):171-176, 1964

[3] T. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4:52-58, 1968

[4] S. Wu and U. Manber. Agrep- a fast approximate pattern-matching tool. *In Proc. USENIX Technical Conference*, pages 153-162, 1992

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

[6] 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

[7] M. Waterman. *Introduction to Computacional Biology*. Chapman and Hall, 1995

[8] W. Beyer, M Stein, T. smith, and S. Ulam. A molecular sequence metric and evolutionary trees. *Mathematical Biosciences*, 19:9-25, 1974

[9] W. Wilber and D. Lipman. Rapid similarity searches in nucleic acid and protein data banks. *In Proc. Of the National Academy of Sciences of the USA*, Volume 80, pages 726-730, 1983

[10] R. Dixon and T. Martin, editors. *Automatic speech and speaker recognition*. IEEE Press, 1979

[11] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8-17, 1965

[12] T. Luczak and W. Szpankowski. A suboptimal lossy data compression based on approximate pattern matching. *IEEE Trans. On Information Theory*, 43:1439-1451, 1997

[13] T. A. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8-19, 1984

[14] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20(10):762-772, 1997

[15] G. Davies and S. Bowsher. Algorithms for pattern matching. *Software--Practice and Experience*, 16(6):575-601, 1986

[16] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAMS Journal on Computing*, 6(1):323-350, 1977

[17] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In N. J. Belkin and C. J. van Rijsbergen, editors, *Proceedings of the 12th International Conference on Research and Development in Information Retrieval*, pages 168-175. ACM Press, 1989

[18] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359-373, 1980

[19] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83-91, 1992

[20] K. Thompson. Regular expression search algorithm. Communications of the ACM, 11:419-422, 1968

[21] V-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1-53, 1961

[22] G. Berry and R. Sethi. From regular expression to deterministic automata. *Theoretical Computer Science*, 48(1):117-126, 1986

[23] A. Amir and G. Benson. Efficient two dimensional compressed matching. In *Proc. Second IEEE Data Compression Conference*, pages 279-288, 1992

[24] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Of the Institute of Electronic and Radio Engineers*, volume 40, pages 1090-1101, 1952

[25] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Transactions on Information Systems*, 15(2): 124-136, 1997

[26] A. Turpin and A.Moffat. Fast file search using text compression. In *Proc. 20th Australian Computer Science Conference*, pages 1-8, 1997

[27] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions of Information Systems* (TOIS), 2000. To appear. Earlier versions in SIGIR'98 and SPIRE'98

[28] 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

[29] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algoritmica*, 20:388-404, 1998

[30] 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

[31] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. Of the ACM*, 18(6):333-340, June 1975

[32] 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

[33] 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

[34] G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. CPM'2000*, LNCS, 2000

[35] 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

[36] 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

[37] 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

[38] Gonzalo Navarro. Regular Expression Searching on Compressed Text. *Journal of Discrete Algorithms (JDA)* 1(5/6):423-443, 2003

[39] R. N. Horspool. Practical fast searching in string. *Software Practice and Experience.* 10(6):501-506, 1980