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

UN ÍNDICE COMPRIMIDO SIMPLE INDEPENDIENTE DEL ALFABETO

ALEJANDRO JOSÉ SALINGER LISBOA

COMISIÓN EXAMINADORA		CALIFICACIONES:		
		NOTA (n°)	(Letras)	FIRMA
PROFESOR GUÍA	:			
Gonzalo Navarro		.....	.....	.....
PROFESOR CO-GUÍA	:			
Ricardo Baeza		.....	.....	.....
PROFESOR INTEGRANTE	:			
Claudio Gutiérrez		.....	.....	.....
NOTA FINAL	:			
EXAMEN DE TÍTULO		.....	.....	.....

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

SANTIAGO - CHILE
ENERO - 2005

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN
FECHA: 13 DE ENERO DE 2005
AUTOR: ALEJANDRO JOSÉ SALINGER LISBOA
PROF. GUÍA: GONZALO NAVARRO.

UN ÍNDICE COMPRIMIDO SIMPLE INDEPENDIENTE DEL ALFABETO

Un índice de texto completo es una estructura construida sobre un texto que permite la búsqueda eficiente de cualquier subcadena en éste. Este trabajo se enmarca en el contexto de los índices de texto que reemplazan el texto mismo, es decir, que no lo necesitan para operar y que pueden reconstruir cualquier porción del texto. El trabajo consiste de la implementación, generalización, y estudio experimental de la eficiencia en cuanto al tiempo y espacio de un nuevo índice comprimido de texto completo llamado Índice FM Huffman. La idea de esta estructura es mejorar el Índice FM, una estructura basada en la transformación de Burrows Wheeler, eliminando su dependencia del tamaño del alfabeto. Esto se logra comprimiendo el texto con Huffman y operándolo como un texto binario.

Se implementaron los algoritmos necesarios para construir el índice sobre un texto y para consultar las ocurrencias de un patrón, extraer las posiciones de estas ocurrencias y mostrar subcadenas del texto. Para los experimentos se consideraron tres archivos de distinta naturaleza: texto en inglés, ADN y proteínas. Se obtuvieron medidas cuantitativas del espacio ocupado por el índice al ser construido sobre estos tres archivos y se comparó su eficiencia en el tiempo con la de otros índices de texto completo existentes, para los tres tipos de consulta mencionados. El Índice FM Huffman resultó ser el más eficiente para el conteo de ocurrencias y reporte de posiciones en el archivo de ADN, y registró tiempos competitivos para los demás archivos.

Se implementó a su vez una versión más general del índice, el FM Huffman k-ario, el cual generaliza la compresión de Huffman para utilizar un alfabeto de salida de k símbolos en vez de dos. La versión de este índice con $k = 4$ mejoró el tiempo y espacio de la versión original en todos los experimentos, conviertiéndose en la mejor alternativa para el conteo de ocurrencias para los tres archivos y en la mejor para el reporte de posiciones para el archivo de ADN. Fue para este archivo, debido a su baja entropía de orden cero, que ambas versiones del FM Huffman registraron su mejor desempeño.

AGRADECIMIENTOS

En primer lugar quisiera agradecer a mi profesor guía, Gonzalo Navarro, por haberme ayudado a llevar este trabajo por el mejor camino mediante sus consejos sensatos y prácticos, los cuales han trascendido el ámbito de esta memoria y han resultado útiles en varios aspectos de mi carrera universitaria.

Agradezco también a mis padres, quienes con su incondiconal apoyo me permitieron dar feliz término a este trabajo y a esta etapa de mi carrera. Les agradezco a ambos sus consejos y paciencia y a mi madre el haberme provisto de un ambiente adecuado para realizar este trabajo.

Quisiera también agradecer a Angélica Aguirre, por sus innumerables consejos.

Por último, quiero agradecer a Bige, quien con su constante amor y apoyo me ayudó a mantener la motivación necesaria para realizar mi trabajo de la mejor manera posible.

ÍNDICE GENERAL

Capítulo 1

INTRODUCCIÓN

Durante los últimos años hemos sido testigos de un rápido crecimiento en el número de textos disponibles en formato digital. Enciclopedias completas y grandes bases de datos con información de tipo biológico, son ejemplos de recursos que han sido llevados a un formato que permite que sean manejados por un computador. Las ventajas de disponer de grandes documentos en formato digital radican en que éstos pueden ser fácilmente reproducidos, distribuidos o ponerse a disposición de una gran cantidad de personas, independiente de su ubicación geográfica, gracias a los avances de las redes comunicacionales. Quizás la ventaja más importante de contar con un documento en formato digital es que se se pueden realizar búsquedas de información en el texto de manera muy eficiente. Es por esto que el desarrollo de algoritmos eficientes para búsqueda en texto ha cobrado mucha importancia en el último tiempo.

El problema más común en esta área es el de buscar un patrón en un texto. Como respuesta al problema, generalmente nos interesa saber si el patrón está o no en el texto así como también el lugar y contexto donde se encuentra. Los algoritmos más simples realizan una búsqueda secuencial del patrón en el texto, por lo que el tiempo que demoran es proporcional al tamaño del texto. Sin embargo, cuando se trata de buscar en grandes bases de datos, los algoritmos secuenciales no son lo suficientemente eficientes. Por ejemplo, cuando queremos buscar una palabra en internet, podríamos esperar más de un día antes de recibir una respuesta completa si es que realizáramos una búsqueda secuencial en todos los textos que hay en la red, incluso si los tuviéramos todos en memoria RAM. En los casos en que el espacio de búsqueda es lo suficientemente grande como para que una búsqueda secuencial no sea suficientemente rápida, se procesa el texto previamente a la búsqueda, construyendo una estructura que permite realizar búsquedas en un tiempo que deja de ser proporcional al tamaño del texto y que de hecho, resulta ser mucho menor. Esta estructura recibe el nombre de índice.

Existen distintos tipos de índices según las operaciones que permiten. Uno de los índices más usados en el área de recuperación de textos es el índice invertido [13]. Este

tipo de índice es adeacuado para lenguaje natural, y sólo recupera eficientemente palabras o frases. En general su estructura consiste en una lista de palabras y las posiciones del texto en que se encuentran. Por lo tanto, para saber si una palabra está o no en un texto, así como también los lugares en que se encuentra, basta con mirar en el índice la entrada asociada a esa palabra. Este índice ha resultado popular debido a sus bajos requerimientos de espacio y velocidad de consulta. La mayoría de los buscadores de internet ocupan una variación de este índice.

En el último tiempo ha cobrado importancia el uso de herramientas computacionales para la búsqueda de patrones en bases de datos que contienen textos con información organizada de una manera distinta a los textos en español o inglés. Por ejemplo, en el ámbito de la biología, las herramientas computacionales han contribuido a la extracción de información relevante a partir de secuencias de ADN de distintas especies. Una secuencia de ADN es una concatenación de caracteres en el conjunto $\{A, C, G, T\}$. La secuencia no contiene separaciones, es decir, no existe el concepto de palabras. Lo mismo ocurre con las secuencias de proteínas y en general con todas las secuencias biológicas. En estos casos, el índice invertido ya no resulta útil, pues los patrones que se pretende buscar no constituyen palabras en el texto, sino que una subcadena de una secuencia de caracteres. Otro ámbito en el cual se han desarrollado herramientas de búsqueda eficientes es el de las bases de datos multimediales. Estas bases de datos contienen codificaciones de archivos de música, imágenes y video. Dentro de los archivos de música, existen por ejemplo las secuencias MIDI, las cuales son una concatenación de caracteres que representan notas y sus duraciones. Al igual que con las secuencias biológicas, en este tipo de secuencias no existen separaciones, por lo que no se puede clasificar el texto en palabras. Un ejemplo más de este tipo de secuencias lo constituyen los textos en idiomas orientales, como el japonés y el chino. Estos lenguajes contienen símbolos ideográficos y, si bien pueden contener palabras, éstas no son reconocibles sintácticamente en el texto, por lo que los buscadores en esos idiomas interpretan los textos como secuencias de caracteres en un alfabeto muy grande. Otros ejemplos de textos que no se pueden, o resulta difícil separar en palabras son los códigos fuentes de programas computacionales y las series de tiempo.

Para este tipo de textos, se utilizan índices de texto completo, los cuales proveen un acceso eficiente a todas las subcadenas del texto, por lo que permiten entonces buscar un patrón consistente de cualquier secuencia de caracteres sin que necesariamente estos caracteres formen una palabra completa. Ejemplos de estos índices son los árboles de sufijos [2, 3, 4] y los arreglos de sufijos [5]. Estos índices permiten encontrar cualquier subcadena del texto rápidamente. Además, permiten operaciones más complicadas que el índice invertido, como por ejemplo realizar búsquedas aproximadas o el descubrimiento

de patrones. Sin embargo, el tamaño de los índices de texto completo es bastante mayor que el de los índices de palabra, llegando a ocupar de cuatro a veinte veces el espacio que ocupa el texto. Por ejemplo, un índice de texto completo construido sobre un texto de 200 Mb ocuparía 4 Gb de memoria.

Las investigaciones recientes se han concentrado en reducir los tamaños de los índices de texto completo [11, 12, 8, 14, 18, 16]. En general, al reducir el espacio que ocupa el índice, puede aumentar el tiempo de búsqueda de patrones. Existe entonces un compromiso entre el tiempo de búsqueda y el espacio utilizado. Existen en la actualidad varios índices llamados *comprimidos* los cuales alcanzan buenos compromisos entre complejidad de tiempo y espacio. Estos índices representan la información necesaria para realizar búsquedas eficientes de manera que ocupen menos espacio que los árboles de sufijos y arreglos de sufijos. La mayoría de estos índices son *oportunísticos*, pues ocupan menos espacio cuando el texto es compresible. A su vez, estos índices son *auto índices*, ya que contienen la información suficiente para reproducir el texto. Este tipo de índice no necesita el texto para operar, y puede reemplazarlo.

Un ejemplo de este tipo de índices es el Índice FM de Ferragina y Manzini [8]. Este auto índice ocupa en la práctica el mismo espacio que el texto comprimido. De hecho, el índice alcanza una tasa de compresión cercana a la de los mejores compresores. Sin embargo, la complejidad espacial contiene una dependencia exponencial en el tamaño del alfabeto, lo que hace que en la práctica resulte imposible su implementación y se deba recurrir a heurísticas, aumentando el tiempo de búsqueda. Este es un ejemplo de que en este tipo de problemas hay veces en que la teoría y la práctica difieren.

El trabajo de esta memoria consiste en estudiar e implementar un nuevo índice de texto completo, llamado *FM Huffman*, el cual pretende ser una mejora al Índice FM, eliminando su fuerte dependencia del tamaño del alfabeto. Esta dependencia no sólo aparece en la complejidad espacial, sino que también en el tiempo necesario para encontrar las posiciones de las ocurrencias y para mostrar subcadenas del texto. El Índice FM Huffman se basa en la idea del Índice FM, con la diferencia de que, antes de construir el índice, comprime el texto para obtener una representación reducida del mismo, la cual resulta en un texto en alfabeto binario. Se construye entonces el índice sobre la secuencia binaria.

Si bien ya existe un desarrollo teórico del Índice FM Huffman [1], este índice no había sido implementado. El objetivo de esta memoria es implementar este índice, estudiar su eficiencia y compararlo en la práctica contra otros desarrollos [16, 18, 8, 22]. La implementación misma del índice resulta interesante pues nacen de ella desafíos como lograr construir el índice sin utilizar demasiado espacio durante la construcción, o idear

implementaciones prácticas y eficientes para soluciones teóricas que resultan inviables de implementar.

Otro objetivo de este trabajo es implementar y estudiar la eficiencia de una versión más general del Índice FM Huffman, llamado FM Huffman k-ario, el cual permite comprimir el texto utilizando un alfabeto de salida de k símbolos en vez de dos.

El segundo capítulo de este documento describe los conceptos básicos necesarios para la comprensión del mismo. El tercer capítulo describe el trabajo relacionado con el tema de esta memoria. En particular, se describen los demás índices involucrados en las comparaciones realizadas. El cuarto capítulo describe la estructura y algoritmos del Índice FM Huffman. En el quinto capítulo se describen algunas consideraciones con respecto a la implementación del índice. El capítulo número seis describe la estructura y algoritmos del Índice FM Huffman k-ario. En el capítulo siete se presentan los resultados obtenidos y su relevancia.

1.1. Trabajo realizado

El objetivo de este trabajo era implementar el Índice FM Huffman y comparar su eficiencia en tiempo y espacio con otras alternativas existentes. Específicamente, la idea era implementar algoritmos eficientes para construir el índice, atendiendo al espacio y tiempo de construcción y obtener medidas cuantitativas del espacio que ocupa el índice según las características del texto sobre el cual se construye. Adicionalmente, se requería implementar los algoritmos de consulta del índice y comparar su desempeño con los algoritmos de consulta de los demás índices.

La implementación del índice se dividió en varias partes. Primero se implementó la construcción del índice, es decir, un programa que, dado un texto, lo procesa y escribe en disco las estructuras que conforman el índice y que servirán luego para ejecutar las consultas. Se consideraron tres tipos de consultas que el usuario puede realizar: conteo de ocurrencias de un patrón en el texto, reporte de las posiciones de las ocurrencias y muestra de texto alrededor de las ocurrencias. La primera etapa de la construcción del índice incluyó sólo las estructuras necesarias para las consultas del primer tipo.

Luego se implementó un programa que lee desde el disco las estructuras de un índice previamente construido y permite a un usuario ingresar patrones y obtener el número de ocurrencias de ellos en el texto correspondiente utilizando la funcionalidad de conteo de ocurrencias del índice. Posteriormente se agregó al programa de construcción del índice la construcción de las estructuras necesarias para realizar los otros dos tipos de consultas, vale decir, mostrar las posiciones de las ocurrencias y un texto alrededor de

ellas. Se agregaron luego estas funcionalidades al programa de consultas.

Una vez que se obtuvo la implementación completa del índice, se procedió a realizar los experimentos de comparación con otros índices. Estos índices fueron el Índice FM [8], el CSA [18], el Índice LZ [16] y el Índice RLFM [22]. Para los experimentos, se construyeron estos índices sobre textos en inglés, secuencias de ADN y secuencias de proteínas. Para los tres archivos se compararon los tiempos de conteo de ocurrencias en función de los largos de los patrones para cada índice. Luego se construyeron cada uno de los índices utilizando distintos parámetros para determinar el espacio que ocupan y se midió el tiempo que demora cada uno en reportar las ocurrencias encontradas y en mostrar caracteres alrededor de ellas, en función del espacio ocupado por el índice.

Luego se implementó el Índice FM Huffman k-ario, implementando primero las estructuras y funciones necesarias para el conteo de ocurrencias y agregando más tarde las estructuras y funcionalidades para reporte de posiciones y muestra de texto. Se llevaron a cabo luego los experimentos de conteo de ocurrencias, reporte de posiciones y muestra de texto para la versión con $k = 4$ de este índice, comparándolo así con la versión original del Índice FM Huffman y con los demás índices.

1.2. Resultados obtenidos

Con respecto al espacio del Índice FM Huffman, se obtuvo que para contar las ocurrencias en el texto en inglés, el índice ocupa 1.68 veces el texto. Es importante mencionar que este espacio incluye a la representación del texto. Recordemos que este índice es un *auto índice*, es decir, no necesita el texto para operar. Para el archivo de ADN el índice ocupa 0.76 veces el texto y para la secuencia de proteínas, 1.45. Comparado con los otros índices, el espacio ocupado por el FM Huffman resultó ser relativamente mayor para el texto en inglés y proteínas, aunque para el archivo de ADN, el espacio ocupado fue uno de los menores.

Con respecto al tiempo de conteo, el tiempo del FM Huffman se ve apenas superado por el del Índice FM con los archivos de proteínas y texto en inglés, mientras que resulta ser el mejor para el archivo de ADN.

A partir del experimento de reporte de posiciones, se obtuvo que con el archivo de ADN, el Índice FM Huffman registra los menores tiempos con respecto a los otros índices, para iguales espacios ocupados. Para los otros dos archivos, en cambio, el FM Huffman se ve superado por el Índice FM y supera a los demás índices pero ocupando mayor espacio que ellos.

El experimento de muestra de caracteres reveló que el Índice FM Huffman aumenta

considerablemente su espacio al incluir las estructuras necesarias para la muestra de texto. El tiempo de muestra del primer carácter fue menor para el Índice FM, con los tres archivos, seguido por el FM Huffman para el archivo de ADN. Para los otros dos archivos, al igual que para el reporte de posiciones, el FM Huffman registra tiempos competitivos, pero ocupando más espacio que los demás índices.

Los expermentos realizados con el Índice FM Huffman k-ario revelaron que con esta versión el tiempo se redujo con respecto a la versión original, reduciéndose a su vez el espacio ocupado. Esta versión del FM Huffman registró los mejores tiempos de conteo para los tres archivos y el mejor tiempo de reporte de posiciones para el archivo de ADN. Pese a no superar al Índice FM para el reporte de posiciones para los otros dos archivos y para la muestra de un carácter de texto para los tres archivos, al reducir el espacio y tiempo del FM Huffman binario, supera a los demás índices en un rango de espacios menores que éste. Además, esta versión registró muy buenos tiempos de muestra por carácter, siendo el menor para el archivo de ADN y siguiendo muy de cerca al Índice FM para el archivo de proteínas.

Luego de ver los resultados se puede decir que ambas versiones del Índice FM Huffman se comportan mejor mientras más pequeña sea la entropía de orden cero del texto. En el caso del archivo de ADN, ambas resultaron ser muy competitivas, superando ambas a los demás índices para conteo de ocurrencias y reporte de posiciones.

Capítulo 2

CONCEPTOS BÁSICOS

El problema clásico de búsqueda en texto es determinar las *occ* ocurrencias de un patrón $P = p_1 p_2 \dots p_m$ en un texto $T = t_1 t_2 \dots t_n$. El texto y el patrón son secuencias de caracteres sobre un alfabeto Σ de tamaño σ. En la práctica uno desea conocer las posiciones en el texto de esas ocurrencias y también un contexto del texto a su alrededor. Usualmente se consulta en varias ocasiones con patrones diferentes, por lo que vale la pena preprocesar el texto para aumentar la velocidad de las búsquedas.

El problema de indexación de texto completo consiste en, dado un texto T, construir una estructura de datos (también llamada *índice*) que permite buscar las ocurrencias de un patrón arbitrario P como una subcadena de T. Este índice es llamado de texto completo para enfatizar el hecho de que la operación de búsquedas de subcadenas es más poderosa que la búsqueda común de una palabra, o de un prefijo de palabra, funcionalidad entregada comúnmente por los índices basados en palabras.

2.1. Árbol de sufijos

Un tipo de índice de texto completo es el árbol de sufijos [2, 3, 4]. Un *trie* de sufijos es una estructura que permite acceso a todos los sufijos de un texto. Las aristas del árbol corresponden a caracteres del texto y para todos los sufijos del texto existe un camino desde la raíz hasta una hoja que recorre los caracteres de ese sufijo. En cada una de estas hojas se almacena la posición del texto del sufijo correspondiente a ella. Un árbol de sufijos es una representación compacta del *trie* de sufijos donde todos los nodos hijos únicos se juntan con sus nodos padres.

Utilizando esta estructura, un patrón se puede buscar de la siguiente manera: se recorre el árbol desde la raíz siguiendo los caracteres del patrón hasta que:

- No existe un camino por donde seguir. En este caso el patrón no aparece en el texto.

- Se llega a una hoja. Esto significa que se encontró un prefijo del patrón y este prefijo es único en el texto. Se debe ir entonces a la posición del texto que indica la hoja y verificar la ocurrencia completa del patrón.
- Se llega al final del patrón antes de llegar a una hoja. En este caso todas las hojas del subárbol cuya raíz es el nodo alcanzado son ocurrencias del patrón. Se recorre este subárbol y se reportan las ocurrencias.

Esta estructura permite alcanzar el tiempo óptimo de búsqueda, el cual es $O(m+occ)$, dado que cada carácter de P debe ser examinado y que las *occ* ocurrencias deben ser reportadas. El espacio que ocupa esta estructura es mayor que el que ocupa el texto. En general, ocupa $O(n \log n)$ bits, cuando el texto ocupa $O(n \log \sigma)$ [1].

La figura 2.1 muestra un ejemplo del árbol de sufijos construido sobre el texto "alabar_a_la_alabarda". Si quisiéramos, por ejemplo, buscar el patrón *barde*, comenzamos recorriendo el árbol desde la raíz según los caracteres y llegaríamos a la hoja que almacena el número 16 sin que se haya terminado el patrón. Entonces, debemos ir al texto y verificar la ocurrencia completa del patrón. Como el patrón no es un prefijo del sufijo que comienza en la posición 16 del texto -difieren en la posición 16-, concluimos que el patrón no está en el texto. Si buscamos el patrón *la*, se nos acabará el patrón antes de llegar a una hoja. En este caso llegamos al nodo al cual llega la arista desde la raíz por la cadena *la*. Todas las posiciones de las hojas de este subárbol corresponden a ocurrencias de *la*, es decir, las posiciones 10, 2 y 14.

2.2. Arreglo de sufijos

Sea $T[1 \dots n] = T[1]T[2] \dots T[n]$ un texto de largo n sobre un alfabeto Σ de tamaño σ. Para cada símbolo del alfabeto se asigna un número distinto en $\{1, 2, \dots, \sigma\}$. El orden de los símbolos es definido por los números. Se asume que $T[n+1] = \$$ es un carácter terminador único cuyo número es 0. Se llama a una subcadena $T[j \dots n]$ un sufijo de T. El arreglo de sufijos $SA[1 \dots n]$ de T es un arreglo de enteros j que representa los sufijos $T[j \dots n]$. Los enteros están ordenados según el orden lexicográfico de los sufijos correspondientes. El orden lexicográfico de dos sufijos se define como:

$$T[i \dots n] < T[j \dots n] \iff T[i] < T[j] \text{ ó } (T[i] = T[j] \text{ y } T[i+1 \dots n] < T[j+1 \dots n])$$

[1] log significa $\log_2$ en este documento

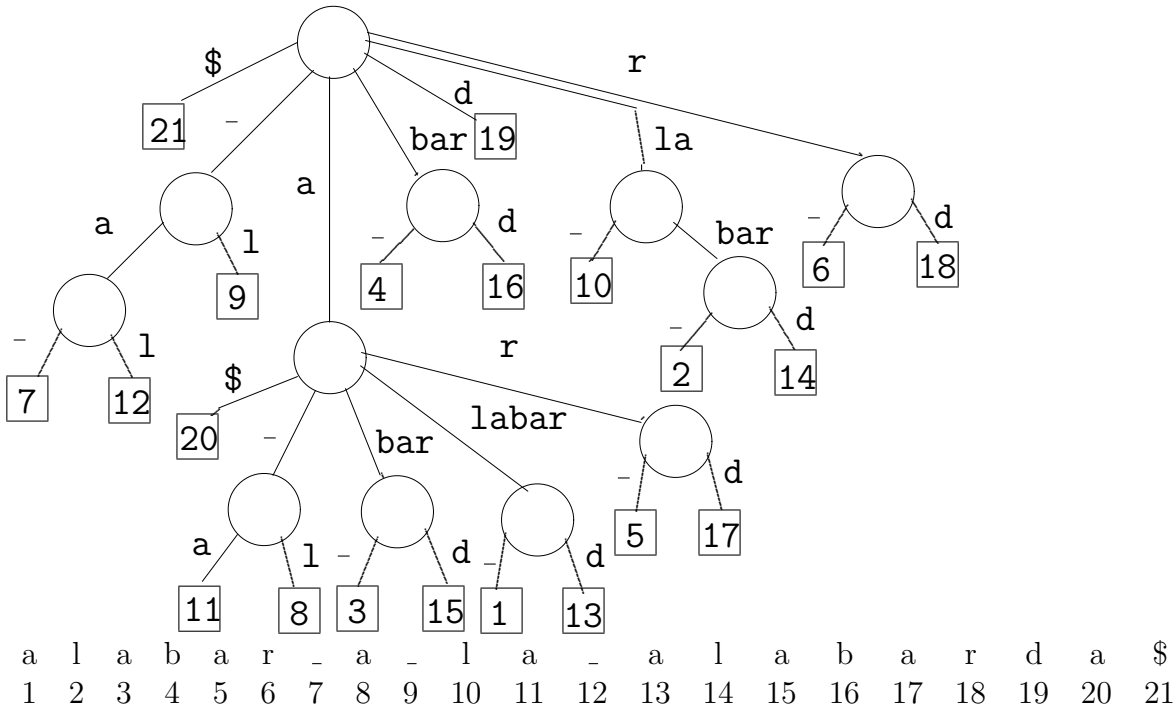


Fig. 2.1: Árbol de sufijos para el texto "alabar_a_la_alabarda". Un carácter especial \$ se agrega al final del texto para que sólo los nodos externos contengan la información de las posiciones.

Como $T[n+1] = \$$ es único, no existen dos sufijos con el mismo orden lexicográfico. Además, como $T[n+1]$ es menor que cualquier otro carácter del alfabeto, siempre se cumple que $SA[0] = n+1$.

Visto de otra manera, un arreglo de sufijos no es más que los números almacenados en las hojas del árbol de sufijos, de izquierda a derecha. Con esta estructura se pueden simular búsquedas sobre el árbol de sufijos, utilizando búsqueda binaria. Dado el arreglo de sufijos, la búsqueda de las ocurrencias del patrón $P = p_1 p_2 \dots p_n$ es trivial. Las ocurrencias forman un intervalo $[sp, ep]$ en el arreglo A tal que los sufijos $t_{A[i]} t_{A[i]+1} \dots t_n$, $sp \leq i \leq ep$, contienen el patrón como prefijo. Este intervalo se puede encontrar usando dos búsquedas binarias. El tiempo de búsqueda es $O(m \log n + occ)$ y el espacio $O(n \log n)$, con una constante menor que la del árbol de sufijos. La figura 2.2 muestra el arreglo de sufijos para el texto "alabar_a_la_alabarda". Si queremos, por ejemplo, buscar el patrón *ar* en el texto, buscamos con búsqueda binaria la posición del arreglo de sufijos que apunta al primer sufijo que es lexicográficamente mayor o igual que *ar*. Esta búsqueda daría como resultado la posición 12, la cual apunta al sufijo que comienza en la posición 5 del texto. Luego buscamos la posición del arreglo de sufijos que apunta al último sufijo del texto que es lexicográficamente mayor o igual que *ar* y que comienza con *ar*. Esta búsqueda daría como resultado la posición 13, que apunta al sufijo que comienza en la

SA	21	7	12	9	20	11	8	3	15	1	13	5	17	4	6	19	10	2	14	6	18
j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Fig. 2.2: Arreglo de sufijos para el texto "alabar_a_la_alabarda".

posición 17 del texto. Entonces, el número de ocurrencias del patrón ar en el texto es $13 - 12 + 1 = 2$ y las posiciones de las ocurrencias son 5 y 17.

2.3. La transformación de Burrows-Wheeler (BWT)

La transformación de Burrows-Wheeler [6] transforma un texto T en una nueva cadena (T^{bwt}) que contiene los mismos caracteres que T pero que usualmente es más fácil de comprimir. La BWT consiste de los siguientes tres pasos:

1. Concatenar un marcador especial \$ al final de T, el cual es lexicográficamente menor que cualquier otro carácter.
2. Formar una matriz conceptual M cuyas filas son las rotaciones cíclicas de la cadena T\$, ordenadas lexicográficamente.
3. Construir el texto transformado L tomando la última columna de M.

Notemos que cada columna de M, y por lo tanto también el texto transformado L, es una permutación de T\$. En particular, la primera columna, a la que llamaremos F, es obtenida al ordenar lexicográficamente los caracteres de T\$. La figura 2.3 muestra un ejemplo de aplicar la transformación sobre el texto "mississippi".

Cuando se ordenan las filas de M se ordenan esencialmente los sufijos de T, debido a la presencia del carácter especial \$. Por lo tanto, hay una fuerte relación entre la matriz M y el arreglo de sufijos construido sobre T pues $SA[i] = j$ si y sólo si la i-ésima fila de M contiene la cadena $t_j t_{j+1} \ldots \$ t_1 \ldots t_{j-1}$. T^{bwt} representa implícitamente al arreglo de sufijos. La matriz M presenta otras propiedades. Para describirlas presentamos la siguiente notación:

- C denota un arreglo de largo σ tal que $C[c]$ contiene el número total de ocurrencias en el texto de los caracteres que son alfabéticamente menores que c.
- $Occ(L, c, i)$ denota el número de ocurrencias del carácter c en al prefijo $L[1, i]$.

Por ejemplo en la figura 2.3 tenemos $C[s] = 8$ y $Occ(L, s, 10) = 4$. Las siguientes propiedades se han demostrado en [6]:

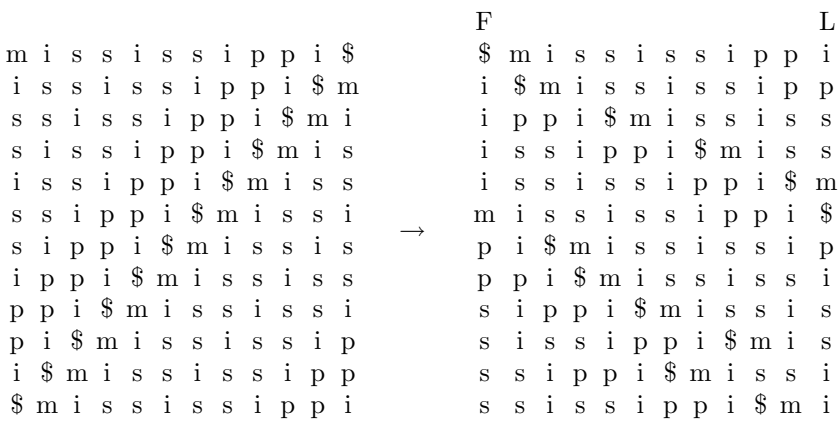


Fig. 2.3: Ejemplo de la transformación de Burrows Wheeler para la cadena "mississippi". Las filas de la matriz de la derecha están ordenadas lexicográficamente. La salida de la BWT es la columna L, en este caso la cadena "ipssm$pissii".

1. Dada la i-ésima fila de M, su último carácter $L[i]$ precede a su primer carácter $F[i]$ en el texto original T.

2. Se define $LF(i) = C[L[i]] + Occ(L, L[i], i)$. LF es el mapeo de los caracteres de la última columna a los de la primera, es decir, $L[i]$ está ubicado en F en la posición $LF(i)$. Por ejemplo en la figura 2.3 tenemos $LF(10) = C[s] + Occ(L, s, 10) = 12$ y de hecho $L[10]$ y $F[LF(1)] = F[12]$ corresponden ambos al primer carácter s en la cadena $mississipi$.

3. Si $T[k]$ es el i-ésimo carácter de L entonces $T[k-1] = L[LF(i)]$. Por ejemplo en la figura 2.3 tenemos que $T[3] = s$ el el décimo carácter de L y tenemos $T[2] = L[LF(10)] = L[12] = i$.

   La última propiedad hace posible que se pueda reconstruir T a partir de T^{bwt}. Inicialmente $i = 1$ y $T[n]$ =L[1] (pues la primera fila de M es $\$T$). Luego, para $k = n, \ldots, 2$ asignamos $T[k-1] = L[LF(i)]$ y $i = LF(i)$. Por ejemplo, para reconstruir el texto $T = mississippi$ de la figura 2.3 comenzamos asignando $T[11] = L[1] = i$. En la primera iteración $(k = 11, i = 1)$ obtenemos $T[10] = L[LF(1)] = L[C[i] + Occ(L, i, 1)] = L[2] = p$. En la segunda iteración $(k = 10, i = LF(1) = 2)$ obtenemos $T[9] = L[LF(2)] = L[C[p] + Occ(L, p, 2)] = L[7] = p$, y así sucesivamente.

2.4. Entropía de un texto

La entropía H de un texto es una medida de la aleatoriedad de sus caracteres. Sea $T[1 \ldots n]$ una cadena de caracteres sobre el alfabeto $\Sigma = \{\alpha_1, \ldots, \alpha_h\}$, y sea n_i el número

de ocurrencias del símbolo α_i en T. La entropía empírica de orden cero de la cadena T se define como[2]:

$$H_0(T) = -\sum_{i=1}^{h} \frac{n_i}{n} \log\left(\frac{n_i}{n}\right)$$

El valor $nH_0(T)$ representa el tamaño de la salida de un compresor ideal que utiliza $-\log\left(\frac{n_i}{n}\right)$ bits para codificar el símbolo α_i. Esta es la máxima compresión alcanzable si se utiliza un código únicamente decodificable en el cual una palabra de código fija es asignada a cada símbolo del alfabeto. Se puede lograr mayor compresión si el código utilizado para cada símbolo depende de los k símbolos que lo preceden. Para cualquier cadena $w \in \Sigma^k$ de largo k y $\alpha_i \in \Sigma$, $n_{w\alpha_i}$ denota el número de ocurrencias en T de la cadena w seguida de α_i, es decir, el número de ocurrencias de la cadena $w\alpha_i$ en T. Sea $n_w = \sum_i n_{w\alpha_i}$. El valor:

$$H_k(T) = -\frac{1}{n} \sum_{w \in \Sigma^k} n_w \left(\sum_{i=1}^{h} \frac{n_{w\alpha_i}}{n_w} \log\left(\frac{n_{w\alpha_i}}{n_w}\right) \right)$$

se llama la entropía empírica de orden k de la cadena T. El valor $nH_k(T)$ representa una cota inferior para el tamaño de la salida de cualquier codificador únicamente decodificable que utiliza para cada símbolo un código que depende solamente del mismo símbolo y de los k símbolos vistos más recientemente. Para cualquier cadena T y $k \geq 0$, se cumple $H_k(T) \geq H_{k+1}(T)$. Entonces, la entropía constituye una medida de una cota inferior para comprimir el texto.

2.5. Codificación de Huffman

La codificación de Huffman [23] es una técnica de compresión de datos que transforma cada símbolo de la codificación original en otro de largo variable, removiendo la redundancia. Generalmente todos los símbolos de la codificación original ocupan el mismo espacio. Por ejemplo, si se trata de un texto de caracteres, cada carácter en el texto se codifica con un byte, es decir, 8 bits. Sin embargo, no todos los caracteres aparecen la misma cantidad de veces en el texto. La idea de la codificación de Huffman es asignar a los caracteres de mayor probabilidad o frecuencia símbolos más cortos, y a los de menor frecuencia símbolos más largos. Así, se logra una codificación del texto completo que ocupa menos espacio que la codificación original.

[2] Se asume que $0 \log 0 = 0$.

El primer paso de esta codificación es crear una serie de reducciones del código original ordenando las probabilidades de los símbolos en consideración y combinando los dos símbolos de menor probabilidad en un sólo símbolo que los reemplaza en la siguiente reducción y cuya probabilidad es la suma de las probabilidades de los códigos combinados. Este proceso se repite hasta que se reduce la representación original a dos símbolos.

La figura 2.4 muestra este proceso para una codificación binaria. A la izquierda, un hipotético conjunto de símbolos y sus probabilidades se ordenan de arriba hacia abajo decrecientemente según el valor de sus probabilidades. Para formar la primera reducción, las dos probabilidades de más abajo, 0,06 y 0,04, se combinan para formar un "símbolo compuesto" con probabilidad 0,1. Este símbolo compuesto y su correspondiente probabilidad se ubican en la columna de la reducción número 1 de modo que las probabilidades de esa columna queden asimismo ordenadas de arriba hacia abajo en forma decreciente de probabilidad. Esto se repite hasta que en la última columna tenemos sólo dos símbolos.

El segundo paso en el procedimiento de Huffman es codificar cada símbolo reducido, empezando con los dos símbolos resultantes del paso anterior y yendo hacia atrás hasta los símbolos originales. Si se trata de una codificación binaria, a cada uno de estos símbolos se le asigna un 1 o un 0. Cuando un símbolo es una combinación de otros dos, se separa en los símbolos combinados y se le asigna a cada uno el prefijo de codificación del combinado y un 1 o un 0 se agregan al final para distinguir el uno del otro. Esta operación se repite para cada símbolo reducido hasta que se alcanza la codificación original.

La figura 2.5 muestra este proceso. Inicialmente se asignan los códigos 0 y 1 a los símbolos compuestos de la parte de la derecha de la tabla. Esta asignación es arbitraria, si se asignaran los códigos al revés, la codificación sería equivalente. Como el símbolo reducido que tiene probabilidad 0,6 fue generado al combinar dos símbolos en la columna de su izquierda, el 0 de su código es utilizado para codificar ambos símbolos, y un 0 y un 1 son concatenados arbitrariamente a cada uno para distinguirlos el uno del otro. Así, los símbolos transitorios de la columna 3 quedan con códigos 00 y 01. Se repite esto hasta llegar a los símbolos originales. Los códigos finales para cada símbolo aparecen a la izquierda de la tabla.

Se puede observar que el proceso de codificación de Huffman construye un árbol binario cuyas hojas son los símbolo originales y cuyos nodos internos corresponden a los símbolos compuestos. El camino desde la raíz a una hoja determina el código de Huffman para el símbolo correspondiente a esa hoja. Esto se puede ver en la figura 2.5.

Código original		Reducción			
Símbolo	Probabilidad	1	2	3	4
a_2	0,4	0,4	0,4	0,4	→ 0,6
a_6	0,3	0,3	0,3	0,3 →	0,4
a_1	0,1	0,1	→ 0,2 →	→ 0,3 →	
a_4	0,1	0,1 →	0,1 →		
a_3	0,06 →	→ 0,1 →			
a_5	0,04 →				

Fig. 2.4: Reducciones del código original. Los símbolos que tienen flechas a su derecha son los que se componen en el símbolo de la siguiente columna que tiene una flecha a su izquierda.

Código original			Reducción			
Símbolo	Probabilidad	Código	1	2	3	4
a_2	0,4	1	0,4 1	0,4 1	0,4 1	← 0,6 0
a_6	0,3	00	0,3 00	0,3 00	0,3 00←	0,4 1
a_1	0,1	011	0,1 011	← 0,2 010←	← 0,3 01←	
a_4	0,1	0100	0,1 0100←	0,1 011←		
a_3	0,06	01010←	← 0,1 0101←			
a_5	0,04	01011←				

Fig. 2.5: Procedimiento de asignación de códigos. Los símbolos que tienen flechas a su izquierda son los que se descomponen en los símbolo de la columna anterior que tienen flechas a su izquierda.

Los símbolos de la columna 4 son los hijos directos de la raíz del árbol. Cada par de símbolos resultantes de descomponer un símbolo de la siguiente columna constituye los hijos del símbolo descompuesto.

El procedimiento de Huffman crea un código óptimo para un conjunto de símbolos y probabilidades sujeto a la restricción que los símbolos son codificados de a uno. Una vez que el código ha sido creado, la codificación y decodificación se realiza utilizando una simple tabla de referencia con los códigos de cada símbolo. El código resultante es un *código de bloque*, pues cada símbolo original se mapea a una secuencia fija de símbolos de codificación. Es *instantáneo*, pues cada palabra del código en una cadena de símbolos puede ser decodificada sin hacer referencia a los siguientes símbolos. Es *únicamente decodificable* pues cualquier cadena de símbolos de la codificación puede ser decodificada de una sola manera. Para el código de la figura 2.5, una lectura de izquierda a derecha de la cadena codificada 010100111100 revela que el primer código válido es 01010, el cual corresponde al símbolo a_3. El siguiente código válido es 011, el cual corresponde al símbolo a_1. Continuando de esta manera se decodifica el mensaje completo, el cual resulta ser $a_3a_1a_2a_2a_6$.

El largo promedio de los símbolos de codificación de Huffman es muy cercano a la

entropía del texto. El largo del texto comprimido será menor que $(H_0 + 1)n$, donde H_0 es la entropía de orden cero del texto y n su largo. En el caso del código de la figura 2.5, el largo promedio del código es:

$$L_{prom} = 0{,}4 \times 1 + 0{,}3 \times 2 + 0{,}1 \times 3 + 0{,}1 \times 4 + 0{,}06 \times 5 + 0{,}04 \times 5$$

$L_{prom} = 2{,}2$ bits/símbolo, mientras que la entropía de orden cero del texto es 2,14.

La codificación de Huffman puede usar dos símbolos para asignar los códigos, como se mostró en el ejemplo, o más de dos símbolos. En general, una codificación que utiliza k símbolos se denomina una codificación Huffman k-aria. En este caso el alfabeto $\{0, 1, \ldots, k-1\}$ es utilizado para codificar los símbolos originales. En cada reducción, se toman los k símbolos de menor frecuencia y se juntan en un nuevo símbolo. El árbol de Huffman resultante es un árbol k-ario. Para que este árbol sea óptimo, la raíz debe tener k hijos. Esto no siempre ocurre si es que se toman k símbolos en cada reducción. Para solucionar esto, en la primera reducción se toma un número de símbolos que asegure que la raíz tendrá k hijos.

Sea k' el número de símbolos que se toman en la primera reducción y σ el número total de símbolos. Suponemos que $\sigma \geq k$. En total debemos reducir σ símbolos. Si en la primera reducción tomamos k' símbolos para formar un símbolo compuesto, hemos reducido el número total de símbolos en $k'-1$. Luego, hacemos un número u de reducciones tomando k símbolos, lo que implica que el se tienen $u(k-1)$ símbolos menos. En la última reducción tomamos k símbolos, restando k símbolos al total. Entonces, se debe cumplir:

$$k' - 1 + u(k-1) + k = \sigma$$

es decir,

$$\sigma - k = u(k-1) + k' - 1$$

y por lo tanto,

$$k' = 1 + ((\sigma - k) \mod (k-1))$$

Esto se cumple si es que $((\sigma - k) \mod (k-1))$ es distinto de cero. Cuando $((\sigma - k) \mod (k-1)) = 0$, $k' = k$. Tomando k' símbolos en la primera reducción, el árbol de Huffman resulta óptimo.

2.6. Funciones Rank y SelectNext

Las funciones Rank y SelectNext son funciones que se calculan sobre arreglos de bits y que se utilizan en la implementación del Índice FM Huffman, así como también en otras implementaciones. Dada una secuencia de bits $B[1 \dots n]$, $rank(B, i)$ es el número de 1's en $B[1 \dots i]$. Se define $rank(B, 0) = 0$. La función inversa de $rank$, $select(B, j)$, entrega la posición del j-ésimo bit encendido en B. Ambas funciones se pueden calcular en tiempo constante utilizando estructuras de datos que almacenan cierta información sobre los arreglos de bits sobre los cuales aplicamos las funciones, las cuales ocupan $o(n)$ bits, donde n es el tamaño de los arreglos [24]. En esta sección describimos estas estructuras y la manera en que nos permiten calcular estas funciones. Para el Índice FM Huffman en vez de utilizar la función *select* se utiliza una versión más simple llamada *selectnext*, cuya implementación resulta más práctica que la de *select*.

2.6.1. La función Rank

Sea B el arreglo de bits de largo n sobre la cual queremos calcular $rank$. La estructura que permite calcular esta función en tiempo constante consiste de tres arreglos que almacenan la cantidad de unos en algunas partes del arreglo completo. Así, el cálculo de $rank$ para cualquier posición de B se calcula sumando los valores de estos tres arreglos.

Se divide el arreglo B en bloques de largo $b = \lfloor \log(n)/2 \rfloor$. A su vez, bloques consecutivos se agrupan en superbloques de largo $s = b\lfloor \log n \rfloor$.

El arreglo R_s almacena, para cada superbloque, el número de bits encendidos hasta antes del comienzo del superbloque. Es decir, para cada superbloque $j, j = 0 \dots \lfloor n/s \rfloor$, $Rs[j] = rank(B, j \cdot s)$. El arreglo R_s necesita en total $O(n \log n)$ pues cada entrada necesita $\log n$ bits y se tienen $n/s = O(n/\log^2 n)$ entradas.

El arreglo R_b almacena, para cada bloque, el número de bits encendidos en el superbloque en el que está contenido hasta antes del comienzo del bloque. Es decir, para cada bloque k contenido en el superbloque $j = k \ div \ \lfloor \log n \rfloor$, $k = 0 \dots \lfloor n/b \rfloor$, $Rb[k] = rank(B, k \cdot b) - rank(B, j \cdot s)$. El arreglo R_b necesita $O(n \log \log n / \log n)$ bits pues cada entrada necesita $O(\log \log n)$ bits, dado que representa el número de bits encendidos en un superbloque de largo $O(\log^2 n)$, y en total hay $n/b = O(n \log n)$ bloques.

Por último, el arreglo Rp almacena, para cada posible cadena de bits S de largo b, y para cada posición i dentro de S, el número de bits encendidos en S hasta i. Es decir, $R_p[S, i] = rank(S, i)$. Este arreglo bidimensional necesita $O(2^b \cdot b \cdot \log b) = O(\sqrt{n} \log n \log \log n)$ bits.

La estructura completa necesita $O(n/\log n + n \log\log n/\log n + \sqrt{n}\log n \log\log n) = o(n)$ bits. La función *rank* se calcula entonces como:

$$rank(B,i) = R_s[i \ div \ s] + R_b[i \ div \ b] + R_p[B[(i \ div \ b)\cdot b + 1 \ldots (i \ div \ b)\cdot b + b], i \ mod \ b]$$

2.6.2. La función SelectNext

La función *select* también puede ser implementada en tiempo constante utilizando $o(n)$ bits adicionales [24]. Sin embargo, es más complicada de implementar y menos práctica que rank.

Para el Índice FM Huffman se utiliza una versión más simple y práctica de *select*, la cual está restringida a encontrar el siguiente bit encendido en B partiendo de cierta posición. Dada una secuencia de bits B, de largo n, $selectnext(B,j)$ es la posición del primer bit encendido en $B[j \ldots n]$, y $n+1$ si es que no existe tal posición.

Al igual que para el cálculo de *rank*, se divide B en bloques y superbloques de largo b y s, respectivamente.

El arreglo N_s almacena, para cada superbloque, la posición del primer bit encendido a partir del comienzo del superbloque. Es decir, para cada superbloque $j, j = 1 \ldots \lfloor n/s \rfloor$, $N_s[j] = selectnext(B, j\cdot s + 1)$. El arreglo N_s necesita en total $O(n/\log n)$ bits dado que cada entrada necesita $O(\log n)$ bits.

El arreglo N_b almacena, para cada bloque k contenido en el superbloque $j = k \ div \ \lfloor \log n \rfloor$, $k = 1 \ldots \lfloor n/b \rfloor$, la posición del primer bit encendido en el superbloque j a partir del comienzo de ese bloque, relativa al superbloque, o $s+1$ si es que esa posición no existe. Es decir $N_b[k] = selectnext(B[j\cdot s + 1 \ldots (j+1)\cdot s], k\cdot b - j\cdot s + 1)$. El arreglo N_b necesita en total $O(n \log\log n/\log n)$ bits pues cada entrada necesita $O(\log\log n)$, ya que representa una posición dentro de un superbloque de largo $O(\log^2 n)$.

Por último, al igual que para la función *rank*, se construye un arreglo bidimensional que almacena, para cada posible cadena de bits S de largo b, y para cada posición i dentro de ella, la posición del primer bit encendido en S a partir de i, o $b+1$ si tal posición no existe. Es decir, $N_p[S,i] = selectnext(S,i)$. Este arreglo necesita $O(2^n \cdot b \cdot \log b) = O(\sqrt{n}\log n \log\log n)$ bits.

Tal como antes, la estructura completa ocupa $o(n)$ bits. Con ella se puede calcular $selectnext(B,i)$ en tiempo $O(1)$ de la siguiente manera:

1. Se calcula $i_b = (i \ div \ b)\cdot b$ y luego $pos = N_p[B[i_b + 1 \ldots i_b + b], i - i_b + 1]$. Si

$pos \leq b$, entonces hay un bit encendido en $B[i \ldots i_b + b - 1]$ y simplemente se retorna $i_b + pos$.

2. De otra manera, $selectnext(B, i) = selectnext(B, i_b + b)$, por lo que tenemos que encontrar la respuesta correspondiente al siguiente bloque. Se calulca $pos = N_b[(i \ div \ b) + 1]$. Si $pos \leq s$, entonces se retorna $((i_b + b) \ div \ s) \cdot s + pos)$.

3. De otra manera, hay sólo ceros en $B[i \ldots i_s + s - 1]$, donde $i_s = (i \ div \ s) \cdot s$, por lo que $selectnext(B, i) = selectnext(B, i_s + s)$. Se retorna $N_s[i \ div \ s + 1]$.

Capítulo 3

TRABAJO RELACIONADO

A continuación se describe el trabajo existente en el área de índices de texto completo que es relevante para esta memoria. En lo que sigue, n es el número de caracteres de un texto T sobre un alfabeto Σ de tamaño σ.

3.1. Índice FM

El Índice FM [7, 8] es una estructura de datos que explota la relación entre el arreglo de sufijos [5] y el algoritmo de transformación de Burrows Wheeler [6]. Esta estructura ocupa $5nH_k(T) + o(n)$ bits de espacio, donde $H_k(T)$ la entropía de orden $k-$ésimo de T, y es capaz de contar las *occ* ocurrencias de un patrón P de largo m en tiempo $O(m)$ y de reportar sus posiciones en tiempo $O(occ \log^{1+\epsilon} n)$, donde $\epsilon > 0$ es una constante arbitraria elegida al momento de construir la estructura. El tiempo que toma mostrar una subcadena de largo L del texto es $O(L + \log^{1+\epsilon} n)$. Dado que la entropía empírica $H_k(T)$ es a lo más $\log \sigma$, en el peor caso esta estructura ocupa $\Theta(n)$ bits. Sin embargo, si el texto T es compresible, la estructura puede alcanzar espacio $o(n)$. Estas complejidades son válidas mientras el tamaño del alfabeto σ sea constante.

La idea principal de este índice es agregar información adicional al texto comprimido, obtenido utilizando la transformación de Burrows Wheeler, para convertirlo en un índice de texto completo que permite la búsqueda de patrones arbitrarios eficientemente.

3.1.1. Compresión basada en la transformación Burrows Wheeler

La salida de la BWT (ver sección 2.3) es la última columna L de la matriz M ($L = bwt(T)$). La BWT no es un algoritmo de compresión por sí mismo pues L sólo es una permutación de $T\$$. Sin embargo, si T cumple ciertas regularidades, L contará con secuencias de caracteres idénticos, lo que resulta ser altamente compresible.

Varios algoritmos de compresión se basan en la BWT. En este índice se utiliza un algoritmo llamado BW_RLX [10], el cual procesa el string L de la siguiente manera:

1. Se utiliza un codificador "move-to-front"[26] (mover al frente, abreviado mtf), para codificar cada carácter de L con el número de caracteres distintos vistos desde su ocurrencia previa. El codificador mantiene una lista, llamada la lista-MTF, inicializada con todos los caracteres de Σ en orden alfabético. Al procesar un nuevo carácter, se agrega a la salida del codificador la posición del carácter y éste se mueve al principio de la lista. En cada instante, la lista está ordenada desde el carácter visto más recientemente al que no ha sido visto hace más pasos. El resultado resultante es la cadena $L^{mtf} = mtf(L)$, la cual es definida sobre el alfabeto $\{0, 1, 2, \ldots, \sigma - 1\}$.

2. Se codifica cada secuencia de ceros en L^{mtf} usando un codificador "run-length" (abreviado rle). Esto consiste en reemplazar cada secuencia 0^m con el número $(m+1)$ escrito en binario, con el primer bit como el de menor significancia y descartando el bit de mayor significancia. Para esta codificación se utilizan dos nuevos símbolos **0** y **1**. $L^{rle} = rle(L^{mtf})$ se define sobre el alfabeto $\{\mathbf{0}, \mathbf{1}, 1, 2, \ldots, \sigma - 1\}$.

3. Se comprime L^{rle} por medio de un código de prefijo de largo variable (abreviado PC), definido de la siguiente manera: los símbolos **0** y **1** son codificados usando dos bits (10 para **0**, 11 para **1**). Para $i = 1, 2, \ldots, \sigma - 1$, el símbolo i se codifica usando $1 + 2\lfloor \log(i+1) \rfloor$ bits: $\lfloor \log(i+1) \rfloor$ 0's seguidos por la representación binaria de $i + 1$, la cual utiliza $1 + \lfloor \log(i+1) \rfloor$ bits. La cadena $PC(L^{rle})$, la cual es la salida del algoritmo BW_RLX, está definida sobre el alfabeto $\{0, 1\}$.

3.1.2. Contando el número de ocurrencias

El algoritmo para contar las ocurrencias de un patrón P en el texto consta de dos componentes básicas: la cadena $Z = BW\_RLX$ y una estructura auxiliar para el cálculo en tiempo constante del valor de $Occ(L, c, q)$ definido como el número de ocurrencias del carácter c en L hasta la posición q.

En la búsqueda se utilizan dos propiedades de la matriz M correspondiente a la BWT: (i) todos los sufijos de T que tienen a P como prefijo ocupan un conjunto contiguo de filas de M y (ii) este conjunto de filas comienza con la fila $First$ y termina con la fila $Last$, donde $First$ es la posición lexicográfica de P en el conjunto ordenado de filas de M. El valor $(Last - First + 1)$ es el número total de ocurrencias del patrón.

La recuperación de las filas $First$ y $Last$ funciona en m fases, numeradas desde m hasta 1. En la fase i-ésima el parámetro $First$ apunta a la primera fila de M que tiene a $P[i,m]$ como prefijo y el parámetro $Last$ apunta a la última fila de M que tiene a $P[i,m]$ como prefijo. En la última fase, $First$ y $Last$ delimitarán las filas de M que contienen todos los sufijos del texto que tienen a P como prefijo. La figura 3.1 muestra el algoritmo de conteo de ocurrencias en pseudocódigo.

```
Algoritmo get_rows(P[1,m])
1.  i ← m, First ← 1, Last ← n
2.  while (First ≤ Last and i ≥ 1) do
3.      c ← P[i]
4.      First ← C[c] + Occ(L, c, First − 1) + 1
5.      Last ← C[c] + Occ(L, c, Last)
6.      i ← i − 1
7.  if Last < First then return "No hay filas con prefijo P[1,m]"
8.  else return (First, Last)
```

Fig. 3.1: Pseudocódigo del algoritmo que encuentra las filas de M que tienen a $P = p_1 \dots p_m$ como prefijo de T. $C[c]$ es el número de caracteres que son lexicográficamente menores que c y $Occ(L, c, q)$ es el número de ocurrencias del carácter en $L[1, q]$.

El tiempo que demora *get_rows* está determinado principalmente por el tiempo que toma realizar $2m$ computaciones de la función Occ. La función $Occ(L, c, q)$ puede calcularse en tiempo $O(1)$, por lo que la búsqueda se realiza en tiempo $O(m)$. La estructura auxiliar que permite esto se llamará $Opp(T)$, y se basa en la idea de particionar L en subcadenas de $\ell = \Theta(\log n)$ caracteres[1], llamados *buckets*, que se denotan como $BL_i = L[(i-1)\ell + 1, i\ell]$, para $i = 1, \dots, n/\ell$. Esta partición induce a una partición de L^{mtf} en n/ℓ *buckets* $BL_1^{mtf}, \dots, BL_{n/\ell}^{mtf}$, también de tamaño ℓ. Para explicar cómo calcular $Occ(L, c, q)$ supondremos que cada corrida de ceros en L^{mtf} está contenida en un sólo *bucket*[2]. Bajo esta suposición, los *buckets* BL_i^{mtf} inducen a una partición del archivo comprimido Z en n/ℓ *buckets comprimidos* de largo variable $BZ_1, \dots, BZ_{n/\ell}$, definidos como $BZ_i = PC(rle(BL_i^{mtf}))$.

El cálculo de $Occ(L, c, q)$ se basa en una descomposición de $L[1, q]$ en tres subcadenas definidas como:

1. el prefijo más largo de $L[1, q]$ que tiene largo múltiplo de ℓ^2.

2. el prefijo más largo del sufijo restante que tiene largo múltiplo de ℓ.

[1] Por simplicidad se supone que n es un múltiplo de ℓ^2.
[2] El caso general en el cual una secuencia de ceros puede cubrir varios *buckets* se discute en [9].

3. el sufijo restante de $L[1,q]$ que es un prefijo del *bucket* BL_i, donde $i = \lceil q/\ell \rceil$. Nótese que BL_i es el *bucket* que contiene al carácter $L[q]$.

Se calcula $Occ(L,c,q)$ sumando el número de ocurrencias de c en cada uno de estas tres subcadenas. El cálculo del número de ocurrencias de c en cada subcadena puede hacerse en tiempo constante utilizando estructuras auxiliares consistentes en arreglos con valores precalculados. Si Z es la salida del algoritmso BW_RLX con entrada $T[1,n]$, el valor $Occ(L,c,q)$ puede ser calculado en tiempo $O(1)$ utilizando $|Z|+O(n \log\log n/\log n)$ bits de espacio. Por lo tanto, el algoritmo *get_rows* calcula el número de ocurrencias de $P[1,m]$ en $T[1,n]$ en tiempo $O(m)$ utilizando un espacio acotado por $5nH_k(T) + O(n \log\log n/\log n)$ bits, para cualquier $k \geq 0$.

El análisis de tiempo para este índice supone que el tamaño del alfabeto es constante. Algunos de los arreglos auxiliares que componen $Opp(T)$ dependen del tamaño del alfabeto. Si se deja de lado esta suposición, el espacio requerido por el índice completo es $5nH_k(T) + O((\sigma \log \sigma) + \log\log n)\frac{n}{\log n} + n^{\gamma}\sigma^{\sigma+1})$, donde $0 < \gamma < 1$. El tiempo requerido para la búsqueda se mantiene igual.

3.1.3. Encontrando las posiciones de las ocurrencias

Suponiendo que ya se tiene el rango $(First, Last)$ de las filas de la matriz M que tienen a P como prefijo, el problema ahora consiste en recuperar las posiciones en T de las $(Last - First + 1)$ ocurrencias del patrón. Es decir, para cada $i = First, First + 1, \ldots, Last$ queremos encontrar la posición $Pos(i)$ en T del sufijo que es prefijo de la i-ésima fila de M. Dada una fila i, se puede calcular el índice j tal tal que $Pos(j) = Pos(i) - 1$ como $j = LF[i] = C[L[i]] + Occ(L, L[i], i)$. Esto nos lleva de la fila cuyo prefijo es $T[Pos(i), n]$ a la fila cuyo prefijo es $T[Pos(i) - 1, n]$. El carácter $L[i]$ no está disponible ya que el texto está comprimido, sin embargo, se puede determinar calculando $Occ(L,c,i)$ y $Occ(L,c,i-1)$ para todo $c \in \Sigma \cup \{\$\}$. Claramente estos dos valores difieren sólo para el carácter $c = L[i]$. La estructura auxiliar de Occ, permite calcular esta función en tiempo $O(1)$, por lo que el paso $Pos(j) = Pos(i) - 1$, llamado *backward_step*, se calcula en tiempo constante si es que se supone que el tamaño del alfabeto es constante.

Dado un índice i, para calcular $Pos(i)$ se supone que para ciertas filas marcadas de M se almacenan las correspondientes posiciones del texto. Entonces, $Pos(i)$ se calcula de la siguiente manera: Si $M[i]$ está marcada, entonces su posición está almacenada y puede determinarse en tiempo constante. De lo contrario, se usa el algoritmo *backward_step* para encontrar la fila i' tal que $Pos(i') = Pos(i) - 1$. Se continúa este proceso hasta que la fila $M[i']$ está marcada. Si se hicieron t iteraciones, se retorna $Pos(i) = Pos(i') + t$.

```
Algoritmo get_position(i)
1.  i' ← i, t ← 0
2.  while la fila i' no está marcada do
3.      i' ← backward_step(i')
4.      t ← t + 1
5.  return Pos(i') + t
```

Fig. 3.2: Pseudocódigo del algoritmo que retorna la posición en T del prefijo de la fila i-ésima de M.

La figura 3.2 muestra el algoritmo que encuentra la posición en T correspondiente a la fila i.

Para marcar las posiciones se elige un parámetro ϵ tal que se marcan una posición cada $\eta = \lceil \log^{1+\epsilon} n \rceil$. La estructura S almacena las posiciones marcadas de M. Esta estructura se diseña de tal manera que ocupe un espacio pequeño y que permita contestar si es que un elemento está o no en tiempo constante, utilizando un esquema de particiones al igual que la estructura $Opp(T)$. La estructura S ocupa $O((n/\eta)(\log\log n + \log n))$ bits. En total, el algoritmo *get_positions* ocupa $O(n/\log^{\epsilon} n)$ bits para almacenar S más el espacio que ocupa la estructura $Opp(T)$.

Con respecto al tiempo, una llamada a *get_position* requiere a lo más η iteraciones, cada una de las cuales toma tiempo constante. Por lo tanto, las *occ* ocurrencias de un patrón P en T pueden recuperarse en tiempo $O(occ\,\eta) = O(occ \log^{1+\epsilon} n)$. Si consideramos el tiempo necesario para buscar P en T, el tiempo total para recuperar las ocurrencias es $O(m + occ \log^{1+\epsilon} n)$ y el espacio total está acotado por $5nH_k(T) + O(n/\log^{\epsilon} n)$, para cualquier $k \geq 0$, suponiendo σ constante. Si es que el tamaño del alfabeto no es constante, el tiempo para reportar las occurrencias se ve modificado. Antes mencionamos que para calcular $L[i]$ calculábamos $Occ(L, c, i)$ y $Occ(L, c, i-1)$ para todo $c \in \Sigma \cup \{\$\}$. Si dejamos de suponer que σ es constante, este cálculo ya no toma tiempo constante sino que toma tiempo $O(\sigma)$. Por lo tanto, el tiempo de reportar las *occ* ocurrencias es ahora $O(occ\ \sigma \log^{1+\epsilon} n)$.

3.1.4. Compromisos para arreglos de sufijos sucintos

En este capítulo se presentan dos estructuras [19, 15] que contribuyen a un compromiso entre espacio y tiempo del Índice FM. Estas estructuras modifican una función interna en la búsqueda hacia atrás que realiza este índice. La primera de estas estructuras mantiene los tiempos de búsqueda, reporte de posiciones y muestra de texto del

Estructura	Espacio
Índice FM	$5H_k n + O((\sigma \log \sigma + \log\log n)\frac{n}{\log n} + n^\gamma \sigma^{\sigma+1})$
Arreglos de Bits	$nH_0(1 + o(n))$
Jerarquía	$nH_0(1 + o(n))$

Estructura	Búsqueda	Posiciones	Muestra de texto
Índice FM	$O(m)$	$O(occ\ \sigma \log^{1+\epsilon} n)$	$O(\sigma(L + \log^{1+\epsilon} n))$
Arreglos de Bits	$O(m)$	$O(\frac{1}{\epsilon} occ\ \sigma \log n)$	$O(\frac{1}{\epsilon}\sigma(L + \log n))$
Jerarquía (promedio)	$O(m(H_0 + 1))$	$O(\frac{1}{\epsilon} occ\ H_0 \log n)$	$O(\frac{1}{\epsilon} H_0(L + \log n))$
Jerarquía (peor caso)	$O(m \log \sigma)$	$O(\frac{1}{\epsilon} occ \log \sigma \log n)$	$O(\frac{1}{\epsilon} \log \sigma(L + \log n))$

Tab. 3.1: Comparación de las complejidades de espacio y tiempo de diferentes arreglos de sufijos sucintos. Las constantes $0 < \gamma < 1$ y $\epsilon > 0$ puden reducirse si se incrementan los términos sublineales escondidos en las complejidades de tiempo. H_k es la entropía de orden k del texto, σ es el tamaño del alfabeto, *occ* es el número de ocurrencias a reportar y L es el largo de la cadena a mostrar.

Índice FM, pero tiene la ventaja de que el espacio que ocupa no depende del tamaño del alfabeto. La segunda estructura alcanza mejores tiempos para reportar posiciones y mostrar una subcadena del texto que el Índice FM. La tabla 3.1 muestra los requerimientos de espacio y tiempo del Índice FM, y de las dos estructuras propuestas.

Reemplazando las estructuras de Occ por arreglos de bits

Recordemos que $Occ(X, c, i)$ es la función que entrega el número de ocurrencias del carácter c en el prefijo $X[1, i]$. La idea descrita en [19] es reemplazar la implementación de Occ del Índice FM con el fin de tener una cadena de bits B_c por cada carácter c, tal que $B_c[i] = 1$ si y sólo si $T^{bwt}[i] = c$. Entonces, $Occ(T^{bwt}, c, i) = rank(B_c, i)$, donde $rank(B, i)$ es el número de bits encendidos en $B[1 \ldots i]$, lo cual puede ser calculado en tiempo constante utilizando $o(n)$ bits extra [24]. Esto significa que T^{bwt} ya no se necesita, junto con su estructuras de acceso, pues éstas sólo se requerían para el cálculo de Occ. El término $O(n^\gamma \sigma^{\sigma+1})$ del espacio ocupado por el Índice FM proviene de las estructuras para acceso rápido a T^{bwt}, mientras que la constante multiplicativa $O((\sigma \log \sigma)/\log n)$ aparece en la representación de Occ. Ambos términos desaparecen al remover Occ y T^{bwt}. Las nuevas estructuras que reemplazan a estas últimas ocupan un espacio acotado por $nH_0 + O(n)$.

Jerarquía de arreglos de bits

La segunda estructura constituye un esquema jerárquico para codificar las cadenas de bits de los caracteres del alfabeto. Esta estructura ha sido utilizada recientemente para otro índice bajo el nombre de *wavelet tree* [15]. La principal ventaja de esta estructura

recae en su eficiencia para reportar las posiciones de las ocurrencias y para mostrar subcadenas del texto.

La estructura consiste en un árbol binario perfectamente balanceado donde cada nodo corresponde a un subconjunto del alfabeto. Los hijos del nodo particionan el subconjunto del nodo en dos subconjuntos. Una cadena de bits en el nodo indica a qué hijo pertenece cada posición del texto.

La primera partición consistirá en ubicar los caracteres $c_1 \dots c_{\lfloor \sigma/2 \rfloor}$ en el hijo izquierdo, y los caracteres $c_{\lfloor \sigma/2 \rfloor + 1} \dots c_\sigma$ en el hijo derecho. Una cadena de bits B de largo n en la raíz tendrá un 1 en la posición i si y sólo si el carácter i-ésimo pertenece al hijo derecho. Los dos hijos de la raíz son procesados recursivamente. Sin embargo, para cada uno de ellos se consideran solamente las posiciones del texto de los caracteres que pertenecen a su subconjunto. Es decir, la cadena de bits del hijo izquierdo de la raíz tendrá solamente $l_1 + \ldots + l_{\lfloor \sigma/2 \rfloor}$ bits y la del hijo derecho sólo $l_{\lfloor \sigma/2 \rfloor + 1} + \ldots + l_\sigma$ bits, donde l_c es el número de veces que aparece c en el texto.

Este árbol tiene altura $\lceil \log \sigma \rceil$ y cada posición del texto es considerada una sola vez en cada nivel. Por lo tanto la estructura necesita a lo más $n \log \sigma$ bits. Además, para cada cadena de bits se calculan las estructuras de datos sublineales que permiten calcular *rank* sobre ellas. La función *rank* sobre un arreglo de bits entrega el número de bits encendidos hasta una cierta posición del arreglo.

El valor de $Occ(T^{bwt}, c, i)$ puede calcularse recorriendo al árbol hasta encontrar la hoja correspondiente a c. Los valores de *rank* en cada nivel permiten posicionarse en el lugar correcto en los hijos.

Esta estructura necesita $n \log \sigma (1 + o(1))$ bits para representar Occ y puede tener acceso a ella en tiempo $O(\log \sigma)$, por lo que se puede buscar en tiempo $O(m \log \sigma)$.

El tiempo de búsqueda puede ser mejorado en el caso promedio reemplazando el árbol balanceado por un árbol de *Huffman* y utilizando este árbol para particionar el alfabeto. El cálculo de Occ se realiza de igual manera que en el árbol balanceado. El número de bits que necesita esta estructura es $n(H_0 + 1)$ y el tiempo de búsqueda se convierte en $O(m(H_0 + 1))$ en promedio, el cual es a lo más $O(m \log \sigma)$, suponiendo que el patrón y el texto siguen una distribución similar. En el peor caso una hoja estará ubicada a una profundidad $O(\log n)$, por lo que la búsqueda en este caso tomaría tiempo $O(m \log n)$.

Incluso, modificando el árbol de Huffman se puede asegurar un tiempo búsqueda de $O(m \log \sigma)$ en el peor caso, manteniendo el tiempo promedio de $O(m(H_0 + 1))$ [21]. Esto se logra manteniendo la estructura del árbol hasta una profundidad $t \log \sigma$, donde t es una constante arbitraria. Todos los subárboles a esa profundidad se convierten en subárboles perfectamente balanceados. La profundidad del árbol ahora es a lo más

$t \log \sigma + \log \sigma = (t+1) \log \sigma$, y por lo tanto la búsqueda en el peor caso toma tiempo $O(m \log \sigma)$.

Reportando las ocurrencias y mostrando el texto. Para reportar las posiciones de las ocurrencias y mostrar subcadenas del texto se utiliza el mismo mecanismo que en el Índice FM original (ver sección 3.1). En el índice original, se utiliza un tiempo $O(\sigma)$ para encontrar el carácter c para el cual se cumple que $T^{bwt}[i] = c$, es decir, $Occ(T^{bwt}, c, i) = Occ(T^{bwt}, c, i-1) + 1$. Esto puede hacerse en tiempo $O(\log \sigma)$ gracias a la estructura jerárquica. Sea B la cadena de bits del nodo raíz del árbol. Si $B[i] = 0$, entonces $T^{bwt}[i] \in \{c_1 \dots c_{\lfloor c/2 \rfloor}\}$, si no, $T^{bwt}[i]$ pertenece a la mitad superior. En el primer caso, se sigue por el hijo izquierdo, en el segundo, por el hijo derecho. Se sigue buscando hasta encontrar el la hoja correspondiente a $T^{bwt}[i]$. Si se usa un árbol de Huffman en vez de un árbol balanceado, entonces el tiempo es $O(H_0 + 1)$ si es que se buscan posiciones aleatorias del texto, y todavía se puede limitar el peor caso a $O(\log \sigma)$.

Entonces, utilizando ϵn bits adicionales, se puede reportar las posiciones de *occ* ocurrencias en tiempo $O(\frac{1}{\epsilon} occ \log \sigma \log n)$ y mostrar un contexto del texto en tiempo $O(\log \sigma (L + \frac{1}{\epsilon} \log n))$.

En esta sección se mostraron dos estructuras que contribuyen a mejorar los compromisos de espacio y tiempo que aparecen en la implementación de la búsqueda del Índice FM, reduciendo la dependencia del tamaño del alfabeto tanto en la complejidad de espacio como de tiempo. La primera estructura mantiene las complejidades de tiempo del Índice FM y remueve completamente la dependencia del tamaño del alfabeto, sin embargo, su tamaño es proporcional a la entropía de orden cero del texto en vez de la entropía orden k que alcanza el Índice FM. La segunda estructura alcanza tiempos de búsqueda un poco mayores que el Índice FM original, pero alcanza mejores tiempos para reportar las posiciones y mostrar subcadenas del texto.

3.1.5. Combinando codificación run-length *con búsqueda reversa*

En esta sección se describe una estructura llamada *Índice RLFM* [22] que requiere $n(H_k \log \sigma + 1 + o(1))$ bits de espacio. Cuenta las ocurrencias de un patrón en el mismo tiempo $O(m)$ logrado por el Índice FM. La complejidad espacial resulta ser mejor mientras $\sigma \log \sigma > \log n$, es decir, este índice ocupa menos espacio excepto para alfabetos muy pequeños. La tabla 3.2 muestra los requerimientos de espacio y tiempo de esta nueva estructura.

La propiedad esencial de la transformación BWT, que la hace apropiada para la com-

Estructura	Espacio
Índice FM	$5H_k n + O((\sigma \log \sigma + \log\log n)\frac{n}{\log n} + n^\gamma \sigma^{\sigma+1}$
RLFM 1	$nH_k H + O(n)$

Estructura	Búsqueda	Posiciones	Muestra de texto
Índice FM	$O(m)$	$O(occ\ \sigma \log^{1+\epsilon} n)$	$O(\sigma(L + \log^{1+\epsilon} n))$
RLFM 1	$O(m)$	$O(\frac{1}{\epsilon} occ\ \sigma \log n)$	$O(\frac{1}{\epsilon}\sigma(L + \log n))$
RLFM 2 (peor caso)	$O(m \log \sigma)$	$O(\frac{1}{\epsilon} occ \log \sigma \log n)$	$O(\frac{1}{\epsilon} \log \sigma(L + \log n))$

Tab. 3.2: Comparación de las complejidades de espacio y tiempo de diferentes arreglos de sufijos sucintos. Las constantes $0 < \gamma < 1$ y $\epsilon > 0$ pueden reducirse si se incrementan los términos sublineales escondidos en las complejidades de tiempo. H_k es la entropía de orden k del texto, σ es el tamaño del alfabeto, *occ* es el número de ocurrencias a reportar y L es el largo de la cadena a mostrar. $H \leq \log \sigma$ es la entropía de una estructura que se describe más adelante.

presión, es que sólo algunos caracteres diferentes aparecen en contextos locales del texto transformado. En particular, tienden a aparecer corridas de ocurrencias consecutivas del mismo carácter. La idea es explotar la compresión *run-length* para representar T^{bwt}. Un arreglo S tendrá un carácter por corrida en T^{bwt}, mientras un arreglo B tendrá n bits y marcará el inicio de cada corrida. Si $T^{bwt} = c_1^{l_1} \dots c_{n'}^{l_{n'}}$ consiste de n' corridas tal que la corrida i-ésima contiene l_i repeticiones del carácter c_i, la representación de T^{bwt} consiste en la cadena $S = c_1 \dots c_{n'}$, de largo n' y un arreglo de bits $B = 10^{l_1-1} \dots 10^{l_{n'}-1}$.

Claramente, S y B contienen información suficiente para reconstruir T^{bwt} calculando $T^{bwt}[i] = S[rank(B, i)]$. La función *rank* puede ser calculada en tiempo constante utilizando $o(n)$ extra bits [24]. Por lo tanto, S y B entregan una representación de T^{bwt} que permite accesar cualquier carácter en tiempo constante y requiere a lo más $n' \log \sigma + n + o(n)$ bits.

A su vez, se necesita calcular $C_T[c] + Occ(T^{bwt}, c, i)$[3] para cualquier c e i. Para esto se utiliza un arreglo de bits B', obtenido al reordenar las corridas de B en orden lexicográfico de los caracteres de cada corrida. Las corridas del mismo carácter se dejan en su orden original. El uso de B' agrega $n + o(n)$ bits a la estructura. También se utiliza C_S, el que juega el mismo rol que C_T pero se refiere a S. Si $p_1 \dots p_{n'}$ es una permutación de $1 \dots n'$ tal que $\forall\ 1 \leq i < n'$ se cumple que $c_{p_i} < c_{p_{i+1}}$ ó $(c_{p_i} = c_{p_{i+1}}$ y $p_i < p_{i+1})$, entonces B' se define como $B' = 10^{l_{p_1}-1} \dots 10^{l_{p_{n'}}-1}$.

Las siguientes propiedades llevan al resultado de cómo calcular $C_T[c] + Occ(T^{bwt}, c, i)$. En los cálculos se utiliza la función inversa de *rank*: $select(B', j)$ es la posición del j-ésimo 1 en B'. Esta función también puede ser calculada en tiempo constante utilizando

[3] Recordemos del capítulo 2.3 que una entrada $C[c]$ del arreglo C almacena el número de ocurrencias de texto T de los caracteres alfabéticamente menores que c. Anotamos C_T para indicar explícitamente que C se refiere a T.

$o(n)$ extra bits [24].

1. Sean S y B' definidos para un string T^{bwt}. Entonces, para cada $c \in \Sigma$ se cumple

$$C_T[c] + 1 = select(B', C_S[c] + 1)$$

2. Sean S, B y B' definidos para un string T^{bwt}. Entonces, para cada $c \in \Sigma$ y $1 \leq i \leq n$, tal que i es la posición final de una corrida en B, se cumple

$$C_T[c] + Occ(T^{bwt}, c, i) = select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) - 1$$

3. Sean S, B y B' definidos para un string T^{bwt}. Entonces, para cada $c \in \Sigma$ y $1 \leq i \leq n$, tal que $T^{bwt}[i] \neq c$, se cumple

$$C_T[c] + Occ(T^{bwt}, c, i) = select(B', C_S[c] + 1 + Occ(S, c, rank(B, i))) - 1$$

4. Sean S, B y B' definidos para un string T^{bwt}. Entonces, para cada $c \in \Sigma$ y $1 \leq i \leq n$, tal que $T^{bwt}[i] = c$, se cumple

$$C_T[c] + Occ(T^{bwt}, c, i) = select(B', C_S[c] + Occ(S, c, rank(B, i))) + i - select(B, rank(B, i))$$

Las funciones *rank* y *select* pueden ser calculadas en tiempo constante, sólo queda entonces saber cuánto tiempo toma calcular *Occ* sobre la cadena S. Para este cálculo se pueden llevar a cabo las mismas soluciones expuestas en el capítulo 3.1.4. Por ejemplo, se puede reemplazar *Occ* por arreglos de bits con $n' = |S|$. Se obtiene un tiempo constante de consulta a *Occ* con $n'(H+1)$ bits de espacio, donde $H \leq \log \sigma$ es la entropía de orden cero de S.

Esta implementación alternativa del Índice FM mantiene el espacio $O(H_k n)$ de este índice mientras que remueve su dependencia exponencial en el tamaño del alfabeto, agregando un factor logarítmico en el tamaño del alfabeto en su requerimiento de espacio. Las complejidades de tiempo permanecen iguales. El índice RLFM resulta más pequeño que el FM cuando $\sigma \log \sigma > \log n$. Incluso para textos gigantescos, el Índice FM sólo es más pequeño para alfabetos muy pequeños. Esto incluye secuencias de ADN por ejemplo, pero no secuencias de proteínas, música o textos de lenguaje natural.

Tal como se explica en el capítulo anterior, es posible implementar la función *Occ* usando un árbol balanceado que mantiene la misma complejidad espacial y cambia el

tiempo de búsqueda de $O(m)$ a $O(m \log \sigma)$, con el beneficio de obtener $S[i]$ en tiempo $O(\log \sigma)$ en vez de $O(\sigma)$. Las complejidades obtenidas se pueden ver en la tabla 3.2 (RLFM2).

3.2. Otros Índices de texto completo

3.2.1. Compressed Suffix Array

El *compressed suffix array* (arreglo de sufijos comprimido) propuesto por Grossi y Vitter [14] es un índice de texto completo que funciona sólo para textos binarios ($\sigma = 2$). Ocupa sólo $O(n)$ bits para un texto de largo n, sin embargo, también utiliza el mismo texto, el que ocupa otros n bits. El índice CSA de Sadakane [18] modifica las estructuras de datos de este índice para que la búsqueda de un patrón pueda hacerse sin necesidad de accesos al texto original y para que funcione para cualquier alfabeto. Este nuevo índice puede encontrar las *occ* ocurrencias de un patrón P, de largo m, en un tiempo $O(m \log n + occ \log^{\epsilon} n)$, para cualquier valor fijo de $0 < \epsilon \leq 1$ sin accesar el texto. A su vez, puede descomprimir una parte del texto de largo L en tiempo $O(L + \log^{\epsilon} n)$. Este índice ocupa $O(nH_0 + n \log \log \sigma)$ bits de espacio si es que $\sigma = polylog(n)$.

El arreglo de sufijos comprimido original

El arreglo de sufijos comprimido de Grossi y Vitter reduce el tamaño del arreglo de sufijos de un texto de largo n de $n \log n$ bits a $O(n)$ bits. Cada elemento del arreglo de sufijos puede ser extraído en tiempo $O(\log^{\epsilon} n)$. Por lo tanto, se puede encontrar un patrón P de largo m, en un tiempo $O((m + \log^{\epsilon} n) \log n)$ usando el texto y el arreglo de sufijos comprimido mediante una búsqueda binaria directa. Una consulta enumerativa toma un tiempo adicional de $O(occ \log^{\epsilon} n)$. Este índice tiene dos desventajas. La primera es que debe utilizar el texto para realizar las búsquedas, por lo que el tamaño total del índice no puede ser menor que el texto, el que ocupa $O(n)$ bits. La segunda desventaja es que una aplicación directa del arreglo de sufijos comprimido con alfabeto binario a un alfabeto mayor resulta en índices de espacio $O(n \log \sigma)$. Por lo tanto, el índice ocupa más espacio que el texto.

El arreglo de sufijos comprimido utiliza una estructura de datos jerárquica. El nivel k-ésimo almacena implícitamente los índices de los sufijos que son múltiplos de 2^k. Un arreglo $SA_k[1 \ldots n_k]$ ($n_k = \frac{n}{2^k}$) almacena los índices que son divisibles por 2^k. Los índices son almacenados en el mismo orden que en el arreglo de sufijos SA.

Algoritmo $lookup_k\ (i)$
1. **if** $k = l$ **then return** $SA_l[i]$
2. $v \leftarrow 0$
3. **while** $B_k[i] = 0$ **do**
4. **if** $i = pos_k^n$ **then return** $n_k - v$
5. $i \leftarrow \Psi_k[i]$
6. $v \leftarrow v + 1$
7. **return** $2^e lookup_{k+e}(rank(B_k, i)) - v$

Fig. 3.3: Pseudo código del algoritmo para calcular SA_k. La variable pos_k^n es necesaria si n_k no es múltiplo de $\log^\epsilon n$. Esta variable representa el orden lexicográfico del último sufijo $T[n_k]$, es decir, $SA_k[pos_k^n] = n_k$.

El arreglo SA_k se convierte entonces en el arreglo de sufijos de una nueva cadena $T_k[1 \ldots n_k]$. Un carácter $T_k[j]$ consiste en la concatenación de 2^k caracteres $T[j2^k \ldots (j+1)2^k - 1]$. Entonces el arreglo SA_k coincide con el arreglo de sufijos de T_k. Por lo tanto, se utiliza la misma técnica de representación de SA_k recursivamente.

Se utilizan niveles $0, e, 2e, \ldots, l$, donde $e = \lceil \epsilon \log\log n \rceil$ $(0 < \epsilon \leq 1)$ y $l = \lceil \log\log n \rceil$. El nivel l-ésimo almacena explícitamente $n/\log n$ índices. Su tamaño es a lo más n bits. El k-ésimo nivel almacena un vector de bits $B_k[1 \ldots n_k]$ y una función $\Psi_k[i]$ $(1 \leq i \leq n_k)$ en vez de $SA_k[1 \ldots n_k]$.

Un elemento $B_k[i]$ del vector de bits indica si $SA_k[i]$ es un múltiplo de 2^k o no. Si $B_k[i] = 1$, $SA_k[i] = j2^k$ y el índice j es almacenado en SA_{k+e} implícitamente si $k + e < l$ o explícitamente si $k + e = l$. El orden lexicográfico i' de un sufijo en SA_{k+e} correspondiente a $SA_k[i]$ es calculado como $i' = rank(B_k, i)$ si $B_k[i] = 1$. Si $B_k[i] = 0$, $SA_k[i] = j2^e - v$ $(1 \leq v < 2^e)$ y es representado por v y un índice i' de SA_k, donde $SA_k[i'] = j2^e$.

La función $\Psi_k[i]$ se definida como: $\Psi_k[i] = i'$ tal que $SA_k[i'] = SA_k[i+1]$, si $SA_k[i] < n_k$, y 0 si $SA_k[i] = n_k$.

Por lo tanto, $SA_k[i]$ se calcula utilizando la relación

$$SA_k[i] = SA_k[\Psi_k[i]] - 1$$

iterativamente v veces mientras $B_k[i] = 0$. La figura 3.3 muestra el algoritmo para calcular SA_k. $SA[i]$ se calcula como $SA[i] = SA_0(i)$, lo cual toma tiempo $O(\log^\epsilon n)$.

CSA: un arreglo de sufijos comprimido que no necesita el texto

El arreglo de sufijos comprimido fue modificado por Sadakane para que pudiera ser utilizado sin el texto. En esta sección se muestra cómo un patrón puede ser buscado con búsqueda binaria sin la necesidad del texto y que el texto puede ser extraído desde el arreglo de sufijos comprimido.

Se utilizan vectores de bits $D_k[1 \dots n_k]$, definidos como: $D_k[i] = 1$ si $i = 1$ o si $T_k[SA_k[i]] \neq T_k[SA_k[i-1]]$ y 0 de lo contrario.

El vector D_0 es utilizado para obtener el carácter $T[SA[i]]$ de un sufijo si es que su orden lexicográfico i es conocido. Se utiliza un arreglo $C[1 \dots \sigma]$ en el cual $C[j]$ almacena el j-ésimo menor carácter aparecido en T y se define la función $C^{-1}[i] = C[rank(D_0, i)]$.

Entonces, $T[SA[i]] = C^{-1}[i]$, y es calculado en tiempo constante para un i dado. Esto se explica pues dado que los sufijos están ordenados lexicográficamente en el arreglo de sufijos, los primeros caracteres $T[SA[i]]$ de los sufijos están ordenados alfabéticamente. Si $D_0[i] = 1$ entonces $T[SA[i]]$ es diferente de $T[SA[i-1]]$. Por lo tanto, $r = rank(D_0, i)$ representa el número de caracteres diferentes en $T[SA[1]], T[SA[2]], \dots, T[SA[i]]$, y $C[r]$ es el carácter $T[SA[i]]$.

Esto significa que no es necesario calcular el valor exacto de $SA[i]$, lo que toma tiempo $O(\log^{\epsilon} n)$, para obtener el primer carácter del sufijo $T[SA[i] \dots n]$. Para calcular los siguientes caracteres, se utiliza la función Ψ. La función Ψ denota a Ψ_0 y Ψ^v denota la composición de Ψ v veces. Entonces, se tiene que $T[SA[i] + v] = C^{-1}[\Psi^v[i]]$, para $0 \leq v \leq n - SA[i]$. En consecuencia, un substring $T[SA[i] \dots SA[i] + m - 1]$ puede ser decodificado en tiempo $O(m)$ utilizando las funciones Ψ y C^{-1} m veces. La figura 3.4 muestra el algoritmo para extraer $T[SA[i] \dots SA[i] + m - 1]$ en $S[1 \dots m]$ en tiempo $O(m)$.

```
Algoritmo substring (i, m)
1.  for j ∈ 1...m
2.      do S[j] ← C^{-1}[i]
3.      i ← Ψ[i]
4.  return S
```

Fig. 3.4: Algoritmo que retorna $T[SA[i] \dots SA[i] + m - 1]$ a partir de un índice del arreglo de sufijos

La búsqueda de un patrón puede hacerse entonces mediante una búsqueda binaria sobre el arreglo de sufijos $SA[1 \dots n]$. En cada iteración de la búsqueda, una subcadena $T[SA[i] \dots SA[i] + m - 1]$ se compara con el patrón $P[1 \dots m]$. Este substring es

decodificado en $O(m)$ y la comparación toma también este tiempo. Por lo tanto, una búsqueda que cuenta las ocurrencias de un patrón de largo m desde el arreglo de sufijos comprimido de un texto de largo n toma tiempo $O(m \log n)$.

Decodificando el texto

El algoritmo *substring* puede decodificar una cadena $T[s \dots e]$ del texto. Sin embargo, este sólo puede hacerse si es que el orden lexicográfico i de $T[s \dots n]$ ($SA[i] = s$), es conocido. Entonces, se necesita calcular la inversa del arreglo de sufijos $i = SA^{-1}[s]$.

La idea del algoritmo, llamado *inverse*, es la siguiente. Se supone que se conoce $i = SA_{k+e}^{-1}[q]$. Entonces existe $i' \in [1, n_k]$ tal que $B_k[i'] = 1$ y $i = rank(B_k, i')$. Luego i' puede ser calculado como $i' = select(B_k, i)$. Como se cumple que $i' = SA_k^{-1}[2^e q]$, se puede calcular $SA_k^{-1}[2^e q + v]$ como $\Psi_k^v[i']$.

Se recorre la estructura jerárquica en la dirección opuesta a la dirección para calcular $SA[i]$. Se almacena el arreglo inverso SA_l^{-1} explícitamente en el nivel l-ésimo. A su vez, se utiliza el directorio de la función *select* para B_k para moverse desde un nivel superior a uno inferior. En cada nivel, se almacena $pos_k^1 - i$, tal que $SA_k[i] = 1$, pues $SA_k^{-1}[0]$ no existe. Este algoritmo calcula $SA^{-1}[j]$ en tiempo $O(\log^\epsilon n)$.

Combinando los algoritmos *substring* e *inverse* se puede extraer una cadena arbitraria $T[s \dots s+m-1]$ de un texto de largo m en tiempo $O(m+\log^\epsilon n)$. El índice del arreglo de sufijos i correspondiente al sufijo $T[s \dots n]$ puede calcularse en $O(\log^\epsilon n)$ con el algoritmo *inverse* y luego la subcadena puede ser extraída en $O(m)$ con el algoritmo *substring*.

3.2.2. Índice LZ

El índice LZ [16] está basado en el Ziv-Lempel Trie [17] que ocupa $4n' \log n'(1+o(1))$ bits de espacio, donde n' es el número de símbolos resultantes de la compresión de un texto con el algoritmo LZ78 o el LZW, lo cual tiende a $H_k n$ asintóticamente. Este índice permite reportar las *occ* ocurrencias de un patrón de largo m en un tiempo $O(m^3 \log \sigma + (m + occ) \log n')$, en el peor caso, donde σ es el tamaño del alfabeto.

Compresión Ziv-Lempel

La idea de la compresión Ziv-Lempel es reemplazar subcadenas en el texto por un puntero a las ocurrencias previas de éstas. Si el puntero ocupa menos espacio que la cadena que reemplaza, se obtiene una compresión de la cadena.

El algoritmo de compresión de Ziv-Lempel, llamado $LZ78$, se basa en un diccionario de bloques. Al principio de la compresión, el diccionario contiene un sólo bloque b_0 de largo 0. El paso de compresión es el siguiente: si suponemos que un prefijo $T_1 \dots T_j$ de T ha sido comprimido en una secuencia de bloques $Z = b_1 \dots b_r$, todos ellos en el diccionario, entonces buscamos el prefijo más largo del resto del texto $T_{j+1} \dots n$ que es un bloque del diccionario. Si llamamos a este bloque b_s, de largo l_s, construimos un nuevo bloque $b_{r+1} = (s, T_{j+l_s+1})$, escribimos el par el final del texto comprimido Z, es decir, $Z = b_1 \dots b_r b_{r+1}$ y agregamos el bloque al diccionario. La forma natural de representar este diccionario es utilizando un trie. El algoritmo LZW [25] es una variación de $LZ78$, en la cual la letra que conforma el segundo elemento de cada par no es codificado, sino que se toma como la primera letra del siguiente bloque (el diccionario comienza con un bloque por letra).

Búsqueda

En esta sección se describe la idea básica para buscar un patrón $P_{1\dots m}$ en un texto $T_{1\dots n}$, el cual ha sido comprimido utilizando el algoritmo $LZ78$ o el LZW en $n' + 1$ bloquees $B_0 \dots B_{n'}$, tal que $B_0 = \epsilon$, $\forall k \neq l, B_k \neq B_l$ y $\forall k \geq 1, \exists l < k, c \in \Sigma, B_k = B_l c$, es decir, no hay dos bloques iguales y cada bloque, excepto B_0, está formado por un bloque previo más una letra al final.

Definimos *rank* de una cadena como la posición de una cadena en un conjunto de cadenas ordenadas lexicográficamente.

A continuación se describen algunas estructuras de datos utilizadas en la búsqueda.

1. *LZTrie*: es el trie formado por los bloques $B_0 \dots B_{n'}$. Dadas las propiedades de la compresión $LZ78$, este trie tiene exactamente $n' + 1$ nodos, cada uno correspondiente a una cadena. Se definen las siguientes operaciones sobre un nodo x:

   a) $id_t(x)$: entrega el identificador del nodo x, es decir, el número k tal que x representa B_k.

   b) $leftrank_t(x)$ y $rightrank_t(x)$ retornan la posición mínima y máxima en orden lexicográfico representadas por los nodos en el subárbol con raíz x, entre el conjunto $B_0 \dots B_{n'}$.

   c) $parent_t(x)$ entrega la posición en el árbol del padre de x.

   d) $child_t(x, c)$ entrega la posición en el árbol del hijo de x por el carácter c, o *null* si éste no existe.

Además, el árbol debe proveer la operación $rth_t(r)$, la cual entrega el nodo que representa la r-ésima cadena en $B_0 \dots B_{n'}$ orden lexicográfico.

2. *RevTrie*: es el trie formado por las cadenas reversas $B_0^r \dots B_{n'}^r$. Se necesitan las mismas operaciones que para el *LZTrie*: $id_r, leftrank_r$, $rightrank_r, parent_r, child_r$ y rth_r.

3. *Node*: es un mapeo de los identificadores de bloques a su correspondiente nodo en el *LZTrie*.

4. *Range*: es una estructura para una búsqueda bidimensional en el espacio $[0 \dots n'] \times [0 \dots n']$. Los puntos almacenados en esta estructura son

$$\{(revrank(B_k^r), rank(B_{k+1})), k \in 0 \dots n' - 1\}$$

donde $revrank$ es el rank lexicográfico en $B_0^r \dots B_{n'}^r$ y $rank$ es el rank lexicográfico en $B_0 \dots B_{n'}$. Para cada punto se almacena el valor correspondiente de k.

Consideremos ahora el proceso de búsqueda. Dependiendo de la disposición de los bloques, podemos encontrar tres tipos de ocurrencias de P en T: la ocurrencia está completamente contenida en un bloque, la ocurrencia cubre dos bloques contiguos, la ocurrencia cubre más de dos bloques.

Se explica a continuación cómo cada tipo de ocurrencia es encontrada:

Ocurrencias en un bloque. Dadas las propiedades del algoritmo de compresión, cada bloque B_k que contiene P está formado por un bloque más corto B_l concatenado con una letra c. Si P no está al final de B_k, entonces B_l también contiene a P. La idea es encontrar el bloque más corto posible B que contiene la ocurrencia de P. Este bloque termina con la cadena P, por lo tanto puede encontrarse fácilmente buscando P^r en *Revtrie*.

Las ocurrencias de este tipo se encuentran se la siguiente manera:

1. Se buscar P^r en *Revtrie*. Se llega a un nodo x tal que todas las cadenas en el subárbol con raíz x representan bloques que terminan con P.

2. Se evalúa $leftrank_r(x)$ y $rightrank_r(x)$, obtaniéndose el intervalo lexicográfico de los bloques que terminan con P.

3. Para cada rank $r \in leftrank_r(x) \dots rightrank_r(x)$, se obtiene el nodo correspondiente en *LZTrie*, $y = Node(rth_r(r))$. Estos son los nodos en el trie normal que terminan con P y se debe reportar todos sus subárboles.

4. Para cada y recorremos el subárbol con raíz y y se reporta cada nodo encontrado.

Ocurrencias que cubren dos bloques. Se puede dividir P en cualquier posición, por lo que se debe probar todas las divisiones posibles. La idea es, para cada posible división, se busca el prefijo reverso del patrón en *RevTrie* y el sufijo del patrón en *LZTrie*. Se tienen entonces dos rangos, los bloques que terminan con la primera parte de P y los que empiezan con la segunda parte de P. Se necesita encontrar los pares de bloques $(k, k+1)$ tal que k está en el primer rango y $k+1$ en el segundo. Los pasos son:

1. Para cada $i \in 1 \dots m-1$, se divide P en $pref = P_{1\dots i}$ y $suf = P_{i+1\dots m}$ y se llevan a cabo los siguientes pasos.
2. Se buscar $pref^r$ en *RevTrie*, obteniéndose x, y se busca suf en *LZTrie*, obteniéndose y.
3. Se busca el rango:

$$[leftrank_r(x) \dots rightrank_r(x)] \times [leftrank_t(x) \dots rightrank_t(x)]$$

4. Para cada par $(k, k+1)$ encontrado, se reporta k.

Ocurrencias que cubren más de dos bloques. Recordando que los algoritmos de compresión aseguran que cada bloque representa una cadena distinta, existirá a lo más un bloque que calce con $P_{i\dots j}$, para cada elección de i y j. La idea es, primero, identificar este único bloque para cada subcadena $P_{i\dots j}$, almacenándose los bloques en m arreglos A_i, donde A_i almacena los bloques correspondientes a $P_{i\dots j}$ para todas las subcadenas de P. Luego, se trata de encontrar concatenaciones de bloques sucesivos B_k, B_{k+1}, etc. que calzan subcadenas contiguas del patrón. Nuevamente, hay sólo un candidato para continuar una ocurrencia de B_k en el patrón. Finalmente, para cada concatenación maximal de bloques $P_{i\dots j} = B_k \dots B_l$, se determina si B_{k-1} termina con $P_{1\dots i-1}$ y si B_{l+1} comienza con $P_{j+1\dots m}$. Si esto ocurre, se reporta una ocurrencia.

Capítulo 4

ÍNDICE FM HUFFMAN

En este capítulo se presenta el Índice FM Huffman [1], una estructura basada en el concepto del Índice FM, pero que no posee la dependencia exponencial del tamaño del texto que sí posee este índice. La idea de este índice es comprimir el texto con el algoritmo de Huffman y después aplicar la transformación de Burrows-Wheeler (BWT). La estructura resultante puede ser utilizada para buscar de igual manera que en el Índice FM. Sobre un texto de largo n, con entropía de orden cero H_0, el Índice FM Huffman necesita $O(n(H_0+1))$ bits de espacio, sin ninguna dependencia del tamaño del alfabeto. El tiempo promedio de búsqueda de un patrón de largo m es $O(m(H_0+1))$. Una posición de una ocurrencia puede ser reportada en un tiempo $O((H_0+1)\log n)$ en el peor caso, mientras que una subcadena del texto de largo L puede ser recuperada en un tiempo promedio $O((H_0+1)L)$.

La primera parte de este capítulo se refiere a la estructura del índice y la forma de construirlo. En la segunda parte se explica la forma de buscar un patrón utilizando este índice y las formas de reportar posiciones y mostrar un texto alrededor de una ocurrencia.

4.1. La estructura del Índice FM Huffman

El índice FM-Huffman funciona de la siguiente manera: primero se comprime el texto $T\$$ utilizando Huffman. La cadena de bits resultante, que llamaremos T', tendrá largo $n' < (H_0+1)n$, dado que la codificación de Huffman binaria resulta en una representación con un sobrecosto máximo de un bit por símbolo. Se define también un segundo arreglo de bits Th, del mismo largo de T', tal que $Th[i] = 1$ si y sólo si i es la posición inicial de una palabra código de Huffman en T'. Estos dos arreglos no serán representados en la estructura del índice.

La idea es buscar en el texto T' en vez de en el texto original T. Entonces, se aplica la transformación BWT al texto T', para obtener $B = (T')^{bwt}$. Sea $A'[1 \ldots n']$ el

arreglo de sufijos del texto T', esto es, una permutación de los números $1 \dots n'$ tal que $T'[A'[i] \dots n']] < T'[A'[i+1] \dots n']]$ en orden lexicográfico, para todo $1 \leq i < n'$. El arreglo de sufijos A' no será representado explícitamente, sino que se representará el arreglo de bits $B[1 \dots n']$, tal que $B[i] = T'[A'[i]-1]$ (con la excepción de que $B[i] = T[n']$ si $A'[i] = 1$). Además, se representa otro arreglo de bits $Bh[1 \dots n']$, tal que $Bh[i] = Th[A'[i]]$. Este arreglo dice si la posición i de A' apunta al comienzo de una palabra de codificación.

La figura 4.1 muestra un ejemplo del índice construido sobre el texto $T =$ "$alabar\_a\_la\_alabarda$". Se muestran el texto T y el arreglo de sufijos A del texto. A la derecha se muestran los códigos resultantes de la compresión de Huffman para cada carácter de T. Se muestra luego el texto comprimido T' y el arreglo Th en forma de caracteres bajo los bits de T. Cada carácter indica la posición donde comienza el código de ese carácter en T'. Más abajo se muestra el arreglo de sufijos A' de T'. Finalmente, se muestran los arreglos de bits B y Bh, las únicas estructuras que realmente se representan. Un punto bajo el bit de la posición i de B, representa un 1 en Bh. Se indica también la posición $p_{\#}$, la cual apunta a la entrada i de A' tal que $A'[i] = 1$.

4.2. Búsqueda de patrones

Se pretende buscar en B exactamente como con el Índice FM. Para ello se necesita el arreglo C y la función Occ. Sin embargo, dado que el alfabeto es binario, Occ puede ser calculado fácilmente como $Occ(B, 1, i) = rank(B, i)$ y $Occ(B, 0, i) = i - rank(B, i)$. A su vez, el arreglo C es tan simple para el texto binario que se puede funcionar sin él: $C[0] = 0$ y $C[1] = n' - rank(B, n')$, es decir, el número de ceros en B. Por lo tanto, $C[c] + Occ(T^{bwt}, c, i)$ es reemplazado en este índice por $i - rank(B, i)$ si $c = 0$ y $n' - rank(B, n') + rank(B, i)$ si $c = 1$.

Sin embargo, se debe introducir un cambio en el cálculo de Occ dado que no se coloca un caracter terminador a la secuencia binaria T' y por lo tanto no aparece ningún terminador en B. Sea # el caracter terminador de la secuencia binaria. En la posición $p_{\#}$ tal que $A'[p_{\#}] = 1$, debería tenerse $B[p_{\#}] = \#$. En cambio, se tiene que $B[p_{\#}]$ es el último bit de T'. Este es el último bit del código asignado por Huffman al terminador \$ de T\$. Dado que al momento de codificar los símbolos con Huffman, se puede elegir arbitrariamente concatenar un 1 o un 0 al final del código, se puede asegurar que el código asociado al terminador \$ tiene un 0 como último bit. Entonces, la secuencia correcta B tendría largo $n' + 1$ y comenzaría con 0, pues esta posición corresponde a $T'[n']$, el caracter que precede la ocurrencia de #, pues $\# < 0 < 1$, y se tendría que $B[p_{\#}] = \#$. Por lo tanto, para obtener el mapeo correcto a la secuencia B, se debe

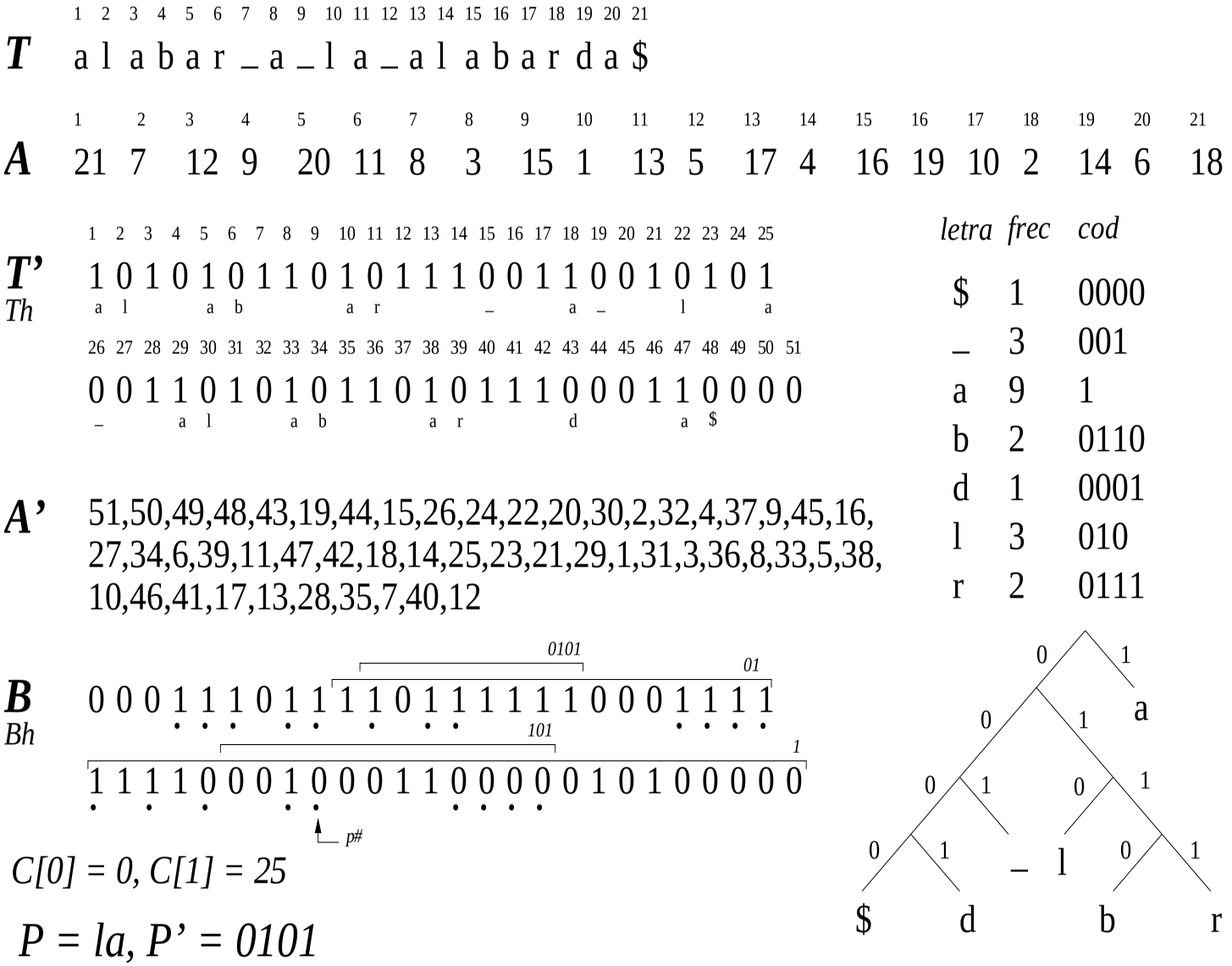


Fig. 4.1: Índice FM Huffman sobre el texto "alabar_a_la_alabarda"

corregir $C[0] + Occ(B, 0, i) = i - rank(B, i) + [i < p_{\#}]$. Esto es, si se suma 1 si es que $i < p_{\#}$. El cálculo de $C[1] + Occ(B, 1, i)$ no se modifica.

El patrón de búsqueda no es el patrón original P, sino que su representación binaria P' utilizando la codificación de Huffman aplicada a T. Convertir P a P' toma tiempo $O(m)$. Si se supone que los caracteres de P tienen la misma distribución que los de T, entonces el largo de P' es $< m(H_0 + 1)$. Este es el número de pasos necesarios para buscar en B utilizando el algoritmo de búsqueda para el Índice FM.

La respuesta a esa búsqueda, sin embargo, es diferente a la de la búsqueda de P en T. La razón es que la búsqueda en T' de P' devuelve el número de sufijos de T' que comienzan con P'. Estos incluyen los sufijos de T que comienzan con P, pero también otras ocurrencias superfluas pueden aparecer. Estas corresponden a sufijos de T' que no comienzan una palabra de codificación de Huffman, pero que igual comienzan con P'.

Esta es la razón por la cual se marcan los sufijos que comienzan una palabra de codificación de Huffman en Bh. En el rango $[sp, ep]$ encontrado por la búsqueda de P' en B, cada bit prendido en $Bh[sp \ldots ep]$ corresponde a una ocurrencia verdadera. Luego el verdadero número de ocurrencias puede ser calculado como $rank(Bh, ep) - rank(Bh, sp - 1)$.

La figura 4.2 muestra el algoritmo de búsqueda en pseudocódigo. La figura 4.1 muestra un ejemplo de búsqueda del patrón $P =$"la" en el texto $T =$"alabar_a_la_alabarda". El patrón se comprime a la cadena $P' = 0101$. La búsqueda ajusta el intervalo de ocurrencias de P' en B calculando los intervalos correspondientes a 1, 01, 101 y finalmente 0101. El intervalo resultante es $[sp, ep] = [11, 18]$, sin embargo, sólo 3 ocurrencias corresponden a ocurrencias reales de P', pues en ese intervalo hay 3 bits encendidos en Bh, correspondientes a las posiciones de *la* en T. El resto son ocurrencias de 0101 pero que no comienzan en el comienzo de un código de Huffman de algún símbolo.

Si se supone que la distribuciones de orden cero de T y P son similares, el tiempo de búsuqeda es $O(m(H_0 + 1))$. Para calcular cuál es el tiempo en el peor caso, debemos determinar cuál es el largo máximo de un código de Huffman. Esto equivale a determinar la altura máxima de un árbol de Huffman con frecuencia total n. Consideremos el camino más largo desde la raíz a una hoja en el árbol de Huffman. El símbolo de la hoja tiene una frecuencia de por lo menos 1. Recorriendo el camino hacia arriba y considerando la suma de las frecuencias encontradas en la otra rama, en cada nodo, nos damos cuenta de que estos números son, al menos: $1, 1, 2, 3, 5, \ldots$, esto es, la secuencia de Fibonacci $F(i)$. Entonces, un árbol de Huffman de profundidad d necesita que el texto tenga un largo de al menos $n \geq 1 + \sum_{i=1}^{d} F(i) = F(d + 2)$. Por lo tanto, el largo máximo de un código es $F^{-1}(n) - 2 = \log_{\phi}(n) - 2 + o(1)$, donde $\phi = (1 + \sqrt{5})/2$.

Entonces, el patrón codificado P' no puede ser más largo que $O(m \log n)$, y por lo

Algoritmo FM-Huffman_Search (P', B, Bh)
1. $i = m'$
2. $sp = 1$; $ep = n'$
3. **while**$((sp \leq ep)$ **and** $(i \geq 1))$ **do**
4. **if** $P'[i] = 0$ **then**
5. $sp = (sp - 1) - rank(B, sp - 1) + 1 + [sp - 1 < p_{\#}]$
6. $ep = ep - rank(B, ep) + [ep < p_{\#}]$
7. **else** $sp = n' - rank(B, n') + rank(B, sp - 1) + 1$
8. $ep = n' - rank(B, n') + rank(B, ep)$
9. $i = i - 1$
10. **if** $ep < sp$ **then** $occ = 0$ **else** $occ = rank(Bh, ep) - rank(Bh, sp - 1)$
11. **if** $occ = 0$ **then return** "no se encontró"
12. **else return** "se encontraron (occ) occurrencias"

Fig. 4.2: Algoritmo para contar el número de occurrencias del patrón $P'[1 \ldots m']$ en el texto $T[1 \ldots n']$.

tanto este es el tiempo de búsqueda en el peor caso.

4.3. Asegurando un tiempo $O(m \log \sigma)$ en el peor caso

Se puede limitar el tiempo de la búsqueda en el peor caso a $O(m \log \sigma)$ manteniendo el tiempo promedio $O(m(H_0 + 1))$ y el espacio $O(n(H_0 + 1))$. Esta técnica, la cual se ha utilizado en [22], consiste en modificar la codificación de Huffman para T, mientras todo el resto permanece intacto.

Se elige una constante t tal que el árbol de Huffman para T se sigue hasta una profundidad $t \log \sigma$. Todos los subárboles a esa profundidad se convierten en subárboles perfectamente balanceados. Entonces, la profundidad del árbol completo es a lo más $t \log \sigma + \log \sigma = (t + 1) \log \sigma$, y por lo tanto la búsqueda toma tiempo $O(m \log \sigma)$ en el peor caso.

Veamos ahora cuál es la complejidad en espacio considerando esta modificación. Si se toma una hoja en el árbol original de Huffman que tiene profundidad $l \geq t \log \sigma$. Sea f su frecuencia y $p = f/n$ su probabilidad empírica. Extendiendo el argumento utilizado al final de la sección 4.2, se tiene que los otros pesos de la rama deben ser a lo menos $1, f, f+1, 2f+1, 3f+2, 5f+3, \ldots$. Esto suma $f \cdot F(l+1) + F(l)$ luego de sumar el peso f de la hoja que se está considerando. Entonces, $n \geq f \cdot F(l+1) + F(l) \geq f \cdot F(l+1)$ y por lo tanto $p \leq 1/F(l+1) \leq \phi^{-l} \leq 1/\sigma^{t \log \phi}$. Entonces el caracter correspondiente a esa hoja puede aparecer en el texto a lo más $n/\sigma^{t \log \phi}$ veces.

Sumando estas cotas superiores de las probabilidades sobre todos los posibles caracteres que puedan estar tan profundos, tenemos que el número total de ocurrencias de estos caracteres en el texto no puede ser más que $n\sigma/\sigma^{t\log\phi} = n/\sigma^{t\log\phi-1}$. La profundidad de estos caracteres puede ser a lo más $(n/\sigma^{t\log\phi-1})(t+1)\log\sigma$. Por otra parte, todos los demás caracteres mantienen su profundidad original, y se supone pesimísticamente que ocupan un total de $n(H_0+1)$ bits. Por lo tanto el punto está en encontrar qué tan grande debe ser t para que $(1/\sigma^{t\log\phi-1})(t+1)\log\sigma = O(H_0+1)$. Resolviendo la inecuación $(1/\sigma^{t\log\phi-1})(t+1)\log\sigma \leq x(H_0+1)$, se obtiene que para cualquier $x \geq 1/(H_0+1)$ es suficiente que

$$t \;\geq\; \frac{1+\log_\sigma\log\sigma+\log_\sigma(t+1)}{\log\phi}$$

para lo cual es suficiente que

$$t \;\geq\; \frac{1+\log_\sigma\log\sigma+t/\ln\sigma}{\log\phi}$$

es decir,

$$t \;\geq\; (1+\log_\sigma\log\sigma)\frac{\ln\sigma}{\ln(\sigma)\log(\phi)-1}$$

lo cual tiende a $\log_\phi 2$ a medida que σ crece. Por lo tanto, t es una constante. Utilizando este valor para t, se tiene que $t\ \log\sigma = (\log\sigma+\log\log\sigma)\ln\sigma/(\ln(\sigma)\log(\phi)-1) = \log_\phi\sigma(1+o(1))$.

El número de bits que se necesitan para T' en total es $n(H_0+1+x(H_0+1)) = n(H_0+2)$, y por lo tanto B y Bh juntos necesitan $n(2H_0+4)(1+o(1))$ bits, sin grandes constantes escondidas en los términos sublineales. Entonces, el espacio que se debe pagar para asegurar un tiempo en el peor caso $O(m\log\sigma)$ es $2n$ bits extra.

El análisis del tiempo promedio de búsqueda es similar. A lo más $m\sigma/\sigma^{t\log\phi}$ veces el tiempo de la búsqueda será $O(\log\sigma)$, mientras el resto de las veces el tiempo será $O(H_0+1)$. De manera equivalente a como se hizo en el análisis de espacio, se busca una constante t tal que $m\sigma/\sigma^{t\log\phi} \leq xm(H_0+1)$ y se encuentra la misma solución que para asegurar un espacio extra $O(n(H_0+1))$. Entonces, el valor de t es el mismo para ambos propósitos, mantener la complejidad espacial de las estructuras y mantener la complejidad temporal de la búsqueda en el caso promedio.

4.4. Reportando las posiciones de las ocurrencias

Además de saber cuántas ocurrencias de un patrón hay en un texto, en la práctica se necesita saber también las posiciones en el texto de estas ocurrencias. Dado el intervalo del arreglo de sufijos que contiene las *occ* occurrencias encontradas, el Índice FM reporta cada posición en tiempo $O(\sigma \log^{1+\epsilon} n)$, para cualquier $\epsilon > 0$. El CSA puede reportar cada posición en tiempo $O(\log^{\epsilon} n)$, donde ϵ se paga en el espacio ocupado.

En esta sección se describe la forma de mostrar las posiciones de las ocurrencias del Índice FM Huffman y se muestra que se puede lograr un tiempo menor al tiempo que utiliza el Índice FM para mostrar las ocurrencias, aunque no menor que el tiempo del CSA. Utilizando $(1+\epsilon)$ bits adicionales, este índice puede reportar cada posición de una ocurrencia en tiempo $O(\frac{1}{\epsilon}(H_0+1)\log n)$.

El primer problema es cómo extraer, en tiempo $O(occ)$, las *occ* posiciones de los bits encencidos en $Bh[sp \dots ep]$. Esto se puede lograr utilizando la función *select*. Recordemos que $select(B, i)$ entrega la posición del i-ésimo 1 de B. Sea $r = rank(Bh, sp-1)$, entonces, las posiciones que buscamos en Bh son $select(Bh, r+1), select(Bh, r+2), \dots, select(Bh, r+occ)$. Recordemos que $occ = rank(Bh, ep) - rank(Bh, sp-1)$. Las posiciones de los bits encendidos en $Bh[sp \dots ep]$ pueden expresarse en función de *selectnext* como $pos_1 = selectnext(Bh, sp)$ y $pos_{i+1} = selectnext(Bh, pos_i + 1)$.

Veamos ahora cómo encontrar las posiciones en el texto de las ocurrencias a partir de las posiciones en Bh correspondientes. Para ello se necesitan algunas estructuras de datos auxiliares. Se elige un valor $\epsilon > 0$ y se toma una muestra de $\lfloor \frac{\epsilon n}{2 \log n} \rfloor$ posiciones de T' en intervalos regulares, con la restricción que se eligen sólo comienzos de códigos de Huffman. Se elige una posición de T' cada $\ell = \lceil \frac{2n'}{\epsilon n} \log n \rceil$, y para cada posición $1+\ell(i-1)$, se elige para la muestra la posición donde comienza el código de la letra representada en $1 + \ell(i-1)$. Dado que ningún código puede ser más largo que $\log_\phi n - 2 + o(1)$, la distancia entre dos muestras consecutivas no puede ser mayor que

$$\ell + \log_\phi n - 2 + o(1) \leq \frac{2}{\epsilon}(H_0+1)\log n + \log_\phi n - 1 + o(1) = O\Big(\frac{1}{\epsilon}(H_0+1)\log n\Big)$$

Se almacena un arreglo TS con las $\lfloor \frac{\epsilon n}{2 \log n} \rfloor$ posiciones de A' que apuntan a las posiciones elegidas de T', ordenados crecientemente por la posición del texto. Más precisamente, $TS[i]$ se refiere a la posición $1 + \ell(i-1)$ en T' y por lo tanto $TS[i] = j$ tal que $A'[j] = select(Th, rank(Th, 1+\ell(i-1)))$. El arreglo TS necesita $\frac{\epsilon n}{2}(1+o(1))$ bits, dado que cada entrada necesita $\log n' \leq \log(n \log min(n, \sigma)) = \log n + O(\log \log min(n, \sigma))$ bits.

Ahora almacenamos en un arreglo ST las posiciones del texto T correspondientes a cada entrada de TS pero ordenado en forma creciente, es decir, $ST[i] = rank(Th, A'[i])$. Este arreglo necesita $\frac{\epsilon n}{2}$ bits. Finalmente, se almacena un arreglo S de n bits tal que $S[i]$ es 1 si es que la i-ésima entrada de A' entre las que apuntan a un comiezno de código está en el conjunto de la muestra. Es decir, $S[i] = 1$ si y sólo si $A'[select(Bh, i)] \in TS$.

El espacio ocupado por estos tres arreglos es de $(1+\epsilon)n(1+o(1))$ bits, aumentando el espacio requerido para el índice a $n(2H_0+3+\epsilon)(1+o(1))$, y $n(2H_0+5+\epsilon)(1+o(1))$ para asegurar un tiempo $O(m \log \sigma)$ en el peor caso.

Veamos cómo se determina la posición del texto correspondiente a una entrada $A'[i]$ del arreglo de sufijos para la cual $Bh[i] = 1$. Se utiliza el arreglo de bits S para ver si es que la posición a la que apunta $A'[i]$ está en el conjunto tomado como muestra. Esto es cierto si y sólo si $S[rank(Bh, i)]$ es igual a 1. Si es que la posición está en la muestra se recupera la posición correspondiente de T como $ST[rank(S, rank(Bh, i))]$. Si es que la posición apuntada por $A'[i]$ no está en el conjunto de muestra, tal como en el Índice FM, se debe determinar la posición i' cuyo valor es $A'[i'] = A'[i] - 1$. Esto corresponde a moverse hacia atrás un bit en T'. Se repite este proceso hasta llegar a un comienzo de código, es decir, a un i' tal que $Bh[i'] = 1$. Ahora debe determinarse nuevamente si es que i' corresponde a una posición de T que está en la muestra utilizando S. Si es que está, se reporta la posición $1 + ST[rank(S, rank(Bh, i'))]$ y se termina. De otra manera se sigue hacia atrás hasta encontrar una posición i'' que corresponda a un comienzo de código. Si esta posición corresponde a una posición de T que está en la muestra se reporta $2 + ST[rank(S, rank(Bh, i''))]$, y así sucesivamente. Este proceso debe finalizar después de $O\left(\frac{1}{\epsilon}(H_0+1)\log n\right)$ pasos hacia atrás en T' pues se consideran posiciones consecutivas en T' y esa es la distancia máxima entre dos posiciones de T' que están en la muestra.

Falta explicar cómo se determina i' a partir de i. En el Índice FM esto se hace a través del mapeo $LF : i' = C[L[i]] + Occ(L, L[i], i)$. En este índice, el mapeo LF sobre A' se implementa como $i' = i - rank(B, i)$ si $B[i] = 0$ y $i' = n' - rank(B, n') + rank(B, i)$ si $B[i] = 1$. Esto permite moverse desde la posición $T'[A'[i]]$ a la posición $T'[A'[i]-1]$. En el peor caso una ocurrencia se puede reportar en tiempo $O\left(\frac{1}{\epsilon}(H_0+1)\log n\right)$. La figura 4.3 muestra el algoritmo para reportar una ocurrencia en pseudocódigo.

La figura 4.4 muestra un ejemplo de reporte de posiciones. Se utiliza un intervalo de largo $\ell = 9$ para la muestra de posiciones en T', eligiéndose las posiciones $1, 10, 19, 28, 37$ y 46. Cada una de estas posiciones se ajusta a la posición del comienzo del código de la palabra representada en ellas, obteniéndose las posiciones $1, 10, 19, 26, 34$ y 43. Estas posiciones, y sus correspondientes posiciones en T, están marcadas con un rectángulo

en la figura. El arreglo TS contiene las posiciones de A' que apuntan a las posiciones elegidas en la muestra, en orden creciente según la posición en T'. Las correspondientes posiciones en B también aparecen marcadas. El arreglo ST almacena las posiciones del texto T que corresponden a cada una de las posiciones de A' almacenadas en TS, en orden creciente de posiciones de A'. El arreglo S muestra, para cada posición i tal que $Bh[i] = 1$, si es que está en la muestra o no.

Se muestra también el proceso de buscar la posición de la primera ocurrencia del patrón *la* en T. La posición en Bh correspondiente a la primera ocurrencia del patrón es $selectnext(Bh, sp) = 11$. Como $S[11] = 0$, esta posición no está en la muestra y debemos avanzar hacia atrás hasta encontrar el comienzo de código de Huffman anterior en T'. Como $B[11] = 1$, la posición inmediatamente anterior es $i = 51 - rank(B, 51) + rank(B, 11) = 51 - 26 + 7 = 32$. Como $Bh[32] = 0$, esta posición no es el comienzo de un código y debemos seguir hacia atrás. Pasamos luego a la posición $12 = 32 - rank(B, 32) + [i < p_{\#}]$. Como $Bh[12] = 0$, seguimos hacia atrás a la posición $6 = 12 - rank(B, 12) + [i < p_{\#}]$. Ahora tenemos que $Bh[6] = 1$, por lo tanto esta posición corresponde al comienzo de un código. Asignamos entonces $d = d + 1 = 1$. A su vez, tenemos que $S[rank(Bh, i)] = S[3] = 1$, por lo que la posición del texto correspondiente a esta posición está en el conjunto de muestra. Retornamos entonces $d + ST[rank(S, rank(Bh, i))] = 1 + ST[rank(S, 3)] = 1 + ST[2] = 1 + 9 = 10$, que es la posición de una ocurrencia de *la* en T. Este proceso se repite para las demás ocurrencias comenzando desde las posiciones $13 = selectnext(Bh, 12)$ y $14 = selectnext(Bh, 14)$.

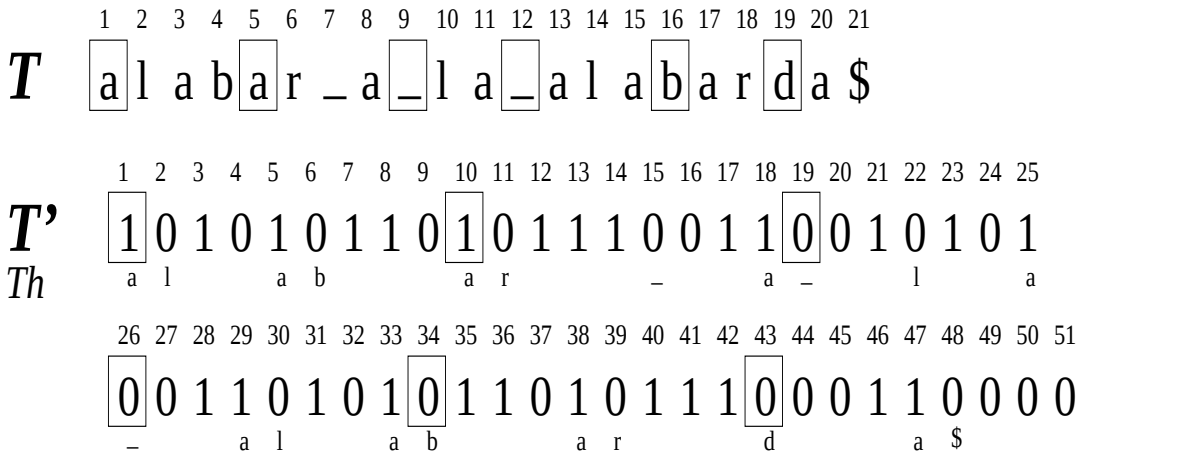
Algoritmo FM-Huffman_Position (i, B, Bh, S, ST)

1. $d = 0$
2. **while** $S[rank(Bh, i)]$ **do**
3. **do if** $B[i] = 0$ **then** $i = rank(B, i) + [i < p_{\#}]$
4. **else** $i = n' - rank(B, n') + rank(B, i)$
5. **while** $Bh[i] = 0$
6. $d = d + 1$
7. **return** $d + ST[rank(S, rank(Bh, i))]$

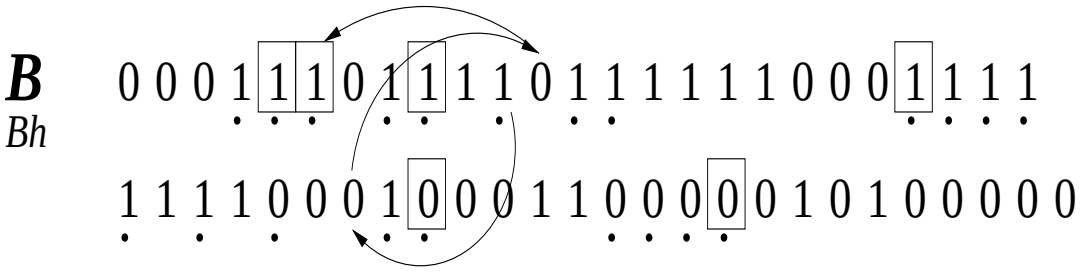
Fig. 4.3: Algoritmo para reportar la posición en el texto de una ocurrencia en $B[i]$. Este algoritmo se llama para cada $i = select(Bh, r + k)$, $1 \leq k \leq occ$, $r = rank(Bh, sp - 1)$.

4.5. Mostrando una subcadena del texto

Además de conocer el número de ocurrencias y las posiciones de un patrón en el texto, resulta útil contar con el contexto alrededor de la ocurrencia. Dado que los *auto*



A' 51,50,49,48,43,19,44,15,26,24,22,20,30,2,32,4,37,9,45,16,
27,34,6,39,11,47,42,18,14,25,23,21,29,1,31,3,36,8,33,5,38,
10,46,41,17,13,28,35,7,40,12



TS 34,42,6,9,22,5

ST 19,9,12,16,1,5

S 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1

Fig. 4.4: Encontrando la posición de la ocurrencia $B[11]$.

índices reemplazan el texto, en general se necesita extraer cualquier subcadena del índice mismo. El Índice FM puede mostrar una cadena de largo L en tiempo $O(\sigma(L+\log^{1+\epsilon} n))$, mientras que el CSA lo puede hacer en tiempo $(L + \log^{\epsilon} n)$. Utilizando las mismas estructuras necesarias para reportar las posiciones, es decir, $(1 + \epsilon)n$ bits adicionales a la estructura original, el Índice FM-Huffman puede mostrar una cadena del texto en tiempo $O(L \log \sigma + \log n)$ sumado al tiempo necesario para encontrar la posición de una ocurrencia. En promedio, suponiendo que se tienen posiciones aleatorias del texto, la complejidad temporal de mostrar una subcadena del texto se transforma en $O((H_0 + 1)(L + \frac{1}{\epsilon} \log n))$.

Para mostrar una subacadena $T[l \dots r]$ de largo L del texto primero se debe ubicar la menor posición del texto que está en la muestra y que es mayor que r. Esto se hace mediante una búsqueda binaria en TS. Dado el valor $TS[j]$ encontrado, la correspondiente posición en el texto es $ST[rank(S, rank(Bh, TS[j]))]$. La posición en A' de la primera posición del texto que sigue a r y que está en la muestra es $i = TS[j]$. Luego, nos movemos hacia atrás en T' mediante el mapeo LF, hasta encontrar el primer bit del código correspondiente a $T[r + 1]$. Esto implica realizar a lo más $O\left(\frac{1}{\epsilon}(H_0 + 1) \log n\right)$ pasos. Ahora continuamos hacia atrás para obtener las L posiciones anteriores en T recorriendo T' y recolectando los bits hasta encontrar el primer bit del código correspondiente a $T[l]$. La cadena de bits recolectada se decodifica para obtener $T[l \dots r]$.

Cada uno de los L caracteres implica un costo de tiempo $O(H_0+1)$ en promedio pues se obtienen los bits de cada código uno por uno. En el peor caso el costo es $O(\log n)$, u $O(\log \sigma)$ utilizando las técnicas descritas en el capítulo 4.3. La complejidad temporal total es $O((H_0 + 1)(L + \frac{1}{\epsilon}))$ en promedio y $O(L \log \sigma + (H_0 + 1)\frac{1}{\epsilon} \log n)$ en el peor caso.

Algoritmo FM-Huffman_Display (l, r, B, Bh, S, ST, TS)
1. $j = min\{k, ST[rank(S, rank(Bh, TS[k]))] > r$ //búsqueda binaria
2. $i = TS[j]$
3. $p = ST[rank(S, rank(Bh, i))]$
4. $L = \langle\ \rangle$
5. **while** $p \geq l$ **do**
6. **do** $L = B[i] \cdot L$
7. **if** $B[i] = 0$ **then** $i = rank(B, i) + [i < p_{\#}]$
8. **else** $i = n' - rank(B, n') + rank(B, i)$
9. **while** $Bh[i] = 0$
10. $p = p - 1$
11. decodificar los primeros $r - l + 1$ caracteres de L

Fig. 4.5: Algoritmo para extraer $T[l \dots r]$.

La figura 4.5 muestra el algoritmo para mostrar una subcadena de T en pseudocódi-

go. Siguiendo con el mismo ejemplo de la figura 4.4, supongamos que quisiéramos mostrar una subcadena del texto alrededor de una de las ocurrencias encontradas de *la* en $T[10]$, por ejemplo $T[8 \dots 13]$. Hacemos una búsqueda binaria en TS y encontramos que $TS[5] = 22$ corresponde a la posición del texto $ST[rank(S, rank(Bh, TS[5]))] = ST[rank(S, rank(Bh, 22))] = ST[rank(S, 9)] = ST[4] = 16$, la cual es la menor posición en la muestra que es mayor que 3. Aplicamos entonces el mapeo LF desde $B[22]$, recolectando todos los códigos de Huffman correspondientes a los caracteres $T[15], T[14], \dots, T[8]$. Partiendo de $B[22]$, el mapeo LF nos llevaría sucesivamente por los bits $B[22]B[39]B[15]B[35]B[23] \dots$. Los bits recolectados son 10101, los que dados vuelta corresponden a la codificación de $T[13 \dots 15] = ala$. Se sigue este proceso hasta llegar a $T[8]$.

Capítulo 5

IMPLEMENTACIÓN

En esta sección se mencionan algunas consideraciones con respecto a la implementación del Índice FM Huffman.

5.1. Supuestos

Durante la implementación del prototipo del índice FM-Huffman, se tomaron los siguientes supuestos:

- El alfabeto de los textos con que trabaja el índice es el alfabeto ASCII.
- El largo de los bloques en que se dividen los arreglos de bits B y Bh para el cálculo de la función $rank$ en tiempo constante nunca será mayor que el largo de la palabra de máquina. Recordemos que este largo b se calcula como $b = \lfloor \log(n)/2 \rfloor$. Esto implica que el arreglo de dos dimensiones Rp, que almacena para cada posible cadena S de bits de largo b y para cada posición i, el valor de $rank(S, i)$, puede utilizar como índice de la cadena una variable de tipo *long*. Esta es una suposición válida si se considera que para que el valor de b sea mayor a $W = 32$, el texto comprimido tendría que tener un largo de más de 2^{61} bytes.
- El arreglo de sufijos del texto comprimido, si bien no se almacena en el índice, se calcula en el proceso de construcción de éste. Además, ciertas posiciones del arreglo de sufijos se almacenan en el arreglo TS, utilizado para mostrar una subcadena del texto. Se supone que un número que indica una posición del arreglo de sufijos, así como también sus valores, puede ser representado con una variable de tipo *long*, usualmente de 32 bits. Esto implica que el arreglo de sufijos, y por lo tanto el texto comprimido no puede tener más de 2^{32} bits. Considerando que el texto comprimido ocupa aproximadamente 0,65 veces el tamaño del texto original, esto implica que este texto no puede ocupar, de igual manera, más de 2^{32} bits.

5.2. Cálculo del arreglo de sufijos

El arreglo de sufijos A' del texto comprimido almacena en orden lexicográfico de los sufijos de T', las posiciones donde éstos comienzan. Si $|T'| = n'$, el arreglo de sufijos deberá almacenar n' números enteros. Usualmente un número entero sin signo ocupa 32 bits, por lo que el arreglo de sufijos ocupa $32n'$ bits. Si consideramos que un tamaño razonable de memoria RAM son 512 Mb (=4.294.967.296 bits), mientras se calcula el arreglo de sufijos sólo podríamos almacenar en memoria real el arreglo de sufijos correspondiente a un texto comprimido de 4.294.967.296/32=134.217.728 bits, es decir, 16 Mb. Si es que se trata de un texto en inglés, un texto comprimido de este tamaño corresponde a un texto de aproximadamente 24,6 Mb.

La idea de un índice es poder permitir búsquedas de manera extremadamente eficiente en comparación a cualquier algoritmo de búsqueda secuencial o en línea. La ventaja del índice sobre estos algoritmos se hace notoria mientras más grande es el archivo donde se busca. Es por esto que es necesario que se pueda construir el índice sobre archivos bastante más grandes que 24 Mb.

Si es que el arreglo de sufijos cabe en memoria principal, la forma natural de calcularlo es comenzar con un arreglo que contenga todas las posiciones de T' y luego ordenarlo comparando lexicográficamente en cada paso la cadena de bits a que apunta cada posición.

Cuando el arreglo de sufijos no cabe en memoria se debe llevar a cabo otra estrategia. La forma en que se implementó el cálculo del arreglo de sufijos en este índice fue la siguiente:

Si el arreglo cabe en memoria, se calcula en memoria de la forma descrita anteriormente. De lo contrario, se calcula el arreglo en bloques que se almacenan en memoria secundaria. Los bloques se determinan según los prefijos de un largo determinado de cada sufijo de T'. Si para la construcción del arreglo se utilizan b bloques, los sufijos de T' se separan según el prefijo de largo $\log b$ de cada sufijo. En el bloque b_i estarán los sufijos que comienzan con la codificación binaria de i. Por ejemplo, si el número de bloques es 8, el bloque 0 corresponderá a los sufijos que comienzan con 000, el 1 a los sufijos que comienzan con 001 y así sucesivamente. Por simplicidad se supone que el número de bloques es una potencia de dos. Si para construir el arreglo de sufijos bastan b' bloques se utilizarán $b = 2^{\lceil \log b' \rceil}$ bloques. En general todos los bloques tendrán tamaños similares pues el número de sufijos de T' que comienza con cada uno de los prefijos a considerar para cada bloque es muy parecido. Si así no fuera, y por ejemplo, el prefijo 000 fuera mucho más frecuente que los demás, T' podría comprimirse nuevamente, pero como T' está comprimido con Huffman, sabemos que su tamaño es cercano a la salida

de un compresor ideal, por lo que no puede ser compresible en forma significativa.

Para cada bloque, los sufijos correspondientes se buscan con el algoritmo de búsqueda secuencial $Shift - And$ [27], y sus posiciones se almacenan en un arreglo de enteros. Este arreglo se ordena como si fuera el arreglo de sufijos completo, es decir, comparando en cada paso lexicográficamente los sufijos a los que apunta cada valor del arreglo. Una vez terminado este proceso, se almacena el arreglo en disco y se continúa con el siguiente bloque. Naturalmente, todos los sufijos correspondientes al bloque i son lexicográficamente menores a los del bloque $i+1$, justamente porque los primeros comienzan con la codificación de i, que es lexicográficamente menor a la codificación de $i+1$. Al final de este proceso, se tiene el arreglo de sufijos separado en bloques, almacenado en memoria secundaria. Se almacena además en memoria un arreglo que indica para cada bloque la posición del arreglo de sufijos correspondiente al primer valor almacenado en ese bloque. Cada vez que se necesita una entrada del arreglo de sufijos durante la construcción del índice, se utiliza este arreglo para determinar en qué bloque se encuentra el valor que se está buscando, se trae el bloque desde memoria secundaria y se recupera el valor buscado.

5.3. Función Rank

Recordemos de la sección 2.6.1 que para el cálculo en tiempo constante de *rank* sobre un arreglo de bits B se divide el arreglo en bloques de tamaño $b = \lfloor \log(n)/2 \rfloor$ y estos bloques se agrupan en superbloques de largo $s = b\lfloor \log n \rfloor$. Para cada superbloque se almacena el número de bits encendidos hasta antes de ese superbloque, y lo mismo se hace con cada bloque, pero contando sólo desde el comienzo del superbloque en que está contenido el bloque. Además, almacenamos una tabla con valores precalculados para cada posible cadena de bits de largo b y cada posición i dentro de ella. En total, estas estructuras necesitan $O(n/\log n + n \log\log n/\log n + \sqrt{n}\log n \log\log n) = o(n)$ bits, donde $n = |B|$. Si tenemos un arreglo de bits B de tamaño 2^{26} (64 megabits), el espacio extra que ocupan estas estructuras es un 81,4 % del tamaño original de la cadena.

En esta implementación, utilizamos una solución más práctica para esta problema [29]. Notemos que la tabla de valores precalculados R_p no hace más que contar el número de 1's en una máscara de bits de largo b. Para una palabra de máquina de 32 bits, probablemente la solución más eficiente para contar el número de 1's es:

```
popc[x&0xFF] + popc[(x>>8)&0xFF] + popc[(x>>16)&0xFF] + popc[x>>24]
```

en la cual se supone que la palabra de máquina tiene 32 bits y *popc* es una tabla con

valores precalculados con el número de 1's totales para todas las posibles cadenas de 8 bits. La idea es utilizar esta tabla para contar los 1's en cada uno de las cuatro cadenas de 8 bits de la palabra de 32 bits. Este procedimiento requiere de sólo 9 operaciones y 4 accesos a una tabla constante y es la solución que se utiliza actualmente en la librería de `Gnu g++`[1]. La tabla R_p almacenaba los valores precalculados de *rank* para todas las cadenas de largo b y para todas las posiciones, por lo tanto, para contar las ocurrencias dentro de un bloque, se miraba la entrada correspondiente al bloque y a la posición relativa dentro de ella. En este caso, esta solución cuenta los bits encendidos en una palabra de 32 bits. Entonces, para obtener el valor de *rank* hasta cierta posición dentro de ella, lo que se hace es, antes de llamar al procedimiento, hacer un *and* de la palabra con una máscara que deja en cero todos los bits posteriores a la posición relativa que nos interesa. Se llama luego a este procedimiento con la palabra modificada, obteniéndose así el resultado deseado.

Dado que en la implementación del índice reemplazamos R_p por esta solución, fijamos $b = W$, donde W es el tamaño de la palabra de máquina. En teoría debiéramos tener superbloques de largo $s = W\lfloor \log n \rfloor$, sin embargo, en la práctica resulta útil tener superbloques que contienen 8 bloques, es decir, de tamaño igual a 256 bits cuando $W = 32$. La ventaja de tener superbloques de 256 bits radica en que la estructura R_b puede implementarse con un arreglo de *bytes*, pues una entrada de este tipo puede codificar un número menor o igual a 256. Entonces, no se hace necesario el uso de una estructura de bits que provea de un acceso a cadenas de un largo variable de bits, sino que se utiliza un arreglo de *bytes*, haciendo más eficiente el cálculo de $Rank$. El arreglo R_s se implementa con un arreglo de *uint* (enteros sin signo), variable de 32 bits. El arreglo R_s necesita entonces $\lceil n/256 \rceil W = n/8$ bits aproximadamente, mientras que el arreglo R_b necesita $\lceil n/256 \rceil \cdot 8 \cdot 8 = n/4$ bits aproximadamente. La tabla precalculada para *rank* tiene $2^8 = 256$ entradas, con una variable de tipo *char* para cada una de ellas, ocupando un total de $256 * 8 = 2048$ bits.

Con respecto a las estructuras originales, el nuevo arreglo R_s ocupa más espacio que el original, sin embargo, existe una considerable reducción del espacio del arreglo R_b con respecto al original, lo que hace que estas estructuras ocupen menos espacio que las de la solución teórica. En ambos casos, los espacios ocupados por R_p y las tablas *popc* son despreciables con respecto a los de los arreglos R_s y R_b cuando n es lo suficientemente grande. Por ejemplo, para un arreglo de bits de largo 2^{26} (64 megabits), el espacio ocupado por las estructuras para *rank* sería ahora un $37,5\,\%$ del tamaño de la cadena de bits. Este porcentaje se mantiene para cualquier valor lo suficientemente grande de n tal que el espacio ocupado por la tabla *popc* es irrelevante.

[1] http://www.gnu.org

5.4. Función SelectNext

Al igual que la función *rank*, la función *selectnext* puede calcularse en tiempo constante utilizando algunas estructuras auxiliares. En la implementación de este índice no se incluyen todas las estructuras, sino que sólo una tabla t, similar a la tabla N_p, que contiene para las posibles cadenas de bits de largo 8, la posición del primer bit encendido, es decir, $t[i] = selectnext(i, 0)$.

Si B es un arreglo de n bits representado por un arreglo de variables de tipo *long*, para calcular $selectnext(B, i)$, ubicamos la palabra donde está ubicado i en el arreglo. Hacemos shift a esa palabra para que sólo queden los bits a partir del bit correspondiente a i. Si esta palabra truncada es distinta de 0, hay un bit encendido en ella, por lo que calculamos la posición del primer bit encendido a partir de i utilizando la tabla. De otra manera, avanzamos en el arreglo de *long* hasta encontrar una palabra que sea distinta de 0. Llamemos $p[1 \dots 32]$ a la palabra encontrada. Si consideramos que una palabra de máquina tiene 32 bits, la tabla t nos permite calcular en tiempo constante *selectnext* sobre una cuarta parte de una palabra de máquina. Entonces, miramos los primeros 8 bits de la palabra, si éstos son distintos de cero, miramos en la tabla la posición del primer bit encendido en esta cadena. Si esta cadena era cero, miramos los segundos 8 bits, y así sucesivamente. Como sabemos que la palabra completa es distinta de cero, por lo menos existe un bit encendido en alguna de las 4 cadenas de 8 bits de la palabra. Cuando encontramos un bit encendido sumamos la cantidad de bits correspondientes a la cantidad de palabras iguales a 0 que encontramos durante la búsqueda de p y retornamos el valor. Si ninguna palabra es distinta de cero, retornamos $n + 1$.

Recordemos que la función *selectnext* se utiliza exclusivamente sobre el arreglo Bh, el cual contiene un 1 en cada posición en que empieza un código de Huffman. El largo promedio de un código de Huffman depende de la naturaleza del texto que se haya comprimido. En un texto en inglés, el largo promedio de un código es de 5 bits. Es decir, en Bh habrá, en promedio, un bit encendido por cada 5 bits. Dada la forma en que se ordena B, se puede considerar que la ubicación de ceros y unos en Bh es casi aleatoria. Por esto, buscar una palabra en el arreglo de bits que sea distinta de cero toma tiempo constante con alta probabilidad. Cuando ya tenemos la palabra p donde sabemos que hay un bit encendido, en el peor caso debemos evaluar tres veces si una cadena de 8 bits es cero y mirar una vez la tabla t para entregar el resultado. Por lo tanto, en la práctica, calcular *selectnext* utilizando la estrategia descrita toma tiempo constante y tiene como ventaja que sólo se utilizan $8 * 2^8$ bits extra, almacenando la tabla como un arreglo de variables de tipo *char*.

5.5. Peor caso versus caso promedio

En la sección 4.3 se describe una estrategia para asegurar que la búsqueda de un patrón en el texto tome tiempo $O(m \log \sigma)$ en el peor caso. Esto se logra modificando el árbol de Huffman de manera que todos los árboles con cierta profundidad t se transformen en árboles perfectamente balanceados. La profundidad del árbol es a lo más $(t+1) \log \sigma$. Como t es constante, la búsqueda toma tiempo $O(m \log \sigma)$ en el peor caso.

Si bien es cierto que la complejidad de tiempo de la búsqueda en el caso promedio se mantiene igual, el tiempo promedio con esta modificación es ligeramente superior al tiempo promedio con el árbol de Huffman original.

Existen algunas aplicaciones en las que es importante garantizar que el tiempo en el peor caso estará acotado. Ejemplos de estas aplicaciones son las que proveen servicios en tiempo real. Sin embargo, en la mayoría de las aplicaciones, resulta más importante que el tiempo sea bajo en el caso promedio. Es por esto que nos hemos centrado en minimizar este tiempo, dejando de lado la técnica que asegura tener un peor caso acotado.

Capítulo 6

HUFFMAN K-ARIO

La idea de comprimir el texto utilizando la codificación de Huffman en el Índice FM Huffman antes de construir el índice es eliminar la dependencia exponencial del tamaño del alfabeto del Índice FM. La codificación de Huffman utilizada es una codificación binaria, es decir, se utilizan dos símbolos para la codificación de los símbolos originales. Como se menciona en la sección 2.5, la codificación de Huffman puede hacerse utilizando más de dos símbolos. En general, cada símbolo original puede ser codificado utilizando una cadena en $\{0, 1, \ldots, k-1\}^*$. Se dice entonces que se utiliza una codificación k-aria. Hasta ahora hemos visto cómo se puede implementar el Índice FM Huffman cuando se comprime el texto con una codificación binaria, es decir, cuando $k = 2$.

Resulta interesante estudiar cómo se comporta el índice cuando se utiliza una codificación de más símbolos, tanto en espacio como en tiempo. Primero, algunas estructuras reducen su tamaño a medida que se aumenta el número de símbolos utilizados en la codificación, como es el caso de Bh, cuyo tamaño es $1/\lceil \log k \rceil$ veces el tamaño original, pues al tenerse más símbolos en la codificación, los códigos de Huffman resultan más cortos y por lo tanto T' tiene menos símbolos. Por otra parte, existirá un aumento en el requerimiento de espacio debido a que ahora se necesitarán estructuras que permiten calcular *Rank* sobre B en tiempo constante para cada símbolo de la codificación, mientras que antes se necesitaba una única estructura para la cadena de bits. A su vez, al aumentar k el nivel de compresión se hace peor, aumentanto el tamaño de B y aumentando aún más el espacio ocupado por las estructuras de *Rank* sobre éste. El tiempo de búsqueda es proporcional al largo del patrón. Como ahora el patrón se codifica con más símbolos, tiene un largo menor que con la codificación binaria. Por lo tanto, el tiempo de búsqueda se reduce a medida que aumenta el número de símbolos de codificación.

Se puede ver entonces que se tienen diferentes compromisos relevantes entre tiempo y espacio dependiendo del valor de k que se use. Por esto, sería interesante estudiar cómo se comporta el índice para distintos valores de k.

En este capítulo se describe la estructura y algoritmos del Índice FM Huffman cuando

se utiliza una codificación de k símbolos para comprimir el texto. Al índice resultante lo llamaremos Índice FM Huffman k-ario.

6.1. La estructura del índice

Supongamos que tenemos un texto T de largo n al cual le concatenamos un símbolo terminador \$ al final. Comprimimos este texto con Huffman k-ario. El texto resultante T' tendrá ahora un largo de $n' < (H_0^{(k)}+1)n$ símbolos, donde $H_0^{(k)}$ es la entropía de orden cero del texto, calculada con logaritmo en base k. Es decir, $H_0^{(k)} = -\sum_{i=1}^{\sigma} \frac{n_i}{n} \log_k \left(\frac{n_i}{n}\right) = H_0/\log_2 k$. Llamamos Th al arreglo de bits que, al igual que antes, almacena las posiciones de los comienzos de códigos de Huffman en T'. Es decir, $Th[i] = 1$ si y sólo si $T'[i]$ es el comienzo de una palabra de la codificación. Th también tiene largo n'.

Aplicamos entonces las transformación de Burrows-Wheeler para obtener $B = (T')^{bwt}$. Calculamos para ello el arreglo de sufijos A' sobre T'. Recordemos que el arreglo de sufijos $A'[1 \dots n']$ es una permutación de los números $1 \dots n'$ tal que $T'[A'[i] \dots n']] < T'[A'[i+1] \dots n']]$ en orden lexicográfico, para todo $1 \leq i < n'$. Al igual que en el Índice FM Huffman original, no almacenamos el arreglo de sufijos sino que almacenamos el arreglo de símbolos $B[1 \dots n']$ tal que $B[i] = T'[A'[i]-1]$, con la excepción de que $B[i] = T[n']$ si $A'[i] = 1$. Es decir, la posición i de B almacena el código del carácter que está justo antes del código del caracter $A'[i]$. También se representa el arreglo de bits $Bh[1 \dots n']$ tal que $Bh[i] = Th[A[i]]$. Recordemos que este arreglo dice si la posición i de A' apunta a un comienzo de palabra de codificación de Huffman.

Veamos cuál es el tamaño de estas estructuras. Naturalmente, cada símbolo de la codificación de Huffman debe a su vez codificarse con $k' = \lceil \log k \rceil$ bits. Por lo tanto, el espacio ocupado por T', y por lo tanto B, es $n'_k = n' \times k'$. Sin embargo, las estructuras Th y Bh sólo necesitan un bit por símbolo de T' y B respectivamente, por lo que su tamaño en bits es n'. Por lo tanto, el espacio ocupado por las estructuras B y Bh es $n' + n' \times k' = n'(1+k') < n(H_0^{(k)}+1)(1+\log k)$. Recordemos que el espacio ocupado por estas estructuras para la versión original del índice era $2n(H_0+1)$. Es interesante notar que la relación $n(H_0^{(k)}+1)(1+\log k) \leq 2n(H_0+1)$ se cumple cuando el valor $\log k$ está entre 1 y H_0. Es decir, esto nos da una idea de el valor de k a utilizar para que las estructuras B y Bh ocupen menos espacio que las del índice con $k = 2$, dependiendo de la entropía del texto sobre el cual se quiere construir el índice.

6.2. Arreglo C y función Occ

Al igual que en la versión original de este índice, la idea es buscar en B tal como se hace con el Índice FM. Para ello necesitamos el arreglo C y la función Occ. En la versión del índice con codificación binaria de Huffman, C no se necesitaba, pues $C[0] = 0$ y $C[1] = n' - rank(B, n')$, es decir, el número de 0's en B. La función Occ se calculaba utilizando la función $rank$, $Occ(B, 1, i) = rank(B, i)$ y $Occ(B, 0, i) = i - rank(B, i)$. En este caso sí necesitamos el arreglo C y ya no podemos calcular Occ con una función que nos entrega el número de 1's en un arreglo de bits. Tendremos entonces un arreglo C tal que $C[c]$ es el número de ocurrencias de los caracteres $0, 1, \ldots, c-1$ en B. Es decir, $C[0] = 0$ y $C[c] = C[c-1] + Occ(B, c, n')$. Necesitamos almacenar $C[c]$ para cada uno de los símbolos de la codificación de Huffman, por lo tanto el arreglo C ocupa un espacio de $O(k \log n)$ bits.

Para el cálculo de la función Occ podemos hacer algo similar a lo que se hace en la versión del índice para $k = 2$. Procesamos B de igual manera a la que se hacía para la función $rank$, pero esta vez almacenamos arreglos con valores precalculados de las ocurrencias de los caracteres $0, 1, \ldots, k-1$. Tendremos entonces arreglos R_s, R_b y R_p para cada uno de los símbolos de la codificación de Huffman. Como cada una de estas estructuras ocupa $o(n')$ bits, tenemos que en total, estas estructuras ocupan $o(kn') = o(kH_0^{(k)}n)$ bits. Bh es procesado de igual manera que para la versión original, pues sólo necesitamos saber la cantidad de 1's hasta cierta posición para saber cuántas ocurrencias reales hay en un intervalo en B. Las estructuras de rank de Bh agregan otros $o(H_0^{(k)}n)$ bits. En total, la estructura ocupa a lo sumo $n(H_0^{(k)} + 1)(1 + \log k) + o(H_0^{(k)}n(k+1))$ bits.

6.3. Búsqueda de patrones

Para buscar un patrón P, primero obtenemos P' codificando P con la misma codificación utilizada para T. El largo de P' será m' símbolos. Suponiendo que el patrón y el texto tienen la misma distribución, se tiene que $m' < m(H_0^{(k)} + 1)$.

La búsqueda de P' se realiza de igual manera que en el Índice FM, utilizando el mapeo LF mediante el arreglo C y la función Occ. Al igual que en el caso del Índice FM Huffman original, debido a que no estamos colocando un terminador a la secuencia codificada, debemos corregir el mapeo LF. Como al construir los códigos de Huffman podemos asignar un 1 o un 0 arbitrariamente a un determinado código (ver sección 2.5), podemos asegurar que el último símbolo de la codificación del terminador \$ de T será 0.

Entonces, para corregir el mapeo LF sumamos 1 a $C[0]+Occ(B,0,i)$ cuando i es menor a la posición $p_{\#}$ tal que $A'[p_{\#}]=1$.

Una vez obtenido el intervalo $[sp, ep]$ resultante de la búsqueda, debemos eliminar posibles ocurrencias de P' que no comienzan en posiciones marcadas como comienzos de códigos de Huffman. El número correcto de ocurrencias es, al igual que en la versión original, $rank(Bh, ep) - rank(Bh, sp-1)$.

El tiempo de búsqueda es proporcional al largo del patrón, es decir el tiempo es $O(m(H_0^{(k)}+1))$. Como $H_0^{(k)} = H_0/\log k$, se cumple que para todo $k>1$, $H_0^{(k+1)} < H_0^{(k)}$. Entonces, mientras más símbolos se utilicen en la codificación de Huffman, menor será el tiempo de búsqueda. Cabe mencionar que este tiempo nunca será menor que $O(m)$.

La figura 6.1 muestra el algoritmo de búsqueda en pseudocódigo.

```
Algoritmo FM-Huffman(k)_Search (P', B, Bh)
1.   i = m'
2.   sp = 1; ep = n'
3.   while((sp ≤ ep) and (i ≥ 1)) do
4.        c = P'[i]
5.        if c = 0 then
6.             sp = Occ(B, c, sp − 1) + 1 + [sp − 1 < p#]
7.             ep = Occ(B, c, ep) + [ep < p#]
8.        else
9.             sp = C[c] + Occ(B, c, sp − 1) + 1
10.            ep = C[c] + Occ(B, c, ep)
11.       i = i − 1
12.  if ep < sp then occ = 0 else occ = rank(Bh, ep) − rank(Bh, sp − 1)
13.  if occ = 0 then return "no se encontró"
14.  else return "se encontraron (occ) occurrencias"
```

Fig. 6.1: Algoritmo para contar el número de occurrencias del patrón $P'[1\ldots m']$ en el texto $T[1\ldots n']$ en el Índice FM Huffman k-ario.

6.4. Reporte de posiciones de las ocurrencias

Para reportar las posiciones de las ocurrencias encontradas en la búsqueda se sigue el mismo procedimiento que en la versión original del índice.

Recordemos de la sección 4.4 que las *occ* ocurrencias encontradas corresponden a los bits encendidos en $Bh[sp\ldots ep]$ en las posiciones $pos_1 = selectnext(Bh, sp)$ y $pos_{i+1} = selectnext(Bh, pos_i + 1)$. Para encontrar las posiciones en el texto correspondientes a

cada posición de una ocurrencia en Bh, necesitamos las estructuras auxiliares ST y S. Estas estructuras se calculan de igual manera que en la versión original del índice. Se toma una muestra de $\lfloor \frac{\epsilon n}{2 \log n} \rfloor$ posiciones de T' a intervalos regulares, con $\epsilon > 0$, con la restricción de que se eligen sólo comienzos de códigos de Huffman. En un arreglo TS se almacenan las posiciones de A' que apuntan a las posiciones elegidas de T'. Al igual que antes, este arreglo necesita $\frac{\epsilon n}{2}(1 + o(1))$ bits. Entonces, almacenamos en ST las posiciones del texto T correspondientes a cada entrada de TS ordenado en forma creciente, necesitándose para ello $\frac{\epsilon n}{2}$ bits. El arreglo S indica si una posición de A' que apunta a un comienzo de código de Huffman está en la muestra o no. Este arreglo necesita n bits.

Para encontrar una posición del texto a partir de una posición i de Bh, al igual que en el índice original, se determina si esa posición está o no en la muestra con el arreglo S. Si la posición está en la muestra, se recupera la posición del texto como $ST[rank(S, rank(Bh, i))]$. Si la posición no está en la muestra, se determina la posición i' tal que $A'[i'] = A'[i] - 1$, lo que corresponde ahora a moverse un símbolo hacia atrás en T'. Se repite esto hasta llegar a un comienzo de palabra. Si esta posición está en la muestra se reporta la posición en T con el arreglo ST. Si es que la posición no está en la muestra, se sigue hacia atrás hasta encontrar una posición que esté en la muestra. Este proceso debe finalizar luego de $O\left(\frac{1}{\epsilon}(H_0^{(k)} + 1) \log n\right)$ pasos, pues ésta es la distancia máxima entre dos posiciones de T' que están en la muestra.

Para encontrar la posición i' tal que $A'[i'] = A'[i] - 1$, al igual que en el Índice FM, se debe calcular el mapeo LF, es decir $i' = C[B[i]] + Occ(B, B[i], i)$. La figura 6.2 muestra el algoritmo de reporte de una posición en pseudocódigo. Este algoritmo toma tiempo $O\left(\frac{1}{\epsilon}(H_0^{(k)} + 1) \log n\right)$ en el peor caso.

```
Algoritmo FM-Huffman_Position(k) (i, B, Bh, S, ST)
1.   d = 0
2.   while S[rank(Bh, i)] do
3.        do  c = B[i]
4.            if  c = 0 then
5.                 i = Occ(B, c, i) + [i < p_#]
6.            else
7.                 i = C[c] + Occ(B, c, i)
8.        while  Bh[i] = 0
9.        d = d + 1
10.  return  d + ST[rank(S, rank(Bh, i))]
```

Fig. 6.2: Algoritmo para reportar la posición en el texto de una ocurrencia en $B[i]$. Este algoritmo se llama para cada $i = select(Bh, r + k)$, $1 \leq k \leq occ$, $r = rank(Bh, sp - 1)$.

6.5. Muestra de una subcadena del texto

La muestra de texto en el Índice FM Huffman k-ário funciona igual que en el Índice FM Huffman original, con la diferencia de que debemos modificar el mapeo LF al igual que en la búsqueda y el reporte de posiciones.

Para mostrar una subcadena $T[l \dots r]$ de largo L ubicamos la primera posición del texto que es mayor que r y que está en la muestra, mediante una búsqueda binaria en TS. A partir de la posición de A' correspondiente a al valor encontrado en TS nos movemos hacia atrás en T' mediante el mapeo LF hasta llegar al primer símbolo del código de $T[r+1]$. Este procesto toma a lo más $O\left(\frac{1}{\epsilon}(H_0^{(k)}+1)\log n\right)$ pasos. Luego se continúa hacia atrás en T' recolectando los símbolos de los códigos de Huffman hasta llegar al primer símbolo de $T[l]$. Esta cadena de símbolos se decodifica para obtener $T[l \dots r]$. Para moverse hacia atrás en T', utilizamos el mapeo $LF : i = C[B[i]] + Occ(B, B[i], i)$. La figura 6.3 muestra este algoritmo en pseudocódigo.

El tiempo que toma este algoritmo es $O((H_0^{(k)}+1)(L+\frac{1}{\epsilon}\log n))$ en promedio y $O(L\log\sigma + (H_0^{(k)}+1)\frac{1}{\epsilon}\log n)$ en el peor caso.

Algoritmo FM-Huffman_Display (l, r, B, Bh, S, ST, TS)
1. $j = min\{k, ST[rank(S, rank(Bh, TS[k]))] > r$ //búsqueda binaria
2. $i = TS[j]$
3. $p = ST[rank(S, rank(Bh, i))]$
4. $L = \langle\ \rangle$
5. **while** $p \geq l$ **do**
6. **do** $c = B[i]$
7. $L = c \cdot L$
8. **if** $c = 0$ **then** $i = Occ(B, c, i) + [i < p_{\#}]$
9. **else** $i = C[c] + Occ(B, c, i)$
10. **while** $Bh[i] = 0$
11. $p = p - 1$
12. decodificar los primeros $r - l + 1$ caracteres de L

Fig. 6.3: Algoritmo para extraer $T[l \dots r]$.

6.6. Implementación

Al igual que la implementación del Índice FM Huffman original, esta implementación difiere en algunos aspectos de la estructura teórica del índice. El principal aspecto en el cual la implementación no sigue la estructura teórica es en la función *Rank*. Recordemos

que para calcular en tiempo constante la función *Rank* dividimos los arreglos de bits en bloques de largo $b = \lfloor \log(n)/2 \rfloor$ y estos bloques se agrupan en superbloques de largo $s = b\lfloor \log n \rfloor$. Para cada superbloque almacenamos el número de bits en 1 hasta el comienzo del superbloque en un arreglo R_s. Para cada bloque, almacenamos el número de bits en 1 hasta el comienzo del bloque, desde comienzo del superbloque correspondiente en un arreglo R_b. Además, almacenamos el arreglo R_p con los valores precalculados de *Rank* para cada posible cadena de largo b y para cada posición dentro de ella. En el caso de las estructuras calculadas para Bh, los arreglos son idénticos a los del Índice FM Huffman original, es decir, se calculan para contar la cantidad de unos en un arreglo de bits. Tal como se hizo con la implementación del índice original, la estructura R_p no se utiliza. En cambio, se utiliza una tabla con valores precalculados de *Rank* para todas las cadenas de 8 bits. Esta tabla se utiliza para calcular el número de 1's en una cadena de 32 bits, sumando los 1's en las cuatro cadenas de 8 bits de ella. Se eligen entonces los bloques de largo $b = 32$. Los superbloques se eligen de largo $s = 256$ bits.

En el caso de las estructuras para el arreglo B, ya no se necesita saber cuántos unos hay hasta cierta posición, sino que interesa saber cuántas ocurrencias de un determinado símbolo de codificación hay hasta una posición dada. En teoría, para cada símbolo c de la codificación se calcula los arreglos $R_s[c]$, $R_s[c]$ y $R_p[c]$, que juegan los mismos papeles que los arreglos para *Rank* de Bh, pero que almacenan las ocurrencias del símbolo c hasta ciertas posiciones. La idea es que si la cadena B tiene n símbolos, los bloques serán de largo $b = \lfloor \log_k(n)/2 \rfloor$. Entonces, deberíamos almacenar en $R_p[c]$, las ocurrencias de c para cada cadena posible de b símbolos y para cada posición dentro de ella. Sin embargo, al igual que con las estructuras de *Rank* para Bh, la estructura R_p no se utiliza. En esta implementación, al igual que en la versión original del índice, se eligen los bloques de largo $b = 32$ símbolos y los superbloques de largo $s = 256$ símbolos.

En vez de utilizar el arreglo R_p para cada símbolo, se calculan tablas para cada símbolo que almacenan el número de ocurrencias del símbolo en cadenas de un largo menor a b símbolos, dependiendo este largo del valor de k. Por ejemplo, para $k = 4$, se calculan tablas que almacenan el número de ocurrencias de cada símbolo en cadenas de 8 bits. Para calcular el número de símbolos en un bloque de 32 símbolos, dependiendo de la posición relativa dentro del bloque, tendremos que calcular el número de símbolos en una o las dos palabras de 16 símbolos que componen el bloque. Se utilizan entonces estas tablas precalculadas para este propósito. En general, cuando el número de bits que ocupa un símbolo, es decir, $\log k$, es potencia de dos, el reemplazo de R_p puede hacerse con tablas que dividan una palabra de máquina en partes iguales. Por ejemplo, para $k = 16$, se pueden seguir utilizando tablas que cuenten el número de ocurrencias de cada símbolo en cadenas de 8 bits. Cuando k no es de esta forma, por ejemplo, para $k = 8$,

la solución debe ser distinta. Una cadena de 8 bits ya no contiene un número entero de símbolos, pues cada uno de ellos ocupa 3 bits, por lo que hay que pensar en largos distintos para las tablas *popc* de cada símbolo. Podríamos por ejemplo precalcular las ocurrencias de cada símbolo para cadenas de 9 bits, pero ya no podríamos hacer los cálculos de *rank* en un bloque tan directamente, pues tendríamos que manejar los bits de cada una de las tres palabras de máquina que conforman el bloque de manera distina. Entonces, para los casos en que $\log k$ no es potencia de dos, el conteo de ocurrencias de un número de símbolos en un bloque se hace más lento.

Al igual que para el caso de $k = 2$, para calcular $Occ(B, c, i)$, se suman las ocurrencias almacenadas en los arreglos R_s y R_b correspondientes a los superbloques y bloques donde se encuentra i, y luego, utilizando las tablas precalculadas, se suman las ocurrencias de c en las cadenas de 32 símbolos.

Veamos el espacio que ocupan los arreglos de $Rank$ para un símbolo c. Si tenemos un arreglo de bits B de n' símbolos, teniendo bloques de 32 símbolos necesitaremos $(n'/256)W = n'/8$ bits para R_s y $(n'/256) * 8 * 8 = n'/4$ bits para R_b. El espacio extra es de $n'/8 + n'/4 = 3n'/8$ bits, pero como ahora B ocupa $n' \cdot \log k$ bits, la fracción de B que ocupan estas estructuras es $|Rank[c](B)|/|B| = (3n'/8)/(n' \log k) = 3/(8 \log k)$, lo cual disminuye a medida que aumenta k. Sin embargo, como se necesitan k arreglos R_s y R_b, el espacio extra ocupado en total por la estructura es $3k/(8 \log k)$. Por ejemplo, para $k = 4$ se necesita un espacio extra de un 75 %, mientras que para $k = 8$ el espacio ocupado por estas estructuras es igual al espacio ocupado por B. Para $k = 16$, el espacio es de un 150 %. Esto nos da una idea de lo que ocurre con el tamaño total del índice a medida que aumentamos k. Es importante mencionar que para $k = 2$ no se aplica esta fórmula, ya que este caso se trata de manera especial, utilizando sólo un arreglo para $Rank$ para contar el número de 1's en B, y no uno para cada uno de los símbolos 0 y 1. El espacio total de las estructuras de $Rank$ para el Índice FM Huffman k-ario, incluyendo las de Bh, implican un sobrecosto de $3(k+1)/(8 \log k)$. El espacio total del índice, es decir, los arreglos B, Bh y sus estructuras para $Rank$, resulta ser $(1 + 1/\log k + 3(k+1)/(8 \log k))$ veces el espacio que ocupa el texto comprimido $(n' \log k)$.

Se podría esperar que si aumenta el número de símbolos, el tiempo de búsqueda se reduzca a una razón de $\log k$. Sin embargo, el cálculo de Occ va disminuyendo su eficiencia pues, como ya vimos, el cálculo del número de símbolos dentro de un bloque se hace más lento, al tener que accesarse más de una palabra de máquina. El acceso a R_s y R_b sigue siendo constante, pero ahora se necesitan en el peor caso $\log k$ accesos a una palabra de máquina para el conteo dentro de los bloques. Esto pasa pues para tener bloques de 32 símbolos se necesitan $32 \log k$ bits, es decir, $\log k$ palabras de máquina. Entonces, la complejidad teórica de la búsqueda cambia de $O(m(H_0^{(k)} + 1))$ a $O(m \log k(H_0^{(k)} + 1)) =$

$O(m(H_0 + \log k))$, lo cual aumenta mientras k crece. Sin embargo, hay otras razones por las cuales el tiempo mejora cuando se aumenta k, como por ejemplo que el tiempo de acceso a R_s y R_b se mantiene igual, y al disminuir el número de llamadas a Occ, dismimuye el tiempo total. Esto muestra que no se puede esperar un mejora inmediata del tiempo de búsqueda cuando se aumenta k.

Para evitar el tiempo de acceso a un bloque de $\log k$ palabras de máquina, podríamos entonces definir los tamaños de los bloques de manera que ocupen 32 bits en vez de 32 símbolos, reduciendo así este tiempo. El problema es que el espacio ocupado por los arreglos R_s y R_b aumentaría. Por ejemplo, para $k = 4$, si se tienen bloques de largo 32 bits, es decir, 16 símbolos, y superbloques de 8 bloques, cada arreglo R_s necesitaría el doble de entradas, y por lo tanto ocuparía $n'/4$ bits, mientras que cada arreglo R_b ocuparía $n'/2$ bits, haciendo un total de $3n'/4$ bits, es decir, 3/8 veces el espacio ocupado por B, lo que multiplicado por los 4 símbolos significaría un 150 % de espacio extra. Entonces, no vale la pena aplicar esta estrategia, pues la mejora en velocidad no justifica tal aumento de espacio.

Capítulo 7

RESULTADOS Y DISCUSIÓN

En este capítulo se muestran los resultados obtenidos de los experimentos realizados con el índice. Se comparó la eficiencia del Índice FM Huffman, en sus dos versiones, con la de implementaciones existentes de otros índices, tanto en el conteo de ocurrencias, reporte de posiciones y muestra de una subcadena del texto.

Los índices que se utilizaron en la comparación fueron el Índice FM[1] [8] (implementación de Gonzalo Navarro [16]), Índice LZ[2] [16], CSA[3] [18] y RLFM[4] [22][5]. Además de estos índices, existen implementaciones del Índice CCSA [20], el Compact SA [28] (ambos son variaciones del CSA), y la versión original del Índice FM [8]. El primero no se utilizó porque es inferior al CSA. El segundo se descartó porque ocupa demasiado espacio y el último ocupa muy poco espacio, pero resulta demasiado lento, por lo que no resulta comparable al Índice FM Huffman.

Se consideraron tres archivos de distinta naturaleza, uno de 80 Mb de texto en inglés, uno de 60 Mb con secuencias de ADN y uno de 55 Mb de secuencias de proteínas. El archivo de texto en inglés fue obtenido de la colección *TREC-3*[6] (Text Retrieval Conference), más específicamente de los archivos `WSJ87-89`. Los archivos de ADN y proteínas fueron obtenidos de las base de datos *BLAST* del *NCBI*[7] (National Center for Biotechnology Information) de Estados Unidos. La secuencia de ADN de 60 Mb fue obtenida del archivo `month.est_others` y la secuencia de 55Mb de proteínas fue obtenida del archivo `swissprot`[8]. Los patrones para las búsquedas fueron elegidos aleatoriamente de cada archivo.

Los experimentos se llevaron a cabo en un computador con 4 procesadores Intel(R)

[1] http://www.dcc.uchile.cl/~gnavarro/software/fmindex.tar.gz
[2] http://www.dcc.uchile.cl/~gnavarro/software/lzindex-1.1.tar.gz
[3] http://www.dcc.uchile.cl/~gnavarro/software/sada.tar.gz
[4] http://www.cs.helsinki.fi/u/vmakinen/software/rlfm.zip
[5] Los códigos fuente, incluidos los de ambas versiones del FM Huffman se encuentran disponibles en http://www.dcc.uchile.cl/~asalinge/codigoIndices
[6] http://trec.nist.gov/
[7] http://www.ncbi.nlm.nih.gov/
[8] ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/

Xeon(TM) CPU 3,06GHz con 512 Kb de cache y 2Gb RAM con el sistema operativo Linux versión 2.6.10-gentoo-r4. La implementación se realizó en el lenguaje C con el compilador gcc versión 3.4.2 (Gentoo Linux 3.4.2-r2, ssp-3.4.1-1, pie-8.7.6.5), utilizando la opción de optimización -O9.

A cotinuación se presentan algunas consideraciones sobre el espacio ocupado por el Índice FM Huffman. Luego se muestran los resultados de tiempos de los experimentos clasificados según tipo de búsqueda.

7.1. Espacio del índice

Para los experimentos realizados se consideraron las versiones del Índice FM Huffman para $k = 2$ y $k = 4$. Es importante señalar que sólo tiene sentido utilizar valores de k que son potencias de dos. La razón de esto radica en que, al codificarse cada símbolo con $\lceil \log k \rceil$ bits, si utilizáramos un valor de k que no es potencia de dos, estaríamos utilizando la misma cantidad de bits para codificar un símbolo que si hubiéramos elegido como número de símbolos la potencia de dos que sucede a k. Entonces, estaríamos desperdiciando la posibilidad de utilizar más símbolos y así lograr una mejor compresión y por consiguiente, un menor espacio.

Si bien por razones de tiempo no se realizaron experimentos con mayores valores de k, resulta interesante saber cómo se comporta el espacio ocupado por el índice cuando se varía este parámetro. La tabla 7.1 muestra los espacios como fracción del texto para distintos valores de k, para los tres archivos. Estos valores constituyen los requerimientos de espacio de este índice para el conteo de ocurrencias y no incluyen las estructuras necesarias para reportar posiciones y mostrar texto. Es importante señalar que no tiene sentido utilizar un valor de k mayor al tamaño del alfabeto para codificar un texto (incluyendo el terminador #).

Podemos ver que los espacios con $k = 4$ resultan ser los menores en los tres casos. Luego de $k = 4$ el espacio de los índices aumenta, aunque manteniéndose en valores razonables hasta $k = 16$. Con $k = 32$ y $k = 64$ los índices ocupan demasiado espacio como para ser comparables al resto. Si bien para $k = 8$ el espacio ocupado por el índice resulta totalmente práctico, probablemente los tiempos con este valor no se reduzcan con respecto a la versión con $k = 4$, pues, como vimos en la sección 6.6, la función *Occ* resulta más lenta, ya al contar las ocurrencias de los símbolos en un bloque, las cadenas utilizadas para las tablas con valores precalculados ya no están alineadas con las palabras de máquina, por lo que se necesita hacer un manejo extra de los bits dentro de ellas y por lo tanto, el conteo se hace menos eficiente. Para $k = 16$ el tiempo sí podría reducirse

k	Fracción del texto		
	Inglés	ADN	Proteínas
2	1,68	0,76	1,45
4	1,52	0,74	1,30
8	1,60	0,91	1,43
16	1,84	—	1,57
32	2,67	—	1,92
64	3,96	—	—

Tab. 7.1: Espacio ocupado como fracción del texto por el Índice FM Huffman k-ario para distintos valores de k. El valor correspondiente a la fila $k = 8$ para ADN corresponde realmente a $k = 5$, pues este es el total de símbolos a codificar para este archivo. A su vez, el valor de la fila $k = 32$ para el archivo de proteínas corresponde en realidad a $k = 24$, pues este es el total de símbolos a codificar.

pues nuevamente podemos sacar ventaja de las tablas precalculadas para cadenas de 8 bits para el conteo de ocurrencias dentro de un bloque. Al estar estas cadenas alineadas con las palabras de máquina, el acceso a ellas resulta eficiente. Sin embargo, el conteo de ocurrencias en un bloque se hace más lento pues se tendrían que accesar en el peor caso 4 palabras de máquina. Es importante notar que este valor de k sólo es relevante para el texto en inglés y proteínas, pues para el texto de ADN no tiene sentido.

Resulta también interesante ver la relación entre los espacios de cada estructura para los valores de $k = 2$ y $k = 4$ y las razones de las diferencias entre ellos. La tabla 7.2 muestra los espacios utilizados por cada una de las estructuras de ambos índices para los tres archivos considerados.

Podemos ver que el espacio ocupado por el índice con $k = 4$ es menor al ocupado por el índice con $k = 2$. Esto se debe a que se reduce el espacio ocupado por la estructura Bh, pues para $k = 4$, el texto comprimido tiene un largo casi igual a la mitad que el texto comprimido para $k = 4$. Por consiguiente, la estructura de *Rank* para Bh también reduce su espacio, en la misma proporción que lo hace Bh. El espacio ocupado por la estructura B se mantiene casi igual para ambos casos, sin embargo, el espacio ocupado por las estructuras de *Rank* para B ocupa aproximadamente el doble de espacio en el caso de $k = 4$. Esto se debe a que si bien B tiene aproximadamente la mitad de símbolos que para $k = 2$, se necesitan ahora arreglos R_s y R_b para los cuatro símbolos de codificación, siendo que antes se necesitaban esos arreglos para un sólo símbolo.

Para valores mayores de k, lo que ocurre es que el espacio de B aumenta pues al utilizar más símbolos la compresión se hace peor. A su vez, Bh disminuye a una razón de aproximadamente $\log k$, misma razón a la que disminuyen sus estructuras de *Rank*. Sin embargo, el espacio de las estructuras de *Rank* de B aumenta más rápidamente,

	Índice FM Huffman $k = 2$			**Índice FM Huffman** $k = 4$		
Estructura	Espacio [Mb]			Espacio [Mb]		
	Inglés	ADN	Proteínas	Inglés	ADN	Proteínas
B	48,98	16,59	29,27	49,81	18,17	29,60
Bh	48,98	16,59	29,27	24,91	9,09	14,80
$Rank(B)$	18,37	6,22	10,97	37,36	13,63	22,20
$Rank(Bh)$	18,37	6,22	10,97	9,34	3,41	5,55
Total	134,69	45,61	80,48	121,41	44,30	72,15
Texto	80,00	60,00	55,53	80,00	60,00	55,53
Fracción	1,68	0,76	1,45	1,52	0,74	1,30

Tab. 7.2: Espacio ocupado por las estructuras de los índices FM Huffman con 2 y 4 símbolos de codificación para los textos de inglés, ADN y proteínas. No se incluyen los espacios ocupados por las tablas precalculadas de *Rank*, las tablas de Huffman, ni el arreglo C para el Huffman k-ario, pues resulta despreciable con respecto a las demás estructuras.

pues se necesitan k estructuras para un arreglo cuyo número de símbolos disminuye a una razón de $\log k$. Este aumento de espacio es el que determina que para valores altos de k, el espacio total resulte tan alto.

7.2. Conteo de ocurrencias

En esta sección se muestran los resultados obtenidos a partir del conteo de ocurrencias ejecutado con cada índice para los tres archivos considerados.

Se muestra el tiempo de búsqueda para cada índice en función del largo del patrón a buscar. El largo se varió de 10 a 100 y para cada largo se buscaron 1000 patrones distintos elegidos aleatoriamente. Cada búsqueda se repitió 1000 veces. Con esta muestra, con el Índice FM Huffman se obtuvo un error promedio de un 1,9 % con un 95 % de confianza para el texto de proteínas, un 3,1 % para ADN, y un 2,7 % para el texto en inglés.

La figura 7.1 muestra los tiempos de búsqueda de un patrón en función de su largo, para cada índice, para cada uno de los tres archivos sobre los cuales se construyeron los índices. El Ínidce CSA requiere de un parámetro de espacio para el conteo de ocurrencias. El valor de este parámetro que se eligió para la comparación con los demás índices fue el que determinara un espacio utilizado cercano al ocupado por el Índice FM Huffman.

Se muestra también el tiempo promedio de búsqueda por carácter buscado en función del tamaño del índice. Este tamaño es el mínimo que necesita cada índice para poder operar con la funcionalidad de conteo de ocurrencias. A diferencia del CSA, los demás índices no necesitan de un parámetro que determine su tamaño si es que sólo se pretende

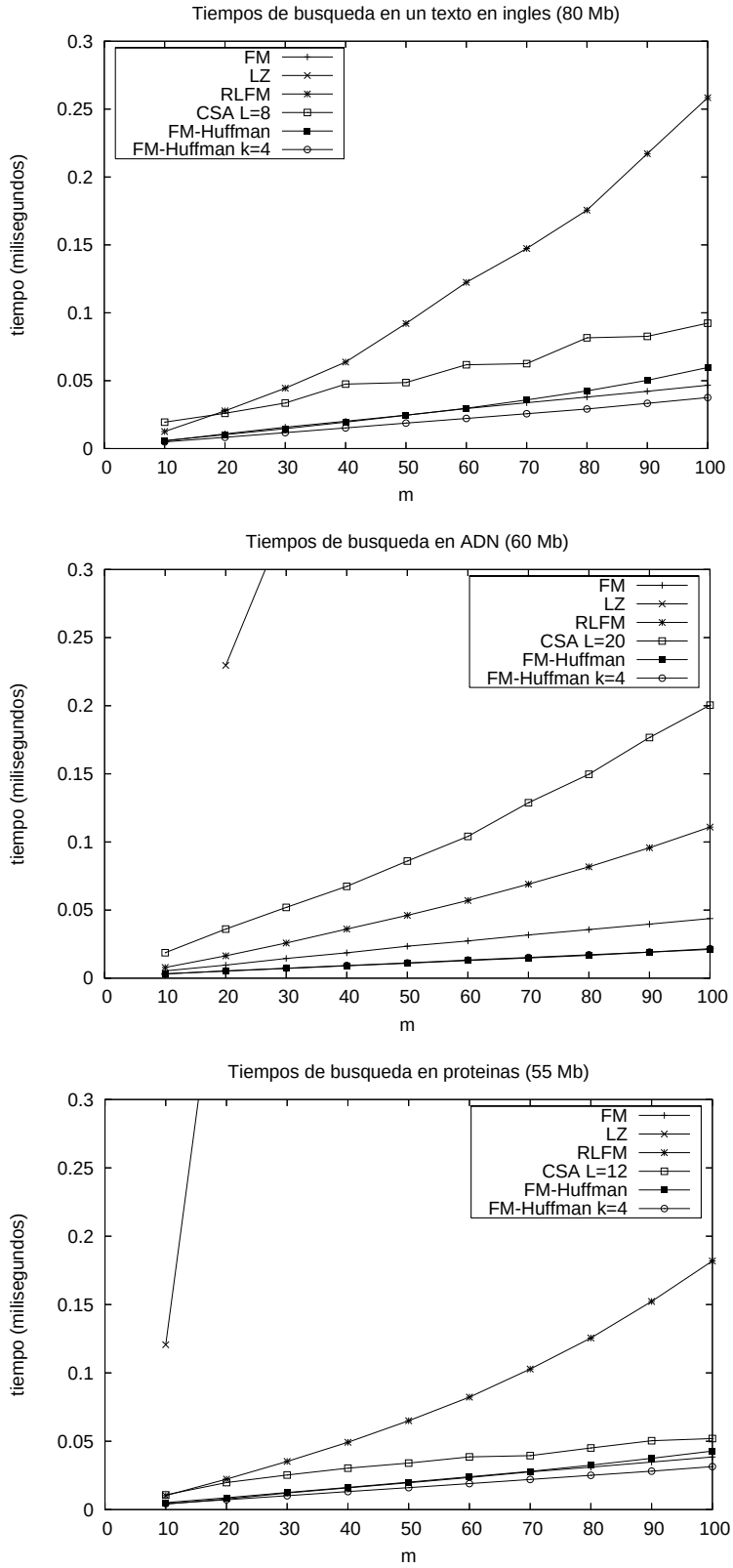


Fig. 7.1: Tiempo de búsqueda en función del largo del patrón sobre un archivo de texto en inglés de 80 Mb, un archivo de secuencias de ADN de 60 Mb, y un archivo de secuencias de proteínas de 55 Mb. El el gráfico correspondiente al texto en inglés, los tiempos del Índice LZ no alcanzan a aparecer, los valores registrados para este índice están en el rango $0{,}5 - 4{,}6$ milisegundos. En el gráfico correspondiente a ADN, el tiempo del Índice LZ para $m = 10$ es 0,26. Este aumento se debe al gran número de ocurrencias de estos patrones, lo que influye en el tiempo de conteo en este índice.

contar las ocurrencias de un patrón. Entonces, cada gráfico muestra un punto como valor del tiempo de búsqueda por carácter y tamaño del índice. Para el índice CSA se muestra una línea, pues para este índice sí se puede especificar un parámetro que implica un compromiso entre tiempo y espacio. El tiempo por carácter para cada largo de patrón es el tiempo de búsqueda dividido por el largo del patrón. El tiempo por carácter que se muestra en el gráfico es el promedio de los tiempos para cada largo.

La figura 7.2 muestra esta información para los tres archivos considerados.

Se puede observar que para los tres archivos, el índice que registró los menores tiempos de conteo de ocurrencias fue el Índice FM Huffman con $k = 4$. En el caso del archivo de ADN, el tiempo de este último fue igual al del Índice FM Huffman original. En el caso del archivo de texto en inglés y proteínas, el segundo mejor tiempo lo registró el Índice FM, siguiéndolo el FM Huffman original, con una diferencia entre ambos muy pequeña y que se nota sólo para los patrones de largo mayor que 70. Para el caso de ADN, el índice que registró el siguiente menor tiempo fue el Índice RLFM, mientras que para los dos otros archivos, resultó mejor el índice CSA. En los tres casos, el índice LZ no resulta competitivo para el conteo de ocurrencias, pues los tiempos que registró resultaron ser más de diez veces mayores que los de los demás índices.

En la figura 7.2, para el texto en inglés, se puede observar que el Índice FM ocupa un espacio de menos de 1,1 veces el texto, con un tiempo de búsqueda por carácter que resulta muy competitivo con respecto al resto de los índices. El menor tiempo lo registra el Índice FM Huffman 4-ario, sin embargo, el espacio ocupado por éste es casi un tercio más que el ocupado por el Índice FM. Ocupando más espacio y con un tiempo muy parecido al del Índice FM, está el Índice FM Huffman con $k = 2$. Se puede observar también que el Índice CSA y el RLFM, con espacios cercanos a 0,8 veces el tamaño del texto, registran tiempos mayores que los índices ya mencionados, sin embargo, el poco espacio que ocupan los hace una alternativa a considerar.

Algo parecido a lo que ocurre con el texto en inglés ocurre con la secuencia de proteínas. Nuevamente el Índice FM, aunque registra un tiempo de búsqueda por carácter mayor al del índice FM Huffman con $k = 4$, ocupa menos espacio que éste, por lo que puede considerarse como alternativa. Sin embargo, el espacio del Índice FM está acotado inferiormente por el espacio que ocupa el texto, pues este índice, en su implementación, trabaja con el texto completo. Esto constituye una desventaja con respecto a los demás índices cuando se trata de textos sobre alfabetos pequeños. Por ejemplo, para el archivo de ADN, se puede observar que los demás índices ocupan menos espacio que el espacio del texto. En este caso, los índices RLFM y CSA pueden alcanzar un espacio muy reducido, sin embargo, sus tiempos de búsqueda por carácter son mayores que los de los

índices FM Huffman con $k = 2$ y $k = 4$, los que de todas maneras ocupan un espacio reducido. Esto hace a estos dos últimos excelentes alternativas para el caso de este archivo. Para los tres archivos, los tiempos de búsqueda por carácter registrados por el Índice LZ resultaron demasiado altos y ni siquiera aparecen en los gráficos.

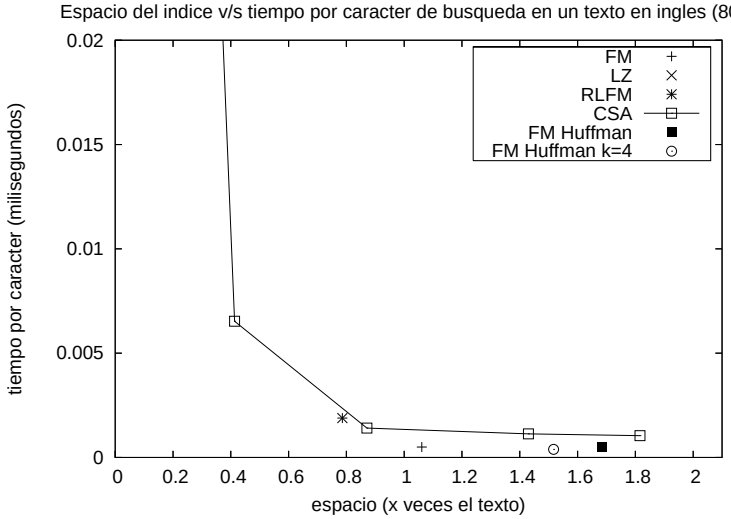
Se puede observar que los tiempos registrados con el Índice FM Huffman que ocupa 4 símbolos en la codificación resultan menores que los tiempos registrados por el Índice FM Huffman original. En teoría, al utilizar 4 símbolos en la codificación, el tiempo debería reducirse a la mitad, pues se reduce a la mitad el número de pasos a realizar en la búsqueda. Sin embargo, el tiempo se redujo, pero no en esa magnitud. Esto puede deberse a que, como vimos en la sección 6.6, la función *Occ* no resulta ser tan rápida como la función *Rank* del FM Huffman original, pues la función que utiliza este último para reemplazar a la tabla precalculada de Rank R_p es más eficiente que la que utiliza el Índice FM Huffman k-ario. Esto se debe a que este último, al utilizar bloques de 32 símbolos de dos bits cada uno, necesita accesar muchas veces a dos palabras de máquina para contar los símbolos en un bloque. En el caso del Índice FM Huffman con $k = 2$, sólo se necesita accesar una palabra. Entonces, aunque el número de pasos en la búsqueda se reduzca a la mitad, en cada paso el tiempo que toma el Índice FM Huffman 4-ario es mayor que el del índice binario. De hecho, podemos observar que para el texto de ADN, los tiempos de ambos índices son casi iguales.

7.3. Reporte de posiciones

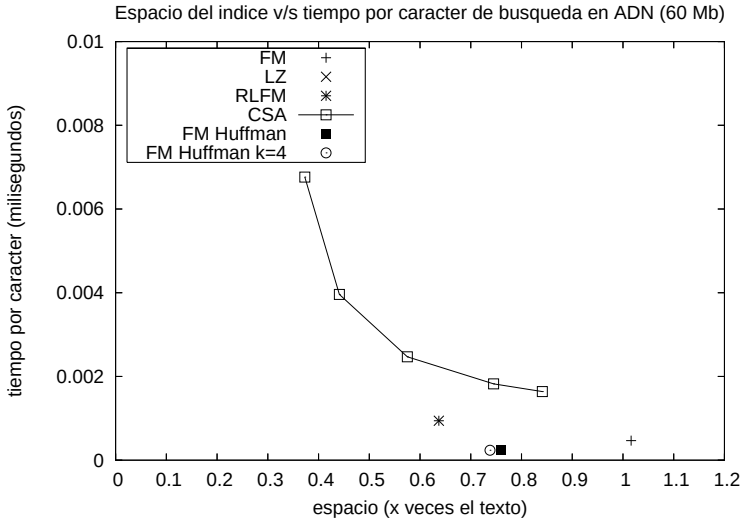
El segundo experimento realizado consistió en medir el tiempo que se demora cada índice en reportar las posiciones de las ocurrencias encontradas.

Sobre los primeros dos archivos, vale decir, texto en inglés, y secuencias de ADN, se extrajeron 1000 patrones aleatorios de largo 10. Para el archivo de proteínas, se utilizaron 1000 patrones aleatorios de largo 5, para así obtener un número significativo de ocurrencias por patrón. Se realizaron las búsquedas de estos patrones y se midió el tiempo de cada índice por ocurrencia reportada. Cabe mencionar que con este número de patrones, se obtuvo un error de un 2,2 % con un 95 % de confianza con el Índice FM Huffman, para proteínas, un 5,1 % para ADN y un 19 % para inglés.

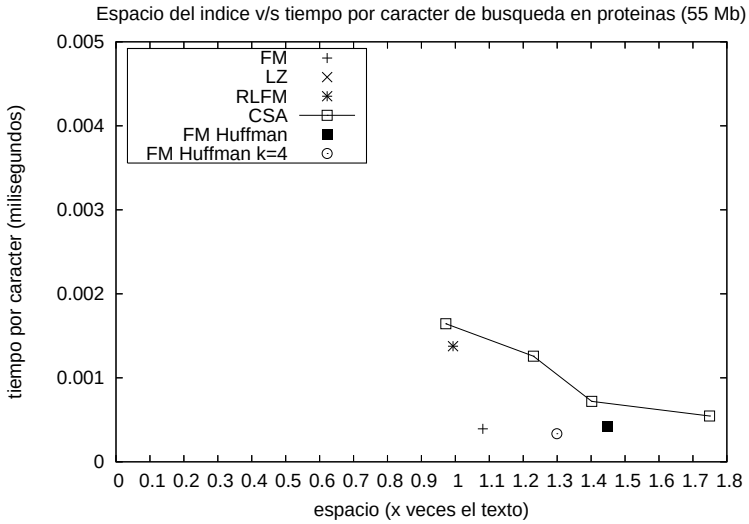
Para cada índice se especifica un parámetro que indica el espacio extra ocupado para el reporte de posiciones y muestra de texto. Este parámetro implica un compromiso entre tiempo y espacio. Para cada índice se tomaron las medidas con distintos valores para este parámetro. Una excepción a esto es el Índice LZ, el cual no tiene un parámetro para ajustar el espacio que ocupa. Por otra parte, el Índice CSA posee un parámetro



El punto correspondiente al Índice LZ es: espacio=1,42, tiempo=0,045.



El punto correspondiente al Índice LZ es: espacio=1,18, tiempo=0,014.



El punto correspondiente al Índice LZ es: espacio=2,44, tiempo=0,028.

Fig. 7.2: Tiempo de búsqueda promedio por carácter en función del tamaño del índice. De arriba hacia abajo, se muestran los resultados de buscar sobre el archivo de texto en inglés de 80 Mb, el archivo de secuencias de ADN de 60 Mb, y el archivo de secuencias de proteínas de 55 Mb.

para el conteo de ocurrencias. Para estos experimentos, se utilizó el mismo valor para este parámetro que para el que determina el espacio extra ocupado para reportar las posiciones, pues eso resulta ser lo óptimo.

La figura 7.3 muestra para los tres archivos, los tiempos por ocurrencia reportada de cada índice en función del tamaño del índice.

Se puede observar que para reportar las ocurrencias en el texto en inglés el índice que registró los menores tiempos fue el Índice FM. Para un mismo espacio ocupado, este índice registra menores tiempos que los índices RLFM, CSA y LZ. Los índices FM Huffman binario y 4-ario registran tiempos similares a los del Índice FM, pero utilizando más espacio que éste. Se puede observar que para registrar tiempos similares, el espacio que ocupa el Índice FM Huffman con $k = 4$ es menor que el que ocupa la versión con $k = 2$. Vemos que el Índice FM Huffman con $k = 4$ resulta mejor que los índices RLFM y CSA para un mismo espacio una vez que los índices ocupan más de 2 veces el tamaño del texto. Esto ocurre a partir de un espacio igual a 2,3 veces el tamaño del texto para el índice con $k = 2$. Para espacios muy pequeños, el índice que resulta mejor, luego del FM, es el CSA.

Se tiene una situación distinta para el archivo de ADN. En este caso, se puede observar que las dos versiones del Índice FM Huffman ocupan espacios similares a los de los demás índices y registran ambos mejores tiempos. Para un mismo espacio, el Índice FM Huffman con $k = 4$ registra tiempos algo mejores que los de la versión binaria. A estos índices le siguen el Índice FM, el RLFM y el CSA para espacios mayores a 1,2 veces el texto. Para espacios menores a este valor, resulta mejor el Índice RLFM, aunque cuando se pasa a espacios menores al tamaño del texto, el Índice CSA registra los menores tiempos.

La reducción de espacio de los Índices FM Huffman con respecto al texto en inglés se debe a que la entropía de orden cero del texto en ADN es pequeña, por lo que el texto comprimido resulta compacto. Como las estructuras del índice ocupan un espacio proporcional al tamaño del texto comprimido, estas estructuras resultan también compactas. Esto pasa pues los tamaños de estos índices dependen de la entropía de orden cero del texto. Para los demás índices, el tamaño depende de la entropía de orden k, la cual en el caso de ADN, es muy parecida a la entropía de orden cero. Es por esto que para este archivo no existe tanta diferencia de espacio entre estos índices y los índices RLFM, LZ y FM como existía para el texto en inglés. El caso del Índice CSA es interesante. Su tamaño también depende de la entropía de orden cero del texto, sin embargo no se aprecia una reducción en el espacio ocupado con el archivo de ADN con respecto al espacio con el texto en inglés. Esta reducción sí puede apreciarse en el gráfico de la

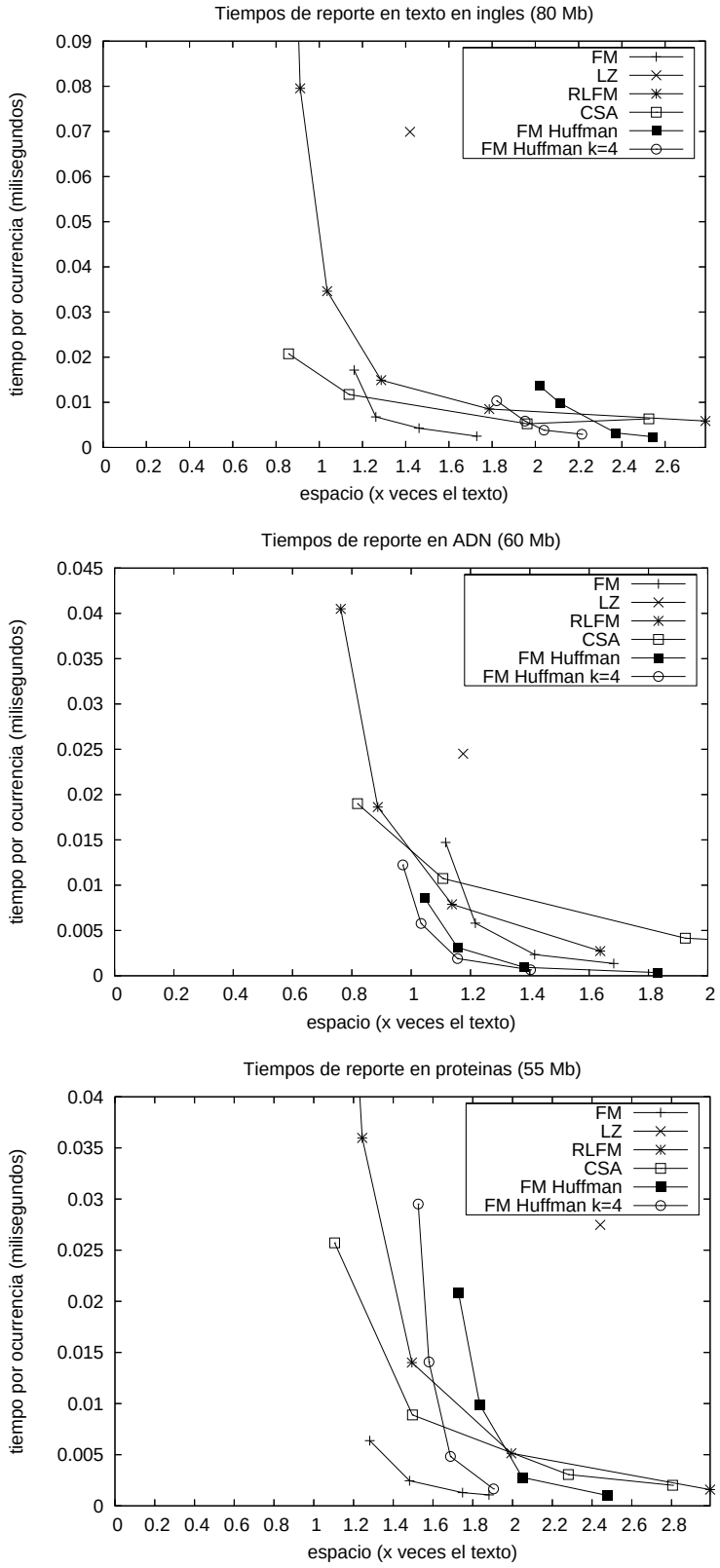


Fig. 7.3: Tiempo de reporte de posiciones de las ocurrencias en función del tamaño del índice. De arriba hacia abajo, se muestran los resultados de buscar sobre el archivo de texto en inglés de 80 Mb, el archivo de secuencias de ADN de 60 Mb y el archivo de secuencias de proteínas de 55 Mb.

figura 7.2, donde el espacio ocupado por el índice es considerablemente menor para el texto de ADN que para los textos de inglés y proteínas. La razón por la cual el CSA no reduce su tamaño construido sobre el archivo de ADN con respecto al resto de los archivos cuando se incluyen las estructuras para reportar es que los intervalos para la muestra son pequeños, por lo que no permiten que se alcance una mejor compresión. Si se utilizaran intervalos más largos, sí se alcanzaría una mejor compresión y por lo tanto un espacio ocupado menor, pero a la vez, los tiempos resultarían demasiado altos como para que el índice sea comparable con el Índice FM Huffman.

Con respecto al tiempo, en los índices FM Huffman, para reportar las posiciones se utiliza el mapeo LF, el cual nos lleva símbolo por símbolo en la estructura B. Como la entropía de orden cero del texto es pequeña, los códigos de Huffman resultan de un largo más corto que con los demás archivos. Mientras menores sean los códigos de Huffman, menos símbolos ocupa cada carácter, y por lo tanto, se ocupan menos pasos en encontrar las posiciones que están en la muestra elegida. Recordemos que cada ocurrencia se puede reportar en tiempo $O\left(\frac{1}{\epsilon}(H_0^{(k)}+1)\log n\right)$ para estos índices, lo que disminuye junto con la entropía de orden cero del texto.

Para el archivo de proteínas, al igual que con el archivo en inglés, el índice que registró los mejores tiempos fue el Índice FM. Para espacios menores que 1,6 veces el tamaño del texto, le siguen los índices CSA y RLFM, en ese orden. Para espacios mayores a ese valor, sigue al Índice FM el Índice FM Huffman con $k = 4$. El Índice FM Huffman con $k = 2$ recién supera al CSA y RLFM para espacios mayores a 2 veces el texto.

7.4. Muestra de texto

Este experimento consistió en medir el tiempo que se demora cada índice en mostrar un texto alrededor de cada ocurrencia encontrada. Para cada índice y para cada archivo de texto, se midió el tiempo que demora cada índice en encontrar el primer carácter a mostrar. Más especificamente, se midió el tiempo que le toma a cada índice buscar un patrón, encontrar cada posición de las ocurrencias y para cada una de ellas realizar todo el trabajo necesario para poder mostrar en pantalla el primer carácter de texto alrededor de la ocurrencia. Usualmente el primer carácter que se muestra es el mismo de la posición encontrada, pero esto depende de cada índice. Lo que se hizo fue tomar el tiempo hasta que el índice mostraba el primer carácter ya sea en la posición de la ocurrencia o en una cercana a él.

Recordemos que, al no utilizar el texto, los índices deben elegir una muestra de posiciones del texto. Para mostrar un carácter del texto, se debe ubicar primero una

posición de la muestra que esté cerca de la posición a mostrar y después recorrer la estructura comprimida hasta llegar a la posición deseada. Mientras más posiciones del texto estén en la muestra, menos tiempo tomará llegar a la posición que se busca. Se midió para cada índice este compromiso entre tiempo y espacio. Se utilizaron los mismos mil patrones que para el experimento de reporte de posiciones para los tres archivos. Con el Índice FM Huffman se obtuvo un error de la medida de un 1,6 % con un 95 % de confianza para proteínas, un 9 % para inglés, y un 1,9 % para ADN.

En la figura 7.4 se presentan los tiempos de muestra del primer carácter en función del espacio ocupado para cada índice y para los tres archivos.

Por otra parte, se midió el tiempo de muestra de texto por carácter mostrado. Para ello, se buscaron los 1000 patrones y se mostraron 100 caracteres alrededor de cada ocurrencia de cada patrón en el texto. Para esto, se ajustó cada índice dejando que mostrara 100 caracteres. En el caso de los índices FM Huffman, FM, y RLFM, los caracteres mostrados comienzan 50 caracteres antes de la posición de la ocurrencia. En el caso del CSA, la muestra de texto comienza en la posición de la ocurrencia. Por su parte, el Índice LZ muestra 100 caracteres dejando al final las posiciones correspondientes al patrón buscado. En este caso, el Índice FM Huffman registró un error de un $0,6\,\%$ con un 95 % de confianza.

Nos interesa sólo el tiempo de muestra por carácter, por lo que a este tiempo se le restó el tiempo que demora cada índice en encontrar el primer carácter a mostrar.

La figura 7.5 muestra, para cada archivo, los tiempos de muestra por carácter junto con los requerimientos mínimos de espacio de cada índice, es decir, el espacio necesario para el conteo de ocurrencias. Se consideran estos espacios para cada índice pues el tiempo de muestra por carácter no depende del espacio extra ocupado por cada índice para el reporte de posiciones y muestra de texto. Esto es cierto para todos los índices, menos para el Índice CSA, para el cual este tiempo sí depende del espacio. Para este índice, se muestran los tiempos para distintos requerimientos de espacio.

Se puede observar en los gráficos que para los tres archivos el Índice FM es el que registra los mejores tiempos para ubicar el primer carácter a mostrar. Para el texto en inglés, le sigue el Índice RLFM, el Índice LZ y el CSA, todo esto para espacios menores a 2 veces el tamaño del texto. Cuando el espacio es mayor, el Índice FM Huffman con $k = 4$ registra mejores tiempos que los índices recién mencionados, y para espacios mayores a 2,4 veces el texto, le sigue el Índice FM Huffman binario.

Al igual que para el reporte de posiciones, la situación es distinta cuando se trata del archivo de ADN. Esta vez las dos versiones de los índices FM Huffman ocupan comparativamente un menor espacio que para el texto en inglés, registrando a su vez menores

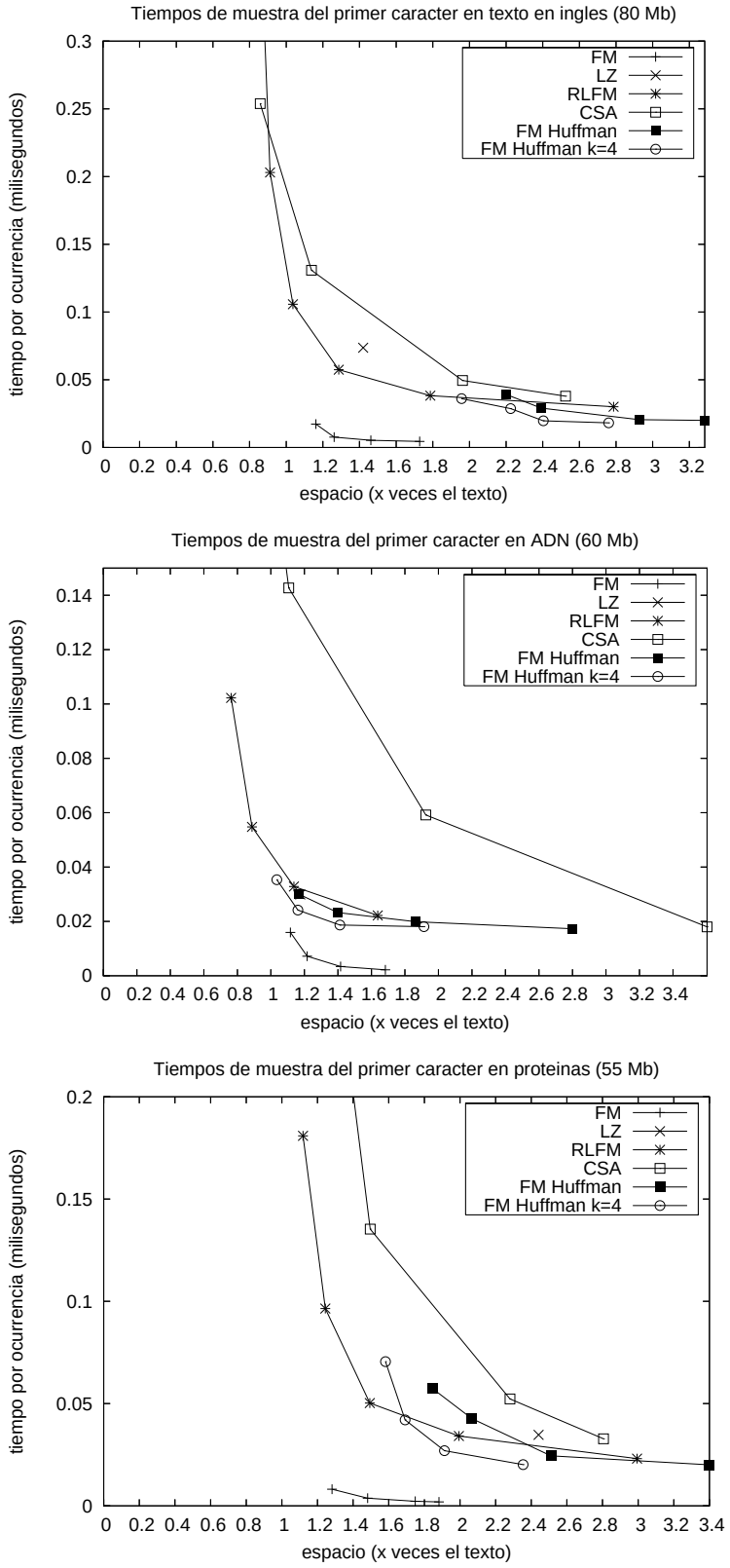


Fig. 7.4: Tiempo de muestra del primer carácter alrededor de las ocurrencias en función del tamaño del índice. De arriba hacia abajo los gráficos corresponden al texto en inglés de 80 Mb, al texto de ADN de 60 Mb y al texto de proteínas de 55 Mb. En el gráfico de ADN, el punto correspondiente al Índice LZ queda oculto por un punto del FM Huffman binario, su valor es: espacio=1,18, tiempo=0,03.

tiempos que los demás índices, a excepción del Índice FM, el cual sigue registrando mejores tiempos para igual espacio. El Índice FM Huffman con $k = 4$ resulta levemente mejor que la versión binaria, y le siguen a éste los índices LZ y RLFM. El Índice CSA registra tiempos considerablemente mayores.

Para el texto con proteínas, el Índice FM sigue siendo el más competitivo. Esta vez el espacio ocupado por los índices FM Huffman aumenta con respecto al ocupado para el archivo de ADN pero disminuye comparativamente con respecto al texto en inglés. Se observa que la versión con $k = 4$ resulta competitiva ahora desde espacios mayores a 1,7 veces el texto. En cambio, la versión con $k = 2$, se ubica después de los índices FM, FM Huffman con $k = 4$ y RLFM, superando al LZ y al CSA, aunque a partir de espacios mayores a 2,4 veces el texto, también supera al RLFM.

Es interesante ver lo que ocurre con los tiempos de muestra por carácter, pues se tienen distintos resultados dependiendo del tipo de archivo de que se trate. Para el texto en inglés, se tiene que el menor tiempo de muestra por carácter lo registra el Índice LZ, seguido de cerca por el Índice FM, aunque este último ocupa un menor espacio, por lo que también resulta una buena alternativa. Con un espacio un poco mayor que estos dos índices, sigue luego el Índice FM Huffman con $k = 4$. La versión con $k = 2$ registra un tiempo mayor al del Índice CSA, para un mismo espacio. El tiempo registrado por el Índice RLFM resultó ser aproximadamente 10 veces mayor que el del Índice LZ, no siendo comparable al resto de los índices en este aspecto.

Para el texto de ADN, nuevamente los Índices FM Huffman cobran importancia, registrando la versión con $k = 4$ el menor tiempo de muestra por carácter. Le sigue el Índice LZ, aunque el siguiente, la versión binaria del FM Huffman registra un tiempo muy cercano con un espacio que resulta ser aproximadamente un tercio del ocupado por el Índice LZ. El Índice FM registra un tiempo mayor que los anteriores y un espacio mayor a las dos versiones del FM Huffman. Más atrás los índices CSA y RLFM registran tiempos considerablemente mayores.

En el caso de la secuencia de proteínas se puede ver que el mejor índice resultó ser el Índice FM, seguido por el Índice FM Huffman con $k = 4$ y por el LZ, aunque este último ocupando un espacio de más del doble que los espacios ocupados por los índices anteriormente mencionados. El Índice FM Huffman binario y el CSA registran, para un mismo espacio, prácticamente el mismo tiempo. Nuevamente el Índice RLFM registra un tiempo mayor que el resto.

Como para mostrar un texto necesitamos encontrar la primera posición a mostrar y luego mostrar cada carácter, nos interesa que ambos tiempos sean lo más pequeños posible. En este sentido, el Índice FM y el Índice FM Huffman con $k = 4$ son los que

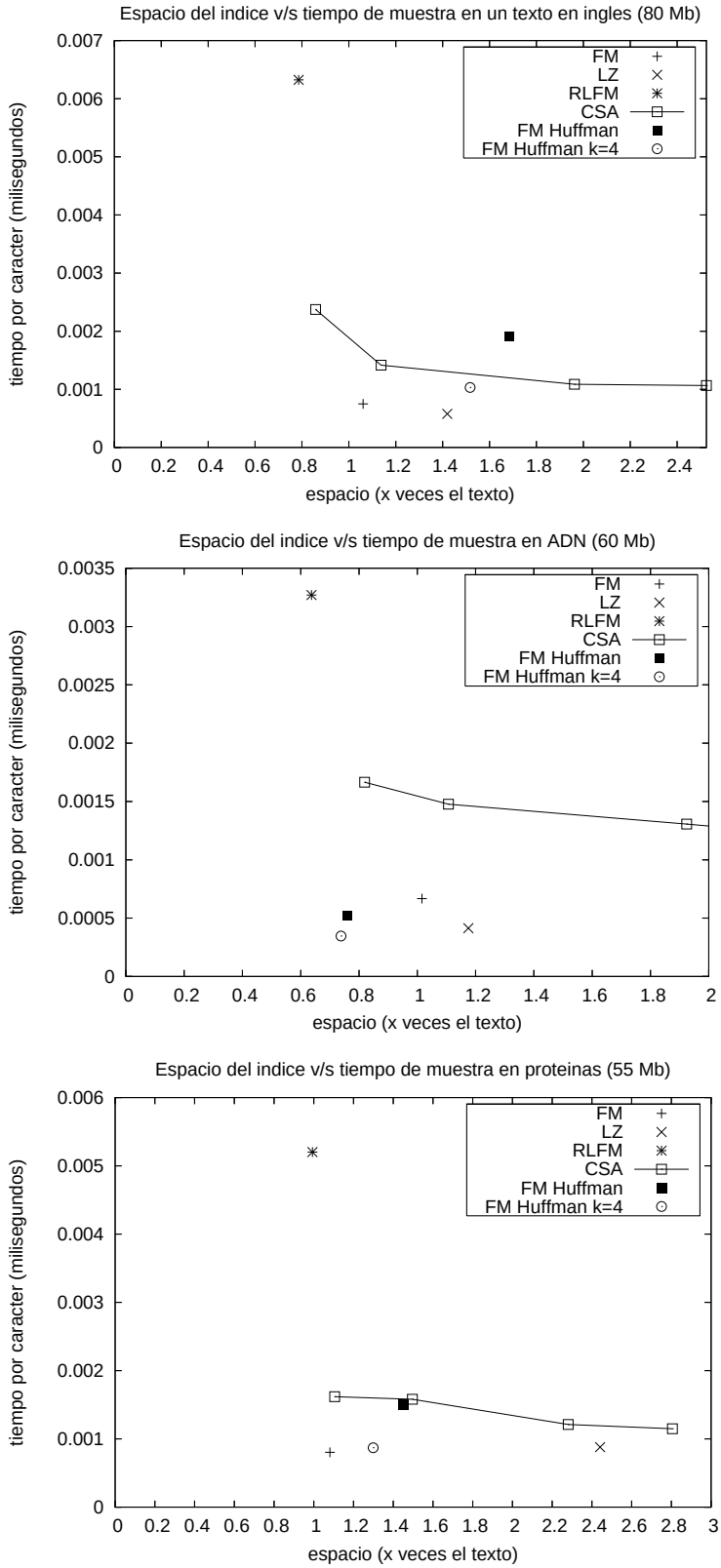


Fig. 7.5: Tiempo de muestra alrededor de las ocurrencias por carácter mostrado y espacio ocupado por cada índice, para el archivo de texto en inglés de 80 Mb, el de ADN de 60 Mb y el de proteínas de 55 Mb.

resultan más competitivos para la muestra de texto.

7.5. Resumen de resultados

Para resumir los resultados obtenidos, se mencionan los índices que registraron el mejor desempeño en cuanto a tiempo y espacio para cada experimento.

En el experimento de conteo de ocurrencias, para los archivos en inglés, ADN y proteínas, el índice que registró los mejores tiempos fue el Índice FM Huffman con $k = 4$, registrando un tiempo igual que la versión original de este índice para el texto de ADN. Para el texto en inglés y el texto de proteínas lo siguen a la par los índices FM y FM Huffman binario, resultando el primero levemente mejor.

Con respecto al compromiso entre tiempo y espacio, el Índice FM registró el mejor balance entre tiempo y espacio para los archivos en inglés y el archivo de proteínas. Para el archivo de ADN, las dos versiones del Índice FM Huffman registran los mejores tiempos de búsqueda por carácter, ocupando un espacio menor que el índice FM.

En el experimento de reporte de posiciones, para el archivo en inglés, el índice que registró los mejores tiempos fue el Índice FM, en comparación con los tiempos de los demás índices al ocupar el mismo espacio que este índice. Le siguen los índices RLFM y CSA aunque ambas versiones de los Índices FM Huffman los superan si se consideran espacios mayores. Para el archivo de ADN, el índice que registró los mejores tiempos de reporte de posiciones considerando el requerimiento de espacio fue el Índice FM Huffman con $k = 4$, seguido de cerca por la versión binaria de este índice. Para el archivo de proteínas, nuevamente el mejor tiempo lo registra el Índice FM, seguido por el CSA para espacios pequeños y por el FM Huffman con $k = 4$ a medida que aumenta el espacio.

En el experimento de muestra del primer carácter, el índice que registró los menores tiempos para los tres archivos fue el Índice FM. Además, registró estos tiempos con un requerimiento de espacio similar a los requerimientos de los demás índices. Las dos versiones del Índice FM Huffman resultaron mejor que los índices CSA y RLFM, aunque ocupando más espacio que ellos en el caso del texto en inglés, siendo la versión con $k = 4$ la mejor entre las dos. Para ADN, estas dos versiones siguen al Índice FM, ocupando el mismo espacio que los demás índices, mientras que para el texto de proteínas, los espacios ocupados por las versiones del FM Huffman aumentaron con respecto a los demás índices, manteniendo mejores tiempos que los índices CSA, RLFM y LZ.

Sobre el tiempo de muestra de cada carácter, los índices que registran los menores tiempos y espacios son el Índice FM y el FM Huffman con $k = 4$, además del LZ para

el texto en inglés y el FM Huffman binario para ADN. Para el texto en inglés el menor tiempo lo registra el Índice LZ, para el ADN, el Índice FM Huffman con $k = 4$, y para proteínas, el Índice FM.

Capítulo 8

CONCLUSIONES

En este trabajo se describe una estructura de datos para realizar búsquedas eficientes en archivos de texto. Esta estructura, el Índice FM Huffman [1], se basa en el Índice FM [8], siendo su idea principal el comprimir el texto usando la codificación de Huffman, antes de aplicar la transformación de Burrows Wheeler. El resultado es una estructura similar a la del Índice FM, pero sin la fuerte dependencia en el tamaño del alfabeto que tiene este último. Construido sobre un texto de n caracteres, el Índice FM Huffman ocupa $O(n(H_0 + 1))$ bits de espacio, y puede realizar una búsqueda de un patrón de largo m en tiempo promedio $O(m(H_0 + 1))$.

En teoría, con el Índice FM Huffman se obtienen mejores complejidades de tiempo para el reporte de ocurrencias y muestra de texto que el Índice FM y un mejor tiempo de búsqueda que el índice CSA [18]. El objetivo principal de este trabajo era implementar el Índice FM Huffman y comparar su eficiencia con respecto a la de otros índices existentes. Para ello, se realizaron una serie de experimentos para medir el desempeño de cada índice con respecto al conteo de ocurrencias, al reporte de posiciones de las ocurrencias y a la muestra de texto alrededor de ellas. En los experimentos se utilizaron archivos de texto de distinta naturaleza. Se utilizó un archivo de texto en inglés, un archivo de secuencias de ADN y otro de secuencias de proteínas. Estos archivos presentan distintos tamaños del alfabeto y distintas distribuciones de los caracteres del alfabeto. Resulta interesante ver cómo se comportan cada índice según el tipo de archivo sobre el cual se construye.

Los resultados obtenidos fueron bastante positivos. Pudimos ver que para el conteo de ocurrencias, el Índice FM Huffman es casi tan bueno como el Índice FM, resultando mejor que éste, y que todos los demás índices, en el caso del archivo de ADN.

Con respecto al espacio ocupado por el índice, pudimos ver que mientras más pequeña es la entropía de orden cero del texto, el Índice FM Huffman ocupa menos espacio. Empíricamente pudimos ver que la razón entre la fracción del espacio del texto que ocupaba el índice y la entropía del texto se mantenía relativamente constante. El texto de inglés utilizado tiene una entropía de 4,86 bits por carácter, y el espacio ocupado por

el índice fue de 1,68 veces el tamaño del texto, es decir, 13,44 bits por carácter. Esto nos da una razón de 2,76. La entropía del texto de proteínas es de 4,17, y el índice ocupa 1,45 veces el espacio ocupado por el texto — 11,6 bits por carácter —, resultando una razón de 2,78. En el caso del texto de ADN, la entropía es de 1,98, el índice ocupa 0,76 veces el tamaño del texto, o sea, 6,08 bits por carácter, para una razón de 3,07. A partir de estos datos podemos decir que el Índice FM Huffman ocupa una fracción del texto que es aproximadamente 2,8 veces su entropía de orden cero.

Entonces, como la entropía del texto de ADN es la más baja, el índice resulta más compacto para este archivo, siendo además la mejor alternativa de búsqueda, pues registra comparativamente los mejores tiempos de conteo de ocurrencias.

El experimento de reporte de posiciones nos permite ver que el requerimiento de espacio del Índice FM Huffman aumenta considerablemente con las estructuras auxiliares necesarias para esta funcionalidad, mientras que los espacios ocupados por los demás índices aumentaron en una proporción menor. Aunque el mejor tiempo para los archivos de proteínas e inglés los registró el Índice FM, el Índice FM Huffman registra buenos tiempos, mejores que los demás índices aunque ocupando un mayor espacio. Para el texto de ADN, el FM Huffman se convierte en la mejor alternativa, pues registra los menores tiempos para un mismo espacio utilizado.

Con respecto a la muestra de texto, el índice FM registró menores tiempos que todo el resto. Al igual que lo ocurrido con el reporte de posiciones, el Índice FM Huffman registró tiempos competitivos pero con un requerimiento de espacio mayor al de los demás índices, aunque para el archivo de ADN, al ocupar un espacio igual que el resto de los índices, este índice se convierte directamente en la mejor alternativa después del Índice FM.

Como parte del trabajo, se implementó una nueva versión del Índice FM Huffman, el Índice FM Huffman k-ario, el cual permite comprimir el texto con Huffman pero utilizando k símbolos en la codificación. Se midió la eficiencia para esta estructura considerando cuatro símbolos en la codificación. El resultado obtenido fue una mejora en todos los aspectos con respecto a la versión binaria del índice. La versión de este índice con $k = 4$ redujo el tiempo de búsqueda de la versión original para lo archivos de proteínas e inglés, y se mantuvo igual para ADN, convirtiéndose en la mejor alternativa para el conteo de ocurrencias. A su vez, con esta versión se redujo el espacio utilizado con respecto a la versión original. Para el reporte de posiciones, si bien se redujo el espacio y tiempo con respecto a la versión con $k = 2$, esta mejora no alcanzó para superar al Índice FM para los archivos de texto en inglés y proteínas. Para el texto de ADN, donde el FM Huffman original ya había resultado mejor que los demás índices, esta versión se convirtió en una

alternativa aún mejor. Para la muestra de texto, la versión del Índice FM Huffman con $k = 4$ también redujo los tiempos y espacio con respecto a la versión binaria, convirtiéndose en la segunda mejor alternativa para la muestra de texto después del Índice FM y registrando incluso el menor tiempo y espacio para la muestra por carácter.

Durante el desarrollo de este trabajo se llevaron a cabo los experimentos con una versión inicial del Índice FM Huffman, el cual utilizaba una función *Rank* distinta a la actual. Los resultados anteriores no habían resultado tan positivos como los actuales. La función *Rank* forma parte fundamental de los algoritmos de consulta de esta estructura, por lo que un aumento en la eficiencia de esta función conlleva a una mejora global de los tiempos de consulta. Si bien el espacio ocupado por las estructuras auxiliares de esta función aumentó de un 31 % de espacio extra a un 37 %, la reducción en tiempo fue notable, convirtiendo al Índice FM Huffman con $k = 4$ en una excelente alternativa como índice de texto completo, especialmente para archivos con bajas entropías, como son las secuencias biológicas de ADN.

Es interesante analizar las razones por las cuales estos experimentos arrojaron algunos resultados que difieren de lo esperado según los análisis teóricos. A través de estos análisis, pudimos ver que el Índice FM Huffman no obtenía mejores complejidades de tiempo que el Índice FM para la búsqueda de patrones. La complejidad del primero es $O(m(H_0 + 1))$ mientras que para el segundo es $O(m)$. Sin embargo, pudimos ver que para el conteo de ocurrencias el Índice FM y el FM Huffman con $k = 2$ resultaron casi iguales para las proteínas y el texto en inglés, mientras que este último resulto mejor para ADN. Incluso, la versión con $k = 4$ resultó mejor para los tres archivos. Esto es un ejemplo de que la teoría y la práctica pueden diferir. La razón por la que ocurrió esto en los experimentos es que se utilizó una implementación práctica del Índice FM que difiere en algunos aspectos con respecto a su explicación teórica. Por ejemplo, durante la búsqueda, el Índice FM calcula *Occ* en tiempo constante, sin embargo, en la implementación se utiliza una estructura similar a las estructuras de *Rank* utilizadas por el Índice FM Huffman, dividiendo el texto permutado en bloques y superbloques. Para calcular *Occ* se deben hacer algunas operaciones que consisten en sumar contadores del superbloque donde se encuentra la posición que nos interesa y recorrer contadores de bloques dentro de un superbloque hasta llegar al bloque relevante correspondiente a la posición. Una vez que se llega al bloque se recorre el texto permutado para sumar el total de ocurrencias de un carácter. Probablemente este cálculo contribuye a un aumento de la constante de la complejidad de búsqueda, haciendo que el Índice FM obtenga tiempos muy parecidos a los del FM Huffman con $k = 2$ y que incluso se vea superado por éste para el texto de ADN. Por otra parte, en teoría el Índice FM Huffman alcanza mejores tiempos de reporte de posiciones y muestra de texto que el Índice FM. Sin embargo,

esto se vio reflejado solamente en el reporte de posiciones con el archivo de ADN. Una vez más, esto se debe a que la implementación utilizada del Índice FM difiere de la presentación teórica de él. En teoría, el texto permutado BWT se comprime, logrando que el espacio del índice resulte mucho menor. Sin embargo, para obtener una entrada de este texto para el mapeo LF, se debe hacer un cálculo que toma un tiempo proporcional al tamaño del alfabeto. La implementación utilizada de este índice no comprime BWT, por lo que un acceso a BWT toma $O(1)$. Esto reduce la complejidad de tiempo de reporte de posiciones y muestra de texto, haciendo que este índice sea muy eficiente en la práctica.

Una idea interesante sería modificar esta implementación del Índice FM para que en vez de representar en texto permutado directamente, lo comprima con Huffman. Esta idea resulta parecida a la del FM Huffman, la diferencia principal es que el FM Huffman comprime primero el texto y después aplica la BWT, en cambio este índice lo haría al revés. Se tendrían algunas diferencias entre ambos índices en cuanto a tiempo y espacio. Con esta idea, esta variación del FM calcularía la función Occ de la misma manera que antes, necesitándose las estructuras de datos correspondientes. En este sentido, la búsqueda sería un poco más lenta que en el FM Huffman, pues la función $Rank$ resulta más eficiente que Occ. Por otra parte, esta versión no necesitaría una estructura equivalente a Bh, pero sí estructuras parecidas a las estructuras de $Rank$ del FM Huffman para el texto permutado. Si bien el tiempo de búsqueda resultaría un poco mayor con respecto al FM Huffman, el espacio requerido sería de casi la mitad, por lo que resultaría una alternaiva bastante competitiva.

Para un trabajo futuro, se propone estudiar la eficiencia del Índice FM Huffman con valores mayores de k. Pudimos ver que en teoría, los tiempos de consultan deberían disminuir a medida que k aumenta debido a que al tenerse códigos de Huffman más cortos, toma menos tiempo recorrer el patrón codificado en el conteo, y a su vez, el número de pasos en el mapeo LF disminuye. Sin embargo, la eficiencia de la función Occ también disminuye, sobre todo cuando se trata de valores de k tales que $\log k$ no es potencia de dos. Por esto, resultaría interesante estudiar la implementación de esta función de una manera tan eficiente como resulta para el caso de $k = 2$ y $k = 4$. Por otra parte, vimos que no se puede aumentar k tanto como queramos, pues el espacio que ocupa el índice aumenta considerablemente al ocupar valores altos de k. Esto acotaría el estudio a utilizar los valores de $k = 8$ y $k = 16$, pues para valores mayores, el espacio ocupado por el índice aumenta demasiado. Con $k = 8$ no se espera una mejoría ya que 8 no es potencia de 2, sin embargo con $k = 16$, el índice podría convertirse en una mejor alternativa para ser usada en la práctica.

El fin último de este estudio futuro sería encontrar el valor óptimo de este parámetro

que hace que el índice sea lo más rápido posible y con un espacio que resulte práctico. De todas maneras, la versión que utiliza cuatro símbolos de codificación, la cual resultó ser la mejor alternativa para el texto de ADN, probablemente es la óptima, dado que una versión con más símbolos para un alfabeto de cuatro símbolos no tiene sentido. Para alfabetos de mayor tamaño, utilizar más símbolos en la codificación podría conllevar a una mejora en los tiempos, convirtiendo al Índice FM Huffman en una mejor solución para el problema de búsqueda en texto.

BIBLIOGRAFíA

[1] S. Grabowski, V. Mäkinen and G. Navarro. *First Huffman, then Burrows-Wheeler: A Simple Alphabet-Independent FM-Index*. Technical Report TR/DCC-2004-4. Departamento de Ciencias de la Computación, Universidad de Chile, 2004. Short paper version in: In *Proc. 11th String Processing and Information Retrieval* (SPIRE'04) (abstract), pages 210-211. LNCS 3246.

[2] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(12):262-272,1976.

[3] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14, pp. 249-260, 1995.

[4] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Ann. Symp. on Switching and Automata Theory*, pp. 1-11, 1973.

[5] U. Manber and G. Myers. Suffix arrays: a new nethod for on-line string searches. *SIAM Journal on Computing*, 22(5):935-948, October 1993.

[6] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. DEC SRC Research Report 124, 1994.

[7] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th Symposium on Discrete Algorithms* (SODA'01), pp. 269-278, 2001.

[8] P. Ferragina and G. Manzini. Oppostunistic data structures with applications. In *Proc. 41st IEEE Symp. on Foundations of Computer Science* (FOCS'00), pages 390-398,2000.

[9] P. Ferragina and G. Manzini. *On Compressing and Indexing Data*. Technical Report: TR-02-01, University of Pisa, 2002.

[10] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407-430, 2001.

[11] J. Kärkkäinen and E. Sutinen. Lempel-Ziv Index for q-Grams. *Algorithmica*, 21(1):137-154, 1998.

[12] S. Kurtz. Reducing the space requirement of suffix trees. *Software - Practice and Experience*, 29(13):1149-1171, 1999.

[13] D.A. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics.* Kluwer Academic Publishers, 1998.

[14] R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd Symposium on Theory of Computing* (STOC'00), pp. 397-406, 2000.

[15] R. Grossi , A.Gupta, and J.S. Vitter. High-order entropy compressed text indexes. In *Proc. 12th Symposium on Discrete Algorithms* (SODA'01), pp. 269-278, 2001.

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

[17] L. Ziv and A. Lempel. Compression of individual sequences via variable lentgh coding. *IEEE Trans. of Information Theory*, 24:530-536, 1978.

[18] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. International Symposium on Algorithms and Computation* (ISAAC'00), LNCS 1969, pp. 410-421, 2000.

[19] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. of 13th Annual Symposium on Discrete Algorithms* (SODA'02), pp. 225–232, 2002

[20] V.Mäkinen and G.Navarro. Compressed compact suffix arrays. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching* (CPM'04), LNCS 3109 pp. 430-433, 2004.

[21] V. Mäkinen and G. Navarro. *New Search Algorithms and Space/Time Tradeoffs for Succinct Suffix Arrays*, Technical report, C-2004-20, Department of Computer Science, University of Helsinki, April 2004.

[22] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. Manuscript, 2004. Short paper version: Run-length FM-index (abstract). In *Proc. DIMACS Workshop: "The Burrows-Wheeler Transform: Ten Years Later"*, August 19-20, 2004, pages 17-19.

[23] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, Dec. 1989.

[24] I. Munro. Tables. In *Proc. of 16th Foundations of Software Technology and Theoretical Computer Science* (FSTTCS'96), pages 37-42, 1996.

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

[26] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive compression scheme. *Communications of the ACM*, 29(4):320-330, 1986.

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

[28] V. Mäkinen. Compact suffix array - a space-efficient full-text index. *Fundamenta Infomaticae*, 56(1-2):191-210,2003

[29] R. González, S. Grabowski, V. Mäkinen and G. Navarro. Practical implementation of rank and select queries. Manuscript, 2004.