Indexación y autoindexación comprimida de documentos como base de su procesado *

Nieves R. Brisaboa, Antonio Fariña, Susana Ladra, Ángeles S. Places, Eduardo Rodríguez

Laboratorio de Bases de Datos, Campus de Elviña s/n, A Coruña, España.
{brisaboa,fari,sladra,asplaces,erodriguezl}@udc.es

Resumen Tradicionalmente, las comunidades de investigación en Recuperación de Información (RI) y en Ingeniería Lingüística (IL) han desarrollado sus investigaciones utilizando habitualmente como base documentos no comprimidos indexados, en su caso, con índices clásicos como los índices invertidos. Por otro lado, la comunidad de investigación en estructuras de datos y algoritmos ha venido desarrollando compresores, índices y autoíndices de texto de gran potencia que, en general, son poco utilizados en las aplicaciones de Bibliotecas Digitales e Ingeniería Lingüística.
En este artículo tratamos de dar a conocer los más recientes avances en indexación y autoindexación comprimida de textos, señalando cómo su uso puede mejorar todas las tareas clásicas de RI e IL.
Además presentamos un modelo de encapsulación, que permitirá a las técnicas de RI e IL realizar el procesamiento de los documentos independientemente de su representación. La idea es que los documentos comprimidos y/o autoindexados, constituyan una capa base encapsulada con una interfaz bien definida, que permita invocar los algoritmos que, internamente, manejan las estructuras que almacenan los documentos.

1. Introducción y motivación

En los últimos años ha habido grandes avances en el ámbito de la compresión e indexación de textos en lenguaje natural. Las nuevas técnicas de compresión de textos [14,15,6] permiten, no sólo reducir su tamaño, sino que permiten procesar un texto comprimido mucho más rápido que la versión sin comprimir. En el caso de la indexación de textos, destaca la aparición de nuevas estructuras de indexación como los índices invertidos comprimidos [2,19] y la de los autoíndices, que dentro del propio índice mantienen una versión implícita del texto. Entre ellos, cabe señalar aquellos que indexan caracteres y permiten recuperar cualquier subcadena del texto indexado [16], o los orientados a palabras [4,7], más adecuados para el texto en lenguaje natural.

* Parcialmente financiado por: el "Ministerio de Ciencia e Innovación"(MICINN ref. TIN2009-14560-C03-02)

Sin embargo, estas técnicas no han sido todavía comúnmente adoptadas por otras disciplinas como la ingeniería lingüística (IL), donde generalmente se trabaja con colecciones de documentos sin comprimir, o la recuperación de información (RI) [19,22], centrada en la recuperación de documentos *relevantes* y que continúa apoyándose en estructuras de indexación tradicionales como los índices invertidos para realizar sus operaciones (cálculo de frecuencia inversa, ranking de documentos, etc.).

La motivación de este artículo es doble: por un lado, revisar el estado del arte en compresión e indexación; por otro lado, se propone un nuevo modelo que, a través de la definición de una serie de primitivas que podrán ser invocadas desde los procesos de IL o IR, permitirá aislar dichos procesos de la forma utilizada para representar la colección de documentos. La idea es permitir que dicha representación se comporte como una caja negra que permita, a través de una interfaz común, la invocación de una serie de primitivas (p. ej: obtener la frecuencia de una palabra, recuperar un trozo de la colección, etc.). De este modo, se pretende dar soporte para que las técnicas de IL y RI puedan aplicarse sobre esta nueva capa que oculta la forma en que la colección está almacenada. Es importante recalcar que pese a permitir realizar el procesamiento sobre la colección comprimida no se pretende simplemente reducir las necesidades de espacio, sino mejorar el rendimiento del sistema resultante.

En la Sección 2 se revisan técnicas de compresión adecuadas para la compresión de Bases de Datos textuales, típicamente junto a índices invertidos, que son descritos en la Sección 3. La siguiente sección se centra en los autoíndices orientados a palabras. La Sección 4 muestra el modelo de encapsulación propuesto, y finalmente presentamos nuestras conclusiones y trabajo futuro.

2. Compresión de textos orientada a Lenguaje Natural

En los últimos años se han desarrollado nuevas técnicas de compresión adecuadas para la compresión de Bases de Datos textuales, que no sólo permiten reducir su tamaño, sino también mejorar su rendimiento [15,6]. Estas técnicas deben permitir la realización de dos operaciones básicas: *i)* realizar búsquedas directas sobre el texto comprimido (buscando el patrón comprimido) que evitan la necesidad de descomprimir la colección para buscar un patrón; y *ii)* realizar descompresión aleatoria; esto es, acceder a cualquier posición del texto comprimido y comenzar la descompresión desde ese punto, sin necesidad de comenzar desde el principio.

Las técnicas de compresión clásicas como el Huffman clásico (orientado a caracteres) [12] cuyo ratio de compresión es muy pobre (60 %), o los de la familia Lempel-Ziv [20,21], que permiten buscar dentro del texto comprimido de forma eficiente, no son adecuadas para manejar colecciones de texto.

En 1989, Moffat [14] propone utilizar palabras, en lugar de caracteres como los símbolos a comprimir. Este hecho abre la puerta a la integración de indexación y compresión debido a que las palabras son los elementos básicos en ambos casos. Además, debido a que la distribución de frecuencias de las pal-

abras en un texto es mucho más sesgada que la de los caracteres, un compresor estadístico semi-estático como el Huffman basado en palabras y orientado a bits [18] es capaz de obtener ratios de compresión próximos al 25 % (frente al 60 % del Huffman clásico). Los compresores semi-estáticos realizan dos pasadas sobre el texto a comprimir: en la primera pasada, obtienen un modelo del texto (el vocabulario con las distintas palabras existentes y su frecuencia) que es utilizado para asignar códigos más cortos a las palabras más frecuentes. En la segunda pasada, sustituyen cada palabra por su código. Dado que cada palabra está representada siempre por el mismo código dentro del texto comprimido, estas técnicas semi-estáticas permiten realizar sobre el texto comprimido las mismas operaciones que pueden realizarse sobre texto plano.

En [15] se presentaron dos nuevas técnicas basadas en Huffman, pero orientadas a byte en lugar de a bit para agilizar la descompresión. La primera técnica *Plain Huffman* utiliza árboles Huffman 256-arios y obtiene ratios de compresión próximos al 30 %. La otra, *Tagged Huffman* (TH), reserva el primer bit de cada byte para marcar si un byte es el primer byte de un código o no, y aplica codificación Huffman sobre los restantes 7 bits de cada byte. La compresión obtenida está en torno al 33-34 %, pero los códigos generados de esta forma se vuelven *síncronos* permitiendo acceso aleatorio al texto comprimido, y la capacidad para realizar búsquedas sobre el texto comprimido utilizando algoritmos de la familia Boyer-Moore[3,11].

En [6] se presenta el *End-Tagged dense code* (ETDC), que mantiene el bit de marca del TH, pero en lugar de marcar el principio, marca el final de cada código. Esto permite utilizar una codificación totalmente secuencial (sin necesidad de utilizar codificación Huffman) y aprovechar todos los valores posibles de cada byte. En la práctica, el ETDC mantiene todas las propiedades del TH (sincronismo, descompresión aleatoria y búsquedas directas mediante Boyer-Moore) a la vez que obtiene ratios de compresión sólo 1 punto porcentual peores que el Plain Huffman (31 %). En [5,6], se presenta una mejora del ETDC, conocida como (s, c)-*Dense Code* (SCDC) que obtiene ratios de compresión sólo 0.2 puntos porcentuales peores que el Plain Huffman.

Los resultados mostrados en [15,6] muestran que estas técnicas son muy rápidas a la hora de comprimir (comprimen 1Gb de texto en poco más de 1 minuto en un pc convencional), siendo las más rápidas del estado del arte en cuanto a descompresión (descomprimiendo en torno a 60Mb/seg.), y permiten realizar búsquedas hasta 8 veces más rápido que sobre el texto sin comprimir.

3. Índices invertidos y compresión

Los índices invertidos son sin duda las estructuras de indexación más simples y utilizadas en recuperación de información (RI) para la indexación de texto en lenguaje natural [2,19], siendo también pieza clave en la mayoría de los motores de búsqueda actuales.

Un índice invertido es básicamente un vector de listas, donde para cada palabra o *término* del vocabulario[1], se almacena una lista que contiene las posiciones donde ésta aparece dentro de la colección de documentos indexada. Existen dos variantes principales [1,22].

La primera variante [19,22], se enfoca hacia la recuperación de documentos *relevantes* ante una consulta dada. En este caso, para cada documento se mantiene un vector, cuyas dimensiones son los términos del vocabulario, y en el que sus valores representan la relevancia de un término en dicho documento. De este modo las listas invertidas almacenan para cada término, los documentos en los que aparece, y el peso en dicho documento. Una consulta normalmente implica mezclar las listas de los términos de la consulta de modo que se recuperen los documentos que se consideren relevantes, lo que requiere combinar los pesos de dichos términos en cada documento.

Por otro lado, los índices invertidos orientados a la *recuperación de texto completo*, se centran en la recuperación de documentos en los que un término aparece. Las listas invertidas almacenan los documentos donde aparece cada término, típicamente ordenadas por orden de aparición. Las consultas que se refieren a una única palabra son resueltas con la simple obtención de la lista invertida de dicha palabra; consultas tipo *OR* implican mezclar listas; y consultas conjuntivas requieren la realización de intersección de listas [2]. Cuando se buscan frases, el índice invertido permite filtrar los documentos en los que aparecen todos los términos de la consulta (mediante intersección de sus listas), siendo necesario un posterior escaneado de dichos documentos para verificar si dicha frase se encuentra en el documento. La búsqueda de frases puede realizarse más eficientemente en el propio índice invertido si éste almacena también la posición en cada documento donde aparece cada palabra. Otras variantes como los índices invertidos con direccionamiento por bloques particionan la colección en bloques, y para cada término apuntan en qué bloques aparece. De esta forma obtienen diferentes relaciones espacio/tiempo en función del tamaño de bloque elegido.

El desarrollo de las técnicas de compresión semi-estáticas orientadas a palabras abrió el camino para que los índices invertidos pudiesen indexar texto comprimido, agilizando su funcionamiento especialmente en aquellos casos en los que las listas invertidas se mantienen en disco, o cuando el índice no almacena información posicional completa. Además, las listas invertidas, que son secuencias de números crecientes, también son altamente compresibles mediante alguna técnica de compresión de enteros (golomb-rice codes, bytecodes, etc.).

El tamaño total de un índice invertido comprimido (incluyendo el del texto comprimido) está en torno al 50-60 % del tamaño original de la colección. En índices invertidos que apunten a documentos o bloques, dicho tamaño puede reducirse hasta un 35-30 %, manteniendo la posibilidad de realizar las mismas operaciones que los índices invertidos sin comprimir. Por ejemplo, en un índice invertido que apunte a bloques, comprima las listas invertidas con golomb-codes,

[1] Hablaremos de vocabulario como el conjunto de palabras diferentes del texto. Sin embargo, también podríamos estar interesados en que el vocabulario no almacenase *stopwords*, o contuviese sólo las *raíces* o *lemas* obtenidos tras un proceso *lematización*.

y donde el texto (1Gb) esté comprimido con ETDC (tamaño (índice + texto) = 37 % del texto original), el tiempo medio para localizar una aparición de 1 palabra está en torno a $10\mu s$, y el de una frase formada por 2 palabras es próximo a $15\mu s$ [7].

4. Autoindexación

En escenarios donde el concepto de palabra no existe (p.ej. secuencias biológicas), y donde se hace necesario ser capaz de buscar cualquier subcadena del texto, los índices invertidos no resultan interesantes. Por este motivo aparecieron estructuras como los *arrays de sufijos (SA)* [13]. Los *SA* son capaces de encontrar cualquier subcadena del texto muy eficientemente (en tiempo $O(\log_2(|texto|)))$, pero pueden ocupar hasta cuatro veces más que el propio texto indexado.

Los *autoíndices* [17,16] surgen para reducir el tamaño de los *SA*. Los autoíndices son estructuras comprimidas que permiten indexar un texto al mismo tiempo que mantienen una representación implícita del mismo, evitando así la necesidad de almacenarlo aparte. Frente a los autoíndices originales, en 2008 surgen dos nuevos autoíndices que indexan palabras y están enfocados a la indexación de texto en lenguaje natural llamados *Byte Oriented Codes Wavelet-Tree (BOCWT)* [4] y *Word Compressed Suffix Array (WCSA)* [7].

Actualmente, la autoindexación constituye un campo de investigación dinámico y prometedor, donde numerosas propuestas[2] han surgido en la última década. En cualquier caso, los nuevos autoíndices siguen sin ser prácticamente utilizados en operaciones de IR [9] o de IL debido posiblemente a que han evolucionado "demasiado" rápido y no han sido desarrollados pensando en proveer ciertas funciones que estas disciplinas necesitan. El modelo propuesto en la Sección 4 pretende permitir dichas operaciones sobre cualquier estructura de indexación, entre ellas sobre los autoíndices *BOCWT* y *WCSA*.

4.1. Autoíndices para texto en lenguaje natural

Los autoíndices que indexan palabras en lugar de caracteres se benefician de tener que indexar un menor número de símbolos (en inglés la longitud media de una palabra es aproximadamente 4-5 caracteres). Este factor los hace más compactos (el mismo autoíndice orientado a caracteres puede ocupar más del doble [7]) y rápidos. A cambio, sólo pueden buscar palabras o frases, no cualquier subcadena del texto.

Los autoíndices siguen el API en `http://pizzachili.dcc.uchile.cl/api.html`. Entre otras, han de solucionar las siguientes operaciones básicas: *i) count(P)*, contar el número de apariciones de un patrón; *ii)) locate(P)*, para encontrar en qué posiciones aparece el patrón buscado y *iii) extract(ini, end)*, que recupera cualquier parte del texto original desde la posición *ini* hasta la *end*, esto es, descomprime el texto contenido por el autoíndice.

[2] Artículos relacionados y códigos fuente disponible en `http://pizzachili.dcc.uchile.cl/`

Word-Compressed Suffix Array (WCSA)

El WCSA [7] es la evolución del *word-based SA (WSA)* presentado en [8]. Dado un texto $T_{1..n}$, se obtiene el vocabulario de palabras, y se le asigna a cada palabra un identificador entero *id*. A continuación se obtiene un array T_{id} sustituyendo cada palabra de T por su *id*. De esta forma T puede ser reemplazado por T_{id} y el vocabulario de palabras (ordenado alfanuméricamente). A continuación se ordenan todos los sufijos de T_{id}, donde un sufijo i representa a la secuencia de enteros $T_{id}[i..n]$, de forma que $T_{id}[SA[i]..n] \preceq T_{id}[SA[i+1]..n]$. En la Figura 1 se muestra un SA que indexa palabras. Se puede observar que $T_{id}[SA[5]..n] = \langle 3, 2, ... \rangle$ es menor que $T_{id}[SA[6]..n] = \langle 3, 6, ... \rangle$.

Figura 1. Estructura de un SA orientado a palabras.

La ventaja de un array de sufijos radica en la posibilidad de buscar cualquier patrón usando una simple búsqueda binaria, debido precisamente a que todos los sufijos apuntados desde SA están ordenados. Dado un patrón P, tal que P es la secuencia de ids de las palabras de una frase, el WSA puede resolver *count(P)* en $O(|P| \log_2 n)$ y *locate(P)* en tiempo $O(|P| \log_2 n + occ)$, siendo *occ* el número de veces que aparece el patrón P. Para buscar la palabra "`azul`" en la Figura 1, simplemente habría que buscar dónde aparece el $id = 1$ en T_{id}. Esas posiciones son apuntadas desde $SA[2..3]$. Esto nos indica que sólo hay 2 apariciones $(2 = 3 - 2 + 1)$, y sus localizaciones en T_{id}; esto es, las posiciones $SA[2] = 1$ y $SA[3] = 7$ de T_{id}. De nuevo, el problema del WSA es que su tamaño es similar al tamaño del texto indexado.

El *WCSA* presentado en [7] junta las ideas del WSA y aquellas mostradas en [17], para obtener un autoíndice comprimido que mantiene la funcionalidad del WSA, ocupando sólo un 35-40 % de tamaño del texto original. Junto al *WCSA*, en [7] también se presentó el *Flexible WCSA (FWCSA)*. El *FWCSA* es el primer autoíndice basado en arrays de sufijos que permite incluir operaciones típicas de la indexación de lenguaje natural como: lematización, eliminación de *stopwords*, *case folding*, etc.

Byte Oriented Codes Wavelet-Tree (BOCWT)

El Byte-Oriented Codes Wavelet Tree (BOC-WT) [4] es un autoíndice que se construye a partir de los bytes de un texto reorganizándolos en una estructura con forma de árbol. Puede aplicarse sobre el texto comprimido obtenido mediante

cualquier técnica de compresión libre de prefijo que sea estadística, semiestática, orientada a byte y a palabras. Mantiene el mismo ratio de compresión y tiempos similares de compresión y descompresión que la técnica de compresión utilizada, pero mejora drásticamente las búsquedas debido a que se consigue autoindexar el texto de forma implícita mediante la estructura de árbol.

De forma resumida, este método recoloca los bytes del texto comprimido siguiendo una estrategia de wavelet tree [10], situando los primeros bytes de cada código en el primer nivel del árbol, los segundos bytes de cada código en el segundo nivel, en la rama correspondiente en función del primer byte del código, y así sucesivamente hasta llegar al último nivel del árbol, que tendrá una profundidad igual a la longitud del código de la palabra más larga. De esta forma, se puede asociar un nodo hoja del árbol con cada palabra de forma que las palabras se pueden buscar de forma eficiente, independientemente del tamaño del texto. Así, esta técnica adquiere propiedades de índice, obteniendo resultados interesantes para contar, localizar y extraer los snippets de las apariciones de cualquier palabra del texto.

5. Modelo de encapsulación

En esta sección presentamos un modelo que encapsula los documentos de la colección representándolos eficientemente mediante alguna de las técnicas descritas en las secciones anteriores. Ya sea una representación comprimida o autoindexada de la colección, el usuario podrá manipular estas estructuras de datos mediante una interfaz común que determinará completamente su comportamiento externo, quedando oculto el funcionamiento interno de la representación.

Así, la representación compacta y eficiente de la colección de documentos se vuelve una "caja negra", siendo solamente necesario definir una interfaz para poder interactuar con ella. Los detalles de implementación se mantienen ocultos, de forma que no es necesario ningún conocimiento exhaustivo de las estructuras de datos complejas que se usan para llevar a cabo las operaciones sobre la colección, ni de los algoritmos que las soportan.

Actualmente están disponibles las implementaciones de las estructuras de indexación descritas en las secciones anteriores y los algoritmos necesarios para realizar las búsquedas sobre ellas. Próximamente se podrá acceder a la interfaz común que a continuación describimos, de forma que facilite su uso para cualquier usuario no experto en el campo de las estructuras de datos y algoritmos complejos y además que permita intercambiar el tipo de representación dependiendo de la funcionalidad deseada.

5.1. Descripción de las primitivas

A continuación describimos de forma detallada la interfaz mediante la que el usuario interactúa con la representación de la colección de documentos, de forma que se opera como si se tratase de una caja negra, ocultando los detalles internos de su funcionamiento.

Aunque se enumeran las primitivas más básicas, esta interfaz es extensible a cualquier tipo de operación que sea necesario realizar sobre la colección de textos. Si bien alguna operación compleja puede verse como una agregado de alguna de las operaciones simples que se proponen, es posible que la operación compleja pueda realizarse de forma más eficiente en algún tipo de índice determinado que la suma de las operaciones individuales. Por ello, aunque no se detallen a continuación, es posible ampliar el conjunto de primitivas de la interfaz.

Representación de la colección

En esta sección detallamos las primitivas más básicas que permiten, por un lado, construir la representación eficiente de la colección de documentos, y por el otro lado, recuperar cualquier documento o la colección entera a partir de la representación.

- `CrearRepresentacion[tipo]<coleccion>`: permite crear la representación eficiente de la colección de documentos. El usuario debe decidir qué tipo de representación es el más adecuado para representar la colección, decisión que dependerá del uso que realizará sobre los documentos. Para ello se detallan todas las posibles configuraciones internas del índice, con sus características, ventajas e inconvenientes. De esta forma el usuario tendrá toda la información necesaria para elegir la representación más adecuada al uso posterior, y la decisión dependerá de la frecuencia en la que requiera el uso de cada una de las operaciones permitidas. Para crear el índice que represente la colección de documentos se utiliza la llamada `CrearRepresentacion`, en la que se debe especificar el *tipo* de índice que se creará para la representación (índice invertido, array de sufijos comprimido, representación con wavelet tree, etc.), y que recibe como argumento la *coleccion* de documentos que se quieren representar de forma eficiente.
  Para utilizar una configuración óptima, el usuario debe elegir la representación que mejor se adapte a sus necesidades. Para asistirle en esta decisión, en la Tabla 1 se compara la funcionalidad de los índices y su eficiencia para resolver las diferentes tareas. Por ejemplo, si lo que se desea normalmente es localizar apariciones de frases, debería utilizar un WCSA.
  Igualmente, si se diese el caso de una decisión de tipología del índice no adecuada por parte del usuario, la representación de la colección podría ser reconstruida con otro parámetro y reemplazar la representación anterior sin ninguna modificación en el resto del código, ya que todos los índices creados siguen la misma interfaz.
- `Añadir<Documento>`: permite añadir un nuevo documento a la colección. Con determinados tipos de representación de la colección es posible añadir nuevos documentos de forma eficiente, sin necesidad de reconstruir totalmente la representación de toda la colección. Con otros tipos de índice, esta operación no es posible, ya que no soporta dinamismo. Independientemente del tipo de representación utilizada, el usuario puede añadir un nuevo documento a la colección y es la propia representación quien se encarga de realizar el proceso

Tarea	II comprimido	WCSA	BOC-WT
Calcular frecuencias de términos	Eficiente sólo si almacena frec.	Bueno	Óptimo
Localizar apariciones de términos	Depende del tamaño de bloque	Bueno	Muy bueno
Localizar apariciones de frases	Ineficiente	Óptimo	Muy bueno
Extraer snippets	Bueno con tamaño de snippet grande	Muy bueno	Bueno
Extraer todo el documento	Óptimo	Muy bueno	Bueno
Búsqueda de lemas	Es posible	Es posible	No disponible actualmente
Consideración de stopwords	Es posible	Es posible	No disponible actualmente
Añadir documentos a la colección	Eficiente	No eficiente (reconstruir)	Eficiente

Tabla 1. Comparativa entre los distintos tipos de índice.

de adición del texto a la representación de la colección, actuando como se ha dicho previamente como una caja negra. Para ello se pasa como parámetro el nuevo *Documento* a la primitiva `Añadir`.

- `ObtenerColeccion`: permite recuperar la colección de documentos original. La primitiva `CrearRepresentacion` permite obtener la representación eficiente de la colección de documentos. Esta representación sustituye al conjunto de textos original a la hora de procesarlos y trabajar sobre ellos. Sin embargo, es posible que el usuario necesite recuperar la colección de documentos, por lo que es necesaria esta primitiva en la interfaz del modelo de encapsulación.
- `ObtenerDocumento<IdDocumento>`: permite recuperar el texto completo original de un documento específico de la colección de forma eficiente a partir de la representación. Simplemente es necesario invocar a la primitiva pasándole como argumento el número identificativo del texto que se quiere recuperar.

Procesamiento de la colección

La representación del documento debe permitir un procesamiento del texto básico, que pueda devolver la palabra que está en determinada posición de un documento de la colección, devolver la siguiente palabra, un fragmento del documento, etc.

- `Vocabulario`: permite obtener el conjunto de todos los términos distintos que contiene la colección de documentos.
- `Termino<IdDocumento><pos>`: permite obtener la palabra que está en una posición determinada de un documento. De esta forma, se podría procesar el texto palabra por palabra utilizando esta primitiva de forma recurrente.
- `ObtenerDocumento<IdDocumento><posInicial><posFinal>`: permite obtener una parte de un documento. En este caso, restringimos la obtención de un documento al fragmento entre dos posiciones del texto dadas.

Recuperación de datos simples sobre la colección

A continuación detallamos una serie de primitivas que permiten recuperar datos simples de la colección de documentos. Estas primitivas son básicas, por ejemplo, para realizar cálculos como la medida de la frecuencia inversa de documento, imprescindibles para el área de Recuperación de Información.

- `NumeroDocumentos`: es una operación básica que permite obtener el número total de documentos que conforman la colección.
- `NumeroTerminos`: devuelve el número total de palabras distintas que contienen todos los documentos de la colección.
- `NumeroTerminos<IdDocumento>`: de forma análoga a la anterior, devuelve el número de términos distintos en un documento, que se pasa como argumento mediante su identificador de documento.
- `FrecuenciaTermino<termino>`: calcula el número de veces que aparece en todos los documentos de la colección un término en concreto, que se pasa mediante argumento a la primitiva. Es decir, calcula la frecuencia de un término en toda la colección.
- `FrecuenciaTermino<termino><IdDocumento>`: es análoga a la anterior, pero restringida a un documento en concreto que se pasa como argumento mediante su identificador de documento. Es decir, calcula la frecuencia de un término en un documento.
- `NumeroDocumentos<termino>`: permite calcular el número de documentos de la colección que contienen un término. Éste se pasa como argumento a la primitiva.

Búsqueda de términos en la colección

En esta sección se describen las primitivas que permiten realizar búsqueda de términos dentro de los documentos de la colección.

- `Documentos<termino>`: obtiene los documentos de la colección que contienen un término. Este término se pasa como argumento a la primitiva.
- `Posiciones<termino><IdDocumento>`: permite obtener todas las posiciones en las que aparece un determinado término en un documento concreto. Tanto el término como el documento se pasan como argumentos.
- `SiguientePosicion<termino><IdDocumento><posPartida>`: obtiene la posición de la siguiente aparición de un término en un documento concreto. Para ello, se establece una posición de partida en el texto y se busca la aparición más cercana a esa posición del documento.
- `Snippets[longitud]<termino><IdDocumento>`: permite obtener todos los snippets de un término en un documento. Se requiere un parámetro para determinar la longitud de los snippets devueltos, es decir, de los trozos de texto que rodean al término en cada aparición del mismo en el documento.
- `SiguienteSnippet[longitud]<termino><IdDocumento><posPartida>`: de forma análoga a la primitiva `SiguientePosicion`, podemos obtener el snippet de la siguiente aparición del término a una posición dada.

Otras funcionalidades

Dependiendo del tipo de representación que se esté utilizando, es posible extender el conjunto de operaciones permitidas. Por ejemplo, si se está utilizando un índice que mantenga información de lemas y variantes de términos, es posible utilizarlo para obtener las diferentes variantes de un término dado en un documento.

Asimismo, esta interfaz está abierta a cualquier demanda de funcionalidad proveniente de cualquier comunidad investigadora que vea interesante el uso de esta representación, que ofrece un acceso, manipulación y almacenamiento eficientes de una colección de documentos, permitiendo la realización de las tareas propias de sus campos de investigación sin necesidad de entender los detalles complejos de las estructuras de datos ni de sus algoritmos.

6. Conclusiones y trabajo futuro

En este trabajo se han revisado los últimos avances en compresión e indexación para texto en lenguaje natural. Hemos puesto especial atención a las técnicas de compresión semi-estáticas orientadas a palabras, que pueden ser integradas junto a estructuras de indexación como los índices invertidos, mejorando su rendimiento en todos los aspectos. En cuanto a técnicas de indexación nos hemos centrado en los índices invertidos comprimidos y en los autoíndices de palabras. Estas estructuras permiten representar una colección de textos de forma compacta al mismo tiempo que permiten manejarla de forma eficiente.

En la investigación en Recuperación de Información e Ingeniería Lingüística no suele ser prioritaria la eficiencia de las técnicas de representación de las colecciones de texto con las que trabaja, centrándose fundamentalmente en aspectos relacionados con la eficacia. Sin embargo, la utilización de estructuras avanzadas de compresión e indexación como base de sus sistemas mejoraría notablemente la eficiencia de los mismos. Por ello, proponemos un modelo que encapsula la representación interna de las colecciones y ofrece al usuario/investigador una interfaz con la funcionalidad necesaria para realizar sus tareas.

Estamos actualmente terminando la implementación de la encapsulación de las estructuras de datos y algoritmos de los distintos métodos citados en este trabajo: un índice invertido comprimido y los autoíndices WCSA y BOC-WT. El siguiente paso será hacerlo público para que cualquier usuario pueda descargarlo y utilizarlo.

Referencias

1. R. Baeza-Yates, A. Moffat, and G. Navarro. *Searching Large Text Collections*, pages 195–244. Kluwer Academic, 2002.
2. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman, 1999.
3. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

4. N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 139–146. ACM Press, 2008.
5. N. Brisaboa, A. Fariña, G. Navarro, and M. Esteller. (s,c)-dense coding: An optimized compression code for natural language text databases. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2857, pages 122–136. Springer, 2003.
6. N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.
7. N. Brisaboa, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Self-indexing natural language. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 121–132. Springer, 2008.
8. Paolo Ferragina and Johannes Fischer. Suffix arrays on words. In *In Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching, volume 4580 of LNCS*, pages 328–339. Springer, 2007.
9. Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *SPIRE '09: Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, pages 1–6, Berlin, Heidelberg, 2009. Springer-Verlag.
10. R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 03)*, pages 841–850, 2003.
11. R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.
12. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
13. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
14. A. Moffat. Word-based text compression. *Softw. Pract. Exper.*, 19(2):185–198, 1989.
15. E. Moura, G. Navarro, N. Z., and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM TOIS*, 18(2):113–139, 2000.
16. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
17. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
18. A. Turpin and A. Moffat. Fast file search using text compression. In *Proc. 20th Australian Computer Science Conference (ACSC)*, pages 1–8, 1997.
19. I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, USA, 1999.
20. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
21. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.
22. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2):article 6, 2006.