Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción y objetivos

Los sistemas de recuperación de información clásicos se utilizan para recuperar documentos en base a su contenido, ofreciendo un conjunto de funcionalidades que ayudan a los usuarios a manipular grandes colecciones de datos. Más concretamente este tipo de sistemas manejan normalmente textos y ocasionalmente imágenes, no obstante la mayoría de las funcionalidades ofertadas por un sistema de recuperación tradicional están orientadas a la indexación, tratamiento y recuperación de entidades atómicas que, tradicionalemente, se han asociado con documentos de texto.

En la actualidad se está incrementando de forma exponencial número de sistemas de recuperación de información que manejan colecciones de documentos semiestructurados debido a la utilización masiva de los lenguajes de etiquetado estándar (SGML [ISO86] y XML [BPSM00, Bra00][1]) para representar la información textual.

Por lo general, las funcionalidades anteriormente mencionadas están orientadas al tratamiento de los datos sin considerar otros aspectos relacionados con información estructural, pues cuando el usuario busca un conjunto de términos el sistema le devuelve una lista ordenada de documentos sin tener en cuenta aspectos estructurales ni en la formulación de la consulta ni en la correspondiente respuesta del sistema. En algunas situaciones el usuario necesita disponer de funcionalidades de búsqueda y recuperación que tengan en cuenta la estructura de los documentos. Por ejemplo, en el caso de los sistemas de recuperación de información jurídica puede ser muy útil proporcionar a los usuarios la posibilidad de incluir restricciones de estructura en sus consultas [HMQS96]. En entornos jurídicos, generalmente los usuarios utilizan información basada en la estructura para localizar los documentos deseados. Esto se debe a que la información jurídica se organiza habitualmente de una manera estructurada (títulos, artículos, etc.). Es más, el dominio de este tipo de información se compone de diferentes tipos de documentos, como leyes, reglas, códigos de leyes, etc., cada uno con una estructura particular. El uso, como se ha indicado cada vez más considerable, de XML en numerosos contextos ha propiciado un notable interés en explotar en el proceso de búsqueda y recuperación la estructura de los documentos. Un posible caso de aplicación puede ser el de las bibliotecas digitales [BVNF98].

La inclusión de la información estructural de los documentos afecta de diversas maneras al diseño e implementación de los sistemas de recuperación de información. En primer lugar, el proceso de indexación debe considerar la estructura de forma adecuada para permitir al usuario realizar búsquedas por contenido y estructura. En segundo lugar, el proceso de recuperación debe utilizar el contenido y la estructura para estimar la relevancia de los

[1] Recomendación del W3C sobre XML 1.1 en `http://www.w3.org/TR/xml11`

documentos. Por último, la interfaz debe permitir al usuario una completa utilización de la estructura del documento [VFC02] ya que la consulta por contenido y estructura sólo se puede llevar a cabo si el usuario es capaz de especificar en la consulta *qué* está buscando y en *dónde* se debe encontrar en los documentos. En este caso, el *qué* hace referencia a la especificación del contenido y el *dónde* hace referencia a la estructura de los documentos. En este caso, lo recuperado serán partes del documento delimitadas por etiquetas del mismo tipo.

Como pauta general, los sistemas de recuperación de información manejan grandes volúmenes de información textual, hecho que implica que se necesite un espacio de almacenamiento considerable y el simple hecho de acceder a la información en respuesta de alguna solicitud de información conlleva un tiempo que puede no ser aceptable. Una de las posibles soluciones a estos problemas es utilizar técnicas de compresión de datos y estrategias de indexación respectivamente. La compresión de datos reduce la cantidad de espacio en disco necesario realizando una codificación de la información de manera adecuada y las estrategias de indexación se utilizan para proporcionar un acceso rápido a los documentos almacenados, de la misma forma que un índice de un libro convencional proporciona un acceso rápido a las páginas del libro en las que aparecen los conceptos mencionados.

Además la compresión de grandes colecciones de documentos no sólo reduce la cantidad de espacio que los datos precisan en disco, sino que también disminuye todos los tiempos de consulta en los sistemas de recuperación de texto en la mayoría de los casos. Las mejoras en los tiempos de proceso se consiguen gracias a la reducción de los tiempos de acceso a disco que se necesitan para acceder al texto comprimido. En las últimas décadas las velocidades de los procesadores se han incrementado mucho más que las de transferencia de los discos, por lo tanto, no es descabellado pensar que puede ser una buena elección cambiar tiempos de transferencia por tiempos de descompresión, o dicho con otras palabras tiempos de acceso a disco por tiempos de proceso. Por otra parte, desde hace unos años investigaciones sobre búsqueda "directa" sobre texto comprimido (por ejemplo búsqueda sobre texto comprimido sin descomprimirlo) nos permiten considerar muy adecuada la opción de comprimir el texto, lo que supone la utilización de menos espacio, y buscar sobre dicho texto comprimido consiguiendo que las búsquedas se realicen más rápido que sobre el texto sin comprimir [WMB99, ZMNBY00].

1.1. Compresión y recuperación de información

El hecho de comprimir textos que vayan a ser manipulados por sistemas de recuperación de información plantea algunos requisitos que descartan el uso de algunos métodos de compresión. Uno de los más importantes es la necesidad de acceder al texto de forma aleatoria sin tener que descomprimirlo con anterioridad. Esta restricción descarta a la mayoría de los métodos adaptativos como los basados en Ziv-Lempel [ZL77, ZL78] y la codificación artimética. Por otro lado, los modelos semiadaptativos como Huffman [Huf52] proporcionan poca compresión. No obstante, cuando se van a comprimir textos escritos en lenguaje natural se ha demostrado que una elección excelente es considerar palabras en lugar de caracteres como los símbolos de la fuente [Mof89] porque las palabras reflejan la verdadera entropía del texto [BCW90] mucho mejor que los caracteres. El uso de un modelo de palabras junto con un codificador de Huffman proporciona razones de compresión cercanas al 25 % debido a la distribución sesgada de las palabras. Estas razones son considerablemente mejores que las obtenidas con los modelos adaptativos clásicos. Estos resultados empeoran ligeramente si utilizamos una codificación de Huffman orientada a byte, en la que cada símbolo de la fuente se codifica como una secuencia de bytes en lugar de bits. Aunque la

razones de compresión se encuentran por encima del 30 % (aunque todavía siguen siendo competitivas) a cambio se obtiene una velocidad de descompresión y una búsqueda mucho más rápida, que son características esenciales y deseables en los sistemas de recuperación de información con textos comprimidos. Por último, el hecho que coincida el alfabeto y el vocabulario de las colecciones de textos permite una búsqueda eficiente y altamente sofisticada tanto en búsqueda secuencial como en los índices invertidos comprimidos sobre los textos [WMB99, NMN+00, MNZB00, ZMNBY00, MW01].

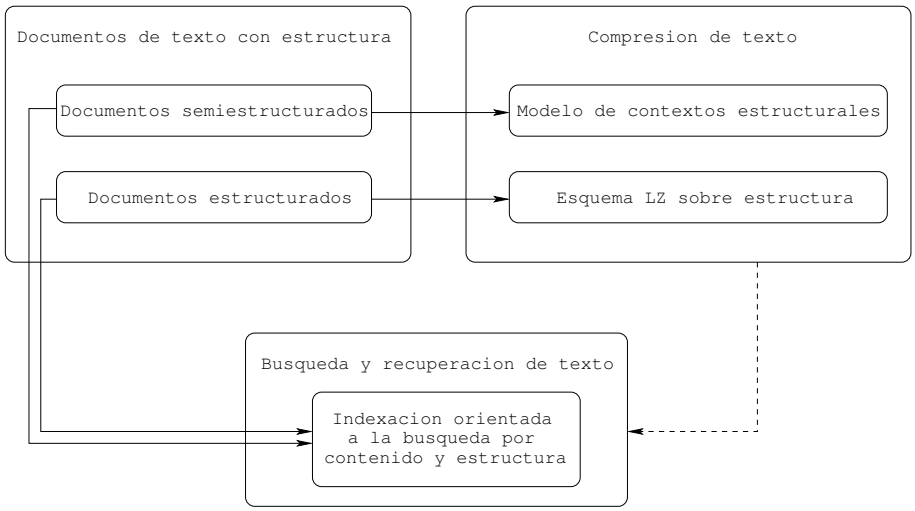


Figura 1.1: Relación entre los conceptos.

El texto escrito en lenguaje natural no se compone sólamente de palabras sino que también está constituido por caracteres de puntuación, separadores y otros caracteres especiales. La secuencia de caracteres entre cada par de palabras consecutivas se denomina *separador*. En [BSTW86] se propone comprimir los textos utilizando dos alfabetos de símbolos disjuntos: uno para las palabras y el otro para los separadores. Los codificadores que utilicen este modelo deberán considerar los textos como una secuencia estricta de dos fuentes de datos independientes y codificarlas por separado. Una vez que se ha determinado que el texto empieza con una palabra o un separador se sabe que que después de codificar una palabra se deberá codificar un separador y viceversa. Este modelo se conoce como *modelo de alfabetos separados*.

Un hecho que el modelo de alfabetos separados no tiene en cuenta es que en la mayoría de las ocasiones a una palabra le sigue un separador formado por un único espacio en blanco. Puesto que aproximadamente un 70 % de los separadores que aparecen en el texto están formado por un sólo espacio en blanco [Mof89], en [MNZB00] se propone un nuevo modelo de datos que utiliza un único alfabeto para codificar tanto las palabras como los separadores y representa el separador formado por un único espacio en blanco de forma implícita. A este modelo se le conoce como *modelo sin espacios* y, por lo tanto, supone que después de cada palabra descodificada aparecerá un espacio en blanco, salvo que el siguiente símbolo que se descodifique sea un separador.

Las técnicas y métodos de compresión clásicos no tienen en cuenta la estructura de los documentos y, consecuentemente, si permiten realizar búsquedas éstas serán exclusivamente búsquedas por contenido. Actualmente existen algunas propuestas que intentan sacar partido de la estructura de los documentos a la hora de mejorar las razones de compresión. Un método de compresión que tiene en cuenta la estructura de los documentos cuando

comprime y descomprime es *XMill*, desarrollado en los laboratorios AT&T [LS00]. XMill es un compresor para datos XML y que utiliza *zlib* como motor principal de compresión. Su principales ventajas es que mantiene los mismos niveles de compresión y velocidad que *gzip*, y que no necesita un esquema de información para comprimir o descomprimir. Dado que es una herramienta pensada exclusivamente para el almacenamiento e intercambio de documentos XML no contempla la búsqueda en elementos de estructura, aunque tiene en cuenta dicha estructura a la hora de realizar la compresión. Otro compresor específico para XML es *XGrind* [TH02] que soporta directamente consultas sobre los documentos comprimidos pero no obtiene ventaja de la estructura.

Existen otras aproximaciones para comprimir datos XML basadas en la utilización de codificadores PPM que sacan partido a la estructura. Un ejemplo es *XMLPPM* [Che01] el cual es un compresor adaptativo que utiliza diferentes modelos PPM. XMLPPM utiliza un parser ESAX, una variante de SAX, para obtener diferentes partes del documento (entendiéndose por "partes" los nombres de las etiquetas, los nombre y valores de los atributos de las etiquetas, el texto, etc.), cada parte se codifica mediante un modelo PPM diferente. XMLPPM es un compresor adaptativo y no se pueden realizar búsquedas ni accesos aleatorios sobre los textos comprimidos.

Por otro lado, los sistemas de recuperación de información necesitan localizar de manera eficiente un término en concreto dentro de los textos de la colección. Para ello es habitual indexar los documentos de la colección y la técnica más generalizada y simple son los índices invertidos [BYRN99, WMB99]. Generalmente, un índice invertido se compone de un vector que conteniene todas las términos distintos de la colección (*vocabulario*) en orden lexicográfico y para cada término del índice se almacena una lista con todas las posiciones del texto o documentos en los que aparece (*lista de ocurrencias*). La granularidad de un índice viene dada por las entidades que estén referenciadas en las posiciones de las listas de ocurrencias, así pues, dependiendo de la granularidad los índices invertidos se pueden clasificar en índices invertidos con direccionamiento a palabra, documento y bloque. En los que las posiciones de las listas de ocurrencias hacen referencia a la posición del texto, documento o bloque de tamaño fijo en dónde aparece la palabra respectivamente. Con independencia de la granularidad del índice, existen diferentes métodos para comprimirlo [MNZ97, WMB99, MZ00]. Es habitual utilizar una codificación por diferencias junto a técnicas de representación de longitud variable de números enteros [Eli75].

Originalmente, las técnicas de indexación se idearon para indexar texto sin comprimir y sin estructura. La adaptación de las técnicas clásicas para que indexen texto comprimido con algún modelo de estático o semiadaptativo es realitivamente sencillo, pero por el contrario no existe una técnica de indexación clara que permita indexar documentos semiestructurados por contenido y estructura simultáneamente. A continuación se comentan algunas propuestas.

Yong Kyu Lee et al. proponen diferentes índices basados en árboles B^+ que permiten la localización de palabras dentro de elementos de estructura [LYYB96]. Un árbol es la representación de la estructura jerárquica de un documento y cada nodo del árbol representa a un determinado elemento de estructura. A cada nodo del árbol se le representa en el índice mediante un entero único, así pues será necesario disponer de algún mecanismo suplementario que permita asociar el identificador con el tipo de elemento de estructura al que representa. Dado que este trabajo tomó como base documentos SGML no tiene en cuenta las etiquetas XML con contenido.

Por otro lado, Navarro y Baeza-Yates proponen en [NBY97a] un modelo genérico en el que utilizan *nodos proximales* para realizar consultas por contenido y estructura, y evitan el problema restringiendo la expresividad del lenguaje de consulta.

Existen numerosos sistemas de recuperación de información que ha integrado con éxito índices invertidos con documentos comprimidos, como por ejemplo *MG*, Managing Gigabytes, [WMB99] que es un software de dominio público, versátil y de propósito general que comprime e indexa textos, documentos escaneados e imágenes. MG utiliza un código tipo Huffman, denominado *Huffword* [ZM95, MT97], para comprimir el texto lo que le proporciona un nivel de compresión competitivo. También mantiene comprimido el índice y obtiene unos tiempos de respuesta aceptables. Puede comprimir documentos semiestructurados tratando la etiqueta como una palabra más, pero no tiene en consideración dicha estructura en el momento de la recuperación.

1.2. Objetivos de la tesis

Los sistemas de recuperación tradicionales tratan a los documentos como entidades atómicas, comprimiéndolos, indexándolos y recuperándolos como elementos indivisibles. Los sistemas de recuperación de información modernos deben ser capaces de manipular representaciones de documentos más elaboradas como, por ejemplo, documentos escritos en formato SGML o XML. Los nuevos estándares de representación de documentos con formato obligan a diseñar e implementar nuevos modelos y herramientas para trabajar con la estructura de los documentos. Esto implica que los documentos no serán considerados como entidades atómicas sino como un conjunto de objetos agregados e interrelacionados que necesitan ser comprimidos, indexados, recuperados y presentados conjunta o separadamente de acuerdo a las necesidades del usuario.

Por otro lado, el almacenamiento, intercambio y manipulación de textos con formato como medio de representación de datos estructurados esta proliferando en todo tipo de aplicaciones, desde las bases de datos textuales y las bibliotecas digitales hasta los servicios web y comercio electrónico. El texto con formato, y en particular los que se ajustan al estándar XML, se está convirtiendo en un modelo a seguir para codificar información con una estructura simple o compleja por un lado y fija o variable por el otro. Aunque hace algún tiempo XML se previó como un mecanismo para describir datos estructurados no ha sido hasta la reciente explosión de la "empresa electrónica" cuando ha mostrado su potencial para describir todas las clases de documentos (facturas, albaranes, recibos, remuneraciones, etc.) que almacenan e intercambian las empresas.

Aunque la información que gestiona una empresa generalmente se almacena en bases de datos relacionales o almacenes de datos, es importante alamacenar copias digitales, en formato XML, de toda la documentación que se ha producido o intercambiado con el paso del tiempo porque un sistema de recuperación de información estructurada podría proporcionar acceso aleatorio a todos esos documentos con formato y también se podrían buscar, visualizar y navegar fácilmente. Como valor añadido, sería deseable que este repositorio ocupara el menor espacio posible.

Así pues, pueden existir documentos con un mayor o menor nivel de estructuración según se ha visto, por ejemplo, los documentos que representan libros, artículos, noticias, etc. generalmente tendrán una estructura más o menos sencilla y los contenidos de los elementos estructurales estarán formados por texto escrito en lenguaje natual y de un tamaño más o menos grande; por el contrario, los documentos que almacenan e intercambian las empresas posiblemente tendrán una estructura más compleja que en el caso anterior y sus contenidos serán de un tamaño mucho más pequeño y estarán formados por nombres, referencias, cantidades, descripciones cortas, etc. Con esto se puede hablar de dos tipos de documentos con estructura:

I. Documentos semiestructurados, que son aquellos que contienen marcas (elementos de estructura) que se utilizan para organizar de una manera semántica los contenidos de los mismos. La estructura de dichos documentos se ajusta a una DTD (definición de tipo de documento) y, por lo tanto, su estructura es variable, es decir que dos documentos que se ajusten a la misma DTD pueden tener una estructura diferente.

II. Documentos estructurados, que son aquellos que tienen una estructura fija y que además contienen etiquetas cuyo significado semántico va más allá del propio texto de la etiqueta, pues son una representación ASCII de tipos de datos.

Con el presente trabajo de tesis, y teniendo en cuenta lo comentado con anterioridad, se pretende cubrir los siguientes objetivos encaminados a diseñar un sistema de recuperación de información que permita realizar búsquedas por contenido y estructura manteniendo sus documentos comprimidos:

I. En el marco teórico de un sistema de recuperación de información que manipule documentos de texto semiestructurado es interesante no sólo recuperar documentos sino también los elementos estructurales que los componen. Por lo tanto será preciso definir y diseñar un índice invertido que considere a los elementos estructurales que forman los documentos semiestructurados como entidades atómicas para que de esta forma se puedan recuperar como respuesta a consultas por contenido y/o estructura, permitiendo resolver consultas del estilo a "¿qué documentos han sido escritos por el autor X?", "¿qué documentos tienen resumen?" o "¿cuáles son los teoremas en los que aparecen las palabras X e Y?", suponiendo que "autor", "resumen" y "teorema" respectivamente están representados como elementos estructurales en una determinada DTD. Además se deberá tener presente la jerarquía de dichos elementos pues una entidad atómica (elemento estructural) que aparezca en el nivel jerárquico n podrá estar compuesta por un conjunto de entidades atómicas (elementos estructurales) del nivel $n+1$.

II. Por otro lado se propondrán, estudiarán y diseñarán diferentes métodos de compresión que tengan en cuenta la estructura a la hora de comprimir los documentos. Por un lado, es de suponer que dichos métodos obtengan mejoras en la razón de compresión porque la estructura de los documentos está organizada teniendo en cuenta factores semánticos y por lo tanto la distribución de los símbolos contenidos en los elementos estructurales del mismo tipo es diferente a la distribución de los símbolos contenido en los elementos estructurales de otro tipo. Asimismo, el hecho de considerar la estructura en la compresión facilita el hecho de resolver consultas por contenido y estructura sobre el texto comprimido.

1.3. Aportaciones de la tesis

Este trabajo propone un modelo de compresión general pensado para comprimir documentos semiestructurados basado en el contenido semántico de los diferentes tipos de elementos de estructura, denominado *modelo de contextos estructurales* y se puede considerar como una extensión del modelo de alfabetos separados [BSTW86]. Éste utiliza un modelo específico para modelar y codificar cada conjunto de tipos de elementos de estructura con significados semánticos similares. En una primera etapa se ha realizado una implementación semiadaptativa (utilizando una versión de Huffman semiadaptativa) con la intención de usarlo en bases de datos textuales con documentos comprimidos [ANF03a, ANF03b] y posteriormente se ha realizado una implementación adaptativa del mismo (utilizando

para ello PPMD+) con vista a utilizarlo para el almacenamiento y transmisión de dicho documentos [AFN04]. Las razones de compresión obtenidas por cada implementación han superado respectivamente a las obtenidas por los sistemas semiadaptativos y adaptativos comentados en el estado arte.

A pesar de que el modelo de contextos estructurales se puede aplicar sobre documentos estructurados se propone una técnica de compresión inspirada en el esquema Lempel-Ziv que permite manipular y comprimir la estructura de una forma más adecuada que el modelo general [ANF04]. El principal elemento del esquema es la denominada transformación LZCS, que es la encargada de sustituir los elementos estructurales completos por una referencia a la primera ocurrencia de la misma. La referencia está representada mediante una etiqueta especial que contiene la posición de la ocurrencia en el texto transformado. La ventaja de la transformación LZCS es que produce texto plano con referencias facilitando la visualización, navegación y búsqueda sobre el mismo.

Para recuperar los documentos por contenido y estructura se ha propuesto un esquema de índice invertido con direccionamiento a elementos estructurales que se puede aplicar sobre documentos con o sin compresión [AFVV02a, AFVV02b]. En este tipo de índice, las posiciones que contienen las listas de ocurrencias no indican ninguna posición de los ficheros sino que referencias filas de una tabla. Dicha tabla se ha denominado tabla de control de estructura y en cada fila contiene la información necesaria para resolver las consultas por contenido y estructura. La propuesta ofrece unas prestaciones intermedias entre los índices invertidos con direccionamiento a palabra y con direccionamiento a documentos.

Aunque el enfoque que se ha dado en este trabajo tanto al modelo de contextos estructurales como al índice invertido con direccionamiento a elementos estructurales se ha limitado al ámbito de los sistemas de recuperación de información y bases de datos textuales con documentos comprimidos, la utilización de los mismos no está limitada a dichas áreas, pudiéndose utilizar en otros campos como por ejemplo en el área de las bibliotecas digitales [AFV+02].

1.4. Estructura de la tesis

A continuación se comentará brevemente la distribución de los contenidos comentados en este trabajo de tesis. El material se ha distribuido, como suele ser habitual, en dos partes claramente diferenciadas: la parte denominada *estado del arte* en la que se comentan algunos trabajos previos que han servido como base a la investigaciones; y la parte conocida como *aportaciones* en la que se detallan las investigaciones y propuestas que conforman esta tesis doctoral.

La parte de "estado del arte" relata los conceptos elementales necesarios involucrados en el ámbito de los sistemas de recuperación de información que mantienen sus documentos comprimidos. En primer lugar, en el capítulo 2 se detallarán los conceptos teóricos más generales relacionados con la compresión de datos y que se manejarán durante la exposición del trabajo. También se enunciarán diferentes las clasificaciones de las técnicas de compresión más habituales, así como la demostración del teorema del recuento que es importante pero muy poco habitual en la literatura relacionada. En el capítulo 3 se entrará un poco más en detalle en la descripción y análisis de los modelos, métodos y técnicas de compresión de texto (también denominada compresión sin pérdida) que serán el punto de partida para las propuestas. El uso de ciertos algoritmos de compresión sin pérdida permite realizar búsquedas sobre los textos comprimidos, en esta línea en el capítulo 4 se exponen con brevedad las técnicas clásicas que se utilizan para buscar patrones en un texto, en un principio sin considerar la estructura de los documentos.

Se comentan además las diferentes técnicas estándar de indexación de texto así como las empleadas en la compresión del índice y que serán útiles para comprender el índice invertido con direccionamiento a elementos estructurales y la compresión del mismo. Para cerrar esta parte, en el capítulo 5 se comentarán brevemente los aspectos y conceptos básicos que están presentes en un sistema de recuperación de información y de los diferentes modelos que intervienen entre una necesidad de información por parte del usuario y la respuesta de la misma. Asimismo, también se hará un repaso exhaustivo de las técnicas modernas que se utilizan en los sistemas de recuperación de información que manejan documentos semiestructurados.

La parte correspondoente a las "aportaciones" comienza en el capítulo capítulo 6 en el cual se enuncia la propuesta de un modelo de compresión genérico que considera y saca partido de la estructura de los documentos semiestructurados. El modelo se ha denominado "modelo de contextos estructurales" y ha sido ideado para comprimir documentos semiestructurados en general. Dado que el modelo es genérico existen diferentes implementaciones del mismo y en dicho capítulo se muestran dos alternativas, una semiadaptativa y la otra adaptativa. Se han obtenido razones de compresión muy competitivas en ambos casos. En la versión semiadaptativa se pueden realizar búsquedas y acceder de forma aleatoria a los documentos comprimidos, mientras en en la version adaptativa no. No obstante, las razones de compresión son mucho mejores en el segundo caso, mostrando claramente la proporción inversa que existe entre dichas funcionalidades y la razón de compresión.

Por otro lado, y aunque el modelo de contextos estructurales se puede utilizar para comprimir documentos estructurados, se ha ideado una técnica de compresión inspirada en la idea de Lempel y Ziv para comprimirlos. En el capítulo 7 se detalla dicha técnica que consiste en transformar los documentos estructurados antes de comprimirlos usando cualquier técnica estándar. La idea principal de la transformación es sustituir las subestructuras que se repetiten por un apuntador a la posición en la que se encuentra la primera. La salida de la transformación continúa siendo texto plano por lo que la visualización, navegación y búsquedas sobre los textos transformados de puede realizar de una manera sencilla. Además, las razones de compresión obtenidas sólo por la transformación son realmente buenas.

Aunque existen en la literatura algunas propuestas de esquemas de indexación que tienen en cuenta la estructura para resolver consultas por contenido y estructura, en el capítulo 8 se expone una propuesta denominada "índice invertido con direccionamiento a elementos estructurales" que facilita la resolución eficiente de dichas consultas. También en el mismo capítulo se ha definido un nuevo modelo de recuperación de información para documentos con estructura de tal forma que permite resolver consultas por contenido y estructura ponderando cada elemento estructural por la densidad de los términos de la consulta que contiene, así como la proximidad de los mismos.

Por último, en el capítulo 9 se comentan los resultados más significativos que se han obtenido durante el desarrollo de las propuestas enunciadas en este trabajo para alcanzar los objetivos inicialmente propuestos, y se esbozan líneas de trabajo futuro para continuar el trabajo que ha comenzado con esta tesis.

Capítulo 2

Compresión de datos

El objetivo básico de la compresión de datos es la reducción del número de bits utilizados para almacenar o transmitir información. Los sistemas de compresión se pueden clasificar en dos grandes familias: los compresores sin pérdida de información (*lossless*) y los compresores con pérdida de información (*lossy*) que se utilizan para comprimir representaciones digitales de señales analógicas, como por ejemplo, sonidos e imágenes. Generalmente, los compresores con pérdida obtienen mejores tasas de compresión que aquellos con pérdida, lo que se debe, en parte, a que los datos descomprimidos no tienen por qué ser una réplica exacta de los datos originales. Esto es así porque la información que se pierde en el proceso de codificación (compresión) no es importante para el sistema auditivo-visual humano. Por otra parte, los compresores sin pérdida garantizan la regeneración exacta de los datos comprimidos tras la descompresión puesto que no se ha producido ningún tipo de pérdida de información durante el proceso de compresión de los mismos. Consecuentemente, este último tipo de compresión se utilizará cuando se desean comprimir ficheros ASCII, registros de una base de datos, información de una hoja de cálculo, etc., es decir, todos los datos comúnmente conocidos como datos de texto, por ello a este tipo de sistemas de compresión se les suele llamar compresores de texto.

2.1. Una definición de información

El concepto de información es antiguo y de complicada definición. En el contexto de la informática, puede decirse que es todo aquello que tiene asociado algún significado. Se puede añadir que será necesario representar la información de alguna manera para poder almacenarla o transmitirla. Decimos entonces que la información se almacena o transmite mediante mensajes. Para que un mensaje contenga información es preciso que aporte algo nuevo o imprevisto, es decir, a mayor dimensión cognoscitiva mayor cantidad de información contiene.

En un proceso de transmisión de información existen tres elementos básicos: el emisor (o fuente) de información, el canal de comunicación y el receptor (o consumidor) de información. La información fluye desde emisor al receptor pasando por el canal, y puede verse alterada en el caso que el canal tenga ruido [Abr63, CT91]. Ahora bien, en el contexto de compresión de datos se supone que en la mayoría de los casos siempre se trabajará con un canal sin ruido y sin memoria —esto es, la probabilidad de aparición de un símbolo no depende de los símbolos que han aparecido con anterioridad—. Teniendo esto en cuenta,

se puede definir la información asociada a un evento (o suceso) transmitido por un canal sin ruido de la siguiente manera.

Definición 2.1 (Información asociada) *La información asociada al evento x_i, que tiene una probabilidad de ocurrencia p_i, en un canal sin ruido ni memoria se puede calcular mediante la expresión*

$$I(x_i) = \log_2 \frac{1}{p_i} = -\log_2 p_i \tag{2.1}$$

Como podemos apreciar, la definición previa de información es coherente si tenemos en cuenta que:

1. Si el evento a ocurre siempre, la ocurrencia del proceso de comunicación no proporciona ninguna información al sujeto.

$$p(a) \to 1 \iff I(a) \to 0$$

2. Si el evento a no ocurre casi nunca, la ocurrencia del proceso de comunicación proporciona mucha información al sujeto.

$$p(a) \to 0 \iff I(a) \to 1$$

2.2. Redundancia y compresión

Todos los sistemas de compresión de datos trabajan reduciendo la cantidad de redundancia del conjunto de datos a comprimir. Si la compresión es efectiva, la cantidad de memoria que precisan los datos originales será mayor que la cantidad de memoria utilizada por los datos comprimidos. En secuencias de textos (y de forma general, en cualquier secuencia de datos) prácticamente sólo se puede encontrar presente la redundancia estadística. Las únicas secuencias de datos que carecen de este tipo de redundancia son las secuencias de datos aleatorios que siguen una distribución uniforme. La redundancia estadística provoca que las probabilidades de ocurrencia de los símbolos codificados sean diferentes entre sí.

Los sistemas de compresión de datos tratan de eliminar cuanta redundancia sea posible. El tipo y cantidad de redundancia a eliminar depende de la naturaleza de los datos que se estén comprimiendo y caracteriza fuertemente al compresor y al descompresor. Existen tres tipos de redundancia, conocidos como redundancia en la codificación, redundancia en la secuencia de aparición y redundancia psico-visual o psico-auditiva:

1. Los compresores basados en la eliminación de la redundancia en la codificación aprovechan que la frecuencia de uso de los símbolos del alfabeto, que se utilicen para representar la información, no siga una distribución uniforme. Esto se debe a que usamos más frecuentemente unos caracteres que otros. En 1952 Huffman propuso el llamado algoritmo de compresión de longitud variable (variable-length coding) o algoritmo de Huffman [Huf52], que elimina este tipo de redundancia. Este tipo de algoritmos no provocan pérdida de información.

2. Los algoritmos que eliminan la redundancia en la secuencia de aparición tienen en cuenta que normalmente existe algún tipo de correlación entre uno o varios de los símbolos ya aparecidos y él o los que van a aparecer. Por ejemplo, en los textos escritos en castellano, la probabilidad de aparición del carácter `m` delante del `p` es

mucho mayor que la probabilidad de aparición del carácter `n` previo al `p`. Existen gran cantidad de algoritmos que codifican la información eliminando este tipo de redundancia tales como RLE (Run Length Encoding) o el propio LZW y que no provocan ninguna pérdida de información.

3. La eliminación de la redundancia psico-visual y psico-auditiva tiene en cuenta que el ojo y el oído humano no responde con la misma sensibilidad a toda la información visual y auditiva. Existe cierta información que puede ser eliminada sin que se produzca un deterioro significativo de la calidad de la imagen o del sonido. La eliminación de este tipo de redundancia produce elevadas relaciones de compresión a costa de la pérdida irreversible de parte de la información. Todos los algoritmos de compresión que usan transformadas [JN84, Kou95], tales como la transformada coseno, la transformada rápida de Walsh-Hadamard o la computacionalmente costosa transformada Karhunen-Loève, se basan en la eliminación de la redundancia psico-visual.

2.3. Comprimir = Modelar + Codificar

Shannon calculó la medida de la entropía del idioma inglés en 1951 [Sha51]. Para ello utilizó hablantes nativos de habla inglesa como predictores y midió la entropía del error de predicción, calculando de esta forma que aproximadamente el 50 % de los caracteres en el idioma inglés son redundantes[1]. Esta idea ha sido utilizada por Rissanen y Langdon [RL81] para indicar que cualquier compresor de datos puede ser descompuesto en dos partes independientes: (1) un modelo de la fuente a comprimir utilizado como predictor y (2) un codificador (ver figura 2.1).

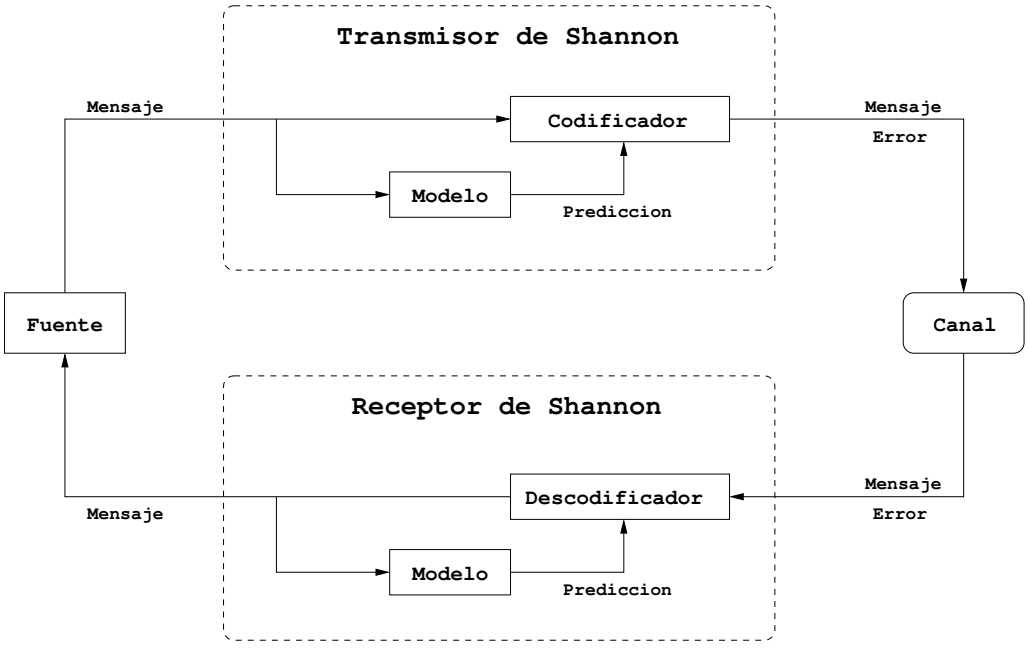


Figura 2.1: Paradigma de la compresión/descompresión formulado por Shannon

[1]En consecuencia, la mitad de las letras o palabras en este lenguaje son superfluas y el resto están determinadas por la estructura estadística del lenguaje. En otras palabras, se podrían expresar las mismas ideas con aproximadamente la mitad de texto escrito

En consecuencia, lo que se conoce habitualmente como compresión de datos es un proceso que consiste en dos etapas diferentes:

1. *Modelado*, o creación de un modelo, en el que se determina una manera lógica de dividir el texto en un conjunto de símbolos, y se realiza una estimación de la probabilidad de que el símbolo modelado sea el próximo símbolo que aparezca en el texto. Así pues, se puede considerar al modelo como una fuente de información que trata de imitar a la fuente de información a comprimir. El descompresor deberá poseer, o ser capaz de crear, un modelo idéntico.

2. *Codificación*, que define cómo se debe de representar cada símbolo modelado en el texto comprimido [BWC89]. La misión principal del codificador es la de encontrar una representación eficiente de la información residuo, que es aquella que el modelo no puede predecir. Básicamente, por cada símbolo-fuente emitido, el modelo genera otro símbolo-fuente, y la discrepancia entre ambos es lo que el codificador representa de forma compacta.

Por lo tanto, cualquier compresor de datos puede describirse en términos de un modelo de la fuente de información que se comprimirá y un codificador.

2.4. Fuentes de información

Una fuente de información discreta es aquella que produce mensajes como sucesiones de símbolos-fuente extraídos de entre una colección finita de símbolos-fuente que forman parte de un conjunto de eventos, $\mathcal{A}$, que se le denomina alfabeto-fuente. La notación

$$\mathcal{A} = \{x_1, \ldots, x_n\}$$

indica un alfabeto-fuente $\mathcal{A}$ compuesto de n símbolos-fuente x_i. Al valor n se le llama base de la fuente de información discreta.

En el ámbito de la teoría de la información, la emisión de símbolos para la construcción de mensajes se considera gobernada por un proceso estocástico, lo que quiere decir que la fuente de información produce los símbolos-fuente de acuerdo con unas determinadas probabilidades de ocurrencia de cada evento, $p(x_i) = p_i$, que son conocidas

$$\mathcal{P} = \{p_1, \ldots, p_n\} \bullet \sum_{i=1}^{n} p_i = 1 \ \wedge \ \forall p_i \geq 0$$

La cantidad de información que recibimos con m mensajes de una fuente de n posibles mensajes debe ser la misma que al recibir *un* mensaje de una fuente con n^m mensajes. Esto nos sugiere una función logarítmica para la cantidad de información, ya que si suponemos que tenemos dos fuentes equiprobables de n y n^m mensajes respectivamente, la información al recibir m mensajes de la primera fuente es $m \times f(n)$, en donde f es una función creciente, por ahora sin determinarç (nótese que al ser la fuente equiprobable, la probabilidad de cualquier mensaje es $1/n$, pero f es función de la inversa de la probabilidad, por lo que depende de n). Por otro lado, la recepción de *un* mensaje de la segunda fuente nos da una información $f(n^m)$. Al hacer la igualdad, tenemos que $m \times f(n) = f(n^m)$, sugiriendo una función logarítmica.

En 1948 Shannon propuso una medida para la información que cumplía todas la expectativas anteriores denominada *entropía* y presentó un teorema que establece una relación entre las probabilidades y los códigos de los símbolos [Sha48], demostrando que la mejor

forma de representar un símbolo que tenga una probabilidad p de aparecer en el texto es utilizar $-\log_2 p$ bits. De esta manera, los símbolos con una elevada probabilidad se codifican con pocos bits y los símbolos con probabilidades bajas se codifican con un número mayor de bits.

2.5. Modelado de la fuente

Si tenemos una muestra lo suficientemente grande de un mensaje generado por la fuente, se cumple que dicha muestra es (estadísticamente hablando) representativa del conjunto de todas las cadenas o secuencias de Markov[2] que se puedan producir por la fuente y, por este motivo, diremos que se trata de un proceso ergódico[3].

Los *modelos de contexto finito* asignan una probabilidad a un símbolo basándose en las frecuencias de los símbolos que han aparecido antes del actual. Así, al asignar la probabilidad a un símbolo en base a los símbolos previos, tenemos la certeza que el descodificador dispondrá de esta información (pues ha descodificado los símbolos precedentes). Los modelos de contexto finito tienen en cuenta que un símbolo ha aparecido recientemente tiene bastante probabilidad de aparecer en un futuro cercano, y cuantas más veces aparezca mayor será la probabilidad que vuelva a aparecer.

Definición 2.2 (Contexto de un símbolo) *El contexto de un símbolo es la secuencia de símbolos de tamaño finito que precede o antecede al símbolo.*

En el modelado de contexto finito, el contexto de un símbolo está formado por la secuencia de símbolos que anteceden al mismo para que el modelo del descodificador tenga las mismas probabilidades que el modelo del codificador cuando va a descodificar un símbolo determinado. En caso contrario no se podría descodififcar correctamente.

En compresión de texto se utiliza el contexto porque la probabilidad de que aparezca una letra en un texto depende en gran medida de las letras que han aparecido con anterioridad. Por ejemplo, en español es mucho más probable que aparezca una "u" después de una "q" que cualquier otra letra.

Definición 2.3 (Orden del modelo) *El orden de un modelo de contexto finito es un valor entero finito que representa la longitud del contexto de cada símbolo considerado por el modelo.*

Al usar un modelo de contexto finito siempre se debe indicar su orden. Por ejemplo, si los símbolos de entrada son bytes, si el contexto de cada símbolo está formado por un byte se dirá que se está utilizando un modelo de orden 1, si se usan dos bytes tenemos un modelo de orden 2, y así sucesivamente. Si no se utiliza ningún contexto se está utilizando un modelo de orden 0, este tipo de modelo es muy utilizado porque supone que la probabilidad de aparición de un símbolo es independiente de los símbolos anteriores. La utilización de modelos de orden mayor a cero obtiene "mejores" probabilidades sólo si el contexto utilizado por el modelo se puede considerar como una secuencia de Markov, si ocurre esto se dice que los datos de entrada son *sensibles al contexto*.

Por lo general la probabilidad de un símbolo se calcula a partir del número de veces que ese símbolo ha aparecido en el texto, pero cuando se utiliza la información del contexto

[2]Una cadena de Markov es una secuencia de símbolos (eventos) en la que la probabilidad de aparición de cada símbolo sólo depende de la secuencia de símbolos finita precedente.

[3]Se dice que una fuente de información es ergódica si una muestra de ella lo suficientemente pequeña es, estadísticamente hablando, representativa de toda la información que es capaz de emitir.

también se debe tener en cuenta el contexto en el que aparece y, por eso, sólo se considerarán las probabilidades de los símbolos que han aparecido en dicho contexto. Así, al utilizar dicha información del contexto se obtienen símbolos con "mejores" probabilidades que si no se tiene en cuenta la misma. A esta distribución se le suele denominar *distribución sesgada*; por otro lado una *distribución plana* es aquella en la que todos los símbolos son equiprobables. La meta de un modelo es obtener una distribución sesgada con la esperanza que proporcione una probabilidad alta para los símbolos que más aparecen en un espacio determinado, para que se puedan codificar con menos bits.

Se pueden aplicar modelos de diferentes órdenes para estimar con más exactitud la probabilidad de un símbolo. Una vez que se dispone de dichos modelos es necesario saber cómo se efectúa el cálculo. Una solución consiste en *mezclar* de modelos [BCW90], la cual combina las probabilidades obtenidas mediante todos los órdenes en una única probabilidad para cada símbolo. Esto se puede hacer sumando la probabilidad estimada para un mismo símbolo en modelos de diferente orden. Sin embargo, las probabilidades obtenidas por modelos de mayor orden tienden a ser más fidedignas y, por lo tanto, permiten alcanzar una mejor compresión; esto se puede reflejar en el proceso de mezclado ponderando (multiplicando por un factor de importancia) las frecuencias de acuerdo al orden del modelo que las ha calculado. Cada contexto tendrá un factor de ponderación que puede estar fijado de antemano o puede ir variando a medida que se vaya realizando la compresión.

2.5.1. Tasa de entropía de la fuente

La tasa de entropía de una fuente es un número que únicamente depende de la naturaleza estadística de la fuente. La unidad de la entropía es "bits/símbolo". Este valor también se conoce simplemente como *entropía* de la distribución de probabilidad, o dicho con otras palabras, la *medida de información media* de la fuente de información. No se puede inventar ninguna codificación que consiga una longitud en bits media por símbolo emitido menor que la entropía de la fuente sobre la que se realiza la codificación. Sin embargo, esto es una cota teórica y los compresores actuales no llegan a dicha cota aunque quedan muy cerca. Por otro lado, es habitual confundir la idea de entropía con la del límite inferior de la compresión. Se puede destacar que el concepto de entropía indica de manera muy pobre el límite inferior de compresión de una fuente de datos al utilizar un determinado modelo. Por lo tanto, la entropía no se puede utilizar como límite inferior para la compresión de datos. Si la fuente es un modelo sencillo se puede calcular dicho valor de una manera sencilla.

Si la secuencia de símbolos emitidos por la fuente se puede predecir con facilidad se necesitará un número menor de bits para codificar la secuencia. Si por el contrario, dicha secuencia no es predecible necesitaremos más bits para realizar la codificación. En el primer caso el valor de la entropía será más pequeño que en el segundo. Así pues, la entropía $\mathcal{H}$ sólo depende de la naturaleza estadística de la fuente y, análogamente, del modelo de predicción que se utiliza. El valor de la entropía no será el mismo para modelos de orden distinto, por lo tanto, la entropía se calcula dependiendo del orden del modelo. Si n es el tamaño del alfabeto-fuente $\mathcal{A}$ tenemos:

Modelo de orden cero: los símbolos son estadísticamente independientes los unos de los otros y cada símbolo de $\mathcal{A}$ tiene la misma probabilidad de aparecer. En este caso la tasa de entropía se calcula como

$$\mathcal{H} = \log_2 n \tag{2.2}$$

Modelo de primer orden: los símbolos son estadísticamente independientes. Sea p_i la probabilidad de aparición del i-ésimo símbolo del alfabeto. La tasa de entropía viene

dada por la ecuación

$$\mathcal{H} = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{2.3}$$

Modelo de segundo orden: Sea $P_{j|i}$ la probabilidad condicional de que aparezca el símbolo j-ésimo precedido del símbolo i-ésimo del alfabeto. La tasa de se obtiene mediante

$$\mathcal{H} = -\sum_{i=1}^{n} p_i \sum_{j=1}^{n} P_{j|i} \log_2 P_{j|i} \tag{2.4}$$

Modelo de tercer orden: Sea $P_{k|j,i}$ la probabilidad condicional de que aparezca el símbolo k-ésimo precedido del símbolo j-ésimo y que éste a su vez está precedido por el símbolo i-ésimo del alfabeto. En este caso, la tasa de entropía es

$$\mathcal{H} = -\sum_{i=1}^{n} p_i \sum_{j=1}^{n} P_{j|i} \sum_{k=1}^{n} P_{k|j,i} \log_2 P_{k|j,i} \tag{2.5}$$

Modelo general: Supongamos que B_m representa los primeros m caracteres del texto. La tasa de entropía en el caso general viene dada por la siguiente expresión

$$\mathcal{H} = \lim_{m\to\infty} -\frac{1}{m} \sum p\left(B_k\right) \log_2 p\left(B_k\right) \tag{2.6}$$

En esta expresión, la suma está por encima de todos los n^m posibles valores de B_m. Es virtualmente imposible calcular la tasa de entropía de acuerdo con dicha expresión.

Todas las definiciones para la tasa de entropía escritas anteriormente son consistentes las unas con las otras. En el ámbito de la compresión de datos generalmente se trabajará con modelos de primer orden, por lo tanto, en este trabajo cuando se hable de entropía (y salvo que se diga lo contrario) se hará referencia a la siguiente definición de la misma.

Definición 2.4 (Tasa de entropía de la fuente) *Sea el conjunto de n sucesos independientes $\mathcal{A}$ y con sus correspondientes probabilidades de aparición $\mathcal{P}$. Se define la entropía de la fuente, $\mathcal{H}(\mathcal{P})$, de la siguiente manera*

$$\mathcal{H}(p_1, \ldots, p_n) = \mathcal{H}(\mathcal{P}) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{2.7}$$

2.5.2. Características de la entropía

La función de la entropía, $\mathcal{H}$, tiene varias propiedades que pueden resultar de utilidad. Dichas propiedades se enunciarán a continuación, para una descripción más detallada consultar [Dro02].

I. La función $\mathcal{H}$ es continua en el intervalo $[0, 1]$, por lo que cambios pequeños en las probabilidades están asociados con cambios pequeños en la cantidad de información.

II. La función $\mathcal{H}$ es simétrica, por lo que el orden de sus argumentos no importa.

III. La función $\mathcal{H}$ tiene límite superior e inferior:

$$0 = \mathcal{H}(1, 0, \ldots, 0) \leq \mathcal{H}(p_1, \ldots, p_n) \leq \mathcal{H}\left(\frac{1}{n}, \ldots, \frac{1}{n}\right) = \log_2 n$$

IV. Axioma del grupo. Si en el conjunto $\mathcal{A} = \{x_1, \ldots, x_n\}$ se forma un grupo $\mathcal{A}_i = \{x_1, \ldots, x_i\}$ entonces, la cantidad de información no cambia después de hacer la escisión y es igual a la información asociada con el conjunto $\mathcal{A} - \mathcal{A}_i$ más la información proporcionada por el conjunto $\mathcal{A}_i$, es decir:

$$\mathcal{H}(p_1, \ldots, p_i, p_{i+1}, \ldots, p_n) = \mathcal{H}(p_1 + \ldots + p_i, p_{i+1}, \ldots, p_n) + (p_1 + \ldots + p_i)\mathcal{H}\left(\frac{p_1}{p_1 + \ldots + p_i}, \ldots, \frac{p_i}{p_1 + \ldots + p_i}\right)$$

V. Propiedad del conjunto vacío. El valor de la entropía no se modifica cuando se añade un nuevo evento con probabilidad cero al conjunto de eventos.

$$\mathcal{H}(p_1, \ldots, p_n, 0) = \mathcal{H}(p_1, \ldots, p_n)$$

VI. La función $f(n) = \mathcal{H}\left(\frac{1}{n}, \ldots, \frac{1}{n}\right)$ es monótona creciente, es decir que

$$f(n) < f(n+i) \quad \forall\, i > 0 \ \wedge \ n > 0$$

Por lo que la cantidad de información es directamente proporcional al número de eventos equiprobables.

2.5.3. Eficiencia y redundancia de una fuente

La entropía $\mathcal{H}(\mathcal{P})$ de una fuente $\mathcal{A}$ con n símbolos-fuente y una distribución de probabilidades $\mathcal{P}$ se maximiza cuando se cumple

$$p_i = \frac{1}{n} \quad \forall\, a_i \in \mathcal{A}$$

en este caso

$$\mathcal{H}(\mathcal{P}) = \log_2 n$$

Definición 2.5 (Eficacia de la fuente) *Se define la* eficacia de la fuente *o* entropía relativa, $E(\mathcal{A})$, *como la razón entre la entropía real y la máxima entropía de la fuente*

$$E(\mathcal{A}) = \frac{\mathcal{H}(\mathcal{P})}{\log_2 n} \tag{2.8}$$

Definición 2.6 (Redundancia de la fuente) *La* redundancia de la fuente *se calcula mediante la siguiente expresión*

$$R(\mathcal{A}) = 1 - E(\mathcal{A}) \tag{2.9}$$

Nótese que se cumple que $0 < E(\mathcal{A}) \leq 1$, y por lo tanto también $0 < R(\mathcal{A}) \leq 1$. El concepto de redundancia de la fuente está ligado al exceso de terminología por parte de la fuente para producir mensajes que contienen una cierta cantidad de información. Una fuente más redundante que otra necesitará producir mensajes más largos para transmitir la misma cantidad de información.

2.6. Codificación de la fuente

En esta sección se enunciarán y comentarán brevemente ciertas propiedades y características de los códigos que se utilizan en compresión de datos. Por este motivo a este tipo de codificacion también se le conoce como *codificación sin ruido ni memoria*, pues la fuente de información es sin ruido y sin memoria. Puesto que todos los teoremas que se enunciarán en la sección son lo suficientemente conocidos no se detallarán las demostraciones o pruebas de los mismos, no obstante si algún lector está interesado en ellas las puede encontrar en [Dro02].

Consideremos el alfabeto-fuente $\mathcal{A} = \{x_1, \ldots, x_n\}$ constituido por n símbolos-fuente y las probabilidades correspondientes $\mathcal{P} = \{p_1, \ldots, p_n\}$ asociadas a los símbolos-fuente. Los símbolos-fuente se codifican mediante las palabras de código $\mathcal{C} = \{c_1, \ldots, c_n\}$ con longitudes $|\mathcal{C}| = \{|c_1|, \ldots, |c_n|\}$. Así, el símbolo-fuente x_i con una probabilidad de aparición de p_i se codifica mediante la palabra de código c_i que tiene una longitud de $|c_i|$ unidades.

Un *código* es una asociación biunívoca entre los conjuntos $\mathcal{A}$ y $\mathcal{C}$, de esta manera se asocia a cada símbolo-fuente x_i una y sólo una palabra de código c_i. Así pues, la *codificación* no es más que una aplicación inyectiva definida como

$$\psi : \mathcal{A} \to \mathcal{C}$$

Obsérvese que en este caso necesariamente el número de elementos de $\mathcal{A}$ tendrá que ser menor o igual que el número de elementos de $\mathcal{C}$. Análogamente la *descodificación* será la aplicación inyectiva inversa a la codificación definida como

$$\psi^{-1} : \mathcal{C} \to \mathcal{A}$$

A un código se le denomina *código binario* si las palabras del código se componen de bits. Por conveniencia, a partir de ahora, y salvo que se diga lo contrario, al hablar de código se hará referencia a un código binario. En compresión de datos interesa minimizar la *longitud media esperada* de las palabras de código $\mathcal{C}$ [MT02] calculada como

$$\bar{L}(\mathcal{C}, \mathcal{P}) = \frac{\sum_{i=1}^{n} p_i |c_i|}{\sum_{i=1}^{n} p_i} \tag{2.10}$$

Además para una determinada distribución de probabilidades $\mathcal{P}$ todos los códigos $\mathcal{C}$ deben cumplir la siguiente desigualdad [BCW90]:

$$\mathcal{H}(\mathcal{P}) \leq \bar{L}(\mathcal{C}, \mathcal{P})$$

Definición 2.7 (Código de longitud variable) *Un código $\mathcal{C}$ con n palabras de código es de longitud variable si al menos existen dos palabras de código que cumplan lo siguiente:*

$$\exists \quad 1 \leq i \leq n \ \wedge \ 1 \leq j \leq n \ \bullet \ |c_i| \neq |c_j|$$

Definición 2.8 (Código descifrable) *Un código $\mathcal{C}$ es* descifrable *si solo existe una manera de dividir la secuencia de palabras de código $c_{i_1} c_{i_2} \ldots c_{i_k}$ en palabras de código de forma inequívoca. Es decir, si $c_{i_1} c_{i_2} \ldots c_{i_k} = c_{j_1} c_{j_2} \ldots c_{j_k}$ entonces $\forall s, i_s = j_s$ (y por lo tanto $c_{i_s} = c_{j_s}$).*

Definición 2.9 (Propiedad del prefijo) *Un código $\mathcal{C}$ posee la propiedad del prefijo si ninguna palabra de código, c_i, se puede obtener a partir de otra palabra de código, c_j, añadiendo bits. O dicho de otra forma, ninguna palabra de código es un prefijo de otra.*

$\mathcal{A}$	p_i	Código 1	Código 2	Código 3
x_1	0.67	000	00	0
x_2	0.11	001	01	100
x_3	0.07	010	100	101
x_4	0.06	011	101	110
x_5	0.05	100	110	1110
x_6	0.04	101	111	1111
$\bar{L}(\mathcal{C},\mathcal{P})$		3.00	2.22	1.75

Tabla 2.1: Ejemplo de tres códigos libres de prefijo con su longitud media esperada, $\bar{L}(\mathcal{C},\mathcal{P})$, expresada en bits por símbolo. Extraído de [MT02].

Definición 2.10 (Código libre de prefijo) *Un código $\mathcal{C}$ es libre de prefijo (o instantáneo) si posee la propiedad del prefijo.*

Está claro que los códigos libres de prefijo son un subconjunto de los códigos descifrables. Un código libre de prefijo nos asegura que la descodificación se realizará a medida que se procese el flujo de bits. Además, como ninguna palabra de código es un prefijo de otra, no se necesitará ninguna puntuación especial para separar dos palabras de código en un mensaje codificado.

Definición 2.11 (Código de redundancia mínima) *Un código $\mathcal{C}$ es de redundancia mínima (u óptimo) para el conjunto de probabilidades $\mathcal{P}$ si $\bar{L}(\mathcal{C},\mathcal{P}) < \bar{L}(\mathcal{C}',\mathcal{P})$ para cada código $\mathcal{C}'$ libre de prefijo de n símbolos.*

O dicho con otras palabras, un código es de redundancia mínima para una distribución de probabilidades si no existe ningún otro código libre de prefijo que tenga por término medio menos bits por símbolo. Es habitual denotar a los códigos libres de prefijo y de redundancia mínima como códigos de Huffman (debido al famoso algoritmo propuesto por David Huffman [Huf52] para obtenerlos) pero, estrictamente hablando, no son lo mismo.

Los códigos de redundancia mínima fueron considerados como los mejores métodos genéricos de compresión hasta el fin de la década de 1970, cuando surgieron los nuevos paradigmas de compresión propuestos por Ziv y Lempel [ZL77, ZL78][4] que superan a los códigos de redundancia mínima en muchos aspectos. Aún así, los códigos de redundancia mínima son la mejor solución en muchas aplicaciones en dónde las técnicas de compresión de datos se aplican en sistemas de recuperación de información.

Corolario 2.1 (Código compresor) *Para que un código $\mathcal{C}$ pueda ser un código compresor, $\mathcal{C}$ deberá ser un código descifrable y de longitud variable. Además es conveniente que sea libre de prefijo y de redundancia mínima.*

Como ejemplo consideremos el alfabeto $\mathcal{A}$ con $n = 6$ que figura en la tabla 2.1. Los símbolos del alfabeto pueden ser cualesquiera y están representados por x_i en la primera columna. Las probabilidades p_i asociadas a cada símbolo están listadas en la segunda columna. Se considerará que el alfabeto de salida es binario. Finalmente la tercera, cuarta y quinta columna muestran posibles codificaciones para el alfabeto y conjunto de probabilidades.

[4] ver §3.4.2 en la página 39 y §3.4.3 en la página 41.

El código 1 es una representación binaria estándar utilizando $\lceil \log_2 n \rceil = 3$ bits para cada palabra de código. Este código no es *completo*, es decir, el código tiene prefijos que no se usan. En este caso ninguna palabra de código comienza por "11".

El código 2 es un código completo que se ha formado a partir del código 1 acortando algunas de las palabras de código a $\lceil \log_2 n \rceil = 2$ bits, manteniendo la propiedad del prefijo. Aqui se ve con claridad que al asignar palabras de código más cortas a los símbolos más frecuentes se obtiene una reducción substancial de la longitud media esperada de 3.00 a 2.22 bits por símbolo.

El código 3 también es completo, realiza más ajustes en las longitudes de las palabras del código y logra obtener la longitud media hasta 1.75 bits por símbolo. Este código es de redundancia mínima y, según lo visto, se puede considerar un código compresor.

Si para este conjunto de probabilidades calculamos el valor de la entropía mediante la ecuación 2.7 el valor que se obtiene es de 1.65 bits por símbolo. Se dice que un código es más eficiente que otro si su longitud media está más próxima al valor de la entropía. Se puede obtener una idea porcentual de la *ineficiencia* de un código relativa a la entropía aplicando la siguiente ecuación:

$$100 \times \left(\frac{\bar{L}(\mathcal{C}, \mathcal{P})}{\mathcal{H}(\mathcal{P})} - 1 \right) \tag{2.11}$$

Así pues, se puede decir que los tres códigos de la tabla 2.1 son un 82 %, 35 % y 6 % ineficientes respectivamente.

2.6.1. La desigualdad de Kraft

Si buscamos minimizar la longitud media de un código ($\bar{L}(\mathcal{C}, \mathcal{P})$ definido en la ecuación 2.10) una pregunta obvia que nos deberemos formular es: ¿cómo pueden ser de pequeñas las palabras de código de un código descifrable?

Supongamos que cada símbolo x_i tiene una probabilidad que es una potencia negativa de dos, es decir, $p_i = 2^{-k_i}$, si se sustituye ese valor en la ecuación 2.1 tenemos que $I(x_i) = k_i$ es un número entero. Si generamos las palabras de código asociadas a cada símbolo se obtendrá que $|c_i| = k_i$ bits, dando como resultado un código cuya longitud esperada es mínima y coincide con el valor de la entropía. Leon Kraft consideró esta observación [Kra49] y enunció el siguiente teorema:

Teorema 2.1 (Teorema de Kraft) *Un código binario libre de prefijo formado por el conjunto de palabras de código* $\{c_1, \ldots, c_n\}$ *debe cumplir la siguiente desigualdad*

$$\sum_{i=1}^{n} 2^{-|c_i|} \leq 1$$

Dado un conjunto de longitudes que cumplen la desigualdad, el teorema garantiza encontrar un código libre de prefijos sin necesidad de modificar dichas longitudes. Es decir, una vez que se ha obtenido el código, un mensaje formado por esas palabras de código se puede descodificar de izquierda a derecha sin ambigüedad. Por otro lado, si un código satisface este teorema no implica que dicho código sea libre de prefijos. Esta relación se puede invertir y entonces se convierte en un requisito para todos los códigos libres de prefijo: si la suma de Kraft, $K(\mathcal{C})$, definida como

$$K(\mathcal{C}) = \sum_{i=1}^{n} 2^{-|c_i|} \tag{2.12}$$

es mayor que 1, entonces el código $\mathcal{C}$ no puede ser libre de prefijo. Como ejemplo obvio, supongamos que tenemos un código formado por n palabras de código c_i de longitud constante $|c_i| = 1$. Entonces $K(\mathcal{C}) = n/2$ y un código libre de prefijo sólo es posible cuando $n \leq 2$.

En 1956 McMillan [McM56] extendió este resultado y mostró que si para un código $\mathcal{C}$ se obtiene $K(\mathcal{C}) \leq 1$, entonces siempre existirá otro código $\mathcal{C}'$ que cumpla $\bar{L}(\mathcal{C}', \mathcal{P}) = \bar{L}(\mathcal{C}, \mathcal{P})$, cuyas palabras de código tales que $|\mathcal{C}'| = |\mathcal{C}|$ y además las palabras de código de $\mathcal{C}'$ están libres de prefijo. Es decir, si encontramos un código $\mathcal{C}$ que satisface el teorema de Kraft lo podemos modificar en un código $\mathcal{C}'$ con la misma longitud media pero libre de prefijo. Karush [Kar61] desarrolló una variante de la prueba de McMillan.

Si se calculan las sumas de Kraft, $K(\mathcal{C})$, para los tres códigos de ejemplo de la tabla 2.1 se obtienen los valores 0.75, 100 y 100 respectivamente. Recordemos que el código 1 de la tabla no es completo, es decir, no se pueden descodificar todas las cadenas de código semi-infinitas. Ahora, la desigualdad de Kraft nos proporciona una definición para esta propiedad.

Definición 2.12 (Código completo) *Un código $\mathcal{C}$ es completo si y sólo si es un código libre de prefijo y su suma de Kraft, $K(\mathcal{C})$, es igual a uno.*

2.7. Taxonomía de los métodos de compresión

Los métodos de compresión se pueden clasificar dependiendo de varios criterios. Ya se ha comentado la clasificación más general de los mismos: sin pérdida y con pérdida. A continuación se verán las clasificaciones de los métodos de compresión dependiendo de la forma de modelado y forma de codificación que posean.

2.7.1. Tipo de modelado

Los métodos de compresión se pueden clasificar de acuerdo con la forma de modelado de datos utilizada, pudiendo ser estáticos, semiestáticos o adaptativos [BWC89].

El modelado estático utiliza un mismo modelo para todos los textos que se procesen, escogiendo dicho modelo cuando se instaura el método. Utilizan tablas de probabilidades prefijadas de antemano, independientes de la secuencia comprimida porque, evidentemente, se presupone el contenido de la misma. En este caso, la reducción en la cantidad de espacio que se necesita para representar los datos puede ser pequeña cuando los datos codificados no se correspondan con el modelo, esta forma de modelado se debe utilizar cuando los únicos requisitos que se exijan sean la velocidad y la simplicidad. El uso de este tipo de modelado es muy específico y un ejemplo podría ser el estándar de compresión de imágenes JPEG. En esta aplicación en concreto se conoce de antemano la función de distribución esperada para los coeficientes espectrales filtrados.

El modelado semiestático (o *probabilístico*) utiliza un modelo para cada texto, o colección. Antes de la compresión propiamente dicha se analiza el texto y se construye un modelo diferente para cada texto, o colección, el cual será utilizado tanto por el codificador como por el decodificador. En este caso las probabilidades utilizadas dependen de la secuencia comprimida, pero éstas permanecen estáticas durante el proceso de codificación. Será necesario realizar dos pasadas sobre la secuencia de símbolos a codificar, por lo que los datos a comprimir deberán de estar almacenados o transmitirse

dos veces. Esto es un inconveniente puesto que los datos no pueden tratarse como un flujo de información. En la primera pasada se calculan las probabilidades asociadas a cada símbolo y en la segunda se realiza la codificación. Será necesario suministrar el modelo, es decir las probabilidades obtenidas, al descompresor para que este sea capaz de obtener el texto original de forma correcta.

El modelado adaptativo (o *dinámico*) que partiendo de un modelo inicial cualquiera actualiza el modelo en cada lectura de un símbolo, evitando así la necesidad de un análisis previo de todo el texto. Se utilizan tablas de probabilidades dependientes de secuencias comprimidas y además, éstas tablas son dinámicas e intentan ajustarse a las características locales de las secuencias para intentar mejorar la compresión. Se pueden destacar dos cosas: en primer lugar, el hecho de tener que realizar una única pasada sobre la secuencia de símbolos a codificar puede parece una simple mejora, pero en determinadas ocasiones es esencial ya que dicha secuencia no tiene que ser almacenada para ser comprimida, y de esta forma, la compresión se puede realizar simultáneamente a la llegada de los símbolos al compresor. Es decir, se puede tratar la secuencia de símbolos como un flujo de información. En segundo lugar, debido a la actualización dinámica del modelo se produce una actualización instantánea de las probabilidades de los símbolos. Por ejemplo, puede ocurrir que un determinado símbolo aparezca muy frecuentemente al inicio de la secuencia y ocasionalmente al final de la misma. Un modelo semiestático asignará a este símbolo una frecuencia ni muy alta ni muy baja, lo que hará que el codificador asigne un código con una longitud media. Pero si se hace uso de un modelo adaptativo, al principio el tamaño del código será pequeño (cuando es más frecuente) y mas largo al final (cuando es menos frecuente). Por el contrario, los compresores adaptativos deben de resolver dos problemas: la actualización del modelo (incrementar la frecuencia de aparición de un símbolo) y la inclusión de nuevos símbolos a dicho modelo.

No es posible enunciar una afirmación genérica del estilo a "los modelos adaptativos siempre serán mejores que los no adaptativos", por la sencilla razón que al usar un modelo estático se podría tener la fortuna de codificar sólo los mensajes que se adaptan perfectamente a él. Hay un conjunto de circunstancias que permiten probar que un modelo adaptativo sólo será ligeramente peor que cualquier modelo estático. Análogamente, se puede decir que un modelo estático puede ser arbitrariamente peor que los modelos adaptativos. Una comparación analítica entre los modelos adaptativos y no adaptativos se puede encontrar en [BCW90].

Los métodos adaptativos son los que más se utilizan en aplicaciones genéricas, pues permiten que se realice la compresión en una sola pasada sobre el texto original. Además, no hay necesidad de transmitir por separado el modelo utilizado en la compresión para que el decodificador comience a trabajar. Los métodos adaptativos cambian de forma dinámica la codificación teniendo en cuenta los símbolos en el proceso de compresión, esto implica que la descodificación debe realizarse secuencialmente de principio a fin imposibilitando una descompresión aislada de un dato o fragmento de texto. Esta característica de los métodos adaptativos imposibilita su utilización en sistemas de recuperación de información basados en texto, en los que son esenciales el acceso aleatorio y la necesidad de descompresión de fragmentos de texto.

Una alternativa que permite utilizar métodos adaptativos en sistemas de recuperación de información es realizar una división en bloques del texto durante la compresión. Por el contrario, los algoritmos adaptativos no son buenos para comprimir pequeñas porciones de texto ya que en estos algoritmos el proceso de modelado va mejorando a medida que se va

procesando el texto, hecho que hace que la compresión en el inicio del proceso sea mala. Moffat y Zobel estudiaron en [ZM95] la posibilidad de utilizar algoritmos adaptativos en sistemas de recuperación de información dividiendo los textos en bloques, pero concluyeron que esta alternativa no es viable.

2.7.2. Tipo de codificación

Otra forma de clasificar los métodos de compresión es dividirlos de acuerdo con el método de codificación utilizado. En este caso, se pueden dividir los métodos en estáticos y de diccionario [BWC89].

Los métodos estadísticos asignan un código a cada símbolo del texto basándose en la probabilidad de aparición de cada símbolo: cuanto mayor sea la probabilidad de aparición de un símbolo, menor será la longitud del código comprimido que lo representa. A su vez, los métodos estadísticos se pueden subdividir en dos grupos según la forma de obtener la codificación: (1) la *codificación de redundancia mínima* y (2) la *codificación aritmética*. Generalmente los primeros se pueden utilizar con modelos estáticos, semiestáticos o adaptativos. Los segundos se utilizan con un modelado adaptativo, aunque existen versiones para modelos estáticos.

Los métodos de diccionario se denominan así porque crean un diccionario de datos durante el proceso de codificación. La codificación se realiza sustituyendo grupos de caracteres consecutivos (frases) por apuntadores a entradas del diccionario. En los métodos de diccionario, los procesos de modelado y codificación son dependientes entre sí, y los diccionarios se construyen de forma dinámica. Así pues, sólo es posible combinar los métodos de diccionario con métodos de modelado adaptativos o semiestáticos.

2.8. Eficiencia de los métodos de compresión

Aparte del tiempo de ejecución y de la cantidad de memoria requerida, los métodos de compresión se pueden analizar por la cantidad de compresión obtenida. Si se supone que n es el tamaño en bytes del fichero comprimido, u el tamaño en bytes del fichero sin comprimir y c es el número de bits usados para representar un símbolo en el alfabeto original.

La cantidad de compresión obtenida se expresa, habitualmente, mediante una de estas maneras:

Tasa de compresión se caracteriza por crecer en la medida en que los resultados mejoran y se calcula mediante la fórmula

$$\left(1 - \frac{n}{u}\right) \tag{2.13}$$

Número de bits por símbolo (BPS) indica el número de bits utilizados para representar un símbolo del texto original y se calcula mediante la fórmula

$$c \times \frac{n}{u} \tag{2.14}$$

Razón de compresión es el porcentaje que el tamaño del texto comprimido representa en relación con el tamaño del texto original y es calculado mediante la fórmula

$$100 \times \frac{n}{u} \tag{2.15}$$

Factor de compresión (X:1) indica la proporción existente entre el tamaño del fichero sin comprimir y el tamaño del fichero comprimido y se expresa típicamente de la forma `X:1` donde X es un entero y se calcula mediante la fórmula

$$X = \frac{u}{n} \tag{2.16}$$

2.9. Limitaciones de la compresión

Es matemáticamente imposible crear un algoritmo de compresión sin pérdida que comprima *todos* los archivos de un mismo tamaño por lo menos un bit. No obstante, de vez en cuando alguien dice haber inventado un nuevo algoritmo que puede hacer tal cosa. Dichos algoritmos proclaman que comprimen secuencias de datos aleatorias y se pueden aplicar recurrentemente, es decir, volver a comprimir la salida obtenida por el mismo compresor, posiblemente varias veces. Así, de esta forma, se pueden alcanzar fastásticas razones de compresión en torno al 1 %. Generalemente las personas que proponen estos métodos se retractan en unos meses; otros van mas allá e incluso los patentan (como es el caso de David C. James que obtuvo una patente en Julio de 1996 del método denominado "Hyper Space").

En otro sentido, también existen *métodos de compresión falsos* —tales como OWS y WIC— que obtienen razones de compresión increíbles. Pero no comprimen todo, sino que almacenan datos sin comprimir en ficheros ocultos o en clusters no usados. De ahí la denominación de métodos falsos y además existe el peligro de perder todos los "datos comprimidos".

2.9.1. El argumento del recuento

Quizá esta sección se podría haber llamado *el teorema del recuento* porque algunas personas piensan que la palabra "argumento" implica que es sólamente una hipótesis, no un hecho matemático probado. Actualmente, el "argumento del recuento" es realmente la prueba del teorema.

Teorema 2.2 (Teorema del recuento) *Ningún algoritmo de compresión sin pérdida puede comprimir* todos *los ficheros de tamaño mayor o igual a n bits, para cualquier entero $n \geq 0$.*

Prueba: Supongamos que un algoritmo sin pérdida puede comprimir todos los ficheros de tamaño mayor o igual a n bits. Si comprimimos con este algoritmo los 2^n ficheros que tienen exactamente n bits, todos los ficheros comprimidos deberán tener un tamaño menor o igual a $n-1$ bits. En ese tamaño se pueden encontrar $2^n - 1$ ficheros comprimidos diferentes (pues obtendremos 2^{n-1} ficheros comprimidos de tamaño $n-1$, 2^{n-2} ficheros comprimidos de tamaño $n-2$, y así sucesivamente hasta tener un fichero de tamaño cero). Entonces con al menos dos ficheros de entrada diferentes se obtiene un mismo fichero comprimido. Por lo tanto, el algoritmo de compresión no puede ser sin pérdida. □

La prueba anterior se conoce como *argumento del recuento*. Utiliza el principio conocido como *pigeon–hole* (paloma–agujero) que dice en términos coloquiales que no se pueden poner "16 palomas" en "15 agujeros" sin usar uno de los agujeros dos veces.

En otras palabras, la compresión de datos se puede considerar como una asociación entre dos conjuntos (de ficheros). Y más en concreto, la compresión sin pérdida es una

asociación biyectiva y en consecuencia ambos conjuntos deben tener el mismo número de elementos.

Se pueden obtener resultados mucho más fuertes sobre el número de ficheros que no se pueden comprimir, pero las pruebas son un poco más complejas. Por ejemplo, en la página de MINC[5] se utiliza un archivo de tamaño negativo para obtener 2^n ficheros comprimidos distindos en lugar de los $2^n - 1$ de tamaño menor o igual a $n - 1$.

Una conclusión que se puede obtener del resultado anterior es que los *algoritmos de compresión iterativos* —es decir, los que vuelven a comprimir la salida[6] obtenida por el mismo compresor varias veces— no existen o no son útiles. Estos compresores no existen por dos razones: (1) porque su existencia contradice el argumento del recuento, y (2) porque si un algoritmo compresor puede ser usado iterativamente es porque en cada iteración introduce cierto nivel de redundancia y esto implica que tarde o temprano generará una expansión. Además, existe una máxima en el diseño de compresores de datos[7] que dice que si un compresor (formado por un modelador y un codificador) consigue en dos pasos más compresión que en un único paso, para algún fichero, entonces existe un modelo tal que aplicando el mismo codificador obtendrá el mismo resultado en un solo paso. El modelo buscado debe explotar la redundancia que el otro modelo elimina en dos o más etapas.

[5] `http://www.pacminc.com`

[6] Es decir, el fichero o secuencia de datos comprimida

[7] Enunciada por J. Gailly y publicada en las "Compression FAQ".

Capítulo 3

Compresión de texto

La compresión de datos sin pérdida es el proceso a través del cual un conjunto de datos se transforma en otro más pequeño, pero conteniendo ambos la misma información. Este procedimiento se lleva a cabo mediante algún tipo de codificación que elimine las redundancias en los datos. Su principal utilidad se encuentra en el almacenamiento de información y en la transmisión de mensajes. En el primer caso se precisa una menor cantidad de espacio de almacenamiento y en el segundo, se consume un ancho de banda menor del canal de comunicación. Así pues, la compresión sin pérdida es una opción interesante para reducir costes al requerir menos espacio en disco y consumir un menor ancho de banda en un canal. El precio que se paga es el coste de computación que se necesita para comprimir y descomprimir los datos. Este inconveniente es cada vez menos significativo debido al progreso de la tecnología. De 1980 a 1995 el tiempo de acceso a disco se ha mantenido aproximadamente constante mientras que la velocidad de proceso se ha incrementado unas dos mil veces aproximadamente.

Básicamente existen dos tipos de compresores, o algoritmos, de compresión sin pérdida:

- Compresores estadísticos.
- Compresores basados en diccionario.

No parece extraño que, habiendo usado una medida de información basada en las probabilidades, nos encontremos con compresores que utilicen las propiedades estadísticas de la fuente de información para mejorar la codificación (el conjunto de símbolos de salida asociados a cada mensaje emitido por la fuente) de los mensajes de la fuente. Estos son los compresores estadísticos.

Por otro lado, los compresores basados en diccionario mantienen un diccionario de las cadenas de mensajes que han sido emitidas anteriormente por la fuente. Cada cadena está representada por un índice en el diccionario. Al procesar los mensajes de la fuente, si una cadena ya ha sido recibida anteriormente, se sustituye en la salida por el índice que ésta ocupa en el diccionario. Como los índices son normalmente más pequeños que las cadenas, se consigue compresión. Otra técnica es utilizar la propia entrada como diccionario, y codificar las repeticiones como un *salto hacia atrás* y una longitud de coincidencia.

3.1. Modelando el texto

En el capítulo anterior se comentó que los métodos de compresión de datos se pueden clasificar de acuerdo con la forma de *modelar* los símbolos del alfabeto, es decir, de acuerdo

con la forma de estimar las probabilidades de los símbolos. Esto modelos (o formas de estimar las probabilidades) también estarán presentes a la hora de comprimir texto.

3.1.1. Alfabetos de palabras

La mayoría de los algoritmos de compresión de texto realizan la compresión a nivel de carácter. Si el algoritmo es adaptativo va "aprendiendo" lentamente las correlaciones entre grupos de dos, tres, cuatro, etc. caracteres y así sucesivamente. En pocas ocasiones el algoritmo tiene la posibilidad de sacar provecho de una rango grande de correlaciones antes de que se alcance el final de la entrada o que se ocupe la memoria reservada para realizar la compresión.

Si los algoritmos de compresión de texto usaran unidades más grandes como símbolos del alfabeto de entrada en lugar de simples caracteres, serían capaces de obtener un beneficio de correlaciones de rango largo y, posiblemente, mejorar la compresión.

Cuando se van a comprimir documentos de texto se pueden utilizar palabras como símbolos del alfabeto de entrada. Utilizando la misma aproximación que Bentley et al. en [BSTW86] se puede decir que un documento de texto consiste en una alternancia de cadenas alfanuméricas y cadenas de puntuación.

Definición 3.1 (Cadena alfanumérica) *Una cadena alfanumérica es una secuencia maximal de caracteres alfanuméricos.*

Definición 3.2 (Cadena de puntuación) *Una cadena de puntuación es una secuencia maximal de caracteres no alfanuméricos.*

Se utilizará el nombre genérico de *palabra* para hacer referencia bien a una cadena alfanumérica bien a una cadena de puntuación. Esta generalización permite descomponer cualquier fichero de texto en secuencias de palabras. Las modificaciones que hay que hacer en los algoritmos basados en alfabetos de caracteres para basarlos en alfabetos de palabras son mínimas y, generalmente, repercuten sobre las estructuras de datos. Por este motivo, las descripciones de los métodos de compresión que se hacen en este capítulo son igualmente válidas tanto para alfabetos de caracteres como para alfabetos de palabras y se utiliza el término *símbolo* para hacer referencia a un elemento del alfabeto de entrada, con independecia que sea un carácter o una palabra.

3.1.2. Modelos adaptativos

Otra opción es la de hacer el algoritmo de Huffman adaptativo [RG98], en el sentido que se va construyendo el árbol de manera dinámica tanto por el compresor como por el descompresor. Así el árbol no tiene que pasarse al descompresor, se realiza un único recorrido de los símbolos codificados y además, debido a la actualización dinámica del modelo conforme la compresión, se produce una adaptación a las probabilidades instantáneas de los símbolos. Así por ejemplo, puede ocurrir que un determinado símbolo aparezca muy frecuentemente al comienzo de la secuencia y sólo ocasionalmente al final. Un uso estático del modelo probabilístico le asignará a este símbolo una probabilidad ni alta ni baja, lo que hará que el codificador le asigne un número de bits intermedio. Pero si el uso que hacemos del modelo es dinámico, la codificación asignada al símbolo puede ser corta al comienzo (cuando es más frecuente) y larga al final (cuando es menos frecuente).

En general, el compresor adaptativo alcanza mejores razones de compresión que el compresor estático a costa de ser más lento. Esto se debe, principalmente, a que la codificación de los símbolos varía durante el transcurso de la compresión. Las mejoras de los niveles de

compresión por el uso de un modelo dinámico dependen de las características globales de la entropía y locales de las secuencias codificadas.

Los compresores adaptativos tienen que resolver dos problemas:

1. La actualización del modelo. Actualizar el modelo significa incrementar la frecuencia de aparición de uno de los símbolos en comparación con el resto de símbolos que no son actualizados.

2. La inclusión de nuevos símbolos en el modelo. La inclusión de símbolos nuevos (no aparecidos hasta ese momento) puede resolverse de dos formas, que constituyen dos versiones distintas del compresor adaptativo. En la primera, el modelo probabilístico contiene todos los símbolos que son susceptibles de ser codificados. La frecuencia inicial para todos ellos es 1 puesto que la compresión no ha comenzado y no se conocen las frecuencias reales. La otra versión —más eficiente en general— parte de un modelo vacío y cuando aparece un símbolo no contemplado, se incluye en el arbol. El primer esquema se denomina *sistema de compresión con un modelo plano* puesto que el árbol de Huffman que resulta es un arbol perfectamente equilibrado, mientras que al segundo esquema se denomina *sistema de compresión con un modelo vacío*.

3.1.3. El problema de la frecuencia cero

El problema de la frecuencia cero [Rob82, CW84a, Mof90, WB91, HV92, MSWB94] tiene sus raíces en el planteamiento filosófico que hizo Kant en su *"Crítica de la razón pura"*. La importancia práctica de este problema en el contexto del modelado adaptativo en la compresión de datos es que cada vez que se codifica un símbolo, todos los símbolos que ocurran a partir de este momento deben ser predichos con una probabilidad mayor a cero. A los símbolos que todavía no han aparecido se les deberá asignar una probabilidad distinta de cero, pequeña y positiva. La dificultad es encontrar la mejor forma de estimar esta probabilidad.

El dilema es que no se tiene ninguna información sobre lo que va a ocurrir. Un principio aparentemente razonable es que a cada alternativa no conocida se le asigne la misma probabilidad. A lo largo de los años, este principio ha sido asumido por varios precursores de teorías de estimación de probabilidades. Sin embargo, a principios de 1778, C. S. Pierce apuntó que eso era una paradoja. Para entenderlo inicialmente se aceptará el supuesto y se desarrollará un ejemplo experimental que permita ver los errores en los que cae dicho supuesto.

Supongamos que se tiene una urna que contiene 100 pelotas de diferentes colores, pero no se tiene información sobre el número de pelotas de cada color. Si una persona toma una pelota de la urna y sin mostrarla dirá si es o no negra. De acuerdo con el principio, se asignarán la mitad de la probabilidad total a cada alternativa. Si a continuación esta persona dice que la pelota puede ser blanca, negra o de ninguno de estos colores, el experimento tiene ahora tres posibles alternativas y, de acuerdo con el principio, se asignará a cada alternativa un tercio de la probabilidad total. Pero se produce una contradicción, pues en la primera estimación se le asigna a la pelota negra una probabilidad de $\frac{1}{2}$ y en la segunda $\frac{1}{3}$.

En vista de esto, se puede observar la dificultad que conlleva la asignación de probabilidades a símbolos que no han aparecido. Cualquier asignación de probabilidades debe ser consecuente con las leyes de la probabilidad y si se quiere hacer una estimación se debe tener conocimiento del problema. Por ejemplo, es posible que si nos basamos en un principio que asuma que los símbolos de un alfabeto son equiprobables, se construya un alfabeto no razonable que incluya símbolos que rara vez ocurren.

Método	Prob. símbolo novel	Prob. símbolo s
A	$1/(m+1)$	$p_s/(m+1)$
B	n/m	$(p_s-1)/m$
C	n/m	$((m-n)/m)\cdot(p_s/m)$
D	$n/2m$	$(2p_s-1)/2m$
X	t_1/m	$((m-t_1)/m)\cdot(p_s/m)$
C'	$n/(m+n)$	$p_s/(m+n)$
X'	$(t_1+1)/(m+t_1+1)$	$p_s/(m+t_1+1)$

Tabla 3.1: Métodos de estimación de probabilidades de símbolos nuevos, en donde m es el número de símbolos codificados hasta el momento, n es el número de símbolos diferentes encontrados, p_s es el número de ocurrencias del símbolo s, t_1 es el número de símbolos que han aparecido una sola vez, es decir, $t_1 = |\{s \bullet p_s = 1\}|$.

Se han propuesto diferentes métodos para estimar la probabilidad del *símbolo de escape*, llamado así porque su presencia en una secuencia comprimida indica que el modelo va a ser actualizado con un símbolo nuevo. Tal y como se esperaba, las diferentes técnicas funcionan bien con algunos mensajes y no tan bien con otros. Es posible seleccionar métodos robustos que funcionen razonablemente bien con la mayoría de los ejemplos prácticos. Sin embargo, no existe un método óptimo para estimar probabilidades.

La tabla 3.1 muestra algunos de estos métodos. En todos los casos se supone que el símbolo que se codificará será el $m+1$ y además que $m \geq 1$, cuando $m = 0$ la probabilidad del símbolo de escape siempre tomará el valor 1, pues el primer símbolo del mensaje siempre será nuevo.

El método A es el más sencillo pero la probabilidad que asigna al caracter de escape es muy pequeña para la mayoría de las aplicaciones prácticas. Clearly y Witten [CW84a] reconocieron los problemas del método A y propusieron el método B, el cual no tiene en consideración a un símbolo hasta que no aparece por segunda vez, por este motivo el método B tiene un considerable coste añadido. Para eliminar este coste Moffat propuso el método C [Mof90] pero tiene el problema añadido de necesitar un caso especial cuando $m = n$ y también cuando $m = 0$. Para solucionarlo Moffat propuso el método C' que obtiene una probabiliad para el caracter de escape ligeramente menor de la que debe ser. El método D se obtuvo como resultado del trabajo Paul Howard y Jeff Vitter [HV92] obteniendo mejoras pequeñas pero consistentes respecto al método C en modelos basados en PPM. El último de los estimadores de escape, método X, fue producto de un estudio de Witten y Bell [WB91] acerca del problema de la frecuencia cero mejorando significativamente las razones de compresión. El método X tiene un problema cuando $t_1 = 0$ y para solucionarlo, los autores propusieron el método X'. Una amplia descripción y una comparación entre los diferentes métodos se puede encontrar en [MT02].

3.1.4. Modelo de alfabetos separados

Los métodos estadísticos presentan la ventaja de ser independientes del modelo utilizado. Con esto se pueden escoger modelos que utilicen conjuntos de símbolos de forma que faciliten la recuperación de información sobre textos comprimidos. Una buena idea para comprimir textos es el modelo basado en palabras [BSTW86], el cual considera las palabras del texto como símbolos durante el proceso de modelado, sustituyendo la idea tradicional de utilizar caracteres. Uno de los motivos del éxito de este nuevo modelo es que

muchos sistemas de recuperación de información utilizan palabras como principal unidad de información para el acceso a la base de datos textual. Por consiguiente, los modelos que utilizan las palabras como símbolos obtienen mejores razones que compresión que las obtenidas por los modelos basados en caracteres [BSTW86, DPS99]. Esto ocurre porque los modelos basados en palabras proporcionan mucha más información sobre la estructura semántica del texto de la que se podría obtener si se utilizan caracteres como símbolos.

Los textos en lenguaje natural se componen de palabras, pero también se componen de cadenas de caracteres que separan las palabras unas de otras. La solución que propone Bentley [BSTW86] para resolver este problema es la de crear dos alfabetos de símbolos disjuntos: uno para modelar las palabras y otro para modelar los separadores. Los métodos de codificación que utilizan este modelo tratan al texto como dos fuentes de datos distintas que se intercalan, y codifican estas secuencias de forma independiente. Esta idea se denominará en este trabajo como *modelo de alfabetos separados*.

La figura 3.1 muestra un ejemplo de la utilización del modelo de alfabetos separados en conjunto con el método de codificación propuesto por David Huffman [Huf52]. En el ejemplo, el conjunto de símbolos para el alfabeto de palabras es {`para`, `cada`, `es`, `una`, `rosa`} con frecuencias 1, 1, 1, 2 y 3 respectivamente; y el conjunto de de separadores es {",␣", "␣"} con frecuencias 1 y 6 respectivamente (␣ representa un espacio).

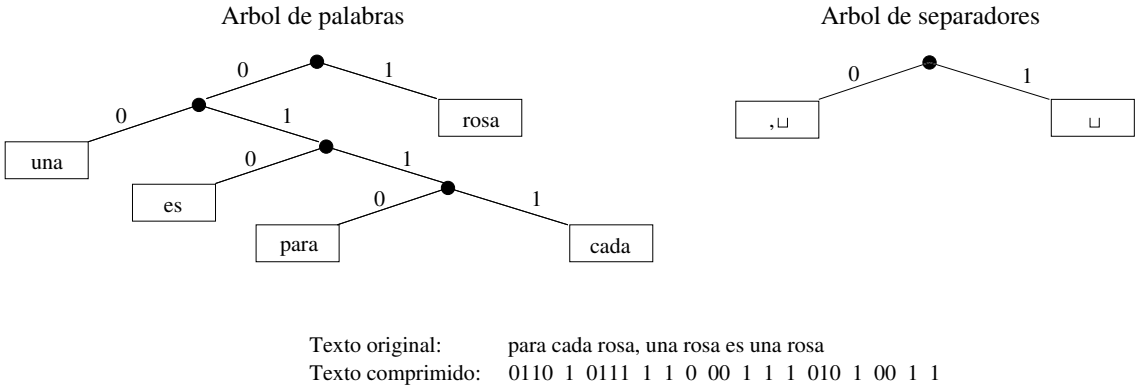


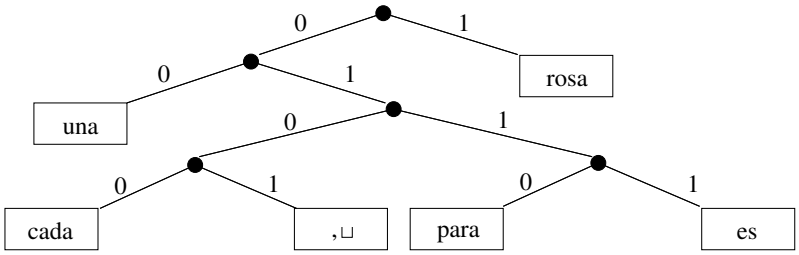
Figura 3.1: Compresión usando codificación de Huffman aplicando el modelo de alfabetos separados

3.1.5. Modelo sin espacios

Un factor importante que no tiene en cuenta el modelo de alfabetos separados es que en la mayoría de los casos a una palabra la sigue un único espacio en blanco como separador. Habitualmente, al menos el 70 % de los de los separadores que aparecen en textos escritos se componen de un único espacio en blanco [Mof89]. Teniendo en cuenta esta información, Silva propone en [Mou99] un nuevo modelo de datos que coloca las palabras y los separadores en un único alfabeto y representa los espacios en blanco de forma implícita, de ahí que el nombre de este modelo se conozca como *modelo sin espacios*.

La figura 3.2 muestra un ejemplo del modelo sin espacios combinado con el método de codificación de Huffman. La idea es sencilla y fácil de implementar. Cada vez que el codificador se encuentre con un separador compuesto únicamente por un espacio en blanco lo ignorará. Sólo es necesario tratar dos casos durante la codificación: si el texto comienza con un espacio en blanco se considerará que el texto comienza con una palabra vacía, y si

el texto termina con un espacio en blanco se inserta una palabra vacía al final del texto. Durante el proceso de decodificación se insertará un espacio en blanco entre dos ocurrencias consecutivas de palabras que no contengan un separador entre ellas.



Texto original: para cada rosa, una rosa es una rosa
Texto comprimido: 0110 0100 1 0101 00 1 0111 00 1

Figura 3.2: Compresión usando codificación de Huffman aplicando el modelo sin espacios

La unión de los símbolos en un único alfabeto produce un aumento en el número de bits que se necesitan para codificar cada símbolo. No obstante, cuando se combina con la propiedad de alternancia entre palabras y separadores, permite que el codificador represente implícitamente los espacios en blanco que aparecen en el texto original. Esto significa que ningún código se asociará al espacio en blanco y que cerca del 35 % del total de los símbolos del texto se representarán sin coste alguno. Puesto que tanto el vocabulario de palabras como el vocabulario de separadores presentan un crecimiento sublinear en torno a $O(\sqrt{s})$ —siendo s el tamaño del texto sin comprimir— se puede decir que el mismo fenómeno también ocurre en la unión de estos conjuntos.

Las dos principales ventajas al utilizar el modelo sin espacios están relacionadas con la reducción del número de símbolos del texto que se le pasan al codificador. La primera ventaja aparece cuando se utilizan codificadores que presentan un alto coste computacional por símbolo codificado, en este caso la utilización del modelo sin espacio puede reducir el tiempo de codificación. La segunda ventaja surge cuando el método de codificación codifica los símbolos ineficientemente, en este caso la aplicación del modelo sin espacios reduce las razones de compresión obtenidas.

3.2. Codificación de redundancia mínima

El objetivo de la codificación de redundancia mínima es asignar una codificación de redundancia mínima[1] al conjunto de mensajes que puede generar la fuente. Si utilizamos un alfabeto binario la longitud media esperada de las palabras de código que se obtiene (tomando como probabilidades las de los mensajes a los que representan) se debe acercar a la cota teórica (entropía) de Shannon [Sha48]. Si es igual, la compresión es perfecta: hay una máxima eficiencia.

Para poder realizar una codificación de este tipo se debe partir de:

- Una fuente de información de n mensajes.

[1] ver definición 2.11

- Las probabilidades de aparición de cada mensaje de la fuente (que pueden ser extraídas a priori de forma experimental o pueden ser dadas y fijas)
- Un alfabeto de salida que consta de una serie de símbolos. Habitualemente se utilizará el alfabeto binario (que consta de los símbolos "0" y "1"), por lo que obtendremos códigos binarios.

La forma de obtener un código de redundancia mínima es construir un árbol que represente la codificación de los mensajes de la fuente, de manera que los nodos hoja contengan cada uno de los símbolos de la fuente. Por lo general se trabajará con un alfabeto de salida binario y por lo tanto, de cada nodo intermedio partirán dos ramas, una para el bit "0" y otra para el bit "1". El código para cada mensaje se construye siguiendo el camino desde el nodo raíz hasta la hoja que representa el mensaje.

A la hora de descomprimir, el descodificador (descompresor) debe poseer el mismo árbol que se creó en el proceso de codificación (compresión): para descodificar sólo hay que leer en orden los bits del canal y seguir el camino desde la raíz hacia abajo bifurcando hacia un lado o hacia otro dependiendo del valor del bit. Eventualmente llegaremos a la hoja que representa el mensaje que se está recibiendo.

Lo verdaderamente importante es que la codificación resultante de la aplicación de estos algoritmos asigna longitudes de código inversamente proporcionales a la probabilidad de aparición de cada mensaje. Es decir, el mensaje que más aparezca tendrá un código más corto, con lo que se logra ahorrar espacio. Recuérdese que los mensajes con más probabilidad son los que menos información suministran, por eso, se utilizan menos símbolos del alfabeto de salida en su codificación.

Existen dos formas de obtener códigos de redundancia mínima: la codificación de Shannon–Fano y la codificación de Huffman. Ambos construyen el árbol de códigos manera que los mensajes con menor probabilidad quedan más abajo en el árbol, sin embargo, Huffman lo hace *botton–up* y Shannon–Fano *top–down*. No obstante, la principal y más importante diferencia entre ambos es que la codificación de Shannon–Fano obtiene códigos muy eficientes[2] pero generalmente no garantiza que se obtenga un código óptimo, por eso se dice que obtiene códigos subóptimos. En cambio, la codificación de Huffman siempre obtiene códigos óptimos y la eficiencia obtenida por sus códigos mejora levemente a la eficiencia obtenida por los códigos de Shannon–Fano, aunque ambas están muy próximas en el caso de diferir.

3.2.1. Codificación de Shannon–Fano

Esta codificación fue la primera que obtuvo buenos resultados y se originó como consecuencia de trabajos independientes de Claude Shannon [Sha48] y Robert Fano [Fan49], de ahí su nombre. La idea principal del algoritmo es que cada posición de bit en una palabra de código debiera corresponder a una elección entre grupos de símbolos que tuvieran aproximadamente la misma probabilidad. Para obtener las palabras de código deberemos hacer lo siguiente:

- Ordenar la secuencia de símbolos de acuerdo a sus probabilidades.
- Dividir la secuencia de símbolos en dos partes de manera que las suma de las probabilidades de los elementos de cada parte sean tan iguales como sea posible.
- Asignar un "0" al primer bit de los códigos de los símbolos de una parte y un "1" a los de la otra.

[2] El valor $\bar{L}(\mathcal{C}, \mathcal{P})$ está muy próximo a la entropía.

- Dividir los dos grupos recursivamente utilizando este mismo método hasta que sólo quede un símbolo por grupo.
- Cuando todos los grupos estén compuestos por un único símbolo, se codifica el texto original.

La figura 3.3 muestra este proceso para la distribución de frecuencias que aparece en la tabla 2.1 de la página 18. El código obtenido es exactamente el que aparece en la figura 2.1 como "código 3", y en este caso en particular, el código es de redundancia mínima.

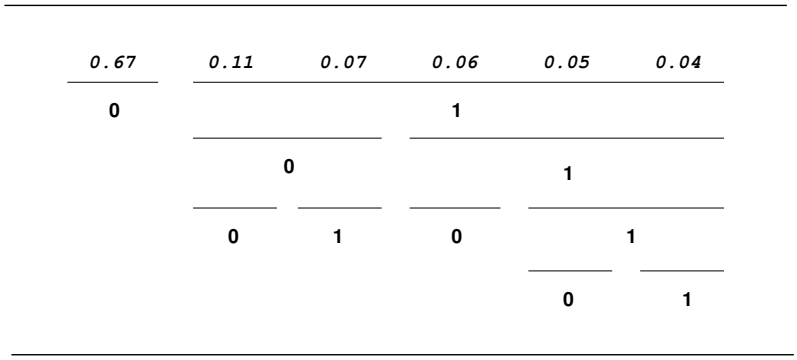


Figura 3.3: Ejemplo de uso del algoritmo de Shannon–Fano para la distribución de probabilidad $\mathcal{P} = \{0,67, 0,11, 0,07, 0,06, 0,05, 0,04\}$. El código obtenido es $\mathcal{C} = \{0, 100, 101, 110, 1110, 1111\}$

Para comprobar que el algoritmo de Shannon–Fano no es siempre efectivo se puede considerar las probabilidades $\mathcal{P} = \{0,4;\ 0,1;\ 0,1;\ 0,1;\ 0,1;\ 0,1;\ 0,1\}$. Para esta distribución se obtiene un conjunto de palabras de código descrito mediante la longitudes $|\mathcal{C}| = \{2, 2, 3, 3, 3, 4, 4\}$, con una longitud media de $\bar{L}(\mathcal{C}, \mathcal{P}) = 2{,}70$ bits por símbolo. Si ahora se considera el código libre de prefijo descrito mediante $|\mathcal{C}'| = \{2, 3, 3, 3, 3, 3, 3\}$ se obtiene una longitud media de $\bar{L}(\mathcal{C}, \mathcal{P}) = 2{,}60$ bits por símbolo, por lo tanto, el código $\mathcal{C}$ no es de redundancia mínima.

3.2.2. Codificación de Huffman

La primera forma de encontar códigos óptimos fue desarrollado por David Huffman en 1952 [Huf52]. Además puntualizó que un código óptimo debe tener ciertas características que están recogidas en el siguiente teorema.

Teorema 3.1 *Para un alfabeto-fuente $\mathcal{A}$, cuyos símbolos-fuente están asociados a una probabilidad de distribución $\mathcal{P}$, existe un código binario de redundancia mínima y libre de prefijo, $\mathcal{C}$, que cumple las siguientes propiedades:*

I. *Si $p_j > p_i$ entonces $|c_j| \leq |c_i|$.*

II. *Las palabras de código que se corresponden con los dos símbolos menos probables tienen la misma longitud.*

III. *Las dos palabras de código más largas son idénticas con excepción del último dígito.*

Este teorema permite encontrar un código para todos los símbolos de la fuente excepto para los dos menos probables, entonces es necesario extender este resultado para que incluya los dos símbolos menos probables. Se define el alfabeto-fuente $\mathcal{A}' = \{x'_1, \ldots, x'_{n-1}\}$ cuyos símbolos se ajustan a la distribución de probabilidades $\mathcal{P}' = \{p_1, \ldots, p_{n-2}, p_{n-1} + p_n\}$; es decir,

$$p(x'_k) = \begin{cases} p(x_k) & \text{si} \quad k \leq n-2 \\ p(x_{n-1}) + p(x_n) & \text{si} \quad k = n-1 \end{cases}$$

Cualquier prefijo de $\mathcal{A}'$ se puede convertir en un prefijo de $\mathcal{A}$ añadiendo simplemente un "0" a la palabra de código c'_{n-1} (que codifica a x'_{n-1}) para obtener c_{n-1} y añadiéndola un "1" para obtener c_n

Teorema 3.2 *Si el código libre de prefijo para $\mathcal{A}$ es de redundancia mínima, entonces el código libre de prefijo para $\mathcal{A}'$ también lo es.*

Estos teoremas son la base teórica del método, aunque la idea básica es muy simple y se puede describir mediante las siguientes etapas:

- Inicialmente se asigna a cada símbolo una palabra de código con cero bits de longitud. Al menos que n sea igual a 1, no se cumple la desigualdad de Kraft[3] por lo que esta asignación no constituye un código libre de prefijo. Se consideran grupos todos los símbolos individuales, evidentemente de tamaño uno, y la probabilidad del grupo como la suma de las probabilidades de sus integrantes.

- Los dos grupos con menos probabilidad se fusionan en uno y se inserta un *bit selector* precediendo a las palabras de código de todos los símbolos involucrados: el bit "0" para los códigos de los símbolos de un grupo y el bit "1" para los del otro. Como consecuencia de esta fusión se reduce el factor $K(\mathcal{C})$ y el número de grupos en una unidad.

- Se repite la etapa anterior utilizando los grupos (y las frecuencias de los mismos) modificados, hasta que sólo quede uno. En ese $K(\mathcal{C}) = 1$ por lo que tendremos un código libre de prefijo y estarán creados los códigos para cada símbolo del alfabeto.

La figura 3.4 muestra este proceso para la distribución de frecuencias que aparece en la tabla 2.1 de la página 18. En la etapa "A" muestra los grupos iniciales formados por un único símbolo, en esta etapa $K(\mathcal{C}) = 6$. En las etapas "B" a "E" se muestran las sucesivas fusiones de los grupos el valor $K(\mathcal{C})$ va disminuyendo de 5 a 2. Finalmente, en la etapa "F" sólo queda un grupo por lo que el proceso finaliza y $K(\mathcal{C}) = 1$. En la figura 3.5 se puede observar el árbol de Huffman generado para el ejemplo, las palabras de código se forman añadiendo los bits con los que están etiquetadas las ramas del camino que va desde la raíz hasta la hoja correspondiente.

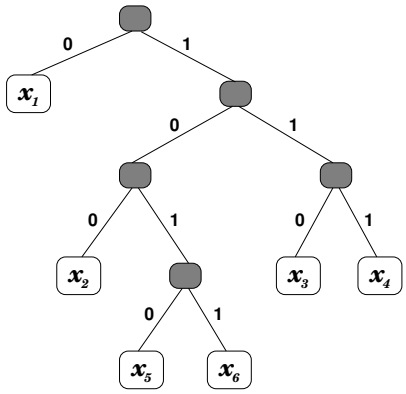


Figura 3.5: Árbol de Huffman del ejemplo

El código final obtenido difiere con el que aparece en la tabla 2.1, esto es debido a que el *bit selector* que se inserta precediendo a las palabras de código

[3] ver §2.6.1 en la página 19

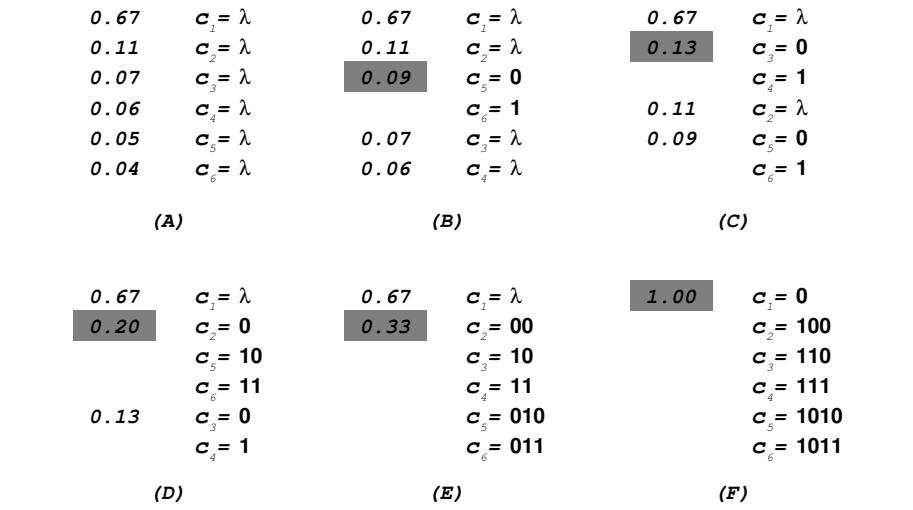


Figura 3.4: Ejemplo de uso del algoritmo de Huffman para la distribución de probabilidad $\mathcal{P} = \{0,67, 0,11, 0,07, 0,06, 0,05, 0,04\}$. El código obtenido es $\mathcal{C} = \{0, 100, 110, 111, 1010, 1011\}$. El grupo creado en cada etapa está indicado en gris. El símbolo λ representa a la cadena vacía.

cuando se fusionan dos grupos en este caso sigue la regla "el bit con valor uno en el grupo con menor probabilidad y el bit con valor cero en el grupo con mayor probabilidad", pero esto es arbitrario y se puede realizar una elección diferente en cada etapa. Sobre las $n-1$ posibles fusiones podemos generar 2^{n-1} códigos de Huffman diferentes y todos ellos son de redundancia minima. De hecho, un aspecto muy importante es que cualquier asignación de palabras de código libres de prefijo que tengan las mismas longitudes que las palabras de código obtenidas por el algoritmo de Huffman produce un código de redundancia mínima, pero no todos los códigos de redundancia mínima son uno de los 2^{n-1} códigos de Huffman.

La implementación del algoritmo es muy sencilla y eficiente si se utiliza una montículo[4] para almacenar las frecuencias de los grupos [Sed90]. En particular la raíz contendrá la menor de todas. Su eficiencia se debe a que las inserciones y extracciones en un montículo se realizan en $O(\log_2 n)$. Otra técnica de implementación del algoritmo de Huffman se puede encontrar en [MK95].

Codificación de Huffman canónica

Schwartz y Kallick [SK64] definieron una clase particularmente interesante de códigos de Huffman, conocida como códigos canónicos o estándar. Los códigos canónicos poseen propiedades matemáticas que facilitan su almacenamiento y manipulación, haciendo que los algoritmos que los generan y manipulan sean más rápidos que otros tipos de codificación y, al mismo tiempo, necesiten menos requisitos de memoria mientras se procesan.

[4]Un montículo es un árbol binario completo de tal manera que cualquier nodo es menor que cualquiera de sus hijos.

La definición original de los códigos canónicos está planteada utilizando base binaria. Sin embargo, esta definición se puede generalizar de tal forma que también se puedan incluir los códigos con una base numérica superior a 2. Dicho esto, un código de Huffman es considerado canónico si posee la *propiedad de la secuencia numérica*, que se puede detallar de la siguiente manera:

I. Los códigos se colocan en orden decreciente de tamaño según las longitudes obtenidas por el algoritmo de Huffman, manteniendo el orden lexicográfico creciente para los códigos de un mismo tamaño.

II. Todos los códigos de un determinado tamaño son números consecutivos.

III. Sea c_1 el último código de l dígitos (o de mayor valor numérico) y c_2 el primer código de $l-1$ dígitos (o el de menor valor numérico), entonces $c_1 = \mathbb{B}(c_2 - 1)$, en donde $\mathbb{B}$ es la base numérica utilizada para generar el código de Huffman.

Por ejemplo, si realizamos una codificación de Huffman canónica para la distribución de frecuencias que mostrada en la tabla 2.1 de la página 18, las longitudes de las palabras de código obtenidas por el algoritmo de Huffman son $|\mathcal{C}| = \{1, 3, 3, 3, 4, 4\}$ (figura 3.4) y por lo tanto el código de Huffman canónico que se obtiene es $\mathcal{C} = \{0, 100, 101, 110, 1110, 1111\}$. Todos los códigos de Shannon–Fano poseen la propiedad de la secuencia numérica y por este motivo a los códigos canónicos a veces también se les denomina códigos de Huffman–Shannon–Fano [Con73].

Una de las ventajas obtenidas de estas propiedades es la posibilidad de representar todos los códigos utilizando el primer valor de cada nivel y su posición dentro del código canónico, lo que representa un coste de almacenamiento cercano a $O(\log v)$, donde v es el número de elementos del alfabeto. Otra ventaja de los códigos canónicos es que no es necesario almacenar un árbol de decodificación. Un estudio de las ventajas obtenidas con el uso de la codificación canónica se puede encontrar en [HL90]. Más información respecto a su implementación, uso y representación se pueden encontrar en [ZM95, MT97, MT02].

Codificación de Huffman adaptativa

La idea fundamental de la codificación de Huffman adaptativa es que si en un árbol de codificación de Huffman un nodo hoja correspondiente a un símbolo cambia de probabilidad o *peso*, todos los nodos que forman el camino desde dicha hoja hasta la raíz en el árbol de Huffman también deben de modificar sus pesos. La reasignación de los pesos de los nodos intermedios debe forzar a reorganizar el árbol, esto tiene cierta semejanza a la generación de un árbol por el algoritmo de Huffman utilizando las nuevas probabilidades.

En 1987 Jeff Vitter [Vit87] detalló las anteriores contribuciones de Newton Faller [Fal73], Robert Gallager [Gal78] y Donald Knuth [Knu85]. Las tres técnicas utilizan las probabilidades como estimadores. Vitter examinó este algoritmo FGK en detalle y lo mejoró para el peor caso. Su trabajo culminó con una implementación disponible en [Vit89]. Al igual que el algoritmo FGK, el programa de Vitter supone que los pesos de los símbolos que aparecen en las hojas cambian de una en una unidad. El caso más general, en el que la probabilidad del símbolo puede cambiar una cantidad arbitraria, fue discutido por Cormack y Horspool [CH84] y utilizan las mismas ideas básicas.

La principal diferencia entre el algoritmo FGK y el método de Vitter es la forma de tratar los símbolos noveles. Los primeros métodos aumentaban el alfabeto fuente añadiendo un símbolo especial con frecuencia cero (denominado *símbolo de escape*), al cual se le asigna una palabra de código al introducirlo en un árbol de Huffman. Cada vez que aparece un

símbolo nuevo se reemplaza este nodo hoja por un nodo interno con dos hijos. En uno de los nodos hijos permanecerá el símbolo de escape con frecuencia cero, mientras que al otro se le asigna el símbolo nuevo con frecuencia 1. Esta aproximación es muy similar al método de escape A[5].

La alternativa es incluir un símbolo de escape explícito e incrementar su frecuencia cada vez que se emplea (método C'). La descripción de la codificación de Huffman adaptativa usa este segundo método, así pues el valor de la frecuencia de cada símbolo en el árbol es positivo y mayor que cero. Milidiú et al. [MLP99] calcularon el coste exacto de un bit en la codificación adaptativa de Huffman.

El concepto básico para entender la codificación de Huffman adaptativa se conoce como la "propiedad de hermandad", a esta propiedad le sigue un teorema.

Definición 3.3 (Propiedad de hermandad) *Un árbol binario en cuyos nodos se almacene una frecuencia posee la propiedad de hermandad si cada nodo tiene un hermano (excepto el raíz) y si el recorrido del árbol en anchura de derecha a izquierda genera una lista de nodos cuyas frecuencias están en orden no creciente.*

Teorema 3.3 (Teorema de Faller–Gallager) *Un árbol que posea la propiedad de hermandad es un árbol de Huffman.*

Para ilustrar el teorema con un ejemplo consideremos el código de Huffman de la figura 3.4 y su correspondiente árbol de Huffman en la figura 3.5. El peso de los nodos no aparece explícitamente pero supongamos que se han procesado 100 símbolos, entonces el peso de cada nodo hoja es su frecuencia multiplicada por 100 y el peso de los nodos intermedios es la suma de los pesos de sus hijos. En este caso el recorrido del árbol en anchura de derecha a izquierda nos devuelve la siguiente "lista de hermandad"

$$[100, 67, 33, 20, 13, 11, 9, 7, 6, 5, 4]$$

La idea fundamental de los algoritmos de codificacion de Huffman adaptativos es preservar la propiedad de hermandad cuando el peso de un nodo hoja se incrementa en una unidad, encontrando su nueva posición en la lista de hermanos y actualizando su posición en el árbol de forma coherente. Se pueden encontrar diferentes soluciones en [RG98, MT02].

3.3. Codificación aritmética

El tutorial de Langdon [Lan84] es una referencia importante que detalla los descubrimientos relacionados con la codificación aritmética desde los orígenes hasta 1984, fecha en la que fue publicado. En la década de los sesenta varias personas manejaron las ideas teóricas básicas en las que asienta la codificación aritmética pero sin llegar a profundizar lo suficiente [Abr63]. No fue hasta mediados de la década de los setenta cuando los trabajos independientes de Rissanen [Ris76] y Pascoe [Pas76] demostraron que se puede obtener una codificación aritmética utilizando aritmética de precisión finita. Una vez que se realizó esta importante observación se obtuvieron varias implementaciones rápidamente [Ris79, RL79, Rub79, Gua80].

A partir de este momento se abren dos líneas de investigación. Una de ellas[6], más orientada al hardware, trata de usar codificadores aritméticos binarios eficientes para utilizarlos

[5] ver §3.1.3 en la página 27

[6] En la que se trabajó en IBM con Ron Arps, Glen Langdon, Joan Mitchell, Jorma Rissanen y Bill Pennebaker entre otros.

en la compresión de imágenes [PM93], y, de forma más general, para representar datos no binarios como una secuencia de elecciones binarias [PMLA88]. La otra línea de investigación está orientada al software y a la compresión de datos en general. Tuvo su punto de partida con Witten et al. en 1987 [WNB87] mediante la publicación de una implementación completa en código C de la codificación aritmética. Desde ese momento se han realizado diferentes implementaciones y mejoras que permiten utilizar la codificación aritmética para comprimir datos sin pérdida [BCW90, HV92, HV94, MNW98].

La codificación aritmética, al igual que la codificación de redundancia mínima, se basa en las probabilidades de ocurrencia de los símbolos de entrada para generar las palabras de código, sin embargo, utiliza un esquema totalmente diferente.

Supongamos que se necesita codificar una fuente de información binaria que emite símbolos con probabilidades desiguales. La codificación de Huffman no es capaz de representar de forma eficiente esta fuente de información porque representaría a cada símbolo con un bit. Durante mucho tiempo, la solución a este problema ha sido extender la fuente en una primera etapa para luego utilizar la codificación de Huffman sobre el alfabeto extendido. Sin embargo, la eficiencia de esta solución sólo es ideal cuando el orden de la extensión es infinito y, por tanto, nunca se podrá representar la fuente sin redundancia.

Por eso se desarrolló una codificación que permitiese codificar fuentes binarias sin extender la fuente. Este tipo de codificación se le conoce como codificación aritmética y asigna a cada posible secuencia de símbolos un subintervalo dentro del intervalo $[0, 1)$ y cuyo tamaño es proporcional al producto de todas las probabilidades de los símbolos que forman la secuencia. El algoritmo básico de codificación recibe una secuencia de símbolos y emite un número cualquiera perteneciente al subintervalo final correspondiente. Teóricamente, el número de bits de la palabra de código será igual a la entropía de la secuencia[7] y por esta razón, la codificación aritmética se considera muy eficiente. A continuación se ilustrará su funcionamiento mediante un ejemplo.

Supongamos que se quiere codificar una entrada que consta de dos símbolos, X e Y, con probabilidades $p(X) = \frac{2}{3}$ y $p(Y) = \frac{1}{3}$. Al recibir una X, dividiremos el intervalo $[0, 1)$ y nos quedamos con los $\frac{2}{3}$ inferiores, por ejemplo. Tendremos entonces el intervalo $\left[0, \frac{2}{3}\right)$ que representa al símbolo X. En caso de haber recibido una Y, habríamos cogido el intervalo del tercio inferior, es decir, $\left[\frac{2}{3}, 1\right)$. En cada paso, dividimos por los $\frac{2}{3}$ inferiores o el tercio superior el intervalo que tengamos. Al final, el código que emitimos es un número binario que cae en el intervalo. Se elegirá aquel que utilice menos bits en su representación.

Con ese número (que representa toda la entrada), junto con la información del número de símbolos codificados y la probabilidad de cada uno, el descodificador puede reconstruir la entrada. Un ejemplo de la codificación de tres mensajes con la distribución de probabilidad anterior se muestra en la figura 3.6.

En la columna "Palabras de código" se especifica qué hilera de bits representa a cada grupo de tres mensajes de la fuente. El codificador construirá sólo los trozos de la tabla que le sean necesarios según la entrada. Tras recibir, por ejemplo, XXY, decide que emitirá el número $\frac{3}{8}$ (en binario .011) en representación de la entrada.

El proceso que el descompresor realizará al recibir los datos comprimidos, el número de mensajes de que consta y las probabilidades de cada mensaje, procederá como sigue: si suponemos que se recibe .011, el descompresor verá que este número pertenece a los $\frac{2}{3}$ inferiores, luego el primer mensaje que apareció fue una X. Sabe que en total son tres, por lo que continúa. También cae dentro de los $\frac{2}{3}$ inferiores de $\left[0, \frac{2}{3}\right)$, es decir, el segundo mensaje también fue una X, hasta ahora XX. Sin embargo, $\frac{3}{8}$ (.011) está en el tercio

[7] O dicho con otras palabras, cuanta más precisión tenga el número del intervalo (más decimales significativos) mayor será la información que contienen los símbolos de entrada.

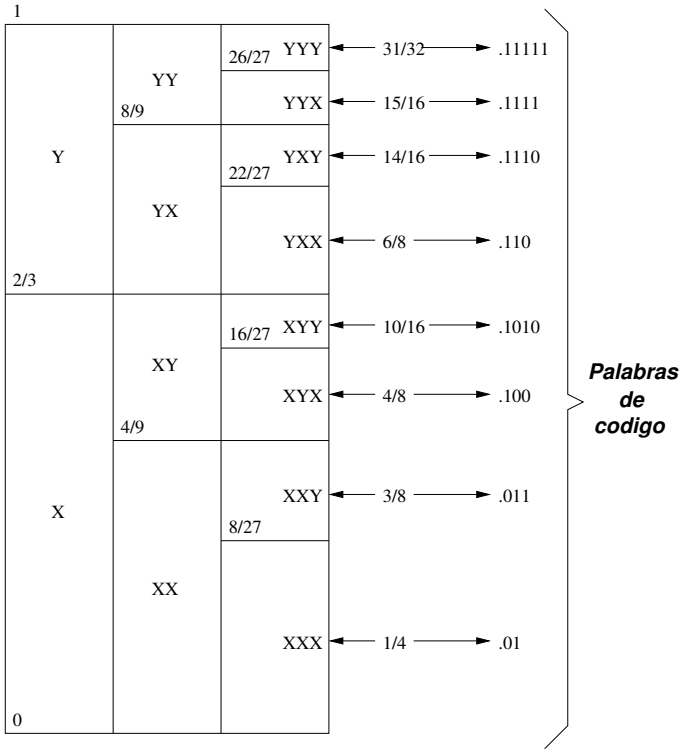


Figura 3.6: Ejemplo de codificación aritmética

superior del intervalo $\left[0, \frac{4}{9}\right)$, por lo que el tercer mensaje es una Y. Eran tres mensajes y por tanto en proceso finaliza. La secuencia original era XXY, y en este caso, ¡con tres bits hemos codificado tres bytes!.

Este algoritmo es muy eficiente, y no deja de ser curioso e ingenioso. El problema reside, al igual que los algoritmos anteriores, en que se deben de proporcionar las probabilidades al descodificador. La solución también tiende aquí hacia una modificación *adaptativa con la entrada*. Los modelos que van dividiendo los intervalos se convierten ahora en modelos adaptativos (cambiantes dinámicamente con la entrada) los cuales son capaces de sincronizar con la mínima información a compresor y descompresor.

3.4. Técnicas de diccionario

Un enfoque diferente a las técnicas anteriores es el que presentan las técnicas de diccionario. Su idea es construir un diccionario tomando como referencia la entrada procesada hasta ese momento. El diccionario contiene las cadenas de símbolos. Estas cadenas están identificadas por un índice, de manera que índice y cadena tienen una correspondencia biunívoca. El resultado práctico de todo esto es que si, en algún momento, la entrada actual que se está procesando es una cadena que está presente en el diccionario, el codificador puede emitir como salida el índice que identifica a dicha cadena en el diccionario. Teniendo en cuenta que las cadenas pueden ser arbitrariamente largas, la emisión del índice en lugar de la cadena representa un ahorro de información, y por lo tanto, compresión.

Los compresores basados en diccionario son, con mucho, los más utilizados. Normalmente se usan en combinación con una codificación de redundancia mínima en dos fases. Las dos familias más importantes de compresores basados en diccionario nacieron de los

trabajos de los matemáticos Abraham Lempel y Jakob Ziv a finales de la década de los setenta conocidos como LZ77 y LZ78 [ZL77, ZL78]. Actualmente se utilizan más los compresores derivados de LZ77, junto con una codificación de redundancia mímina a la salida. Esta estrategia también se puede aplicar en los compresores derivados de LZ78.

Dentro de este grupo se incluyen los algoritmos de codificación de secuencias, que se pueden considerar que poseen un diccionario de un byte de longitud. A continuación se describirán los algoritmos basados en técnicas de diccionario más relevantes.

3.4.1. Codificación de secuencias

Los algoritmos utilizados para codificar secuencias de símbolos son los más sencillos e ineficientes que existen. Cuando se oye por primera vez el término compresión de datos, la mayoría de la gente piensa en sustituir de alguna manera un número de caracteres repetidos por un código que indique que "el carácter X se repite N veces". Así es, este es el principal objetivo de estos algoritmos: reducir las cadenas de caracteres idénticos a una indicación del carácter y la longitud de la repetición. A continuación se comentarán brevemente los algoritmos de codificación de secuencias más relevantes.

RLE estándar

La idea básica de la técnica estándar RLE (Run Length Encoding) estándar en codificar una secuencia de símbolos repetidos usando el símbolo que se repite y el número de veces que se repite (típicamente este valor es el número de ocurrencias menos uno). Se sustituye cada secuencia por una pareja de códigos XY, dónde X representa el símbolo e Y indica el número de repeticiones. Por ejemplo, la cadena $aaabbc$ se codificará como $a2b1c0$. El descodificador realizará el proceso inverso. A partir del código $a2$ obtendrá la secuencia aaa, del código $b1$ la secuencia bb y del código $c0$ la secuencia c.

RLE binario

Si la fuente de información es binaria sólo existen dos símbolos diferentes. Por lo tanto, no es necesario indicar el símbolo X que forma la secuencia, pues las secuencias de símbolos consecutivos se intercalarán. Por ejemplo, la secuencia formada por cuatro unos, un cero y veinte unos se puede codificar por la secuencia de código "4 1 15 0 5" suponiendo que la primera serie está formada por unos y que se utilizan 4 bits para representar las longitudes de las series. La descodificación es inmediata.

RLE MNP-5

El protocolo de transmisión de datos a través del módem MNP-5 (Microcom Networking Protocol) [Hel96] utiliza una variante de RLE para comprimir secuencias generadas a partir de alfabetos de 256 símbolos. El algoritmo codifica series que tengan al menos tres caracteres, de manera que si una secuencia está formada por n símbolos x iguales ($n \geq 3$) se codificará mediante el código $xxxY$, en dónde Y representa la longitud de la serie menos 3 ($Y = n - 3$) expresada en binario natural con una precisión de 8 bits.

3.4.2. LZ77

Fué ideado por Jacob Ziv y Abraham Lempel, y publicado en [ZL77], si bien ya había sido presentado anteriormente en el IEEE International Symposium on Information

Theory celebrado en Ronneby, Suecia, en Junio de 1976. Desde entonces recibe el nombre de compresión Lempel-Ziv o, para abreviar, LZ77.

El algoritmo LZ77 mantiene un registro de los últimos caracteres procesados de la entrada pero no construye un diccionario explícto. En cada instante el algoritmo está procesando un punto de la entrada, los n caracteres anteriores forman la historia del algoritmo ó *ventana*, lo que equivale al *diccionario*. Los caracteres posteriores al punto actual constituyen lo que se denomina el *lookahead buffer* ó buffer de adelanto. En cada paso, la secuencia de símbolos que comienza en el punto actual de la entrada se busca en la *ventana*. Si se encuentra una coincidencia en la ventana que se considere lo suficientemente larga, en la salida se sustituye la cadena coincidente por un par que indica el *desplazamiento hacia atrás* y la *longitud* de la cadena coincidente. Como los pares desplazamiento–longitud ocupan menos que la cadena coincidente, se obtiene compresión. Si no se encuentra una coincidencia, la salida es una copia literal de la entrada. A continuación, se avanza la posición actual (y consecuentemente la ventana) la longitud de la coincidencia si la hubo, o bien un símbolo si no hubo coincidencia. El hecho de ir desplazando la ventana sobre la entrada hace que estos algoritmos se denominen de *ventana deslizante*.

Algoritmo 3.1 (Esquema LZ77 básico)

while (*"lookahead buffer"* $\neq \emptyset$) ***do***
    obtener una referencia (posición, longitud) para la coincidencia
    más larga en la ventana del "lookahead buffer";
    if (*longitud* $>$ *LONGITUD_MINIMA_DE_COINCIDENCIA*)
      then
        emitir el par (posición, longitud);
        desplazar la ventana "posición" caracteres;
      else
        emitir el primer caracter del "lookahead buffer";
        desplazar la ventana 1 posición;
    fi
od

El descompresor es relativamente sencillo, ya que su *ventana* está formada por los datos que ha descomprimido anteriormente, y cuando recibe un par desplazamiento–longitud sólo tiene que copiar en la salida el número de símbolos indicado en el campo de longitud comenzando desde la posición referenciada en el campo del desplazamiento. En el caso de recibir un literal, este es copiado tal cual a la salida.

Cuanto mayor sea el tamaño de la ventana, mayor será la compresión que se obtenga. Esto es así porque también será mayor la historia sobre la que se buscan las posibles coincidencias, y por tanto, se tiene una mayor probabilidad de encontrar una coincidencia más larga. Sin embargo, un tamaño de ventana grande implica que se necesite más espacio para codificar los valores de los desplazamientos. Por otro lado, las coincidencias más cortas (dos o tres símbolos) son las más probables y es deseable que se ahorre espacio cuando se codifican los pares desplazamiento–longitud de las coincidencias más probables. Esto restringe el tamaño de la ventana a un valor tal que haga que desplazamiento quepa en poco espacio. Estos dos problemas enfrentados hacen que el tamaño de la ventana sea una consideración importante. La figura 3.7[8] muestra los conceptos básicos.

Por otro lado, el tamaño mínimo de la coincidencia está muy relacionado con el tamaño de la ventana y la longitud máxima admitida de una coincidencia. Está claro que estos dos

[8]Tomado en parte de LZRW3a de Ross N. Williams [Wil91].

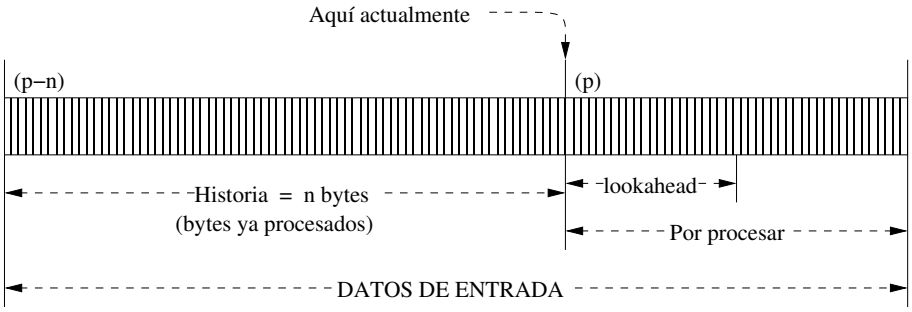


Figura 3.7: Proceso del algoritmo LZ77

últimos parámetros delimitan el número de bits que se deben de utilizar para codificar los pares desplazamiento–longitud. Para aprovechar la codificación binaria, el tamaño de la ventana debe ser una potencia de dos. Así todos los posibles valores de desplazamiento serán posibles y no se desaprovecha ninguno. Típicamente se suelen utilizar 16 bits para codificar un par desplazamiento–longitud, de estos, 12 bits se utilizan para codificar el desplazamiento, por lo que el tamaño de la ventana deberá ser de 4096 bytes, y los 4 bits restantes para codificar la longitud, por tanto la coincidencia máxima podrá ser de 15 bytes a lo sumo[9]).

Esta idea tan sencilla posteriormente ha ido tomando cuerpo con otras mejoras [RPE81, Jak85, FG89, Wil91] que, básicamente, se diferencian en la forma de buscar las coincidencias en la ventana. Esto es un aspecto muy importante a efectos prácticos, pues cuanto más eficientemente se realice esta búsqueda más rápido será el algoritmo. Otras mejoras tratan de mejorar la compresión, por ejemplo realizando una "búsqueda inteligente de las coincidencias", eso se refiere a que una coincidencia no se produce si el siguiente símbolo de la entrada produce una coincidencia que es más larga (al menos dos símbolos más) que la actual [AWG+].

3.4.3. LZ78/LZW

Terry A. Welch, de Unisys, introdujo y patentó una variante de LZ78 [ZL78] llamada LZW (Lempel–Ziv–Welch) [Wel84]. Esta variante se ha hecho muy famosa por ser la que se utiliza para comprimir en el formato de imágenes GIF. Se trata de una modificación de LZ78 que, a costa de una tasa de compresión menor, ofrece mejoras en cuanto a velocidad y uso de memoria, lo que hizo que se usara en los modems que soportan el protocolo V42bis. La variante LZW también está implementada en el programa *compress* de UNIX [TMD+91]. La amplia difusión que ha tenido esta variante ha hecho que, en la mayoría de los casos, cuando se habla de LZ78 indirectamente se está haciendo referencia a LZW.

El propósito del algoritmo es construir un diccionario en el que se guardan todas las cadenas (secuencias de símbolos) que han aparecido en la entrada. A cada cadena se le asigna un identificador (un número) que la representa. Al ir codificando la entrada, si nos encontramos con una cadena que ya está en el diccionario, la salida del algoritmo será el identificador de la cadena en el diccionario. El descompresor debe ir construyendo el mismo diccionario que el compresor, pero la ventaja es que el diccionario no tiene que pasarse de forma explícita ya que éste está implícito en la codificación.

[9]En realidad 18 bytes ya que, con estos valores, el tamaño mínimo de la coincidencia debe ser de 3 bytes para que haya compresión y dicho tamaño mínimo se supone implícito en esa magnitud.

Algoritmo 3.2 (Esquema LZW)

$\mathcal{D} \leftarrow \mathcal{A}$
$w \leftarrow \emptyset$
while $\neg EOF$ ***do***
    leer caracter K
    if $wK \in \mathcal{D}$
      then
        $w \leftarrow wK$
      else
        emitir el código de w
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{wK\}$
        $w \leftarrow K$
    fi
od

El algoritmo 3.2 muestra esta idea de codificación de una manera más precisa. Inicialmente se han introducen en el diccionario $\mathcal{D}$ todos los símbolos del alfabeto-fuente, $\mathcal{A}$[10]. Cada carácter de la entrada se va leyendo secuencialmente y en w se va construyendo la cadena que se buscará en $\mathcal{D}$. Al principio $wK = K$ (ya que $w = \emptyset$) y se realiza la asignación $w = K$. Cuando se lee el siguiente carácter, K', se busca en el diccionario la cadena $wK' = KK'$ que, evidentemente, no está. En este caso la salida es el identificador de w (que en este caso vale K) y se añade wK a $\mathcal{D}$ (y, consecuentemente, se le asigna un identificador). A continuación el proceso se repite con el siguiente caracter hasta alcanzar el fin de la entrada.

Al igual que en el caso de LZ77, hay varios aspectos del algoritmo que están relacionados con su rendimiento y eficiencia. En primer lugar, cada vez que se lee un nuevo símbolo es necesario buscar una cadena en el diccionario, cuanto más rápida se realice esta búsqueda más rápido será el algoritmo. Típicamente se suelen utilizar técnicas de hashing [Knu73] o tries [GBYS92] para realizar este cometido.

En segundo lugar, se ha visto que el diccionario va creciendo a medida que se van insertando nuevas cadenas, pero en la práctica no se dispone de cantidades de memoria ilimitadas, por lo que se debe establecer un tamaño máximo para el diccionario. Para establecer dicho valor se debe tener en cuenta el tamaño del identificador, pues cuantas más cadenas tenga el diccionario más bits se necesitarán para representar sus identificadores y, por lo tanto, la compresión empeorará. Cuando se alcance en tamaño máximo establecido para el diccionario habrá que eliminar las cadenas almacenadas, pero ¿se deben desechar todas las cadenas del diccionario y vaciarlo por completo o se pueden aprovechar las cadenas que más se han usado y dejarlas en el diccionario? Esta es una decisión difícil y no hay solución óptima ni mucho menos. Se ha visto que si el diccionario se vacía completamente la compresión empeora aunque mejora la velocidad. Una alternativa propuesta en los algoritmos *shrinking* [PKW98] y *compress* [TMD+91] es emitir códigos especiales que indican que se deben eliminar las cadenas menos utilizadas y que se utilizan para sincronizar los diccionarios del compresor y descompresor.

Una posible mejora del algoritmo LZW consiste en considerar que los identificadores son representados por un número variable de bits. Al hacerlo así, se pueden aplicar técnicas

[10] Desde un principio, estos algoritmos estaban pensados para codificar una fuente que emite caracteres ASCII, por lo tanto $\mathcal{A}$ estará formado por todos los caracteres ASCII e inicialmente en el diccionario se introducen las 256 posibles cadenas de longitud 1.

estadísticas que reasignen los identificadores, de manera que se representen con menos bits los identificadores que más se están usando en cada instante y con más bits los que menos se usan.

Algunas de las variantes más interesantes de LZW son la codificación MW de Miller y Wegman [MW85], la codificación AP de Storer [Sto88] y la codificación Y [Ber]. El algoritmo LZW construye cadenas cortas pues en cada paso se concatena un único símbolo, por el contrario la codificación MW concatena las dos últimas cadenas en lugar de concatenar sólo el último carácter. Así se construyen cadenas más largas desde el principio y posiblemente se mejora la compresión. Por su parte, la codificación AP introduce en el diccionario todas las cadenas que se obtienen al unir la cadena que se encontró en el diccionario en la ietración anterior con todos los prefijos de la cadena que se ha encontrado en el diccionario en esta iteración. De esta forma se construyen y se introducen en el diccionario más cadenas, que posiblemente aparezcan después ya que se han creado a partir de la entrada. Por último, la codificación Y es una combinación de los anteriores, pues para cada símbolo de la entrada se construye una lista de los "prefijos de coincidencia" a los que añade el símbolo leído. Al diccionario se añaden las cadenas resultantes que no están presentes. De esta forma se intentan cubrir más posibilidades.

3.5. Otras técnicas de compresión

3.5.1. Compresores predictivos

Los compresores predictivos son, al contrario que los anteriores, totalmente adaptativos. Procuran predecir el siguiente mensaje de la entrada tomando como base de conocimiento la entrada procesada hasta ese momento (en el fondo, también se basan en probabilidades). Si el mensaje que se encuentra a la entrada coincide con el predicho, su codificación se podrá hacer con menos bits. Si no, su codificación se hará con más bits, que permitirán entonces sincronizar al descompresor para que mantenga sus tablas internas idénticas a las del compresor sin pasárselas explícitamente.

Poseen ciertas ventajas sobre los algoritmos anteriores, entre ellas su velocidad: al actuar sobre un mensaje cada vez y realizar una predicción que generalmente suele ser de cálculo sencillo, son capaces de dar una alta velocidad de compresión y descompresión. Velocidad y sencillez son dos conceptos que generalmente van unidos, por lo que estos algoritmos, a la vez que rápidos, resultan sencillos de programar. Por ello se pueden convertir en una solución barata para sistemas de compresión transparente en tiempo real, con unas relaciones de compresión aceptables.

Sin embargo, para algunas aplicaciones, no son tan aceptables las relaciones de compresión. Además, su mejora es sustancialmente difícil: la predicción es muy escurridiza en cualquier ámbito. Una idea sería introducir información adicional para cada tipo de fichero que nos diera un patrón con el que predecir en mejores condiciones.

Quizá el compresor más rápido que se haya diseñado nunca pertenece al grupo de los compresores predictivos. Se llama *predictor*, y fue inventado en 1987 por Timo Raita y Jukka Teuhola, de la Universidad de Turku, en Finlandia [RT87]. Fue patentado en 1993 por K. Thomas, y se usa en el draft de Internet *PPP Predictor Compression Protocol* [11]. Servirá para mostrar un ejemplo de este tipo de compresores.

Su método de predicción es sencillo: predice el siguiente carácter a partir de los dos anteriores de la entrada. Para ello construye una matriz de 256×256 que guarda en cada casilla $m[i, j]$ el byte que anteriormente siguió a dos entradas consecutivas de valores ASCII

[11] `ftp://venera.isi.edu/internet-drafts/draft-ietf-pppext-predictor-00.txt`

i y j. Cuando se va procesando la entrada, el algoritmo siempre sabe qué dos mensajes precedieron al actual (salvo para los dos primeros mensajes de la entrada, pero esto no es problema), por ejemplo p_1 y p_2. Con esta información, su predicción para el carácter actual, pongamos c, será $m[p_1, p_2]$. Si acierta, es decir, si $c = m[p_1, p_2]$, la salida será sólo un bit, que, puesto a 1 informa de que se ha logrado predecir el mensaje actual. Si no acierta, su salida será un bit puesto a 0, indicando que no se predijo y a continuación el mensaje no predicho. Además, actualiza la tabla para que la vez siguiente sea capaz de predecir: $m[p_1, p_2] = c$. Con esta información es capaz de sincronizar al descompresor de manera que la tabla que ambos poseen en memoria es idéntica.

El descompresor parte de la tabla vacía inicial, igual que el compresor. Al recibir un bit a 1, sabe que el mensaje que se transmitió es el que se encuentra en la tabla en la posición indicada por los dos últimos mensajes resultado de la descompresión realizada hasta el momento. Si recibe un bit a 0, sabe que la siguiente información será el mensaje tal cual, y podrá actualizar su tabla al igual que el compresor.

No es éste el único ejemplo de compresores predictivos. Hay otras alternativas, como la que se incluye en PK-ZIP 1.x con el algoritmo *Reducing*, que considera un conjunto de símbolos que suceden a cada carácter —*follower sets*— que guardan los caracteres que han seguido con más probabilidad a un carácter dado. Como el conjunto de seguidores es pequeño, para identificar un seguidor se pueden utilizar menos bits. El trabajo del compresor es aquí difícil, ya que tiene que calcular el tamaño óptimo de los conjuntos de seguidores de manera que no se pierdan bits inútilmente. El problema aquí también es que los conjuntos deben ser pasados al descompresor. Un ejemplo en pseudo-código que muestra la descompresión y la codificación que se hace del conjunto de seguidores se puede encontrar en [PKW98].

Por último, estos algoritmos son malos a la hora de controlar grandes espacios de caracteres iguales. Por ello es conveniente hacer la compresión en dos partes, como la realiza el *Reducing*, primero comprimiendo los caracteres consecutivos iguales utilizando una variante de RLE ("Run Length Encoding", comentado con anterioridad) y a la salida el compresor predictivo.

3.5.2. La transformación de Burrows–Wheeler

La transformación de Burrows–Wheeler (BWT) [BW94] se toma como base para una nueva generación de compresores de texto muy eficientes, que consiguen razones de compresión próximas a las obtenidas por los algoritmos adaptativos basados en modelos de Markov y además tienen una velocidad comparable a la familia de algoritmos LZ. Manzini demostró en [Man99] que los algoritmos basados en la transformación de Burrows–Wheeler tienen acotado el tamaño de la salida en términos de la entropía de k-ésimo orden de la cadena de entrada. También concluyó que estos algoritmos conllevan unas pequeñas sobrecargas en las salidas por lo que no son óptimos en el sentido clásico, pero en la práctica no degradan su rendimiento.

La transformación de Burrows–Wheeler es un *algoritmo reversible de ordenación de cadenas*, por lo tanto, no comprime ni descomprime niguna cadena de caracteres, sólo la transforma: la entrada del algoritmo es una cadena de caracteres y la salida es otra cadena formada por los mismos símbolos pero en diferente orden. Ese orden posibilita comprimir la cadena transformada de forma eficiente. Los autores definen ésta transformación como un algoritmo de codificación de datos sin pérdida basado en la transformación de bloques de símbolos. La secuencia que se quiere comprimir debe ser procesada de esta forma. Cuanto más grandes son los bloques, mejores razones de compresión se pueden obtener.

Esto plantea el inconveniente de que la compresión no puede realizarse de forma adaptativa y no se podría utilizar en compresores en tiempo real.

Burrows y Wheeler demostraron que es un algoritmo efectivo en la mayoría de los casos, obteniendo en el peor de los casos una complejidad en tiempo de $O(n^2 \log n)$, además sugieren que se puede implementar un árbol de sufijos para realizar dicha transformación. No obstante, existen varios algoritmos efectivos que se utilizan en la construcción de árboles de sufijos: en [McC76, Ukk95] se detallan algoritmos con una complejidad en tiempo lineal pero con un consumo elevado de memoria. Balkenhof et al. [BKS99] utilizan una versión del algoritmo de McCreight, que a su vez ésta fue modificada por Kurtz [Kur99], cuya complejidad en espacio fue reducida a $20n$ en el peor caso y en el caso medio a $10n$. Por otra parte, el algoritmo de Manber y Myers [MM93] posee una complejidad temporal en el peor caso de $O(n \log n)$ y una complejidad espacial de $O(8n)$; el algoritmo de Bentley y Sedgewick [BS97] está basado en *quicksort* y obtiene una complejidad temporal en el peor caso de $O(n^2)$ y en el caso medio de $O(n \log n)$, su principal ventaja es el bajo consumo de memoria estimado en $5n$. Por otra lado, Larsson y Sadakane demostraron que, en aplicaciones prácticas, los algoritmos con complejidad mayor a la lineal pueden ejecutarse más rápido que los algoritmos con complejidad lineal para la mayoría de las secuencias [LS99]. También propusieron un algortimo con complejidad temporal en el peor caso de $O(n \log n)$ y una complejidad espacial de $8n$.

Para comprimir un texto utilizando la transformación de Burrows–Wheeler en primer lugar será necesario transformar la secuencia de entrada utilizando la transformación directa y luego aplicando un algoritmo de compresión de texto. Para descomprimirlo será necesario aplicar el correspondiente algoritmo de descompresión y luego restaurar la secuencia descomprimida a su forma original aplicando la transformación inversa. A continuación se detallan dichas transformaciones.

Transformación BWT directa

La transformación se aplica sobre un tamaño de bloque n prefijado de antemano. Para transformar un bloque se siguen los siguientes pasos:

1. Leer la cadena S de n caracteres.
2. Construir una matriz cuadrada M de orden n tal que su primera fila sea la cadena S. La fila i es la rotación cíclica hacia la izquierda en una unidad de la cadena que se encuentra en la fila $i-1$.
3. Ordenar lexicográficamente la matriz M por filas. Este paso se realiza en un tiempo $O(n \log_2 n)$.
4. Una de las filas de la matriz contendrá la cadena original S. Sea I el índice de la fila que contiene la cadena S, comenzando a enumerar desde cero.
5. Sea L la cadena formada por los caracteres que forman la última columna ($L = \{M_{0,n-1}, M_{1,n-1}, \ldots M_{n-1,n-1}\}$). El resultado de la transformación es el par (L, I).

Transformación BWT inversa

Para describir la transformación inversa se utilizará la misma notación que en la transformación directa. El algortimo inverso recibe como entrada la salida, (L, I), del algoritmo directo y la salida obtenida es la cadena S de longitud n. Para deshacer la transformación de un bloque se siguen los siguientes pasos:

1. Se crea una cadena F ordenando lexicográficamente la cadena L. Se puede observar que cualquier columna de la matriz M es una permutación de la cadena S original. Por lo tanto, L y F son permutaciones de S. Las filas de M están ordenadas y F se corresponde con la primera columna de M.

2. Utilizando F y L se calcula un vector $\vec{T}$ que indica la correspondencia entre los caracteres de ambas cadenas, en el sentido de que si L_j es la k-ésima instancia del caracter `c` en L, entonces $\vec{T}_j = i$ en la que F_i es la k-ésima instancia de `c` en F. Nótese que $\vec{T}$ representa una correspondencia uno a uno entre los elementos de F y los elementos de L, además $F_{\vec{T}_j} = L_j$.

3. Para restaurar la cadena S original se realiza el siguiente proceso:

   a) $k \leftarrow I \ \wedge \ i \leftarrow 0$

   b) Realizar n veces: $S_i \leftarrow L_k \ \wedge \ k \leftarrow \vec{T}_k \ \wedge \ i \leftarrow i + 1$

La codificación MTF

Además de presentar la transformación, los autores también dan una idea de cómo se puede comprimir eficientemente la secuencia transformada. Dicha propuesta consiste en aplicar la codificación MTF (Move To Front) que básicamente consiste en volver a transformar los símbolos para que se puedan comprimir fácilmente mediante un algoritmo de redundancia mínima.

El algoritmo codifica la salida (L, I) obtenida por la transformada directa. Recordemos que L es la cadena de longitud n e I es el índice. Para aplicar la técnica MTF hay que seguir los siguientes pasos:

1. Se define un vector de enteros, $\vec{R}$, de tamaño n

2. Inicializar la lista $\boldsymbol{Y}$ de manera que contenga todos los símbolos del alfabeto una sóla vez.

3. Para cada valor de $i = 0, \ldots, n-1$ hacer se asigna a $\vec{R}_i$ el número de caracteres que preceden al caracter L_i en la lista $\boldsymbol{Y}$. A continuación se mueve el caracter L_i al principio de $\boldsymbol{Y}$[12].

4. La salida es el vector $\vec{R}$ de n enteros y el índice I, $(\vec{R}, I)$.

Los autores proponen que se puede aplicar una codificación Huffman o aritmética sobre el vector $\vec{R}$, considerando a cada elemento del vector como un símbolo independiente que se codificará. No obstante se puede aplicar cualquier otra técnica de codificación. Fenwick investigó la posibilidad de no aplicar la transformación MTF a la cadena L para intentar aprovechar las ventajas de la codificación de redundancia mínima adaptativa, pero su trabajo mostró que la razón de compresión empeoraba un 5-10 % [Fen96]. Schindler propuso otra modificación para esta transformación pero sin lograr ninguna mejora significativa [Sch97]. Balkenhol et al. propusieron mover al principio de la lista L (entendiéndose por principio la posición 0) sólo los símbolos que aparezcan en la posición 1, y los caracteres que se localizan en posiciones mayores se mueven a la posición 1. Esta versión mejora la razón de compresión y se denota mediante las siglas MTF-1 [BKS99].

El problema que debe resolver el algoritmo MTF se conoce como problema de actualización de la lista. Las características de los elementos que forman la lista $\boldsymbol{Y}$ hacen que

[12] Es decir, al frente del vector, de ahí su nombre.

los algoritmos clásicos para resolver este problema no obtengan buenos resultados. Albers propuso un algoritmo denominado *time stamp* que permite resolver el problema de actualización de la lista [Alb95]. Posteriormente ha comprobado que si se reemplaza el algoritmo MTF por dicho algoritmo se obtiene una pequeña mejora en las razones de compresión para algunas secuencias, mientras que en la mayoría de los casos los resultados son peores [AM96].

La descodificación MTF

El algoritmo de decodificación de MTF obtiene el par (L, I) a partir del par $(\vec{R}, I)$. Si el vector $\vec{R}$ se encuentra codificado, previamente deberá ser descodificado utilizando el algoritmo adecuado. Una vez obtenido el vector $\vec{R}$ se deben seguir los siguientes pasos:

1. Inicializar la lista $\boldsymbol{Y}$ de caracteres del alfabeto en el mismo orden que la generada por el codificador.

2. Para cada valor de $i = 0, \ldots, n-1$ se asigna a L_i el caracter que ocupa la posición $\vec{R}_i$ en la lista $\boldsymbol{Y}$. A continuación se mueve dicho caracter al principio de $\boldsymbol{Y}$.

La transformación Zero–Run

Si se estudian los valores que aparecen en las componentes de $\vec{R}$ se puede observar que por término medio un 60 % de los valores son cero, y además existen muchas secuencias de ceros seguidos (originalmente denominadas *zero–runs*). La transformación zero–run, generalmente denotada por RLE-0, se expone en [Fen96, BK00] aunque en 1995 fue propuesta por Wheeler pero nunca publicada[13]. Wheeler también propuso una codificación eficiente para dicha transformación.

Si las secuencias siempre están formadas por el valor 0 y si se utiliza el binario para codificar las longitudes de las secuencias, la transformación zero–run propone utilizar los dígitos binarios con peso 1 y 2 en lugar de los habituales 0 y 1. Por tanto, la secuencia de bits $x_0 x_1 x_2 \ldots x_{n-1}$ representa el valor

$$\sum_{i=0}^{n-1} 2^i (1 + x_i) = \sum_{i=0}^{n-1} 2^i + \sum_{i=0}^{n-1} 2^i x_i = (2^n - 1) + \sum_{i=0}^{n-1} 2^i x_i$$

Para la mayoría de los valores el bit más significativo está implícito y es necesario codificarlo. Entonces, el valor se representa empleando un número menor de bits y el valor 0 no se puede representar, pero en este caso carece de importancia. La codificación se puede realizar de dos maneras:

1. Incrementar el valor en uno y codificar el valor modificado en binario natural sin precisión pero ignorando el bit con valor 1 que tenga más peso.

2. Codificar en binario natural el número, emitir el bit de menos peso y desplazar hacia la derecha el número para eliminar ese bit. Antes de emitir un bit es necesario decrementar en una unidad el valor actual. Se repite el mismo proceso mientras el valor sea diferente de cero antes del desplazamiento.

[13] Wheeler envió dichos resultados al grupo de noticias `comp.compression.research` y están disponibles, vía FTP anónimo, en `ftp://ftp.cl.cam.ac.uk/users/djw3`

Una secuencia de dos caracteres idénticos "...cc"señalan una secuencia. La codificación de la longitud de la secuencia se realiza tal y como se ha mostrado anteriormente, utilizando dos símbolos: c y $c \oplus c1$, donde $\oplus$ denota una operación OR-exclusiva. Si el símbolo que sigue a la secuencia es $c \oplus 1$ ó $c \oplus 2$ se inserta un $c \oplus 2$ extra como terminador, por lo tanto en el decodificación se ignorará cualquier símbolo $c \oplus 2$ que finalice la secuencia.

3.6. PPM ("Prediction by Partial Matching")

La idea original la propusieron Cleary y Witten en 1984 [CW84a], posteriormente Moffat planteó e implementó una serie de mejoras sobre la idea original [Mof90]. Los resultados de dicha implementación mejoraban a todos los métodos de compresión existentes, aunque en la práctica se utilizan más los métodos basados en los esquemas de codificación Ziv–Lempel [ZL77, ZL78] ya que su atractivo reside en su velocidad en lugar de en sus niveles de compresión cuando se comparan con PPM; de hecho, su compresión es peor que la obtenida por PPM en todas las baterías de pruebas prácticas que se ha llevado a cabo [BCW90].

El modelado "Prediction by Partial Matching" o PPM es una técnica de modelado estadística de contexto finitos de caracteres que se puede ver como una mezcla de varios modelos de contexto finito de orden fijo que se utilizarán para realizar predicciones respecto al siguiente caracter de la secuencia de entrada. Las probabilidades de predicción para cada contexto del modelo se calculan a partir del número de ocurrencias de cada símbolo que se actualiza de forma dinámica, y la distribución predicha se utiliza para codificar de forma relativa el símbolo que se está procesando actualmente utilizando la codificación aritmética [WNB87, MNC95]. La longitud máxima del contexto es una constante prefijada de antemano, y se ha demostrado que generalmente para textos escritos en lenguaje natural no se mejora la compresión cuando esa constante es más grande que 5 [CW84a, Mof90, Bun97].

La idea básica de PPM es utilizar los últimos símbolos de la secuencia de entrada para predecir el siguiente. Para realizar dicha predicción PPM utiliza una serie de modelos de contexto de orden k fijo, con diferentes valores de k que varían desde cero hasta algún máximo predeterminado. Para cada modelo se guarda una relación de todos los símbolos que siguen a cada subsecuencia de longitud k observada en la entrada así como el número de veces que ha ocurrido. Las probabilidades de predicción se calculan a partir de esos valores. Las probabilidades asociadas a cada símbolo que sigue a los últimos k símbolos en el pasado se utilizan para predecir el símbolo siguiente. Por lo tanto, se obtiene una distribución de probabilidad predicha independiente para cada modelo.

Esas probabilidades se combinan de forma efectiva en una y el símbolo actual se codifica empleando la codificación aritmética empleando esa probabilidad relativa. La combinación se lleva a cabo utilizando probabilidades de escape. Por defecto, el modelo con el mayor orden k se utiliza para codificar. No obstante, si se encuentra con un símbolo nuevo en este contexto no se puede utilizar este modelo para realizar la codificación y entonces se emite un *símbolo de escape* que indicará al descodificador que debe cambiar al modelo de orden $k-1$. Este proceso continúa hasta que se llega a un modelo en el que el símbolo actual no es nuevo y, en este momento, el símbolo se codifica de acuerdo a la probabilidad predicha por dicho modelo. Para tener la certeza que el proceso termina, se supone que un modelo que contiene todos los símbolos del alfabeto de la fuente está presente por debajo del nivel más bajo. Este mecanismo mezcla de forma efectiva los modelos de diferente orden en una proporción que depende de los valores de las probabilidades de escape que se están utilizando en cada momento.

Se debe escoger cuidadosamente el método que se utilizará para asociar la probabilidad a un símbolo de escape[14] pues depende de la naturaleza de la fuente y, por lo tanto, es crucial para obtener una buena compresión. De acuerdo al método que se utilice que una implementación concreta para asignar dicha probabilidad se obtienen diferentes versiones de PPM, que generalmente se denotan mediante PPMκ, en dónde κ indica el método que se ha utilizado. John Clearly e Ian Witten obtuvieron resultados al palicar los métodos A y B [CW84a], por otro lado los métodos C y D se desarrollaron a partir de estudios sobre el paradigma de PPM [Mof90, HV92]. Actualmente se considera que el método D (con el que se forma PPMD) es la elección más apropiada para PPM en el caso general [Tea98]. Si por el contrario, no existe ninguna suposición a priori acerca de la naturaleza de los símbolos-fuente, no existe ninguna base teórica que fundamente una elección en particular; algunas alternativas se han evaluado en [WB91, Bun97].

Una alternativa factible es el denominado PPM* [CTW95, CT97], el cual no tiene límite superior para la longitud de los contextos y se mantiene la información completa todo el tiempo. Para estimar la probabilidad del símbolo siguiente se utiliza el contexto determinístico[15] más pequeño en primer lugar. Si no existe un contexto determinístico se utiliza el contexto de mayor orden disponible. PPM* mejora la compresión respecto a PPM pero consume muchos más recursos.

[14] ver §3.1.3 en la página 27

[15] Un contexto es determinístico si predice exactamente un símbolo.

Capítulo 4

Búsqueda en texto

Hoy en día es habitual encontrarnos con colecciones de documentos de gran tamaño, las cuales exigen técnicas especializadas de indexación que permitan un rápido acceso a los documentos que buscan los usuarios. La opción más trivial a la hora de encontrar una palabra dentro de una colección de texto es la de efectuar una búsqueda secuencial, o búsqueda en línea (*online search*), que consiste en ir recorriendo secuencialmente todas las palabras de todos los documentos y comprobando la coincidencia de la palabra recorrida con la palabra buscada. Este tipo de búsqueda sólo será apropiada cuando el texto sea pequeño (por ejemplo, unos pocos megabytes) y muy volátil (es decir, que se realicen modificaciones muy frecuentemente), o bien, cuando no se pueda soportar el espacio adicional que ocupa el índice. Una segunda opción es construir estructuras de datos sobre el texto, denominadas índices, que permitan acelerar la búsqueda. Será asequible construir y mantener un índice cuando el texto de la colección sea grande y semiestático, es decir, que se actualice en intervalos regulares de tiempo, como por ejemplo, diariamente, semanalmente, etc. Actualmente, la mayoría de las técnicas de indexación con más éxito que se utilizan en bases de datos de tamaño medio (en torno a unos 200 megabytes) combinan la búsqueda secuencial con técnicas de indexación.

La búsqueda de patrones cubre un ámplio rango de problemas que van desde tareas relativamente sencillas como localizar caracteres en una cadena hasta buscar ocurrencias aproximadas de un patrón complejo en una base de datos. El interés en los problemas de búsqueda de patrones más complejos se ha incrementado de manera considerable en los últimos, especialmente en el campo de de la recuperación de información y la biología computacional. Los algoritmos de búsqueda secuencial para un único patrón son los más conocido y habituales, como por ejemplo el algoritmo de Knuth–Morris–Pratt (KMP) [KMP77], el algoritmo de Boyer–Moore [BM77] y sus variantes (Boyer–Moore–Horspool, BMH [Hor80] y Boyer–More–Sunday, BMS [Sun90]) y el algoritmo Shift–Or [BYG92]. No obstante también existen otras técnicas de búsqueda de patrones como la *búsqueda de patrones múltiple* que permite buscar un conjunto de patrones como si se tratará de un único patrón. Por otro lado, la *búsqueda de patrones extendida* comprende la búsqueda de expresiones regulares y la búsqueda aproximada. La primera de ella se encarga de buscar el conjunto de patrones que se ajusten a la expresión regular dada y que aparecen en el texto, para ello se suele untilizar un autómata finito no determinista. La segunda se encarga de buscar las ocurrencias de un patrón en el texto permitiendo que dichas ocurrencias puedan tener un número limitado de diferencias respecto al patrón original. Una ámplia descripción de las búsquedas de patrones múltiple y extendida se pueden encontrar en [NR02].

4.1. Caracterizando del lenguaje natural

Las técnicas de búsqueda de patrones no son suficientes en el campo de la recuperación de información debido a que estos sistemas deben responder a las consultas de usuario en el menor tiempo posible. Para ello se utilizan índices, que son estructuras de datos persistente construidas a partir de las palabras que aparecen en los textos de una colección. No obstante, las técnicas de búsqueda de patrones se siguen utilizando en la recuperación de información para localizar patrones en regiones de texto restringidas y para realizar búsquedas por proximidad.

El texto se compone de símbolos los cuales forman parte de un alfabeto finito. Podemos dividir esos símbolos en dos subconjuntos disjuntos: los símbolos que se utilizan para separar palabras y los símbolos que pertenecen a las palabras. También sabemos que los símbolos no se distribuyen uniformemente. Si consideramos las letras (de la `a` a la `z`), podemos observar que las vocales se utilizan más a menudo que las consonantes. Por ejemplo, en castellano la letra que aparece mas frecuentemente es la `a`, mientras que en inglés es la letra `e`. El modelo binomial es un modelo sencillo que se utiliza para generar texto. En él, cada símbolo se genera con una determinada probabilidad. No obstante, en el lenguaje natural existe una dependencia con los símbolos anteriores y además las vocales y ciertas consonantes tienen una alta probabilidad de aparecer. Por ejemplo, en castellano la letra `n` no puede aparecer antes de la letra `p`, en inglés la letra `f` no puede aparecer antes de la letra `c`. Se puede utilizar un modelo de contexto finito o modelo de Markov[1] para reflejar esta dependencia. El modelo puede utilizar una, dos o más letras para generar el símbolo siguiente. Si utilizamos k letras, diremos que el modelo es de orden k (así, el modelo binomial se considera como un modelo de orden 0). Se pueden usar estos modelos tomando las palabras como símbolos. Otros modelos mas complicados son los modelos de estados finitos que generan lenguajes regulares, y los modelos de gramática que definen lenguajes libres de contexto y otros. Sin embargo, el hecho de encontrar una gramática que genere el lenguaje natural es todavía un difícil problema abierto.

La siguiente cuestión es saber cómo están distribuidas las diferentes palabras dentro de cada documento. Un modelo aproximado es la *ley de Zipf* [Zip49], que intenta obtener la distribución de las frecuencias (número de ocurrencias) de las palabras en el texto. La ley afirma que la frecuencia de la i-ésima palabra más frecuente es $\frac{1}{i^\theta}$ veces la frecuencia de la palabra más frecuente. Esto implica que en un texto de n palabras con un vocabulario de V palabras, la i-ésima palabra más frecuente aparece $f(i)$ veces, siendo

$$f(i) = \frac{n}{i^\theta H_V(\theta)} \tag{4.1}$$

en donde $H_V(\theta)$ es el número armónico de orden θ de V definido como

$$H_V(\theta) = \sum_{j=1}^{V} \frac{1}{j^\theta} \tag{4.2}$$

es decir, la suma de todas las frecuencias es n. El valor de θ depende del texto. En la formulación más sencilla $\theta = 1$ y por consiguiente $H_V(\theta) \cong O(\log n)$. No obstante esta versión simplificada es muy inexacta, y el caso $\theta > 1$ (concretamente, entre 1,5 y 2,0) es mucho más adecuado para los datos reales, con una distribución muy sesgada, y en este caso $H_V(\theta) \cong O(1)$. Un modelo mejor obtenido a partir de datos experimentales, denominado

[1] ver §2.5 en la página 13

distribución de Mandelbrot, es

$$f(i) = \frac{k}{(c+i)^{\theta}} \tag{4.3}$$

en donde c e i son parámetros adicionales cumpliendo que $c > 0$ y $0 \leq i < 1$, y k es del orden de la suma de las frecuencias más el número de palabras del vocabulario (n).

Otro tema es la distribución de las palabras dentro de los documentos de la colección. Un modelo sencillo es considerar que cada palabra aparece el mismo número de veces en cada documento de la colección. Sin embargo, en la práctica esto no es cierto. Un modelo mejor es considerar una distribución binomial negativa, que dice que la fracción de los documentos que contienen una palabra κ veces es

$$F(\kappa) = \binom{\alpha + \kappa - 1}{\kappa} p^{\kappa}(1+p)^{-\alpha-\kappa} \tag{4.4}$$

en donde p y α son parámetros que dependen de la palabra y del documento de la colección. Las últimas referencias dan modelos derivados de la distribución de Poisson.

El siguiente punto que se comentará es el número de palabras distintas que contiene un documento. Este conjunto de palabras se conoce como *vocabulario del documento*. Para predecir el crecimiento del tamaño del vocabulario en textos en lenguaje natural, se utilizará la llamada *ley de Heaps* [Hea78]. Es una ley muy precisa que afirma que el vocabulario de un texto de n palabras es de tamaño

$$V = Kn^{\beta} \cong O(n^{\beta}) \tag{4.5}$$

en donde K y β dependen del texto en cuestión. Generalmente K varia entre 1 y 100, y β es un valor positivo menor que uno. Experimentalmente se ha visto que:

1. los valores más habituales para K oscilan entre 5 y 50
2. los valores de β están comprendidos entre 0,4 y 0,6
3. V es menos del 1 % de n.

Por lo tanto, el tamaño del vocabulario de un texto crece sublinealmente con el tamaño del texto, en una proporción en torno a la raíz cuadrada de su tamaño. Nótese que el conjunto de palabras diferentes de un lenguaje está fijado por una constante, por ejemplo, el número de palabras diferentes en castellano es finito. Sin embargo, el límite es tan elevado que es mucho más acertado asumir que el tamaño del vocabulario es $O(n^{\beta})$ en lugar de $O(1)$, aunque dicho número se pudiera afianzar para un número suficientemente grande de textos. Por otro lado, muchos autores argumentan que dicho número hay que mantenerlo elevado en cualquier caso debido a que existen errores ortográficos o mecanógrafos. La ley de Heaps se aplica a colecciones de documentos porque las predicciones del modelo son más exactas debido a que el tamaño total del texto aumenta. Por otra parte, este modelo también es válido cuando se aplica a la web.

Finalmente, el último aspecto a tratar es la longitud media de las palabras. Esto relaciona el tamaño del texto medido en palabras con el tamaño del texto medido en bytes, sin tener en cuenta los signos de puntuación y otros símbolos extra. Lo que nos permite definir el espacio total que necesita el vocabulario, excluyendo las palabras vacías. La ley de Heaps conlleva que la longitud de las palabras en el vocabulario se incrementa logarítmicamente con el tamaño del texto. En la práctica, la longitud media de las palabras en el global del texto es constante debido a que las palabras más cortas son bastante comunes (sobre todo las palabras vacías). Este equilibrio entre las palabras cortas y las palabras largas, tal

que la longitud media de palabra permanezca constante, se ha expuesto muchas veces en diferentes contextos, y además se puede expresar mediante un modelo de estados finitos en el que el carácter espacio no puede aparecer dos veces seguido y tiene una probabilidad cercana al 0,2 y hay 26 letras. Este modelo sencillo es consecuente con las leyes de Zipf y Heaps.

4.2. Índices invertidos

El índice invertido, o fichero invertido, es la estructura más elemental para la recuperación de palabras, aparte de ser una estructura de indexación sencilla y popular que se puede aplicar a grandes colecciones de textos en lenguaje natural.

Típicamente un índice invertido se compone de un vector que contiene todos los términos diferentes que aparecen en los textos de la colección en orden lexicográfico, lo que se denomina *vocabulario* y que se encuentra almacenado en un *diccionario*. A cada término del vocabulario se le asocia una lista con todas las posiciones en las que aparece dicho término en el texto. A cada lista se le denomina *lista de ocurrencias* o *lista de posiciones*; y al conjunto de todas esas listas se le denomina *ocurrencias*. Esas posiciones pueden hacer referencia a palabras, a caracteres o a documentos.

Para buscar un término en un índice invertido se deberá de localizar, en primer lugar, dicho término en el diccionario y posteriormente recuperar la lista de ocurrencias. Para buscar una frase o un patrón de proximidad (o dicho de otra manera, se buscarán varios términos consecutivos o cercanos, respectivamente) cada término se debe de buscar por separado para luego intersectar las listas de ocurrencias según la consecutividad o la proximidad de las palabras en el texto. Dependiendo con qué se correspondan las posiciones en una lista de ocurrencias se facilitará un tipo de búsqueda u otro. Así, las ocurrencias a documentos —la posición i se refiere al i-ésimo documento— agilizan la localización y ordenación de documentos relevantes, las ocurrencias a palabras —la posición i se refiere a la i-ésima palabra— simplifican las consultas de proximidad, mientras que las posiciones a caracteres —la posición i se refiere al i-ésimo carácter facilitan el acceso directo al texto para emparejar posiciones de texto —*matching text positions*—.

La *ley de Zipf*[2] muestra que la distribución de las palabras dentro de un texto está muy sesgada —hay unos pocos cientos de palabras que se hallan en un 50 % del texto y el resto aparece poco— las palabras muy frecuentes, al igual que las palabras vacías, se pueden ignorar. Una palabra vacía —*stopword*— es una palabra que no suministra significado en el lenguaje natural y por lo tanto se puede ignorar ya que no debiera ser objeto de búsqueda; estas palabras son artículos, preposiciones y otras palabras que no aportan relevancia dado que son demasiado comunes. Afortunadamente, las palabras más frecuentes son palabras vacías y por lo tanto, aproximadamente la mitad de las palabras que aparecen en el texto no se considerarán a efectos de búsqueda. Esto permite reducir significativamente el espacio adicional que ocupan los índices al trabajar con textos en lenguaje natural.

4.2.1. Índices clásicos

Tal y como se ha visto en el apartado anterior, el espacio requerido por el vocabulario es relativamente pequeño de acuerdo con la ley de Heaps y en la práctica se puede ubicar en la memoria principal del ordenador (por ejemplo, 5 megabytes para 1 gigabyte de texto). Por otro lado, las listas de ocurrencias necesitan mucho más espacio.

[2] ver §4.1 en la página 52

Las consideraciones más importantes para evaluar la eficiencia de un esquema de indexación son los requisitos de espacio y los tiempos de construcción, actualización y búsqueda. Tanto los requisitos de tiempo como los de espacio están en función de la granularidad del índice, la cual define qué unidad de información se representa. Se pueden identificar varios tipos de índices invertidos dependiendo de su granularidad [BYRN99], que pueden variar desde los índices que permiten procesar rápidamente las consultas pero que son lentos en su construcción y demandan mas espacio hasta los que son lentos a la hora de procesar consultas pero rápidos en la construcción del índice y demandan menos espacio.

El primero de ellos se conoce como índice invertido con direccionamiento a palabra (*word addressing inverted index*). Si está implementado adecuadamente es el más rápido a la hora de resolver la mayoría de las consultas. Utiliza un esquema muy simple, en el cual el índice apunta a todas las posiciones de todas las palabras del texto. Lo más probable es que su tiempo de construcción y los requisitos de espacio que demande serán elevados. Las ocurrencias pueden rondar en torno al 60 % del tamaño del texto. Esto se puede reducir hasta el 35 % omitiendo las palabras vacías (*stopwords*) del vocabulario. Las palabras vacías representan del 40 % al 50 % de todas las palabras del texto. Sin embargo, la disposición de un 35 % de tamaño extra puede suponer todavía un requisito elevado de espacio si disponemos de una colección grande, y en consecuencia existen diferentes técnicas que nos permitirán reducir el espacio que ocupan las listas de ocurrencias.

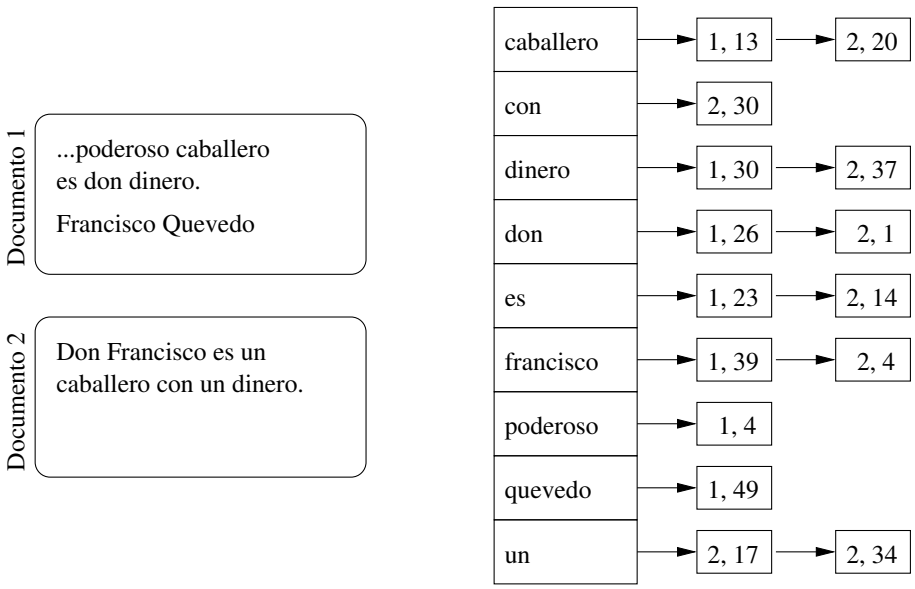


Figura 4.1: Ejemplo de índice invertido con direccionamiento a palabra. En cada nodo de la lista de ocurrencias se almacena el documento y la posición en los que aparece la palabra.

El segundo tipo de índice se conoce como índice invertido con direccionamiento a documento (*document addressing inverted index*). En este tipo de indexación, las listas invertidas no apuntan exactamente a las palabras sino que apuntan a los documentos en los que aparecen dichas palabras. Esto nos permite ahorrar espacio por que todas las ocurrencias de la misma palabra en el mismo documento están referenciadas sólo una vez, y los punteros debieran de ser más pequeños porque hay menos documentos que posiciones de texto. Los requisitos habituales de espacio para el direccionamiento de documentos se encuentran en torno al 25 % del tamaño del texto. Las consultas que constan de una única palabra se

resuelven directamente sin necesidad de acceder al texto. Esto es así porque si una palabra aparece en un documento este se recupera en su totalidad. Por el contrario, las consultas de frase o de proximidad no se pueden resolver con la información almacenada en el índice. Dos palabras pueden estar en el mismo documento pero pueden o no pueden formar una frase o estar cercanas. Para este tipo de consultas deberemos de buscar directamente en el texto de los documentos en los que aparezcan todas las palabras relevantes.

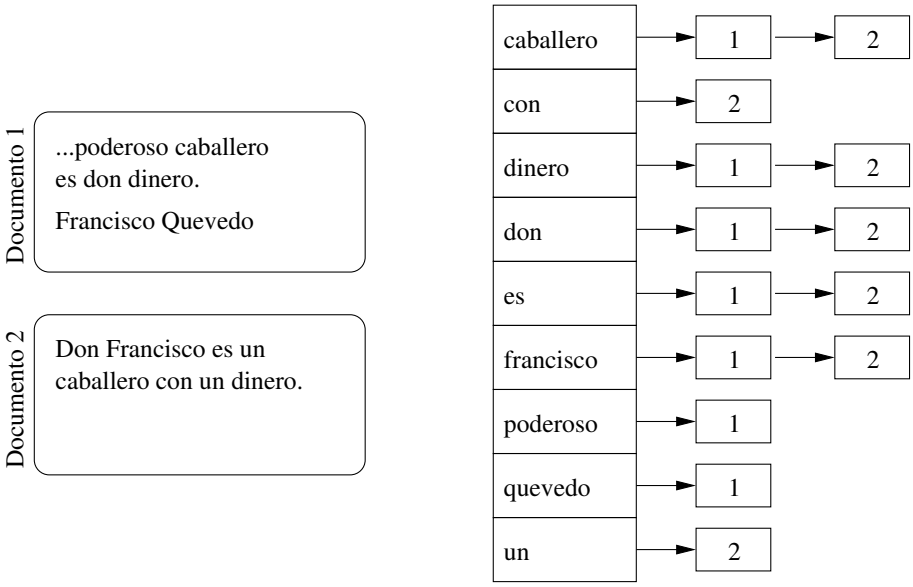


Figura 4.2: Ejemplo de índice invertido con direccionamiento a documento. En cada nodo de la lista de ocurrencias se almacena el documento en el que aparece la palabra.

El tercer tipo se denomina índice invertido con direccionamiento a bloque (*block addressing index*) va un poco más allá. Divide el texto en bloques de tamaño fijo, los cuales pueden agrupar a muchos documentos, ser parte de un documento o puede estar superpuesto sobre los límites de documentos. El índice sólo almacena los bloques en los que aparece cada palabra. Dado que normalmente hay muchos menos bloques que documentos el espacio que ocupará el índice será muy pequeño y se puede elegir de acuerdo con las necesidades del usuario. Por otro lado, alguna consulta deberá de resolverse utilizando una búsqueda secuencial sobre el texto porque no se conoce en qué documentos del bloque aparece la palabra. El índice se utiliza como un dispositivo para filtrar los bloques de la colección que no contienen las palabras consultadas. Este último esquema de indexación fue propuesto inicialmente en Glimpse [MW94], una de las características de Glimpse es que mantiene un índice transparente de todos los ficheros de usuario. El índice ocupa poco espacio y se mantiene actualizado mediante reconstrucciones periódicas, además permite encontrar en cualquier momento los ficheros de usuario que contengan un determinado patrón de búsqueda. Glimpse también se utiliza para indexar sitos web, proporcionando una búsqueda rápida en sus páginas web mediante una técnica de indexación con una sobrecarga baja.

4.2.2. Índices para documentos estructurados

Cuando se manejan documentos con estructura puede ser interesante localizar palabras en elementos de estructura. Una de las primeras propuestas la realizaron Yong Kyu Lee et al. [LYYB96] que propuesieron un tipo de índice basado en un árbol B^+ que permite

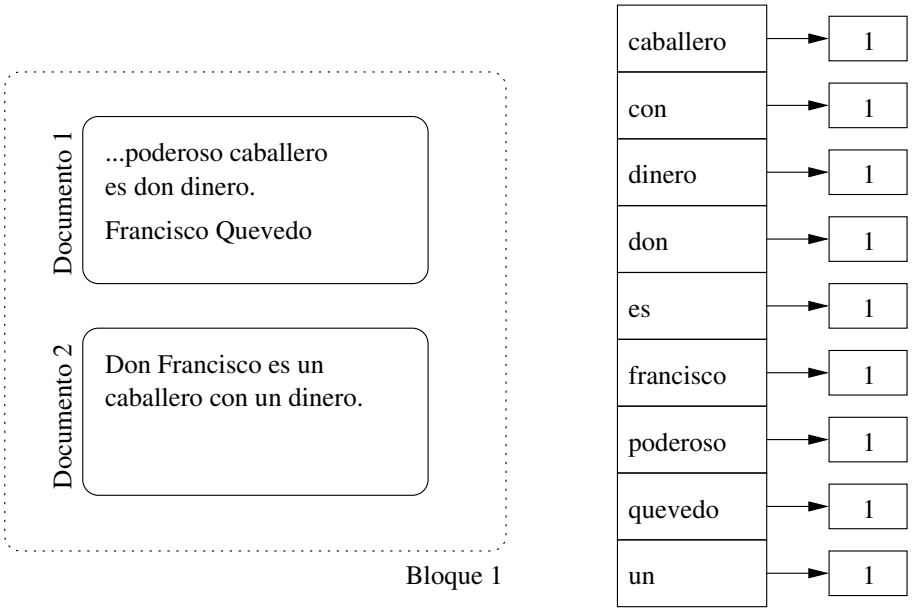


Figura 4.3: Ejemplo de índice invertido con direccionamiento a bloque. Se supone que existe un bloque que engloba a los dos documentos. En cada nodo de la lista de ocurrencias se almacena el bloque en el que aparece la palabra.

localizar palabras dentro de documentos estructurados. Un árbol es la representación de la estructura jerárquica de un documento y cada nodo del árbol representa a un determinado elemento de estructura. A cada nodo del árbol se le representa en el índice mediante un entero único, así pues será necesario disponer de algún mecanismo suplementario que permita asociar el identificador con el tipo de elemento de estructura al que representa.

Por otro lado, en [Kot02] se propone un índice eficiente que extiende el fichero invertido para combinarlo con un índice de caminos, que referencia los términos del fichero invertido con el lugar en dónde se encuentran dentro de la estructura del documento. Se propone utilizar un índice lexicográfico que se usará para resolver consultas que se efectúen sobre la estructura y el contenido de los documentos XML pudiendose obtener rankings como respesta a las consultas sobre la estructura de los documentos.

Previamente a la construcción del índice se eliminan las palabras vacías y las etiquetas irrelevantes, a continuación se realiza un proceso de lematización y por último se realiza una estimación de la distribución de todos los términos que se utilizará para obtener el ranking. El resultado de este proceso genera un árbol denominado *XML summary* que es el punto de partida para crear el índice invertido y el índice de caminos.

También se propone una forma de resolver las consultas con estructura, para ello se devuelve la intersección del conjunto de los documentos que contienen la estructura (o camino dentro del árbol de estructura) de la consulta con el conjunto de documentos que continen los terminos de la consulta. Si la intersección fuera vacía se podría trabajar con la unión de ambos conjuntos. Para obtener el ranking de documentos se debe realizar el pesado de los mismos, para ello se formula una variación del esquema td–idf y se tiene en cuenta la importancia de la estructura (ponderándolo mediante un coeficiente dado por un experto) en la que se encuentra cada término. De esta manera, en la creación del ranking no solo se tiene en cuenta la distribución de los términos en los documentos de la colección sino también la posición del término en la estructura.

Otra aproximación enfatiza la construcción del índice aplicando técnicas específicas de recuperación de información en documentos estructurados [CWC03], de manera que no sólo se trate de encontrar los documentos relevantes (que puede hacerse con cualquier modelo de recuperación clásico) sino también la parte de ellos que pueda resultar más interesante al usuario. Para ello se construye un índice que tiene la misma estructura que el documento indexado mediante un mecanismo de propagación y poda que en primer lugar obtiene e indexa todos los términos de los nodos hoja que contienen texto. A continuación indexan los nodos que contienen hijos ya indexados, para ello se buscan todos los términos comunes en los nodos hijos, se pasan al nodo padre y se eliminan del índice de los hijos. Este proceso se repite hasta que se complete la estructura del documento. A medida que se va construyendo el índice se van asignando los pesos correspondientes a cada término empleando una variación del esquema td–idf, el cual permite obtener los pesos combinando la frecuencia con la que el término aparece en el documento con la frecuencia con la que aparece en los nodos.

Una vez construido el índice se pueden empezar a resolver consultas. Para resolver una consulta se obtiene una lista de documentos empleando un método de recuperación de información tradicional, a continuación, y una vez que el usuario ha escogido un documento para examinar, se analizan las relevancias de todos los nodos candidatos que responden a la consulta y se obtiene un ranking de nodos del documento. Por lo tanto, esta técnica sólo evalúa los documentos que se han seleccionado previamente por el usuario de entre los ofrecidos por un sistema de recuperación de información tradicional. Y sólo en esos documentos se ponderan los elementos estructurales que los forman, seleccionando aquellas que son más relevantes. No obstante, es probable que documentos no seleccionados contengan partes que puedan tener mayor interés para el usuario, mayor incluso que las evaluadas, y sin embargo no se tienen en cuenta con esta propuesta.

4.3. Compresión del índice

Parece difícil que las técnicas de compresión de texto se puedan combinar con los ficheros invertidos debido a la necesidad de acceder a posiciones aleatorias del texto y a realizar búsquedas secuenciales sobre partes del texto. Esas necesidades hacen que tradicionalmente los índices invertidos se apliquen sobre texto sin comprimir. Las técnicas de compresión de texto más recientes [MT97, NMN+00] no sólo permiten reducir el texto en un 25-30 % de su tamaño original sino que también permiten buscar sobre el texto comprimido sin necesidad de descomprimirlo [MNZB00] de una forma mucho más rápida que si se realiza la misma búsqueda sobre texto descomprimido. Todas estas técnicas de compresión se basan en una codificación de Huffman en las que cada tabla de símbolos es el vocabulario del texto.

La compresión es una técnica ortogonal que se utiliza para reducir los requisitos de espacio en los índices invertidos. Se pueden comprimir los documentos y el índice. Existen diferentes técnicas de codificación de números enteros positivos que se pueden aplicar para comprimir las listas de ocurrencias de los índices invertidos. El resultado de esa codificación es una secuencia de bits, el tamaño de esa secuencia no es constante. Las técnicas más conocidas se comentarán a continuación.

4.3.1. Codificación por diferencias

La codificación por diferencias se utiliza como un preprocesamiento que sirva de ayuda para comprimir listas de enteros ordenadas. La idea principal para reducir el tamaño de los índices invertidos es que las listas de ocurrencias de cada palabra estén ordenadas de forma

creciente, y de esta manera se pueda almacenar la diferencia (o hueco) entre un elemento y el elemento de la posición anterior. El primer elemento de la lista contendrá su valor original.

La decodificación se realiza de forma secuencial, se utiliza el primer elemento para calcular el valor del segundo, el segundo para calcular el valor del tercero y así sucesivamente. Realizando la codificación de la lista de esta manera se obtiene una lista que contiene números menores que los originales. Estos números pequeños se pueden representar normalmente con menos bits si se aplica un método de codificación adecuado. Como en las listas grandes esas diferencias son más pequeñas, es de suponer que las listas grandes tengan una mejor compresión y la reducción de espacio puede rondar en torno al 10 %.

4.3.2. Codificación unaria

Es la técnica más sencilla utilizada para codificar enteros positivos. Un entero α cualquiera se codifica mediante $\alpha - 1$ bits con valor '1' seguidos de otro bit con valor '0'. Por ejemplo, el número 3 se codifica como 110. La codificación unaria de los diez primeros enteros se muestra en la tabla 4.1. Esta técnica sólo sería aceptable para números muy pequeños (por ejemplo, menores o iguales a 16), pero se comenta porque se emplea en otras técnicas de codificación de enteros.

4.3.3. Códigos de Elias

Elias propuso en [Eli75] diferentes técnicas para representar números enteros utilizando un número variable de bits. Una de las propuestas se conoce como Elias–γ, la cual dice que para codificar un entero α cualquiera se deberá de obtener el código unario del número $1+\lfloor \log_2 \alpha \rfloor$ y concatenar los $\lfloor \log_2 \alpha \rfloor$ bits que representan al número $\alpha - 2^{\lfloor \log_2 \alpha \rfloor}$ en binario. Por ejemplo, el número $\alpha = 5$ se codifica como $1 + \log_2 5 = 3$ expresado en código unario, concatenandolo con el número $5 - 2^{\lfloor \log_2 5 \rfloor} = 1$ en binario y usando $\lfloor \log_2 5 \rfloor = 2$ bits; obteniendo como resultado el código 11001. Se pueden encontrar otros ejemplos en la tabla 4.1.

Otro método de codificación propuesto en [Eli75] se conoce como Elias–δ. En este método también se concatenan dos partes para formar el código final. Para codificar un entero α cualquiera se debe de obtener el código Elias–γ del número $1 + \lfloor \log_2 \alpha \rfloor$ y a esta secuencia de bits se le concatenan los $\lfloor \log_2 \alpha \rfloor$ bits que representan al número $\alpha - 2^{\lfloor \log_2 \alpha \rfloor}$ expresado en binario. Por ejemplo, para codificar el número $\alpha = 5$ se debe de obtener el código de Elias–γ del número $1 + \lfloor \log_2 5 \rfloor = 3$, que es la secuencia 101, concatenandolo con el número $5 - 2^{\lfloor \log_2 5 \rfloor} = 1$ en binario y usando $\lfloor \log_2 5 \rfloor = 2$ bits; obteniendo como resultado el código 10101. Se pueden encontrar otros ejemplos en la tabla 4.1.

4.3.4. Códigos de Golomb y Rice

Los códigos de Golomb [Gol66] y Rice [Ric79] son códigos de prefijos muy rápidos de implementar, aunque no logran la eficiencia que, dentro de los códigos de prefijos, alcanzan los códigos Huffman.

Cada código de Golomb se distingue con un parámetro, de forma que su adaptación se consigue mediante la estimación del valor de este parámetro.

Para construir el código de Golomb de un alfabeto, sus símbolos se ordenan por orden de probabilidad decreciente, y se les asigna un número entero de acuerdo con este orden, comenzando por el '0' para el símbolo más probable.

Entero	Unario	Elias–γ	Elias–δ
n=1	0	0	0
n=2	10	100	1000
n=3	110	101	1001
n=4	1110	11000	10100
n=5	11110	11001	10101
n=6	111110	11010	10110
n=7	1111110	11011	10111
n=8	11111110	1110000	11000000
n=9	111111110	1110001	11000001
n=10	1111111110	1110011	11000010

Tabla 4.1: Ejemplos de codificación unaria y codificaciones de Elias para los primeros números enteros.

Golomb Rice	m=1 k=0	m=2 k=1	m=3	m=4 k=2	m=5	m=6	m=7	m=8 k=3
n=0	0	00	00	000	000	000	000	0000
n=1	10	01	010	001	001	001	0010	0001
n=2	110	100	011	010	010	0100	0011	0010
n=3	1110	101	100	011	0110	0101	0100	0011
n=4	11110	1100	1010	1000	0111	0110	0101	0100
n=5	$1^5$0	1101	1011	1001	1000	0111	0110	0101
n=6	$1^6$0	11100	1100	1010	1001	1000	0111	0110
n=7	$1^7$0	11101	11010	1011	1010	1001	1000	0111
n=8	$1^8$0	111100	11011	11000	10110	10100	10010	10000
n=9	$1^9$0	111101	11100	11001	10111	10101	10011	10001

Tabla 4.2: Ejemplos de códigos Golomb y Rice para distintos valores de los parámetros. Los superíndices de la forma 1^n indican cadenas de n unos seguidos.

Para codificar el entero usando el código de Golomb de parámetro m, se comienza determinando la parte entera del cociente n/m y se saca este valor utilizando un código unario. A continuación se calcula $n \mod m$ y se saca este valor utilizando un código binario ajustado, de forma que algunas veces se utiliza exactamente la parte entera de $\log_2 m$ bits y otras veces se utiliza un bit más que este valor.

El código de Rice es equivalente al código de Golomb, pero limitando los valores del parámetro m a potencias enteras de 2, es decir, $m = 2k$. Esto facilita la implementación hardware del algoritmo, ya que la parte entera del cociente $n/2k$ se puede determinar con desplazamientos y el valor de n módulo $2k$ se obtiene a partir de los k bits menos significativos de n.

La tabla 4.2 muestra ejemplos de códigos Golomb y Rice para distintos parámetros. Por razones de espacio, las cadenas de cinco o más unos seguidos se han abreviado por 1^j, donde j representa el número de unos de la cadena.

4.4. Otras estructuras de indexación

A continuación un breve recorrido por otras alternativas a los ficheros invertidos, quizá no tan populares pero más específicas para determinados tipos de consulta. Los ficheros

invertidos parten de la premisa de que el texto es una secuencia de palabras, lo cual encarece ciertas consultas, como por ejemplo la búsqueda por frases.

Árboles y arrays de sufijos: permiten resolver consultas más complejas que los ficheros invertidos, aunque son menos eficientes que éstos para consultas simples. Sus principales inconvenientes son que el proceso de construcción es muy costoso, que el texto debe ser accesible en tiempo de consulta y que no proporciona los resultados en el mismo orden en el que aparecen en el texto. Esta estructura sólo se puede utilizar para indexar palabras, por lo que es adecuada para una amplia gama de aplicaciones, como las bases de datos genéricas. Este tipo de índice considera el texto como una gran cadena de caracteres. Cada posición del texto se considera que es un sufijo del texto. No es dificil ver que dos sufijos que comienzan en posiciones diferentes son lexicográficamente diferentes[3] y, por lo tanto, se puede identificar a cada sufijo de forma unívoca mediante su posición. Los *puntos de indexación* señalan el inicio de las posiciones del texto que se pueden recuperar y son los elementos que se indexan.

- Un árbol de sufijos es una estructura de datos construida sobre el conjunto de todos los sufijos de un texto; los apuntadores a los sufijos se almacenan en los nodos hojas. En [Ukk95] se describe un método de construción lineal de árboles de sufijos y en [KU94] una forma de comprimirlos.
- Los arrays de sufijos son una implementación efienciente que reduce el coste de almacenamiento de los árboles de sufijos. Inicialmente se presentaron en [MM93] junto con un algoritmo de construcción para construirlos usando $O(n \log n)$ comparaciones de caracteres. Con anterioridad se describieron en [Gon87] bajo el nombre de "PAT arrays". Un algoritmo de construcción para arrays de sufijos grandes se puede encontrar en [GBYS92]. La utilización de supra-índices sobre el array de sufijos se pospuso en [BYBZ96], mientras que diferentes técnicas modificadas de búsqueda binaria encaminadas a reducir el tiempo de búsqueda en disco se presentaron en [BNB+95]. Una forma de combinar la compresión con los arrays de sufijos se puede encontrar en [MNZ97].

Ficheros de firma: son estructuras de indexación orientadas a palabra y basados en hashing [FC87]. Presentan una baja sobrecarga de almacenamiento (10 % 20 % sobre el tamaño total del texto) e implican búsqueda secuencial en el índice. Su complejidad es lineal, y aunque la constante de linealidad es baja los ficheros invertidos son mejores para la mayoría de aplicaciones. Utiliza técnicas de hashing para mapear las palabras en máscaras de bits de tamaño B, de manera que cada fragmento de texto de tamaño b tiene asignada una máscara de B bits correspondiente a realizar un OR con la máscara de todos los términos que contiene. La idea fundamental es considerar que, dado un fragmento de texto, si contiene una determinada palabra los bits activos de dicha palabra estarán activos también en la máscara del fragmento. El principal inconveniente que presentan es la posibilidad de que los bits activos de un término estén también activos en un fragmento de texto en el que no aparece (false drop). El punto más delicado de esta técnica es asegurar una probabilidad suficientemente baja de que se produzca este fenómeno. Otro punto crítico es la construcción de las máscaras en un tiempo tan corto como sea posible. Son muy adecuados para consultas de tipo frase o proximidad (similitud), porque todas las palabras de la consulta deben estar presentes en la máscara de cada uno de los elementos recuperados, lo cual además hace que la probabilidad de false drop sea más baja. La construcción

[3] Suponiendo que un caracter es más pequeño que sus precesores.

es bastante sencilla, se divide el texto en bloques y cada bloque tiene una entrada asociada en el fichero de signatures. Además, tiene la ventaja de que no hay que reconstruir todo el índice al añadir nuevo texto a la colección, basta con añadir nuevos registros al fichero. Existen diferentes formas de almacenar los ficheros de firma y diferentes formas de comprimirlos [FC88].

Capítulo 5

Recuperación de información

La recuperación de información es el campo de la informática dedicado a la gestión de elementos textuales de información. A diferencia de los sistemas de bases de datos habituales (relacionales), el texto no se puede almacenar en una tabla, mediante registros y campos, de manera que hay que buscar soluciones alternativas para gestionar este tipo de información.

El principal problema de este tipo de sistemas es que tratan con información y no con datos, por lo que no existe una representación formal ni de los documentos de texto ni de las consultas realizadas. Como consecuencia, estos sistemas utilizan una representación aproximada de ambas entidades que intenta capturar su semántica, y que se denomina *vista lógica*.

Se definen dos fases diferenciadas en el funcionamiento de un sistema de recuperación de información:

Fase de indexación cuyo objetivo es construir las representaciones aproximadas de la información que contiene cada documento de la colección; en términos generales, se basa en las palabras que aparecen en cada documento y en el número de veces que aparecen.

Fase de búsqueda cuyo objetivo es construir una representación aproximada de la consulta formulada por el usuario y evaluar su semejanza con los documentos de la colección; los documentos considerados relevantes serán aquellos cuya representación guarde algún parecido con la de la consulta, siendo mayor el nivel de relevancia cuanto mayor sea este parecido.

La mayoría de sistemas genéricos de recuperación de información proporcionan herramientas potentes y eficientes para acceder a los documentos según su contenido: el usuario especifica la información que quiere recuperar a través de una consulta, a menudo en lenguaje natural, y los documentos que el sistema estima que son relevantes para los requisitos del usuario se le presentan a éste a través de una interfaz. Una buena interfaz permitirá un proceso de recuperación interactivo, estimulando al usuario a revisar los documentos recuperados y reformular la consulta inicial, de forma manual o automática; esta última es la forma más habitual, a través de lo que se denomina *realimentación de relevancia* (relevance feedback). Dada la complejidad del trabajo de recuperación y la imprecisión de la descripción de la consulta hecha por el usuario, el modo interactivo es ampliamente reconocido como la forma más eficiente de mejorar el rendimiento del acceso a la información.

Desde el punto de vista computacional, la recuperación de información consiste en construir índices eficientes, procesar las consultas de los usuarios lo más rápidamente posible y desarrollar algoritmos de clasificación (ranking) que mejoren la calidad de la información encontrada. Los documentos se representan mediante el conjunto de términos relevantes que aparecen en ellos.

Desde el punto de vista del usuario, la recuperación de información consiste en estudiar el comportamiento del usuario, interpretar su necesidad de información y analizar cómo ésta afecta a la organización y el funcionamiento del sistema.

5.1. Modelado de la recuperación de información

Podemos definir término de indexación desde dos puntos de vista:

- Desde un punto de vista restrictivo, un término de indexación es una palabra clave (o grupo de ellas interrelacionadas) tal que tiene significado propio, por sí misma, sin necesidad de ir acompañada de otras para poder definirlo
- Desde un punto de vista más general, un término de indexación puede ser cualquier palabra que aparece en el texto de cualquier documento de una colección

El fundamento de todos los sistemas de recuperación de información basados en términos de indexación es que el contenido semántico de los documentos, o de la necesidad de información expresada por el usuario, puede expresarse a partir de un conjunto de términos de indexación.

Uno de los puntos más conflictivos en cualquiera de estos sistemas es la necesidad de determinar qué documentos son relevantes y cuáles no, siendo necesario un algoritmo que realice la clasificación de los documentos.

El algoritmo actuará de acuerdo a una serie de premisas para evaluar qué es relevante y qué no lo es, de manera que diferentes conjuntos de premisas determinan diferentes modelos recuperación de información.

5.1.1. Evaluación de la efectividad en recuperación de la información

Para compara los distintos modelos de recuperación de información se han descrito distintas métricas, ninguna enteramente satisfactoria debido a que se representa de forma escalar algo que tiene una naturaleza espacial.

Se definen dos medidas muy importantes, la *precisión* que es la proporción de documentos recuperados que son relevantes y la *recuperación* que es la proporción de documentos relevantes que son recuperados. Manipulando el número de documentos que el modelo considera relevantes se puede aumentar una a costa de la otra.

Otros datos de interés son la llamada *precisión interpolada* que es la mayor precisión alcanzada para recuperaciones iguales o mayores que la dada; o la *precisión media* calculada a partir de las precisiones interpoladas para distintas recuperaciones.

5.2. Recuperación de información clásica

Los modelos clásicos consideran que un documento está descrito por un conjunto de términos representativos denominados "términos de indexación", de manera que un término de indexación es una palabra cuyo significado representa parte de la semántica global del documento al que pertenece.

Es evidente que, dado un documento, habrá términos más representativos que otros; esta diferencia se representa mediante la asignación de pesos (numéricos) a cada término de indexación de un documento.

Definición 5.1 *Sea t el número total de términos y k_i un término genérico y, por lo tanto, $K = \{k_1, \ldots, k_t\}$ es el conjunto de todos los términos. A cada término k_i de un documento d_j se le asocia un peso $w_{i,j} > 0$. Si un término no aparece en un documento entonces su correspondiente $w_{i,j} = 0$. Un documento d_j está asociado con un vector de términos $\vec{d_j} = (w_{1,j}, w_{2,j}, \ldots, w_{t,j})$. La función g_i devuelve el peso asociado con el término k_i de cualquier vector t-dimensional, es decir, $g_i(\vec{d_j} = w_{i,j})$.*

Normalmente, los pesos de los términos se consideran mutuamente independientes (simplificación a partir de la inexistencia de correlación entre la aparición de cada término con el resto), de manera que el conocimiento del peso de un término en un documento no dice nada acerca del valor asociado a otro término.

A continuación se realizará una breve descripción de los modelos clásicos utilizados en recuperación de información (booleano, vectorial y probabilístico).

5.2.1. Modelo booleano

El modelo booleano está basado en la teoría de conjuntos y el álgebra booleana [vR79, BYRN99]. Se considera que los términos de indexación están presentes o ausentes en el documento; los pesos son binarios, es decir, $w_{i,j} \in \{0, 1\}$. Las consultas son expresiones booleanas con una semántica bien definida: términos de indexación unidos por operadores booleanos *and*, *or* y *not*. El modelo booleano predice si un documento es relevante o no sin término medio, por lo que no considera que un documento responda parcialmente a la consulta.

Definición 5.2 *En el modelo booleano los pesos asociados a cada término son binarios, $w_{i,j} \in \{0, 1\}$. Una consulta q es una expresión booleana convencional. Sea $\vec{q}_{dnf}$ la forma normal disyuntiva de la consulta q. Sea $\vec{q}_{cc}$ cualquiera de la componentes conjuntivas de $\vec{q}_{dnf}$. La similitud del documento d_j en la consulta q se define como*

$$sim(d_j, q) = \begin{cases} 1 & si \quad \exists \vec{q}_{cc} \bullet (\vec{q}_{cc} \in \vec{q}_{dnf}) \wedge \left(\forall k_i, g_i(\vec{d_j}) = g_i(\vec{q}_{cc})\right) \\ 0 & en\ caso\ contrario \end{cases} \tag{5.1}$$

El modelo booleano predice que el documento d_j es relevante a la consulta q si $sim(d_j, q) = 1$. En caso contrario se dice que no es relevante.

La principal ventaja de modelo booleano es su simplicidad. Sus inconvenientes residen en que realiza una búsqueda exacta, es decir, recupera muchos o muy pocos documentos; y, por otro lado, no es sencillo trasladar las necesidades de información a expresiones booleanas aunque éstas tienen una semantica precisa. Con todo esto se puede decir que el modelo booleano responde más a la recuperación de datos que a la recuperación de información.

5.2.2. Modelo vectorial

El modelo vectorial supone un avance respecto al modelo booleano pues se reconoce que el empleo de pesos binarios limita de manera considerable la obtención de respuestas intermedias. Por eso propone un marco de trabajo en el que los pesos son valores no binarios para poder obtener respuestas intermedias. Los pesos de los términos se utilizan para obtener el *grado de similitud* entre los documentos y la respuesta [SL68, SY73].

Definición 5.3 *En el modelo vectorial los pesos $w_{i,j}$ asociados al par (k_i, d_j) son valores positivos y no binarios. A los términos de las consultas también se les asocia un peso. Sea $w_{i,q} \geq 0$ el peso asociado al par (k_i, q). Entonces, el vector de consulta $\vec{q}$ se define como $\vec{q} = (w_{1,q}, w_{2,q}, \ldots, w_{t,q})$ en donde t es el número total de términos. Y consecuentemente, el vector para el documento d_j se representa mediante $\vec{d_j} = (w_{1,j}, w_{2,j}, \ldots, w_{t,j})$*

Por lo tanto, los documentos y la consulta se representan utilizando vectores t-dimensionales. El modelo vectorial propone utilizar como medida de similitud la correlación entre el vector de un documento, $\vec{d_j}$, y el vector de la consulta, $\vec{q}$. Esta correlación se puede cuantificar, por ejemplo, por el *coseno del ángulo* que forman los vectores:

$$\begin{aligned} sim(d_j, q) &= \frac{\vec{d_j} \cdot \vec{q}}{|\vec{d_j}| \times |\vec{q}|} \\ &= \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{i=1}^{t} w_{i,q}^2}} \end{aligned}$$

Ya que $w_{i,j} \geq 0$ y $w_{i,q} \geq 0$ el valor de $sim(q_d, j)$ se encuentra comprendido en el rango $[0, +1]$. Además de predecir si un documento es relevante o no, el modelo vectorial ordena los documentos de acuerdo al grado de similitud y permite recuperar documentos que respondan parcialmente a la consulta. Los pesos de los términos se pueden calcular de varias maneras [SM83]. A continuación se describirá la más conocida.

Definición 5.4 *Sea N el número total de documentos y n_i el número de documentos en los que aparece el término k_i. Sea $freq_{i,j}$ el número de veces que el término k_i aparece en el documento d_j (es decir, la frecuencia del término k_i en el documento d_j o "factor tf"). Entonces, la frecuencia normalizada $f_{i,j}$ del término k_i en el documento d_j se calcula mediante la expresión*

$$f_{i,j} = \frac{freq_{i,j}}{\text{máx}_l \ freq_{l,j}}$$

en dónde el máximo se obtiene de entre todos los términos que aparecen en el documento d_j. Si el término k_i no aparece en el documento d_j entonces $f_{i,j} = 0$. Sea idf_i la frecuencia inversa del documento del término k_i calculada como

$$idf_i = \log \frac{N}{n_i}$$

Los mejores esquemas de pesado de términos utilizan pesos obtenidos mediante la expresión

$$w_{i,j} = f_{i,j} \times \log \frac{N}{n_i}$$

o variaciones de dicha expresión.

Por este motivo, las estrategias de pesado de términos se denominan *esquemas tf–idf* [SJ72, SJ73]. Para asignar pesos a los términos de la consulta Salton y Buckley [SB88] propusieron la siguiente expresión

$$w_{i,q} = \left(\frac{1}{2} + \frac{\frac{1}{2} freq_{i,q}}{\text{máx}_l \ freq_{l,q}} \right) \times \log \frac{N}{n_i}$$

en donde $freq_{i,q}$ es la frecuencia del término k_i en el texto de la información solicitada q.

Una de las principales ventajas del modelo vectorial es que su esquema de pesado de términos mejora el rendimiento de la recuperación y, además permite recuperar documentos que responden a la consulta de forma aproximada. Otra ventaja es que la ecuación de correlación entre los vectores permite ordenar los documentos de acuerdo al grado de similitud. Teóricamente, el modelo vectorial tiene la desventaja que supone que no existen dependencias entre los términos.

5.2.3. Modelo probabilístico

El modelo probabilístico trata de estimar la probabilidad de que el usuario vaya a considerar interesante (relevante) el documento d_j como respuesta a una consulta q [RSJ76, vR79]. Para ello, el modelo supone que la probabilidad de relevancia solo depende de la consulta y de la representación del documento. Se supone que existe un conjunto de documentos relevantes para la consulta q. Típicamente, a este *conjunto ideal* se le conoce como R y debe maximizar la probabilidad de relevancia.

Definición 5.5 *En el modelo probabilístico los pesos asociados a los documentos y a las consultas son binarios,* $w_{i,j} \in \{0,1\} \wedge w_{i,q} \in \{0,1\}$. *Una consulta* q *es un subconjunto de términos. Sea* R *el conjunto de documentos relevantes conocidos (o propuestos), y* $\bar{R}$ *el conjunto complemento de* R *(es decir, el conjunto de documentos no relevantes). Sea* $P(R|\vec{d_j})$ *la probabilidad de que el documento* d_j *sea relevante a la consulta* q *y* $P(\bar{R}|\vec{d_j})$ *la probabilidad de que* d_j *no sea relevante a la consulta* q. *La similitud* $sim(d_j, q)$ *del documento* d_j *a la consulta* q *se define mediante la expresión*

$$sim(d_j, q) = \frac{P(R|\vec{d_j})}{P(\bar{R}|\vec{d_j})} \tag{5.2}$$

A efectos prácticos, la función de similitud tal y como está expresada en la ecuación anterior, no es operativa. Después de un conjunto de suposiciones y operaciones se obtiene la siguiente expresión

$$sim(d_j, q) \approx \sum_{i=1}^{t} \left(w_{i,q} \times w_{i,j} \times \left(\log \left(\frac{P(k_i|R)}{1 - P(k_i|R)} \right) + \log \left(\frac{1 - P(k_i|\bar{R})}{P(k_i|\bar{R})} \right) \right) \right) \tag{5.3}$$

en donde $P(k_i|R)$ es la probabilidad de que el término k_i esté presente en un documento elegido al azar en el conjunto R, $P(\bar{k}_i|R)$ es la probabilidad de que el término k_i no esté presente en un documento elegido al azar en el conjunto R, por último, $P(k_i|\bar{R})$ y $P(\bar{k}_i|\bar{R})$ tienen significados análogos a los anteriores y complementarios entre sí.

Una vez que se ha obtenido una expresión operativa de la similitud, el único problema que queda por resolver es que no se conoce el conjunto R inicial, necesario para calcular las probabilidades $P(k_i|R)$ y $P(k_i|\bar{R})$. Una posible solución es suponer que $P(k_i|R)$ es constante para todos los términos e igual a $\frac{1}{2}$, y además también se supone que la distribución de términos entre los documentos no relevantes se puede aproximar por la distribución de términos entre todos los documentos de la colección. Con estas dos suposiciones tenemos que $P(k_i|R) = \frac{1}{2}$ y $P(k_i|\bar{R}) = \frac{n_i}{N}$, en donde n_i es el número de documentos que contienen el término k_i y N es el número total de documentos en la colección.

Esas probabilidades se pueden mejorar una vez conseguido un ranking inicial. Sea V el número inicial de documentos recuperados y V_i el número de documentos recuperados que contienen el término k_i, entonces

$$\begin{aligned} P(k_i|R) &= \frac{V_i + \frac{n_i}{N}}{V+1} \\ P(k_i|\bar{R}) &= \frac{n_i - V_i + \frac{n_i}{N}}{N - V + 1} \end{aligned}$$

La principal ventajas del modelo probabilístico es que ordena los documentos en orden decreciente de la probabilidad de ser relevantes. Pero necesita una separación inicial entre documentos relevantes y no relevantes cosa que no siempre es posible, por otro lado, al manejar pesos binarios no tiene en cuenta la frecuencia de aparición de los términos en los documentos, y tampoco tiene en cuenta la dependencia entre los términos.

5.3. Ampliaciones de los modelos clásicos

En esta sección se comentarán brevemente diferentes ampliaciones de los modelos clásicos que se han enunciado con el paso del tiempo con el propósito de mejorar o añadir prestaciones a dichos modelos clásicos. Por lo general, la mayoría de las propuestas son teóricas y existen pocos sistemas de recuperación de información reales que las implementen. Para mejorar el modelo booleano se utilizan modelos alternativos basados en la teoría de conjuntos y su principal objetivo es recuperar los resultados intermedios que no proporciona el dicho modelo. Por otro lado, las extensiones del modelo vectorial y probabilístico las llevan a cabo los modelos algebráicos alternativos y los modelos probabilísticos alternativos respectivamente y tratan de reflejar las dependencias entre los términos.

5.3.1. Modelos alternativos basados en la teoría de conjuntos

Habitualmente los documentos y las consultas se representan mediante conjuntos de palabras clave. Esta representación los describe de forma parcial respecto a los contenidos semánticos reales de los mismos y, en consecuencia, la adecuación de un documento a una consulta es vago. El *modelo de conjuntos difuso* [OMK91] permite modelar este hecho pues considera que los términos de una consulta definen un conjunto difuso y que cada documento tiene un nivel de pertenencia a dicho conjunto, que generalmente se expresa mediante un valor real perteneciente al intervalo $[0, 1)$. Este concepto se utiliza en la recuperación de información difusa [Rad76, Rad77, Rad79] que, en líneas generales, amplía el conjunto de términos de la consulta con términos relacionados que se obtienen del tesauro. Así se recuperan documentos adicionales que se pueden considerar relevantes y que normalmente no se recuperarían.

La recuperación booleana es simple y elegante, pero no se genera un ranking con las respuestas porque no existe ninguna forma de pesado de términos. Esto es debido a que no existe una situación intermedia, ya el término está o no presente en el documento. Por lo tanto, el criterio de decisión binario no es útil en la mayoría de las situaciones prácticas. El *modelo booleano extendido* [SFW83, Lee94] intenta proporcionar un ranking de respuestas mediante una normalización de los términos. Si sólo se tienen en consideración dos términos, x e y, se pueden representar tanto las consultas como los documentos en una gráfica bidimensional. Un documento j se posiciona en ese espacio dependiendo de los pesos normalizados $w_{x,j}, w_{y,j} \in [0, 1]$ de documentos respecto a ambos términos. Dependiendo de que la consulta sea conjuntiva o disyuntiva se aplican funciones de similitud diferentes [LKKL93], las cuales interpretan las operaciones booleanas en términos de distancias algebráicas y devuelven un valor numérico, permitiendo generar el ranking.

5.3.2. Modelos algebráicos alternativos

Los modelos clásicos presuponen que los términos del texto son independientes entre sí y habitualmente en el modelo vectorial esta independencia entre los términos se interpreta de una manera más restritiva en el sentido que cada pareja de vectores es ortogonal, es decir, $\vec{k}_i$ y $\vec{k}_j$ cumplen que $\vec{k}_i \cdot \vec{k}_j = 0 \ \forall i, j$.

El *modelo vectorial generalizado* [WZW85, WZR87] supone que los vectores que representar a los términos son linealmente independientes, en cambio cada pareja de vectores no es ortogonal. Por lo tanto, en el modelo vectorial generalizado dos vectores de términos pueden no ser ortogonales y, por lo tanto, no se pueden considerar estos vectores como los vectores ortogonales que componen la base del espacio vectorial. La idea principal es introducir parejas de vectores ortogonales $\vec{m}_i$ asociados con el conjunto de los términos

originales independientes o "minterms" y adoptar este conjunto de vectores como base del subespacio vectorial de interés. Entonces ahora los términos se encuentrán relacionados mediante los vectores $\vec{m}_i$.

La idea de utilizar las dependencias entre los términos mejora el rendimiento de la recuperación sigue siendo un aspecto controvertido y no está lo suficientemente claro que el marco del espacio vectorial generalizado proporcione una ventaja clara en situaciones prácticas, siendo además más complejo y computacionalmente más caro que el modelo vectorial clásico.

El *modelo de indexación de semántica latente* [FDD+88, BCB92] se basa en la idea de representar los documentos y las consultas mediante un vector que pertenece a un espacio de dimensión menor que está asociado con conceptos en lugar de términos. Esto se lleva a cabo mapeando los vectores de los términos en el espacio de dimensión menor. El motivo para hacer esto es que la recuperación en el espacio reducido puede ser superior que la recuperación en el espacio de los términos original, pues las ideas que expresan un texto están más relacionadas con los conceptos que describen que con los términos que se utilizan en la descripción.

Este modelo introduce una interesante conceptualización del problema de la recuperación de información basado en la teoría de la descomposición de valores singulares que puede permitir una recuperación superior en la mayoría de las situaciones prácticas.

Por otro lado se la logrado simular el modelo vectorial y las iteraciones del usuario mediante un *modelo de red neuronal* [Kwo89, Kwo90, WH91, Kwo95]. Éste está formado por tres capas: una para representar los términos de la consulta, otra para representar los términos que aparecen en los documentos y la última para representar los propios documentos. Cada término de la consulta, del documento y cada documento respectivamente se representan mediante un nodo en la red o neurona. El proceso de inferencia comienza cuando los nodos de los términos de la consulta envían señales a los nodos de los términos de documento correspondientes.

A continuación los nodos de los términos de documento que han recibido una señal deben generar señales que se envían a los nodos de documento. La recepción de estas señales completa la primera etapa en la que una señal se propaga desde los términos de la consulta hasta los documentos en los que aparecen.

Los nodos de documento generan nuevas señales que se devuelven a los nodos de los términos de documento, los cuales vuelven a repetir el proceso. En cada iteración las señales se van debilitando hasta que desaparecen, por lo que el proceso terminará en algún momento y, además, Se puede activar un documento que no contenga ningún término de la consulta. El ranking se puede construir a partir de los niveles de activación de los nodos de documento, que después de la primera iteración coincidirá con el obtenido por el modelo vectorial. Para mejorar el rendimiento de la recuperación el proceso de activación continua modificando el ranking en cada iteración de forma análoga al ciclo de retroalimentación por relevancia que efectúa el usuario. No obstante, no existe una conclusión evidente que el modelo de red neuronal obtenga un mejor rendimiento de la recuperación.

5.3.3. Modelos probabilísticos alternativos

Las redes bayesianas están basadas en métodos probabilísticos y se utilizan exitosamente en entornos caracterizados por la incertidumbre. En recuperación de información se han utilizado como extensión de los modelos probabilísticos pues ofrecen ventajas importantes con las que abordar la incertidumbre intrínseca de dicha disciplina [CLvRC91]. Esto se

debe a que la consulta de usuario es una vaga expresión de su necesidad de información, y a que la forma de representar los documentos también introduce incertidumbre.

Una red bayesiana es un grafo dirigido acíclico en donde los nodos representan las variables aleatorias del problema que se desea resolver. En esta clase de grafos, el conocimiento se representa de forma cualitativa mostrando las dependencias que existen entre las variables y de forma cuantitativa expresando la fuerza con que se suponen dichas relaciones, medida mediante una distribución de probabilidad condicionada [Pea88].

A continuación se describirán brevemente los modelos de recuperación de información basados en redes bayesianas más relevantes. El primero de ellos es el denominado *modelo de red de inferencia* [TC90, TC91] y se basa en una red en la que se distinguen dos subredes: una red fija de documentos para cada colección formada por dos tipos de nodos que representan los términos de los documentos y los documentos respectivamente, de manera que de un nodo de documento salen arcos hacia los nodos de los términos por los que han sido indexados. Por otro lado se distingue la red de consulta que se crea cuando el usuario formula una consulta al sistema y contiene nodos de consulta y nodos de términos, de manera que de un nodos de término salen arcos hacia los nodos de consulta correspondientes. Ambas subredes se conectan mediante los nodos de términos que existen en ambas. Una vez que se han estimado las probabilidades, la inferencia se hace instanciando cada documento sucesivamente y calculando la probabilidad de que la consulta quede satisfecha dado el documento que ha sido observado, es decir, $p(q|d)$. Una vez que todas las propagaciones han finalizado, se genera la ordenación de documentos correspondiente.

Una alternativa al modelo de red de inferencia se presenta en [GIS96] cuya diferencia con éste es que se cambia la orientación de los arcos. Formalmente, para una consulta q los documentos se ordenan según la probabilidad $P(d|q)$. Para ello se instancian los nodos de la consulta, propagando sólo una vez y calculando la probabilidad de que cada documento sea relevante dada la consulta.

Por su parte el *modelo de red de certidumbre* [RNM96] sólo considera dos tipos de nodos: nodos de documento y nodos de término. Estos nodos están conectados por arcos que van desde los nodos de término a los nodos de documento. La consulta se considera como un tipo especial de documento y se propaga una sóla vez para obtener una ordenación según $p(d|q)$.

5.4. Recuperación de información en documentos estructurados

La reciente proliferación de documentos XML para almacenar y organizar información textual ha originado una creciente demanda de recuperación de información efectiva sobre este tipo de documentos y que además utilice tanto la estructura como la información contenida en dicha estructura para devolver documentos o partes de los mismos como respuesta a una consulta.

Cuando se aplican las técnicas de recuperación de información tradicionales para recuperar documentos XML (o partes de los mismos) no se obtienen los resultados deseados pues estas técnicas ignoran la estructura de los documentos dado que solo permiten efectuar "consultas planas" y, por lo tanto, se pierde una gran parte del aspecto semántico de la pregunta. Por lo tanto, para recuperar información estructurada se debe tener en cuenta tanto el contenido como la estructura de los documentos.

La estructura de los documentos se utiliza para facilitar una focalización de las respuestas del sistema a unidades de documento más adecuadas (esta mayor adecuación está

relacionada con la noción de especificidad del componente de documento en la consulta). Además el problema de las estrategias de indexación de información estructurada es crítico para el diseño de tales estrategias de recuperación. Sin embargo, si se basan en el uso explícito de la estructura lógica o en la recuperación de pasajes, los modelos de recuperación que manejan documentos estructurados siguen siendo experimentales y hay una falta de experimentación extensiva en colecciones grandes. En particular sigue siendo difícil compararlos a un nivel cualitativo debido a que sigue habiendo una carencia de metodología experimental y de las correspondientes colecciones de prueba para su realización [Chi01].

Ross Wilkinson [Wil94] fue uno de los primeros en estudiar la posibilidad de devolver una o varias partes de un documento como respuesta a una consulta de recuperación de información. Su trabajo pone de manifiesto que los documentos completos se pueden valorar bien a partir del conocimiento de las partes pero, en cambio, no es conveniente realizar una valoración de las partes a partir de los del conocimiento de los documentos completos. El uso de ambas informaciones mejora la precisión de los primeros puestos del ranking, además el conocimiento de la estructura se revela como un componente interesante para la recuperación.

Los modelos de recuperación de información que permiten combinar la información textual y la información estructural de los documentos se denominan *modelos de recuperación de texto estructurado*. Es importante distinguir estos modelos de los de recuperación de datos, *data retrieval*, pues los modelos de recuperación de texto estructurado permiten recuperar documentos que satisfagan parcialmente la consulta, mientras que en una recuperación de datos sólo se recuperar elementos que satisfacen la consulta en su totalidad.

5.4.1. Taxonomía de los SRI estructurados

La inclusión de la estructura en los procesos de indexación y recuperación afecta en varios aspectos al diseño e implementación de un sistema de recuperación de información:

1. La indexación debe capturar la estructura de manera adecuada para permitir la recuperación tanto por contenido como por estructura.

2. El proceso de recuperación debe considerar tanto la estructura como el contenido a la hora de estimar la relevancia de un documento u otro componente estructural.

3. Surge la necesidad de disponer de interfaces que permitan formular de forma ágil y sencilla este tipo de consultas y que ayuden a visualizar de forma adecuada los elementos recuperados, en lo que podríamos llamar un intento por focalizar la recuperación de la información.

Desde un punto de vista general, se pueden tener nueve posibles tipos de sistemas de recuperación de información en función de la consideración de la estructura y/o el contenido de la misma en los procesos de indexación y recuperación [Veg99]. Se pueden identificar mediante dos conjuntos de dos letras, es decir, dos letras por cada uno de los mencionados procesos. Se usarán las letras C y S para identificar el uso de contenido o estructura. La tabla 5.1 muestra las nueve clases de sistemas.

Esta tabla indica qué tipo de función de recuperación necesitará el sistema, f, y que tipo de información debe proporcionar el usuario al formular la consulta, los argumentos c y/o s de la función f. La descripción de los diferentes tipos de sistemas se pueden encontrar en [Veg99].

Indexación	Consulta C	Consulta C+S	Consulta S
C	$q = f_c(c)$	$q = f_c(c, s)$	
C + S	$q = f_{c+s}(c)$	$q = f_{c+s}(c, s)$	$q = f_{c+s}(s)$
S		$q = f_s(c, s)$	$q = f_s(s)$

Tabla 5.1: Clasificación de los sistemas de recuperación de información respecto al uso de contenido y estructura.

5.4.2. Recuperación de pasajes

La idea de recuperar partes de documentos no es nueva, aunque los motivos y las propuestas eran bastante diferentes en el pasado. La recuperación de partes de documentos, llamadas *pasajes*, es la propuesta más antigua y estudiada en este campo [O'C75, O'C80, MSDWZ93, Cal94]. El principio que fundamenta la recuperación de pasajes es bastante simple: dentro de los documentos de texto, los pasajes son secuencias textuales de palabras consecutivas que presentan cierta homogeneidad en su temática y que, por lo tanto, pueden ser relevantes para el usuario [MS94]. Por "palabras consecutivas" entendemos una secuencia de palabras según el "orden de lectura" del documento de texto (esto es desde la primera página a la última). Una vez aislados, los pasajes se consideran como documentos separados y se indexan y recuperan del modo clásico, basado en los modelos estándar de indexación y recuperación (booleano, vectorial, probabilístico, etc). Según esta propuesta, los documentos se consideran como secuencias lineales (a menudo no se solapan) de pasajes que pueden o no coincidir exactamente con unidades estructurales tales como párrafos, capítulos o secciones. Debido a que a priori se ignora la estructura lógica oculta de los documentos, el problema principal es encontrar una "buena" segmentación de los documentos en una secuencia de pasajes acertada. Empezando por la primera palabra del documento hasta la última, las principales cuestiones que se deben solventar son: ¿en cuántos pasajes dividir el documento? y ¿cuáles serían los límites de los mismos dentro del texto? [Chi01].

Un método clásico para localizar pasajes consiste en analizar la distribución de las palabras dentro de una ventana de tamaño fijo que se va desplazando. Los límites de los pasajes se encuentran cuando se observa un cambio significativo en la distribución del vocabulario local al compararlo con distribución previa.

También se han realizado propuestas basadas en pasajes más cercanas a la noción de estructura lógica, pero este tipo de propuestas no pueden aprovechar al máximo de la estructura lógica de los documentos, pero el mayor inconveniente con el que nos encontramos es que los pasajes se corresponden con unos segmentos secuenciales estáticos pre-procesados (y por lo tanto no jerárquicos) cuyo tamaño no se puede adaptar de forma dinámica a las consultas que hacen referencia a la estructura específica de los documentos. En otras palabras, la recuperación de pasajes es una útil extensión del concepto estándar de documento pero que no es suficiente para responder a consultas que referencian estructura debido a la naturaleza de los pasajes.

5.4.3. Modelo de listas no solapadas

Esta propuesta de Burkowski [Bur92a, Bur92b] se basa en la división de cada documento en regiones de texto disjuntas pero que recubren todo el documento (índices densos), produciendo una lista. El proceso se realiza para los distintos tipos de regiones que aparecen en el documento, produciéndose múltiples listas: lista de capítulos, lista de párrafos,

etc., de manera que no existe solapamiento entre regiones de una misma lista y si se puede producir entre regiones de listas diferentes (ver Figura 5.1).

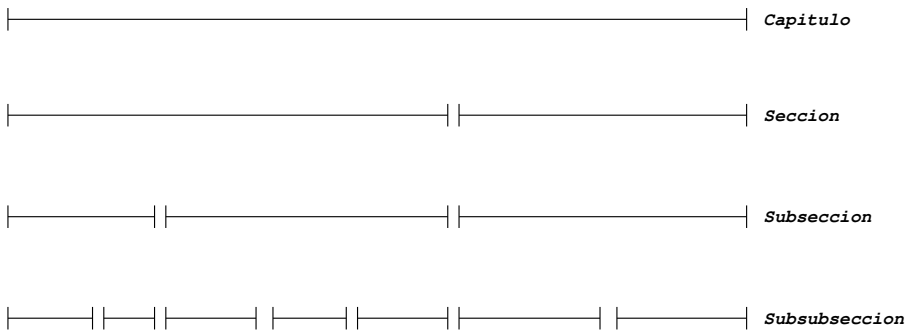


Figura 5.1: Representación de la estructura de un documento de texto mediante cuatro listas de indexación (densas) diferentes.

Para permitir la búsqueda por regiones de texto, además de por términos de indexación, se construye un índice (habitualmente mediante un fichero invertido) en el que hay una entrada por cada tipo de componente estructural; asociada a cada entrada hay una lista de las regiones que representan nodos de ese tipo. Además, cada lista puede ser *mezclada* con el índice correspondiente a las ocurrencias de los diferentes términos.

Debido a que las regiones pertenecientes a una misma lista no se solapan, el tipo de consultas a realizar sobre este tipo de modelos son sencillas, del tipo "seleccionar una región que contenga el término τ", "seleccionar una región A que no contenga a otra región B (A y B pertenecientes a listas distintas)", "seleccionar una región no contenida dentro de ninguna otra".

5.4.4. Modelo de nodos proximales

Propuesto por Gonzalo Navarro y Ricardo Baeza-Yates [NBY95, BYN96, NBY97b] esta propuesta permite definir estructuras de indexación jerárquicas diferentes para un mismo documento (índices no densos), de manera que cada componente de estos índices es un nodo asociado a una región de texto. Dada una consulta, el conjunto respuesta sólo podrá contener nodos pertenecientes a una misma estructura jerárquica. La Figura 5.2 muestra una estructura de indexación jerárquica de cuatro niveles.

Dado un índice jerárquico para la estructura, que proporciona las posiciones en el documento de cada nodo, y un típico fichero invertido de indexación de las ocurrencias de los términos (denso), que indica dónde se producen éstas, las consultas se pueden resolver aplicando primero el índice denso para localizar las ocurrencias de los términos de consulta y después, para cada una de ellas, recorrer el índice de estructura para determinar cuáles responden a los requisitos de estructura planteados en dicha consulta.

Se intenta llegar a un compromiso entre expresividad, búsquedas mediante términos, búsquedas sobre la estructura y combinaciones de ambos tipos, y eficiencia, detectando primero en qué nodos hay ocurrencias y comprobando después si son estructuralmente válidos para la consulta planteada.

Una propuesta de procesamiento más sofisticado, en pos de una mayor eficiencia, sería el siguiente. El primer paso sería localizar, para la primera ocurrencia del índice de términos, el nodo de mayor profundidad que contiene la ocurrencia; después, para el resto de casos, se comprobaría si ese nodo localizado como el más profundo contiene esta nueva ocurrencia,

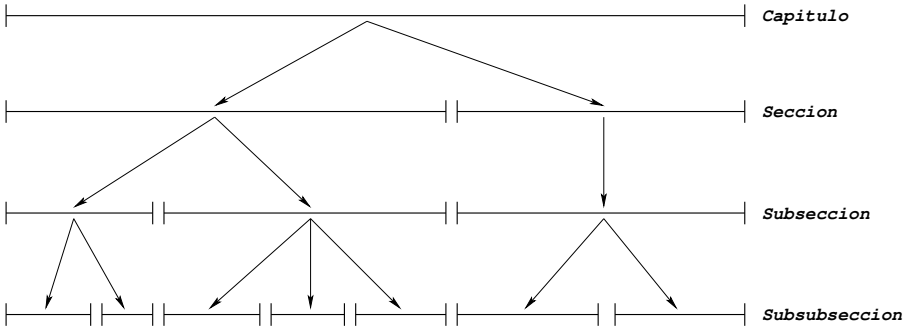


Figura 5.2: Representación de la estructura de un documento de texto mediante el modelo de nodos proximales.

y en caso de ser así se podrá concluir que todos los nodos de nivel superior que le contienen presentan una ocurrencia de ese término (y no sería necesario revisar el resto de la jerarquía) y serán considerados como relevantes los que respondan al perfil de estructura descrito en la consulta.

La idea de revisar los nodos próximos primero y no tener así que revisar el resto de índice jerárquico si no es necesario da nombre a este modelo. Las posibles consultas a formular para este modelo permiten mayor complejidad que en el modelo basado en listas no solapadas.

5.4.5. Recuperación de los "mejores puntos de entrada"

En ciertas situaciones quizá pueda ser más interesante determinar cuáles pueden ser los mejores puntos de entrada al documento en lugar de seleccionar los nodos más relevantes [KLR02]. Los puntos de entrada al documento son los lugares en donde se debe abrir el documento para que el usuario acceda a las partes más relevantes del mismo, de acuerdo a la consulta realizada. Para llevar a cabo este cometido no basta con conocer cuáles son esos puntos sino que también se deben obtener las relaciones entre ellos. Los mejores puntos de entrada (BEP's) se definen basándose en los siguientes criterios:

1. Se recupera un nodo padre en lugar de sus hijos cuando la mayoría de ellos se hayan considerado relevantes.

2. Cuando se obtiene una secuencia de nodos muy relacionados que se han recuperado de acuerdo con el criterio anterior, entonces como respuesta solo se devuelve el primero de ellos.

No obstante, obtener unos buenos resultados, o BEP's, dependerá en gran medida de la relevancia que se obtenga para los nodos.

5.4.6. Generalizaciones del esquema tf–idf

En [GS02b, GS02a] se llega a la conclusión que la recuperación de información tradicional no es adecuada para colecciones de documentos XML porque, en primer lugar, los documentos XML tienen una estructura jerárquica en forma de árbol y por lo tanto, las consultas debieran permitir hacer referencia no sólo al contenido sino también a detalles de esa estructura. De esta forma, se puede restringir el contexto que interesa al usuario a algún

elemento o elementos concretos de los documentos XML. En consecuencia, en el cálculo de la relevancia se debe reflejar de forma apropiada tanto la estructura del documento como las restricciones que la consulta hace sobre esa estructura. Los autores consideran que los diferentes niveles del árbol tienen distinta importancia para la consulta e introducen un *factor de corrección* que disminuye el valor del peso obtenido tanto más cuanto más descendamos en el árbol del documento, por lo tanto, consideran que los contenidos que aparecen en los primeros niveles de la estructura son más relevantes que los que aparecen en niveles inferiores. El usuario determina de forma dinámica la importancia de los diferentes niveles del árbol cuando determina la granularidad al efectúa una recuperación anidada (*nested retrieval*).

En segundo lugar, los documentos XML guardan una información muy heterogénea y el usuario puede estar interesado sólo en parte del documento. Los autores denominan *categorías* a cada una de estas partes y dependiendo que la consulta se refiera a una (*single-category retrieval*) o a varias (*multi-category retrieval*) categorías, se formulan distintos procedimientos para calcular el peso de los documentos.

La recuperación de una o varias categorías junto con la recuperación anidada constituyen una forma flexible de recuperar información en documentos XML. Estas ideas se ha implementado en PowerDB-XML, que es un sistema que genera rankings consistentes a las consultas de usuario que pueden hacer referencia sólo al contenido o al contenido y a la estructura. Las primeras son similares a las utilizadas en un sistema de recuperación de información tradicional, mientras las segundas permiten además restringir en la consulta una o varias partes de la estructura de los documentos.

Para obtener la relevancia parte de la formulación *tf–idf* del modelo vectorial para responder a consultas que no referencian la estructura. Por otro lado, establece una formulación especial cuando el usuario está interesado en una categoría, otra cuando lo está en varias y una tercera cuando está interesado en un subárbol de la estructura (recuperación anidada) siendo cada formulación una particularización de las siguientes.

Definición 5.6 (Relevancia para una categoría) *Sea e un elemento que pertenece a una categoría cat. Sea t un término y $tf(t,e)$ la frecuencia con que t aparece en e. Sea N_{cat} el número de elementos de la categoría cat y $ef_{cat}(t)$ la frecuencia con que el término t aparece en los elementos de la categoría cat. La relevancia del elemento e para una pregunta q referida a una categoría cat es:*

$$RSV(e,q) = \sum_{t \in terms(q)} tf(t,e) ief_{cat}(t)^2 tf(t,q) \tag{5.4}$$

en dónde la frecuencia inversa de un término t en la categoría se calcula mediante $ief_{cat}(t) = \log \frac{N_{cat}}{ef_{cat}(t)}$

Para calcular la relevancia de un elemento para varias categorías se deben calcular inicialmente las frecuencias correspondientes a cada categoría para integrarlas de la siguiente manera.

Definición 5.7 (Relevancia para varias categorías) *Sea M el conjunto de las categorías y $N_{Mcat} = \sum_{cat \in M}$ el número de elementos que pertenecen a M. Sea $ef_{cat}(t)$ la frecuencia con que el término t aparece en los elementos de la categoría cat, en este caso la frecuencia inversa de un término t se calcula de la siguiente manera $ief_{Mcat}(t) = \log \frac{N_{Mcat}}{\sum_{cat \in M} ef_{cat}(t)}$. La relevancia de un elemento e para una pregunta q referida a varias categorías usando de nuevo el ranking* tf–idf *es:*

$$RSV(e,q) = \sum_{t \in terms(q)} tf(t,e) ief_{Mcat}(t)^2 tf(t,q) \tag{5.5}$$

Se puede comprobar que esta definición es equivalente a la dada para obtener la relevancia de una categoría cuando M esté formado por un único elemento. Finalmente se enunciará la ecuación que permite obtener la relevancia de un subárbol de la estructura, para ello se debe tener en cuenta que un subárbol puede contener varias categorías y también que los términos que aparecen en la raíz del subárbol se deben considerar más relevantes que los que aparecen en niveles inferiores. Se puede observar que las definiciones anteriores son particularizaciones de ésta.

Definición 5.8 (Relevancia para la recuperación anidada) *Sea e el elemento raíz al que se refiere la consulta q y $SE(e)$ el conjunto de elementos que aparecen en el subárbol (incluyendo a e). Sea $aw_l \in [0,1]$ el factor de corrección asociado con el elemento $l \in SE(e)$. La categoría de cada elemento interno, denotada mediante $cat(se)$, se genera de forma dinámica y $ief_{cat(se)}(t)$ es la frecuencia inversa del término t en la categoría $cat(se)$. La relevancia de la recuperación anidada se calcula mediante*

$$\begin{aligned} RSV &= \sum_{se \in SE(e)} \sum_{t \in terms(q)} tf(t, se) \left(\prod_{l \in path(e,se)} aw_l \right) ief_{cat(se)}(t)^2 tf(t,q) \\ &= \sum_{se \in SE(e)} \left(\left(\prod_{l \in path(e,se)} aw_l \right) \sum_{t \in terms(q)} tf(t, se) ief_{cat(se)}(t)^2 tf(t,q) \right) \end{aligned} \tag{5.6}$$

El hecho de suponer que los términos que aparecen en los niveles superiores de la jerarquía son más relevantes que los que aparecen en los inferiores puede convertirse en un inconveniente pues en ciertos tipos de documentos está consideración de importancia se puede invertir.

La propuesta de Torsten Schlieder y Holger Meuss [SM00, SM02] es otra técnica de consultar y recuperar documentos XML que tiene similitudes con el modelo vectorial y la equivalencia entre árboles (*tree matching*). La idea principal es adaptar técnicas de equivalencia entre árboles para permitir enunciar consultas con estructura y que puedan obtener como respuesta documentos XML que respondan de forma parcial a dicha consulta. Para ello se extiende el modelo vectorial para que tenga en cuenta la estructura y, por lo tanto las consultas y los documentos XML se representan mediante árboles etiquetados: las colecciones son un árbol y los *documentos lógicos* son subárboles pertenecientes a dicho árbol. Un *término estructural* es un árbol etiquetado sobre el que se da un conjunto de etiquetas, viene a representar el papel de los términos en documentos sin estructura. Los terminos estructurales aparecen tanto en los documentos lógicos como en la consulta. Se cuenta el número de ocurrencias de cada término estructural dentro de los documentos lógicos y el número de documentos lógicos que contienen un término estructural y normalizando estos valores se obtiene el peso del término. Estos pesos se utilizan para construir los vectores de los documentos; los pesos del vector de la pregunta son definidos por el usuario. Estos vectores son comparados utilizando el producto escalar. Ajustando los parámetros en las ecuaciones se pueden simular tanto el modelo vectorial como la equivalencia entre árboles.

Así pues, se introduce una técnica que genera un ranking de documentos XML respecto de consultas con estructura, pero requiere de un conocimiento previo del usuario sobre la estructura de los documentos de la colección. Al igual que en el caso anterior, cuando se efectúan consultas sin hacer referencia a la estructura se obtiene el ranking utilizando (en este caso, simulando) el modelo vectorial.

En [CMM+03] se presenta una ampliación del modelo vectorial para buscar colecciones XML mediante fragmentos XML y clasificar los resultados según su relevancia. Se describe un modelo que beneficia a las consultas con contextos relativamente específicos pero con necesidades de información imprecisas respecto al modelo vectorial clásico. Aunque no es adecuado para todos los tipos de consultas. Haciendo una semejanza con el modelo vectorial

en dónde las consultas de texto libre y los documentos son objetos de la misma naturaleza, los autores sugieren que la consulta se exprese en la misma forma que los documentos XML, de manera que un "fragmento de XML" se compare con otro "fragmento de XML" para recuperar un objeto del mismo tipo.

Para ello se realiza una extensión del modelo vectorial que permite integrar una medida de semejanza entre caminos XML y se define un nuevo mecanismo de clasificación derivado de la extensión del modelo. El modelo se ha probado con la colección INEX consiguiéndose una precisión muy alta. Los resultados también indican que las consultas que están bien especificadas en términos de los contextos requeridos tienen más posibilidades de recoger los beneficios de las medidas más complejas estadísticas y de de semejanza entre contextos. Sin embargo, estos resultados se deben considerar como resultados iniciales debido a que se ha estudiado un grupo muy limitado de consultas.

Capítulo 6

Modelo de contextos estructurales

La compresión de texto (y en general la compresión de datos) confía en que los datos que se van a comprimir contengan algún tipo de redundancia, puesto que se han generado aplicando ciertas reglas, y que el compresor pueda inferirlas para así predecir con exactitud dichos datos. Un compresor puede reducir el tamaño de un archivo determinando qué datos son más frecuentes para asignarles un código más pequeño que el que se asigna a los datos menos frecuentes. Por lo tanto, la compresión tiene dos etapas cláramente diferenciadas, una se encarga de realizar conjeturas sobre los símbolos más frecuentes, y la otra que emite las "decisiones" de la primera.

Usando la terminología correcta diremos que la primera etapa se denomina "modelo" y trata de predecir la probabilidad de la fuente de información generando una distribución de la probabilidad de los símbolos del alfabeto de entrada. Y la segunda etapa se denomina "codificador" que crea y emite los códigos para los símbolos del alfabeto de entrada basándose en las probabilidades obtenidas por el modelo. Cuanto más cercanas esten las predicciones efectuadas por el modelo a los datos de entrada mayor será la compresión que se obtenga. Para mayor información acerca del modelado y codificación consultar §2.5 y §2.6 respectivamente.

En este punto se presentan las dos cuestiones esenciales de la compresión de datos: ¿qué reglas se deben seguir para asignar probabilidades a los símbolos? y dada una distribución de la probabilidad, ¿cuáles son los códigos más pequeños que se pueden asignar? La teoría de la información da respuesta a estas preguntas: un modelo de Markov y la entropía respectivamente, cuando se usa un único modelo para comprimir.

Ahora bien, cuando se desean comprimir los documentos que manipulan las bases de datos textuales o los sistemas de recuperación de información se plantean algunos requisitos que permiten descartar algunos métodos de compresión. Uno de los mas importantes es la necesidad de acceder al texto de forma aleatoria sin tener que descomprimirlo desde el principio. Esto descarta la mayoría de los métodos adaptativos como los basados en Ziv-Lempel y la codificación artimética. Por otro lado, los modelos semiadaptativos como Huffman [Huf52] proporcionan poca compresión en textos. En el caso de comprimir textos escritos en lenguaje natural, se ha mostrado que es una excelente elección considerar palabras, y no caracteres, como los símbolos de la fuente [Mof89]. Gracias a la distribución sesgada de las palabras, la utilización de este modelo junto con un codificador Huffman proporcionan razones de compresión cercanas al 25 %, mucho mejor que las obtenidas con los mejores modelos adaptativos. Estos resultados empeoran ligeramente si utilizamos una codificación Huffman orientada a byte, en la que cada símbolo de la fuente se codifica como una secuencia de

bytes en lugar de bits. Aunque la razones de compresión se elevan al 30 % (que todavía son competitivas) a cambio se obtiene una descompresión y una búsqueda mucho más rápida, que son características esenciales para las bases de datos comprimidas. Finalmente, el hecho que el alfabeto y el vocabulario de las colecciones de textos coincida permite una búsqueda eficiente y altamente sofisticada, tanto en búsqueda secuencial y en los índices invertidos comprimidos sobre los textos [WMB99, MNZB00, NMN+00, ZMNBY00].

Aunque el área de las bases de datos textuales y los sistemas de recuperación de información con documentos comprimidos escritos en lenguaje natural ha recorrido un largo camino desde el final de los ochenta, es interesante lo poco que se ha hecho al respecto de considerar la estructura de los textos en este marco. Gracias a la amplia aceptación de SGML, HTML y XML como estándares para el almacenamiento, intercambio y presentación de documentos, las bases de datos semiestructuradas se están convirtiendo en un estándar de hecho. Se han propuesto algunas técnicas para explotar la estructura del texto de los documentos, como *XMill* [LS00] y *XMLPPM* [Che01], pero no están diseñados para permitir la búsqueda sobre el texto. Otros, como *XGrind* [TH02], permiten buscar pero no sacan ventaja de la estructura.

En este capítulo se explora la posibilidad de considerar la estructura del texto en el contexto de una base de datos textual o de un sistema de recuperación de información con documentos semiestructurados comprimidos. El objetivo principal es sacar ventaja de la estructura en el proceso de compresión manteniendo las características deseables de una compresión Huffman basada en palabras semiadaptativa. La idea es utilizar modelos separados para comprimir el texto contenido en los diferentes elementos de estructura, es decir, en la etapa de modelado se dispondrá de una regla adicional que permitirá seleccionar un modelo u otro dependiendo del elemento estructural en que que se encuentre el símbolo actual. Por ejemplo, en un archivo de correo electrónico se puede utilizar un modelo diferente para los campos `From:`, `Subject:`, `Date:`, `Body:`, etc.

La posible ganancia que se puede obtener utilizando esta idea es clara, pero el precio que hay que pagar es que se deben almacenar varios modelos en lugar de uno. Este precio puede ser o no ser conveniente. En el ejemplo anterior, el hecho de codificar las fechas por separado es, probablemente, una buena idea, pero codificar los asuntos de los mensajes con un modelo y los cuerpos de los mensajes con otro probablemente no merezca la pena debido a que se necesita un espacio extra para almacenar ambos modelos (p.e. dos árboles de Huffman). De ahí que también se ha diseñado una técnica para *fusionar* los modelos si se puede predecir qué es conveniente en términos de longitudes de archivos comprimidos. Aunque el problema de encontrar fusiones óptimas parece ser un problema combinacional complicado se ha diseñado una heurística que obtiene automáticamente una fusión razonablemente buena de un conjunto inicial de modelos separados, uno por elemento estructural.

6.1. El modelo de contextos estructurales

Un codificador basado en el modelo de alfabetos separados[1] debe manejar dos modelos de símbolos-fuente disjuntos. Con uno se modelarán todos los separadores que aparezcan en el texto y con el otro todas la palabras de dicho texto. Cuando se manejan documentos semiestructurados (como documentos SGML o XML) esta idea continúa siendo adecuada, pero el mecanismo se puede mejorar para obtener un redimiento mejor.

En la mayoría de los casos, los textos escritos en lenguaje natural están estructurados de una manera semánticamente significativa. Esto quiere decir que se puede esperar que, al

[1] ver §3.1.4 en la página 28

menos para algunos elementos estructurales, la distribución de las palabras que componen los textos de un elemento estructural concreto difiere respecto a las de otros elementos. Por ejemplo, en el caso de un archivo de correo electrónico mostrado con anterioridad si las etiquetas estructurales se corresponden con los campos que posee un fichero de correo electrónico, se puede esperar que el campo `From:` contenga nombres y direcciones de correo electrónico, el campo `Date:` contenga fechas y que en el contenido de los campos `Subject:` y `Body:` aparezca texto libre.

La idea subyacente en el modelo de contextos estructurales, o SCM[2], es la utilización de un *modelo de codificación específico* independiente para codificar el texto contenido en un determinado elemento de estructura o *contexto estructural* (por ejemplo, el texto contenido entre las etiquetas XML de apertura y cierre). El uso de modelos diferentes para codificar elementos estructurales diferentes mejorará la razón de compresión cuando la intersección del conjunto de palabras que aparecen bajo un elemento de estructura determinado con el conjunto de palabras de otro es pequeña o la distribución de las palabras es muy diferente.

El modelo propuesto es general y no depende del modelo de codificación específico que se utilice para modelar y codificar los elementos estructurales. Pero, evidentemente, dependiendo del tipo de modelo específico que se utilice se dispondrá o no de determinadas características que son intrínsecas a dicho modelo de codificación específico. Es decir, si dicho modelo de codificación específico es un modelo semiadaptativo será necesario realizar dos pasadas sobre el texto para comprimirlo pero se podrán realizar accesos aleatorios y búsquedas sobre el texto comprimdo. Por el contrario, si el modelo de codificación es adaptativo sólo será preciso realizar una pasada sobre el texto y se obtendrán mejores razones de compresión pero no se podrán realizar accesos aleatorios y búsquedas sobre el texto comprimdo.

En cualquier caso, siempre deberá existir al menos un modelo de codificación específico, llamado *modelo por defecto*, que se utilizará al comienzo del proceso y para codificar aquellos elementos estructurales que no tengan asignado ningún modelo por el motivo que sea. Los sepradores se pueden modelar y codificar en el modelo específico en el que aparezcan

No existe ningún tipo de restricción conceptual que impida utilizar diferentes modelos de codificación específicos en un mismo modelo de contexto. Es decir, se podría optar por seleccionar un determinado tipo de modelo para codificar el contenido de un elemento estructural y utilizar otro tipo de modelo distinto para codificar el contenido de un elemento diferente. Esto puede ser útil cuando el contenido de un documento semiestructurado sea heterogéneo.

Para probar la propuesta de SCM se han definido dos versiones del modelo: SCMWBH y SCMPPM.

SCMWBH es la versión semiadaptativa de SCM pensada para ser utilizada en bases de datos textuales y sistemas de recuperación con compresión. Cada contexto se codifica empleando un codificador de Huffman basado en palabras[3], de manera que se mantienen todas las características de dicho codificador y se obtienen ganancias significantivas respecto a métodos similares insensibles a la estructura.

SCMPPM es la versión adaptativa de SCM que intenta explotar al máximo las propiedades semánticas de la estructura en el proceso de compresión. Cada contexto se codifica empleando un codificador PPMD+ basado en caracteres, pero en este caso, para visualizar o buscar en el documento comprimido es necesario descomprimirlo

[2]Sigla procedente del nombre en inglés del modelo: *Structural Contexts Model*
[3]Word-Based Huffman (WBH)

desde el principio, no obstante es una excelente elección para almacenar o transmitir documentos semiestructurados.

6.2. SCMWBH

El modelo de contextos estructurales, al igual que en el modelo de alfabetos separados, utilizará un modelo, denominado *modelo de separadores*, para almacenar todos los separadores que aparezcan en el texto con independencia del elemento estructural en el que se encuentren. Asimismo, se supone que existe una alternancia entre palabras y separadores, en caso contrario, se deberá insertar bien una "palabra vacía" o bien un "separador vacío". Por último, existirá al menos un modelo utilizado para comenzar a comprimir el texto y codificar las palabras que no se encuentren dentro de ningún elemento estructural.

6.2.1. Comprimiendo del texto

En esta versión el texto se comprimirá utilizando el método de Huffman basado en palabras [Huf52, BSTW86]. Por lo tanto, se considera que el texto está constituido por una secuencia alternada de palabras y separadores, en dónde una palabra es una cualquier secuencia consecutiva máxima de caracteres alfanuméricos del texto y un separador es cualquier secuencia máxima de caracteres no alfanuméricos del texto.

Además se deben tener en cuenta un tipo especial de palabras: las *etiquetas*. Un etiqueta es un código que se inserta en el texto para representar la estructura, el formato o estilo de los datos. La utilización de caracteres delimitadores permite distinguir la etiqueta del texto que la rodea. Habitualmente se suelen utilizar los caracteres `'<'` y `'>'` como los delimitadores de inicio y final de las etiquetas XML o SGML. Por otro lado, para marcar la estructura del documento se suelen utilizar dos tipos de etiquetas: las *etiquetas de inicio* que marcan dónde comienza un elemento estructural, `'<...>'`; y las *etiquetas de final* que marcan dónde termina un elemento estructural, `'</...>'`.

En este caso se considerará la etiqueta completa (es decir, incluyendo sus correspondientes caracteres delimitadores) como una palabra y se utilizarán para determinar cuándo se debe conmutar el modelo en las etapas de modelado, codificación y descodificación. Cuando aparezcan etiquetas de inicio o fin de estructura se deberá utilizar el modelo o contexto correspondiente, para ello se puede emplear el algoritmo 6.1.

Algoritmo 6.1 (Cambio de contexto)

modelo_actual ← *modelo_defecto*
while ∃*símbolos* ***do***
    palabra ← *obtener_símbolo()*
    if (*palabra es separador*)
      then
        almacenar/codificar/descodificar(palabra, modelo_separadores)
      else
        almacenar/codificar/descodificar(palabra, modelo_actual)
        if (*palabra* ∈ *etiquetas_inicio_estructura*)
          then
            push(modelo_actual)
            modelo_actual ← *modelo(palabra)*
          else
            if (*palabra* ∈ *etiquetas_fin_estructura*)

```
                    then
                        modelo_actual ← pop()
                fi
            fi
        fi
od
```

6.2.2. Consideración de bloques de texto

En algunos textos o colecciones pueden existir palabras que sean más habituales en algunas partes (y por lo tanto, tienen una frecuencia de aparición alta) pero menos habituales en otras (frecuencia de aparición baja). Por ejemplo, esto es habitual en colecciones de archivos de "news" en las que existen temas de conversación que hoy están de moda pero mañana dejan de estarlo. Teniendo presente este hecho, existe la posibilidad de aplicar SCM sobre diferentes regiones o bloques de texto conjuntamente con elementos estructurales. Los bloques de texto son fragmentos de un documento o colección consecutivos y de un tamaño b prefijado con anterioridad. Por lo tanto, otra alternativa es tener en cuenta un modelo diferente para cada elemento estructural que esté presente en cada bloque. El objetivo de la división del documento o colección en bloques es mejorar la razón de compresión adaptando las probabilidades de distribución de las palabras a medida que van cambiando en el documento o colección.

Para cada bloque de texto se dispone un modelo por defecto independiente, pero sigue existiendo un único modelo de separadores para todo el documento o colección.

Todos los bloques tienen un tamaño b invariable (excepto quizá el último) tal y como se ha comentado con anterioridad. La elección del tamaño b de los bloques incidirá de forma negativa sobre la velocidad y las necesidades de memoria del sistema, pero no obstante podrá incidir de forma positiva sobre las razones de compresión. En cualquier caso, siempre existirá una sobrecarga que será mayor cuanto más pequeño sea el tamaño del bloque, pues será necesario instanciar más modelos y disponer de más memoria para mantenerlos. El hecho de no considerar bloques de texto en el proceso puede verse como un caso especial en el que $b = \infty$.

Además, cuantos más modelos se fusionen (explicado en §6.2.4) más lento será dicho proceso. Por otro lado, la elección de un tamaño excesivamente grande para los bloques de texto no permite realizar la adaptación a las variaciones de las probabilidades de distribución lo suficientemente rápido.

6.2.3. Estimación de la entropía

La entropía de una fuente[4] es una cantidad que sólo depende del modelo y habitualmente se mide en *bits/símbolo*. También se suele considerar como una función de la probabilidad de distribución de los símbolos de la fuente (bajo el modelo) y hace referencia a la cantidad media de información suministrada por un símbolo emitido por la fuente. Por lo que se puede decir que la entropía nos proporciona un límite inferior de referencia para el tamaño de un fichero comprimido para un determinado modelo.

El teorema fundamental de Shannon [Sha48] establece que la entropía de la distribución de probabilidad $\{p_i\}$ es $\sum_i p_i \log_2(1/p_i)$ bits, o dicho con otras palabras, la manera más óptima para codificar el símbolo i es utilizando $\log_2(1/p_i)$ bits. En un modelo de orden cero la probabilidad de un símbolo se define sin tener en consideración los símbolos que

[4] ver §2.5.1 en la página 14

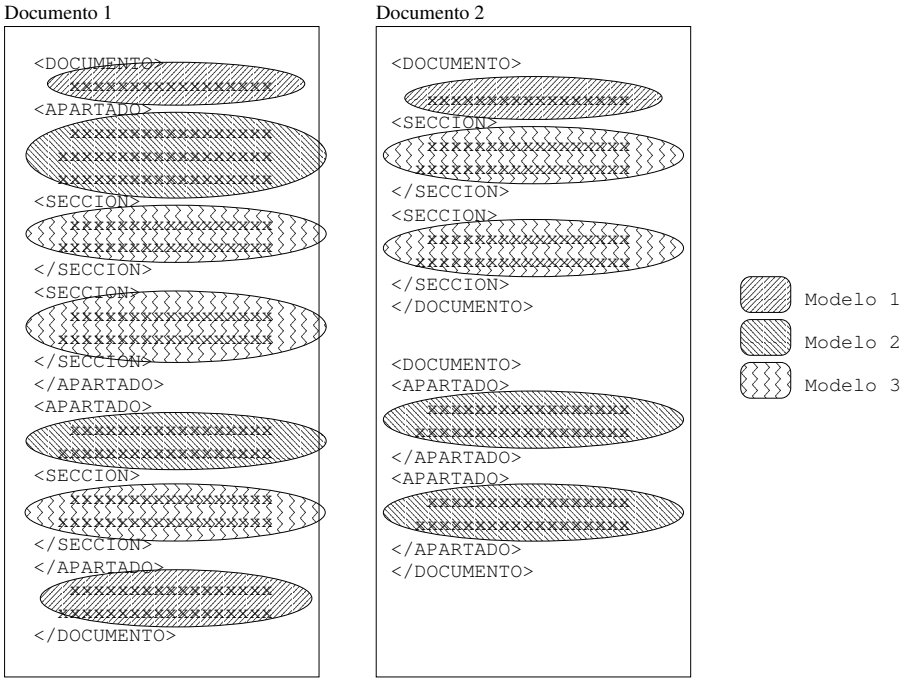


Figura 6.1: Ejemplo de modelos disjuntos.

lo rodean. En la práctica, la probabilidad real de los símbolos no se conoce pero se puede estimar utilizando las frecuencias de los mismos observadas en el texto.

La definición 2.7 en la página 15 permite calcular el valor de la entropía cuando sólo se utiliza un modelo para codificar el texto, si se quiere estimar el valor de la entropía cuando se están utilizando varios modelos disjuntos en la codificación se deberán combinar las entropías de cada modelo.

Definición 6.1 (Modelos disjuntos) *Sea $\mathcal{M} = \{M_1, \ldots, M_N\}$ el conjunto formado por los N modelos de codificación específicos utilizados para codificar un documento. Sea $\mathcal{R} = \{r_1, \ldots, r_n\}$ el conjunto de todas las regiones de texto en las que se ha dividido el documento y $\mathcal{R}_i \subset \mathcal{R}$ el conjunto de todas las regiones de texto del documento que se codifican utilizando el modelo $M_i, \forall i \in 1..N$. Se dice que los modelos $M_1, \ldots, M_N$ son* modelos disjuntos *si y sólo si se cumple*

$$(\cup_{k \in 1..n} \mathcal{R}_k = \mathcal{R}) \ \wedge \ (\forall i \in 1..N \ \nexists j \in 1..N, j \neq i \ \bullet \ \mathcal{R}_i \cap \mathcal{R}_j \neq \emptyset)$$

Es decir, para considerar que N modelos de codificación son modelos disjuntos se debe cumplir que ningún fragmento (o parte del mismo) modelado o codificado con un modelo se vuelva a modelar o codificar con otro modelo diferente. En la figura 6.1 se puede observar de forma gráfica un ejemplo de modelos disjuntos. La definición anterior se ha dado para un único documento, no obstante, extender este concepto para una colección de documentos es inmediato.

Definición 6.2 (Entropía de orden cero de un modelo) *Sea n^d el número total de símbolos que codifica el modelo d, T_v^d el número de símbolos diferentes que codifica el modelo d y f_i^d la frecuencia del símbolo i en el modelo d calculada mediante la ecuación*

$$f_i^d = \frac{\mathrm{o}_i^d}{n^d} \tag{6.1}$$

en donde o_i^d *es el número de veces que el modelo d codifica el símbolo i. Se puede reformular la ecuación 2.7 para obtener la entropía para los símbolos del modelo d:*

$$\mathcal{H}^d = \sum_{i=1}^{T_v^d} f_i^d \log_2 \frac{1}{f_i^d} \tag{6.2}$$

Definición 6.3 (Entropía de orden cero de modelos disjuntos) *Si N es el número total de modelos, la entropía conjunta de orden cero de los N modelos semiadaptativos disjuntos,* $\mathcal{H}$*, es un valor medio ponderado de las estimaciones de las entropías que aporta cada modelo de forma individual (*$\mathcal{H}^d, d \in 1 \ldots N$*):*

$$\mathcal{H} = \frac{\sum_{d=1}^{N} n^d \, \mathcal{H}^d}{n} \tag{6.3}$$

6.2.4. Fusión de modelos semiadaptativos

Como ya se ha comentado, para codificar cada elemento estructural y/o bloque de texto se utilizará un modelo (semiadaptativo) específico. Posiblemente esta elección no produzca unos resultados óptimos debido a la sobrecarga que se produce cuando se almacenan en el fichero comprimido los diccionarios de los modelos. Más concretamente, se logrará una mejora en la razón de compresión si se fusionan dos modelos cuyos diccionarios tienen muchos símbolos en común y con distribuciones de probabilidad similares.

En esta sección se comentará un método general que se ha desarrollado y que permite agrupar eficientemente elementos estructurales y/o bloques de texto similares bajo un mismo modelo adaptativo y, por lo tanto, se agrupan los símbolos comunes y con distribución de probabilidad similar de ambos modelos. Se utiliza el factor de entropía para estimar el tamaño del texto comprimido con un modelo determinado, aunque en una aplicación práctica se puede ejecutar el algoritmo de Huffman para obtener el tamaño exacto del texto comprimido con el modelo. No obstante el factor de entropía proporciona estimaciones muy próximas al valor exacto.

Definición 6.4 (Contribución estimada al tamaño estimada de un modelo) *Sea* $\mathcal{V}^d$ *el tamaño en bits del volcabulario que forma el diccionario del modelo d, y sea* $\mathcal{H}^d$ *la entropía de orden cero calculada para dicho modelo. Se puede estimar que la contribución del modelo d al tamaño final del documento mediante la siguiente expresión*

$$\mathcal{T}^d = \mathcal{V}^d + n^d \mathcal{H}^d \tag{6.4}$$

Teniendo en cuenta la definición anterior se llega a la conclusión que se los modelos i y j (con sus respectivos diccionarios) se fusionarán bajo un sólo modelo (usando un único diccionario) cuando la suma de sus contribuciones al tamaño estimadas sea mayor que la contribución al tamaño estimada del modelo resultante de la unión de ambos. Es decir, cuando

$$\mathcal{T}^i + \mathcal{T}^j > \mathcal{T}^{i \cup j} \tag{6.5}$$

Para calcular $\mathcal{T}^{i \cup j}$ se debe obtener la unión de los vocabularios y la estimación de la entropía de la unión. Esto se puede realizar en tiempo lineal respecto a los tamaños de los vocabularios.

Definición 6.5 (Ahorro estimado de una fusión) *Sea* $\mathcal{A}^{i \cup j}$ *el ahorro estimado de la fusión de los modelos i y j. Entonces*

$$\mathcal{A}^{i \cup j} = \mathcal{T}^i + \mathcal{T}^j - \mathcal{T}^{i \cup j} \tag{6.6}$$

El algoritmo 6.2 lleva a cabo el proceso óptimo de fusión. Comienza asignando un modelo independiente a cada elemento estructural y bloque (si fuera preciso), además de un modelo por defecto por bloque. Recordemos que el modelo que se utiliza para modelar y codificar los separadores no interviene en este proceso, por lo que no figura en dicho proceso. El objetivo principal del algoritmo es obtener grupos de modelos diferentes los cuales codificarán textos con similares características y para identificar esos textos similares se utilizará el concepto de entropía. El algoritmo va fusionando parejas de modelos de manera progresiva hasta que se estime que no existe ninguna otra fusión entre parejas de modelos con la que se obtenga una mejora de compresión. Puede parecer que el proceso de la obtención óptima de los grupos es un problema computacionalmente complejo, pero se utilizará una heurística que resuelve el dicho problema razonablemente rápido y obteniendo buenos resultados.

Se comenzará calculando las contribuciones estimadas al tamaño final, $\mathcal{T}^i$ para cada modelo i $\forall i \in 1..N$, y además se calcularán las contribuciones conjuntas $\mathcal{T}^{i\cup j}$ para todos las parejas i, j $\forall i \in 1..N-1,\ j \in i+1..N$ posibles de modelos a la vez que se va obteniendo el ahorro estimado para cada posible fusión de modelos $\mathcal{A}^{i\cup j}$. Una vez efectuados todos los cálculos se procederá a fusionar los modelos i y j que hayan obtenido el mayor ahorro $\mathcal{A}^{i\cup j}$ positivo. Para realizar la fusión se eliminan los modelos i y j del conjunto y se introduce un nuevo modelo generado equivalente a a la unión de ambos. Una vez introducido el modelo $i \cup j$ en el conjunto sólo será preciso calcular su contribución estimada al tamaño y las contribuciones conjuntas (junto con el valor del ahorro estimado) del nuevo modelo con el resto. Este proceso se repite hasta que se obtengan todos los valores $\mathcal{A}^{i\cup j}$ negativos.

Algoritmo 6.2 (Fusión de modelos)

do
  $mayor\_ahorro \leftarrow 0$
  for $1 \leq i < j \leq N$ ***do***
    $ahorro\_actual \leftarrow \mathcal{T}^i + \mathcal{T}^j - \mathcal{T}^{i\cup j}$
    if $(ahorro\_actual > mayor\_ahorro)$
      then
        $mayor\_ahorro \leftarrow ahorro\_actual$
        $bi \leftarrow i$
        $bj \leftarrow j$
    fi
  od
  if $(mayor\_ahorro > 0)$
    then
      $d_{bi} \leftarrow fusionar\_modelos(d_{bi}, d_{bj})$
      $d_{bj} \leftarrow d_N$
      $N \leftarrow N - 1$
  fi
od ***while*** $(mayor\_ahorro > 0)$

En el algoritmo se han omitido los detalles de cómo se calculan y obtienen los valores $\mathcal{T}$ de las estimaciones con por motivos de claridad y legibilidad del código. Inicialmente su coste es $O(VN^3)$ cuando existen N modelos y el tamaño del vocabulario es V. No obstante, este coste se puede reducir a $O(VN^2 \log N)$ realizando pequeñas mejoras sobre el código como por ejemplo recalcular sólo los ahorros estimados del nuevo diccionario introducido como se ha comentado con anterioridad, o mantener almacenadas las parejas de modelos en una cola de prioridad por ahorro estimado.

6.3. Evaluación del modelo

Para analizar de forma empírica y evaluar el rendimiento del modelo en su versión semiadaptativa se ha desarrollado un prototipo que implementa el modelo de contextos estructurales utilizando un código de Huffman orientado a palabra como modelo específico. Se ha utilizado una codificación artimética de caracteres para comprimir los diccionarios de cada código de Huffman. Las pruebas se han realizado en un PC con un procesador Pentium III a 500 MHz, 512 Mbytes de memoria principal y Linux como sistema operativo.

6.3.1. Las colecciones de prueba

Para efectuar los experimentos se han seleccionado diferentes tamaños de las colecciones WSJ, ZIFF y AP procedentes del conjunto de colecciones de experimentación TREC-3 [Har95]. En la tabla 6.1 se pueden observar las características generales de las colecciones. Es importante destacar que cada subcolección está formada por un conjunto de documentos y que además incluye la subcolección de tamaño menor inmediato, por lo tanto los documentos que están presentes en una determinada subcolección también lo están en la subcolecciones de mayor tamaño.

Tamaño	#P.T.	#P.V.	Razón
TREC-WSJ			
1.221.659	193.899	18.380	9,479 %
5.516.592	874.586	38.750	4,430 %
10.510.481	1.669.506	52.218	3,127 %
21.235.547	3.370.544	71.832	2,131 %
42.113.697	6.690.067	97.190	1,452 %
62.963.963	10.015.765	116.221	1,160 %
104.942.941	16.672.690	144.701	0,867 %
TREC-ZIFF			
1.021.882	161.900	12.924	7,982 %
6.083.389	992.067	35.555	3,583 %
11.164.171	1.821.015	51.094	2,805 %
21.306.059	3.489.650	71.136	2,038 %
42.659.558	6.970.106	102.737	1,473 %
62.966.279	10.272.649	125.326	1,219 %
105.709.264	17.289.782	165.113	0,954 %
TREC-AP			
1.185.968	195.915	19.103	9,750 %
5.805.776	956.340	41.263	4,314 %
10.469.592	1.721.137	54.058	3,140 %
21.219.693	3.486.098	73.820	2,117 %
42.523.572	6.985.763	101.480	1,452 %
63.343.648	10.411.824	122.340	1,175 %
105.018.927	17.252.119	157.376	0,912 %

Tabla 6.1: Características de las colecciones. Para cada colección se muestra su tamaño en bytes (Tamaño), el número total de palabras (#P.T.), el número de palabras diferentes o número de términos del vocabulario (#P.V.) y la relación entre ambos (Razón).

La estructura de los documentos que forman cada una de las tres colecciones es muy similar: tienen un sólo nivel de estructura que delimita el documento mediante la etiqueta estructural <DOC> y dentro de cada documento existen otras etiquetas estructurales que se utilizan para indicar el identificador de documento, la fecha, el título, el autor, la fuente que originó el documento, el contenido del mismo, las palabras clave, etc.

6.3.2. Análisis del rendimiento

En esta sección se analizará el rendimiento del modelo semiadaptativo de acuerdo a todo lo comentado con anterioridad: se comprobará si se obtienen mejoras respecto a la técnica básica (el modelo de alfabetos separados), la bondad de la técnica de fusión de modelos y del uso de bloques de texto.

La velocidad media para comprimir las colecciones es de 128 Kbytes/seg aproximadamente cuando no se utlizan bloques de texto. En este valor está incluido el tiempo necesario para modelar, fusionar los modelos y comprimir el texto. El tiempo empleado para fusionar los modelos varía desde los 4.35 segundos en las colecciones de un megabyte hasta los 40,27 segundos para las colecciones de cien megabytes. El impacto del tiempo de fusión es mayor para las colecciones más pequeñas (en torno al 50 % del tiempo total), pero es mucho menos significativo para las colecciones grandes (en torno al 5 % del tiempo total). Esto se debe a que el tiempo necesario para fusionar los modelos es lineal respecto al tamaño del vocabulario, el cual crece sublinealmente respecto al tamaño de la colección [Hea78] y habitualmente está cercano a $O(\sqrt{n})$. Además el tiempo de fusión depende de forma cuadrática del número de etiquetas de estructura diferentes presentes en la colección, generalemente este número suele ser pequeño y no crece a medida que crece el tamaño de la colección sino que depende de la DTD ("Definición de Tipo de Documento") de los documentos.

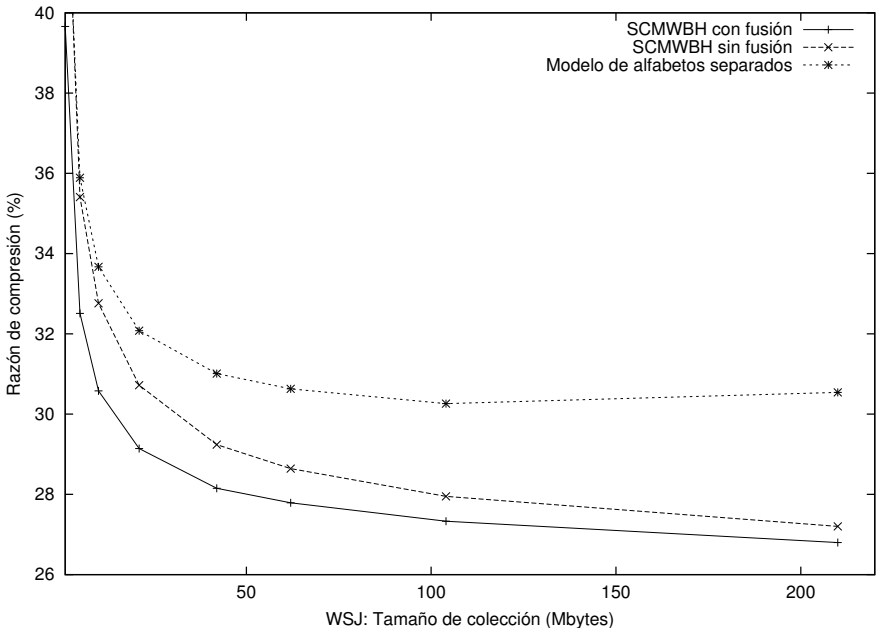
En la tabla 6.2 y en la figura 6.3.2 se puede ver una comparación de los resultados de compresión obtenidos por el modelo de alfabetos separados (MAS, un "ancestro elemental" del modelo de contextos estructurales) y el modelo de contextos estructurales (SCM) aplicando o no la técnica de fusión de modelos para las subcolecciones de WSJ y esta vez, añadiendo una subcolección de unos 200 Mbytes aproximadamente[5]. Para textos pequeños el tamaño del vocabulario es significativo respecto al tamaño del texto y consecuentemente SCM sin fusión paga un alto precio al separar los modelos no obteniendo mejora alguna sobre el modelo de alfabetos separados, pero a medida que el tamaño de la colección aumenta el impacto del vocabulario se va reduciendo y para tamaños grandes SCM sin fusión obteniene aproximadamente un 10 % de compresión adicional respecto al modelo de alfabetos separados. Por otro lado, SCM con fusión obtiene unos resultados similares al modelo sin fusión (un 12,25 % de compresión adicional respecto al modelo de alfabetos separados) pero su rendimiento es mucho mejor en el caso de los textos pequeños en el que comienza obteniendo una mejora en la compresión en torno a un 11 % en la colección de de 1 Mbyte respecto al modelo sin fusión.

A partir de este momento y en vista de los resultados anteriores sólo se considerarán las razones de compresión obtenidas aplicando la técnica de fusión de modelos. En la tabla 6.3 se muestra el tamaño original de cada colección junto a los tamaños comprimidos y razones de compresión correspondientes procedentes de aplicar el modelo de contextos estructurales con fusión de modelos. Se puede comprobar que las razones de compresión son mejores para colecciónes mayores y cuando el impacto del vocabulario es menor [Hea78], tal y como se puede contrastar en la tabla 6.1.

[5]Concretamente, el tamaño exacto de dicha subcolección es de 210.009.482 bytes y aparecen 33.359.445 palabras en total, de las cuales 194.324 son diferentes (0,582 %)

WSJ	SCM con fusión		SCM sin fusión		MAS	
Original	Compr.	Ratio	Compr.	Ratio	Compr.	Ratio
1.221.659	484.575	39,66 %	545.348	44,64 %	543.882	44,52 %
5.516.592	1.793.950	32,51 %	1.953.425	35,41 %	1.979.904	35,89 %
10.510.481	3.214.613	30,58 %	3.443.233	32,76 %	3.538.878	33,67 %
21.235.547	6.190.051	29,14 %	6.523.560	30,72 %	6.812.363	32,08 %
42.113.697	11.858.566	28,15 %	12.314.045	29,24 %	13.059.457	31,01 %
62.963.963	17.498.136	27,79 %	18.032.879	28,64 %	19.285.861	30,63 %
104.942.941	28.681.879	27,33 %	29.331.552	27,95 %	31.755.733	30,26 %
210.009.482	56.282.542	26,80 %	57.122.579	27,20 %	64.136.895	30,54 %

Tabla 6.2: Compression ratios using different models, for WSJ.



La tabla 6.4 muestra el número de modelos que se han fusionado. La columna "Inicial" indica con cuantos modelos se comenzó el proceso: el modelo por defecto y el de separadores más uno por elemento estructural diferente con la excepción del elemento `<DOC>` que se codifica utilizando el modelo por defecto y que señala el principio y el final de un documento. La columna "Final" indica el número de modelos que se utilizarán una vez ejecutado el proceso de fusión.

Para dar una idea de los modelos fusionados se comentará un ejemplo: para las subcolecciones pequeñas de WSJ los modelos que codificaban los elementos `<DOCNO>` y `<DOCID>` se fusionaron pues ambos contenían números y referencias internas (constituidas también por números). Otro grupo de modelos que se fusionaron fueron los que codificaban los elementos estructurales `<HL>`, `<LP>` y `<TEXT>` que contienen los textos de las noticias (cabeceras, resúmenes de los teletipos y cuerpos respectivamente). Para las subcolecciones de mayor tamaño sólo se fusionó el último grupo de tres modelos ya que el impacto de almacenar más vocabularios es menor y, por lo tanto, se efectúan menos fusiones. Esto demuestra que la idea intuitiva de fusionar los modelos cuyos vocabularios son similares es correcta.

TREC-WSJ			TREC-ZIFF			TREC-AP		
Original	Compr.	Ratio	Original	Compr.	Ratio	Original	Compr.	Ratio
1.221.659	484.575	39,66 %	1.021.882	376.180	36,81 %	1.185.968	492.832	41,55 %
5.516.592	1.793.950	32,51 %	6.083.389	1.956.195	32,15 %	5.805.776	1.952.979	33,63 %
10.510.481	3.214.613	30,58 %	11.164.171	3.480.842	31,17 %	10.469.592	3.315.087	31,66 %
21.235.547	6.190.051	29,14 %	21.306.059	6.414.762	30,10 %	21.219.693	6.371.426	30,02 %
42.113.697	11.858.566	28,15 %	42.659.558	12.452.756	29,19 %	42.523.572	12.307.072	28,94 %
62.963.963	17.498.136	27,79 %	62.966.279	18.131.869	28,79 %	63.343.648	18.054.387	28,50 %
104.942.941	28.681.879	27,33 %	105.709.264	29.972.861	28,35 %	105.018.927	29.479.824	28,07 %

Tabla 6.3: Sizes and compression ratios for each collection with merge.

Tamaño	TREC-WSJ		TREC-ZIFF		TREC-AP	
Aprox.(Mb)	Inicial	Final	Inicial	Final	Inicial	Final
1	11	8	10	4	9	5
5	11	8	10	4	9	5
10	11	8	10	4	9	7
20	11	9	10	6	9	7
40	11	9	10	6	9	7
60	11	9	10	6	9	7
100	11	9	10	7	9	7

Tabla 6.4: Número de modelos utilizados.

El método empleado para predecir la aportación al tamaño final de los modelos en el proceso de fusión ha sido muy preciso ya que los valores calulados como predicciones suponían aproximadamente el 99 % del valor real.

Aprox.	Chunk size (Mbytes)				
Size(Mb)	∞	2	4	8	16
1	39,66 %	39,66 %	39,66 %	39,66 %	39,66 %
5	32,51 %	32,51 %	32,51 %	32,51 %	32,51 %
10	30,58 %	30,57 %	30,57 %	30,58 %	30,58 %
20	29,14 %	29,13 %	29,13 %	29,13 %	29,14 %
40	28,15 %	28,13 %	28,13 %	28,14 %	28,14 %
60	27,79 %	27,76 %	27,76 %	27,76 %	27,77 %
100	27,33 %	27,28 %	27,28 %	27,28 %	27,29 %

Tabla 6.5: Compression ratios using differents chunk sizes in Mbytes. ∞ size shows compression ratio without using chunks.

A continuación se considerará el impacto debido a la utilización de bloques de texto. La tabla 6.5 muestra una comparativa de las razones de compresión obtenidas cuando se utilizan bloques de diferentes tamaños en las mismas subcolecciones de WSJ. Los resultados no han sido significativos ya que la mayor ganacia obtenida ha sido del 0,03 %. Esto puede ser debido a las características de la colección WSJ: todos los textos tienen una probabilidad de distribución de las palabras muy uniforme en todo el texto. De hecho, todos los modelos

de bloques diferentes que codifican los elementos <HL>, <LP> y <TEXT> se han fusionado. Por otro lado, el tiempo necesario para fusionar todos los modelos crece de manera proporcional al número de modelos. Estos resultados se pueden extrapolar a todas las colecciones TREC involucradas en este estudio. Se puede concluir que en este caso en concreto la utilización de bloques de textos no produce ningún beneficio.

6.3.3. Comparación con otros

Por último se ha comparado el prototipo que implementa el modelo de contextos estructurales con fusión de modelos contra otros sistemas de compresión. Por un lado se han escogido sistemas de compresión que consideran la estructura de los documentos durante la compresión y por otro se han escogido sistemas de compresión de propósito general.

Entre los primeros se encuentran *XMill* [LS00] y *XMLPPM* [Che01], los cuales se han descrito en §?? pero se recordará que *Xmill* es un compresor específico para XML basado en los algoritmos de Ziv-Lempel y Huffman y capaz de manejar la estructura de los documentos y que *XMLPPM* también es un compresor específico para XML pero en este caso basado en PPM.

Entre los segundos se encuentra *MG* [WMB99] que es un software de dominio público, versátil y de propósito general utilizado para comprimir y recuperar textos e imágenes[6] y también sistemas de compresión estándar muy utilizados en la práctica: (1) *compress* de UNIX, que implementa el algoritmo LZW; (2) *zip* y (3) *gzip*, que utilizan el algoritmo LZ77 junto a una variante del algoritmo de Huffman; (4) *bzip2*, que aplica una codificación de Huffman a un bloque de texto previamente transformado mediante la transformación de Burrows-Wheeler.

Se han comprimido todas las colecciones con todos sistemas[7]. La tablas 6.6, 6.7 y 6.8 muestran las razones de compresión obtenidas por *MG*, *XMill* y *XMLPPM* para los diferentes tamaños de colección de WSJ, ZIFF y AP respectivamente. Por otro lado, las razones de compresión obtenidas para cada tamaño de colección por los compresores de propósito general se muestran en las tablas 6.9, 6.10 y 6.11, Para realizar la comparación, se han utilizado la razón de compresión media obtenida por cada compresor para cada tamaño de colección. Dichos valores medios se pueden observar en las tablas 6.12 y 6.13.

En la figura 6.3 se muestra gráficamente la comparación entre todos sistemas. *XMill*, *compress*, *zip* y *gzip* han obtenido una razón de compresión media aproximadamente constante para todos los tamaños de colección debido a que utilizan variantes de esquemas LZ para comprimir. Su compresión no es competitiva en este experimento.

Por el contrario, la razón media de compresión obtenida por *bzip2* también se puede considerar aproximadamente constante para todos los tamaños de colección pero, en este caso, es mejor que SCMWBH debido a que aplica una codificación tipo Huffman sobre un bloque de texto al que se le ha aplicado la transformación de Burrows-Wheeler con anterioridad, hecho que le permite obtener razones de compresión muy buenas y próximas a las obtenidas por los compresores PPM.

En esta misma línea, *XMLPPM* ha obtenido la mejor compresión en todos los casos, demostrando que la idea de utilizar contextos estructurales durante el proceso de compresión es buena. Pero *XMLPPM* es un sistema de compresión adaptativo y, consecuentemente, no es apto para realizar accesos directos en bases de datos textuales o sistemas de recuperación de información que manejen grandes volumenes de información. Nótese que la diferencia

[6] MG comprime los documentos utilizando una variante del código de Huffman basado en palabras denominada *Huffword* y no considera la estructura de los doumentos.

[7] Para comprimir las colecciones con XMLPPM se necesitó realizar unas pequeñas modificaciones en las fuentes, pero éstas no afectaron a la compresión de la colección.

	XMLPPM		MG		XMILL	
Original	Compr.	Ratio	Compr.	Ratio	Compr.	Ratio
1.221.659	326.312	27,71 %	425.744	34,84 %	458.752	44,89 %
5.516.592	1.468.464	26,61 %	1.710.099	30,99 %	2.084.864	34,27 %
10.510.481	2.798.920	26,62 %	3.160.119	30,06 %	3.985.408	35,69 %
21.235.547	5.661.663	26,61 %	6.234.389	29,35 %	8.069.120	37,87 %
42.113.697	11.253.925	26,72 %	12.163.403	28,88 %	16.039.936	37,60 %
62.963.963	16.855.946	26,77 %	18.095.467	28,73 %	24.039.424	38,17 %
104.942.941	27.975.875	26,65 %	30.010.169	28,59 %	39.858.176	37,70 %

Tabla 6.6: WSJ

	XMLPPM		MG		XMILL	
Original	Compr.	Ratio	Compr.	Ratio	Compr.	Ratio
1.021.882	225.679	22,08 %	333.027	32,58 %	323.584	27,28 %
6.083.389	1.429.176	23,49 %	1.841.153	30,26 %	2.043.904	35,20 %
11.164.171	2.671.332	23,92 %	3.322.720	29,76 %	3.805.184	36,34 %
21.306.059	5.128.324	24,06 %	6.241.441	29,29 %	7.286.784	34,34 %
42.659.558	10.293.422	24,12 %	12.324.378	28,88 %	14.581.760	34,29 %
62.966.279	15.184.153	24,11 %	18.089.592	28,73 %	21.495.808	33,93 %
105.709.264	25.544.517	24,16 %	30.251.563	28,61 %	36.155.392	34,42 %

Tabla 6.7: ZIFF

	XMLPPM		MG		XMILL	
Original	Compr.	Ratio	Compr.	Ratio	Compr.	Ratio
1.185.968	324.598	27,36 %	418.260	35,26 %	454.654	37,21 %
5.805.776	1.567.422	26,99 %	1.794.699	30,91 %	2.199.552	39,87 %
10.469.592	2.809.926	26,83 %	3.140.100	29,99 %	3.936.256	37,45 %
21.219.693	5.678.039	26,75 %	6.210.040	29,26 %	7.925.760	37,32 %
42.523.572	11.403.823	26,81 %	12.229.573	28,75 %	15.912.960	37,78 %
63.343.648	17.009.410	26,85 %	18.106.319	28,58 %	23.752.704	37,72 %
105.018.927	28.259.538	26,90 %	29.861.213	28,43 %	39.428.096	37,57 %

Tabla 6.8: AP

	compress		zip	
Original	Compr.	Ratio	Compr.	Ratio
1.221659	513.219	42,01 %	452.961	37,07 %
5.516592	2.274.373	41,22 %	2.048.596	37,13 %
10.510481	4.299.009	40,90 %	3.902.830	37,13 %
21.235547	8.624.461	40,61 %	7.892.494	37,16 %
42.113697	17.232.659	40,91 %	15.879.609	37,70 %
62.963963	25.448.959	40,41 %	23.272.506	36,96 %
104.942941	42.582.725	40,57 %	39.081.826	37,24 %
	gzip		bzip2	
1.221659	452.842	37,06 %	335.324	27,44 %
5.516592	2.048.477	37,13 %	1.481.542	26,85 %
10.510481	3.902.711	37,13 %	2.829.938	26,92 %
21.235547	7.892.375	37,16 %	5.717.221	26,92 %
42.113697	15.879.490	37,70 %	11.511.852	27,33 %
62.963963	23.478.276	37,28 %	17.012.427	27,01 %
104.942941	39.081.707	37,24 %	28.189.940	26,86 %

Tabla 6.9: WSJ

	compress		zip	
Original	Compr.	Ratio	Compr.	Ratio
1.021.882	398.049	38,95 %	317.976	31,11 %
6.083.389	2.412.159	39,65 %	1.980.826	32,56 %
11.164.171	4.464.634	39,99 %	3.685.389	33,01 %
21.306.059	8.579.227	40,26 %	7.050.128	33,08 %
42.659.558	17.145.089	40,19 %	14.128.415	33,11 %
62.966.279	25.333.447	40,23 %	20.845.809	33,10 %
105.709.264	42.597.729	40,29 %	35.079.942	33,18 %
	gzip		bzip2	
1.021.882	317.856	31,10 %	238.009	23,29 %
6.083.389	1.980.706	32,55 %	1.494.959	24,57 %
11.164.171	3.685.269	33,00 %	2.791.323	25,00 %
21.306.059	7.050.008	33,08 %	5.359.636	25,15 %
42.659.558	14.128.295	33,11 %	10.737.330	25,16 %
62.966.279	20.845.689	33,10 %	15.823.506	25,13 %
105.709.264	35.079.822	33,18 %	26.644.276	25,20 %

Tabla 6.10: ZIFF

	compress		zip	
Original	Compr.	Ratio	Compr.	Ratio
1.185.968	501.523	42,28 %	446399	37,64 %
5.805.776	2.411.981	41,54 %	2182160	37,58 %
10.469.592	4.330.405	41,36 %	3906082	37,30 %
21.219.693	8.754.305	41,25 %	7880527	37,13 %
42.523.572	17.497.091	41,14 %	15830369	37,22 %
63.343.648	26.051.877	41,12 %	23624214	37,29 %
105.018.927	43.200.789	41,13 %	39213732	37,33 %
	gzip		bzip2	
1.185.968	446.281	37,63 %	335.746	28,30 %
5.805.776	2.182.042	37,58 %	1.592.134	27,42 %
10.469.592	3.905.964	37,30 %	2.847.587	27,19 %
21.219.693	7.880.409	37,13 %	5.752.327	27,10 %
42.523.572	15.830.251	37,22 %	11.547.530	27,15 %
63.343.648	23.624.096	37,29 %	17.219.058	27,18 %
105.018.927	39.213.614	37,33 %	28.601.000	27,23 %

Tabla 6.11: AP

Tamaño	SCMWBH	MG	XMill	XMLPPM
1	39,34 %	34,22 %	36,46 %	25,38 %
5	32,76 %	30,72 %	36,44 %	25,70 %
10	31,13 %	29,93 %	36,49 %	25,79 %
20	29,75 %	29,30 %	36,51 %	25,80 %
40	28,76 %	28,83 %	36,55 %	25,88 %
60	28,36 %	28,67 %	36,61 %	25,91 %
100	27,91 %	28,54 %	36,56 %	25,90 %

Tabla 6.12: Razones medias obtenidas para cada tamaño de colección.

Tamaño	compress	zip	gzip	bzip2
1	41,08 %	35,27 %	35,26 %	28,95 %
5	40,80 %	35,75 %	35,66 %	28,94 %
10	40,75 %	35,81 %	35,81 %	26,37 %
20	40,70 %	35,79 %	35,79 %	26,39 %
40	40,74 %	36,01 %	36,01 %	26,54 %
60	40,58 %	35,78 %	35,89 %	26,44 %
100	40,66 %	35,91 %	35,91 %	26,43 %

Tabla 6.13: Razones medias

entre *XMLPPM* y *SCMWBH* disminuye cuanto más grande es la colección. En cualquier caso, la diferencia de compresión es un precio pequeño por permitir el acceso directo sobre los documentos comprimidos.

SCMWBH es mejor que *MG* para tamaños de colección medianos y grandes, pero no para los pequeños. Esto se debe a la penalización en el almacenamiento que paga SCM al manejar más de un modelo. *SCMWBH* comienza a superar a *MG* a partir de 40 Mbytes y en 100 Mbytes mejora a *MG* en un 2,25 %.

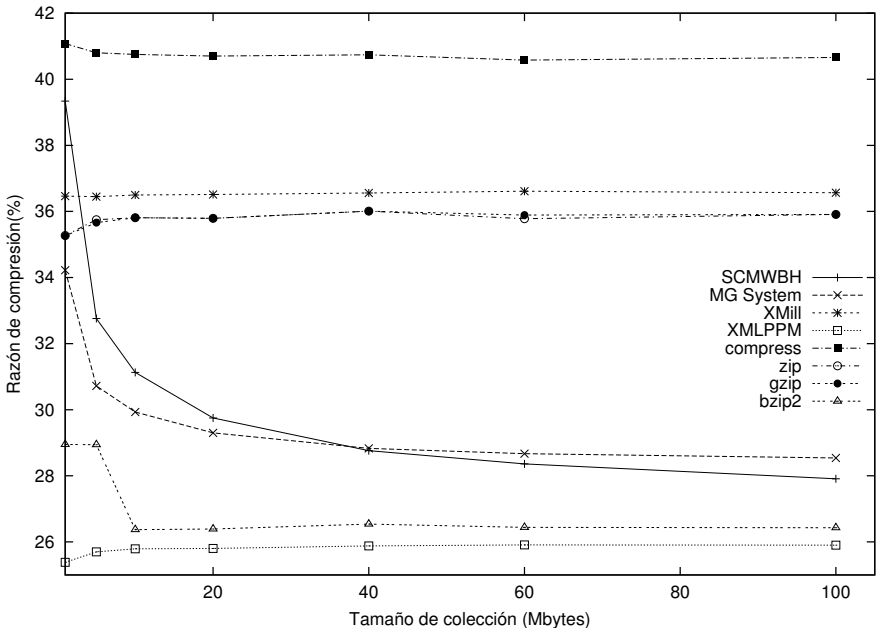


Figura 6.2: Comparación entre SCMWBH y otros.

Por otro

6.4. SCMPPM

A la vista de los resultados obtenidos en la versión adaptativa de SCM, en esta sección se tratará de explotar tanto como sea posible los aspectos semánticos de los elementos estructurales en el proceso de compresión. Como se ha comprobado, la estructura de los documentos guarda un cierto significado semántico que los compresores clásicos no aprovechan y si se sacrifica la posibilidad de acceder de forma directa sobre los textos comprimidos se pueden alcanzar razones de compresión francamente buenas.

La idea continúa siendo la misma pero en este caso se utilizan diferentes modelos PPM para codificar los símbolos que aparecen dentro de un tipo concreto de elemento estructural, de manera que, probablemente, el modelo PPM realice mejores predicciones que cuando procesa los contenidos de varios elementos estructurales. La utilización de modelos PPM separados mejorará la razón de compresión cuando la distribución de los símbolos que se encuentran en tipos diferentes de elementos estructurales es muy diferente. No obstante, existe una penalización adicional al emitir tantos símbolos de escape como símbolos nuevos repetidos en cada modelo, en lugar de uno. Además, se necesita disponer de más memoria para mantener los diferentes modelos PPM (uno por cada tipo de elemento estructural) en

lugar de uno. No obstante, si se utiliza un sólo modelo para codificar el texto, éste puede ser un modelo de mayor orden para que utilice la misma cantidad de memoria.

6.4.1. Variantes de SCMPPM

Originalmente, los modelos PPM están pensados para trabajar con caracteres por lo que la idea de tener un modelo (modelo de separadores) para codificar exclusivamente los separadores puede estar en entredicho pues, al trabajar con caracteres, es necesario señalar el final de una palabra o separador mediante algún símbolo especial. Una alternativa a la inclusión de dicho símbolo puede ser la eliminación del modelo de separadores y codificar los separadores en el modelo del contexto estructural correspondiente. De esta forma se obtienen dos versiones de SCMPPM:

SCMPPM 1.1 mantiene la idea original de SCM y utiliza un modelo exclusivo para codificar todos los separadores que aparecen en el texto. No obstante, como los modelos manejan caracteres será necesario incluir un símbolo especial que indique el final de una palabra y otro que indique el final de un separador. A estos símbolos se les denominará *símbolos de sincronización*. Por otro lado, existirá un modelo PPM para codificar el texto contenido en cada elemento estructural del documento y un modelo por defecto mediante el cual se codificará el texto que no está incluido en ningún elemento estrucutural.

SCMPPM 1.2 modifica la idea original y elimina el modelo de separadores del esquema. En este caso, los caracteres que forman los separadores se codificarán en los modelos que se encargan de codificar los símbolos que forman cada contexto estructural. Al igual que en el caso anterior existirá un modelo PPM para codificar el texto contenido en cada elemento estructural, aparte del modelo por defecto.

Inicialmente, no queda claro cuál de las dos versiones obtendrá mejores resultados. Por un lado, la versión 1.1 sobrecarga los modelo introduciendo un símbolo especial de fin de palabra/separador con una frecuencia de aparición realmente alta. Por otro lado, la versión 1.2 evita introducir símbolos especiales pero sobrecarga los modelos codificando en cada uno los caracteres que forman los seperadores.

Una posible solución encaminada a evitar sobrecargar los modelos es la de utilizar un modelo PPM orientado a secuencias de caracteres o palabras, de manera que se pueda utilizar el esquema original sin ningún tipo de modificación. Pero estudios sobre la variante de PPM orientada a palabras han llegado a la conclusión que ésta obtiene peores resultados que la variante orientada a caracteres [Mof89]. No obstante la inclusión de este tipo de modelos en el esquema SCM general puede mejorar la compresión obtenida con los modelos orientados a caracteres debido a que al separar las palabras de los separadores se predicen qué palabras siguen a otra dentro de un mismo contexto estructural y qué separadores siguen a uno dado en todo el texto. Además también se sigue obteniendo cierta ventaja de la estructura al codificar textos similares con diferentes modelos. El problema que presentan las variantes de PPM orientadas a palabras es evidente: la gran cantidad de memoria que necesitan.

6.4.2. Evaluación del modelo

Se ha implementado un par de prototipos que implementan las versiones 1.1 y 1.2 de SCMPPM y que se han utilizado para analizar de forma empírica y evaluar el rendimiendo de ambas. Se ha elegido la variante PPMD+ para implementar ambos prototipos [CW84b].

Para realizar los experimento se han escogido las mismas subcolecciones con las que se ha probado SCMWBH (ver §6.3.1 en la página 87) y cada elemento de estructura diferente se codificará utilizando un modelo PPMD+ independiente. Recordemos que dichas colecciones tienen un único nivel de estructura, en el que la etiquieta `<DOC>` indica dónde comienzan los documentos y dentro de cada documento aprecen diferentes etiqueta que se utilizan para marcar cuál es el identificador de documento, la fecha, el título, el autor, el origen, el contenido, las palabras clave, etc. Al igual que en el caso anterior, se supone que el texto contenidos dentro de cada elemento estructural se modelará y codificará empleando un modelo PPMD+ independiente.

En la tabla 6.4.2 se puede observar una comparativa entre SCMPPM contra la técnica básica, es decir, utilizar un sólo modelo PPMD+ para modelar y codificar todo el texto. En la versión de SCMPPM se ha utilizado $k = 5$ como valor del orden máximo de los modelos de contexto usados por cada PPMD+, por otro lado, para la técnica básica se ha utilizado el valor $k = 6$ para que la cantidad de memoria requerida por ambas técnicas sea aproximadamente la misma. Dicha tabla muestra las razones de compresión medias para las tres colecciones de prueba, se puede observar que SCMPPM obtiene mejoras por encima del 1 % a medida que el tamaño de la colección aumenta.

Tamaño	PPMD+	SCMPPM
1	24,44 %	24,89 %
5	22,65 %	22,78 %
10	22,12 %	22,10 %
20	21,61 %	21,55 %
40	21,20 %	21,08 %
60	20,99 %	20,84 %
100	20,80 %	20,60 %

También se ha comparado SCMPPM (con $k = 5$) respecto a otros sistemas de compresión conocidos, por un lado se considerarán los compresores que no consideran la estructura o aspectos de la misma a la hora de comprimir y por otro lado los que sí la que tienen presente. Entre los primeros se incluirán *MG* y compresores de propósito general (*compress*, *zip*, *gzip*, *bzip2*), y entre los segundos *XMill*, *XMLPPM* y la versión semiadaptativa de SCM, *SCMWBH*, descrita con anterioridad. Las razones de compresión medias obtenidas para cada tamaño de colección aparecen en las tablas 6.12 y 6.13. En ellas se puede observar que los compresores de propósitos general obtienen razones de compresión aproximadamente constantes, asimismo *XMill* también las obtiene debido a que utiliza *zlib* como motor de compresión. Las razones de compresión obtenidas por todos ellos, excepto *bzip2*, no son competitivas en este experimento: SCMPPM mejora la compresión en un 77 %. Entre los compresores de propósito general, las mejores razones de compresión las ha obtenido sin lugar a dudas *bzip2*, el cual aplica una codificación de tipo Huffman a un bloque de texto previamente transformado mediante la transformación de Burrows-Wheeler, hecho que permite obtener razones de compresión a las obtenidas por la famila de compresores estadísticos PPM, no obstante, SCMPPM mejora la compresión obtenida por *bzip2* en un 28 %. Por otro lado y tal y como se espera de los métodos de Huffman basados en palabras, *MG* y *SCMWBH* mejoran sus razones de compresión a medida que aumenta el tamaño de la colección pero la compresión obtenida por SCMPPM las mejora en más del 35 %. Por último, *XMLPPM* se ha mostrado como la alternativa más competitiva a SCMPPM pero obtiene una razón de compresión aproximadamente constante y, consecuentemente, no

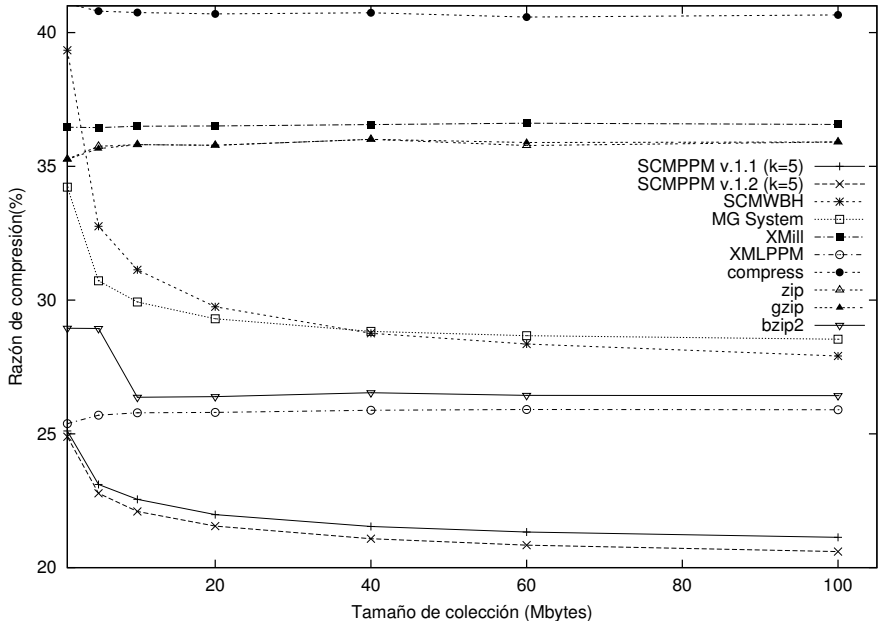


Figura 6.3: Comparación entre SCMPPM y otros.

mejora sus razones de compresión a medida que aumenta el tamaño del texto a comprimir. Esto es por lo que para la colección de 1 Mbyte las razones de compresión de ambos son similares pero para la de 100 Mbytes SCMPPM mejora en un 25 % la compresión obtenida por *XMLPPM*.

Se han realizado diferentes mediciones variando el orden máximo k de los modelos de contexto que utiliza cada PPMD+ para predecir la probabilidad de aparición del símbolo siguiente. Por lo tanto, cada subcolección se ha comprimido con cada prototipo siete veces, variando el orden máximo en cada ocasión. Como orden máximo se han elegido todos los valores pertenecientes al intervalo $k \in [3.,9]$. En las tablas A.1, A.2 y A.3 se pueden observar las razones de compresión obtenidas por cada prototipo y orden máximo k para las colecciones WSJ, ZIFF y AP respectivamente.

Aunque no existen diferencias muy marcadas entre ambas versiones se puede observar que, en general, para valores de k pequeños la versión 1.2 es mejor que la 1.1, pero a medida que va aumentando k la compresión mejora y esa diferencia va disminuyendo debido a que ambos se aproximan a órdenes de contexto en los que se pueden predecir un gran número de palabras completas, y esto implica que el modelo del texto sea más exacto que al utilizar grupos pequeños de caracteres [BCW90].

En las figuras 6.4 y 6.5 se puede observar la evolución de las razones de compresión a medida que varía el valor de k y el tamaño de colección WSJ para las versión 1.1 y 1.2 respectivamente. No existen diferencias significativas entre ambas versiones (ver tabla A.1) y las gráficas para las colecciones AP y ZIFF son muy similares por lo que no se muestran.

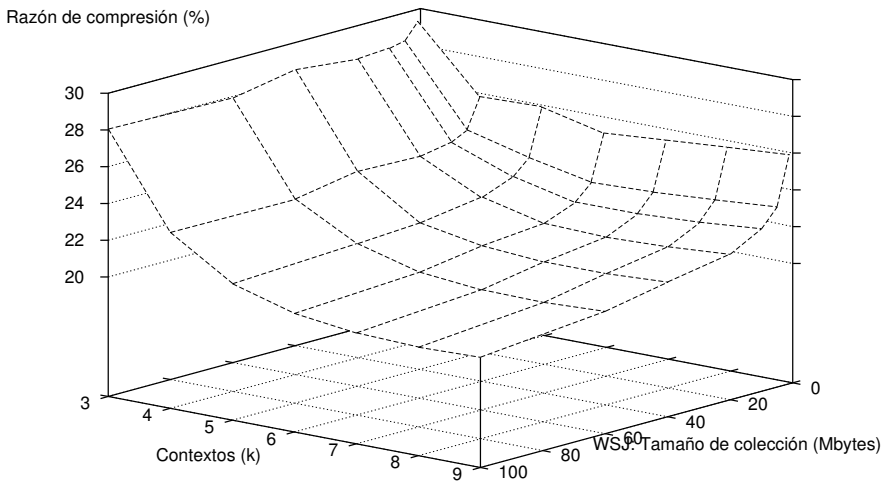


Figura 6.4: SCMPPM 1.1: Razones de compresión variando k.

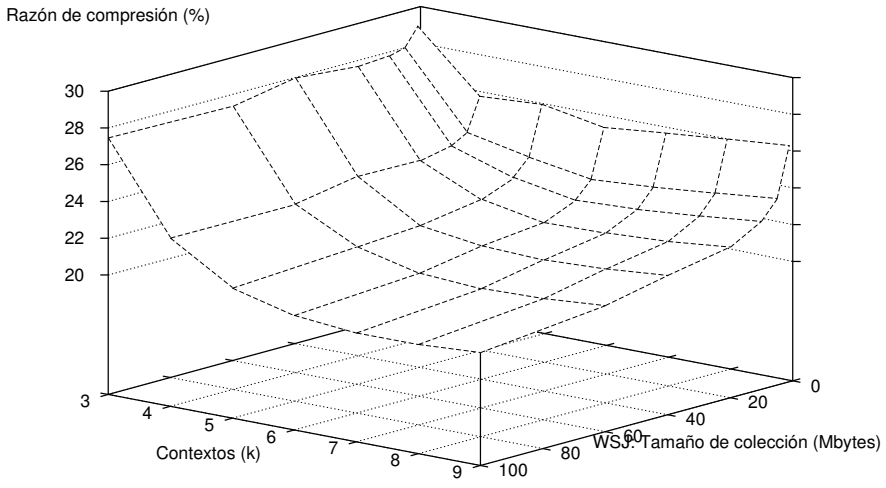


Figura 6.5: SCMPPM 1.2:Razones de compresión variando k.

Capítulo 7

Un esquema LZ sobre estructura

En el capítulo 6 se ha propuesto un modelo de compresión genérico que aprovecha la estructura de los documentos para mejorar la compresión. Como se ha visto, cuando se aplica dicho modelo para comprimir los documentos que contiene las bases de datos textuales y los sistemas de recuperación de información se deben utilizar modelos específicos semiadaptativos para permitir efectuar búsquedas sobre los textos codificados y acceder de forma aleatoria sobre los mismos. Ahora bien, cuando se desea comprimir documentos altamente estructurados (cada vez más utilizados en la denominada "empresa electrónica") también se puede utilizar el modelo de contextos estructurales pero, en este tipo de documentos, existe una redundancia a nivel estructural que no está contemplada en dicho modelo.

El presente capítulo se presenta una propuesta encaminada a comprimir de forma eficiente textos muy estructurados (que generalmente representan datos estructurados), como por ejemplo los formularios en los que dentro de cada campo aparece una cadena de texto pequeña. Las colecciones compuestas por este tipo de documentos/formularios contienen mucha redundancia que no es manipulada correctamente por los métodos de compresión clásicos. Al mismo tiempo, se desea que la colección comprimida pueda ser navegable, visualizable y accesible de forma sencilla. Existen métodos de compresión que tienen en cuenta la estructura pero que no tienen en consideración estas capacidades: los textos se deben descomprimir antes de acceder a ellos.

Se ha desarrollado un método de compresión inspirado en el esquema de compresión Lempel-Ziv en el que se colapsan subárboles repetidos bajo una referencia. El método se ha denominado LZCS, siglas procedentes de "Lempel-Ziv Compression of Structure", su denominación anglosajona. Se ha obtenido una compresión excelente que supera a la compresión proporcionada por los métodos clásicos y competitiva con los métodos que consideran la estructura durante el proceso de compresión.

Además el algoritmo LZCS codifica el texto realizando una única pasada sobre el texto por lo que puede emitir el texto comprimido a medida que procesa el mismo sin comprimir. Esto hace que LZCS sea adecuado para utilizarlo sobre una red de comunicación sin introducir ningún retardo en la transmisión. La salida del algoritmo LZCS sigue siendo un texto plano, lo que facilita su transmisión sobre canales ASCII específicos. La salida del algoritmo LZCS se puede comprimir, en una segunda pasada, utilizando cualquier método de codificación pero es preferible que se utilice un método de codificación que mantenga las propiedades de navegabilidad y acceso aleatorio sobre el texto codificado.

7.1. Descripción de LZCS

LZCS es una nueva técnica para comprimir texto estructurado que permite navegar por la estructura comprimida. Por lo tanto, LZCS se puede integrar en un sistema de recuperación con textos estrucurados sin pérder eficiencia cuando se efectúan las búsquedas o cuando se visualizan los resultados. La idea principal se basa en el concepto introducido por Ziv y Lempel, de manera que los elementos estructurales y los bloques de texto repetidos se reemplazan por una referencia a la primera aparición en el texto procesado. El resultado es un texto estructurado válido con etiquetas especiales adicionales que se utilizan para representar las referencias en el texto. Además, dicho texto estructurado se puede transmitir, manipular o visualizar empleando los mecanismos habituales, o incluso se puede comprimir utilizando algún compresor estándar sin pérdida.

Los documentos se visualizan de forma habitual hasta que aparezca una referencia que, como ya se ha comentado, se encuentra representada mediante una etiqueta especial. Cuando aparece una referencia será preciso introducir la posición actual en una pila (para poder reanudar el proceso una vez que la referencia se haya resuelto) y a continuación se realiza un desplazamiento a la posición indicada por la referencia. La referencia se resolverá (finalizará) cuando se alcance la correspondiente etiqueta que señale el final del elemento estructural si la posición referenciada comienza con una etiqueta que indica el inicio de un elemento estructural. Si, por el contrario, la referencia no comienza con una etiqueta de inicio, se resolverá cuando aparezca una etiqueta de inicio. Cuando termina el texto referenciado se debe sacar de la pila la posición de origen previa y el proceso continua. En algunos casos pueden aparecer otras referencias en el texto referenciado, en este caso se repite el mismo proceso. Se puede emplear un procedimiento similar para buscar o navegar por la estructura arbórea del documento.

Puesto que los documentos que genera LZCS son navegables, una buena idea es comprimirlos empleando un método de compresión semiadaptativo, como por ejemplo Huffman orientado a palabra. Después de esto los documentos no se pueden visualizar como texto convencional (pues es necesario descomprimirlos) pero continúan siendo navegables y accesibles de manera aleatoria.

A continuación se describirá formalmente la transformación LZCS.

7.1.1. Definición formal

Definición 7.1 (Bloque de texto) *Un bloque de texto será cualquier secuencia maximal de caracteres alfanuméricos consecutivos que no contenga etiquetas que señalen el inicio o fin de un elemento estructural o representen una referencia.*

Definición 7.2 (Elemento estructural) *Un elemento estructural será cualquier secuencia de caracteres consecutivos que comience con una etiquena de inicio de elemento estructural y que termine con la etiqueta de final de elemento estructural correspondiente.*

Teniendo presente la definición anterior, un elemento estructural puede contener uno o más bloques de texto, uno o más elementos estructurales y/o una o más referencias. Para simplificar, los otros tipos de etiquetas válidas que pueden aparecer en un documento XML (como por ejemplo etiquetas de comentario, etiquetas autocontenidas y demás) serán tratadas como si se tratasen de texto convencional y sólo las etiquetas que marcan el principio y el final de los elementos estructurales se utilizarán para identificar dichos elementos.

Es habitual representar la estructura de los documentos en forma de árbol. Si en este caso se utiliza esa representación, los bloques de texto se representarán mediante las hojas y los elementos estructurales se representarán mediante subárboles.

Definición 7.3 (Nodo) *Un nodo será bien un bloque de texto, bien un elemento estructural.*

Una vez definido el concepto de nodo se puede decir que el objetivo principal de LZCS es sustituir subárboles por referencias a subárboles equivalentes que aparecieron con anterioridad.

Definición 7.4 (Nodos equivalentes) *Sean $\mathcal{N}_1$ y $\mathcal{N}_2$ dos nodos que aparecen en una colección. Diremos que el nodo $\mathcal{N}_1$ es equivalente al nodo $\mathcal{N}_2$ si y sólo si $\mathcal{N}_1$ es textualmente igual a $\mathcal{N}_2$.*

Definición 7.5 (Transformación LZCS) *La transformación LZCS sustituye cada nodo maximal equivalente a otro que apareció con anterioridad por una referencia a la primera ocurrencia del mismo en el texto. El resto de elementos permanece inalterado. Por "maximal" se entiende que el nodo sustituido no es un descendiente de otro que pueda ser reemplazado.*

Una *referencia* se representa en la salida mediante una etiqueta especial, esta etiqueta especial se construye mediante los símbolos `<@` y `>` que se utilizan para marcar respectivamente el pricipio y el final de la etiqueta de referencia. El contenido de esta etiqueta estará formado por dígitos que expresan una cantidad entera positiva, la cual indica la posición absoluta (evidentemente, dicha posición será menor que la posición actual) en la que comienza el elemento referenciado. Por cuestiones de optimización de espacio (y consecuentemente, para mejorar la razón de compresión) se sugiere expresar este número en una base grande, por ejemplo en base 62 utilizando como dígitos los caracteres `0..9`, `A..Z` y `a..z`.

En algún momento puede suceder que un bloque de texto referenciado sea más pequeño que la propia etiqueta de referencia (como por ejemplo cuando el bloque de texto está formado únicamente por el caracter `'\n'`). Bajo estas circustancias, el hecho de sustituir el bloque por la referencia no es una buena elección pues el resultado de la sustitución provoca un aumento de tamaño, justo el resultado contrario que se desea obtener. Por lo tanto no se sustituirán los bloque de texto que sean más pequeños que un parámetro l especificado por el usuario. La elección del valor del parámetro l influye sobre la razón de compresión, pero no sobre la validez de la transformación.

7.1.2. Ejemplo

A continuación se ilustrará el funcionamiento de la transformación LZCS mediante un ejemplo. Para simplificar supongamos que se desea comprimir con LZCS una colección formada por tres documentos estructurados. La estructura de los documentos está representada en la figura 7.1 y está formada por tres elementos estructurales diferentes. Cada tipo de elemento estructural está representado mediante un círculo: el elemento estructural de tipo 1 está representado por un círculo dibujado con una línea continúa, el elemento estructural de tipo 2 dibujado con una línea discontinua y el de tipo 3 dibujado con una línea punteada. Los bloques de textos están representados mediante cuadrados y las letras y números actúan como los identificadores de los nodos.

Para cubrir todas las posibilidades, supongamos que los bloques de texto identificados en la figura mediante 1, 4, 7 y 9 son equivalentes. También son equivalentes los bloques 3 y 10 por una parte y los bloques 6 y 8 por otra. Con estas equivalencias entre bloques de texto los documentos tienen partes repetidas (o dicho con otras palabras, los documentos tienen

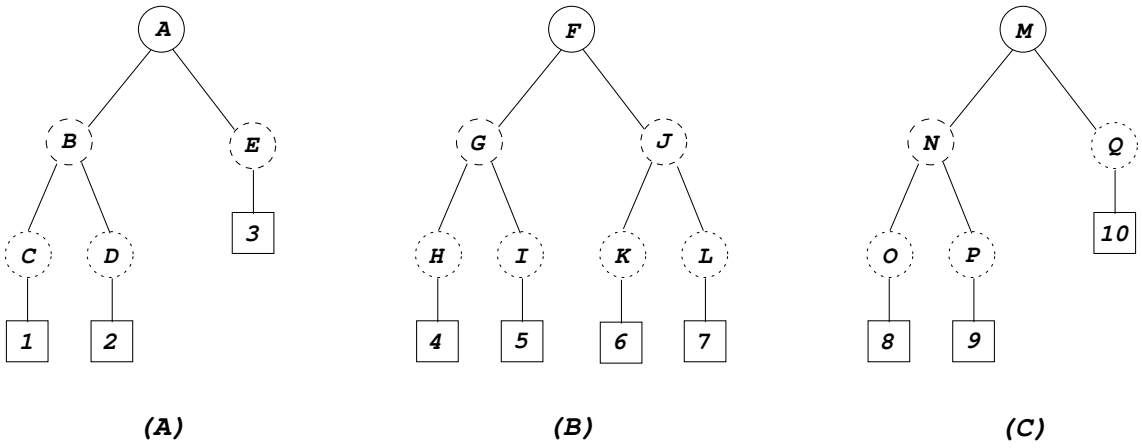


Figura 7.1: Tres documentos de ejemplo.

subárboles equivalentes). La figura 7.2 muestra de una manera visual estas correspondencias y la figura 7.3 muestra la colección transformada con LZCS, en la que los triángulos representan a las referencias.

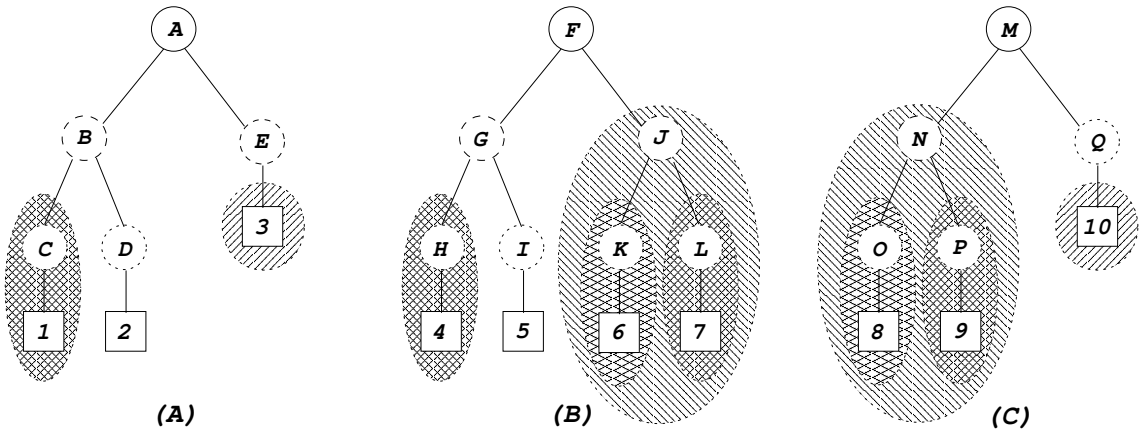


Figura 7.2: Subárboles equivalentes de los documentos.

7.2. Una implementación eficiente

La forma más sencilla de implementar la transformación LZCS es buscar cada nuevo elemento estructural en la totalidad del texto procesado. Haciéndolo así se obtiene un algoritmo cuya complejidad temporal es de $O(n^2)$, que es inaceptable. Se mostrará como obtener una complejidad temporal media de $O(n)$ empleando técnicas de hashing.

Una vez procesado un bloque de texto se obiene su firma digital, por ejemplo utilizando el algortimo MD5 [Riv92]. Una vez obtenida la firma digital se comprueba si el bloque es equivalente a alguno previo, para ello se comparan la firma digital del bloque con las firmas digitales de los bloques anteriores que se encuentran en una tabla hash, si el bloque no es equivalente se copia literalmente en la salida y su firma se añade al conjunto (tabla hash) de firmas de bloques de texto "originales" junto con la posición en la que aparece, que es la

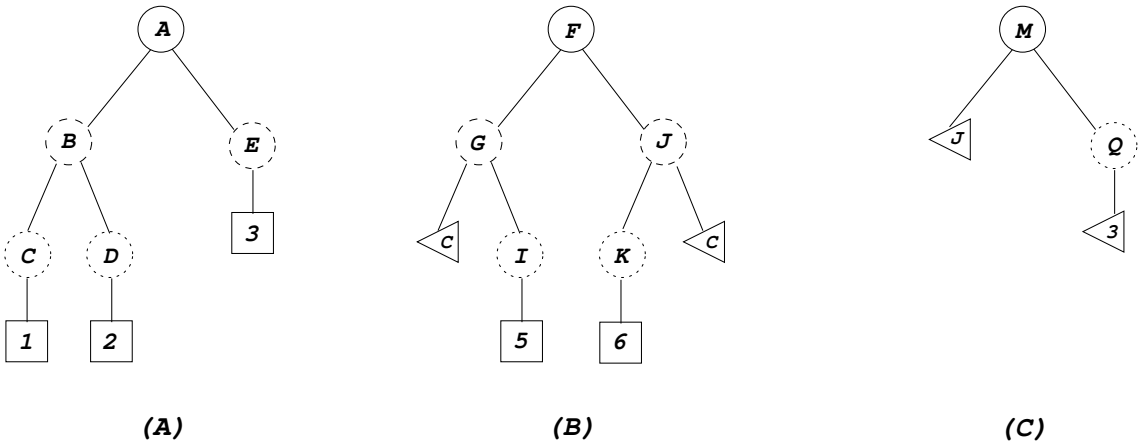


Figura 7.3: Los tres documentos de ejmplo después de aplicar la transformación LZCS. Las referencias están representadas mediante triángulos.

primera ocurrencia de dicho bloque. Por el contrario, si se encuentra un bloque equivalente (se encontró una coincidencia entre las firmas digitales) se obtiene la posición de la primera ocurrencia del bloque y en la salida se escribe una referencia a dicha posición. Puesto que los algoritmos de firma digital no aseguran la unicidad de la firma, cuando aparece una coincidencia los textos se comparan para asegurar que no se obtienen *falsas coincidencias* y de esta manera evitar corromper los datos al realizar la transformación.

Por otro lado, para poder aplicar técnicas de hashing a los elementos de estructura se debe generar una *firma de nodo* para poder almacenarla junto con su posición de inicio si el nodo en cuestión no ha aparecido con anterioridad. Al contrario que sucede con los bloques de texto, no existen algoritmos estándar para crear firmas de nodo y por lo tanto se ha propuesto uno.

7.2.1. Firma digital de un nodo

La generación de la firma de un nodo se realiza de abjo hacia arriba (*bottom–up*), por lo tanto para obtener la firma de un nodo padre será preciso obtener la firma de todos sus nodos hijos.

Definición 7.6 (Firma de nodo) *La firma de un nodo se genera mediante la concatenación de la cadena que identifica su etiqueta de inicio de estructura con los identificadores de los nodos hijos. Éstos son: bien su posición de inicio en el texto si no son referencias o bien las posiciones que referencian en caso contrario.*

El lema 7.2 demuestra que la firma de un nodo en una colección es única. Cada vez que se procesa un elemento estructural se obtiene su firma de nodo y se comprueba que dicha firma ha aparecido con anterioridad. El nodo se puede sustituir por una referencia si la firma del nodo obtenida coincide con alguna firma ya obtenida, es decir, el nodo actual es equivalente con algún otro previo.

Lema 7.1 *Sean $\mathcal{N}_1$ y $\mathcal{N}_2$ dos nodos que aparecen en una colección transformada con LZCS hasta $\mathcal{N}_2$ y además $\mathcal{N}_1$ precede a $\mathcal{N}_2$. Entonces $\mathcal{N}_1$ es equivalente a $\mathcal{N}_2$ si y sólo si $\mathcal{N}_2$ es una referencia a $\mathcal{N}_1$ o $\mathcal{N}_1$ y $\mathcal{N}_2$ son referencias iguales.*

Prueba: Se demostrará la equivalencia en ambas direcciones.

1. Si $\mathcal{N}_1$ es equivalente a $\mathcal{N}_2$ entonces debe ocurrir que $\mathcal{N}_2$ es una referencia a $\mathcal{N}_1$ o bien que $\mathcal{N}_1$ y $\mathcal{N}_2$ son referencias iguales porque la transformación LZCS sustituye el nodo $\mathcal{N}_2$ por una referencia a la primera aparición del nodo:

   a) Si $\mathcal{N}_1$ es la primera ocurrencia entonces $\mathcal{N}_2$ se sustituye por una referencia a $\mathcal{N}_1$.

   b) En caso contrario, sea $\mathcal{N}_0$ la primera ocurrencia de $\mathcal{N}_2$, entonces $\mathcal{N}_2$ se sustituye por una referencia a $\mathcal{N}_0$, pero también $\mathcal{N}_1$ fué sustituido por una referencia a $\mathcal{N}_0$.

2. Si $\mathcal{N}_2$ es una referencia a $\mathcal{N}_1$, o $\mathcal{N}_2$ y $\mathcal{N}_1$ son referencias iguales, entonces $\mathcal{N}_1$ es equivalente a $\mathcal{N}_2$, porque sucede que tanto el contenido de $\mathcal{N}_1$ como el contenido de $\mathcal{N}_2$ son textualmente iguales, o bien el contenido de $\mathcal{N}_1$ es textualmente igual al contenido de $\mathcal{N}_0$ y además también el contenido de $\mathcal{N}_2$ es textualmente igual al contenido de $\mathcal{N}_0$. □

A continuación, y teniendo presente el resultado del lema 7.1, se demostrará que la firma de un nodo es única y que la forma de generarla es correcta.

Lema 7.2 *Los nodos $\mathcal{N}$ y $\mathcal{N}'$ son equivalentes si y sólo si sus firmas de nodo son iguales.*

Prueba: Se puede observar que para que un nodo se repita es necesario que todos sus hijos también se repitan. Entonces, un nodo $\mathcal{N}$, padre de $\mathcal{N}_1 \ldots \mathcal{N}_k$, es textualmente igual al nodo previo $\mathcal{N}'$, padre de $\mathcal{N}'_1 \ldots \mathcal{N}'_k$, si y solo si los identificadores de las etiquetas de $\mathcal{N}$ y $\mathcal{N}'$ son iguales y $\forall i \in 1..k, \mathcal{N}'_i$ es equivalente a $\mathcal{N}_i$. La última parte del lema 7.1 dice que $\mathcal{N}'_i$ es una referencia a $\mathcal{N}_i$ o que $\mathcal{N}'_i$ es una referencia a algún $\mathcal{N}_0$ y que además $\mathcal{N}_i$ referencia a dicho $\mathcal{N}_0$. Entonces, si se utiliza la definición 7.6 para generar las firmas de los nodos $\mathcal{N}$ y $\mathcal{N}'$, las firmas de los nodos serán iguales y sólo si los nodos son equivalentes. □

A continuación se comentará brevemente el algoritmo que implementa la transformación LZCS. La firma de un nodo se calcula cuando aparece la etiqueta que señala el final del correspondiente elemento estructural, una vez calculada se busca en el conjunto (tabla hash) de firmas de nodos. El nodo actual se sustituye por una referencia si su firma está presente en el conjunto. No obstante en este punto, no se tiene la certeza que el nodo actual sea un subárbol repetido máximo y, por lo tanto, la sustitución sólo se efectúa en memoria y no se escribe nada en la salida. Por el contrario, si la firma del nodo actual no está presente en el conjunto de firmas entonces el subárbol actual no es equivalente a ningún otro previo y, consecuentemente, todos los nodos hijos no volcados en la salida (con el propósito de buscar subárboles maximales) y el nodo actual se deben escribir en la salida y, por último la firma del nodo actual se añade al conjunto de firmas de nodos.

La figura 7.1 muestra el algoritmo básico que realiza la transformación LZCS. El conjunto *SubárbolPrevio* contiene los elementos estructurales que se ha sustituido por una referencia pero que todavía no se escrito en la salida debido a que se está buscando un posible elemento estructural maximal que los contenga, por lo tanto, si en un momento dado se está procesando un nodo el conjunto *SubárbolPrevio* contendrá los ascendientes y los hermanos izquierdos del nodo en cuestión. Por su parte, *NodeSigSet* y *TextSigSet* contienen los conjuntos de firmas de los nodos y bloques de texto respectivamente que han aparecido hasta el momento.

Algoritmo 7.1 (Algoritmo básico de la transformación LZCS)

```
NodeSigSet ← ∅
TextSigSet ← ∅
SubárbolPrevio ← ∅
while ∃nodos do
        nodo_actual ← obtener_nodo()    // en postorden
        if (nodo_actual es un bloque_de_texto)
          then
                firma_actual ← MD5(nodo_actual)
                if (firma_actual ∈ TextSigSet)
                  then
                        referencia ← TextSigSet.referencia(firma_actual)
                        SubárbolPrevio.add(referencia)
                   else
                        posición_actual ← PosiciónInicial(nodo_actual)
                        TextSigSet.añadir(firma_actual, posición_actual)
                        Emitir SubárbolPrevio
                        Emitir nodo_actual
                        SubárbolPrevio ← ∅
                fi
           else
                firma_actual ← FirmarNodo(nodo_actual)
                if (firma_actual ∈ NodeSigSet)
                  then
                        referencia ← NodeSigSet.referencia(firma_actual)
                        SubárbolPrevio.borrar_hijo(nodo_actual)
                        SubárbolPrevio.añadir(referencia)
                   else
                        posición_actual ← StartPosition(nodo_actual)
                        NodeSigSet.añadir(firma_actual, posición_actual)
                        Emitir SubárbolPrevio
                        Emitir nodo_actual
                        SubárbolPrevio ← ∅
                fi
        fi
od
Emitir SubárbolPrevio
```

Por el contrario, la descompresión es muy sencilla: en la salida se empieza escribiendo el texto y cuando se llega a una etiqueta de referencia se escribe el texto que se encuentra en la posición referenciada. El texto referenciado finalizará cuando se alcance una etiqueta que marque el final de un elemento estructural, si dicho texto comienza con la etiqueta que indica el inicio de estructura correspondiente. Pero si el texto referenciado no comienza con una etiqueta de inicio de estructura entonces la referencia se resolverá cuando se alcance la primera etiqueta de inicio de estructura.

7.2.2. Ejemplo

Para ilustrar con un ejemplo el funcionamiento del algortimo propuesto se retomará la colección de documentos del ejemplo de §7.1.2 en la página 103. Los documentos se procesarán de izquierda a derecha según están representados en la figura 7.1. En el primer documento no se realiza ninguna sustitución porque en dicho documento no existe ningún nodo equivante. La salida contiene una copia exacta del primer documento cuando se empieza a procesar el segundo documento. Ahora bien, el bloque de texto 4 se sustituye por una referencia al bloque de texto 1 ya que son equivalentes. En la figura 7.4-A esta referencia está representada por un triángulo. Como el elemento estructural que contiene el bloque de texto 4 también coincide con el que contiene el bloque de texto 1 (en este caso tenemos dos nodos equivalentes), la referencia anterior se vuelve a sustituir por otra que contiene al elemento estructural (figura 7.4-B). Ocurre lo mismo con el bloque de texto 7 (figuras 7.4-C y 7.4-D).

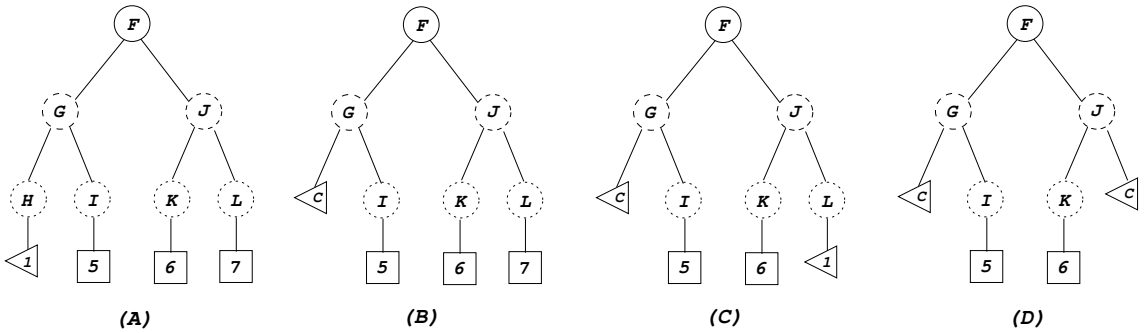


Figura 7.4: Representación paso a paso de las sustituciones que se han efectuado en el segundo documento.

Finalmente se empieza a procesar el tercer documento y, en primer lugar, se efectúan las sustituciones de los bloques de texto 8 y 9, así como las sustituciones de sus correspondientes elementos estructurales (paso a paso las sustituciones se muestran en las figuras 7.5-A a 7.5-D). Una vez que se ha procesado el elemento estructural N se verifica que se puede sustituir completemente por el elemento J porque ambos son equivalentes: tienen el mismo número de hijos y sus hijos son equivalentes uno a uno y de izquierda a derecha (figura 7.5-E). Por último, se sustituye el bloque de texto 10 por una referencia al bloque equivalente 3 (figura 7.5-F). En este caso no se puede sustituir el elemento estructural Q por una referencia al E puesto que no son equivalentes (un mismo contenido está ubicado en tipos de elementos estructurales diferentes).

7.3. Evaluación de la propuesta

El modelo LZCS se ha probado utilizando diferentes colecciones de XForms, las cuales se corresponden con documentos reales utilizados en la pequeña y mediana empresa chilena. XForms[1], un dialecto de XML, es una recomendación candidata de W3C con el objetivo de especificar formularios web, que separa de una manera muy precisa el contenido semántico de los aspectos de presentación. En concreto, la utilización de XForms está siendo habitual en la representación e intercambio de información y transacciones entre empresas.

[1] `http://www.w3.org/MarkUp/Forms`

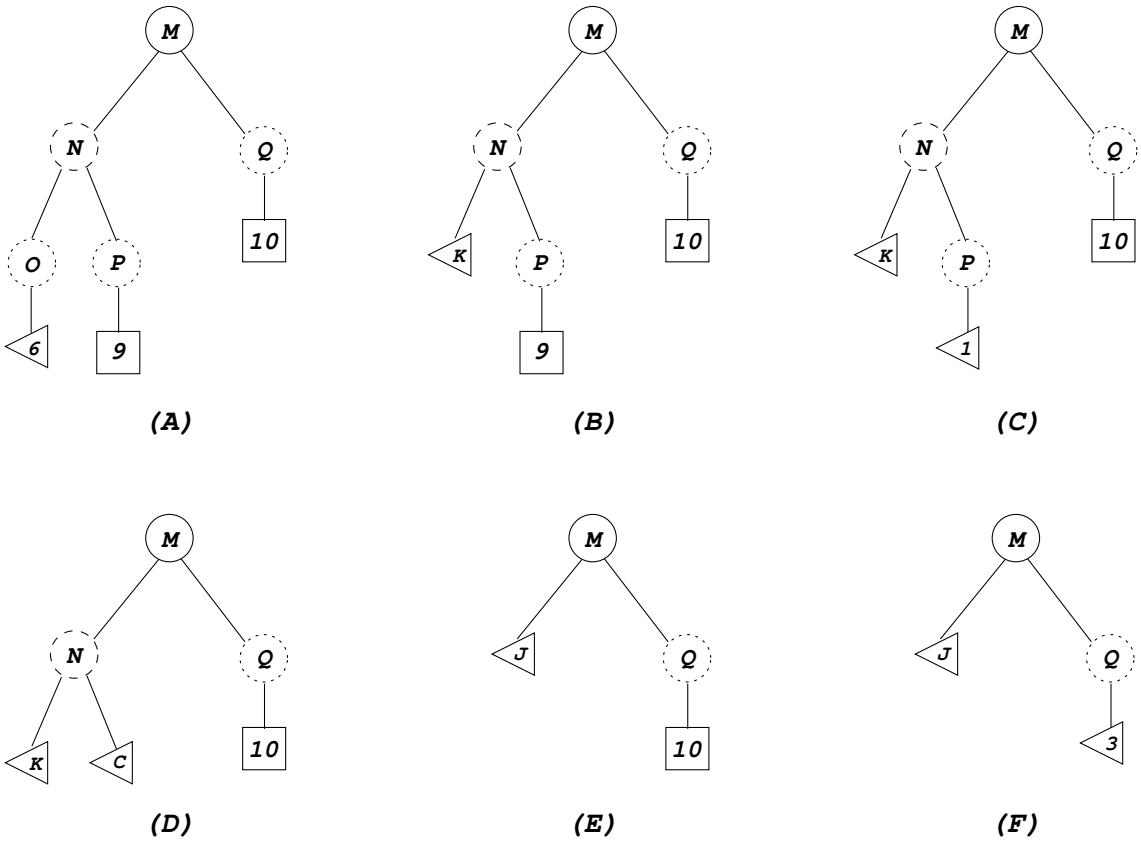


Figura 7.5: Representación paso a paso de las sustituciones que se han efectuado en el tercer documento.

Por cuestiones de privacidad no se han podido utilizar bases de datos de XForms actuales, pero podemos simularlas de una manera muy precisa. Se han obtenido cinco tipos diferentes de formularios (por ejemplo facturas) que tienen varios campos y cada uno tiene un vocabulario controlado (por ejemplo nombres de piezas o fechas) al que se ha tenido acceso. Por lo tanto, se ha generado el contenido de las bases de datos de forma aleatoria, eligiendo el contenido de cada campo de su vocabulario controlado correspondiente. Aunque las bases de datos obtenidas de esta manera son válidas para experimentar con ellas, es necesario destacar que los datos reales posiblemente tengan más redundancia que los datos obtenidos de forma aleatoria y, por lo tanto, nos encontramos ante una "situación pesimista".

A continuación se realiza una breve descripción de los cinco tipos de formularios que se han utilizado.

- XForm de tipo 1: Centralización de remuneraciones. Representa la contabilización de las remuneraciones mensuales, por cantidades totales y con desglose de su composición. Es un documento muy usado.

- XForm de tipo 2: Factura de venta. Es un documento legal chileno.

- XForm de tipo 3: Factura de compra. Es un documento legal chileno, similar al anterior.

- XForm de tipo 4: Orden de trabajo. Es el documento que usan las empresas de proyectos de instalaciones de calefacción para registrar el detalle contable de un trabajo contratado.

- XForm de tipo 5: Cubicación de trabajo. Es el documento que usan en las empresas que construyen letreros y señales bajo demanda para determinar las partes y el costo de un trabajo a desarrollar. Las empresas constructoras usan un documento similar.

Para realizar los experimentos se han utilizado las colecciones de XForms de tipo 4 y 5 por separado y de forma indivisible debido a su pequeño tamaño; por otro lado, se han seleccionado diferentes tamaños de colecciones de XForms de tipo 1, 2 y 3.

Para mejorar la razón de compresión se ha aplicado una codificación de Huffman basada en palabras [Huf52, BSTW86] sobre las colecciones transformadas con LZCS. El texto transformado es una secuencia finita formada por palabras, separadores y etiquetas; en donde una palabra es cualquier secuencia máxima de caracteres alfanuméricos, un separador es cualquier secuencia máxima de caracteres no alfanuméricos y, en este caso, una etiqueta será un código que se ha insertado en el texto para representar el inicio o el final de un elemento estructural o una referencia y que se diferencia del resto del texto por que dicho código se representa de forma habitual entre los símbolos '<' y '>' que actúan como delimitadores. Las palabras, los separadores y las etiquetas anteriormente citadas serán consideradas como símbolos del alfabteto de la fuente en la codificación de Huffman basada en palabras que se utiliza para comprimir el texto transformado.

En todos los experimentos, se ha probado LZCS variando el valor del parámetro l (ver §7.1.1) para comprobar la incidencia del valor de l, que indica cuál será el tamaño mínimo de los bloques de texto para que éstos sean candidatos a ser sustituidos, sobre la razón de compresión.

Cuando $l = 0$ se realizan todas las sustituciones posibles, por el contrario, cuando $l = \infty$ no se sustituye ningún bloque de texto que no esté incluido en un elemento estructural, por lo que se puede decir que en este caso sólo se sutituyen los elementos estructurales. En conclusión, se efectuarán más o menos sustituciones dependiendo del valor que se le asigne al parámetro l y, consecuentemente, la elección de dicho valor influye sobre la razón de compresión pero no sobre la validez de la transformación.

Las tablas 7.1, 7.2 y 7.3 muestran las razones de compresión obtenidas por la transformación LZCS y por la codificación de Huffman basada en palabras sobre la transformación para los diferentes tamaños de las colecciones de los tipos de XForm 1, 2 y 3 respectivamente. Por otro lado, la tabla 7.6 muestra las razones de compresión obtenidas por la transformación LZCS y por la codificación de Huffman basada en palabras sobre la transformación para las colecciones de XForms de tipo 4 y 5. En la primera parte de las tablas se indica la compresión obtenida al realizar únicamente la transformación LZCS mientras que en la segunda parte se muestra la compresión obtenida después de aplicar una codificación de Huffman basada en palabras al texto transformado.

A la vista de los resultados se puede afirmar las peores razones de compresión se han obtenido para $l = \infty$ y la compresión cuando $l = 0$ ha obtenido unos resultados intermedios en la mayoría de los casos llegándose a alcanzar mejoras de la razón de compresión de hasta el 48 % en la colección más grande de XForms de tipo 1. La excepción se encuentra en las colecciones de XForms de tipo 3 mayores de 1 Mbyte en las que se invierten los resultados mejorando el caso $l = 0$ a $l = \infty$ en un 28 % para el tamaño de colección más grande. Esto se debe a las propiedades intrínsecas de las colección y más concretamente a la cantidad y tamaño de bloques de texto que no se encuentran dentro de ningún elemento estructural. No obstante, en la mayoría de las situaciones las razones de compresión obtenidas para los

l / Tamaño	1.019.893	4.717.732	10.228.875	19.671.383	42.162.164
Transformación LZCS					
0	1,5748 %	0,3416 %	0,1575 %	0,0819 %	0,0382 %
4	1,5429 %	0,3346 %	0,1543 %	0,0802 %	0,0374 %
5	1,5422 %	0,3345 %	0,1542 %	0,0802 %	0,0374 %
6	1,5457 %	0,3352 %	0,1546 %	0,0804 %	0,0375 %
7	1,5457 %	0,3352 %	0,1546 %	0,0804 %	0,0375 %
8	1,5457 %	0,3352 %	0,1546 %	0,0804 %	0,0375 %
∞	2,9942 %	0,6602 %	0,3045 %	0,1583 %	0,0738 %
Transformación LZCS + Word-Huffman					
0	0,9428 %	0,2093 %	0,0982 %	0,0516 %	0,0240 %
4	0,8957 %	0,1975 %	0,0934 %	0,0490 %	0,0231 %
5	0,8729 %	0,1935 %	0,0905 %	0,0472 %	0,0224 %
6	0,8642 %	0,1922 %	0,0894 %	0,0471 %	0,0223 %
7	0,8519 %	0,1897 %	0,0881 %	0,0456 %	0,0219 %
8	0,8538 %	0,1888 %	0,0875 %	0,0462 %	0,0215 %
∞	1,1017 %	0,2441 %	0,1125 %	0,0586 %	0,0273 %

Tabla 7.1: Compression ratios for XForms type 1

l / Tamaño	1.013.567	4.763.039	10.016.725	20.448.225	48.236.698
Transformación LZCS					
0	8,5544 %	6,9556 %	6,6463 %	6,4266 %	6,2918 %
4	6,6795 %	5,0548 %	4,7550 %	4,4890 %	4,3131 %
5	6,6723 %	5,0514 %	4,7522 %	4,4866 %	4,3111 %
6	6,6840 %	5,0595 %	4,7598 %	4,4932 %	4,3170 %
7	6,6845 %	5,0596 %	4,7598 %	4,4933 %	4,3170 %
8	6,6847 %	5,0597 %	4,7599 %	4,4933 %	4,3170 %
∞	9,8002 %	7,8051 %	7,4946 %	6,9354 %	6,5136 %
Transformación LZCS + Word-Huffman					
0	2,7170 %	1,4900 %	1,2706 %	1,1216 %	1,0204 %
4	2,5413 %	1,3854 %	1,1779 %	1,0284 %	0,9220 %
5	2,5267 %	1,3814 %	1,1733 %	1,0291 %	0,9247 %
6	2,5206 %	1,3892 %	1,1766 %	1,0318 %	0,9283 %
7	2,5134 %	1,3817 %	1,1739 %	1,0295 %	0,9269 %
8	2,5029 %	1,3810 %	1,1702 %	1,0253 %	0,9259 %
∞	3,3635 %	2,2189 %	2,0187 %	1,8024 %	1,6372 %

Tabla 7.2: Compression ratios for XForms type 2

valores de $l = 0$ y $l = \infty$ son peores que las obtenidas por valores intermedios de l. Los valores intermedios de l proporcionan niveles de compresión similares con variaciones muy pequeñas, mejorando como mínimo la compresión obtenida en un 18 % con respecto a los valores extremos de l en el pero caso posible.

En cualquier caso la compresión que ha obtenido la transformación LZCS de por sí es sorprendentemente buena, sobre todo si se tiene en cuenta que la salida de la transformación sigue siendo un documento de texto. Cuando se aplica una codificación de Huffman basada

l / Tamaño	1.020.706	5.012.738	10.986.407	19.913.073	44.298.164
Transformación LZCS					
0	10,7640 %	9,6690 %	9,5549 %	9,5390 %	9,5540 %
4	7,7738 %	6,5113 %	6,2935 %	6,1853 %	6,0885 %
5	7,7667 %	6,5081 %	6,2911 %	6,1834 %	6,0872 %
6	7,7789 %	6,5159 %	6,2974 %	6,1886 %	6,0910 %
7	7,7792 %	6,5159 %	6,2974 %	6,1886 %	6,0910 %
8	7,7798 %	6,5161 %	6,2975 %	6,1886 %	6,0910 %
∞	10,9101 %	8,9237 %	8,3035 %	7,8511 %	7,3380 %
Transformación LZCS + Word-Huffman					
0	2,4900 %	1,5589 %	1,4367 %	1,3990 %	1,3868 %
4	2,3682 %	1,4960 %	1,3803 %	1,3454 %	1,3294 %
5	2,3245 %	1,4918 %	1,3809 %	1,3472 %	1,3293 %
6	2,3261 %	1,4885 %	1,3850 %	1,3514 %	1,3344 %
7	2,3228 %	1,4858 %	1,3843 %	1,3508 %	1,3339 %
8	2,3114 %	1,4829 %	1,3817 %	1,3500 %	1,3337 %
∞	3,2297 %	2,2411 %	2,0258 %	1,8950 %	1,7722 %

Tabla 7.3: Compression ratios for XForms type 3

l / Tamaño	T.4 (7.536.091)	T.5 (6.017.393)
Transformación LZCS		
0	5,3831 %	4,9790 %
4	4,8863 %	3,6246 %
5	4,8861 %	3,6245 %
6	4,9119 %	3,6245 %
7	4,9121 %	3,6245 %
8	4,9124 %	3,6245 %
∞	9,2891 %	5,0859 %
Transformación LZCS + Word-Huffman		
0	0,8928 %	0,9394 %
4	0,8937 %	0,8474 %
5	0,8967 %	0,8463 %
6	0,9042 %	0,8393 %
7	0,8970 %	0,8393 %
8	0,8930 %	0,8413 %
∞	1,9535 %	1,2566 %

Figura 7.6: Compression ratios for XForms types 4 and 5

en palabras sobre el texto transformado se obtiene una compresión todavía mejor llegando a reducir el texto transformado por LZCS de un 20 % a un 60 % de su tamaño.

En la implementación propuesta se comentó que para para comprobar la existencia previa del bloque de texto actual se comprobaba la existencia de su firma digital en el conjunto de firmas digitales de los bloques anteriores y origiales, no obstante cuando existe una coincidencia se compraran los bloques de textos equivalentes para evitar una posible *falsa coincidencia* ya que los algoritmos de firma digital no aseguran la unicidad de la

firma. A raiz de este comentario se puede apuntar que en todos los experimentos realizados siempre que se ha encontrado un bloque equivalente nunca se ha producido una falsa coincidencia.

Por último, se comparará LZCS contra otros sistemas de compresión que ni permiten navegar ni acceder de forma aleatoria sobre los textos comprimidos. Se han elegido dos tipos de sistemas de compresión para comparar: unos consideran la estructura a la hora de comprimir (como *XMill* y *XMLPPM* descritos brevemente en §1.1) y los otros son estándar. La mayoría de los sistemas estándar están basados en los esquemas LZ clásicos. Los sistemas estándar que se han elegido para realizar la comparación contra LZCS son aquellos que generalmente se encuentran disponibles que cualquier distribución de Linux, son los siguientes: (1) *compress* de UNIX, que implementa el algoritmo LZW; (2) *zip* y (3) *gzip*, que utilizan el algoritmo LZ77 junto a una variante del algoritmo de Huffman; (4) *bzip2*, que aplica una codificación de Huffman a un bloque de texto previamente transformado mediante la transformación de Burrows-Wheeler.

En general, la compresión obtenida por *bzip2* es considerablemente mejor que la que obtienen los compresores más convecionales basados en LZ77/LZ78 y se aproxima a las obtenidas por la famila de compresores estadísticos PPM.

Se han comprimido todas las colecciones con los sistemas descritos con anterioridad, las tablas 7.4, 7.5 y 7.6 muestran las razones de compresión obtenidas en cada caso para las colecciones de XForms de tipo 1, 2 y 3 respectivamente y sus correspondientes representaciones gráficas se pueden observar en las figuras 7.7 ,7.8 y 7.9. Por otro lado, las razones de compresión obtenidas para las colecciones de XForms de tipos 4 y 5 se muestran en la tabla 7.10. Tanto en las tablas como en las gráficas LZCS utiliza el valor de l que obtiene la mejor compresión en cada caso.

Method / Size	1.019.893	4.717.732	10.228.875	19.671.383	42.162.164
compress	9,9360 %	7,2605 %	7,5636 %	7,6195 %	7,6303 %
zip	1,3928 %	1,3207 %	1,3084 %	1,2981 %	1,2922 %
gzip	1,3808 %	1,3181 %	1,3072 %	1,2975 %	1,2919 %
bzip2	0,6558 %	0,4674 %	0,4404 %	0,4226 %	0,4214 %
XMill	0,8232 %	0,5642 %	0,5054 %	0,5079 %	0,5046 %
XMLPPM	0,8402 %	0,7070 %	0,6871 %	0,6783 %	0,6741 %
Word Huffman	10,1260 %	9,7787 %	9,7270 %	9,7058 %	9,6935 %
lzcs ($l = 8$)	0,8538 %	0,1888 %	0,0875 %	0,0462 %	0,0215 %

Tabla 7.4: Compression ratios for XForms type 1

Entre los compresores estándar, *compress* y Huffman orientado a palabra obtienen las peores razones de compresión, las cuales no son competitivas en este experimento. A éstos les siguen *zip* y *gzip* ambos con niveles de compresión muy similares. El mejor compresor dentro de la categoria de los compresores estándar es, sin lugar a dudas, *bzip2* que, en general, sigue siendo inferior a LZCS aunque en algunos casos por un pequeño margen. El motivo de estos resultados (relativamente malos) de los compresores estándar se debe a que estas técnicas no tienen en consideración la estructura de los documentos en el proceso de compresión, hecho que la LZCS obtiene una ventaja significativa. Además, es necesario recordar que LZCS permite navegar y acceder de manera aleatoria sobre el texto comprido, lo que no es nada sencillo con los compresores estándar.

En el caso de los métodos que consideran la estructura de los documentos a la hora de comprimir se puede comentar que LZCS es significativamente mejor que *XMill* en todas

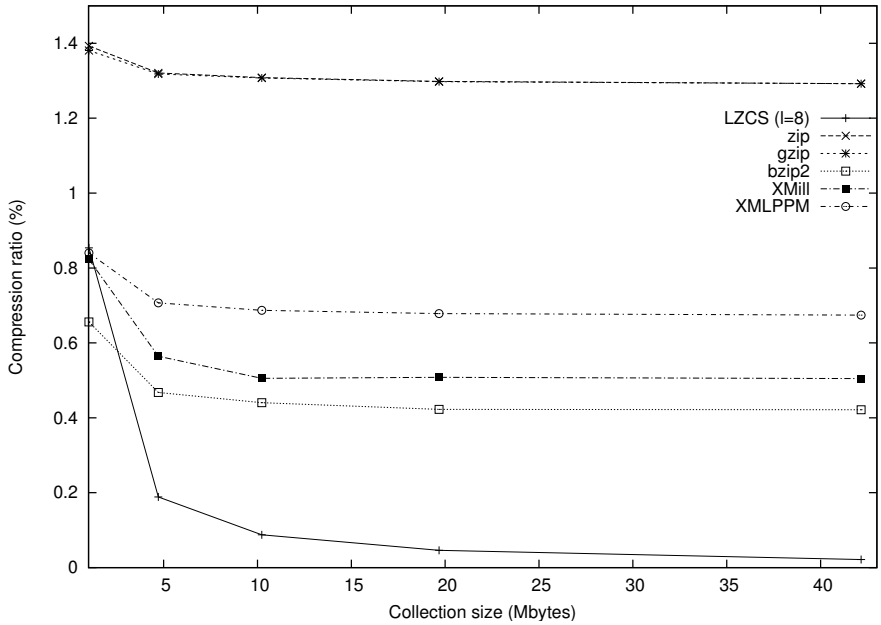


Figura 7.7: Comparison between LZCS and others, for XForms type 1.

Method / Size	1.013.567	4.763.039	10.016.725	20.448.225	48.236.698
compress	10,3540 %	9,7158 %	9,5922 %	9,2859 %	9,0033 %
zip	5,1950 %	5,2894 %	5,2216 %	5,1863 %	4,9180 %
gzip	5,1829 %	5,2868 %	5,2204 %	5,1857 %	4,9178 %
bzip2	1,4485 %	1,1534 %	1,1264 %	1,0885 %	1,0784 %
XMill	2,9952 %	2,2301 %	2,1969 %	2,2191 %	2,2626 %
XMLPPM	0,9415 %	0,6906 %	0,6474 %	0,6115 %	0,5891 %
Word Huffman	13,9091 %	13,1410 %	13,0332 %	12,8100 %	12,6466 %
lzcs ($l = 4$)	2,5413 %	1,3854 %	1,1779 %	1,0284 %	0,9220 %

Tabla 7.5: Compression ratios for XForms type 2

la colecciones, obteniendo textos comprimidos un 5 % más pequeños en el peor caso y veinticinco veces más pequeños en el mejor. Por otro lado, *XMLPPM* obtiene la mejor compresión en la mayoría de los casos, con la notable excepción de las colecciones de XForms de tipo 1 en las que LZCS está a mucha distancia de ser superado. El principal problema que tiene *XMLPPM* es que es adaptativo y, por consiguiente, no es adecuado para navegar o acceder de manera aleatoria sobre los documentos comprimidos.

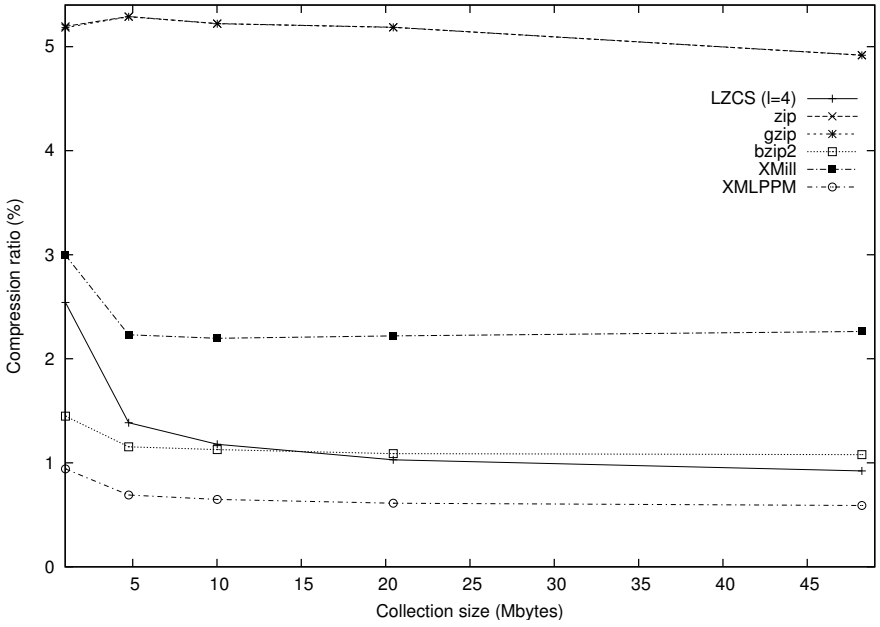


Figura 7.8: Comparison between LZCS and others, for XForms type 2.

Method / Size	1.020.706	5.012.738	10.986.407	19.913.073	44.298.164
compress	9,5717 %	8,5933 %	8,1583 %	7,8351 %	7,4073 %
zip	4,4958 %	5,0429 %	4,8860 %	4,5526 %	4,0591 %
gzip	4,4837 %	5,0404 %	4,8849 %	4,5519 %	4,0588 %
bzip2	1,2513 %	1,0539 %	1,0639 %	1,0745 %	1,0769 %
XMill	1,7454 %	1,1772 %	1,2250 %	1,3795 %	1,6285 %
XMLPPM	0,8841 %	0,7113 %	0,6885 %	0,6796 %	0,6756 %
Word Huffman	12,7213 %	12,0866 %	11,8947 %	11,7448 %	11,5500 %
lzcs ($l = 5$)	2,3245 %	1,4918 %	1,3809 %	1,3472 %	1,3293 %

Tabla 7.6: Compression ratios for XForms type 3

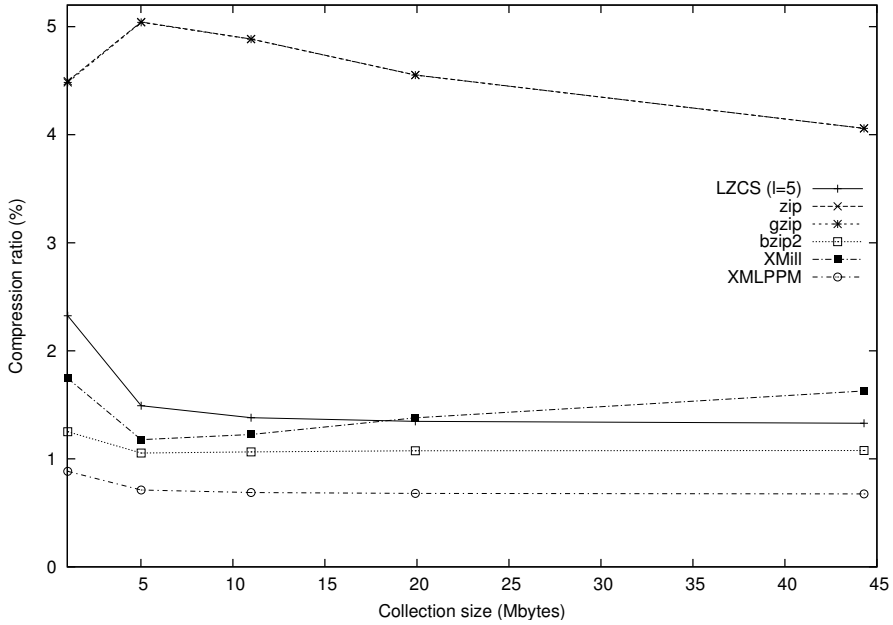


Figura 7.9: Comparison between LZCS and others, for XForms type 3.

Method / Size	T.4 (7.536.091)	T.5 (6.017.393)
compress	10,3003 %	10,3958 %
zip	2,1055 %	4,4350 %
gzip	2,1040 %	4,4331 %
bzip2	0,9527 %	0,8430 %
XMill	0,9426 %	0,9242 %
XMLPPM	0,7124 %	0,5530 %
Word Huffman	13,9941 %	12,4417 %
lzcs ($l = 6$)	0,9042 %	0,8393 %
lzcs ($l = 8$)	0,8930 %	0,8413 %

Figura 7.10: Compression ratios for XForms types 4 and 5

Capítulo 8

Indexación y recuperación por contenido y estructura

La recuperación de información debe representar, almacenar, organizar y acceder los elementos de información. La representación y organización deberían ser suficientes para permitir al usuario acceder fácilmente a la información que este precise, pero la especificación de las necesidades de información de los usuarios no es un problema trivial. Plantear una necesidad de información de manera que el sistema de recuperación pueda captar qué necesita el usuario no es una tarea sencilla. Generalmente, los sistemas de recuperación de información capturan la necesidad del usuario a través de una consulta, que estará formada por un conjunto de términos de búsqueda.

La recuperación de datos implica determinar qué elementos de información contienen los términos de búsqueda planteados por el usuario en su consulta entre un conjunto de partida; en un contexto de recuperación de información esto no es suficiente pues el usuario está interesado en obtener aquellos elementos que tratan acerca de un tema, es decir, elementos de información relacionados con un determinado concepto. La principal diferencia radica en que en la recuperación de datos el usuario especifica de forma precisa una serie de características que se deben cumplir, mientras que en la recuperación de información no existe esa precisión y como consecuencia se pueden recuperar elementos que pueden no estar relacionados con lo que se desea buscar (lo cual sería un error crítico en recuperación de datos). El motivo de esta "permisibilidad" en lo que respecta a la información recuperada radica en que los sistemas de recuperación de información son, habitualmente, sistemas de información textual, en los que la estructura no está siempre bien definida y la ambigüedad es un habitual compañero de viaje, siempre unida al lenguaje natural.

Los sistemas de recuperación de datos cubren las necesidades del usuario haciendo uso de una base de datos, pero para realizar la denominada recuperación de información serán necesarios sistemas que hagan una interpretación del contenido de los elementos de información (documentos, apartados, etc.) sobre los que realizar la consulta, asimismo será preciso realizar una interpretación del contenido de la consulta planteada por el usuario para establecer qué elementos son *relevantes* a dicha consulta. El concepto de relevancia es fundamental en el mundo de la recuperación de información.

En los sistemas de recuperación de información existen dos etapas cláramente diferenciadas y necesarias para llevar a cabo con éxito el proceso de recuperación: la indexación y la consulta.

- La indexación procesa los documentos para obtener un índice que considere la fre-

cuencia de aparición de los términos de indexación. La indexación se realiza "off–line", por lo que el tamaño del índice construido y la representatividad del mismo (el modo en que representa los documentos originales) se suelen considerar como parámetros de calidad [BYRN99, WMB99].

- La consulta (generalmente un proceso "on–line") recupera un conjunto de documentos ordenado por relevancia (importacia asignada por el sistema) a partir de una necesidad de información expresada a través un interfaz de consulta en un determinado lenguaje. Antes de aplicar la consulta al índice se debe transformar de la misma manera en la que se transformaron los documentos durante el proceso de indexación para mantener la homogeneidad en el proceso. Como resultado se obtendrán un conjunto de documentos recuperados que le serán presentados al usuario de la manera más adecuada posible. El proceso puede acabar aquí, de modo que el usuario selecciona de los documentos recuperados aquellos que le resultan interesantes, o bien se refina la consulta con la información que aporta el usuario sobre la pertinencia de los resultados.

El uso de sistemas de recuperación de información y bibliotecas digitales está aumentando día a día, y cada vez son más los sistemas que deben manejar documentos semiestructurados debido a la amplia aceptación que han tenido los lenguajes de marcado, sobre todo el estándar XML. Las dos etapas anteriormente mencionadas deberán considerar la estructura de los documentos de forma adecuada para que de esta manera se puedan recuperar los documentos o las partes de los mismos más relevantes a las consultas planteadas.

8.1. Direccionando la estructura

Recordemos que un índice invertido se compone de un vector que contiene todas las palabras distintas susceptibles de ser buscadas en la colección en orden lexicográfico (a este vector se le denomina *vocabulario* y a cada palabra que lo compone se denomina *término*) y, para cada término del vocabulario, se almacena una lista en la que figuran todas las ubicaciones en la colección del término al que corresponde la lista (por eso se denomina *lista de ocurrencias*). La *granularidad* de un índice depende de a qué hagan referencia dichas ubicaciones, si indican las posiciones del texto en las que aparecen los términos referenciados, se estará hablando de un índice invertido con direccionamiento a caracteres; por otro lado, si éstas indican los documentos en los que aparecen los términos referenciados, se estará hablando de un índice invertido con direccionamiento a documentos. También existe el direccionamiento a bloques en el que las ubicaciones de las listas de ocurrencias hacen referencia a bloques de texto que generalmente son de un tamaño considerable y engloban varios documentos de la colección, por lo que se obtiene un índice de pequeño tamaño.

Dependiendo de la granularidad del índice se favorecerán determinadas propiedades del mismo, por ejemplo, un índice que direccione caracteres tendrá mayor tamaño que otro que direccione documentos sobre la misma fuente de datos. No obstante, el primero permitirá localizar el lugar exacto en dónde se encuentra una palabra y facilitará las búsquedas por proximidad pero en ciertas ocasiones (por ejemplo cuando un archivo físico contenga varios documentos lógicos) será necesario procesar el texto para obtener el documento en el que se encuentra el término buscado, hecho que no ocurre con el segundo tipo de índice que devuelve directamente el documento en el que aparece dicho término pero, por el contrario, si se desea obtener la posición exacta del término o realizar busquedas por proximidad es necesario procesar el texto del documento.

Ahora bien, cuando con estos tipos de índices se intentan resolver consultas por contenido y estructura siempre será necesario procesar el texto de los documentos para saber en qué elementos de estructura se encuentra un término buscado y, de esta manera, determinar si dicho término es relevante o no a la consulta. El hecho de procesar en el texto (generalemente, para realizar búsquedas sobre gran parte del texto) de los documentos implica un incremento en los tiempos de respuesta que puede no ser aceptable en la mayoría de las situaciones prácticas.

En esta sección se propone un esquema de indexación que permite realizar búsquedas de términos dentro de elementos estructurales de forma eficiente, eliminando la necesidad de procesar el texto para comprobar dónde se encuentran dichos términos. La idea es introducir un nivel más de granularidad haciendo que los elementos de las listas de ocurrencias del índice invertido hagan referencia a elementos estructurales. Inicialmente se puede pensar en que las listas invertidas contengan las posiciones dónde comienza el elemento de estructura que contiene el término en cuestión y a continuación se accede al texto para obtener el nombre de la etiqueta. Esta solución es un poco mejor que utilizar un índice con direccionamiento a caracteres o documentos pero presenta dos serios inconvenientes: en primer lugar es necesario acceder al texto de la colección tantas veces como ocurrencias de la palabra con lo que el tiempo de respuesta se consigue mejorar muy levemente; y en segundo lugar, con este planteamiento es necesario procesar gran parte del texto a la hora de resolver las relaciones de inclusión entre los elementos estructurales dado que se tiene que buscar la etiqueta (o etiquetas) que incluyen al elemento estructual dado.

La manera de direccionar la estructura debe permitir realizar búsquedas por contenido y estructura de forma eficiente y además debe facilitar la resolución de las relaciones de inclusión entre elementos estructurales. Para ello se propone un sistema que se apoya en una estructura de control complementaria denominada *tabla de control de estructura* y que se utiliza para ayudar a resolver las relaciones de inclusión sin necesidad de procesar el texto. Cada elemento estructural presente en la colección deberá tener una y sólo una entrada en la tabla de control de estructura. Una entrada será un registro (elemento o fila) de dicha tabla (vector) y en él se almacenará toda la información que se vaya a necesitar de cada elemento estructural.

Para direccionar la estructura sólo hay que escribir el número de registro en el campo de referencia de la lista de ocurrencias correspondiente al elemento estructural en el que aparece la palabra a la que pertenece dicha lista. Así, cuando se desee recuperar un término para resolver una consulta por contenido y estructura, se recuperará la lista de ocurrencias correspondiente a su entrada en el índice invertido y, para cada elemento de la lista, se accede al registro indicado para identificar en qué elemento de estructura está contenido. El tiempo de acceso a la tabla de control es $O(1)$ ya que se trata de un vector de registros. Dependiendo de las necesidades del sistema de recuperación y si fuera necesario las listas de ocurrencias también pueden incluir, junto al número de registro de la tabla de control, el número de veces que aparece el término dentro de cada elemento estructural.

Además, este planteamiento también permite resolver consultas acerca de la estructura de los documentos siempre y cuando los registros que constituyen la tabla de control contengan suficiente información. Por ejemplo, se pueden resolver consultas del estilo a "¿qué documentos tienen título?". Para resolver este tipo de consultas es necesario procesar la información almacenada en todos los registros que componen la tabla de control de estructura. Generalmente este proceso se llevará a cabo en un tiempo $O(n)$, lineal respecto al número total de registros almacenados en la tabla de control siempre y cuando la información almacenada en dichos registros sea suficiente para resolver la consulta. Esta información puede variar de una implementación a otra y dependerá de las funcionalidades

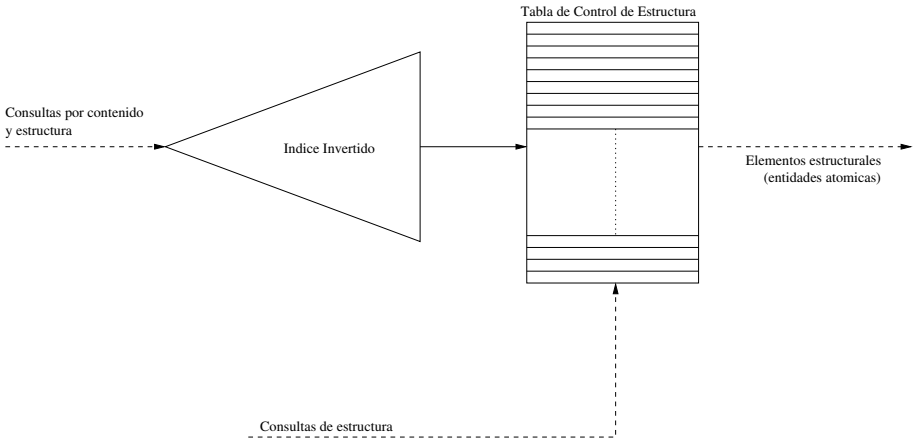


Figura 8.1: Esquema de consulta por contenido y estructura usando el esquema de indexación propuesto.

que el sistema de recuperación proporcione sobre la semántica de la estructura. En la figura 8.1 se muestra gráficamente la forma de resolver las consultas por contenido y estructura utilizando el esquema de indexación propuesto.

8.1.1. Descripción del índice

Habitualmente una colección de documentos semiestructurados estará formada por varios archivos y cada uno de los cuales contendrá al menos un documento lógico. Por otro lado, es habitual que todos los documentos semiestructurados que forman una colección se ajusten a una o varias DTD[1] y el número de etiquetas estructurales que lo componen suele ser un número finito y, generalmente, pequeño. Además, para conocer qué elementos están contenidos dentro de otro basta con saber qué región del texto cubre cada uno y para ello basta con saber en que posiciones comienza y finaliza cada elemento estructural.

Fichero	Estructura	Posición Inicio	Posición Final

Figura 8.2: Registro elemental de la tabla de control de estructura

En la figura 8.2 se muestra la organización básica de un registro elemental candidato a formar la tabla de control de estructura. En cada campo de dicho registro se representará para cada elemento de estructura que forma la colección el fichero en el que aparece, el tipo de elemento y las posiciones en las que comienza y termina dicho fichero.

No obstante, el registro presentado en la figura 8.2 es una de las versiones más elementales del mismo y dependiendo de las necesidades de recuperación, tal y como se ha comentado con anterioridad, a este registro se le pueden añadir más campos que permitan mejorar los tiempos de respuesta del sistema de recuperación, como por ejemplo, una autorreferencia que apunte al elemento estructural padre del actual o al elemento estructural que representa al documento que contiene al actual.

Hasta ahora se ha supuesto que todos los documentos que constituyen la colección se ajustan a la misma DTD, no obstante no existe ningún inconveniente indexar una colección

[1] Definición de Tipo de Documento

en las que existan documentos con DTD's diferentes. En este caso, es necesario remarcar que el hecho de que en estas colecciones de documentos puede haber elementos estructurales distintos (con diferente nombre) pero semánticamente representan el mismo concepto, este hecho constituye un verdadero problema para el sistema de recuperación y, por ello, un adecuado seguimiento y mantenimiento de conjuntos de elementos estructurales sinónimos mejorará la precisión y la efectividad del sistema.

8.1.2. Ejemplo de indexación

Para ilustrar la técnica de indexación mediante un ejemplo supóngase que una colección de documentos está formada por los ficheros que se muestran en la figura 8.3. Dichos documentos se ajustan a una DTD que sirve para representar libros muy sencillos, en la cual toda la información que contiene un libro está delimitada entre las etiquetas `<DOC>` y `</DOC>`, un libro tendrá un título y uno o varios autores (delimitados entre las etiquetas `<TITULO>...</TITULO>` y `<AUTOR>...</AUTOR>` repectivamente) y además estará formado por secciones (etiquetas `<SEC>...</SEC>` las cuales pueden contener subsecciones (etiquetas `<SUBSEC>...</SUBSEC>`).

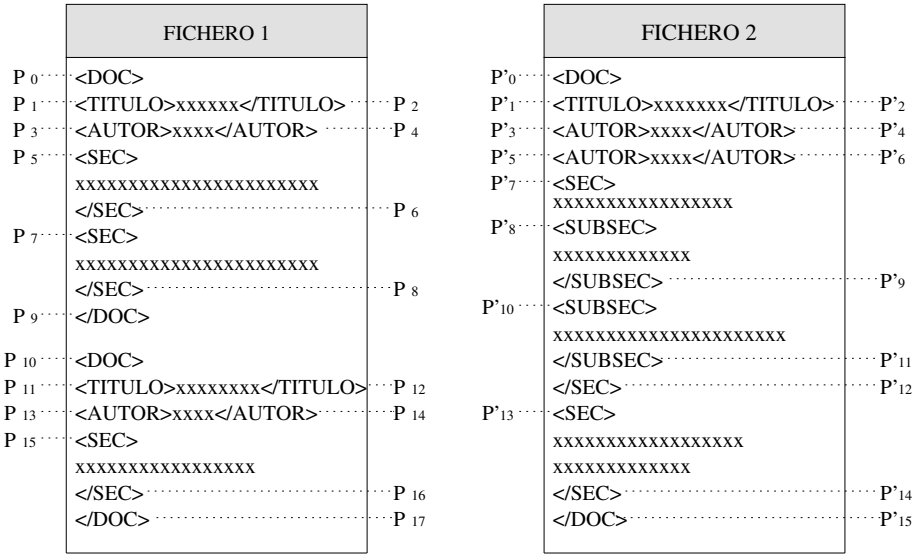


Figura 8.3: Ficheros de ejemplo

A efectos prácticos y como una manera de disminuir el espacio que ocupa la tabla de control de estructura se pueden representar mediante un identificador numérico tanto los ficheros que forman la colección como el tipo de elemento estructural en dicha tabla. Si se aplican las equivalencias entre los nombres que se muestran en la figura 8.4 se obtiene la tabla de control de estructura que se ilustra en la figura 8.5.

Como ya se ha comentado, en las listas de ocurrencias del índice se almacena el número de registro de la tabla de control que se corresponde al lugar en el que aparece la ocurrencia del término, o lo que es lo mismo, se almacena la identificación de una determinada estructura en un fichero en concreto. Junto al número de registro también se puede almacenar el número de veces que aparece dicho término dentro de ese elemento estructural, esto puede ser útil para generar un ranking de los documentos recuperados o los mejores puntos de

Fichero	Id.
Fichero 1	1
Fichero 2	2

Estructura	Id.
DOC	1
TITULO	2
AUTOR	3
SEC	4
SUBSEC	5

Figura 8.4: Equivalencias para el ejemplo

	Fichero	Estructura	Inicio	Fin
Reg.1	1	1	P_0	P_9
Reg.2	1	2	P_1	P_2
Reg.3	1	3	P_3	P_4
Reg.4	1	4	P_5	P_6
Reg.5	1	4	P_7	P_8
Reg.6	1	1	P_{10}	P_{17}
Reg.7	1	2	P_{11}	P_{12}
Reg.8	1	3	P_{13}	P_{14}
Reg.9	1	4	P_{15}	P_{16}
Reg.10	2	1	P'_0	P'_{15}
Reg.11	2	2	P'_1	P'_2
Reg.12	2	3	P'_3	P'_4
Reg.13	2	3	P'_5	P'_6
Reg.14	2	4	P'_7	P'_{12}
Reg.15	2	5	P'_8	P'_9
Reg.16	2	5	P'_{10}	P'_{11}
Reg.17	2	4	P'_{13}	P'_{14}

Figura 8.5: TCE para el ejemplo

entrada[2] a un documento como respuesta a una consulta de usuario.

Considerando el ejemplo anterior y suponiendo que un término τ cualquiera aparece una vez en el título del primer documento del primer fichero, tres veces en la segunda sección de ese mismo documento y cuatro veces en la primera subsección del primer tema del documento que se encuentra almacenado en el segundo fichero, la lista de ocurrencias para ese término se puede expresar como

$$\tau \rightarrow [(2,1),(5,3),(15,4)]$$

Cuando se desea buscar una palabra en un elemento de estructura, se recorre secuencialmente la lista de ocurrencias de la palabra. Para cada elemento de la lista se consulta el registro de la TCE indicado y se compara el campo de estructura con el identificador de la estructura que queremos buscar.

La TCE también mantiene información implícita sobre la jerarquía de la estructura. Se puede saber qué elementos contienen a otros comparando las posiciones de inicio y final de cada estructura almacenadas en la tabla. Esto permite resolver consultas no sólo por contenido sino también sobre la estructura, por ejemplo permite resolver preguntas del tipo"¿qué documentos tienen título?".

[2]Mejor punto de entrada (BEP, "best entry point"): lugar del documento que se considera que comienza la parte relevante del mismo de acuerdo a la consulta realizada.

8.1.3. Compresión del índice

La técnica de indexación comentada con anterioridad permite recuperar los elementos estructurales que respondan a una consulta de usuario de manera eficiente. No obstante, el hecho de disponer del índice para resolver las consultas implica un coste de almacenamiento adicional y éste puede convertirse en un inconveniente en algunas situaciones, como por ejemplo en sistemas de recuperación de información residentes en CD-ROM.

Para disminuir el espacio ocupado por el índice se pueden aplicar las técnicas comentadas en §4.3, no obstante será preciso codificar, empleando alguna técnica de codificación de números enteros, tanto la lista de ocurrencias como la tabla de control de estructura. La compresión del índice se realiza de la misma forma que cualquier otro tipo de índice invertido, para ello se separa el vocabulario, que generalmente no se suele codificar para facilitar el acceso a los términos, de las listas de ocurrencias y junto con cada término del vocabulario aparece un apuntador que señala la posición en la que comienza la lista de ocurrencias correspondiente. Por otro lado, las listas de ocurrencias se codifican y se escriben secuencialmente bien a continuación del vocabulario o bien en un fichero aparte. Según se ha comentado con anterioridad, cada elemento que compone las listas de ocurrencias del índice está formado por parejas de números enteros: uno indica elemento estructural en el que aparece el término mediante una referencia a la tabla de control de estrucutra, y el otro indica cuántas veces aparece.

Como las listas de ocurrencias se encuentran ordenadas de acuerdo con la referencia a la tabla de control se sugiere aplicar una codificación por diferencias[3] seguida de una codificación Elias–δ[4] que obtiene buenos resultados al codificar magnitudes de tamaño medio y grande. Por otro lado, aunque el número de ocurrencias del término no suele ser de una magnitud muy grande también se codificará usando Elias–δ pero no una codificación por diferencias dado que no están ordenados.

De la misma manera, se podría comprimir la tabla de control de estructura empleando técnicas de codificación de enteros. En dicha tabla aparecen magnitudes de todo tipo y por eso se sugiere codificarla empleando Elias–δ de nuevo. Ahora bien, los registros que forman la tabla no son accesibles de forma aleatoria una vez que se ha realizado la codificación; no obstante la información contenida en la tabla de control de estructura comprimida es accesible añadiendo una pequeña sobrecarga al esquema. Se plantean dos posibilidades a la hora de hacer accesible dicha información:

1. Hacer que las referencias en las listas de ocurrencias contengan la posición (en número de bits) en la que comienza el registro que mantiene la información de la estructura correspondiente en lugar del número de registro en cuestión.

2. Mantener las referencias de las listas de ocurrencias tal y como se han definido y se accede a la información del elemento estructural mediante el número de registro pero previamente se debe acceder a un vector, que tendrá tantos elementos como entradas (registros) en la tabla de control y cada elemento de dicho vector contendrá la posición (en número de bits) en la que se encuentra la entrada respectiva en la tabla de control y una vez obtenida dicha posición se accede a la información en la tabla de control de estructura comprimida.

Para comprimir la tabla de control de estructura se sugiere la segunda opción pues es previsible que conlleve una menor sobrecarga debido a que en el primer caso las diferencias entre dos elementos consecutivos tendrán magnitudes mayores y consecuentemente la

[3] §4.3.1 en la página 58
[4] §4.3.3 en la página 59

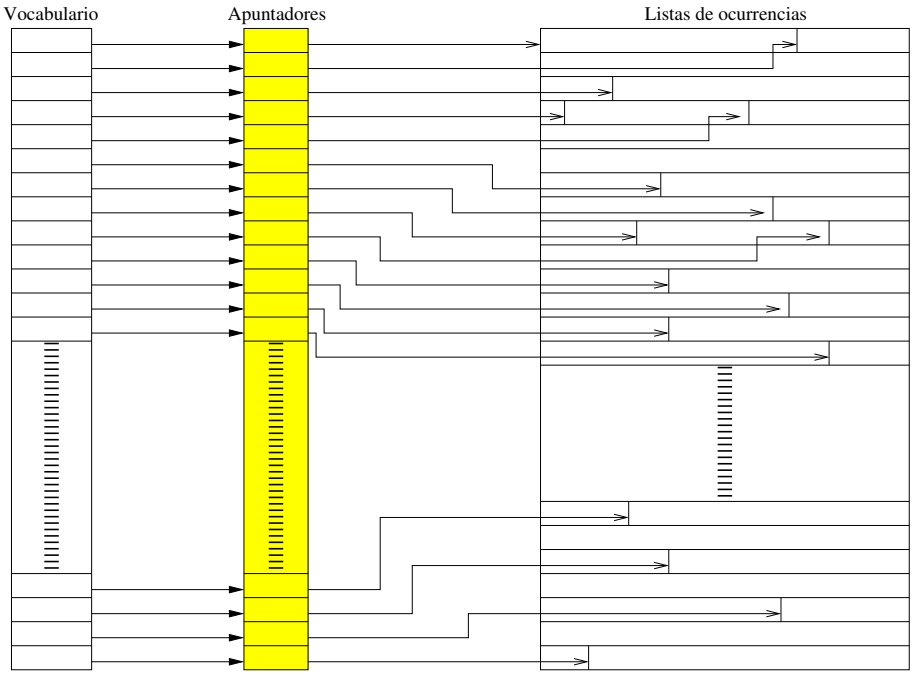


Figura 8.6: Esquema de compresión de la tabla de control.

codificación resultante ocupará más espacio que en el segundo caso, ya que la sobrecarga introducida es proporcional al número total de términos mientras que en el segundo caso es proporcional al número total de elementos estructurales de la colección. El esquema elegido está representado gráficamente en la figura 8.6.

8.2. Álgebra de regiones para documentos con estructura

En los apartados anteriores se ha descrito una técnica de indexación sobre documentos semiestructurados que permite resolver consultas por contenido y estructura. El índice permite recuperar la lista de elementos estructurales que contengan un término de la consulta. También proporciona información acerca de la localización de cada elemento en su correspondiente documento, que en la versión descrita con anterioridad se lleva a cabo mediante los campos que contienen las posiciones de inicio y fin del elemento estructural. Esta información es suficiente para resolver las consultas más sencillas, pero se muestra insuficiente a la hora de resolver consultas más elaboradas. Para ello se ha definido un álgebra[5] que permite manipular elementos estructurales en el dominio de los mismos.

En la tabla de control de estructura se encuentran representados todos los nodos que representan la estructura de los documentos y que son candidatos a ser recuperados a través de una consulta de usuario. Además, la tabla de control de estructura representa de forma implícita la estructura arbórea que simboliza la jerarquía de cada documento que forma la colección. Una propiedad colateral que está presente en dicha tabla es que cuando se efectúa un recorrido secuencial en orden creciente de los elementos representados en ella, se realiza un recorrido primero en profundidad del árbol de estructura de cada documento según su orden de indexación.

[5]Un álgebra es un marco de trabajo formal destinado al tratamiento de la información basado en operadores y valores de dominio.

Las operaciones definidas en el álgebra están encaminadas a resolver los requisitos de información que necesita un sistema de recuperación de información que manipule documentos con estructura, requisitos relacionados con la relación jerárquica de un elemento en concreto respecto a los demás. La propuesta tiene en cuenta la estructura de la tabla de control definida en §8.1 pero, en este caso y por cuestiones de simplicidad, se considerará que todos los documentos se encuentran en un mismo fichero físico[6]. En primer lugar se definirá de formalmente un elemento estructural y la manera de identificarlo y a continuación se detallarán las relaciones y operaciones entre ellos. Sea S un elemento estructural de tipo s presente en un documento almacenado entre las posiciones Λ_S y Ω_S del fichero.

Se define $\mathcal{T}$ como el conjunto de todos los elementos estructurales incluidos en la tabla de control y el número total de elementos de dicha tabla viene dado por $|\mathcal{T}|$. Para dos elementos estructurales cualesquiera, $S \in \mathcal{T}$ y $S' \in \mathcal{T}$, en ningún caso puede suceder que sus correspondientes posiciones de inicio o final sean iguales, y además siempre se debe cumplir que $\Omega_S > \Lambda_S \; \forall \; S \in \mathcal{T}$.

Definición 8.1 (Identificación inequívoca de elementos) *La identificación inequívoca del elemento S en la colección de documentos se puede realizar mediante su tipo s (nombre del elemento estructural) y sus posiciones inicial y final, Λ_S y Ω_S, en el fichero:*

$$S = \{s, \Lambda_S, \Omega_S\} \tag{8.1}$$

Cada elemento de estructura susceptible de ser manejado se incluirá en la tabla de control de estructura y se le asignará un identificador único, I_S, que se corresponderá con la posición (fila) que ocupa dentro de la tabla de control, comenzando la enumeración desde cero.

Definición 8.2 (Relaciones de inclusión entre elementos) *Sea $S \in \mathcal{T}$ y $S' \in \mathcal{T}$ elementos estructurales. S estará contenido en S', $S \lhd S'$, si y sólo si la posición de inicio de S es posterior a la de S' y la posición final de S es anterior a la de S':*

$$S \lhd S' \iff \Lambda_S > \Lambda_{S'} \; \wedge \; \Omega_S < \Omega_{S'} \tag{8.2}$$

Análogamente, S contiene a S', y se denotará mediante $S \rhd S'$, si y sólo si la posición de inicio de S es anterior a la de S' y la posición final de S es posterior a la de S':

$$S \rhd S' \iff \Lambda_S < \Lambda_{S'} \; \wedge \; \Omega_S > \Omega_{S'} \tag{8.3}$$

Una propiedad adicional relacionada con la definición anterior es que si tenemos dos elementos $S, S' \in \mathcal{T}$ y se cumple que $S \lhd S'$, entonces se deduce de manera inmediata que $S' \rhd S$ y viceversa.

Definición 8.3 (Conjuntos de inclusión) *Sea $S \in \mathcal{T}$ un elemento estructural. $\mathcal{E}_S$ es el conjunto de todos los elementos estructurales contenidos en S y está definido como:*

$$\mathcal{E}_S = \{S' \in \mathcal{T} \; \bullet \; S' \lhd S\} \tag{8.4}$$

$\mathcal{P}_S$ es el conjunto de todos los elementos estructurales que contienen a S y se define como:

$$\mathcal{P}_S = \{S' \in \mathcal{T} \; \bullet \; S' \rhd S\} \tag{8.5}$$

[6]La extrapolación de las definiciones cuando se dispone de varios ficheros físicos es simple e inmediata.

A los conjuntos $\mathcal{E}_S$ y $\mathcal{P}_S$ también se les denominará *conjunto de descendientes* y *conjunto de ascendientes* de S respectivamente. Dada la relación de inclusión todos los elementos contenidos en el conjunto $\mathcal{E}_S$ se relacionan con S a través del cierre transitivo de la relación[7] de inclusión $\lhd$. Análogamente, todos los elementos contenidos en el conjunto $\mathcal{P}_S$ se relacionan con S a través del cierre transitivo de la relación de inclusión $\rhd$. Como comentario final se puede enunciar que así como las relaciones $\lhd$ y $\rhd$ son simétricas la una respecto a la otra, sus respectivos cierres transitivos no son simétricos el uno respecto al otro.

Definición 8.4 (Elemento origen de documento) *El elemento estructural S será un elemento origen de documento (o documento) si no está contenido en ningún otro elemento estructural, $S \lhd \emptyset$, o lo que es lo mismo, si y sólo si las posiciones inicial y final de S son o ambas mayores o ambas menores que las respectivas de cualquier otro elemento perteneciente al conjunto $\mathcal{T} - \mathcal{E}_S - \{S\}$:*

$$S \lhd \emptyset \iff \forall S' \in \mathcal{T} - \mathcal{E}_S - \{S\} \bullet (\Lambda_S > \Lambda_{S'} \wedge \Omega_S > \Omega_{S'}) \vee (\Lambda_S < \Lambda_{S'} \wedge \Omega_S < \Omega_{S'}) \quad (8.6)$$

Llamamos $\mathcal{D}$ al conjunto de todos los elementos origen de documento, definido como:

$$\mathcal{D} = \{S \in \mathcal{T} \bullet S \lhd \emptyset\} \quad (8.7)$$

El conjunto $\mathcal{D}$ es un subconjunto de $\mathcal{T}$, $\mathcal{D} \subset \mathcal{T}$, ya que $\mathcal{T}$ contiene todos los elementos estructurales.

Definición 8.5 (Elemento padre) *Sean $S \in \mathcal{T}$ y $S' \in \mathcal{T}$ elementos estructurales presentes en la colección con $S \in \mathcal{P}_{S'}$. S es el padre (o ascendiente inmediato) de S', denotado como $S \blacktriangle S'$, si y sólo si S es el elemento estructural que ocupa la posición anterior inmediata a S' en la jerarquía:*

$$S \blacktriangle S' \iff S' \notin \mathcal{D} \mid \forall\, S'' \in (\mathcal{P}_{S'} - \{S\}) \bullet \Lambda_S > \Lambda_{S''} \quad (8.8)$$

Para cualquier elemento estructural $S \in \mathcal{T} - \mathcal{D}$ existirá un y sólo un elemento padre. Los únicos elementos estructurales que no tienen elemento padre son los pertenecientes al conjunto $\mathcal{D}$:

$$\emptyset \blacktriangle S \iff S \in \mathcal{D} \quad (8.9)$$

Definición 8.6 (Elemento hijo) *Sean $S \in \mathcal{T}$ y $S' \in \mathcal{T}$ dos elementos estructurales. S es un hijo (o descendiente inmediato) de S', denotado mediante $S \blacktriangledown S'$, si y sólo si S pertenece al conjunto $\mathcal{E}_{S'}$ y además las posiciones inicial y final de S son o ambas mayores o ambas menores que las respectivas de cualquier otro elemento perteneciente al conjunto $\mathcal{E}_{S'} - \mathcal{E}_S - \{S\}$:*

$$S \blacktriangledown S' \iff S \in \mathcal{E}_{S'} \mid \forall S'' \in \mathcal{E}_{S'} - \mathcal{E}_S - \{S\} \bullet (\Lambda_S > \Lambda_{S''} \wedge \Omega_S > \Omega_{S''}) \vee (\Lambda_S < \Lambda_{S''} \wedge \Omega_S < \Omega_{S''}) \quad (8.10)$$

Definición 8.7 (Conjunto de hijos) *Sea $S \in \mathcal{T}$ un elemento estructural. Sea $\mathcal{C}_S$ el conjunto formado por todos los hijos (o descendientes inmediatos) definido como*

$$\mathcal{C}_S = \{S' \in \mathcal{E}_S \bullet S' \blacktriangledown S\} \quad (8.11)$$

Con los operadores "padre" e "hijo" también nos encontramos con propiedad adicional relacionada con ambos y es que si tenemos dos elementos $S, S' \in \mathcal{T}$ y se cumple que $S \blacktriangle S'$, entonces se deduce de manera inmediata que $S' \blacktriangledown S$ y viceversa.

[7] La menor relación binaria transitiva sobre el conjunto de los nodos que contiene la relación de inclusión.

Definición 8.8 (Elementos hoja) *Sea $S \in \mathcal{T}$ un elemento estructural. S es un elemento hoja si S no tiene ningún elemento hijo, $\mathcal{C}_S = \emptyset \ \vee \ S \blacktriangle \emptyset$. $\mathcal{L}$, el conjunto de todos los elementos hoja, se define como:*

$$\mathcal{L} = \{S \in \mathcal{T} \mid \mathcal{C}_S = \emptyset\} \tag{8.12}$$

Definición 8.9 (Elementos hermanos) *Sean $S \in \mathcal{T}$, $S' \in \mathcal{T}$ y $S'' \in \mathcal{T}$ elementos estructurales. S' es hermano de S'', y se denotará mediante $S' \bowtie S''$ si y sólo si S es padre de S' y S'':*

$$S' \bowtie S'' \iff S \blacktriangle S' \ \wedge \ S \blacktriangle S'' \tag{8.13}$$

$$S' \bowtie S'' \iff S' \in \mathcal{C}_S \ \wedge \ S'' \in \mathcal{C}_S \tag{8.14}$$

8.2.1. Transitividad entre los operadores

En determinadas ocasiones si el elemento estructural A está relacionado con el elemento B y éste a su vez con un tercero C, puede ser muy útil saber a priori cómo están relacionados los elementos A y C y esto se puede saber empleando las relaciones de transitividad entre los operadores. En la tabla 8.1 se muestran todas las posibles combinaciones entre todas posibles las parejas de operadores definidos con anterioridad así como la correspondiente relación transitiva entre cada par de parejas. En algunas circustancias no es posible determinar la relación transitiva, estos casos se reflejan en la tabla utilizando el símbolo "`?`"; en otros casos no existe ningún tipo de relación transitiva inmediate entre los operadores y este hecho se refleja en la tabla usando el acrónimo "`SR`". Por último, existe un caso en el que la relación transitiva es imposible dadas las relaciones entre los operadores implicados debido a la naturaleza arbórea de la estructura de los documentos y se encuentra representado mediante el acrónimo "`IMP`" en la tabla.

B *op* C / A *op* B	$\triangleright$	$\triangleleft$	$\blacktriangle$	$\blacktriangledown$	$\bowtie$
$\triangleright$	$\triangleright$	?	$\triangleright$	$\triangleright$	$\triangleright$
$\triangleleft$	?	$\triangleleft$	?	$\triangleleft$	SR
$\blacktriangle$	$\triangleright$	$\triangleleft$	$\triangleright$	IMP	$\blacktriangle$
$\blacktriangledown$	?	$\triangleleft$	$\bowtie$	$\triangleleft$	SR
$\bowtie$	SR	$\triangleleft$	SR	$\blacktriangledown$	$\bowtie$

Tabla 8.1: Tabla de transitividades entre los operadores

8.3. Recuperación

En esta sección se propone una aproximación a un modelo de recuperación de información que realiza un análisis de la relevancia teniendo en cuenta la estructura de los documentos y el tamaño de los elementos estructurales. El objetivo principal es intentar mejorar las prestaciones de los sistemas de recuperación de información permitiendo resolver de forma eficiente las consultas en las que esté presente la estructura y mejorando los rankings de respuestas. Para ello se debe tener en consideración que, al recuperar elementos estructurales de un nivel jerárquico superior, los términos de la consulta aparezcan o no en un mismo elemento estructural de nivel jerárquico inferior y el tamaño de los elementos estructurales que contengan los términos de la consulta. Así pues, la propuesta intenta

primar a los elementos estructurales más pequeños y que contengan todos los términos de la consulta.

A la hora de enunciar el modelo de asignación de pesos a los elementos estructurales se han tenido en cuenta los siguientes objetivos:

I. Obtener una clasificación no sólo a nivel de documento sino también en cualquiera de los subniveles en que éstos estén organizados.

II. Primar aquellos elementos que contengan los términos de la consulta de *forma más densa*, es decir, si dos elementos contienen el mismo número de veces los distintos términos de la consulta se primará al más pequeño, y si dos elementos tienen el mismo tamaño se primará aquel que contenga los términos de la consulta más veces.

III. Considerar como elemento más relevante aquel que contenga los términos de la consulta dentro de un mismo subelemento frente aquel otro que los contenga en diferentes subelementos.

IV. A igualdad de número de términos de la consulta presentes en un elemento estructural se considerarán más relevantes aquellos elementos que contengan todos los términos de la consulta frente a los que les falte un término y, a su vez, éstos más relevantes que aquellos que les falten dos términos y así sucesivamente.

Una de las principales motivaciones es recuperar y ordenar los elementos estructurales (de nivel determinado) de acuerdo con el concepto de densidad de información, es decir, interesa recuperar aquellos elementos estructurales que contengan proporcionalmente un mayor número de términos de la consulta. La idea subyacente es intentar recuperar aquellos elementos estructurales que contengan la información requerida de forma más concentrada y concisa, la cual generalmente se encuentra encuentra en los nodos que ocupan los niveles inferiores de los documentos semiestructurados. Con este enfoque el ranking de nodos que se obtendrá no responderá explícitamente ni a un enfoque puramente booleano ni vectorial pues, en determinadas circunstancias, se pueden encontrar nodos a los que les falta algún término de la consulta con peso mayor (más relevantes) que otros que contienen todos los términos de la consulta. Por otro lado, esta propuesta permite clasificar los documentos que contienen todos los términos de la consulta de acuerdo con los criterios expuestos con anterioridad.

8.3.1. Descripción de la propuesta

Viendo la estructura de los documentos en forma de árbol, el cálculo de los pesos de los distintos elementos se comienza a calcular desde los nodos hoja para luego ir subiendo por niveles hasta alcanzar el tipo o nivel del nodo recuperable. Por consiguiente, se propocionarán dos medidas: una para pesar los nodos hoja (que, probablemente, contendrán los textos de la colección) dependiendo del número de veces que contengan los términos de la consulta; y otra medida para obtener el peso del resto de los nodos. En todo momento se tendrá en cuenta cuántos términos de la consulta están presentes en un nodo, el número de veces que aparezcan, lo significativo que sean y el tamaño de dicho nodo.

Pesado de los nodos hoja

Para realizar el pesado de un nodo $S \in \mathcal{L}$ se parte del número de veces que aparece cada término de la consulta $\mathcal{Q} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ en S.

Definición 8.10 (Grupos excluyentes) *Sea* $\Im_{S,\mathcal{Q}} = \{\tau_1, \ldots, \tau_1, \tau_2, \ldots, \tau_2, \ldots, \tau_n, \ldots, \tau_n\}$ *el conjunto con repetición de los* n *términos de la consulta* $\mathcal{Q}$ *que aparecen en el elemento estructural* S*. El conjunto* $\Im_{S,\mathcal{Q}}$ *se puede dividir en* k *subconjuntos sin repetición o grupos excluyentes* $\mathcal{G}_i$ *con* $0 \leq i < k$*, de manera que contenga el mayor número posible de términos diferentes de* $\mathcal{Q}$ *extraídos del conjunto* $\{\Im_{S,\mathcal{Q}} - \cup_{j=0}^{i-1}\mathcal{G}_j\}$*, una vez formados los* k *grupos excluyentes se debe cumplir que*

$$\Im_{S,\mathcal{Q}} = \cup_{i=0}^{k-1}\mathcal{G}_i \tag{8.15}$$

Por último, $\mathcal{G}$ *es el conjunto de los* k *grupos formados a partir de* $\Im_{S,\mathcal{Q}}$ *definido como* $\mathcal{G} = \{\mathcal{G}_i,\ 0 \leq i < k\}$

Definición 8.11 (Grupo completo) *Sea* $\mathcal{G}_i$ *un grupo excluyente formado a partir de* $\Im_{S,\mathcal{Q}}$*. Se dirá que* $\mathcal{G}_i$ *es un grupo completo si y sólo si* $\mathcal{G}_i$ *contiene todos los términos de* $\mathcal{Q}$*, es decir, si* $\mathcal{G}_i = \mathcal{Q}$*.*

Una vez formados los k grupos excluyentes hay que ver cuántos grupos completos hay, cuántos a los que les falta un término ($|\mathcal{Q}| - |\mathcal{G}_i| = 1$), cuántos a los que les faltan dos términos ($|\mathcal{Q}| - |\mathcal{G}_i| = 2$) y así sucesivamente. Por ejemplo, si la consulta fuera $\mathcal{Q} = \{a, b, c\}$ y un elemento hoja contiene cuatro veces el término a, tres el término b y una el c entonces el elemento estructural contiene un grupo completo (a, b, c), dos veces el grupo (a, b) y una vez el grupo (a).

Definición 8.12 (Importancia de un término) *Sea* $|\mathcal{Q}|$ *el número de términos de la consulta y* N_i *el número de veces que el término* i *aparece en la colección. La importancia del término* $x \in \mathcal{Q}$*,* Υ_x*, se calcula de la siguiente manera*

$$\Upsilon_x = \begin{cases} 1 & si \quad |\mathcal{Q}| = 1 \\ \frac{\sum_{y \in \mathcal{Q}, y \neq x} N_y}{(|\mathcal{Q}|-1)\sum_{y \in \mathcal{Q}} N_y} & si \quad |\mathcal{Q}| > 1 \end{cases} \tag{8.16}$$

Definición 8.13 (Importancia de un grupo) *La importancia del grupo* $\mathcal{G}_i$ *se calcula mediante la expresión*

$$I_{\mathcal{G}_i} = 1 - \sum_{x \in \{\mathcal{Q} - \mathcal{G}_i\}} \Upsilon_x \tag{8.17}$$

Definición 8.14 (Pesado de los nodos hoja) *Sea* T *el número de términos totales presentes en el texto del nodo* S *y* $f_{\mathcal{G}_i}$ *el número de términos de la consulta* $\mathcal{Q}$ *que faltan en el grupo* $\mathcal{G}_i$ *se obtiene mediante la función* $f(\mathcal{Q}, \mathcal{G}_i) = |\{\mathcal{Q} - \mathcal{G}_i\}|$ *El peso del nodo* $S \in \mathcal{L}$ *respecto a la consulta* $\mathcal{Q}$*,* $\omega'(S, \mathcal{Q})$*, se calcula mediante*

$$\omega'(S, \mathcal{Q}) = \frac{|\mathcal{Q}|}{T} \sum_{\forall \mathcal{G}_i \in \mathcal{G}} \frac{I_{\mathcal{G}_i}}{|\mathcal{Q}|^{f(\mathcal{Q}, \mathcal{G}_i)}} \tag{8.18}$$

Ejemplo de aplicación

Supóngase que se desea realizar la consulta $\mathcal{Q} = \{a, b, c\}$ y que se dispone de los documentos representados en la figura 8.7. Para simplificar también se supondrá que:

1. Los nodos hoja son los únicos que contienen texto.
2. Todos los nodos hoja contienen el mismo número de términos (en este caso $T = 10$).
3. En todos los nodos aparecen términos relevantes a la consulta.

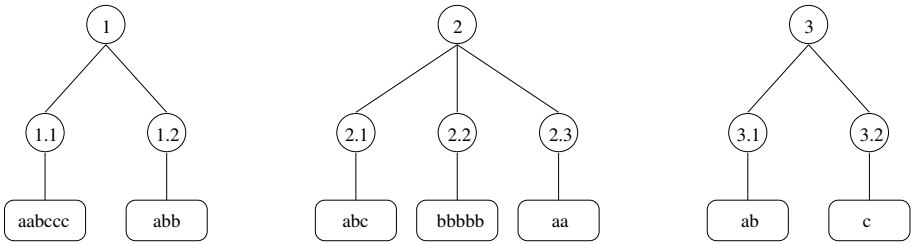


Figura 8.7: Conjunto 1 de documentos de ejemplo

Nodo	$\omega'(S_x, \mathcal{Q})$	Ranking
$S_{1,1}$	0,3856	1
$S_{1,2}$	0,0704	3
$S_{2,1}$	0,3000	2
$S_{2,2}$	0,0454	5
$S_{2,3}$	0,0227	6
$S_{3,1}$	0,0613	4
$S_{3,2}$	0,0128	7

Tabla 8.2:

Así pues, tenemos que $N_a = 7$, $N_b = 10$ y $N_c = 5$ con $|\mathcal{Q}| = 3$. Con estos datos, la relevancia del nodo $S_{1,1}$ siguiendo el modelo propuesto se calcula de la siguiente manera:

$$\begin{aligned}
\mathcal{G} &= \overbrace{\{a,b,c\}}^{\mathcal{G}_0} \cup \overbrace{\{a,c\}}^{\mathcal{G}_1} \cup \overbrace{\{c\}}^{\mathcal{G}_2} \\
\Upsilon_a &= \tfrac{N_b+N_c}{2\times(N_a+N_b+N_c)} = \tfrac{15}{44} \\
\Upsilon_b &= \tfrac{N_a+N_c}{2\times(N_a+N_b+N_c)} = \tfrac{12}{44} = \tfrac{3}{11} \\
I_{\mathcal{G}_0} &= 1 \\
I_{\mathcal{G}_1} &= 1 - \Upsilon_b = \tfrac{8}{11} \\
I_{\mathcal{G}_2} &= 1 - \Upsilon_a - \Upsilon_b = \tfrac{17}{44} \\
\omega(S_{1,1}, \mathcal{Q}) &= \tfrac{|\mathcal{Q}|}{T} \times \left(\tfrac{I_{\mathcal{G}_0}}{|\mathcal{Q}|^{f(\mathcal{Q},\mathcal{G}_0)}} + \tfrac{I_{\mathcal{G}_1}}{|\mathcal{Q}|^{f(\mathcal{Q},\mathcal{G}_1)}} + \tfrac{I_{\mathcal{G}_2}}{|\mathcal{Q}|^{f(\mathcal{Q},\mathcal{G}_2)}} \right) \\
&= \tfrac{3}{10} \times \left(\tfrac{1}{3^0} + \tfrac{\frac{8}{11}}{3^1} + \tfrac{\frac{17}{44}}{3^2} \right) = \tfrac{509}{1320} \approx 0,3856
\end{aligned}$$

El cálculo de la relevancia para el resto de los nodos se realizaría de manera similar. En la tabla 8.2 se muestran las relevancias aproximadas obtenidas para la consulta $\mathcal{Q}$ por cada nodo hoja presente en los documentos de ejemplo citados con anterioridad.

A la vista de los resultados, los elementos con más peso son el $S_{1,1}$ y el $S_{2,1}$ ya que son los únicos que contienen todos los términos de la consulta. Además el elemento $S_{1,1}$ contiene más términos de $\mathcal{Q}$ presentes en grupos incompletos por lo que es el tiene más relevancia. Por otro lado, los elementos $S_{1,2}$ y $S_{3,1}$ sólo les falta un término de la consulta por lo que siguen a los anteriores en el ranking. Se puede comentar al respecto que el elemento $S_{1,2}$ es más revelante que el $S_{3,1}$ porque contiene un término de $\mathcal{Q}$ más. Por último se sitúan los elementos en los que sólo aparece un término de la consulta y al igual que en el caso anterior, los nodos que tienen un mayor número de términos de la consulta son más relevantes.

Pesado de los nodos de niveles superiores

Para realizar el pesado de los nodos de niveles superiores se intentará primar aquellos elementos estructurales que contengan nodos que respondan a la consulta en los niveles más bajos frente aquellos que lo hagan en niveles superiores, ya que los primeros tendrán los términos de la consulta más próximos que los segundos. En el cálculo del peso de un nodo $S \notin \mathcal{L}$ se tendrán en cuenta todos los nodos $S' \in \mathcal{E}_S$.

Para calcular el peso de un nodo S cualquiera presente en una colección de documentos repecto a la consulta $\mathcal{Q}$, $\omega(S, \mathcal{Q})$, se propone un método que consiste en pesar los nodos hoja descendientes del mismo y multiplicar el valor máximo obtenido por $1/2$. A continuación se le suma el resultado de multiplicar $1/4$ por el peso máximo obtenido pesando los nodos del nivel anterior como si se tratasen de nodos hoja. El proceso continúa hasta alcanzar el nivel en el que se encuentra el nodo S, multiplicando siempre por $1/2^{n+1}$ el valor del peso máximo obtenido por los nodos de nivel n (comenzando a contar desde los nodos hoja hacía sus ascendentes).

Algoritmo 8.1 (Pesado de un nodo S cualquiera)

$\omega(S, \mathcal{Q}) \leftarrow 0$
$n \leftarrow 0$
if $(\mathcal{E}_S = \emptyset)$
  then
    $\mathcal{X} \leftarrow \{S\}$
  else
    $\mathcal{X} \leftarrow \mathcal{L} \cap \mathcal{E}_S$
fi
while $(\mathcal{X} \neq \{S\})$ ***do***
    $mayor\_peso \leftarrow \text{máx}\,(\omega'(S', \mathcal{Q})\ \forall S' \in \mathcal{X})$
    $\omega(S, \mathcal{Q}) \leftarrow \omega(S, \mathcal{Q}) + \frac{1}{2^{n+1}} \times mayor\_peso$
    $n \leftarrow n + 1$
    $\mathcal{X} \leftarrow \{S_p \mid \exists S' \in \mathcal{X} \ \bullet \ S_p \blacktriangle S'\}$
od
if $(n = 0)$
  then
    $n \leftarrow -1$
fi
$\omega(S, \mathcal{Q}) \leftarrow \omega(S, \mathcal{Q}) + \frac{1}{2^{n+1}} \times \omega'(S, \mathcal{Q})$

El algoritmo 8.1 detalla la manera de calcular la relevancia de un nodo S cualquiera presente en la colección. Mediante $\omega(S, \mathcal{Q})$ se denota el cálculo del peso del nodo S como si se tratara de un nodo hoja tal y como se ha detallado con anterioridad. La condición que aparece al final habilita al algoritmo para obtener el peso de los nodos hoja de forma correcta y así poder calcular el peso de cualquier nodo de la colección.

Ejemplo de aplicación

Volvamos al ejemplo anterior para calcular el peso de los nodos S_1, S_2 y S_3 correspondientes al ejemplo ilustrado en la figura 8.7. Al igual que en el caso anterior, se mostrará la forma de calcular el peso de un nodo (el S_1 esta vez), el resto de los nodos se obtiene de forma similar. Para calcular el peso del nodo S_1 como un nodo hoja se debe tener en

Modelo →	Modelo de densidad		Modelo vectorial	
Nodo ↓	$\omega(S_x, \mathcal{Q})$	Ranking	$\text{sim}(S_x, \mathcal{Q})$	Ranking
S_1	0,305	1	1,000	1
S_2	0,187	2	0,878	3
S_3	0,068	3	1,000	1

Tabla 8.3:

cuenta que los términos de la consulta que contiene están incluidos en sus hijos y, en este caso, el número de términos presentes en los textos que lo componen es la suma del número de términos de los hijos ($T = 20$ en este caso). Con esta información, primero se detalla la relevancia del nodo S_1 como si fuera un nodo hoja, $\omega(S_1, \mathcal{Q})$, y a continuación se calcula el peso del nodo considerando sus descendientes, $\omega(S_1, \mathcal{Q})$, teniendo en cuenta que peso del nodo $S_{1,1}$ ha sido el mayor entre sus hijos.

$$
\begin{aligned}
\mathcal{G} &= \overbrace{\{a,b,c\}}^{\mathcal{G}_0} \cup \overbrace{\{a,b,c\}}^{\mathcal{G}_1} \cup \overbrace{\{a,b,c\}}^{\mathcal{G}_2} \\
I_{\mathcal{G}_0} &= 1 \\
I_{\mathcal{G}_1} &= 1 \\
I_{\mathcal{G}_2} &= 1 \\
\omega(S_1, \mathcal{Q}) &= \frac{|\mathcal{Q}|}{T} \times \left(\frac{I_{\mathcal{G}_0}}{|\mathcal{Q}|^{f(\mathcal{Q},\mathcal{G}_0)}} + \frac{I_{\mathcal{G}_1}}{|\mathcal{Q}|^{f(\mathcal{Q},\mathcal{G}_1)}} + \frac{I_{\mathcal{G}_2}}{|\mathcal{Q}|^{f(\mathcal{Q},\mathcal{G}_2)}} \right) \\
&= \frac{3}{20} \times \left(\frac{1}{3^0} + \frac{1}{3^0} + \frac{1}{3^0} \right) = \frac{9}{20} \\
\omega(S_1, \mathcal{Q}) &= \frac{1}{2} \times \omega(S_{1,1}, \mathcal{Q}) + \frac{1}{4} \times \omega(S_1, \mathcal{Q}) \\
&= \frac{1}{2} \times \frac{509}{1320} + \frac{1}{4} \times \frac{9}{20} = \frac{403}{1320} \approx 0,305
\end{aligned}
$$

En la tabla 8.3 se pueden observar los pesos obtenidos para los nodos origen de documento, $S_x \in \mathcal{D}$, presentandos en la figura 8.7. También en la misma tabla se pueden observar los valores que se han obtenido para la función de similitud del modelo vectorial básico descrito en §5.2.2 para la consulta $\mathcal{Q} = \{a, b, c\}$ y suponiendo que la frecuencia de aparición en los documentos de los términos que no se encuentran presentes en $\mathcal{Q}$ es 1. Además, también se supone que el número total de documentos, N, es igual al total de documentos de ejemplo de la colección más uno debido a que en los documentos mostrados en la figura 8.7 aparecen todos los términos de la consulta. El modelo vectorial considera irrelevante un documento de la colección que contenga todos los términos de la consulta (la función de similitud de ese documento respecto a la consulta devolverá el valor cero); por otro lado, el modelo de densidad propuesto considera relevantes todos los documentos que contengan al menos un término de la consulta y en el caso que aparezcan todos éstos se ordenarán dependiendo de la proximidad de los términos y el contexto en el que se encuentren. Consecuentemente, en esta situación no se puede efectuar una comparación entre ambos modelos pero si se añade un documento adicional "vacío" o que no contenga ningún término ni de la consulta ni que aparezca en los documentos anteriores entonces la comparación es posible. Al añadir el documento "vacío" los resultados obtenidos por el modelo de densidad no se ven afectados mientras que la función de similitud del modelo vectorial devuelve el valor 1 a los documentos que contienen todos los términos de la consulta.

Para finalizar se verá un ejemplo más para poner de manifiesto la importancia de la proximidad. Sea la colección formada por los documentos representados en la figura 8.8, para simplificar las operaciones se supondrá que el número de términos de todos los documentos es $T = 12$ y los elementos hijos de cada nodo se reparten los términos del padre de forma equitativa. Al igual que en el caso anterior la consulta será $\mathcal{Q} = \{a, b, c\}$ y

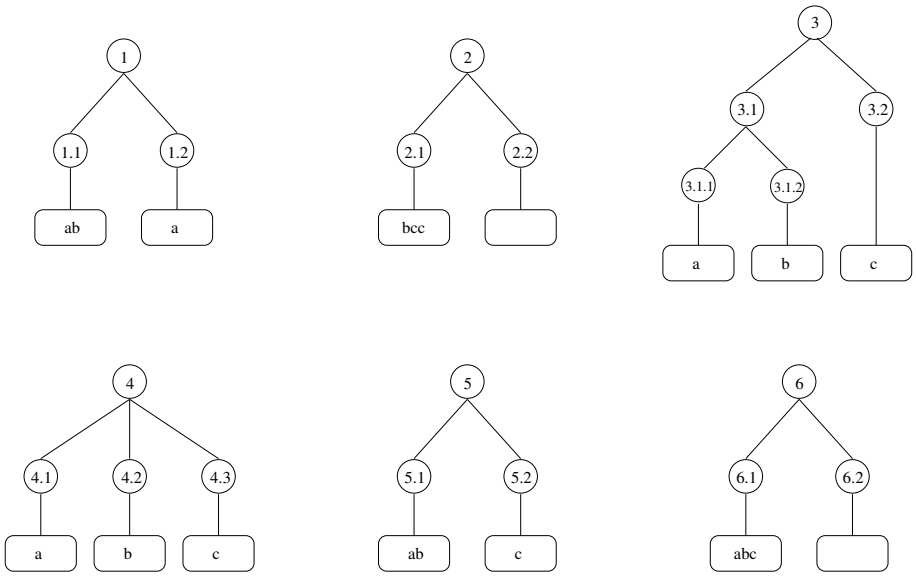


Figura 8.8: Conjunto 2 de documentos de ejemplo

Modelo →	Modelo de densidad		Modelo vectorial	
Nodo ↓	$\omega(S_x, \mathcal{Q})$	Ranking	$\text{sim}(S_x, \mathcal{Q})$	Ranking
S_1	0,0601	5	0,249	5
S_2	0,0648	4	0,249	5
S_3	0,0590	6	1,000	1
S_4	0,0902	3	1,000	1
S_5	0,1180	2	1,000	1
S_6	0,3125	1	1,000	1

Tabla 8.4:

en la figura se puede observar que los tres términos son igualmente significativos (aparecen el mismo número de veces en la colección). La relevancia obtenida para cada documento se puede observar en la tabla 8.4.

A los documentos S_1 y S_2 les falta un término de la consulta, sin embargo como el S_2 los tiene más próximos (todos dentro de un mismo elemento estructural) queda delante en el ranking. Por otro lado, el documento S_3 tiene todos los términos de la consulta una vez cada uno pero muy dispersos y en diferentes niveles, consecuentemente se ha clasificado en último lugar de acuerdo a los principios de concentración y concisión que se persiguen. El hecho de encontrar términos de la consulta en diferentes partes (niveles estructurales) del documento induce a pensar que dichos términos se encuentran en diferentes contextos o entornos lingüísticos y, posiblemente, el documento responde en su conjunto a la consulta vagamente. Siguiendo el mismo razonamiento que en el caso anterior, se puede decir que los documentos S_1 y S_2 responden en su conjunto a la consulta mejor que el documento S_3.

Por último, los documentos S_4, S_5 y S_6 también contienen todos los términos de la consulta una sóla vez pero más próximos que en el documento S_3 y por eso el modelo les asigna un peso teniendo en cuenta la concentración de los términos: los tres se clasifican mejor

que los documentos anteriormente citados pues tienen todos los términos de la consulta relativamente cercanos. El documento S_6 es el que tiene mayor peso (mejor clasificado) ya que todos los términos de la consulta aparecen en un mismo elemento y le siguen S_5 y S_4 ajustándose a los criterios expuestos anteriormente.

Los rankings obtenidos con estos ejemplos han sido comparados con los obtenidos aplicando el modelo vectorial obteniendo resultados muy similares: ambos coinciden tanto en las primeras pociones como en las últimas, existiendo pequeñas diferencias en las posiciones intermedias. Estas diferencias se deben fundamentalmente a la diferente forma de pesar lo significativo que son unos términos frente a otros, y se justifican con los objetivos iniciales que se han marcado.

8.4. Evaluación de la propuesta

Se ha desarrollado un prototipo que implementa el índice propuesto con anterioridad, para probarlo se han utilizado diferentes tamaños de la colección WSJ de la TREC-3 [Har95] cuyas características se pueden observar en la tabla 6.1 de la página 87[8]. La propuesta planteada se ha comparado con los índices invertidos con direccionamiento a palabra y con direccionamiento a documento ya que, en una primera etapa, se ha supuesto que el nuevo índice tendría unas prestaciones intermedias entre los citados. Por otro lado también se ha realizado la compresión del índice utilizando un vector intermedio según lo comentado en la sección 8.1.3. Los otros dos tipos de índices también se han comprimido utilizando la codificiación de enteros Elias–δ, la misma que utiliza el prototipo.

Las tablas 8.5 y 8.6 muestran el tamaño del índice con y sin compresión y la razón de compresión del mismo obtenidos en cada tamaño de colección al indexarlas utilizando el direccionamiento a palabras y a documentos respectivamente. Tal y como se podía esperar, el índice con direccionamiento a palabra ocupa de un 70 % a un 80 % más de espacio adicional cuando se compran los tamaños de ambos índices sin comprimir, no obstante, esta diferencia se dispara hasta porcenjates que varían entre un 380 % y un 530 % cuando se comparan los respectivos índices comprimidos.

WSJ	Índice	Índice comprimido	Razón
1	1.701.232	623.642	36,66 %
5	7.320.272	3.094.244	42,27 %
10	13.799.568	6.203.278	44,95 %
20	27.591.112	13.255.492	48,04 %
40	54.400.648	27.798.822	51,10 %
60	81.213.112	42.924.152	52,85 %
100	134.800.872	69.930.554	51,88 %
200	268.944.744	138.491.790	51,49 %

Tabla 8.5: Tamaño. IIDW

Por su parte, la tabla 8.7 muestra para cada tamaño de colección, el tamaño del índice con direccionamiento a estructura con y sin compresión así como la razón de compresión del mismo. Para realizar los experimentos se han indexado todas las etiquetas que aparecen en la DTD de los documentos y, evidentemente, en los tamaños de los índices está incluido

[8]La colección más grande no figura en ella, tiene un tamaño 210.009.482 bytes y aparecen 33.359.445 palabras en total, de las cuales 194.324 son diferentes (0,582 %)

WSJ	Índice	Índice comprimido	Razón
1	999.608	128.842	12,89 %
5	4.154.792	586.528	14,12 %
10	7.730.232	1.108.892	14,34 %
20	15.287.784	2.232.912	14,61 %
40	29.848.744	4.400.616	14,74 %
60	44.451.272	6.578.584	14,80 %
100	73.942.720	10.943.184	14,80 %
200	149.162.480	21.854.518	14,65 %

Tabla 8.6: Tamaño. IIDD

el tamaño de la tabla de control de estructura correspondiente y el tamaño del vector utiliado en la compresión de la misma. En la tabla 8.8 se pueden observar el tamaño de la tabla de control con y sin compresión, del tamaño de vector generado en la compresión y la razón de compresión de la misma. Cada elemento de dicha tabla contiene la posición en la que se encuentra la entrada respectiva en la tabla de control codificada, y por lo tanto, su tamaño es lineal respecto al número de elementos de dicha tabla. Por otro lado, el número de elementos de la tabla de control de estructura es lineal respecto al número de elementos estructurales totales presentes en los documentos.

El uso de la tabla de control de estructura supone un incremento en el tamaño final del índice. En los experimentos realizados cuando los índices no se encuentran comprimidos el espacio que ocupa la tabla de control de estructura respecto al tamaño total varía entre un 5,7 % para la colección más pequeña y un 7 % para la más grande; por otro lado, cuando los índices están comprimidos la proporción aumenta hasta un 14,3 % para la colección más pequeña y hasta un 18,1 % para la mayor. Esta información sugiere que la compresión obtenida en las listas de ocurrencias es mayor que la obtenida en la tabla de control de estructura.

En la comparación con los otros dos tipos de índices, a simple vista se puede observar que los tamaños obtenidos para los índices con direccionamiento a estructura se encuentran entre los tamaños de los índices con direccionamiento a palabra y documento, aunque mucho más cercanos a los tamaños del índice con direcionamiento a documentos. Concretamente, el índice con direccionamiento a palabra sin comprimir ocupa de un 66 % más en la colección más pequeña a un 75 % en la colección más grande que el de direccionamiento a estructura; no obstante, el índice con direccionamiento a palabra comprimido ocupa de un 200 % a un 285 % más que la propuesta.

Por otro lado, el tamaño del índice propuesto sin comprimir ocupa un 2,5 % más que el índice con direccionamiento a documento; en este caso la relación se mantiene constante para todos los tamaños de colección y no supone un incremento demasiado grande, pero esta variación dependerá de la naturaleza de la colección de documentos, teniendo especial relevancia el número de elementos estructurales que se indexan y la distribución de los términos dentro de los mismos. La diferencia aumenta cuando se comparan los índices comprimidos ya que el índice con direccionamiento a estructura ocupa de un 61 % más en la colección más pequeña a un 75 % más en la colección mayor; aunque este incremento en tamaño es significativo respecto al caso anterior la diferencia es mucho menor cuando se compara con el índice con direccionamiento a palabra.

WSJ	Índice	Índice comprimido	Razón
1	1.024.448	207.736	20,28 %
5	4.261.576	942.028	22,11 %
10	7.946.120	1.785.832	22,47 %
20	15.718.184	3.594.888	22,87 %
40	30.719.168	7.094.094	23,09 %
60	45.740.880	10.614.950	23,21 %
100	76.044.880	17.732.918	23,32 %
200	152.942.800	35.988.794	23,53 %

Tabla 8.7: Tamaño. IIDE

WSJ	TCE	TCE comp.	Vector	Total	Razón
1	58.640	22.392	7.330	29.722	50,69 %
5	264.208	119.134	33.026	152.160	57,59 %
10	497.040	229.940	62.130	292.070	58,76 %
20	997.072	426.832	124.634	551.466	55,31 %
40	1.944.112	860.058	243.014	1.103.072	56,74 %
60	2.897.648	1.305.456	362.206	1.667.662	57,55 %
100	4.963.424	2.299.368	620.428	2.919.796	58,83 %
200	10.815.600	5.193.848	1.351.950	6.545.798	60,52 %

Tabla 8.8: Tamaño. Tabla de control de estructura.

Capítulo 9

Conclusiones y trabajo futuro

En el marco de este trabajo se ha considerado el aspecto semántico de la estructura en los procesos de compresión, indexación y recuperación sobre textos con estructura. Unos de los principales objetivos ha sido estudiar y sacar partido de la estructura de los documentos en los procesos comentados con anterioridad. Un sistema de recuperación de información que maneje documentos con estructura comprimidos puede aprovechar la estructura de los mismos para mejorar las razones de compresión para que a la vez permita resolver eficientemente consultas por contenido y estructura y mostrar los resultados de manera adecuada.

Con el propósito de mejorar las razones de compresión, se ha propuesto un modelo genérico que permite comprimir documentos semiestructurados basándose en el aspecto semántico de la estructura. La principal idea es la de considerar que las frecuencias de las palabras del texto que se encuentran dentro de un mismo tipo de elemento estructural se deben ajustar a una misma distribución. El modelo se ha denominado modelo de contextos estructurales (SCM) y codifica con un mismo modelo los textos que se encuentran en cada tipo de elemento estructural. En su versión semiadaptitiva se han utilizado codificadores de Huffman basados en palabras para tal cometido, denominándose SCMWBH. Además, esta idea se ha enriquecido mediante una heurística que permite agrupar de forma adecuada diferentes tipos de elementos estructurales, de manera que cada grupo se codifique con un modelo específico independiente. Por otro lado, el impacto del modelo en el rendimiento del proceso de recuperación es insignificante, de hecho es similar al rendimiento de la recuperación sobre documentos comprimidos.

Se ha mostrado que, actualmente, esta idea mejora las razones de compresión en más de un 10 % respecto a la técnica básica (modelo de alfabetos separados). Por otro lado, el prototipo se ha comparado con los sistemas de compresión referenciados en el "estado del arte" y se ha comprobado que dicho prototipo obtiene la mejor tasa de compresión para tamaños de colección medios y grandes (que, por cierto, son los más interesantes en el ámbito de las bases de datos textuales: 40 megabytes o más) respecto a técnicas que permiten el acceso directo sobre el texto comprimido, esencial para las bases de datos comprimidas. En colecciones de gran tamaño, la diferencia del prototipo con el sistema que obtiene mejor compresión no llega al 7,2 %, pero dicho sistema, a diferencia del propuesto aquí, no permite el acceso directo sobre los textos comprimidos.

También se ha propuesto una versión adaptativa, denominada SCMPPM, que combina la técnica genérica SCM junto con un modelado PPMD+, con la finalidad de almacenar y transmitir documentos semiestructurados. Se ha mostrado que esta versión mejora la

compresión en más de un 1 % respecto a la compresión con un único modelo PPMD+, y en un 25 % respecto a los compresores citados en el "estado del arte".

Por otro lado, también se ha presentado un método denominado LZCS, un esquema de compresión inspirado en el esquema Lempel-Ziv, pensado para comprimir documentos estructurados. La idea principal de LZCS es sustituir subestructuras completas por una referencia a una ocurrencia previa de la misma, de manera que se capture la redundancia que introduce la estructura en una colección de documentos. En general, la transformación LZCS tiene las siguientes ventajas:

I. Razones de compresión muy buenas, que mejoran las obtenidas por los métodos clásicos y la mayoría de las obtenidas por los métodos de compresión que consideran la estructura.

II. Facilidad en la navegación, visualización y acceso aleatorio sobre las colecciones comprimidas.

III. Compresión y descompresión en una sóla etapa y rapidez del proceso.

En la experimentación realizada, sólo XMLPPM comprime mejor que LZCS pero, como ya se ha comentado con anterioridad, con XMLPPM es imposible realizar un acceso aleatorio a un documento en concreto puesto que es adaptativo y necesita descomprimir todos los documentos que preceden al deseado. Con todo lo dicho, se puede descartar el uso de XMLPPM en el escenario de las bases de datos textuales comprimidas.

Quizá uno de los problemas que se le pueden achacar a LZCS es la eficiencia en la etapa de compresión, que tiene complejidad cuadrática si se emplea la definición dada. Para solucionarlo se puede diseñar un algoritmo de compresión con complejidad lineal empleando un esquema de hashing.

En muchos escenarios se pueden añadir documentos a la colección, pero en ningún caso se eliminarán o modificarán. LZCS se puede adaptar con facilidad para permitir la inserción de nuevos documentos, pero se necesita más trabajo para permitir la modificación o el borrado de documentos. Asimismo, puede ser interesante diseñar esquemas de indexación que permitan realizar buscar con rapidez documentos que contengan algunos términos o subestructuras dadas, teniendo presente que la colección está comprimida.

El proceso clásico de recuperación de información considera los documentos como unidades mínimas de información que se indexan y recuperan como un todo. La moderna evolución del diseño y almacenamiento de documentos ha introducido desde hace tiempo representaciones más elaboradas de los documentos; estándares como SGML, HTML y actualmente XML son, por supuesto, una gran contribución en este campo. Estos estándares son la causa fundamental de la actual evolución hacía los modernos documentos electrónicos. En este contexto, la recuperación de documentos estructurados se refiere a la indexación y recuperación de la información según una estructura dada de documentos. Esto significa que los documentos ya no se consideran como entidades mínimas, sino como conjuntos de objetos interrelacionados que pueden recuperarse por separado; dada una consulta de recuperación, se puede recuperar el conjunto de componentes de un documento que sean más pertinentes para esa consulta.

La recuperación de información para documentos XML es un campo de trabajo que adquiere cada vez más auge. La utilización de la estructura de estos documentos, junto con su contenido, para dicha recuperación es una constante en todos los trabajos que se están desarrollando sobre el tema. En este sentido, se ha propuesto un esquema de indexación capaz de recuperar elementos estructurales denominado "índice invertido con

direccionamiento a elementos de estructura" cuyas prestaciones se sitúan entre un índice invertido con direccionamiento a palabra y uno con direccionamiento a documento.

Además también se ha desarrollado un modelo de recuperación de información que toma como base (y considera) dos ideas que han demostrado ser válidas durante el pesado de los elementos estructurales: la densidad de los términos de la consulta y su proximidad. El modelo propuesto tiene en cuenta ambas ideas, sin embargo, respecto a la proximidad de los términos de la pregunta sólo pondera si éstos aparecen o no en los mismos elementos de estructura y cómo están próximos dichos términos dentro de los elementos de estructura. Con esta aproximación se obtiene una clasificación de los nodos que no se ajustará de una manera explícita ni a un enfoque puramente booleano ni vectorial pues, en algunas situaciones, se pueden encontrar nodos a los que les falta algún término de la consulta mejor clasificados que otros que contienen todos los términos de la consulta dependiendo de la cantidad y proporción de términos de la consulta presentes de dicho nodo. La idea que se ha perseguido es la de recuperar aquellos elementos que puedan ser más interesantes al usuario desde el punto de vista de la "concentración" de la información, es decir, se supone que a un usuario le interesará recuperar aquellos documentos (o partes de los mismos) en los que la información solicitada (o parte de ella) a través de la consulta se encuentre de forma más concisa[1].

9.1. Líneas de trabajo futuro

Como resultado del trabajo realizado en esta tesis se han identificado una serie de cuestiones y temas en los que parece interesante profundizar en el futuro. Por un lado, SCMWBH es un proptotipo con una implementación básica y se pueden realizar una serie de mejoras que lo hagan más competitivo. Por ejemplo, se puede afinar la técnica que se emplea para fusionar los modelos pues se ha comprobado que, en general, la predicciones realizadas son aproximadamente un 98-99 % del tamaño real, por lo que se puede añadir un valor medio a cada predicción que permita aproximarse con mayor exactitud al tamaño real.

En lo referente a la aplicación del modelo junto con bloques de texto no ha resultado muy prometedora, no obstante su efecto se puede probar con otro tipo de colecciones en las que la distribución de las palabras en el texto de cada elemento estructural varíe en diferentes partes de las mismas. Y en relación con el estudio del método en sí, se debe estudiar más en profundidad la relación que existe entre el tipo y densidad de la estructura y las mejoras obtenidas con el método, ya que su éxito está basado en una suposición semántica y podría ser interesante comprobar cómo funciona con otras colecciones de texto.

Respecto a la versión adaptativa de SCM, actualmente se está trabajando en la utilización de palabras y separadores como base del alfabeto fuente de símbolos de entrada para el modelo PPM, pues las palabras modelan la entropía del texto mejor que los caracteres [BCW90]. Por ejemplo, un codificador semiadaptativo tipo Huffman que considere caracteres como símbolos del alfabeto fuente obtendrá, por término medio, un fichero comprimido cuyo tamaño es aproximadamente el 60 % del tamaño del fichero original cuando se procesan textos escritos en lenguaje natural. Un codificador semiadaptativo tipo Huffman que considere palabras como símbolos del alfabeto fuente obtendrá, también por término medio, un fichero comprimido cuyo tamaño rondará el 25 % del tamaño original [ZMNBY00]. Otro ejemplo es el algoritmo WLZW (Ziv-Lempel sobre palabras) [BSTW86, DPS99].

[1] Es decir, que tiene brevedad y economía de medios en el modo de expresar un concepto con exactitud.

Se puede destacar que SCMPPM es equivalente a tener un contexto extra constituido por el último elemento de estructura procesado, el cual precede al primer caracter del contexto actual. Bajo este punto de vista se pueden realizar diversas generalizaciones, como utilizar más de un contexto estructural (por ejemplo, los dos últimos elementos de estructura que contengan el texto actual) e intercalar el contexto estructural con el del carácter en orden diferente al actual (primero el contexto estructural y luego el contexto del carácter).

Por otro lado, el trabajo con esquemas de compresión de tipo LZ ha planteado la idea de estudiar la posibilidad de su utilización conjuntamente con técnicas de búsqueda de patrones permitiendo errores. La idea principal es utilizar un esquema LZ77 de manera que al buscar la cadena en la ventana se permita que esta tenga una o varias diferencias respecto con la de la posición actual, de manera que se realiza una sustitución más grande pero será necesario codificar en qué posiciones se han producido dichos errores y cuáles son para recuperar la cadena original en el proceso de descodificación. Será necesario comprobar que esta idea obtiene mejoras respecto al esquema básico, además se deberán establecer heurísticas que permitan seleccionar y codificar las diferencias entre las cadenas.

Finalmente, en lo referente a la indexación y recuperación, el uso del índice invertido con direccionamiento a elementos estructurales sugiere la posibilidad de realizar búsquedas por proximidad teniendo en cuenta el aspecto semántico de los textos. Respecto al modelo propuesto, se debe realizar un análisis más profundo y efectuar algo más que unas pocas "consultas experimentales" con colecciones de prueba más grandes y estándar, como INEX[2].

Puede ser interesante adaptar el modelo para obtener una relación de los "mejores puntos de entrada" de los documentos en lugar de su clasificación general. Esta posibilidad ha surgido a raíz del estudio de los factores que el modelo considera a la hora de obtener los pesos de los nodos por un lado y de los resultados obtenidos por otro. Recordemos que los "mejores puntos de entrada" son aquellas partes o lugares del documento, devueltas como respuesta a una consulta, en las se sugiere que un usuario comience a leer el documento y el hecho que el modelo prime las partes del documento en las que existe una mayor concentración de los términos de la consulta induce a pensar que la aplicación de dicho modelo para tal cometido obtenga resultados satisfactorios.

[2]"Initiative for the Evaluation of XML Retrieval": `http://qmir.dcs.qmul.ac.uk/INEX`

Apéndice A

Razones obtenidas por SCMPPM variando la constante k

A.1. WSJ

	SCM 1.1		SCM 1.2	
Original	Compr.	Razón	Compr.	Razón
$k = 3$				
1.221.659	358.819	29,37 %	354.222	29,00 %
5.516.592	1.571.275	28,48 %	1.544.404	28,00 %
10.510.481	2.972.950	28,29 %	2.918.416	27,77 %
21.235.547	5.988.508	28,20 %	5.873.237	27,66 %
42.113.697	12.011.428	28,52 %	11.773.609	27,96 %
62.963.963	17.567.135	27,90 %	17.213.930	27,34 %
104.942.941	29.438.340	28,05 %	28.816.791	27,46 %
$k = 4$				
1.221.659	316.587	25,91 %	315.276	25,81 %
5.516.592	1.339.255	24,28 %	1.325.542	24,03 %
10.510.481	2.504.614	23,83 %	2.473.412	23,53 %
21.235.547	4.996.208	23,53 %	4.923.926	23,19 %
42.113.697	9.950.950	23,63 %	9.791.886	23,25 %
62.963.963	14.514.035	23,05 %	14.269.246	22,66 %
104.942.941	24.211.775	23,07 %	23.758.911	22,64 %
$k = 5$				
1.221.659	317.418	25,98 %	318.045	26,03 %
5.516.592	1.291.923	23,41 %	1.286.742	23,32 %
10.510.481	2.376.155	22,61 %	2.360.587	22,46 %
21.235.547	4.664.372	21,96 %	4.609.569	21,72 %
42.113.697	9.044.291	21,48 %	8.936.802	21,22 %
62.963.963	13.384.987	21,26 %	13.204.020	20,97 %
104.942.941	21.945.890	20,91 %	21.585.711	20,56 %
$k = 6$				
1.221.659	307.993	25,21 %	310.312	25,40 %
5.516.592	1.252.996	22,71 %	1.255.615	22,76 %

Continúa ↪

↩ Continúa

	SCM 1.1		SCM 1.2	
Original	Compr.	Razón	Compr.	Razón
10.510.481	2.299.260	21,87 %	2.298.500	21,86 %
21.235.547	4.497.007	21,17 %	4.482.002	21,10 %
42.113.697	8.792.296	20,87 %	8.738.817	20,75 %
62.963.963	12.817.945	20,35 %	12.716.647	20,19 %
104.942.941	20.932.454	19,94 %	20.699.120	19,72 %
$k = 7$				
1.221.659	311.198	25,47 %	314.463	25,74 %
5.516.592	1.258.985	22,82 %	1.267.794	22,98 %
10.510.481	2.300.551	21,88 %	2.312.315	22,00 %
21.235.547	4.474.022	21,06 %	4.484.858	21,19 %
42.113.697	8.690.443	20,63 %	8.688.375	20,63 %
62.963.963	12.620.795	20,04 %	12.593.098	20,00 %
104.942.941	20.496.910	19,53 %	20.379.274	19,41 %
$k = 8$				
1.221.659	314.282	25,72 %	318.397	26,06 %
5.516.592	1.271.415	23,04 %	1.285.969	23,31 %
10.510.481	2.319.151	22,06 %	2.343.571	22,29 %
21.235.547	4.495.912	21,17 %	4.534.922	21,35 %
42.113.697	8.694.891	20,64 %	8.752.624	20,78 %
62.963.963	12.591.704	19,99 %	12.653.486	20,09 %
104.942.941	20.358.852	19,39 %	20.392.965	19,43 %
$k = 9$				
1.221.659	317.257	25,96 %	321.793	26,34 %
5.516.592	1.286.742	23,32 %	1.304.663	23,64 %
10.510.481	2.347.215	22,33 %	2.379.507	22,63 %
21.235.547	4.546.547	21,41 %	4.604.310	21,68 %
42.113.697	8.775.317	20,83 %	8.876.001	21,07 %
62.963.963	12.687.973	20,15 %	12.816.319	20,35 %
104.942.941	20.460.578	19,49 %	20.609.664	19,63 %

Tabla A.1: Sizes and compression ratios for WSJ collections

A.2. ZIFF

	SCM 1.1		SCM 1.2	
Original	Compr.	Razón	Compr.	Razón
$k = 3$				
1.021.882	274.931	26,90 %	265.224	25,95 %
6.083.389	1.683.030	27,67 %	1.625.257	26,72 %
11.164.171	3.107.935	27,84 %	3.001.468	26,88 %
21.306.059	5.929.474	27,83 %	5.723.628	26,86 %
42.659.558	11.843.516	27,76 %	11.425.013	26,78 %

Continúa ↪

↩ Continúa

	SCM 1.1		SCM 1.2	
Original	Compr.	Razón	Compr.	Razón
62.966.279	17.448.511	27,71 %	16.825.296	26,72 %
105.709.264	29.354.113	27,77 %	28.305.988	26,78 %
$k = 4$				
1.021.882	242.054	23,69 %	234.571	22,95 %
6.083.389	1.444.200	23,74 %	1.396.533	22,96 %
11.164.171	2.652.841	23,76 %	2.563.122	22,96 %
21.306.059	5.039.130	23,65 %	4.860.413	22,81 %
42.659.558	10.006.778	23,46 %	9.634.177	22,58 %
62.966.279	14.694.697	23,34 %	14.131.438	22,44 %
105.709.264	24.682.112	23,35 %	23.722.797	22,44 %
$k = 5$				
1.021.882	238.105	23,30 %	231.689	22,67 %
6.083.389	1.370.269	22,52 %	1.329.657	21,85 %
11.164.171	2.487.704	22,29 %	2.389.978	21,41 %
21.306.059	4.680.723	21,97 %	4.523.838	21,23 %
42.659.558	9.184.051	21,52 %	8.849.495	20,74 %
62.966.279	13.398.491	21,27 %	12.884.870	20,46 %
105.709.264	22.373.443	21,16 %	21.485.900	20,32 %
$k = 6$				
1.021.882	231.832	22,68 %	226.491	22,16 %
6.083.389	1.340.453	22,03 %	1.306.515	21,47 %
11.164.171	2.433.377	21,79 %	2.368.788	21,21 %
21.306.059	4.567.454	21,43 %	4.433.994	20,81 %
42.659.558	8.931.809	20,93 %	8.643.937	20,26 %
62.966.279	13.004.073	20,65 %	12.557.867	19,94 %
105.709.264	21.658.537	20,48 %	20.877.329	19,74 %
$k = 7$				
1.021.882	232.532	22,75 %	228.011	22,31 %
6.083.389	1.339.794	22,02 %	1.312.620	21,57 %
11.164.171	2.424.533	21,71 %	2.372.704	21,25 %
21.306.059	4.530.997	21,26 %	4.422.727	20,75 %
42.659.558	8.808.637	20,64 %	8.572.638	20,09 %
62.966.279	12.777.471	20,29 %	12.407.268	19,70 %
105.709.264	21.189.260	20,04 %	20.533.362	19,42 %
$k = 8$				
1.021.882	233.569	22,85 %	230.089	22,51 %
6.083.389	1.348.845	22,17 %	1.328.037	21,83 %
11.164.171	2.437.926	21,83 %	2.398.362	21,48 %
21.306.059	4.543.919	21,32 %	4.460.431	20,93 %
42.659.558	8.796.750	20,62 %	8.614.154	20,19 %
62.966.279	12.723.903	20,20 %	12.435.544	19,74 %
105.709.264	21.020.710	19,88 %	20.504.458	19,39 %
$k = 9$				
1.021.882	234.956	22,99 %	232.055	22,70 %
6.083.389	1.361.778	22,38 %	1.345.059	22,11 %

Continúa ↪

↩ Continúa

	SCM 1.1		SCM 1.2	
Original	Compr.	Razón	Compr.	Razón
11.164.171	2.461.153	22,04 %	2.429.508	21,76 %
21.306.059	4.583.691	21,51 %	4.516.745	21,19 %
42.659.558	8.855.121	20,75 %	8.708.795	20,41 %
62.966.279	12.786.778	20,30 %	12.553.878	19,93 %
105.709.264	21.072.530	19,93 %	20.651.891	19,53 %

Tabla A.2: Sizes and compression ratios for ZIFF collections

A.3. AP

	SCM 1.1		SCM 1.2	
Original	Compr.	Razón	Compr.	Razón
$k = 3$				
1.185.968	359.020	30,27 %	354.895	29,92 %
5.805.776	1.698.951	29,26 %	1.673.618	28,83 %
10.469.592	3.041.663	29,05 %	2.993.253	28,59 %
21.219.693	6.129.940	28,89 %	6.026.064	28,40 %
42.523.572	12.272.060	28,86 %	12.057.967	28,36 %
63.343.648	18.264.370	28,83 %	17.940.515	28,32 %
105.018.927	30.316.540	28,87 %	29.769.890	28,35 %
$k = 4$				
1.185.968	319.138	26,91 %	316.662	26,70 %
5.805.776	1.455.211	25,06 %	1.436.485	24,74 %
10.469.592	2.577.725	24,62 %	2.539.528	24,26 %
21.219.693	5.144.440	24,24 %	5.057.261	23,83 %
42.523.572	10.236.231	24,07 %	10.047.875	23,63 %
63.343.648	15.201.370	24,00 %	14.910.083	23,54 %
105.018.927	25.189.378	23,99 %	24.684.401	23,50 %
$k = 5$				
1.185.968	309.111	26,06 %	307.953	25,97 %
5.805.776	1.358.422	23,40 %	1.346.689	23,19 %
10.469.592	2.375.626	22,76 %	2.349.236	22,44 %
21.219.693	4.675.626	22,03 %	4.609.569	21,72 %
42.523.572	9.203.654	21,64 %	9.053.350	21,29 %
63.343.648	13.599.079	21,46 %	13.360.777	21,09 %
105.018.927	22.397.857	21,32 %	21.970.522	20,92 %
$k = 6$				
1.185.968	308.634	26,02 %	308.674	26,02 %
5.805.776	1.345.142	23,16 %	1.340.055	23,08 %
10.469.592	2.342.534	22,37 %	2.327.175	22,22 %
21.219.693	4.582.231	21,59 %	4.537.074	21,38 %
42.523.572	8.963.369	21,07 %	8.851.285	20,81 %

Continúa ↪

↩ Continúa

	SCM 1.1		SCM 1.2	
Original	Compr.	Razón	Compr.	Razón
63.343.648	13.198.923	20,83 %	13.013.953	20,54 %
105.018.927	21.651.971	20,61 %	21.305.255	20,28 %
$k = 7$				
1.185.968	311.147	26,23 %	312.460	26,34 %
5.805.776	1.344.670	23,16 %	1.347.679	23,21 %
10.469.592	2.331.826	22,27 %	2.331.391	22,26 %
21.219.693	4.532.667	21,36 %	4.517.439	21,28 %
42.523.572	8.807.487	20,71 %	8.754.455	20,58 %
63.343.648	12.918.520	20,39 %	12.819.765	20,23 %
105.018.927	21.089.917	20,08 %	20.881.860	19,88 %
$k = 8$				
1.185.968	313.777	26,45 %	315.988	26,64 %
5.805.776	1.352.262	23,29 %	1.361.926	23,45 %
10.469.592	2.340.020	22,35 %	2.352.861	22,47 %
21.219.693	4.531.103	21,35 %	4.543.820	21,41 %
42.523.572	8.763.877	20,60 %	8.768.166	20,61 %
63.343.648	12.818.548	20,23 %	12.804.430	20,21 %
105.018.927	20.847.056	19,85 %	20.777.658	19,78 %
$k = 9$				
1.185.968	316.215	26,66 %	319.109	26,90 %
5.805.776	1.363.891	23,49 %	1.378.465	23,74 %
10.469.592	2.358.875	22,53 %	2.381.520	22,74 %
21.219.693	4.559.859	21,48 %	4.594.595	21,65 %
42.523.572	8.797.645	20,68 %	8.849.641	20,81 %
63.343.648	12.844.635	20,27 %	12.903.912	20,37 %
105.018.927	20.833.999	19,83 %	20.888.577	19,89 %

Tabla A.3: Sizes and compression ratios for AP collections

Bibliografía

[Abr63] N. Abramson. *Information Theory and Coding.* McGraw-Hill, 1963.

[AFN04] J. Adiego, P. de la Fuente, and G. Navarro. Merging prediction by partial matching with structural contexts model. In *Data Compression Conference (DCC'04), Snowbird, Utah, USA*, page 522. IEEE Computer Society TCC, March 2004.

[AFV+02] J. Adiego, P. de la Fuente, J. Vegas, M. Villarroel, and A. Pedrero. Bibliotecas digitales con documentos comprimidos: una arquitectura. In *Terceras Jornadas de Bibliotecas Digitales (JBIDI 2002), El Escorial, Madrid, España*, pages 133–142, Noviembre 2002.

[AFVV02a] J. Adiego, P. de la Fuente, J. Vegas, and M. Villarroel. Una técnica para compresión y acceso de documentos estructurados. *Novática, Número monográfico: Recuperación de información y la Web*, 157:34–40, Mayo/Junio 2002.

[AFVV02b] J. Adiego, P. de la Fuente, J. Vegas, and M. A. Villarroel. System for compressing and retrieving structured documents. *Upgrade - The European Online Magazine for the IT Professional - Information Retrieval and the Web*, III(3):62–69, June 2002.

[Alb95] S. Albers. Improved randomized on-line algorithms for the list update problem. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms*, pages 412–419, New York, NY, USA, January 1995. ACM Press.

[AM96] S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium*, volume 1099 of *Lecture Notes in Computer Science*, pages 514–525, Paderborn, Germany, 8–12 July 1996. Springer-Verlag.

[ANF03a] J. Adiego, G. Navarro, and P. de la Fuente. Compressing semistructured text databases. In *25th European Conference on Information Retrieval Research (ECIR'03), Pisa, Italia*, pages 153–167. Spring-Verlag, LNCS 2633, April 2003.

[ANF03b] J. Adiego, G. Navarro, and P. de la Fuente. SCM: Structural contexts model for improving compression in semistructured text. In *10th International Symposium on String Processing and Information Retrieval (SPIRE 2003), Manaus, Brazil*, pages 153–167. Spring-Verlag, LNCS 2857, October 2003.

[ANF04] J. Adiego, G. Navarro, and P. de la Fuente. Lempel-ziv compression of structured text. In *Data Compression Conference (DCC'04), Snowbird, Utah, USA*, pages 112–121. IEEE Computer Society TCC, March 2004.

[AWG+] M. Adler, R. Wales, J.-L. Gailly, K. U. Rommel, and I. Mandrichenko. Fuentes de Infozip Zip 2.x / Unzip 5.x.

[BCB92] B. T. Bartell, G. W. Cottrell, and R. K. Belew. Latent semantic indexing is an optimal special case of multidimensional scaling. In *Proceedings of the 15th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 161–167. ACM Press, 1992.

[BCW90] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, N.J., 1990.

[Ber] D. J. Bernstein. Yabba, paquete con código fuente y documentación sobre "Y coding".

[BK00] B. Balkenhol and S. Kurtz. Universal data compression based on the burrows-wheeler transformation: Theory and practice. *IEEE Transactions on Computers*, 49(10):1043–1053, 2000.

[BKS99] B. Balkenhol, S. Kurtz, and Y. M. Shtarkov. Modifications of the burrows and wheeler data compression algorithm. In *Data Compression Conference.* IEEE Computer Society TCC, 1999.

[BM77] R. S. Boyer and J. S. Moore. A fast string search algorithm. *Communications of the ACM*, 20(10):726–772, 1977.

[BNB+95] E. Barbosa, G. Navarro, R. Baeza-Yates, C. Peleberg, and N. Ziviani. Optimized binary search and text retrieval. In *Proc. of European Symposium on Algorithms*, LNCS 979, pages 311–326. Springer–Velarg, 1995.

[Bos] J. Bosak. The shakespeare plays marked up in xml. http://www.ibiblio.org/bosak/.

[BPSM00] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0.* W3C Consortium, second edition, 2000. http://www.w3.org/TR/REC-xml.

[Bra00] N. Bradley. *The XML Companion.* Addison–Wesley, second edition, 2000.

[BS97] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, New Orleans, Louisiana, 5–7 January 1997.

[BSTW86] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.

[Bun97] S. Bunton. Semantically motivated improvements for PPM variants. *The Computer Journal*, 40(2/3):76–94, 1997.

[Bur92a] F. Burkowski. An algebra for hierarchically organized text-dominated databases. *Information Processing & Management*, 28(3):333–348, 1992.

[Bur92b] F. Burkowski. Retrieval activities in a database consisting of heterogeneous collections of structured text. In *Proc. of the 15th Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 112–125, 1992.

[BVNF98] R. Baeza-Yates, J. Vegas, G. Navarro, and P. de la Fuente. A model of visual query language for structured text. In *Proceedings of SPIRE'98*, pages 7–13. IEEE Computer Society, Sep 1998.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[BWC89] T. C. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 4(21):557–591, 1989.

[BYBZ96] R Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

[BYG92] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, October 1992.

[BYN96] R. Baeza-Yates and G. Navarro. Integrating contents and structure in text retrieval. *ACM SIGMOD Record*, 25(1):67–79, March 1996.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley-Longman, may 1999.

[Cal94] J. P. Callan. Passage-Level Evidence in Document Retrieval. In W. Bruce. Croft and C.J. van Rijsbergen, editors, *Proceedings of the Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 302–310, Dublin, Ireland, July 1994. Spring-Verlag.

[CH84] G. V. Cormack and R. N. Horspool. Algoritms for adaptive huffman codes. *Information Processing Letters*, 18(3):159–165, 1984.

[Che01] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proc. Data Compression Conference (DCC 2001)*, pages 163–, 2001.

[Chi01] Y. Chiaramella. Information retrieval and structured documents. *Third European Summer-School, ESSIR 2000, Varenna, Italy, September 11-15, 2000.*, 1980:286–309, January 2001.

[CLvRC91] F. Crestani, M. Lalmas, C. J. van Rijsbergen, and L. Campbell. Is this document relevant?... probably. a survey of probabilistic models in information retrieval. *ACM Computing Surveys*, 30(4):528–552, 1991.

[CMM+03] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. Searching xml documents via xml fragments. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and developm ent in informaion retrieval*, pages 151–158. ACM Press, 2003.

[Con73] J. B. Connell. A huffman–shannon–fano code. *Proceedings IEEE*, 7(61):1046–1047, July 1973.

[CT91] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley, New-York, 1991.

[CT97] J. G. Clearly and W. J. Teahan. Unbounded length contexts for ppm. *The Computer Journal*, 40(2/3):67–75, 1997.

[CTW95] J. G. Clearly, W. J. Teahan, and I. H. Witten. Unbounded length contexts for ppm. In *Data Compression Conference*, pages 52–61. IEEE Computer Society TCC, 1995.

[CW84a] J. G. Clearly and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, April 1984.

[CW84b] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, April 1984.

[CWC03] H. Cui, J.-R. Wen, and J.-R. Chua. Hierarchical Indexing and Flexible Element Retrieval for Structured Document. In *Proc. European Conference on Information Retrieval (ECIR'03)*, LNCS 2633, pages 73–87. Springer, 2003.

[DPS99] J. Dvorský, J. Pokorný, and V. Snásel. Word-based compression methods and indexing for text retrieval systems. In *Advances in Databases and Information Systems, Third East European Conference, ADBIS'99, Maribor, Slovenia, September 13-16, 1999, Proceedings*, volume 1691 of *Lecture Notes in Computer Science*, pages 75–84. Springer, 1999.

[Dro02] A. Drozdek. *Elements of Data Compression*. Thomson Learning, first edition, 2002.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, 1975.

[Fal73] N. Faller. An adaptive system for data compression. In *Conference Record of the 7th IEEE Asilomar Conference on Circuits, Systems and Computers*, pages 593–597, San Francisco, 1973. IEEE Press.

[Fan49] R.M. Fano. The transmission of information. Technical Report 65, Research Laboratory of Electronics, MIT, Cambridge, MA, 1949.

[FC87] C. Faloutsos and S. Christodoulakis. Description and performance analysis of signature file methods. *ACM Transactions on Information Systems*, 5(3):237–257, 1987.

[FC88] C. Faloutsos and R. Chan. Text access methods for optical and large magnetic disks: design and performance comparision. In *VLDB'88*, pages 280–293, Los Angeles, CA, USA, 1988.

[FDD+88] G. W. Furnas, S. Deerwester, S. T. Dumais, T. K. Landauer, R. A. Harshman, L. A. Streeter, and K. E. Lochbaum. Information retrieval using a singular value decomposition model of latent semantic structure. In *Proceedings of the 11th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 465–480. ACM Press, 1988.

[Fen96] P. Fenwick. Block-sorting text compression — final report, 1996.

[FG89] E. Fiala and D. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, Apr 1989.

[Gal78] R. G. Gallager. Variations on a theme of huffman. *IEEE Transactions on Information Theory*, IT-24(6):668–674, 1978.

[GBYS92] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In B. Frakes and R. Baeza-Yates, editors, *Information Retrieval Data Structures & Algorithms*. Prentice-Hall, 1992.

[GIS96] D. Ghazfan, M. Indrawan, and B. Srinivasan. Toward meaningful bayesian networks for information retrieval systems. In *Proceedings of the IPMU'96 Conference*, pages 841–846, 1996.

[Gol66] S. W. Golomb. Run-lenght encodings. *IEEE Transactions on Information Theory*, IT-12(3):399–401, 1966.

[Gon87] G. Gonnet. Examples of pat applied to the oxford english dictionary. Technical Report OED-87-02, UW Centre for the New OED and Text Research. University of Waterloo, 1987.

[GS02a] T. Grabs and H. J. Schek. Eth zürich at inex: Flexible information retrieval from xml with powerbd-xml. In *Proceedings of the First Workshop of the INitiative for the Evaluation of XML Retrieval (INEX)*, pages 141–149, December 2002.

[GS02b] T. Grabs and H. J. Schek. Generating vector spaces on-the-fly for flexible xml retrieval. In *XML and Information Retrieval Workshop. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Tampere, Finland, August*, 2002.

[Gua80] M. Guauzzo. A general miminum-redundancy source-coding algorithm. *IEEE Transactions on Information Theory*, IT-26(1):15–25, January 1980.

[Har95] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[Hea78] H. S. Heaps. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, 1978.

[Hel96] G. Held. *Data and Image Compression*. Willey, 1996.

[HL90] D. S. Hirschberg and D. A. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 4(33):449–459, 1990.

[HMQS96] G. Haider, C. Magnusson-Sjöberg, G. Quirchmayr, and V. Sebald. The comparative part of the corpus legis project - using SGML for intelligent information retrieval of legal documents. In *EXPERSYS-96, Artificial Intelligence Applications*, pages 181–186, 1996.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Engineers*, 40(9):1098–1101, 1952.

[HV92] P. G. Howard and J. S. Vitter. Practical implementations of arithmetic coding. In J. A. Storer, editor, *Image and Text Compression*, pages 85–112. Kluwer Academic, Norwell, Massachusetts, 1992.

[HV94] P. G. Howard and J. S. Vitter. Arithmetic coding for data compression. Technical Report DUKE–TR–1994–09, Duke University, 1994.

[ISO86] ISO 8879:1986. Standard Generalized Markup Language (SGML). *Information Processing - Text and Office System*, Oct 1986.

[Jak85] M. Jakobsson. Compression of character strings by an adaptive dictionary. *Tidskrift for Informations Behandeling, Denmark*, 25(4), 1985.

[JN84] N. Jayant and P.Ñoll. *Digital Coding of Waveforms*. Prentice–Hall, 1984.

[Kar61] J. Karush. A simple proof of an inequality of mcmillan. *Institute of Radio Engineers Transactions on Information Theory*, IT-7(2):118, April 1961.

[KLR02] G. Kazai, M. Lalmas, and T. Rölleke. Focussed structured document retrieval. In *9th International Symposium on String Processing and Information Retrieval (SPIRE 2002), Lisbon, Portugal*, pages 241–247, September 2002.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[Knu73] D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching*. Addison–Wesley, Reading, Massachusetts, 1973.

[Knu85] D. Knuth. Dynamic huffman coding. *Journal of Algorithms*, 6(2):163–180, 1985.

[Kot02] E. Kotsakis. Structured information retrieval in xml documents. In *Proceedings of the seventeenth ACM Symposium on Applied Computing (SAC 2002), Madrid, Spain*, pages 663–667, March 2002.

[Kou95] W. Kou. *Digital Image Compression: Algorithms and Standards*. Kluwer Academic Publishers, 1995.

[Kra49] L. G. Kraft. A device for quantizing, grouping and coding amplitude modulated pulses. Master's thesis, MIT, Canbridge, Massachusetts, 1949.

[KU94] J. Karkkäinen and E. Ukkonen. Two and higher dimensional pattern matching in optimal expected time. In *Proc. of ACM-SIAM Symposium on Discrete Algorithms*, pages 715–723, San Francisco, USA, 1994.

[Kur99] S. Kurtz. Reducing the space requirement of suffix trees. *SOFTPREX: Software–Practice and Experience*, 29, 1999.

[Kwo89] K. L. Kwok. A neural network for probabilistic information retrieval. In *Proceedings of the 12th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–30. ACM Press, 1989.

[Kwo90] K. L. Kwok. Experiments with a component theory of probabilistic information retrieval based on single terms as documents components. *ACM Transactions on Information Systems*, 8(4):363–386, October 1990.

[Kwo95] K. L. Kwok. A network approach to probabilistic information retrieval. *ACM Transactions on Information Systems*, 13(3):324–353, July 1995.

[Lan84] G. G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, March 1984.

[Lee94] J. H. Lee. Properties of extended boolean models in information retrieval. In *Proceedings of the 17th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 182–190. ACM Press, 1994.

[LKKL93] J. H. Lee, W. Y. Kim, M. H. Kim, and Y. J. Lee. On the evaluation of boolean operators in the extended boolean retrieval framework. In *Proceedings of the 16th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 291–297. ACM Press, 1993.

[LS99] N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999.

[LS00] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, 2000.

[LYYB96] Y. K. Lee, S.-J. Yoo, K. Yoon, and P. B. Berra. Index structures for structured documents. In *ACM First International Conference on Digital Libraries*, pages 91–99. ACM, 1996.

[Man99] G. Manzini. An analysis of the burrows-wheeler transform. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[McC76] E. M. McCreight. A space-economic suffix tree construction algorithm. *Jrnl. A.C.M.*, 23(2):262–272, April 1976.

[McM56] B. McMillan. Two inequalities implied by unique decipherability. *Institute of Radio Engineers Transactions on Information Theory*, IT-2:115–116, December 1956.

[MK95] A. Moffat and J. Katajanen. In-place calculation of minimum-redundancy codes. In *Proc. Workshop on Algorithms and Data Structures*, LNCS 955, pages 393–402. Springer–Velarg, August 1995.

[MLP99] R. L. Milidiú, E. S. Laber, and A. A. Pessoa. Bounding the compression loss of the fgk algorithm. *Journal of Algorithms*, 32(2):195–211, 1999.

[MM93] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[MNC95] A. Moffat, R. Neal, and J. G. Clearly. Arithmetic coding revisited. In *Data Compression Conference*. IEEE Computer Society TCC, 1995.

[MNW98] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, July 1998.

[MNZ97] E. S. de Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In Carleton University Press International Informatics Series, editor, *Proceedings of the Fourth South American Workshop on String Processing*, pages 95–111, 1997.

[MNZB00] E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[Mof89] A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.

[Mof90] A. Moffat. Implementing the ppm data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, 1990.

[Mou99] E. S. de Moura. *Compressão de dados aplicada a sistemas de recuperação de informação.* PhD thesis, Univ. Federal de Minas Gerais. Belo Horizonte, 1999.

[MS94] E. Mittendorf and P. Schäuble. Document and Passage Retrieval Based on Hidden Markov Models. In *Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, Dublin, Ireland, July 1994.

[MSDWZ93] A. Moffat, R. Sacks-Davis, R. Wilkinson, and J. Zobel. Retrieval of partial documents. In *Text REtrieval Conference*, pages 181–190, 1993.

[MSWB94] A. Moffat, N. Sharman, I. H. Witten, and T. C. Bell. An empirical evaluation of coding methods for multi-symbol alphabets. *Information Processing & Management*, 30(6):791–804, November 1994.

[MT97] A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997.

[MT02] A. Moffat and A. Turpin. *Compression and Coding Algorithms.* Kluwer Academic Publishers, first edition, 2002.

[MW85] V. Miller and V. M. Wegman. Variations on a theme by ziv and lempel. *Combinatorial Algorithms on Words*, 12:131–140, 1985.

[MW94] U. Manber and S. Wu. Glimpse: a tool to search through entire file systems. In *Proceedings Winter USENIX Conference*, pages 23–32, 1994.

[MW01] A. Moffat and R. Wan. RE-store: A system for compressing, browsing and searching large documents. In *Proc. 8th Intl. Symp. on String Processing and Information Retrieval (SPIRE 2001)*, pages 162–174, 2001.

[MZ00] E. S. de Moura and N. Ziviani. Construção eficiente de índices para bases de dados textuais. In *SBBD'2000 - XV Simpósio Brasileiro de Banco de Dados. João Pessoa. Paraíba*, Oct 2000.

[NBY95] G. Navarro and R. Baeza-Yates. A language for queries on structure and contents of textual databases. In *Proc. of the 18th Annual Int. ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 93–101, July 1995.

[NBY97a] G. Navarro and R. Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. *Information Systems*, 15(4):400–435, 1997.

[NBY97b] G. Navarro and R. Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. *ACM Transactions of Office and Information Systems*, 15(4):401–435, 1997.

[NMN+00] G. Navarro, E. S. de Moura, M. S. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[O'C75] J. O'Connor. Retrieval of answer-sentences and answer-figures from papers by text searching. *Information Retrieval & Management*, 11(5/7):155–164, 1975.

[O'C80] J. O'Connor. Answer-passage retrieval by text searching. *Journal of the American Society for Information Science*, 31(4):227–239, July 1980.

[OMK91] Y. Ogawa, T. Morita, and K. Kobayashi. A fuzzy document retrieval system using the keyword connection matrix and learning method. *Fuzzy Sets and Systems*, 39:163–179, 1991.

[Pas76] R. Pascoe. *Source coding algorithms for fast data compression*. PhD thesis, Stanford University, 1976.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kauffman Publishers, Inc., 1988.

[PKW98] PKWARE. APPNOTE.TXT de la distribución de PK-ZIP 1.x y 2.x, Jan 1998.

[PM93] W.B. Pennebaker and J.L. Mitchell. *JPEG: Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, first edition, 1993.

[PMLA88] W.B. Pennebaker, J.L. Mitchell, G.G. Langdon, and R.B. Arps. An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder. *IBM Journal of Research and Development*, 32(6):717–726, November 1988.

[Rad76] T. Radecki. Mathematical model of information retrieval system based on the concept of fuzzy thesaurus. *Information Processing & Management*, 12:313–318, 1976.

[Rad77] T. Radecki. Mathematical model of time-effective information retrieval system based on the theory of fuzzy sets. *Information Processing & Management*, 13:109–116, 1977.

[Rad79] T. Radecki. Fuzzy sets theoretical approach to document retrieval. *Information Processing & Management*, 15:247–259, 1979.

[RG98] V.G. Ruiz and I. García. Compresión de texto basada en un modelo probabilístico de orden 0 y un codificador de huffman. Technical Report I.R. Nº 1, Dpto. de Arquitectura de Computadores y Electrónica. Universidad de Almería, Spain, 1998.

[Ric79] R. F. Rice. Some practical universal noiseless coding techniques. Technical Report 79-22, Jet Propulsion Laboratory, Pasadena, California, March 1979.

[Ris76] J. Rissanen. Generalised kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198–203, May 1976.

[Ris79] J. J. Rissanen. Arithmetic coding as number representation. In *Acta. Polytech. Scandinavica*, volume Math 31, pages 44–51, 1979.

[Riv92] R. Rivest. The MD5 message-digest algorithm. RFC 1321. MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.

[RL79] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, March 1979.

[RL81] J. Rissanen and G. G. Langdon. Universal modeling and coding. *IEEE Transactions on Information Theory*, IT-27(1):12–23, January 1981.

[RNM96] B. Ribeiro-Neto and R. Muntz. A belief network model for ir. In *Proceedings of the 19th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 253–260. ACM Press, 1996.

[Rob82] M. G. Roberts. *Local Order Estimating Markovian Analysis for Noisless Source Coding and Authorship Identification.* PhD thesis, Stanford University, 1982.

[RPE81] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithms for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981.

[RSJ76] G. C. Robertson and K. Sparck-Jones. Relevance weighting of search terms. *Journal of the American Society for Information Sciences*, 27(3):129–146, 1976.

[RT87] T. Raita and J. Teuhola. Predictive text compression by hashing. In *Proceedings of the Tenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Storage/Retrieval Techniques I, pages 223–233, 1987.

[Rub79] F. Rubin. Arithmetic coding using fixed precision registers. *IEEE Transactions on Information Theory*, IT-25(6):672–675, November 1979.

[SB88] G. Salton and C. Buckley. Term-weighting approaches in automatic retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[Sch97] M. Schindler. A fast block-sorting algorithm for lossless data compression. In *Data Compression Conference.* IEEE Computer Society TCC, 1997.

[Sed90] R. Sedgewick. *Algorithms in C.* Addison–Wesley, Reading, Masschussetts, 1990.

[SFW83] G. Salton, E. A. Fox, and H. Wu. Extended boolean information retrieval. *Communications of the ACM*, 26(11):1022–1036, November 1983.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:398–403, July 1948.

[Sha51] C. E. Shannon. Prediction and entropy of printed english. *Bell Systems Technical Journal*, 30:50–64, 1951.

[SJ72] K. Sparck-Jones. A statistical interpretation of term specifity and its application to retrieval. *Journal of Documentation*, 28(1):11–20, 1972.

[SJ73] K. Sparck-Jones. A statistical interpretation of term specifity and its application to retrieval. *Information Storage and Retrieval*, 9(11):619–633, 1973.

[SK64] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7:166–169, 1964.

[SL68] G. Salton and M. E. Lesk. Computer evaluation of indexing and text processing. *Journal of the ACM*, 15(1):8–36, January 1968.

[SM83] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw Hill Book Co., New York, 1983.

[SM00] T. Schlieder and H. Meuss. Result ranking for structured queries against XML documents. In *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[SM02] T. Schlieder and H. Meuss. Querying and ranking XML documents. *JASIST*, 53(6):489–503, 2002.

[Sto88] J. A. Storer. *Data Compression: Methods and Theory.* Addison Wesley, Rockville, Md., 1988.

[Sun90] D. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[SY73] G. Salton and C. S. Yang. On the specification of term values in automatic indexing. *Journal of Documentation*, 29:351–372, 1973.

[TC90] H. Turtle and W. B. Croft. Inference networks for document retrieval. In *Proceedings of the 13th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1–24. ACM Press, 1990.

[TC91] H. Turtle and W. B. Croft. Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems*, 9(3):187–222, July 1991.

[Tea98] W. J. Teahan. *Modelling English Text.* PhD thesis, University of Waikato. New Zeland, 1998.

[TH02] P. Tolani and J. R. Haritsa. XGRIND: A query-friendly XML compressor. In *ICDE*, 2002.

[TMD+91] S. Thomas, J. McKie, S. Davis, K. Turkowski, J. Woods, J. Orost, and D. Mack. Fuentes de compress para UNIX 4.12, 1991.

[Ukk95] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, 1995. TR A-1993-1, Dept. Comp. Sci., U. of Helsinki, Finland.

[Veg99] J. Vegas. *Un Sistema de Recuperación de Información sobre Estructura y Contenido*. PhD thesis, Departamento de Informática, Universidad de Valladolid. Valladolid. Spain, 1999.

[VFC02] J. Vegas, P. de la Fuente, and F. Crestani. A graphical user interface for structured document retrieval. In *Advances in Information Retrieval, 24th BCS-IRSG European Colloquium on IR Research Proceedings*, volume 2291 of *Lecture Notes in Computer Science*. Springer, March 2002.

[Vit87] J. S. Vitter. Design and analysis of dinamic huffman coding. *Journal of the ACM*, 34(4):825–845, 1987.

[Vit89] J. S. Vitter. Algorithm 673: Dynamic huffman coding. *ACM Transactions on Mathematical Software*, 15(2):158–167, 1989.

[vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, 1979.

[WB91] I. H. Witten and T. C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, 1991.

[Wel84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[WH91] R. Wilkinson and P. Hingston. Using the cosine measure in a neural network for document retrieval. In *Proceedings of the 14th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 202–210. ACM Press, 1991.

[Wil91] R. N. Williams. Fuentes de LZRW1a a LZRW3a, 1991.

[Wil94] R. Wilkinson. Effective retrieval of structured documents. In *Research and Development in Information Retrieval*, pages 311–317, 1994.

[WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, Inc., second edition, 1999.

[WNB87] I. H. Witten, R. M. Neal, and J. G. Bell. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–541, June 1987.

[WZR87] S. K. M. Wong, W. Ziarko, and V. V. Raghanvan. On modeling of information retrieval concepts in vector spaces. *ACM Transactions on Database Systems*, 13(1):299–321, 1987.

[WZW85] S. K. M. Wong, W. Ziarko, and P. C. N. Wong. Generalized vector space model in information retrieval. In *Proceedings of the 8th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 18–25. ACM Press, 1985.

[Zip49] G. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison–Wesley, 1949.

[ZL77] J. Ziv and A. Lempel. An universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, 1978.

[ZM95] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. *Software - Practice and Experience*, 25(8):891–903, 1995.

[ZMNBY00] N. Ziviani, E. S. de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, November 2000.