UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SOLUCIÓN DE CONSULTAS COMPLEJAS EN UN ÍNDICE DE TEXTO COMPRIMIDO

PEDRO IGNACIO MORALES CASTILLO

COMISIÓN EXAMINADORA		CALIFICACIONES: NOTA (n^o)	(LETRAS)	FIRMA
PROFESOR GUÍA SR. GONZALO NAVARRO	:	.....	.....	.....
PROFESOR CO-GUÍA SR. CLAUDIO GUTIÉRREZ	:	.....	.....	.....
PROFESOR INTEGRANTE SR. DIONISIO GONZÁLEZ	:	.....	.....	.....
NOTA FINAL EXAMEN DE TÍTULO	:	.....	.....	.....

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

SANTIAGO DE CHILE
ABRIL 2005

Resumen Ejecutivo

Actualmente la investigación en el área de índices apunta a reducir el espacio que usan estas estructuras. Esto da origen a los *autíndices comprimidos*, que además de usar menos espacio que un índice tradicional, no requieren una copia del texto disponible pues se puede recuperar a partir del índice. Estos índices sólo proveen de búsqueda exacta de patrones.

Este trabajo de título tuvo como objetivo agregar las funcionalidades necesarias a un autoíndice comprimido para poder realizar búsquedas aproximadas, sin necesidad de modificar la estructura del índice. Por búsqueda aproximada se entiende el poder buscar un patrón permitiendo k errores en las ocurrencias de éste.

La técnica elegida fue la de particionar el patrón en $k+1$ partes. Cada una de estas partes se busca en el índice y despues se verifica una ocurrencia del patrón original completo. El índice elegido fue el *LZ-index*, el cual está basado en el algoritmo de compresión de *Ziv-Lempel*. Se escogió este índice porque por su diseño permite recuperar el texto de un calce de manera rápida.

Se desarrollaron implementaciones de creciente nivel de complejidad, buscando reducir los tiempos de búsqueda. La primera fue una implementación básica del algoritmo sin optimizaciones. A continuación se implementaron mejoras aprovechando propiedades específicas del *LZ-index*. Finalmente se implementaron mejoras en la partición del patrón, las cuales ya eran conocidas pero se dificultaban notablemente en el *LZ-index*. Los resultados se compararon contra una implementación del mismo método sobre un autoíndice más clásico, el *FM-index*, basado en la transformada de *Burrows-Wheeler*.

Los experimentos se realizaron sobre cuatro colecciones de texto de distintos tipos y tamaños.

Los resultados muestran que la implementación sobre el *LZ-index* es siempre más rápida que la realizada sobre el *FM-index*.

Con respecto a la comparación con búsqueda secuencial, los resultados son similares a otros esquemas de búsqueda aproximada. Para textos muy pequeños no vale la pena indexar, pues la búsqueda secuencial es siempre más rápida. En el resto de los casos, para errores de hasta un 10 % del largo del patrón a buscar, la búsqueda indexada es más rápida que su contraparte sobre texto plano. A medida que aumenta el porcentaje de error, el desempeño se va deteriorando hasta el punto de que para un 20 % de error siempre resulta más rápido la búsqueda secuencial. El rango de error inferior al 20 % es el interesante en la mayoría de las aplicaciones.

Índice general

Capítulo 1

Introducción

Con la llegada de las comunicaciones digitales, grandes y siempre crecientes volúmenes de datos se encuentran disponibles de manera electrónica en los más diversos formatos. Muchos de éstos encuentran en el texto plano su mejor formato de representación, como son bases de datos sobre genoma y proteínas, código fuente de programas y bibliotecas digitales, por mencionar algunas.

Para una utilización cabal de éstos se hace indispensable el uso de algoritmos y estructuras eficientes que permitan extraer información mediante consultas y búsquedas. Es en este ámbito que se desarrolla la búsqueda en texto y la indexación.

Por búsqueda en texto entenderemos el problema de, dado un patrón y un texto, poder encontrar todas las posiciones en el texto donde se encuentre el patrón. Para esta tarea existen diversos algoritmos. En los más simples se recorre secuencialmente todo el texto en busca de una ocurrencia, y en los más elaborados a lo máximo que se puede aspirar es a saltar algunos caracteres del texto, haciendo la revisión más rápida. Aún así existen colecciones de datos para las cuales estas aproximaciones son demasiado lentas, ya sea debido al tamaño de la colección o por el número de consultas que en ésta se realizan. Para estas situaciones se hace necesario desarrollar soluciones particulares.

La *indexación* surge como respuesta a este problema. Indexar consiste en crear estructuras persistentes en base a los datos existentes. Éstas se diseñan para resolver consultas más rápido sin la necesidad de tener que revisar la colección de texto completa.

Uno de los ejemplos más populares de esta técnica en la recuperación de información la encontramos en el *índice invertido*. Éste corresponde a un índice diseñado para permitir búsquedas muy eficientes de palabras y secuencias de palabras (frases), con la ventaja adicional de que utilizan muy poco espacio. Una de sus

aplicaciones más conocidas es como índice de los motores de búsqueda Web.

El problema es que este diseño requiere poder definir una noción de palabra, algo que no es siempre posible. Por ejemplo el ADN y las proteínas, se representan por cadenas de caracteres sin separaciones y sin una división clara. Tampoco es aplicable a los textos en lenguajes orientales como son el japonés, chino y coreano y en bases de datos de música.

Para esto surgen los *índices de texto completo*, donde se puede buscar por cualquier cadena del texto. En general esto se logra almacenando todos los sufijos del texto, lo que implica un alto uso de memoria. Ejemplos de estos índices son el árbol de sufijos y el arreglo de sufijos.

El uso de memoria de estos índices los hace absolutamente inutilizables con colecciones de texto relativamente grandes. Por ejemplo, un texto de 1 Gigabyte (Gb) podría ocupar desde 4 Gb en un arreglo de sufijos, hasta 20 Gb en un árbol de sufijos. Es fácil ver que ésto rápidamente escapa a la memoria disponible incluso en sistemas actuales. A esto se suma que no existen representaciones eficientes de estas estructuras para su uso en memoria secundaria, y que aún si existieran se perdería el sentido de la indexación, puesto que los accesos a disco son órdenes de magnitud más lentos que los accesos a memoria.

Esta necesidad da origen a los *índices sucintos*. La idea de estas estructuras es poder disminuir el espacio utilizado manteniendo la eficiencia dentro de márgenes aceptables. Es así como se desarrolla el concepto de *estructura oportunista*, que es aquella cuyo uso de espacio disminuye si la entrada es compresible. Esta disminución se obtiene sin un castigo significativo en el desempeño de las consultas.

Sin embargo, aún contando con índices de memoria primaria, a veces es necesario ir a revisar el texto en disco, por ejemplo para reportar las ocurrencias, o verificar algún calce. Ésto repercute de manera negativa en el desempeño final de la solución. Como respuesta a ésto se desarrollaron los *autoíndices*, que corresponden a índices que además contienen el texto que indexan. De esta manera no es necesario ir a disco, aunque sí pagar un precio en tiempo computación y espacio utilizado por la estructura.

Ejemplos de autoíndices sucintos son el *FM-index*, el *CSA*, y el *LZ-index*, que varían según el principio en que se fundamentan.

Debido lo nuevo de la investigación en el área es que estos índices actualmente sólo permiten búsquedas de cadenas simples. Este tipo de búsqueda representa sólo una pequeña fracción de lo requerido actualmente. Por ejemplo, en áreas como la biología

computacional se utiliza casi únicamente la búsqueda de expresiones regulares y la búsqueda aproximada.

La búsqueda aproximada consiste en poder encontrar un patrón en el texto permitiendo un número máximo k de *diferencias* entre el patrón y las ocurrencias. Es interesante notar que existen distintas definiciones de qué es una diferencia, según el problema que se esté tratando de resolver.

El trabajo de esta memoria consistió en tomar el LZ-index y agregarle la búsqueda aproximada de patrones, con la restricción de que no se debía modificar la estructura actual del índice para esto. Se estudió la eficiencia de distintas soluciones y la competitividad de éstas con respecto a la solución secuencial sobre el texto sin indexar. Junto con esto se implementó la misma funcionalidad en un segundo autoíndice sucinto como manera de evaluar el desempeño de la estructura seleccionada dentro del ámbito de la búsqueda indexada.

1.1. Objetivos

1.1.1. General

El objetivo final de este trabajo de memoria es hacer las modificaciones necesarias para que el LZ-index pueda contestar consultas de búsqueda aproximada.

1.1.2. Específicos

- Implementar búsqueda aproximada de manera simple sobre el índice.
- Implementar búsqueda aproximada con optimizaciones sobre el índice.
- Implementar búsqueda aproximada de manera simple en un segundo índice para realizar experimentos comparativos.
- Conducir experimentos para evaluar la eficiencia de las soluciones.

Capítulo 2

Conceptos Básicos

2.1. Búsqueda en texto

Definiremos la búsqueda en texto o calce de patrones como el problema de encontrar todas las ocurrencias del patrón $P = p_1 p_2 \dots p_m$ en el texto $T = t_1 t_2 \dots t_n$, $n > m$, ambos secuencias de caracteres sobre un alfabeto Σ.

2.1.1. Búsqueda aproximada

Por búsqueda aproximada se entenderá el problema de encontrar calces de un patrón en el texto permitiendo un número máximo k de diferencias entre el patrón y sus ocurrencias.

Para esto es necesario un modelo que defina qué es una "diferencia". El modelo más popular es el de *distancia de edición* o *distancia de Levenshtein* [8].

Bajo este modelo una diferencia corresponde a una *operación de edición* que puede ser la inserción, borrado o substitución de un carácter. Es decir, la distancia de edición entre dos strings t_1 y t_2, $ed(t_1, t_2)$ es el mínimo número de operaciones de edición necesarias para convertir t_1 en t_2, o viceversa. Por ejemplo, ed(azabar, alabarda) = 3, reemplazando la "z" por "l" y agregando "da" al final del texto.

Para resolver este problema existen distintas soluciones. Revisaremos dos algoritmos: el más antiguo que existe para este propósito y que es el más flexible, la *programación dinámica* y uno de los algoritmos más eficientes en la práctica, *particionamiento en* $k + 1$ *partes*

Programación dinámica

Previo a entender la búsqueda usando este algoritmo, primero debemos revisar cómo se computa la distancia de edición entre dos cadenas x e y.

Para esto, se llena una matriz $M_{0...|x|,0...|y|}$, donde $M_{i,j}$ representa el mínimo número de operaciones de edición para calzar $x_{1...i}$ con $y_{1...j}$, es decir, $M_{i,j} = ed(x_{1...i}, y_{1...j})$. Esto se computa de la siguiente manera:

$$\begin{aligned} M_{0,0} &= 0 \\ M_{i,j} &= \text{mín}(M_{i-1,j-1} + \delta(x_i, y_i), M_{i-1,j} + 1, M_{i,j-1} + 1) \end{aligned}$$

donde $\delta(a, b) = 0$ si $a = b$ y 1 si no.

La explicación de esta fórmula es la siguiente: $M_{0,0}$ es la distancia de edición entre dos strings vacíos. Para dos strings de largo i y j suponemos inductivamente que todas las distancias entre strings más cortos han sido calculadas, y tratamos de convertir $x_{1...i}$ en $y_{1...j}$.

Consideremos los caracteres x_i e y_j. Si son iguales, entonces no es necesaria ninguna operación, y el costo es el mismo que el de convertir $x_{1...i-1}$ en $y_{1...j-1}$, es decir $M_{i-1,j-1}$. Si son distintos, entonces puede hacerse una de tres cosas: reemplazar x_i por y_j y convertir $x_{1...i-1}$ en $y_{1...j-1}$ a un costo $M_{i-1,j-1} + 1$; borrar x_i y convertir $x_{1...i-1}$ en $y_{1...j}$ a un costo $M_{i-1,j} + 1$; o insertar y_j al final de $x_{1...i}$ y convertir $x_{1...i}$ en $y_{1...j-1}$ a un costo $M_{i,j-1} + 1$.

A partir de la matriz es posible determinar un *camino óptimo*, es decir, una secuencia de celdas de la matriz de costo mínimo que va de $M_{0,0}$ a $M_{|x|,|y|}$. En la figura 2.1 podemos ver como se vería la matriz de distancias de edición para $ed(azabar, alabarda)$.

		A	L	A	B	A	R	D	A
	0	1	2	3	4	5	6	7	8
A	1	**0**	1	2	3	4	5	6	7
Z	2	1	**1**	2	3	4	5	6	7
A	3	2	2	**1**	2	3	4	5	6
B	4	3	3	2	**1**	2	3	4	5
A	5	4	4	3	2	**1**	2	3	4
R	6	5	5	4	3	2	**1**	**2**	**3**

ALABARDA

AZABAR

Figura 2.1: Matriz de distancia de edición para $ed(azabar, alabarda)$.

La búsqueda de P en T es esencialmente lo mismo que computar la distancia

de edición con $x = P$ y $y = T$. La única diferencia es que debemos permitir que el calce comience en cualquier posición del texto. Esto se logra poniendo $M_{0,j} = 0, \quad \forall j \in 0 \ldots n$. Luego se reporta como calce toda posición dentro del texto para la cual $M_{m,*} \leq k$. Podemos ver esto en la figura 2.2.

		A	L	A	B	A	R	D	A
	0	0	0	0	0	0	0	0	0
A	1	0	1	0	1	0	1	1	0
Z	2	1	1	1	1	1	1	2	1
A	3	2	2	1	2	1	2	2	2
B	4	3	3	2	1	2	2	3	3
A	5	4	4	3	2	1	2	3	3
R	6	5	5	4	3	**2**	**1**	**2**	3

Figura 2.2: Ejemplo de matriz de búsqueda de programación dinámica para$k = 2$ y $m = 6$.

Particionamiento en $k + 1$ partes

El segundo algoritmo que nos interesa mencionar hace uso del hecho que, al buscar un patrón con k errores, si éste se divide en $k+1$ pedazos, entonces al menos uno de ellos debe aparecer sin errores en una ocurrencia.

La idea es entonces buscar estas partes con algún algoritmo de búsqueda exacta y luego revisar la vecindad de estas ocurrencias para verificar el calce. Hay que tomar las consideraciones adecuadas para evitar reportar más de una vez la misma ocurrencia.

La vecindad debe ser suficientemente grande para abarcar todas las ocurrencias. Las ocurrencias son a lo sumo de largo $m+k$. Si el pedazo del patrón $p_{i_1 \ldots i_2}$ calza en el texto en la posición $t_{j \ldots j+(i_2-i_1)}$, entonces la ocurrencia puede comenzar a lo más i_1-1+k posiciones antes de t_j. Esto es debido a que si las diferencias son inserciones, todas pueden ubicarse al comienzo de la ocurrencia. A la vez las ocurrencias pueden terminar a lo más $m - i_2 + k$ posiciones después de $t_{j+(i_2-i_1)}$, pues bajo la misma premisa, todas las inserciones pueden situarse al final de la ocurrencia. Esto significa que el área que es necesario revisar será $T_{j-(i_1-1)-k \ldots j+(m-i_2)+k}$, de largo $m + 2k$.

2.2. Estructuras

2.2.1. Trie

Un trie corresponde a una estructura diseñada para almacenar strings donde hay un nodo para cada prefijo común.

Por ejemplo, el trie que representa a los strings *"alabar"*,*"a"*, *"la"*, *"alabarda"*, *"para"*, *"apalabrarla"* se puede apreciar en la figura 2.3.

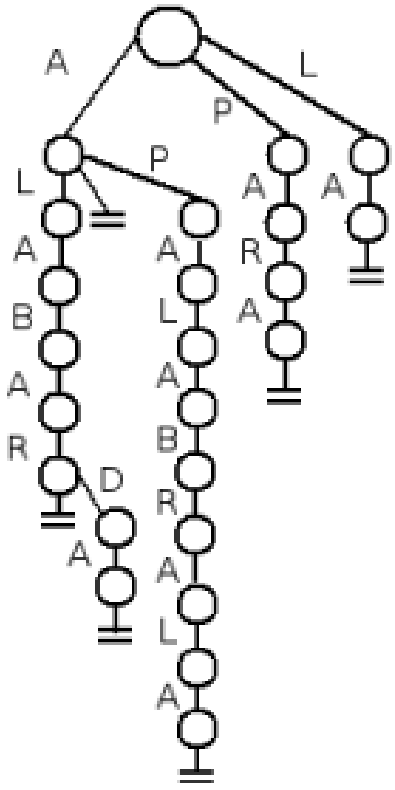

Figura 2.3: Trie para las palabras *"alabar"*,*"a"*, *"la"*, *"alabarda"*, *"para"*, *"apalabrarla"*.

Para su construcción se inserta cada string a partir de la raíz, produciendo un nuevo nodo en aquellas ocasiones donde el camino no existía anteriormente. Es interesante notar que se puede ver si un string de largo m se encuentra o no en el trie en tiempo $O(m)$.

Si bien esta estructura no es un índice, sirve para la implementación de los índices de texto completo más populares

2.2.2. Trie de sufijos

Como su nombre lo hace suponer, es un trie donde se insertan todos los sufijos del texto que se desea indexar, representados por la posición de éste en el texto

original. Un ejemplo de esto se puede apreciar en la figura 2.4. Sólo utilizaremos una parte del texto habitual de los ejemplos para poder obtener una representación de un tamaño adecuado para desplegar.

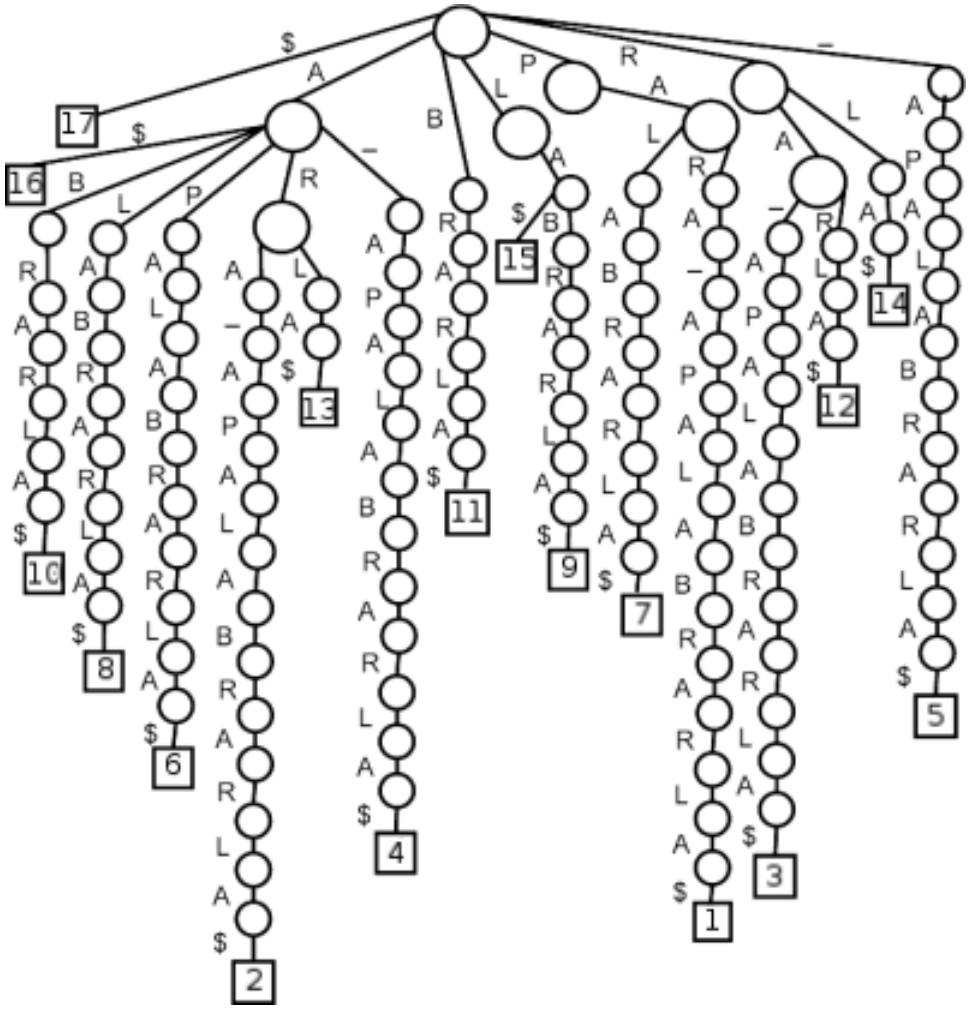

Figura 2.4: Trie de sufijos para el texto “para apalabrarla”.

Se concatena un carácter terminador especial “$” al texto, menor lexicográficamente que cualquier otro y que no pertenece al alfabeto. El propósito de esto es que ningún sufijo sea prefijo de otro.

El resultado de esto es un índice de texto completo que permite buscar todas las ocurrencias de cualquier substring de largo m en tiempo $O(m)$. Esto es otorgado por el trie.

La búsqueda se fundamenta en que todo substring es prefijo de algún sufijo. Así que para buscar un string en el texto simplemente se comienza desde la raíz y se va bajando por el hijo correspondiente a la letra que se leyó en el patrón. Si no hay un hijo por donde bajar, entonces ese patrón no está presente en el texto. Al llegar al

nodo que representa la última letra del patrón se reportan todas las posiciones del subárbol.

Si bien la búsqueda en esta estructura es sumamente eficiente, el problema que presenta es que es de tamaño $O(n^2)$, con lo que rápidamente se torna inmanejable.

2.2.3. Árbol de sufijos

Un árbol de sufijos es una estructura muy similar a un trie de sufijos. La diferencia radica en que en esta estructura los nodos con sólo un hijo son unidos con ese hijo. Esto se conoce como compresión de caminos unarios. En la figura 2.5 podemos apreciar el árbol de sufijos del mismo ejemplo anterior.

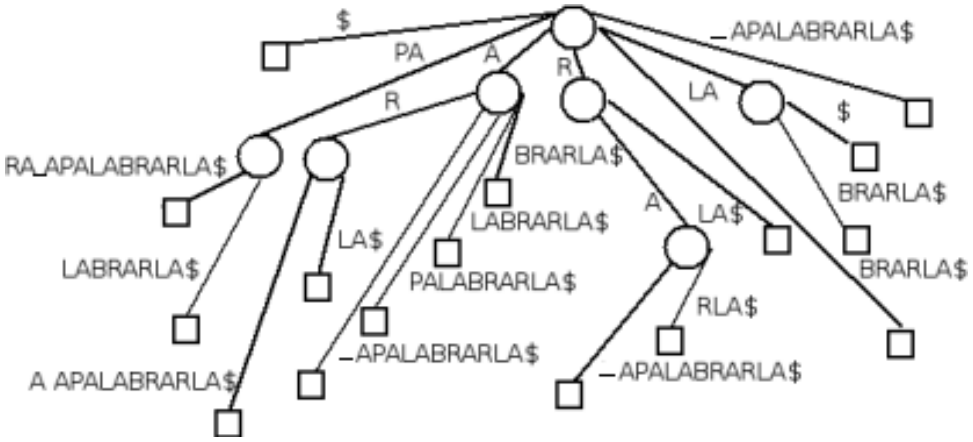

Figura 2.5: Árbol de sufijos de "apalabrarla".

La búsqueda se realiza de la misma manera que en un trie de sufijos, obteniéndose la misma complejidad temporal, pero con la ventaja de que esta estructura puede ser construida en tiempo lineal y además ocupa $O(n)$ espacio en el peor caso, pues es un árbol con n hojas donde los nodos tienen aridad al menos 2.

Aún así, la constante asociada al espacio es muy grande, del orden de veinte veces el tamaño del texto original. Además resulta costoso de construir y y manejar en memoria secundaria.

Junto con búsqueda de strings, esta estructura se presta para otras tareas, como son:

- Búsqueda del string más largo repetido.
- Búsqueda del substring más largo común a dos textos.
- Búsqueda de palíndromes.

2.2.4. Arreglo de sufijos

Desarrollado a partir de los esfuerzos por reducir el uso de memoria de las estructuras antes mencionadas, el arreglo de sufijos corresponde a un arreglo de todas las posiciones iniciales de los sufijos de un string en orden lexicográfico, como se ve en la figura 2.6. Es como si se construyera un árbol de sufijos, pero sólo se tomaran las hojas. La estructura continúa siendo $O(n)$ en el espacio, pero ahora la constante es alrededor de 4.

17	16	10	8	6	2	13	4	11	15	9	7	1	3	12	14	5

Figura 2.6: Arreglo de sufijos para el texto "apalabrarla".

Al estar ordenadas, las búsquedas tradicionales de strings se convierten en una búsqueda binaria dentro del arreglo. Los subárboles del árbol de sufijos se transforman en segmentos del arreglo, por lo que son necesarias dos búsquedas para definir los límites del intervalo de posiciones en el texto donde se encuentra el string buscado. Esta búsqueda tiene una penalización de $O(\log n)$ en tiempo respecto al árbol de sufijos.

Al igual que el árbol de sufijos, el arreglo de sufijos requiere el texto disponible para comparar contra el patrón y para reportar las ocurrencias, puesto que estas estructuras no reemplazan el texto que indexan.

2.3. Compresión

Por compresión entenderemos el proceso mediante el cual un conjunto de datos son codificados de manera de poder ser representados en menos espacio.

Existen dos tipos de compresión:

Sin pérdida: La información sólo es codificada, pudiéndose siempre recuperar íntegramente los datos originales.

Con pérdida: Parte de la información es descartada en base a algún criterio y lo restante es codificado, haciendo imposible la recuperación completa de los datos originales en su forma exacta, sólo una aproximación de ellos.

Ejemplos de compresión con pérdida son la codificación *MP3* de audio, donde frecuencias que son definidas como "inaudibles" no se representan, o en formatos de

imágenes como *JPEG*, que descartan aspectos poco visibles al ojo humano. Esta información descartable es modelada matemáticamente de manera de poder definir cuánta información descartar, y con esto, qué razón de compresión alcanzar.

En el caso de la compresión sin pérdida encontramos fundamentalmente todos los formatos de respaldo de archivos, como son *ZIP*, *GZIP*, y *RAR*. Este método es el usado para comprimir datos com otexto, puesto que nos interesa recuperar exactamente la información original.

Al hablar de compresión es importante saber cuánto es posible comprimir una entrada, de manera de poder comparar y saber cuán bueno es un compresor. Para este propósito estudiaremos la *entropía* de un texto en la siguiente sección.

2.3.1. Entropía empírica de un texto

A grandes rasgos, la entropía corresponde a la media del desorden en algún sistema. Aplicado a teoría de la información, se refiere a cuánto azar hay en una fuente, como un texto o un archivo binario, o lo que es lo mismo, cuánta información es necesaria para poder recrearla.

Sea s un string de largo n sobre un alfabeto $\Sigma = \{\alpha_1, \ldots, \alpha_h\}$, y sea n_i el número de ocurrencias del símbolo α_i en s. Se define la *entropía empírica de orden 0* [2] de un string:

$$H_0(s) = -\sum_{i=1}^{h} \frac{n_i}{n} \log_2(\frac{n_i}{n})$$

donde $|s|H_0(s)$ representa la salida de un compresor ideal que usa $-\log_2 \frac{n_i}{n}$ bits para codificar el símbolo α_i. Esto corresponde a la máxima compresión que se puede alcanzar en esquemas sin pérdida usando una codificación unívocamente decodificable, donde se asigna un código fijo a cada uno de los símbolos del alfabeto. Se puede lograr una mayor compresión si el código asignado a cada símbolo depende de los k símbolos que lo preceden. Para cualquier palabra de largo $w \in \Sigma^k$, sea w_s el string que contiene los caracteres a continuación de w en s. Entonces llamamos *entropía empírica de orden k* [2] del string s al valor:

$$H_k(s) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_s| H_0(w_s)$$

Si llamamos a w "contexto", entonces podemos interpretar esta fórmula como la dependencia de los símbolos de su contexto. El valor $|s|H_k(s)$ corresponde a una cota inferior a la compresión que se puede alcanzar con un compresor que utilice un

contexto de tamaño k para codificar los símbolos de s.

La entropía sirve para poder tener una noción de cuán bueno es un esquema de compresión, y al diseñar estructuras sucintas saber cuán cerca uno está de la representación óptima posible.

2.3.2. Compresión estadística

También conocida como *codificación de entropía*, se basa en relacionar los largos de los códigos a utilizar con la probabilidad de aparición de los símbolos. Usualmente los codificadores de entropía se utilizan para comprimir datos reemplazando símbolos representados por códigos de largo fijo con símbolos representados por códigos proporcionales al logaritmo negativo de la probabilidad. De esta manera, los símbolos más frecuentes adquieran los códigos más cortos.

Los ejemplos más comunes de esta técnica son los códigos de Huffman [7], *range encoding* [9] y codificación aritmética [1].

2.3.3. Compresión con diccionarios

La idea tras esta técnica es utilizar un diccionario que indique de qué manera codificar los datos para obtener la compresión.

Para que este método funcione es fundamental que el codificador y el decodificador posean el mismo diccionario. Para esto existen dos opciones:

- *El diccionario es fijo y lo poseen los programas.* En este caso el diccionario se encuentra definido de antemano y no cambia ante diferentes entradas al compresor. Esto puede resultar en una mala compresión al no ser apropiado para los datos.

- *El diccionario es dinámico y se agrega a los datos.* En este caso se construye durante el proceso de compresión y se adapta a las diversas entradas, pero debe diseñarse adecuadamente su representación para no ocupar demasiado espacio, pudiendo generar tasas negativas de compresión.

La selección de uno u otro método dependerá de las restricciones del problema que se desee solucionar y una cuidadosa evaluación de las ventajas/desventajas de cada uno. Ejemplos de esta técnica son *Ziv-Lempel* 77 [12] y 78 [13], y *Byte-Pair Enconding* [4].

Como Ziv-Lempel es fundamental para el desarrollo de esta memoria, a continuación se revisará en mayor detalle este algoritmo.

2.3.4. Compresión Ziv-Lempel

El esquema de compresión Ziv-Lempel (LZ77 [12], LZ78 [13]) consiste en reemplazar substrings del texto por un puntero a una ocurrencia previa de éstas. La compresión se obtiene cuando el puntero ocupa menos espacio que el string reemplazado.

El algoritmo LZ78 en particular está basado en un diccionario de bloques de compresión. Cada nuevo bloque se forma buscando el prefijo más largo del texto aún sin comprimir que coincide con algún bloque del diccionario, y se agrega el siguiente carácter que hace único al nuevo bloque.

El formato de la salida comprimida corresponde a una secuencia de bloques $T = B_0, \ldots, B_n$ representados por pares *(a,b)*, donde a es el identificador del bloque referenciado y b es la letra a continuación de este bloque. Una importante propiedad de este esquema es que cada bloque es único, es decir no hay dos bloques representando el mismo substring. Esto se logra agregando al final del texto un carácter que no está en el alfabeto, "$", y que es menor lexicográficamente que todos los demás caracteres.

En la figura 2.7 se puede observar el resultado de comprimir la frase "*alabar a la alabarda para apalabrarla*" mediante LZ78.

$$(0,a)(0,l)(1,b)(0,\_)(1,\_)(2,a)(5,a)(7,b)(4,d)(6,p)(4,a)(8,p)(1,l)(3,r)(4,l)(1,\$)$$

Figura 2.7: Ejemplo compresión LZ78.

Este esquema de compresión converge a la entropía de orden k de la fuente, para todo k, aunque la convergencia es lenta.

Capítulo 3

Trabajo Relacionado

A continuación revisaremos los desarrollos más importantes que están relacionados de forma directa con el ámbito de los índices.

Uno de los primeros avances importantes en el ámbito de la indexación son las estructuras sucintas. Éstas corresponden a estructuras diseñadas teniendo como principal consideración el uso de poco espacio. Esto resulta fundamental en el ámbito de los índices de texto completo. Un ejemplo de estos índices es el arreglo de sufijos comprimido de Grossi y Vitter [5].

El segundo desarrollo necesario para los actuales índices es la investigación sobre autoíndices. Hasta hace un tiempo, los índices eran estructuras anexas al texto mismo. De esta manera, cualquier esquema de indexación utilizaba al menos el espacio del texto, pues éste no se podía descartar. Para reportar lo calzado, o un contexto del calce, y a veces incluso para buscar, se hacía necesario visitar el archivo del texto.

La idea de los autoíndices es que el texto completo se puede recuperar a partir del índice. De esta forma se pueden obtener reducciones en el espacio utilizado que de otra manera serían imposibles. Una de las propiedades que se utilizan para este fin es el *oportunismo*, es decir, aprovechar que los datos de entrada son compresibles para poder reducir el espacio usado sin causar un impacto significativo en la velocidad de las consultas.

A continuación revisaremos los autoíndices más relevantes de la literatura.

3.1. FM-index

El FM-index [2] [3] corresponde a un índice comprimido que combina la transformada de Burrows-Wheeler (BWT) con el arreglo de sufijos. A continuación revisaremos la BWT y luego las estructuras del FM-index.

La BWT es una permutación reversible del texto. La idea es reordenar los caracteres de manera que el string resultante sea más fácil de comprimir con algoritmos de compresión de datos localmente adaptativos como *move-to-front coding* y un codificador estadístico.

Sea $T = t_1 t_2 ... t_n$ un texto sobre el alfabeto Σ. La transformación entonces consiste en los siguientes pasos:

1. Agregar un carácter especial # de orden lexicográfico menor que todo carácter del alfabeto.

2. Generar una matriz conceptual M, donde las filas son rotaciones cíclicas del string $T\#$. La matriz es de $(n+1) \times (n+1)$.

3. Ordenadas las filas de M lexicográficamente.

4. Obtener la transformada L del texto, que corresponde a la última columna de M.

	F	L
alabarda#	#	a
labarda#a	a	d
abarda#al	a	l
barda#ala	a	#
arda#alab	a	b
rda#alaba	b	a
da#alabar	d	r
a#alabard	l	a
#alabarda	r	a

Figura 3.1: Cálculo de la BWT.

El resultado se puede observar en la figura 3.1. Podemos notar que, por construcción, todas las columnas de M son permutaciones de $T\#$, en particular

L y la primera columna F de la matriz, que es igual a los caracteres de L en orden lexicográfico.

Existe una relación entre la matriz M y el arreglo de sufijos A del texto $T\#$. Al agregar el carácter # garantizamos que ningún sufijo es un prefijo de otro sufijo. Entonces, al ordenar las filas de M, esencialmente estamos ordenando los sufijos de $T\#$. Es así que $A[i]$ apunta al sufijo de $T\#$ que es un prefijo de la i-ésima fila de M.

Otra de las propiedades de la BWT es que es reversible. Para esto definimos lo siguiente:

- Para $c \in \Sigma$, $C[c]$ corresponde al número total de ocurrencias en T de los caracteres de orden lexicográfico menor que c.
- Para $c \in \Sigma$, $Occ(c, k)$ corresponde al número de ocurrencias de c en el prefijo $[1, k]$.

Y las siguientes propiedades de M:

a. Dada la i-ésima columna de M, su último carácter $L[i]$ precede a su primer carácter $F[i]$ en el texto T. Es decir, $T = \ldots L[i]F[i] \ldots$.

b. Sea $LF(i) = C[L(i)] + Occ(L[i], i)$. El carácter en F que corresponde a $L[i]$ está ubicado en la posición $LF(i)$. LF corresponde a *Last-to-First mapping* (mapeo de último a primero, con respecto a las columnas).

Entonces si $T[k]$ corresponde al i-ésimo carácter de L, $T[k-1] = L[LF(i)]$. De esta manera se puede obtener todo el texto hacia atrás.

El FM-index se compone de una representación comprimida de L, junto con estructuras auxiliares que permiten computar en tiempo constante $O(1)$ el valor de $Occ(c, k)$. Soporta dos operaciones de búsqueda: *count* y *locate*.

Count toma un patrón $P = p_1..p_m$ y retorna el número de ocurrencias de P en T. Para esto utiliza algunas propiedades estructurales de M:

- Todos los sufijos del texto T que tienen como prefijo al patrón P ocupan un rango de filas de M.
- Este rango de filas empieza en la posición sp y termina en ep, donde sp es la posición lexicográfica del string P entre las filas ordenadas de M.

Las posiciones de sp y ep son determinadas en m etapas en las que se mantiene el invariante de que, en la etapa i, sp apunta a la primera fila de M de prefijo $P[i,m]$ y ep apunta a la última fila de prefijo $P[i,m]$. Al finalizar la etapa m, y si $sp \leq ep$ entonces $(ep-sp+1)$ representa el número de ocurrencias de P en T. Esto es similar a realizar el conteo sobre el arreglo de sufijos.

El procedimiento *locate* toma un índice i de una fila de la matriz M y retorna la posición de inicio en T del sufijo correspondiente a $M[i]$, que llamaremos $pos(i)$. Si se desea computar las posiciones de todas las ocurrencias de P en T entonces basta llamar a $locate(i)$ con $i = sp...ep$, con sp y ep computados por *count*.

Para el cómputo de *locate* lo que se hace es realizar un muestreo de un conjunto apropiado de filas de M y para éstas se conserva un mapeo explícito de sus posiciones en T. De esta forma, si i es una fila mapeada, entonces $pos(i)$ está directamente disponible. De no ser así, se usa el mapeo LF y el método para retroceder en el texto y encontrar la fila i' correspondiente al sufijo $T[pos(i)-1,n]$. Este proceso se repite v veces hasta encontrar una fila i'' que sí esté marcada. Entonces $pos(i) = pos(i'') + v$.

En este mismo proceso se obtienen los caracteres $T[pos(i''), pos(i')]$, procedimiento que se puede generalizar para recuperar cualquier substring del texto.

3.2. Suffix Array Comprimido (CSA)

Revisaremos el CSA de Sadakane [11] que se basa en el de Grossi y Vitter [5]. Este último ocupa sólo $O(n)$ bits para un texto de largo n, pero sin embargo requiere el texto disponible. La estructura de Sadakane es un autoíndice.

En el CSA el arreglo de sufijos $A[1,n]$ se representa mediante una secuencia de números $\psi(i)$, tal que $A[\psi(i)] = A[i] + 1$. Esta función es monótonamente creciente por partes, es decir, ψ se incrementa en las zonas de A donde el sufijo comienza con el mismo carácter c. Esto permite una representación sucinta de la secuencia.

Básicamente codificamos la secuencia de manera diferencial, es decir, codificamos $\psi(i) - \psi(i-1)$. Si hay una repetición dentro del arreglo, por ejemplo $A[j \ldots j+l] = A[i \ldots i+l] + 1$, entonces $\psi(i \ldots i+l) = j \ldots j+l$ y por lo tanto $\psi(i) - \psi(i-1) = 1$ en toda esa área. Utilizamos un método de codificación que favorezca los números pequeños y que permita un tiempo constante de acceso a la función ψ.

Para conseguir que sea un autoíndice se utiliza el mismo arreglo $C[1 \ldots \sigma]$ que vimos en 3.1. Con esto es posible descartar el texto.

La búsqueda se realiza simulando una búsqueda binaria en el arreglo de sufijos.

Esto se hace extrayendo de la estructura strings de la forma $t_{A[i]}t_{A[i]+1}t_{A[i]+2}\ldots$ para cualquier i que se requiera por la búsqueda binaria. El primer carácter $t_{A[i]}$ es fácil de obtener ya que todos los primeros caracteres están ordenados en el arreglo de sufijos. Este carácter c es tal que $C[c] < i \leq C[c+1]$, que se puede encontrar en tiempo constante utilizando pequeñas estructuras adicionales. Una vez que tenemos el primer carácter nos movemos a $i' = \psi(i)$ y continuamos con $t_{A[i']} = t_{A[i]+1}$.

Para reportar las ocurrencias se toman muestras del texto con algún intervalo conveniente. Luego se puede obtener los caracteres no muestreados mediante la función ψ.

3.3. LZ-index

El LZ-index [10] es un índice basado en los algoritmos de compresión de la familia Ziv-Lempel (LZ), en particular en este caso el LZ78 [13], revisado en mayor detalle en la sección 2.3.4. A continuación revisaremos la estructura del índice y el procedimiento de búsqueda en él.

3.3.1. Estructuras de datos

El índice en sí mismo está compuesto por cuatro estructuras:

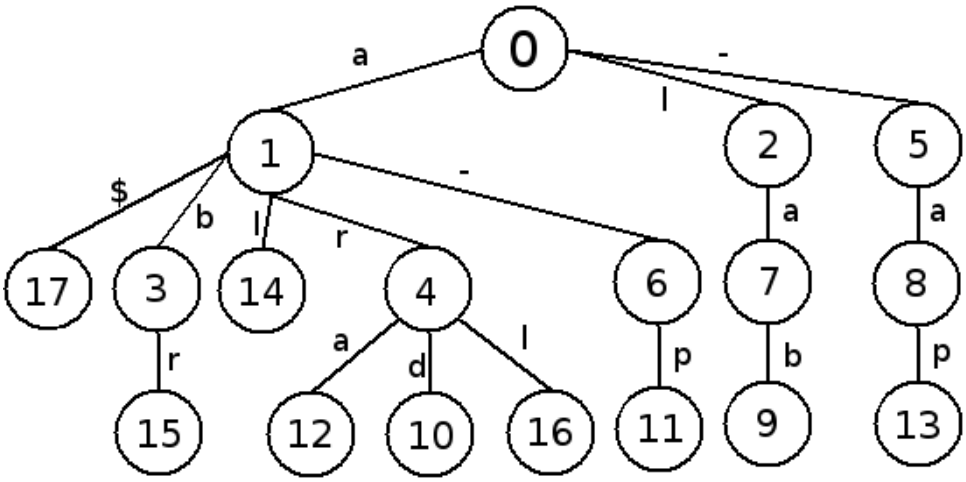

Figura 3.2: El *LZTrie*: trie de los bloques de compresión LZ para el texto *"Alabar a la alabarda para apalabrarla"*.

1. *LZTrie*: Es el trie formado por los bloques de compresión del algoritmo LZ. Por las propiedades del LZ78, este árbol posee exactamente $n+1$ nodos, cada uno

correspondiente a un string distinto. En la figura 3.2 podemos ver el *LZTrie* de los bloques de compresión presentados en la figura 2.7

2. *RevTrie*: Es el trie formado por los strings reversos de los contenidos en los bloques de compresión ($B_0^r \dots B_n^r$). A diferencia del *LZTrie*, puede tener nodos internos que no correspondan a bloques.

3. *Node*: es un mapeo de los identificadores de bloque a sus nodos correspondientes en el *LZTrie*.

4. *Range*: es una estructura para búsqueda bi-dimensional en el espacio $[0 \dots n] \times [0 \dots n]$. Los puntos almacenados corresponden a $\{(revrank(B_k^r), rank(B_{k+1})), k \in 0 \dots n-1\}$, siendo *revrank* la posición lexicográfica en $B_0^r \dots B_n^r$, y $rank$ la posición lexicográfica en $B_0 \dots B_n$. Para cada uno de estos puntos se almacena el valor de k.

Junto con esto, el *LZTrie* y *RevTrie* deben permitir aplicar las siguientes operaciones en cada nodo x:

(a) $id(x)$ entrega el identificador del nodo x, que corresponde al número k tal que x representa a B_k.

(b) $leftrank(x)$ y $rightrank(x)$ que entregan la mínima y máxima posición lexicográfica de los bloques representados por los nodos del subárbol de raíz x, ambas dentro del conjunto $B_0 \dots B_n$.

(c) $parent(x)$ entrega la posición en el trie del padre del nodo x.

(d) $child(x, c)$ entrega la posición en el trie del hijo del nodo x a través del carácter c, o *null* si no existe tal hijo.

Cuando las operaciones sean aplicadas en *RevTrie* las llamaremos $id_r(x)$, $leftrank_r(x)$, $rightrank_r(x)$, $parent_r(x)$ y $child_r(x, c)$, respectivamente.

Junto con esto el trie debe permitir implementar la operación $rth(r)$, la cual dado un orden r retorna el id del nodo que representa el r-ésimo string en $B_0 \dots B_n$, en orden lexicográfico. En la figura 3.3 podemos ver el *LZTrie* y cómo se relacionan las operaciones con él.

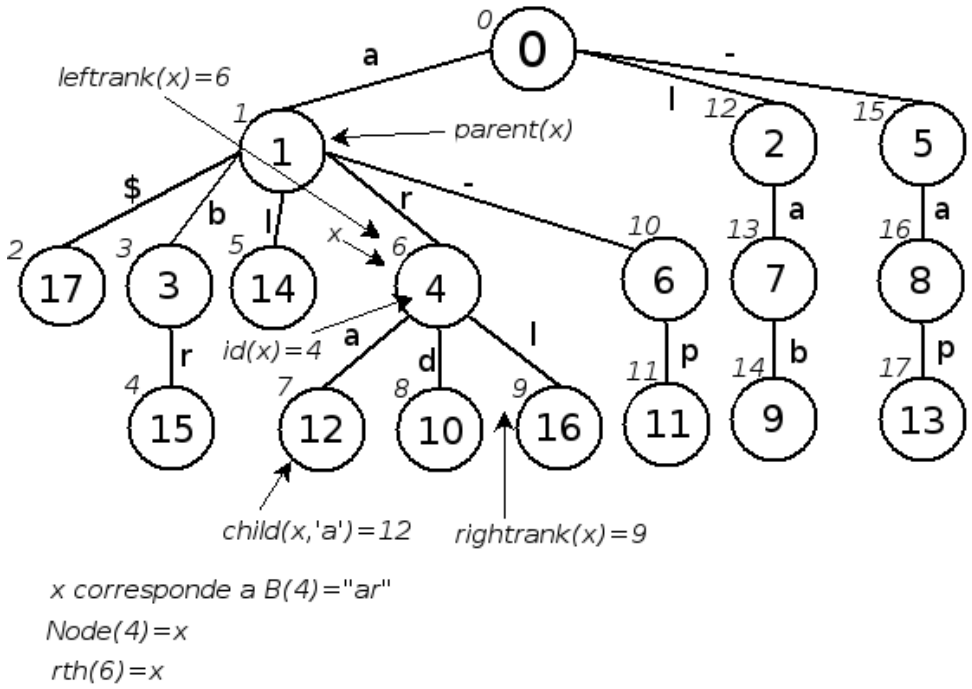

Figura 3.3: LZTrie y sus operaciones.

3.3.2. Búsqueda

Considerando los bloques de compresión que se obtienen, una ocurrencia del patrón dentro del texto puede ocurrir de 3 maneras:

(a) Tipo 1: La ocurrencia se produce completamente dentro de un bloque.

Debido a las propiedades de LZ, cada bloque B_k que contiene el patrón está conformado por un bloque más corto B_t concatenado con una letra final. Si el patrón no ocurre al final del bloque B_k, entonces B_t contiene el patrón también. Para encontrar el bloque más corto que contiene la referencia al patrón, simplemente buscamos el patrón reverso P^r en *RevTrie*. Luego, para encontrar todas las ocurrencias que están en un único bloque:

1. Se busca P^r en *RevTrie*. El nodo x al que se llega es tal que todo nodo en el subárbol de raíz x representa un bloque que termina en P.
2. Se evalúa $leftrank_r(x)$ y $rightrank_r(x)$, obteniéndose el intervalo lexicográfico en *RevTrie* de bloques que terminan en P.
3. Para cada $r \in leftrank_r(x) \ldots rightrank_r(x)$, se obtiene el nodo correspondiente en el *LZTrie* mediante la operación $y = Node(rth_r(r))$. Con

los nodos que terminan en P identificados en el trie normal, lo que resta es reportar los subárboles de estos nodos, puesto que todos ellos contienen a P.

4. Para cada nodo y, recorrer todo el subárbol de raíz y y reportar todos los nodos encontrados.

(b) Tipo 2: La ocurrencia abarca dos bloques.

En este caso lo que se hace es dividir el patrón en dos partes de las $m-1$ formas posibles. Para cada partición se busca el prefijo reverso en *RevTrie* y el sufijo en *LZTrie*. De esta manera se obtienen todos los bloques que terminan con el comienzo del patrón y todos los que empiezan con el final de éste. Se necesita encontrar los pares de bloques $(k, k+1)$ tal que k está en el primer grupo y $k+1$ en el segundo. Para esto se utiliza la estructura *Range* antes mencionada.

Los pasos del algoritmo son los siguientes:

1. Para cada $i \in 1 \ldots m-1$, dividir P en $pref = P_{1\ldots i}$ y $suff = P_{i+1\ldots m}$.
2. Para cada una de estas particiones, se busca $pref^r$ en *RevTrie*, obteniéndose el nodo a. A la vez, se busca $suff$ en el *LZTrie* obteniéndose el nodo b.
3. Utilizando la estructura *Range* se busca el rango $[leftrank_r(a) \ldots rightrank_r(a)] \times [leftrank(b) \ldots rightrank(b)]$
4. Para cada par $(k, k+1)$, se reporta k.

(c) Tipo 3: La ocurrencia abarca tres o más bloques.

Este es el caso más complejo. Lo importante es recordar que cada bloque contiene un string distinto. Entonces, para cualquier substring $P_{i\ldots j}$ del patrón existe a lo sumo un bloque que calce con él.

La idea es para cada substring $P_{i\ldots j}$ encontrar el bloque donde este calza. El identificador del bloque se almacena en m arreglos A_i, donde A_i contiene los bloques correspondientes a $P_{i\ldots j}$ para todo j. Luego se buscan concatenaciones de bloques consecutivos B_k, B_{k+1} que calcen substrings contiguos del patrón. Finalmente para cada concatenación maximal de bloques $P_{i\ldots j} = B_k \ldots B_t$ se revisa que B_{k-1} termine con $P_{1\ldots i-1}$ y que B_{t+1} comience con $P_{j+1\ldots m}$. De ser así, se reporta una ocurrencia.

El algoritmo es:

1. Para cada $1 \leq i \leq j \leq m$ se busca $P_{i...k}$ en el *LZTrie*, guardándose el nodo encontrado en $C_{i,j}$ y agregando $(id(x), j)$ al arreglo A_i. La búsqueda se realiza incrementando i, y para cada i se incrementa j. De esta manera sólo se realiza una búsqueda en el trie para cada i. Si no existe un nodo correspondiente a $P_{i...j}$, se deja de buscar y se almacenan valores nulos en $C_{i,j'}$, para $j' \geq j$. Al terminar cada ciclo de i se ordena el arreglo A_i por número de bloque. Se marca todo $C_{i,j}$ como *no usado*.

2. Para cada $1 \leq i \leq j \leq m$, al incrementar j se trata de incrementar hacia la derecha el calce de $P_{i...j}$. Sean S y S_0 iguales a $id(C_{i,j})$, y se busca $(S+1, r)$ en A_{j+1}. Si r existe, se marca $C_{j+1,r}$ como *usado*, se incrementa S y se repite el proceso a partir de $j = r$. Se detiene el proceso cuando la ocurrencia no puede ser extendida, es decir, no existe r.

   a. Para cada ocurrencia maximal $P_{i...r}$ encontrada que termine en el bloque S tal que $r < m$, verificar si el bloque $S+1$ comienza con $P_{r+1...m}$, que es lo mismo que ver si $leftrank(Node(S+1)) \in leftrank(C_{r+1,m}) \ldots rightrank(C_{r+1,m})$. Hay que notar que $leftrank(Node(S+1))$ es el rank exacto del nodo $S+1$, puesto que cada nodo interno es el primero entre los *ranks* de su subárbol. También hay que notar que no puede haber una ocurrencia si $C_{r+1,m}$ es nulo. Si $r < m$ y el bloque $S+1$ no comienza con la continuación correspondiente del patrón, entonces hay que detenerse y continuar con la siguiente ocurrencia maximal.

   b. Si $i > 1$, entonces verificar si el bloque $S_0 - 1$ termina con $P_{1...i-1}$. Para esto, se encuentra $Node(S_0 - 1)$ y se utiliza la operación *parent* para verificar si los últimos $i-1$ nodos, leídos hacia atrás, son iguales a $P^r_{1...i-1}$. Si $i > 1$, y el bloque $S_0 - 1$ no termina con $P_{1...i-1}$, entonces nos detenemos y pasamos a la siguiente ocurrencia maximal.

   c. Reportar el nodo $S_o - 1$ como el inicio del calce. Sabemos que $P_{1...i}$ se encuentra al final de este bloque.

Es importante notar que se debe verificar que la ocurrencia se expanda por al menos 3 bloques. De no ser así, se rechaza.

3.3.3. Notas sobre la implementación

Para la comprensión del trabajo desarrollado resulta mucho más simple considerar que el índice provee funciones de búsqueda para cada tipo de ocurrencia. Estas funciones realizan los pasos ya mencionados y cuando determinan un calce retornan la posición de término de éste. Esta posición es representada mediante el identificador del bloque y un *offset* dentro de éste, es decir cuántos caracteres antes del final del bloque termina efectivamente el calce.

Para este trabajo también resulta muy importante la manera en que se obtiene el texto a partir de un calce, por lo que se detallará el procedimiento a continuación.

Con el identificador de bloque de un calce b vamos al nodo correspondiente a éste en el *LZTrie* mediante $x = Node(b)$. En cada uno de los nodos se obtiene el carácter por el cual se sube al padre. El texto del bloque se obtiene en sentido inverso, subiendo por el trie hasta la raíz utilizando la función *parent* en cada uno de los nodos.

Si se desea mostrar una ocurrencia que comienza en el bloque k, basta con ir al *LZTrie* usando $Node(k)$, y luego usando *parent* obtener los caracteres del bloque en orden reverso. Si la ocurrencia abarca más de un bloque, se hace lo mismo con los bloques $k+1$, $k+2$ y así sucesivamente, hasta que se muestra el patrón completo. Se puede hacer lo mismo con $k-1$, $k-2$, ..., para mostrar un contexto más grande. Como se ve, usando este procedimiento se puede recuperar todo el texto para $k \in 0 \ldots n$.

Es importante mencionar que dado que cada bloque es único, cada nodo en el *LZTrie* representa sólo a un bloque, por lo que es casi equivalente referirse a nodos o bloques. Hay que recordar que cuando se habla del nodo x, en el fondo también se está hablando del bloque $B_{id(x)}$.

Capítulo 4

Desarrollo

4.1. Concepto general

Tras la revisión del la estructura del LZ-index y las facilidades que provee es necesario analizar de qué manera se puede implementar la búsqueda aproximada sin realizar modificaciones a la estructura.

Como se discutió en 2.1.1, uno de los algoritmos para búsqueda aproximada consiste en dividir el patrón de manera que alguna de sus partes aparezca *sin errores* en el texto. Luego es necesario verificar si el calce de esa parte corresponde o no a una ocurrencia del patrón completo.

El índice permite recuperar de manera rápida los calces exactos. Además, al ser un autoíndice es posible obtener el texto del calce y un contexto de éstos directamente desde el índice. Cabe notar que en particular el LZ-index está diseñado para permitir que la recuperación del texto sea relativamente rápida. Esto hace esperable que la combinación del algoritmo de particionamiento y el LZ-index sé buenos resultados.

Se comenzará con una implementación básica. Luego se buscará mejorar los tiempos de búsqueda se obtendrán a partir de un recorrido más eficiente de las estructuras y de un mejor uso de la información obtenida. A su vez se revisarán los beneficios que se puedan obtener de un particionamiento más inteligente del patrón a buscar.

4.2. Implementación básica

Esta implementación busca agregar la funcionalidad de búsqueda aproximada de la manera más simple posible, sin hacer optimizaciones en el procedimiento

para obtener mejores tiempos. Servirá como piso de comparación con las siguientes implementaciones. Como ya se discutió, el problema se reduce a una búsqueda y una verificación. Esta última se realizará con el método de *programación dinámica* visto en la sección 2.1.1.

El primer paso en el procedimiento es inicializar la columna de programación dinámica en base al patrón P.

A continuación, el patrón $P = p_1 p_2 \dots p_m$ de largo m se divide en $k+1$ partes. Cada parte se denominará O_i de largo o_i cada una, con $i = 1, \dots, k+1$. Los largos de las partes son iguales entre sí, y de no ser posible, la diferencia es de a lo más un carácter. Es decir, si el patrón es "*ladraba*" y se está buscando con $k = 2$ errores, las partes serían *lad*, *ra* y *ba*.

Cada una de estas partes divide al patrón P en tres secciones lógicas $P = A_i \cdot O_i \cdot D_i$, donde A_i y D_i corresponden a las partes que están antes y después de O_i en el patrón, respectivamente. Para esta implementación nos interesa el largo a_i y d_i de cada una de estas secciones, que se definen como:

$$a_i = \begin{cases} 0 & \text{si} \quad i = 1, \\ \sum_{j=1}^{i-1} o_j & \text{si} \quad i \neq 1 \end{cases}$$

$$d_i = \begin{cases} m - \sum_{j=1}^{i} o_j & \text{si} \quad i \neq k+1, \\ 0 & \text{si} \quad i = k+1 \end{cases}$$

Para cada parte O_i se buscan sus calces de tipo 1, 2 y 3 en el índice. Si ésta fuera una búsqueda normal el paso siguiente sería reportar los calces. En este caso, en cambio, hay que verificar la ocurrencia del patrón original con los errores permitidos. Para esto es necesario leer un string del texto de tamaño $k + m + k$. Esto debido a que la ocurrencia con errores puede empezar a lo más k caracteres antes o terminar a lo sumo k caracteres después del string original. Es necesario resaltar que la posición reportada al encontrar una ocurrencia es siempre un final de bloque. De esta manera, si el calce queda dentro del bloque, la variable $offset$ indica la distancia entre esta posición y el final del bloque reportado. Luego, para cada nodo del índice se sigue el siguiente procedimiento:

- Se lee un string de $o_i + a_i + k + offset$ caracteres hacia atrás de la posición reportada. El texto del bloque actual se obtiene subiendo por el *LZTrie* obteniendo cada carácter por el puntero que se había bajado. Si es necesario

leer más caracteres de los contenidos en este bloque, se pasa al nodo que representa al bloque anterior y se repite la operación antes mencionada. Esto se hace para todos los bloques que sean necesarios hasta leer todo el string. Como se va leyendo carácter a carácter, cuando se ha leído la cantidad necesaria, se detiene el proceso.

- El string anterior es leído de manera inversa. Se almacena al revés para poder utilizarlo como un string común y corriente.

- Nos volvemos a ubicar al final del bloque original del calce.

- Si la cantidad de caracteres leídos es mayor que $2k + m$ se descartan los caracteres sobrantes del string. De otra forma, se se leen $d_i - offset + k$ caracteres hacia adelante. Al estar ubicados al final de un bloque, se va al nodo que representa al bloque siguiente. Esto, al igual que ir al nodo anterior, se hace utilizando el mapeo de bloques a nodos. Entonces, se lee el texto del bloque, invirtiéndolo inmediatamente y concatenándolo con el string resultante principal. Para cada bloque leído se chequea el largo de este string resultante. Si este valor es mayor o igual a $2k+m$ se detiene el proceso y se descartan los caracteres restantes. Si no, se repite el proceso.

- Se utiliza el string obtenido para ejecutar programación dinámica con respecto al patrón P y considerando los k errores. Los calces obtenidos a partir de la programación dinámica se reportan en el formato $bloque, offset$. Esta misma información es almacenada en una tabla de *hash* para evitar reportar calces repetidos.

Como ejemplo, veamos el proceso de búsqueda del patrón "*azabar*" con $k = 2$ errores.

El patrón se divide en 3 partes de dos caracteres cada una. La primera parte ("*az*"), no se encuentra en el texto, y por lo tanto no generará verificación alguna. La parte "*ab*" requiere de verificar los bloques 3 y 15, mientras que la parte "*ar*" los bloques 4, 10, 12 y 16. El detalle exacto del texto a verificar se puede observar en la figura 4.1.

Esta aproximación presenta algunas ineficiencias. Si vemos las distintas partes en el *LZTrie* de la figura 3.2, podremos notar que el string "*ar*" es común a todos los bloques que hay que revisar. Esto se ve en que los nodos corresponden al subárbol del nodo 4, que es una ocurrencia de tipo 1. Para cada nodo de ese subárbol lo

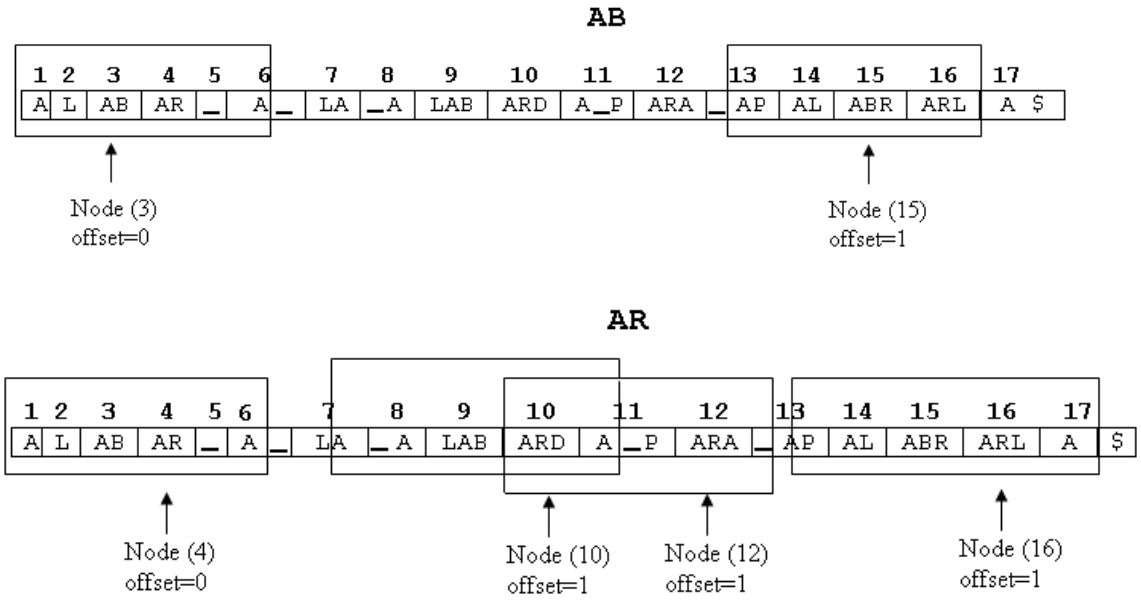

Figura 4.1: Bloques que se deben verificar y sus contextos.

que se está haciendo es ir al bloque correspondiente, obtener el texto de éste y revisarlo, recorriendo cuatro veces parte de la estructura en forma innecesaria. Esto se ve agravado por el cálculo de la programación dinámica, puesto que no sólo se leen repetidas veces strings comunes, sino que además todas las veces se vuelve a calcular el valor de la columna de programación dinámica para éstos. Sería muy interesante poder evitar este tipo de situaciones, puesto que el recorrido de la estructura y la obtención de los caracteres de los bloques son las operaciones más caras de computar.

4.3. Implementación factorizando verificaciones

Los problemas antes planteados se presentan particularmente en las ocurrencias de tipo 1, donde cada calce encontrado implica un subárbol que reportar, y por lo tanto revisar. Por este motivo centraremos nuestras optimizaciones en este tipo de calce. Nos referiremos a un *calce primario* como aquel que corresponde a la raíz del subárbol (es decir, el y del punto (a),3 de la sección 3.3.2, página 25), y como *calces secundarios* a todos los nodos dentro de este.

Al igual que en la sección anterior, cada una de las partes del patrón lo divide en tres secciones lógicas tales que $P = A_i O_i D_i$, de largos a_i, o_i y d_i cada una. Estas mismas secciones se ven definidas en el string que se lee para verificar en la forma $A'_i O_i D'_i$, de largos a'_i, o_i y d'_i, como vemos en la figura 4.2.

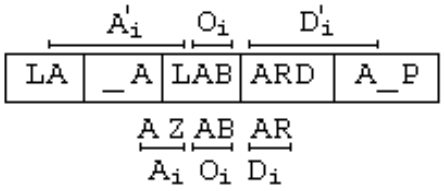

Figura 4.2: Secciones lógicas del patrón representadas en el string que se verifica, con $k = 2$ errores.

Esta vez no se verificará obteniendo A'_i y D'_i completos y luego buscando $A_iO_iD_i$ en $A'_iO_iD'_i$, sino que se calcularán distancias de edición entre A_i y sufijos de A'_i, y entre D_i y prefijos de D'_i. Cabe notar que debe reportarse una ocurrencia de $A_iO_iD_i$ en $A'_iO_iD'_i$, gatillada por O_i y terminada en $D'_i[j]$ si y sólo si:

$$\min_{j \geq 1} ed(A_i, A'_i[j \ldots a'_i]) + ed(D_i, D'_i[1 \ldots j]) \leq k \tag{4.1}$$

Llamaremos $sed(A_i, A'_i) = \min_{j \geq 1} ed(A_i, A'_i[j \ldots a'_i])$, y corresponde a la mínimo distancia de edición obtenida entre A_i y los sufijos de A'_i, lo que implica en el fondo el error k' con que se encontró esta parte.

Comencemos considernado el cálculo de D_i contra D'_i. Observemos la figura 4.3. En ésta un calce primario se representa mediante el nodo x. El string contenido en el bloque de compresión representado por x corresponde al camino desde este nodo hasta la raíz. Al ser un calce primario, sabemos que el string del bloque termina con O_i. Tras verificar una ocurrencia en este nodo, hay que revisar los calces secundarios del subárbol.

Podemos notar que, al bajar al nodo y a través del carácter c, en el fondo lo que estamos haciendo es obtener los primeros caracteres de D'_i. La diferencia con una búsqueda tradicional en un árbol es que en vez de sólo continuar bajando, en cada nodo es necesario ir a los bloques que se encuentran a continuación del representado por el nodo y para obtener el texto que constituye D'_i, descontando el primer carácter que sería c.

Veamos esto con un ejemplo. Supongamos que buscamos *arma* con $k = 1$ error. Esto significa dividir el patrón en dos partes: *"ar"* y *"ma"*. Las ocurrencias de tipo 1 de *"ar"* nos entregarán como calce primario el nodo de identificador 4 en la figura 3.2.

A continuación es necesario verificar si este nodo corresponde a una ocurrencia del patrón. Para esto hay que comparar los 3 caracteres que estén a continuación en el texto. Esto significa obtener el texto de los bloques 5 y 6.

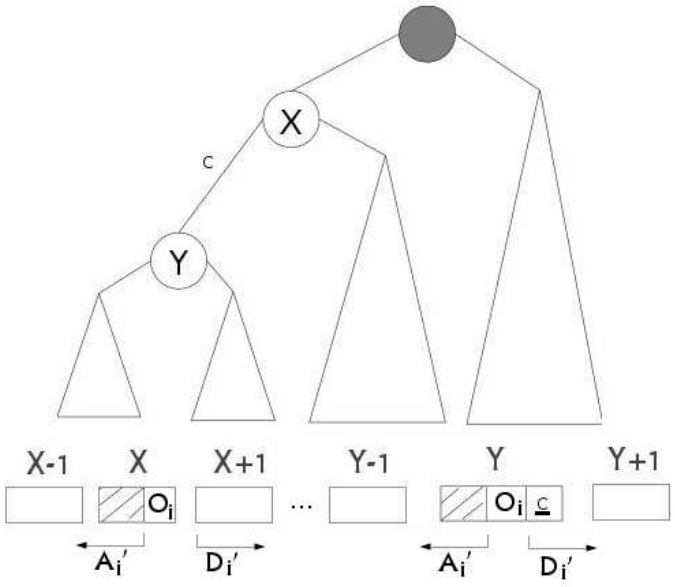

Figura 4.3: LZTrie, vista genérica.

Una vez realizado esto pasamos al primer calce secundario, que sería el nodo de identificador 12. Aquí también se deben revisar las ocurrencias del patrón. Esta vez, sin embargo, sólo necesitamos obtener dos caracteres a continuación del bloque actual, puesto que el primero de los tres originales está dado por el carácter a través del cual se bajó en el subárbol. Este procedimiento se repite para todos los calces secundarios del subárbol de raíz 4.

Tras comprender esto, resulta natural el definir el recorrido del árbol en una forma de *backtracking*, esquematizado en la figura 4.4. Recorremos en profundidad, conservando el estado de la columna de cálculo de distancia de edición para D_i y la profundidad relativa en el subárbol del calce primario, que llamaremos $prof$. Esta columna se almacena en cada nodo, se actualiza al bajar en el árbol con el carácter a través del cual se bajó, y se descarta al subir. El procedimiento corresponde conceptualmente a realizar la búsqueda mediante la matriz de distancia de edición. Al bajar se agregan columnas a la matriz y al subir se remueven de tal manera que al volver a bajar la columna se computa en base a la columna anterior. De esta forma se obtiene la ilusión de nunca haber recorrido el otro camino.

En cada nodo se utiliza una copia de la columna de distancia correspondiente al nodo para verificar D_i contra el texto D'_i de los bloques siguientes. Sin embargo, para poder definir una ocurrencia del patrón no basta sólo con el error de D_i versus D'_i, sino también el de A_i contra A'_i, como se detalla en la ecuación 4.1.

Una manera intuitiva de entender esta ecuación es ver que si A_i calzó con un

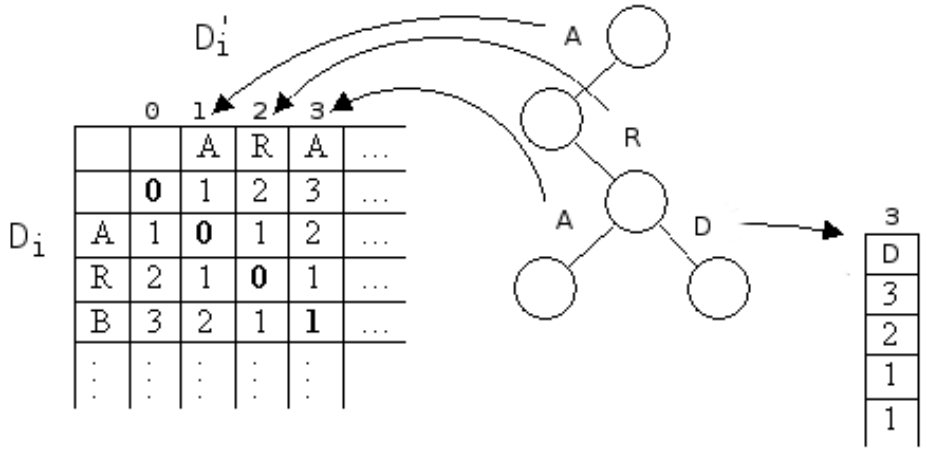

Figura 4.4: Procedimiento de bactracking en el árbol.

sufijo de A'_i con k' errores, entonces para que exista una ocurrencia es necesario que D_i calce con un prefijo de D'_i con a lo sumo con $k - k'$ errores. Este resultado se puede generalizar, ya que si en cualquier momento el error calculado $sed(A_i, A'_i)$ excede k, entonces no es posible que exista una ocurrencia del patrón, y entonces se pasa al siguiente nodo.

Para calcular $sed(A_i, A'_i)$ es necesario tener en cuenta que A'_i puede encontrarse:

- Completamente contenida en el bloque de calce primario de O_i, representado por el nodo x. En este caso A'_i es el mismo para todo el subárbol. La situación es similar en el caso de que $A'_i = \varepsilon$.

- Completamente contenida en los bloques que anteceden al nodo x. En este caso A'_i es distinto para cada nodo del subárbol.

- Parte de A'_i en el nodo x, y el resto en los bloques que lo anteceden.

Podemos ver ejemplificado esto en la figura 4.2. Para la búsqueda ahí presentada, donde la parte O_i corresponde al texto *"ab"*, es claro que parte de A'_i (la letra *"l"*) se encuentra en el mismo bloque que O_i, sin embargo el resto (*"a_ a"*) está en el bloque anterior. Si la parte O_i fuera el texto *"ar"*, entonces es claro que todo A'_i se encontraría en los bloques anteriores.

Para poder reutilizar el máximo de información, la verificación de A_i se realiza en sentido inverso. Es decir, la columna de distancia de edición se construye para el string A_i^r. Al realizar la verificación de forma inversa es posible ahorrarse la necesidad de leer el bloque completo y luego darlo vuelta. La búsqueda de A_i se hace distinto dependiendo de dónde se encuentre A'_i:

- Si se encuentra absolutamente contenida en el bloque es posible determinar el número de errores de esa parte ($sed(A_i, A'_i)$)para todo el subárbol sin necesidad de volver a realizar un cálculo posterior.

- Si A'_i se encuentra sólo en los bloques anteriores, cada verificación parte con una columna recién inicializada.

- Si A'_i está dividido en $A'_i = A'''_i A''_i$, con A''_i contenido en el bloque O_i y A'''_i en bloques anteriores, entonces el cálculo de $ed(A_i, A''_i)$ es común a todo el subárbol y las búsquedas en los bloques anteriores se inicializan con A''_i evitando reprocesarlo cada vez.

Finalmente es necesario un arreglo H para conservar la última fila de la matriz conceptual del cálculo de $ed(D_i, D'_i)$. El motivo de esto es que para todo calce secundario, parte de D'_i ya se ha recorrido, y dependiendo de $sed(A_i, A'_i)$ en ese nodo, puede que exista un calce terminado en algún prefijo ya recorrido de D'_i.

En términos de implementación, se utiliza un stack para almacenar los estados del bactracking. Esto ocupa menos espacio que realizar un recorrido recursivo en el árbol.

El algoritmo de búsqueda para la parte O_i es el siguiente:

- Se inicializan las columnas de cálculo de distancia de edición $Columna_A$ y $Columna_D$ para A^r_i y D_i, respectivamente. Se inicializa el arreglo H, y $prof$ y $Columna_D$ se almacena en el stack.

- Se buscan los calces primarios de tipo 1 de O_i.

- Para cada nodo x correspondiente a un calce primario:

    - Se obtiene el bloque que lo representa. De existir, se determina la parte A''_i de A'_i contenida en el bloque y la parte A'''_i en bloques previos. Sus longitudes son a''_i y a'''_i, respectivamente.
    - Actualizamos la $Columna_A$ procesando A''^r_i.
    - Verificamos el calce en x. Para esto :
        - Se hacen copian las columnas de distancia.
        - Se obtienen $a_i+k-a''_i$ caracteres correspondientes a A'''_i de los bloques $x-1$, $x-2$, ... Se obtiene el error $k' = sed(A_i, A'_i)$.

- Si $k' \leq k$, obtenemos $d_i + k$ caracteres de los bloques $x+1$, $x+2$, ..., y se computan las distancias. Los calces encontrados son reportados.

- A continuación comienza el backtracking. Para cada calce secundario y:
  - En base al indicador $prof$ de profundidad relativa se determina si es un hijo, hermano o ancestro del nodo anterior.
  - Si es un hijo, implica que bajamos en el trie. Entonces:
    - Se obtiene el carácter por el cual se llegó a él. Se copia $Columna_D$ del padre y se actualiza esta columna y el arreglo H en base al nuevo carácter. La columna se almacena en el stack.
  - Si es un hermano quiere decir que seguimos en el mismo nivel del nodo anterior. A continuación:
    - Se borra del stack las columnas del hermano previo, y se retrocede una posición en H.
    - Se obtiene el carácter por el que se llegó a este hermano. Copiamos $Columna_D$ del padre, y actualizamos H y $Columna_D$ en base al carácter. La nueva columna se almacena en el stack.
  - Finalmente, si es un ancestro quiere decir que hemos subido en el árbol. Las subidas no son necesariamente progresivas, es decir, podemos subir más de un nivel. En ese caso:
    - Se borran del stack todos los estados que tienen una profundidad $prof$ mayor o igual a la del nodo actual, y se retrocede en H tantos espacios como niveles se subieron.
    - Se obtiene el carácter por el cual se llegó a este nodo desde su padre.
    - Se copia $Columna_D$ del padre y se actualiza ésta y el arreglo H en base al carácter obtenido. La nueva columna se almacena en el stack.
  - Una vez realizado esto, se verifican las ocurrencias:
    - Se hacen copias de las columnas de distancia.
    - Se obtienen $a_i + k - a''_i$ caracteres correspondientes a A'''_i de los bloques $y-1$, $y-2$, ... Se obtiene el error k'.
    - Se revisa en el arreglo H si $H[h] + k' \leq k$, para algún h. De ser así las ocurrencias son reportados.

- ◇ Si tras la actualización con el carácter por el que se bajó en los distintos casos el error $ed(D_i, D'_i[j \ldots prof]) + k' \leq k$. de ser así, se reporta una ocurrencia.
- ◇ Finalmente, si $k' \leq k$, obtenemos $d_i + k - prof$ caracteres de $y+1$, $y+2$, ..., y se computan las distancias. Los calces encontrados son reportados.

Al terminar el procedimiento se han recorrido los subárboles de todos los calces de tipo 1 de O_i. Los calces de tipo 2 y 3 se revisan utilizando las distancias de edición para obtener el beneficio de detenerse a tiempo si es que $j > k$, pero no se hace ninguna elaboración sobre sus verificaciones. La razón para esto radica en que los calces de tipo 2 y 3 son todos diferentes. Esto es producto que el calce de O_i se extiende a través de dos bloques en el tipo 2 y de tres o más bloques en el tipo 3, donde estos conjuntos de bloques son diferentes y únicos para cada calce. Esto resulta en que no es posible reutilizar información entre los calces de cada tipo.

4.4. Optimización de la partición

En la sección 4.3 se consideró una de las dos posibilidades de optimización que el procedimiento seleccionado permite. En esta sección discutiremos una optimización relacionada con el algoritmo de búsqueda mismo.

Hasta este minuto la división del patrón se ha hecho en partes iguales, en la medida de lo posible. Si bien esto resulta razonable y simple, las partes resultantes tienen un gran impacto en los tiempos de búsqueda.

Por ejemplo en nuestro texto "*alabar a la alabarda para apalabrarla*", si buscamos "*labra*" con dos errores, las partes resultantes serán "*la*", "*br*" y "*a*". Esto genera un total de 22 verificaciones. Sin embargo si el mismo patrón se divide en "*la*", "*b*" y "*ra*" sólo requiere 10 verificaciones.

Es claro que el número de verificaciones a realizar depende directamente de cuán comunes son las partes resultantes de la partición dentro del texto. La mejor selección de particionamiento será aquella que minimice el número total de verificaciones que deben realizarse en el índice. Esto se puede determinar con la siguiente recurrencia, donde $M_{i,j}$ corresponde al mínimo de ocurrencias que se deben verificar al dividir $P_{i,n}$ en j partes. En nuestro caso buscamos $M_{1,k+1}$.

$$M_{i,j} = \min_{i \le i' \le m} (occ(P_{i...i'}) + M_{i'+1,j-1}), \qquad i = 1 \ldots m, \quad j = 1 \ldots k+1 \quad (4.2)$$

$$M_{i,1} = occ(P_{i...m}) \quad (4.3)$$

En esta fórmula $occ(P_{i...j})$ corresponde al número total de calces del substring $P_{i...j}$ en el texto, es decir:

$$occ(P_{i...j}) = occTipo1(P_{i...j}) + occTipo2(P_{i...j}) + occTipo3(P_{i...j})$$

El problema de esto es que el tiempo que toma calcular la mejor partición probablemente resulte mayor que la ganancia obtenida en el tiempo de búsqueda. Para evitar esta situación se debe modificar el cálculo de la fórmula para poder hacerlo lo más rápido posible, aún si esto no garantiza que se obtenga la mejor partición.

Las primeras modificaciones dicen relación con el cómputo de *occ*. Si bien los calces de tipo 1 son relativamente baratos de calcular, $occTipo2$ y $occTipo3$ resultan mucho más caros. Para paliar esto, sólo $occTipo1$ es efectivamente calculado, $occTipo2$ es estimado en base a estos calces y $occTipo3$ se ignora, ya que probablemente corresponda a pocos calces, en comparación a las otras.

La estimación para $occTipo2$ es muy simple: según la sección 3.3.2, para buscar las ocurrencias de tipo 2 en el LZ-index el patrón se divide en dos partes $P_{1,w}$ y $P_{w+1,n}$, de todas las formas posibles. La idea es encontrar aquellas que terminan en un bloque y cuya continuación empieza en el siguiente. Para nuestra estimación se calcula la probabilidad de que el bloque a continuación de uno que termina con $P_{1,w}$ empiece con $P_{w+1,n}$, como si fueran eventos independientes. Es decir calculamos la probabilidad conjunta de los eventos, que corresponde aproximadamente a:

$$\frac{calces\ primarios\ de\ P_{1,w}}{n_{bloques}} \times \frac{calces\ primarios\ de\ P_{w+1,m}}{n_{bloques}}$$

Esto se convierte en un número de calces multiplicándolo por $n_{bloques}$.

Resulta evidente que durante el cálculo de la recurrencia será necesario obtener el valor de $occ(P_{i...j})$ para el mismo substring más de una vez. Para no perder el trabajo ya realizado se utiliza una matriz G de $m \times m$, donde en la celda $G[i,j]$ se almacena el número calces obtenidos para el pedazo $P_{i...j}$.

Se puede notar que $occ(P_{i...j}) \geq \text{máx}(occ(P_{i-1...j}), occ(P_{i...j+1}))$, puesto que el substring $P_{i...j}$ está completamente contenido en los substrings $P_{i-1...j}$ y $P_{i...j+1}$.

Esta observación permite acelerar el cálculo de la recurrencia utilizando máx$(occ(P_{i-1\ldots j}), occ(P_{i\ldots j+1}))$ como una cota. Cuando se necesita saber el valor de $G[i,j] = occ(P_{i\ldots j})$, antes de calcularlo se revisa si utilizando $G[i-1,j]$ o $G[i,j+1]$ el valor obtenido sólo para el tramo $P_{i\ldots j}$ es mayor que el mínimo actual para todo P. De ser así, no es necesario determinar el valor real de los calces, puesto que esa partición no es la óptima. De ser menor, se computa $occ(P_{i\ldots j})$ para la celda.

El cálculo de la recurrencia se realiza aumentando i, y para cada i, aumentando j. En cada paso el mínimo de las ocurrencias se calcula disminuyendo i', es decir, se comienza con m, hasta llegar a i. El propósito de este recorrido es poder contar con una cota para la celda $G[i,j]$ antes de calcularla, y además avanzar de las celdas más baratas de calcular (mayor largo, menos ocurrencias) a las más caras. La matriz G se inicializa con 0, de esta manera si una celda que no está calculada es utilizada como cota, se obliga al cálculo del valor real de las ocurrencias.

Adicionalmente la fórmula de la recurrencia se modifica de la siguiente manera:

$$M_{i,j} = \min_{i+l \leq i' \leq m} (occ(P_{i\ldots i'}) + M_{i'+1,j-1}) \qquad i = 1 \ldots m, \quad j = 1 \ldots k+1$$

Donde $l = \lfloor m/(k+1) \rfloor$. La idea tras esta modificación es buscar pedazos que sean como mínimo del largo de la partición equiespaciada. La heurística tras esto es que en general pedazos muy cortos generan demasiadas verificaciones y por lo tanto es improbable que sirvan, pero calcular sus calces es costoso. Entonces buscamos acotar el largo mínimo de las partes que deseamos obtener.

Finalmente, en cada celda $M_{i,j}$ se almacena el mínimo obtenido y el índice i' para el cual éste se obtuvo. Si bien este índice no se utiliza en el cómputo de la recurrencia, es necesario al recuperar la partición a utilizar. Para esto se recorre la matriz M para $i = 0$, $j = k+1$ y se busca el mínimo de ocurrencias. Al encontrarse, se guarda i' que representa el primer índice donde cortar el patrón. A continuación se busca el siguiente en $M_{i'+1,j-1}$ hasta obtener todas las partes ($j = 0$).

La búsqueda en el índice se realiza utilizando la implementación que factoriza verificaciones vista en la sección anterior con las partes obtenidas del proceso de optimización.

Cabe notar que inicialmente se pensaba experimentar con distintos l para evaluar como influía esto en el tiempo de optimización y en la obtención de cortes de mejor o peor calidad. Tras algunas pruebas iniciales se determinó que el tiempo de optimización es tan alto que no valía la pena la experimentación adicional.

4.5. Implementaciones sobre otro índice

La idea tras esta implementación es proveer al trabajo desarrollado de un punto de comparación significativo. Si bien es interesante saber cómo funciona la búsqueda aproximada indexada con respecto a su par en texto plano, resulta muy atractivo también saber cómo se desempeña este índice con respecto a algún otro para esta misma labor.

Para este propósito se tomó la implementación del FM-index desarrollada en [10] y se agregó la funcionalidad de búsqueda aproximada.

Al igual que en el caso del LZ-index, no se desea modificar la estructura del índice, por lo que se implementó la solución básica discutida en la sección 4.2 y la optimización de la partición vista en la sección 4.4. No se intentó cambiar la búsqueda aproximada por dos cálculos de distancia como en la sección 4.3, ya que para cada verificación en el FM-index es necesario obtener todo el texto que se encuentra entre dos posiciones muestreadas para poder localizar el calce. Esto dificulta la aplicación de la técnica descrita.

En el caso de la implementación con optimización de la partición la única diferencia con el algoritmo descrito es que se utilizan el número de calces exactos para el cálculo de la recurrencia. Esto aprovecha que el FM-index es particularmente rápido para contar los calces.

Para la implementación básica se utilizó el esquema simple de particionamiento discutido en la sección 4.2.

Posteriormente y como se discutió en la sección 3.1, tras buscar una parte del patrón se obtiene un intervalo en el arreglo de sufijos correspondiente a los calces. Para cada valor en el intervalo se busca y descomprime el texto necesario para la verificación. En ambas implementaciones la verificación se realizó utilizando programación dinámica.

Capítulo 5

Resultados experimentales

Para evaluar las soluciones implementadas se realizaron experimentos con cuatro colecciones de texto:

ZIFF2 Corresponde a 84 Megabytes (Mb) obtenidos del disco "*ZIFF-2*" de la colección *TREC-3*.

ADN Archivo de 52 Mb correspondiente al ADN del *Homo Sapiens* obtenidos de la base de datos de secuenciación genética *GenBank*[1], con líneas cortadas cada 60 caracteres.

PROTEINAS 71 Mb de la base de datos de secuenciación de proteínas *Swiss-Prot*[2].

JRRT El texto completo de la trilogía "*El señor de los anillos*", de *J.R.R. Tolkien*. El archivo ocupa 2,8 Mb.

Las pruebas se ejecutaron en un procesador Pentium 4 de 3,06 GHz con 1 Gigabyte (Gb) de RAM y 512 Kilobyte (Kb) de caché, corriendo Linux kernel 2,6,8. El código se compiló con los *flags -O2 -fomit-frame-pointer*.

Los patrones utilizados para las búsquedas se obtuvieron de cada texto. Para esto se leyó el largo requerido a partir de una posición al azar. En el caso del `ADN` se omitieron patrones que contuvieran la letra 'N', que representa un carácter desconocido y no es buscable. Para `ZIFF2` se omitieron los patrones que contuvieran *tags* o caracteres no visibles como '&'. Para la obtención de los patrones de

[1] http://www.ncbi.nlm.nih.gov/Genbank/index.html
[2] http://au.expasy.org/sprot/

`PROTEINAS` se omite la línea que identifica cada proteína, compuesta por el nombre y otros datos.

Cada patrón se buscó con $k = 0 \ldots i$ tal que el *nivel de error* $\alpha = i/m = 20\,\%$.

El FM-index original realiza una compresión muy eficaz del texto, lo que hace que ocupe muy poco espacio. Esto repercute en la velocidad de recuperación del texto, siendo muy lento para esta tarea. Para comparar de una manera justa ambos índices se utiliza una implementación que ocupa mucho más espacio y que lo usa de forma muy eficiente. Se configura este FM-index para realizar un muestreo del texto tal que use el mismo espacio en memoria principal que el LZ-index. En la tabla 5.1 podemos ver el espacio que usan los índices en memoria principal.

Texto	**Tamaño en memoria (Mb)**	**% del texto**
ZIFF2	128	152 %
ADN	63	121 %
PROTEINAS	164	231 %
JRRT	5	179 %

Cuadro 5.1: Tamaño del índice en memoria principal.

Por último, para la búsqueda en texto plano se utiliza una implementación de la división en $k + 1$ partes, que es el método más rápido de búsqueda en la mayoría de los casos de interés. La búsqueda de las partes se realiza con el algoritmo de búsqueda *Set-Horspool*, que corresponde a una extensión del algoritmo *Horspool* [6] para soportar búsquedas multipatrón, y la verificación de las ocurrencias con programación dinámica.

En la tabla 5.2 se presentan los tamaños del alfabeto de cada archivo. Dado que la búsqueda es de texto completo se consideran como caracteres del alfabeto todos los caracteres del texto. Por *alfabeto del patrón* nos referimos al tamaño del alfabeto desde el cual se eligieron los patrones. Por ejemplo, en el caso del texto `ZIFF2`, al remover las líneas antes mencionadas el alfabeto se reduce en 4 caracteres.

Texto	$\lvert\Sigma_{texto}\rvert$	$\lvert\Sigma_{patrones}\rvert$
ZIFF	96	92
ADN	6	5
PROTEINAS	88	24
JRRT	111	111

Cuadro 5.2: Tamaño del alfabeto para cada texto.

Si bien `JRRT` y `ZIFF` corresponden a texto en inglés, la primera colección tiene un alfabeto más amplio producto del uso por parte de Tolkien de tildes y caracteres

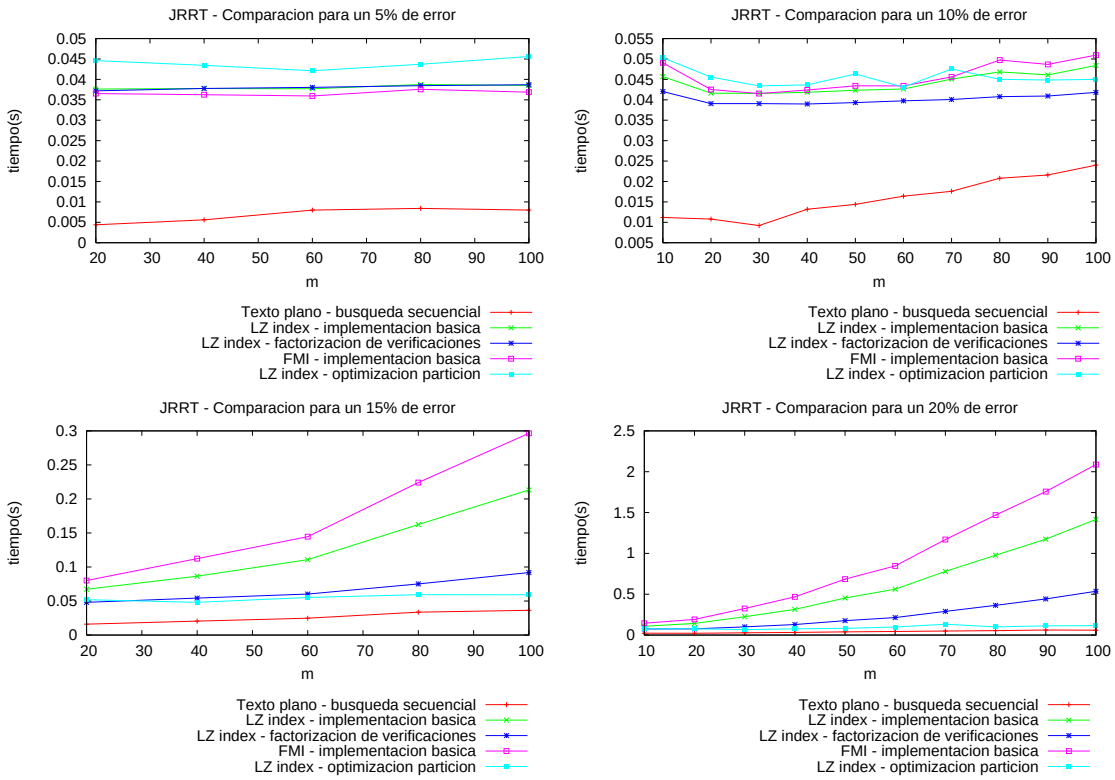

Figura 5.1: Gráficos para `JRRT`.

con diéresis (ä, ë, etc), entre otras cosas.

Los resultados experimentales se pueden observar en los gráficos para `JRRT`(figura 5.1), `ZIFF2` (figura 5.2), `ADN` (figura 5.3) y `PROTEINAS`(figura 5.4). En ellos se representan los valores de tiempo de búsqueda versus el largo del patrón según el porcentaje de error para cada texto. En cada uno de estos gráficos se compara el desempeño de las distintas implementaciones. Para la implementación de optimización de la partición sólo se presenta el tiempo de búsqueda considerando $l = \lfloor m/(k+1) \rfloor$, esto debido a que los tiempos de optimización resultan demasiado elevados, como se comentó en la sección 4.4.

La primera observación que podemos realizar es que en general la implementación factorizando verificaciones es más rápida que la implementación básica, y la búsqueda mediante la implementación que optimiza la partición es más rápida que las dos anteriores. Esto corresponde a los resultados esperados. Efectivamente las estrategias implementadas entregan mejoras en los tiempos de búsqueda.

También como era esperado se puede observar que la implementación básica sobre el LZ-index es más rápida que esta misma implementación sobre el FM-index. Podemos notar que el comportamiento de éstas es prácticamente idéntico, con la

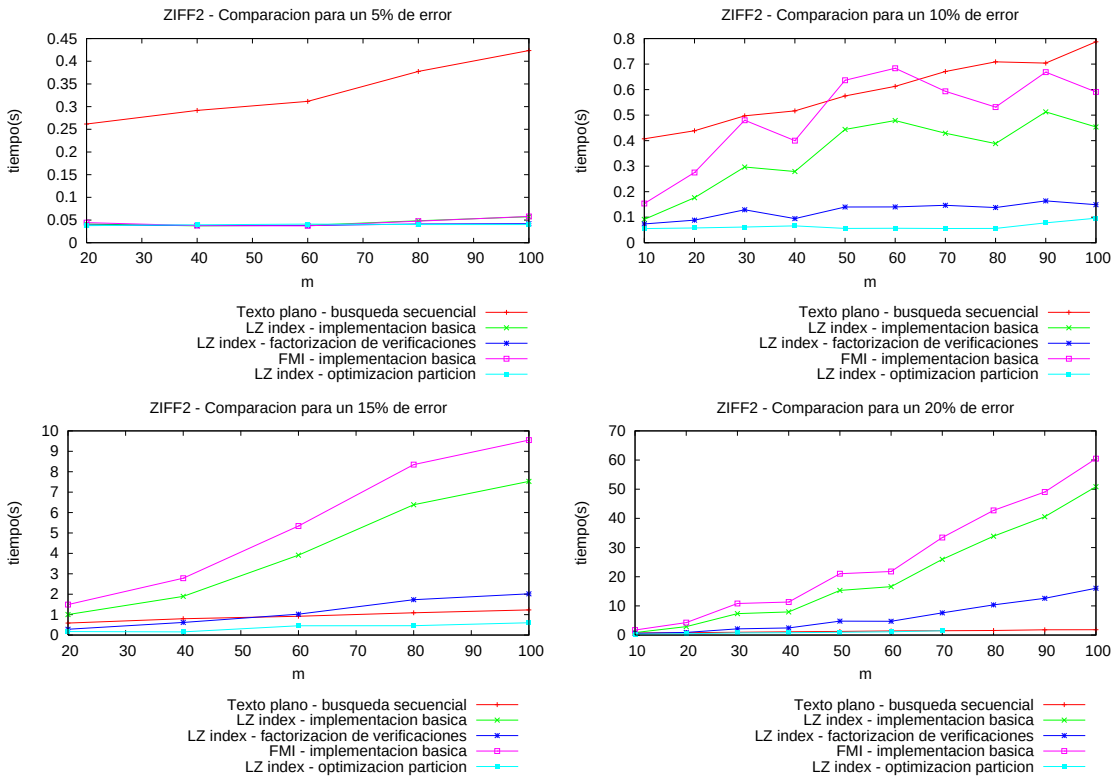

Figura 5.2: Gráficos para `ZIFF2`.

salvedad de un desplazamiento en los tiempos de búsqueda. Esta diferencia en el desempeño se asocia al más alto costo de la obtención del texto en el FM-index.

Llama la atención el que las curvas no sean monótonamente crecientes, como sería lo esperado. Esto se puede observar por ejemplo en los gráficos para un error del 10 % sobre `ADN` y `ZIFF2`.

La explicación para esto reside en que para un porcentaje de error dado y dependiendo de el largo del patrón, el tamaño de las partes es distinto. En el caso de la figura mencionada, podemos notar que cuando el patrón es de largo 40 el tiempo de búsqueda se reduce. Ahora, un error del 10 % en un patrón de largo 30 corresponde a 3 errores, lo que implica dividir el patrón en 4 partes de aproximadamente 7 caracteres cada una. Sin embargo, para un patrón de largo 40 el mismo porcentaje de error corresponde a 4 errores. Esto significa dividir el patrón en 5 partes de 8 caracteres.

Como ya se ha discutido antes, partes más cortas generan un mayor número de verificaciones. Esto vuelve a recalcar la importancia que tiene la selección de las partes, como se vio en la sección 4.4.

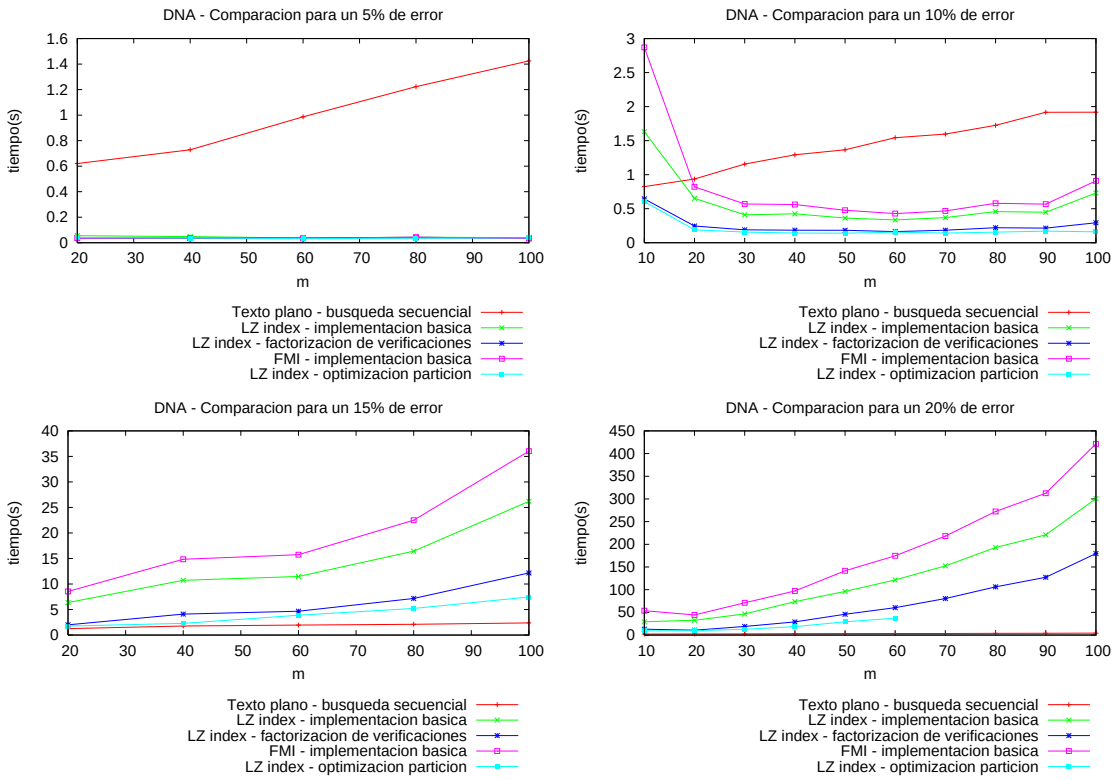

Figura 5.3: Gráficos para `ADN`.

Observando con más detalle cada conjunto de gráficos podemos notar que para `ZIFF2`, `ADN` y `PROTEINAS` las búsquedas con un errores de 5 % y 10 % son más rápidas al realizarse de manera indexada que sobre el texto plano. Por el contrario, sin importar el porcentaje de error, la búsqueda secuencial en `JRRT` resulta ser la mejor alternativa.

Esto sirve como indicador de que existe un tamaño de archivo a partir del cual indexar resulta interesante. Antes de esto siempre resulta mejor buscar con el método secuencial.

Es interesante notar que en el gráfico para un error del 10 % en `PROTEINAS`, cuando el patrón es de $m = 90$ caracteres los tiempos de las implementaciones básicas se degradan notablemente. Debido a que la factorización de las verificaciones se comporta de mejor manera, es posible pensar que los patrones utilizados resultaron en calces de tipo 1 con un subárbol importante que revisar (pensemos que si el calce de la primera parte es de tipo 1, entonces el subárbol tiene una profundidad de 81 niveles).

A medida que aumentamos el porcentaje de error el comportamiento varía dependiendo del texto.

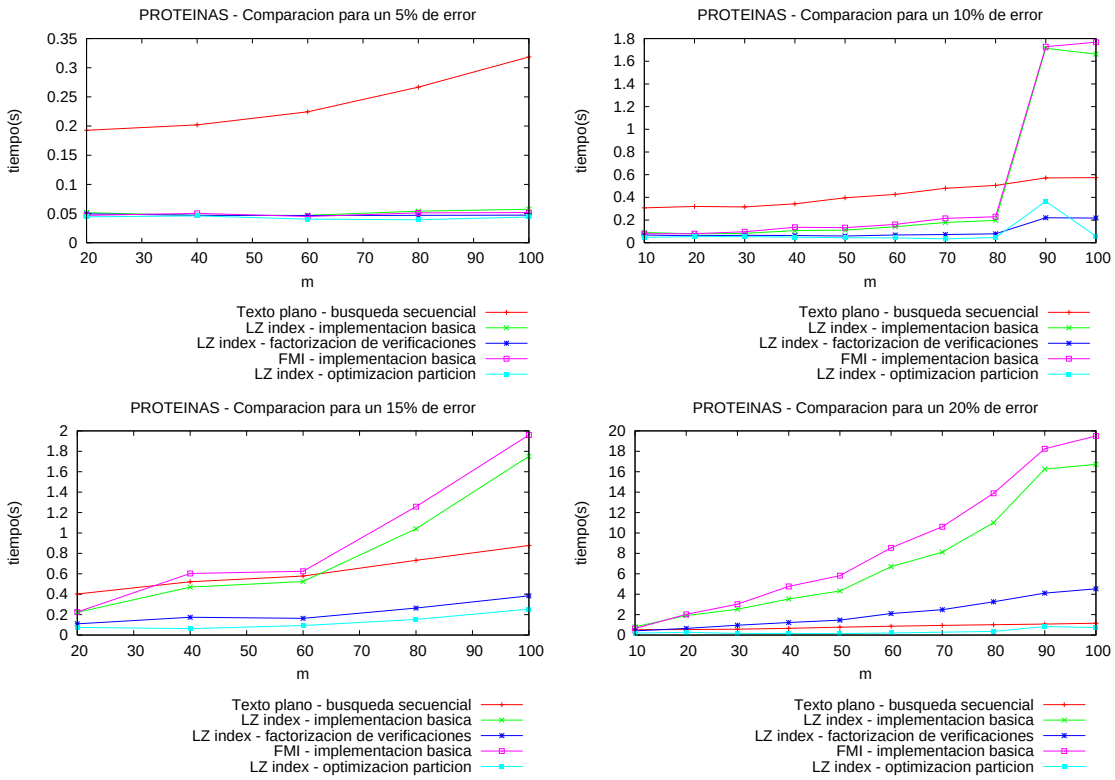

Figura 5.4: Gráfico para `PROTEINAS`.

En el caso de `ADN`, a partir de un error del 15 % la búsqueda secuencial es siempre más rápida que la mejor alternativa indexada. Esto se debe al alfabeto del texto. Al aumentar el porcentaje de error las partes que hay que verificar son cada vez más pequeñas. Como el alfabeto también es pequeño, la probabilidad de que una cadena corta se encuentre muchas veces en el texto es alta. Esto significa realizar muchas verificaciones. La ventaja la búsqueda secuencial reside en que hace sólo una pasada por el texto.

Revisando los gráficos de `ZIFF2`, cuando el error es de un 15 % es posible notar que las implementaciones básicas se vuelven más lentas que la búsqueda secuencial, y la implementación factorizando verificaciones sólo es más rápida para patrones de $m < 50$. La búsqueda con la partición optimizada resulta ser más rápida, incluso con un error del 20 %.

En `PROTEINAS` se puede observar que para un 15 % la implementación básica sobe el FM-index sólo es mejor que la búsqueda secuencial para patrones de hasta $m = 30$. A la vez, la misma implementación pero sobre el LZ-index es más rápida para patrones de hasta $m = 60$ caracteres. Sin embargo, las otras dos implementaciones son entre 3 y 5 veces más rápidas que la búsqueda secuencial. Para el 20 % de error

sólo la partición optimizada resulta más rápida que la búsqueda sobre el texto plano.

Capítulo 6

Conclusiones

En este trabajo se estudia la implementación de la búsqueda aproximada sobre un autoíndice sucinto sin modificar la estructura del índice. El índice en cuestión es el LZ-index [10], basado en el algoritmo de compresión de Ziv-Lempel.

Se elige el LZ-index ya que si bien es más lento para contar las ocurrencias de strings sin errores, su diseño permite recuperar el texto de las ocurrencias más rápido que en otros índices similares. Esto resulta fundamental para la solución propuesta.

El objetivo principal de este trabajo fue implementar la búsqueda aproximada utilizando tres soluciones con niveles crecientes de optimización en busca de mejores tiempos de búsqueda. Lo primero fue una implementación básica de la búsqueda aproximada. Después se mejoró el recorrido sobre el índice mediante la factorización de verificaciones en las ocurrencias de tipo 1. Finalmente se buscó reducir las verificaciones mediante un particionamiento óptimo. Una vez concluidas las implementaciones, se debía conducir experimentos para evaluar las soluciones. En teoría este método debería ser más rápido, bajo ciertas condiciones, que la mejor búsqueda aproximada sobre texto plano.

Junto con las comparaciones antes mencionadas se habilitó la misma funcionalidad en el FM-index [2], otro autoíndice sucinto. El objetivo de esto fue evaluar la solución dentro del ámbito de la búsqueda indexada. Para que la comparación fuera imparcial, se configuraron los índices de manera de usar el mismo espacio en memoria.

En los experimentos se utilizaron archivos de texto de diversas propiedades con el fin de medir el comportamiento de las soluciones ante éstas. Estos corresponden a un archivo de secuencias de ADN, un archivo de secuencias de proteínas, un archivo con texto en inglés y un archivo con texto en dos idiomas.

Los resultados obtenidos apoyan la tesis propuesta. Las solución más simple

resulta ser la más lenta y a medida que se incorporan optimizaciones los tiempos de búsqueda se reducen.

También cabe notar que la implementación básica sobre el LZ-index es más rápida que esta misma implementación en el FM-index. Esto es debido a que el FM-index es más lento para recuperar el texto, incluso con el espacio extra que se le otorga para igualar el espacio usado por el LZ-index.

En el caso de textos cortos, ninguna aproximación indexada supera la búsqueda secuencial sobre el texto. Esto se debe al tamaño del archivo, y nos permite inferir que no vale la pena indexar archivos muy pequeños.

En textos mejores (≥ 50 Mb), y para un 5 % y 10 % de error, resulta más rápida la búsqueda indexada. Esto abre nuevas posibilidades en los campos donde estos rangos de error son habituales.

Para porcentajes de error superiores, el comportamiento varía dependiendo del alfabeto y el tamaño del texto. Por ejemplo, en un archivo grande y un alfabeto pequeño (`ADN`), la búsqueda indexada se degrada rápidamente. Esto no ocurre de la misma manera para `PROTEINAS` y `ZIFF2`.

Es particularmente interesante el que la búsqueda con la partición optimizada está a la par de la búsqueda secuencial para un porcentaje de error mayor al 10 %. En algunas ocasiones incluso la supera. El problema de esta solución son los tiempos de optimización, que anulan la ganancia obtenida en la búsqueda. Desarroollar una heurística de tiempo constante sería muy beneficioso.

Para las aplicaciones actuales de la búsqueda aproximada es claro que la indexación del texto ofrece un excelente alternativa a los métodos tradicionales. Una posibilidad muy interesante es mejorar el método de optimización, que de ser posible, podría llevar a reemplazar definitivamente la búsqueda secuencial en ámbitos como la biología computacional.

6.1. Trabajo futuro

Existen variadas posibilidades de investigación futura, tanto dentro del ámbito de la búsqueda aproximada, como en el mismo LZ-index.

Como se discutió, hasta el momento no hay implementaciones de búsqueda de expresiones regulares en un autoíndice sucinto. Esto resulta sumamente deseable, puesto que es una herramienta necesaria para poder reemplazar la búsqueda secuencial. Para este propósito se podría utilizar una aproximación similar a la

usada en este trabajo, es decir, obtener un conjunto de strings necesarios para una ocurrencia, buscar estos en el índice y posteriormente verificar los calces.

Los esquemas aquí implementados aún pueden ser mejorados. Por ejemplo, se podría factorizar las partes del patrón que son iguales, y de esta manera durante un recorrido del índice verificar las distintas posibilidades.

El área más interesante para mejorar es la optimización de la partición. Esto producto que resulta decisivo para que la implementación que sólo factoriza verificaciones sea más rápida que la búsqueda secuencial en la mayoría de los casos y no sólo para niveles de error de hasta $10\,\% - 15\,\%$.

Bibliografía

[1] I. H. Witten A. Moffat, R. Neal. Arithmetic coding revisited. *Proceedings of the Data Compression Conference, Snowbird, Utah*, pages 202–211, March 1995.

[2] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of 41st IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[3] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of 12th IEEE Symposium on Discrete Algorithms*, pages 269–278, 2001.

[4] Philip Gage. A new algorithm for data compression. *The C Users Journal*, february 1994.

[5] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of 32nd Symposium of Theory of Computing*, pages 397–406, 2000.

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

[7] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, pages 1098–1102, September 1952.

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

[9] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. *Video and Data Compression*, march 1979.

[10] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.

[11] K. Sadakane. Compressed Text Databases with Efficient Query Algorithms based on the Compressed Suffix Array. In *Proceedings of 11th International Symposium on Algorithms and Computation*, number 1969 in LNCS, pages 410–421, 2000.

[12] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–342, 1977.

[13] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.