Estructuras de Datos Compactas

Las diferencias de velocidad en la jerarquía de memoria son cada vez más pronunciadas. Esto ha propiciado una tendencia al estudio de las estructuras de datos que puedan operar usando poco espacio, ya que aunque realicen más operaciones, las pueden ejecutar en una memoria muchas veces más rápida. En una época en que la cantidad de datos a manipular es gigantesca, y las memorias donde pueden ser almacenados son las más lentas de la jerarquía, las estructuras de datos compactas están ofreciendo soluciones muy atractivas en áreas como recuperación de información, Web, bioinformática, bases de datos multimediales, y muchas otras. En este artículo paso revista a algunos de los resultados más espectaculares que ha obtenido esta fascinante área de investigación, e incluyo algunas contribuciones propias relevantes.

INTRODUCCIÓN

Gonzalo Navarro
Profesor Titular, DCC, Universidad de Chile. Doctor en Ciencias mención Computación de la misma Universidad (1998). Sus áreas de interés incluyen algoritmos y estructuras de datos, búsqueda en texto, compresión y búsqueda en espacios métricos.
gnavarro@dcc.uchile.cl

En 1965, el co-fundador de Intel, Gordon Moore, estimaba que "el número de transistores que consigue el menor costo por transistor en un circuito integrado se duplica cada 24 meses". La llamada "Ley de Moore" se ha mantenido válida durante décadas y se aplica a varios aspectos de la tecnología de los computadores: las capacidades de la memoria principal (RAM), el poder de las CPUs, las capacidades de almacenamiento de los discos, etc. Este fabuloso incremento en las capacidades del hardware es en gran parte el responsable de la revolución tecnológica de los últimos 50 años, permitiendo afrontar desafíos de una magnitud impensable hace unos pocos años.

Sin embargo, existen otros aspectos del hardware que no siguen esta ley. Los más notorios son posiblemente la velocidad de acceso al disco y a la RAM. La respuesta a este estancamiento en las velocidades de las memorias ha sido el enriquecer esta jerarquía con nuevas componentes más rápidas, aunque también más caras y (por lo mismo) menores. Las cachés L1, de unos pocos KBs y prácticamente pegadas al chip de la CPU, y luego las L2, de unos pocos MBs, ofrecen "buffers" intermedios, en tamaño y velocidad, entre los registros de la CPU y la RAM de unos pocos GBs. Estos buffers pueden ser desde sólo 2 hasta 50 veces más rápidos que la RAM, y a su vez la RAM es unas cien mil veces más rápida que el disco. Recientemente, las memorias Flash han agregado un nivel entre estas dos, resultando unas 100 veces más rápidas (y menores en capacidad) que los discos. Diversas razones económicas (velocidad vs costo por bit) y físicas (límites fundamentales, distancia a la CPU) hacen pensar que esta jerarquía de memoria se mantendrá por mucho tiempo.

Por otro lado, tenemos las aplicaciones cada vez más ambiciosas en términos de cantidad de datos a generar y manipular. La Web sigue creciendo a un ritmo exorbitante, con cientos de terabytes sólo en la parte estática, y duplicándose aproximadamente cada ocho meses. La minería de su utilización (qué sitios visitan los usuarios, qué consultas hacen) genera un flujo gigantesco de datos diariamente. La bio-informática está cerca de lograr secuenciar cada individuo a un costo razonable, lo cual abre las puertas a tener que mantener los tres mil millones de bases del genoma humano para cada persona. Las aplicaciones astronómicas amenazan con inundarnos de terabytes de datos por día provenientes de los telescopios. La TV y multimedios digitales generan exabytes (millones de terabytes) de material al año. Las tecnologías actuales pueden permitir *almacenar* esta cantidad de datos, pero eso es lo de menos. Lo que se necesita es *manipular* esos datos: analizarlos, transformarlos, extraer información útil de ellos. Y esta tarea es extremadamente penosa si requiere acceder a los datos de forma intensiva y debe llevarse a cabo sobre los datos almacenados en disco.

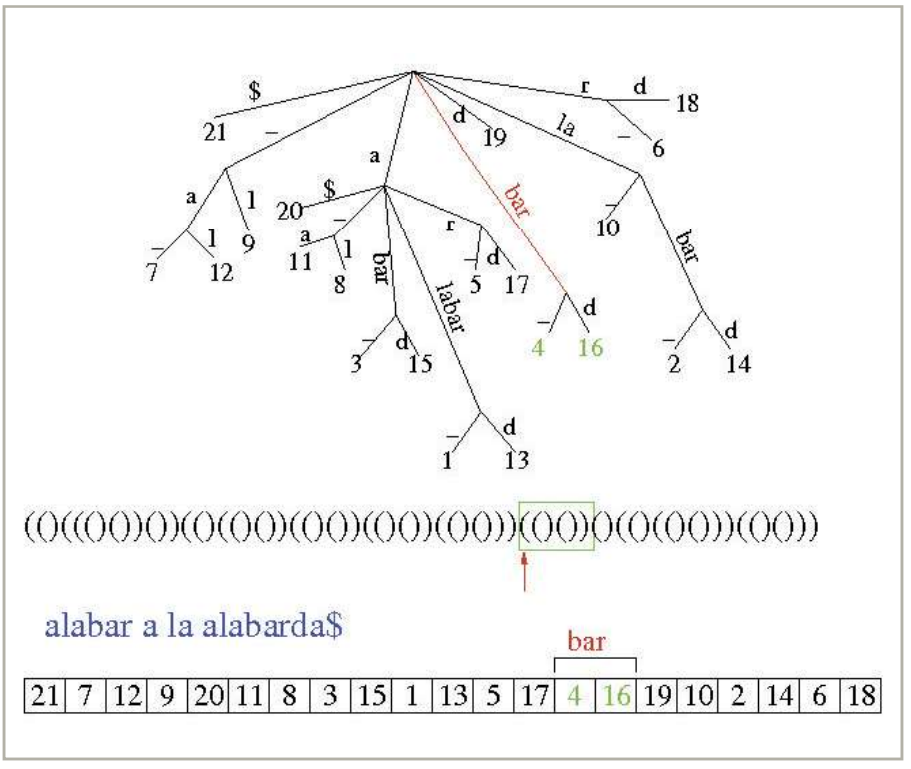
Tomaré un ejemplo bastante modesto como motivación concreta. Supongamos que queremos almacenar un genoma humano (sólo uno, que es lo usual todavía). Este contiene unas tres mil millones de bases, que podemos ver como símbolos sobre un alfabeto de 4 caracteres: A, C, G, T (pueden aparecer otros, pero evitaré complicaciones innecesarias). Almacenando cada carácter en un byte resulta ser cerca de un espacio de 3GB, perfectamente manejable en la RAM de un computador estándar moderno. Más aún, codificando cada carácter en 2 bits podemos almacenarlo en poco más de 700MB.

Sin embargo, los bio-informáticos lo que realmente necesitan es *analizar* este genoma, no sólo almacenarlo. Los tipos de análisis que llevan a cabo consisten, por ejemplo, en buscar secuencias de ADN (es decir, cadenas sobre el alfabeto de cuatro caracteres) para ver dónde ocurren en el genoma, en forma exacta o permitiendo algunas diferencias entre lo que se busca y lo que se encuentra; buscar zonas del genoma que se repiten, en forma exacta o aproximada; descubrir patrones de interés estadístico; etc. Estas búsquedas serían impracticables (por ejemplo, de tiempo cuadrático en el largo del genoma) si no se utilizaran estructuras de datos sobre la secuencia, llamadas *índices*.

El índice más popular por su eficiencia y versatilidad es el *árbol de sufijos*. Éste permite realizar tareas sorprendentemente complejas con un costo de tiempo razonable. Sin embargo, requiere de 10 a 20 bytes por carácter de la secuencia, es decir, unos 30 a 60GB para un genoma humano. Unido a que este tamaño excede en un orden de magnitud lo que hoy en día se puede almacenar en una RAM estándar, se da la desafortunada situación de que los algoritmos sobre árboles de sufijos no son particularmente locales en su acceso a los datos, lo que se paga muy caro cuando se accede al disco. Esta combinación inutiliza, en la práctica, lo que en principio es una excelente solución algorítmica.

A lo largo del artículo mostraré cómo las estructuras de datos compactas permiten representar el árbol de sufijos en poco espacio más del necesario para almacenar la secuencia misma, ofreciendo una solución completamente practicable en espacio y tiempo. En el camino pasaré revista a distintas estructuras de datos compactas necesarias para la solución, y a varias contribuciones propias recientes.

La investigación en estructuras de datos compactas tiene dos componentes: uno es el algorítmico, como es de esperar. El otro, de teoría de la información. Representar los datos en el menor espacio posible obliga a considerar cuánta información hay en ellos y cómo puede representarse en forma compacta. Sin embargo, no se trata sólo de compresión, donde basta justamente con *representar* los datos. Esta vez se necesita



considerar el costo de *acceder* a ellos *sin descomprimirlos*. Esta estrecha interacción entre algoritmos y teoría de la información es una de las cosas más fascinantes de esta área.

EL ÁRBOL DE SUFIJOS

Considere un texto $T[1,n]$ sobre un alfabeto Σ de tamaño σ. Este texto contiene n sufijos: $T[i,n]$ para cada i. Cada sufijo se puede identificar por su posición de partida, i. Consideremos que el último carácter de T es especial, $T[n]$ = \$, menor lexicográficamente que todos los otros. Un *arreglo de sufijos* es un arreglo de enteros $A[1,n]$, donde los sufijos se listan en *orden lexicográfico*. Un *árbol de sufijos* es (1) un árbol cuyas hojas son las celdas del arreglo de sufijos, en el mismo orden, (2) tiene las aristas rotuladas con cadenas de caracteres, y (3) la concatenación de los rótulos de la raíz hasta cada hoja es el sufijo correspondiente (no completo, sino expandido hasta el punto en que ningún otro sufijo empieza igual). El árbol no tiene nodos unarios (con un sólo hijo), de haberlos se deben unir en una sola arista. La figura muestra un árbol (arriba) y arreglo de sufijos (abajo) para el texto *"alabar a la alabarda"*. Ignore por ahora los paréntesis que se interponen entre ambos y lo destacado en color.

La utilidad del arreglo de sufijos reside en que todo substring del texto es el prefijo de algún sufijo. Por lo tanto, encontrar todas las ocurrencias de un cierto patrón $P[1,m]$ en T equivale a encontrar todos los sufijos que comienzan con P. Estos forman un rango lexicográfico, por lo cual aparecen todos en un intervalo contiguo del arreglo de sufijos (ejemplificado en la figura con la búsqueda de "bar"). Este se puede determinar mediante dos búsquedas binarias, en tiempo $O(m \log n)$, y luego todas las posiciones del patrón están listadas en el intervalo. Este tiempo es muy bueno, pero el arreglo de sufijos consta de n enteros, que requieren n log n bits , mientras que el texto sólo requiere n log σ bits[1], lo cual es considerablemente menor.

El árbol de sufijos permite realizar esta búsqueda en tiempo $O(m)$, mediante bajar en el árbol y reportar las hojas del subárbol. Si bien la cantidad de nodos en el árbol de sufijos es $O(n)$, y por lo tanto requiere $O(n \log n)$ bits, la constante es de tres a cinco veces la del arreglo de sufijos en la práctica. A cambio, el árbol de sufijos permite hacer otras cosas difíciles o imposibles de simular eficientemente en el arreglo de sufijos. Por ejemplo, para encontrar la auto-repetición más larga en T, basta con recorrer las hojas y reportar la más profunda (en términos del largo de la cadena que representa), lo que resuelve en tiempo lineal algo que requeriría tiempo cuadrático sin el árbol. Otras operaciones muy propias del árbol son el "suffix link", que lleva del nodo que representa la cadena aX al que representa X, donde a es un carácter; y el "ancestro común más bajo" entre dos nodos u y v.

¿CÓMO REPRESENTAR UN ÁRBOL?

Un elemento a comprimir del árbol de sufijos es la topología misma del árbol. Hago notar que hay soluciones donde se prescinde completamente del árbol, pero lo retendré por su valor pedagógico.

Un árbol general con n nodos se representa usualmente con $O(n)$ punteros, los cuales al menos necesitan $O(n \log n)$ bits para que cada puntero pueda distinguir entre n nodos distintos a apuntar. Sin embargo, es fácil ver que esto es un grosero exceso, al menos en términos de la Teoría de la Información. La cantidad de árboles generales distintos de n nodos es el número de Catalán $Cn\text{-}1 = \text{comb}(2n\text{-}2,n\text{-}1)/n = \Theta(4^n/n^{3/2})$, por lo cual deberían bastar $\log C_{n-1} = 2n - O(\log n)$ bits para representar cualquier árbol de n nodos[2].

No es especialmente difícil representar un árbol general usando $2n$ bits. Por ejemplo, podemos recorrer el árbol en pre-orden, abriendo un paréntesis cada vez que bajamos por una arista y cerrándolo cuando subimos. El resultado para el árbol de la figura se dibuja debajo del mismo. Esta representación permite reconstruir la topología del árbol original. ¡Pero eso no es lo que buscamos! Lo que necesitamos es poder navegar en esta representación de paréntesis, tal como navegamos un árbol con punteros.

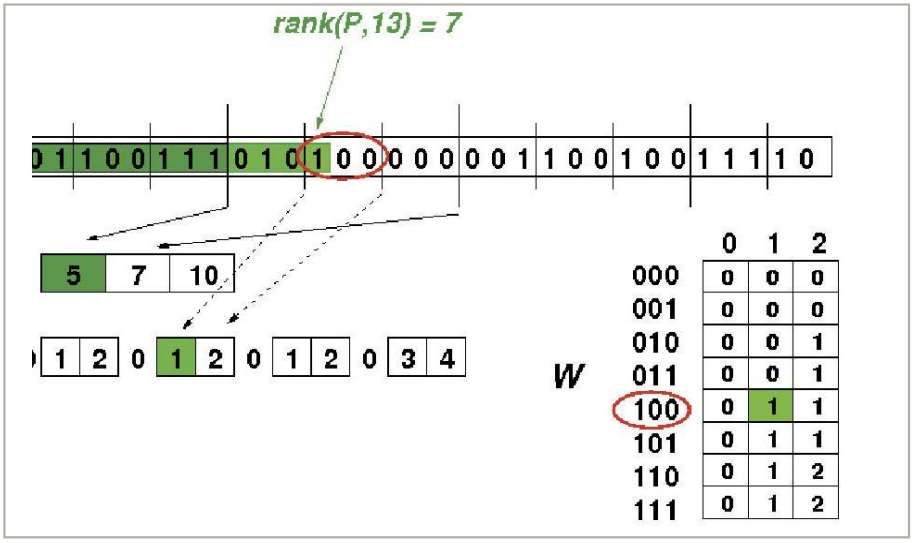
Consideremos que un nodo se identificará con la posición de su paréntesis abierto en la secuencia. Lo primero que quisiéramos sería conocer cuál es su posición en un recorrido en pre-orden (por ejemplo, para almacenar información asociada al nodo en otro arreglo). Esto no es más que la cantidad de paréntesis abiertos hasta el de ese nodo. Si representamos la secuencia de paréntesis en un bitmap $P[1,2n]$, usando el 1 para el paréntesis que abre y 0 para el que cierra, la operación se llama:

$preorden(i) = rank1(P,i)$ = número de 1s en $P[1,i]$.

Asimismo, nótese que la profundidad del nodo i es la cantidad de paréntesis que abren menos la cantidad que cierran hasta la posición i. Esto se llama:

$profundidad(i) = rank_1(P,i) - rank_0(P,i) = 2\, rank_1(P,i) - i$.

¿Cómo podemos calcular rank en tiempo constante, sin utilizar demasiado espacio? La solución es simple y práctica. Consideraremos sólo $rank_1$, ya que $rank_0(P,i) = i - rank_1(P,i)$. Dividimos el bitmap en súperbloques de largo s y bloques de largo b (donde b divide



[1] Los logaritmos serán en base 2 en este artículo.

[2] Dada la pobreza del editor que he sido obligado a usar para este articulo, o mi pobre conocimiento de él, estoy utilizando comb(n,m) para denotar el número combinatorio $n!/(m!(n\text{-}m)!)$.

a *s*). Pre-calculamos una tabla *S*[1,*n/s*], donde S[j] = $rank_1$(P,j*s) es el valor de rank1 hasta el comienzo de cada súper-bloque. Pre-calculamos otra tabla *B*[1,*n/b*], donde *B*[*k*] = $rank_1$(*P*,*k***b*) – *S*[floor(*k***b*/*s*)] es el valor de $rank_1$ hasta el comienzo de cada bloque, pero relativo al comienzo de su súper-bloque. Entonces, si *i* = *j***s* + *r* = *k***b* + *r'*, donde *0* <= *r* < *s* y 0 <= *r'* < *b*, significa que *i* pertenece al súper-bloque *s*, bloque *b*, y tiene offset *r'* dentro de *b*. Entonces podemos calcular en tiempo constante

$rank_1(P,i) = S[j] + B[k] + rank_1(P[k*b+1..k*b+b],r'),$

donde el último valor se calcula recorriendo *P* directamente en tiempo *O(b/w)*, siendo *w* la cantidad de bits en la palabra de máquina, mediante técnicas de "popcount". La siguiente figura ilustra el proceso, con s=*9, b=3*, y una tabla *W* para procesar *rank* en los bloques en tiempo constante.

Para calcular el espacio necesario se debe considerar que *S* necesita *n/s*log n* bits, *B* necesita *n/b*log s* bits (pues no se representan números mayores que s), y w >= log *n*, la suposición normal en el modelo RAM si se entiende que el computador puede direccionar un número entre *n*. Definiendo *s* = $\log^2 n$ y *b* = log *n*, tenemos tiempo constante y espacio extra O(*n* log log *n* / log *n*) = *o*(*n*) bits.

Algunas operaciones de navegación sobre el árbol son muy simples. Por ejemplo, el primer hijo del nodo *i es i+1*, a menos que sea un paréntesis que cierra, en cuyo caso *i* es una hoja sin hijos. Otras operaciones son más complejas. Muchas se construyen sobre dos primitivas fundamentales:

findclose(i) = la posición del paréntesis que cierra *a i*,

enclose(*i*) = la posición del paréntesis que abre más cercano a *i* y que contiene a *i*.

Con estas primitivas, algunas operaciones son sencillas. Por ejemplo, el siguiente hermano de *i* es *findclose(i)+1* (si es que abre. Pues si cierra significa que *i* es el último hijo de su padre), el padre de *i* es *enclose(i)*, el tamaño del sub-árbol de *i* es *(findclose(i) – i + 1) / 2*, etc.

Es imposible detallar la solución a todas las operaciones en este artículo. Para dar una idea, explicaré superficialmente la forma de resolver *findclose(i)*. Se divide la secuencia en bloques de tamaño *b*, y se clasifican los paréntesis que abren como *cercanos* si cierran en el mismo bloque en que abren, y lejanos si no. Los *lejanos* se subdividen en *pioneros*, si el paréntesis lejano previo no cierra en su mismo bloque. Con esta definición resulta que (1) existen *O(n/b)* pioneros (pues conectando los bloques de los pioneros con el de sus parejas resulta un grafo planar), (2) si *i* no es pionero entonces *findclose(i)* está en el mismo bloque del pionero *j* que precede a *i*, y se puede encontrar en tiempo *O(b/w)* recorriendo desde *findclose(j)* con técnicas tipo popcount. Basta entonces marcar los pioneros en un bitmap y almacenar sus respuestas explícitamente, para poder responder *findclose(i)* en tiempo constante si se elige *b* = *O*(log *n*). Esto, sin embargo, todavía requiere de *O(n)* bits extra para almacenar las respuestas a los pioneros. En cambio, se utilizan dos niveles de recursión formando una nueva secuencia de paréntesis pioneros y sus parejas, y volviendo a aplicar la técnica sobre ella. Sólo en el segundo nivel se almacenan explícitamente las respuestas para los pioneros de los pioneros, con lo que se logra el espacio extra *o(n)* y tiempo aún constante.

El bitmap donde se marcan los pioneros, por otro lado, se puede comprimir porque contiene solamente *O(n / log n)* 1s, ocupando espacio *o(n)* y ofreciendo acceso y *rank* en tiempo constante. Esto se consigue, por ejemplo, cortándolo en bloques de largo *b* = (log *n*)/2 y codificando cada bloque como un par *(c,o)*, donde *c* (la *clase*) es la cantidad de 1s del bloque, y *o* (el *offset*) es el índice de ese bloque en una lista de todos los bloques de esa clase (estas listas deben almacenarse explícitamente, pero en total tienen sólo O(sqrt(*n*)) elementos). Los bloques con pocos o muchos 1s tienen clases pequeñas y por ello se necesitan pocos bits para su componente *o*. Con esta técnica, un bitmap *B* con *m* 1s se codifica en $n*H_0(B) + o(n)$ bits, donde $H_0(B) = (m/n) \log n/m + O(m/n)$ es la entropía de orden *cero* de *B*. A esto se le suman otros o*(n)* bits para direccionar la secuencia de campos o y para calcular *rank*.

¿CÓMO REPRESENTAR UN ARREGLO DE SUFIJOS?

Una técnica muy relevante para representar un arreglo de sufijos se basa en la Transformación de Burrows-Wheeler (BWT). Esta produce una permutación *T'* del texto *T*, que se forma escribiendo el carácter que precede a cada sufijo, en el orden en que se listan en el arreglo de sufijos. Resulta que *T'* es más fácil de comprimir que *T* porque contiene largas zonas con el mismo carácter o pocos caracteres distintos (por ejemplo, todas las "C" que preceden a "hile" en un texto probablemente aparecerán seguidas en *T'*), y por ello la BWT se usa en softwares de compresión tan eficientes como *bzip2*.

Ahora bien, la BWT tiene otra propiedad muy relevante para la búsqueda de patrones, que viene de su conexión con el arreglo de sufijos. Digamos que *A* es el arreglo de sufijos de *T*, y que *A[sp,ep]* es el rango de todos los sufijos que comienzan con P. Entonces el rango de sufijos que comienza con *aP*, siendo *a* un carácter, es *A[sp',ep']*, donde

sp' = C[a] + $rank_a$(T',sp-1)+1 y ep' = C[a] + $rank_a$(T',ep).

Aquí *C* es un arreglo que almacena, para cada carácter, la cantidad de ocurrencias en *T* de caracteres menores a él, y $rank_a(S,i)$ es la cantidad de ocurrencias de *a* en S[1,i]. La razón de esta fórmula es que C[*a*] nos ubica en la zona de *A* de los sufijos que empiezan con *a*. De ellos, los que siguen con *P* son aquellos de *A[sp,ep]* que están precedidos del carácter *a*, es decir, *T'[i] = a, para sp <= i <= ep*, mientras que los que están precedidos de *a* pero son anteriores a *P* son los que cumplen *T'[i] = a* para *i<sp*. Similarmente, si *A[i]=j*, es decir la posición *i* del arreglo de sufijos apunta a la posición *j* del texto, entonces la posición de A que apunta a la posición previa del texto, j-1, es

LF(i) = C[T'[i]] + rankT'[i](T',i).

Con esta herramienta nos basta poder calcular $rank_a$ sobre la secuencia *T'* en tiempo *O(t)* para buscar *P*[1,*m*] en *T* en tiempo *O(t*m)*. Así comenzamos con *sp* = 1 y *ep* = *n*, y procesamos los caracteres de *P* uno a uno hacia atrás. Asimismo, nos basta hacer un muestreo regular en *T*, anotando desde dónde se apuntan estas posiciones muestreadas en el arreglo de sufijos, para poder recuperar los caracteres del texto hacia atrás desde la posición muestreada usando *LF* (una búsqueda binaria en *C* nos dice en la zona de qué carácter de *A* estamos en cada paso). Para esto necesitamos poder acceder a cualquier *T'*[*i*] y, nuevamente, poder calcular $rank_a$ sobre *T'*, y tendremos la capacidad de recuperar cualquier sub-cadena de *T*. Un

mecanismo similar nos permite calcular un valor *A[i]* sin almacenar *A*.

El desafío que nos queda, entonces, es representar *T'* sin usar mucho espacio, y de forma de poder acceder a cualquier carácter y calcular $rank_a$ eficientemente. Con esto seremos capaces de eliminar no sólo *A* sino también *T* mismo, y reemplazar todo por *T'*.

Una estructura extremadamente elegante que permite hacer esto es el *wavelet tree*. Este es un árbol binario balanceado que representará la secuencia S[1,*n*] dividiendo el alfabeto: el árbol tiene σ hojas y log σ niveles. La raíz representa la secuencia completa. Su hijo izquierdo representa la sub-secuencia de *S* formada por los caracteres de la primera mitad, y el hijo derecho, la sub-secuencia formada por los caracteres de la segunda mitad del alfabeto. Recursivamente, los hijos del segundo nivel dividen el alfabeto en cuatro partes y así sucesivamente.

Cada nodo *v* del wavelet tree representa entonces una sub-secuencia de *S*. Sin embargo, ésta no se almacena, sino solamente un bitmap B_v que indica, para cada posición, si el carácter correspondiente pertenece a su hijo izquierdo o derecho. Es fácil ver que el total de bits almacenados en cada nivel siempre suma *n*, y por lo tanto el total de bits que ocupa el wavelet tree es n log σ, lo mismo que si se representara *S* en forma plana. Pronto hablaremos de su compresibilidad.

Supongamos que queremos averiguar *S[i]* a partir del *wavelet tree* de *S*. Sea v el nodo raíz del *wavelet tree*. *Si B[i]=0*, significa que *S[i]* pertenece a la primera mitad del alfabeto, y que por lo tanto el carácter está representado en el hijo izquierdo de *v*. La posición donde está representado depende de cuántos ceros hay en B_v hasta la posición *i*. Es decir, la búsqueda debe continuar por el hijo izquierdo y debemos reescribir $i = rank_0(B_v, i)$. Similarmente, si $B_v[i]=1$, debemos ir al hijo derecho con $i = rank_1(Bv, i)$. Continuamos este proceso hasta llegar a una hoja, la cual corresponde precisamente al carácter *S[i]*. El tiempo total de obtener el carácter es entonces $O(\log \sigma)$.

El procedimiento para calcular $rank_a(S,i)$ es parecido, sólo que la decisión sobre por cuál rama bajar se toma considerando el carácter *a* y no el valor $B_v[i]$. Cuando se llega a la hoja correspondiente al carácter *a*, el valor de *i* es precisamente $rank_a(S,i)$. La siguiente figura ilustra el proceso para $rank_a(S,16) = 5$, donde *S* es precisamente la BWT de *"alabar a la alabarda$"*.

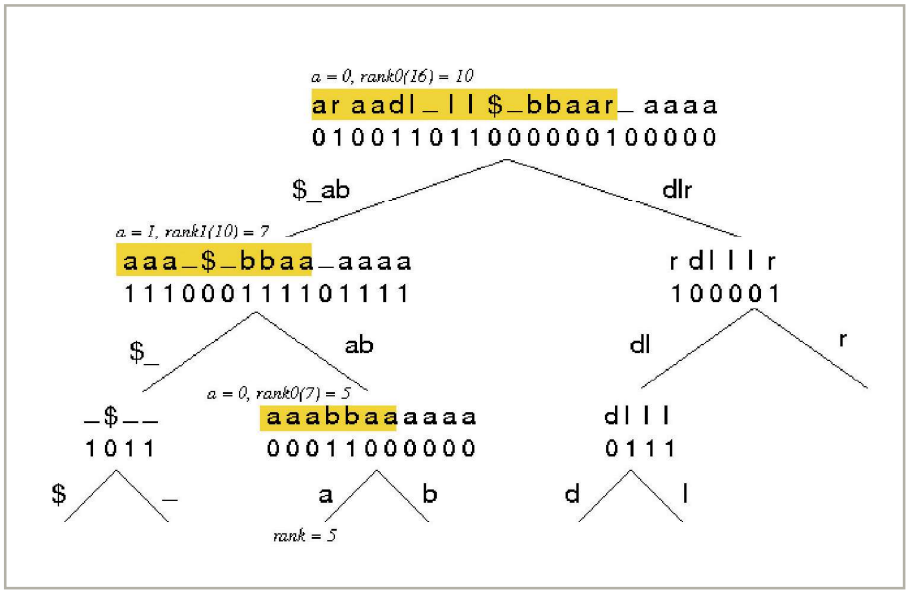
Más aún, si usamos para los bitmaps la representación comprimida de clases y offsets que describimos, resulta que la suma de los $|B_v| H_0(Bv) + o(|B_v|)$ sobre todos los bitmaps del *wavelet tree* totaliza $nH_0(S) + o(n \log \sigma)$, donde $H_0(S)$ es la entropía de orden cero de *S*, $nH_0(S) = \Sigma_a n_a \log n/n_a$, donde *na* es la cantidad de ocurrencias del carácter *a* en *S*.

La entropía de orden cero es una cota inferior a la cantidad de bits por carácter que puede emitir un compresor estadístico de orden cero, es decir, aquél que modela la frecuencia de cada carácter independientemente de los demás. Nótese que la entropía de orden cero de *T'* y de *T* es la misma, por ser permutaciones uno del otro. De este modo se puede obtener un resultado sorprendente. (Las fórmulas donde aparece ε están relacionadas con el período de muestreo mencionado anteriormente.)

Es posible representar un texto T[1,n] sobre un alfabeto σ usando $nH_0(T)+o(n \log \sigma)$ bits, es decir obteniendo compresión, de modo que se puede extraer cualquier substring de largo l del texto en tiempo $O(\log^{1+\varepsilon} n + l \log \sigma)$, contar la cantidad de ocurrencias de un patrón P[1,m] en tiempo $O(m \log \sigma)$, y reportar la posición en el texto de cualquiera de esas ocurrencias en tiempo $O(\log^{1+\varepsilon} n)$, para cualquier constante $\varepsilon > 0$.

Obtuvimos este resultado en 2004 con Paolo Ferragina y Giovanni Manzini, de la Universidad de Pisa, Italia, y con Veli Mäkinen, de la Universidad de Helsinki, Finlandia. La técnica de buscar usando la BWT ya existía (había sido inventada por los coautores italianos en 2000), y el índice se llamaba FM-index. Un problema que tenía era, precisamente, la forma de comprimir *S*. Los *wavelet trees* también existían desde 2003. Nuestra contribución fue notar que los *wavelet trees* se adaptaban maravillosamente bien al problema que tenía el FM-index, y con ello convertimos al FM-index por fin en una estructura competitiva en la práctica. Publicamos el artículo en *SPIRE* ese año [1] y, en 2007, con algunas mejoras, en *ACM Transactions on Algorithms* [2], una de las dos mejores revistas de algoritmos. Con Paolo Ferragina, su estudiante Rossano Venturini y mi alumno de doctorado Rodrigo González, creamos el sitio *PizzaChili*, en http://pizzachili.dcc.uchile.cl y http://pizzachili.di.unipi.it, donde hay colecciones de texto para experimentar y varios índices de este tipo implementados para su uso libre en la investigación, docencia o industria. Este FM-index es, efectivamente, uno de los más competitivos. Publicamos los resultados prácticos en *ACM Journal of Experimental Algorithms* [3].

Algo un tanto insatisfactorio del resultado enunciado es que la compresión de orden cero es bastante pobre en muchos casos. Un compresor que considera los *k* caracteres anteriores como contexto para modelar la frecuencia puede obtener mucha mejor compresión (por ejemplo la técnica PPM, usada en compresores como *ppmdi*, posiblemente el compresor de propósito general más efectivo). La cota inferior para



estos compresores es la entropía de orden k, $H_k(S)$, cuya definición más bien técnica omito.

En el artículo mencionado obteníamos en realidad compresión de orden k, mediante particionar el *wavelet tree* en varios, de acuerdo a los contextos de largo k apuntados por el arreglo de sufijos. La técnica requería un tiempo y espacio de construcción considerable. Durante años varios grupos trabajamos en distintas técnicas para obtener compresión a orden k, todas ellas bastante complejas.

En 2007 trabajábamos con Veli en una versión dinámica de nuestro FM-index, es decir, que manejara una colección de textos donde se pudiera insertar y eliminar textos. No lográbamos dar con una forma de mantener actualizada la partición en varios *wavelet trees* que nos garantizara compresión a orden k. Tal vez esa frustración nos iluminó para que notáramos lo que nadie parecía haber visto antes:

Cuando se aplica el método de compresión de pares (c,o) a los bitmaps del wavelet tree, y este wavelet tree representa la BWT de un texto T[1,n] sobre un alfabeto de tamaño σ, el espacio resulta ser automáticamente $nH_k(T) + o(n \log \sigma)$, para cualquier $k <= \alpha * \log_\sigma n$ y constante $0 < \alpha < 1$.

Esto significa que no había que hacer nada para lograr compresión de orden k, una verdadera ironía después de tanto esfuerzo. Puede ser curioso para el lector el que los experimentos no nos indicaran que el índice se comportaba mejor de lo que nosotros creíamos. La verdad es que no tuvo la oportunidad: nunca habíamos implementado la técnica de los pares *(c,o)*, sino que habíamos optado por usar bitmaps descomprimidos y darle forma de árbol de Huffman al *wavelet tree*, lo cual se veía mucho más práctico y obtenía compresión de orden cero (pero no más que eso). El episodio per se es toda una lección de teoría versus práctica. Luego de este descubrimiento, mi alumno de magíster Francisco Claude implementó la idea y la incorporó a PizzaChili, donde efectivamente resulta ser en muchos casos la mejor alternativa, comprimiendo mucho más que las otras y obteniendo tiempos competitivos.

Publicamos el resultado teórico con Veli en *SPIRE 2007* [4], y luego en 2008 en *ACM Transactions on Algorithms* [5], como parte de un trabajo mucho mayor sobre manejo de colecciones comprimidas y estructuras de datos compactas dinámicas en general. El resultado es el mejor arreglo de sufijos comprimidos existente en términos teóricos, y en cierto sentido cerró la investigación en este tema (si bien hay muchas ramificaciones en las que se sigue trabajando, como memoria secundaria, búsquedas complejas, otros modelos de compresibilidad, etc.). Este trabajo sobre dinamismo, combinado con una mejora posterior obtenida con Rodrigo (publicado en *LATIN 2008* [6] y la versión de revista por aparecer en *Theoretical Computer Science* [7]), también parece haber cerrado el tema del manejo de colecciones dinámicas. Este resultado también permite algo importante: construir el índice utilizando espacio cercano al de la estructura final. Previamente se necesitaba construir la estructura descomprimida para luego comprimirla, lo que plantea un evidente problema de practicidad. En la actualidad mi alumno de magíster Fernando Krell trabaja en implementar esta variante dinámica.

EL ÁRBOL DE SUFIJOS COMPRIMIDO

Parece un juego de palabras astuto decir que, teniendo un arreglo de sufijos comprimido y un árbol comprimido, se tiene un árbol de sufijos comprimido. No está tan lejos de la verdad. Todo nodo del árbol de sufijos se corresponde con un intervalo del arreglo, y no es difícil hacer el mapeo mediante la determinación de cuántas hojas contiene el sub-árbol del nodo. El mapeo inverso requiere determinar el ancestro común más bajo (LCA), que se mencionó anteriormente entre las operaciones que se pueden resolver en árboles comprimidos. Más precisamente, el nodo correspondiente a un intervalo es el LCA de las hojas pertenecientes a las dos puntas del intervalo del arreglo de sufijos. Las cadenas que rotulan las aristas también se pueden obtener: la cadena común entre el primer y último sufijo del intervalo de un nodo v, que el arreglo de sufijos comprimido permite extraer, es la concatenación de los rótulos de las aristas desde la raíz hasta v.

Estas ideas pueden extenderse a una solución que ocupa los $2n$ bits del árbol más el espacio del arreglo de sufijos comprimido, y simula casi todas las operaciones del árbol de sufijos en tiempo constante. Algo insatisfactorio, sin embargo, es que el arreglo de sufijos *determina completamente el árbol*, por lo cual esos $2n$ bits son, estrictamente hablando, redundantes. Esta redundancia se nota mucho cuando el alfabeto es pequeño, como en el caso del ADN. A continuación expondré una solución que prácticamente elimina esa redundancia, al costo de tener tiempos de operación polilogarítmicos.

La idea es no almacenar los nodos del árbol de sufijos, sino trabajar únicamente con intervalos del arreglo. Sin embargo, para poder resolver las operaciones se necesita muestrear algunos nodos del árbol de sufijos, digamos $O(n/s)$, y representarlos explícitamente. Este muestreo debe garantizar que si tomo s veces sucesivamente el suffix-link a partir de un nodo, pasaré necesariamente por un nodo muestreado. No es obvio cómo conseguir ese muestreo, pero tampoco demasiado difícil. Representaremos el árbol de los nodos muestreados usando paréntesis, y utilizaremos también un bitmap que nos permitirá determinar el ancestro muestreado común más bajo (LCSA) entre dos hojas (posiciones del arreglo de sufijos). La siguiente fórmula, válida para todo i, es clave en la solución:

$$sdep(LCA(v,v')) >= i + sdep(LCSA(slink^i(v_l), slink^i(v'_l))),$$

donde $sdep(u)$ es el largo del string que lleva de la raíz al nodo u, $slink^i$ es la aplicación del suffix link i veces, v^l es la hoja izquierda del nodo v, y $v'r$ es la hoja derecha del nodo v'. Debido a la condición del muestreo, debe valer la igualdad para algún i entre 0 y s-1, y esto permite detectar cuál es el nodo muestreado en el camino de suffix links. Para estos nodos muestreados guardamos información explícita como $sdep$, y luego podemos devolvernos de los suffix links ejecutados, obteniendo la información de sdep para el nodo original. La fórmula sirve tanto para obtener LCA(v,v') como para obtener $sdep(v)$ = sdep(LCA(v,v)), y también $slink(v)$ = LCA($slink(v_l), slink(v_r)$).

Lo que se necesita para poder aplicar la fórmula es poder calcular el suffix link de una hoja, o de una posición del arreglo de sufijos. Esta es otra hoja, es decir otra posición del arreglo de sufijos: Si $A[i]=j$, el suffix link de i es la posición i' que apunta a j+1. Nótese que esto es precisamente la inversa de la función LF(i). Esta se puede calcular con la fórmula

$$LF^{-1}(i) = select_a(T', i-C[a]),$$

donde *a* es el carácter con el que empiezan los sufijos de la zona del arreglo de sufijos a la que *i* pertenece, y $select_a(S,j)$ es la inversa de $rank_a$: la posición de la *j*-ésima ocurrencia de *a* en *S*. Se calcula usando el *wavelet tree* de *S* en tiempo $O(\log \sigma)$, recorriéndolo desde la hoja de *a* hacia la raíz, y ejecutando select sobre los bitmaps del *wavelet tree*. El *select* sobre bits es más complejo que el *rank*, pero también se puede resolver en tiempo constante y espacio extra sub-lineal.

Una vez que estas tres operaciones claves sobre intervalos del arreglo de sufijos están resueltas, varias otras se pueden construir sobre ellas o directamente usando el arreglo de sufijos. Por ejemplo, el padre de un nodo v es el menor entre los intervalos $LCA(v_l-1,v_l)$ y $LCA(v_r,v_r+1)$, mientras que el *i*-ésimo carácter de la cadena representada por un nodo *v* es el que corresponde a la zona del arreglo de sufijos donde cae $LF^{-i}(v_l)$. Esta inversa iterada se puede calcular mediante $LF^{-i}(j) = A^{-1}[A[j]+i]$. Tanto *A* como A^{-1} se calculan con los muestreos del texto almacenados en el arreglo de sufijos comprimido.

Eligiendo adecuadamente *s*, podemos tener un árbol de sufijos donde el espacio extra sobre el del arreglo de sufijos es despreciable, y que soporta una amplia gama de operaciones en tiempo $o(\log^2 n)$. Publicamos este trabajo con Luís Russo y su director Arlindo Oliveira, de la Universidad Técnica de Lisboa, Portugal, en *LATIN* 2008 [8], y la versión dinámica en *CPM* 2008 [9]. La implementación muestra que con un 10% de espacio extra sobre el arreglo de sufijos comprimido se obtienen tiempos de operación razonables (entre centésimas y cienmilésimas de segundo, dependiendo de la operación). Para el genoma humano el espacio total sería aproximadamente 1.6 GB (incluyendo la secuencia). Paralelamente, estoy trabajando con mi alumno de magíster Rodrigo Cánovas en implementar en forma práctica otra propuesta publicada con Veli y Johannes Fischer [10], donde los tiempos son sub-logarítmicos y el espacio es algo peor, si bien aún depende de la entropía de orden k.

CONCLUSIONES Y PERSPECTIVAS

En este artículo he querido dar al lector un panorama del tipo de trabajo que se hace en estructuras de datos compactas, mediante la elección de un problema paradigmático y el recorrido por la investigación que llevó a resolverlo. Este camino pasa por varios problemas fundamentales y elegantes de estructuras de datos, y por algunas contribuciones propias. He evitado, para no abrumar al lector, las referencias a cada resultado mencionado, limitándome a las atingentes a mi propia contribución (¡que era lo que al fin y al cabo se me había pedido exponer!). Para quien quiera saber más recomiendo un survey reciente [11].

Las estructuras de datos compactas resultan un área tremendamente atractiva desde varios puntos de vista. A quienes aman la belleza de los algoritmos y las estructuras de datos, les ofrece un campo nuevo donde hay que reinventar soluciones para problemas que están completamente resueltos en el ámbito clásico. A quienes se sienten fascinados por la teoría de la información y la compresión, les brinda una combinación exquisita y sutil con las estructuras de datos, donde el problema trasciende al de meramente representar la información, obligando a investigar tanto entre oportunidades de compresión como de eficiencia de acceso que no existen en la compresión clásica. E invitando a profundizar en la dualidad entre compresión e indexación, ambas basadas de algún modo en extraer las regularidades de los datos. A quienes disfrutan con las soluciones prácticas a desafíos formidables, les ofrece una herramienta fresca y poderosa para hacer frente a la masividad de la información típica de nuestros tiempos. Y a quienes toda la vida hemos tratado de caminar por el estrecho desfiladero entre la teoría y la práctica, intentando no caer en la teoría inútil ni en la práctica vacía de fundamento científico, nos da un maravilloso ejemplo de cómo ambas pueden hacer milagros cuando trabajan juntas.

Para finalizar doy las gracias a mis ex-alumnos Francisco Claude y Rodrigo Paredes, quienes aportaron con valiosas sugerencias. BITS

REFERENCIAS

[1] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An Alphabet-Friendly FM-index. *Proc. SPIRE'04*, pages 150-160. LNCS 3246.

[2] Paolo Ferragina, Giovani Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed Representations of Sequences and Full-Text Indexes. *ACM Transactions on Algorithms (TALG)* 3(2), article 20, 24 pages, 2007.

[3] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed Text Indexes: From Theory to Practice. *ACM Journal of Experimental Algorithmics (JEA)* 13:article 12, 2009. 30 pages.

[4] Veli Mäkinen and Gonzalo Navarro. Implicit Compression Boosting with Applications to Self-Indexing. *Proc. SPIRE'07*, pages 214-226. LNCS 4726.

[5] Veli Mäkinen and Gonzalo Navarro. Dynamic Entropy-Compressed Sequences and Full-Text Indexes. *ACM Transactions on Algorithms (TALG)* 4(3):article 32, 2008. 38 pages.

[6] Rodrigo González and Gonzalo Navarro. Improved Dynamic Rank-Select Entropy-Bound Structures. *Proc. LATIN'08*, pages 374-386. LNCS 4967.

[7] Rodrigo González and Gonzalo Navarro. Rank/Select on Dynamic Compressed Sequences and Applications. To appear in *Theoretical Computer Science*.

[8] Luís Russo, Gonzalo Navarro, and Arlindo Oliveira. Fully-Compressed Suffix Trees. *Proc. LATIN'08*, pages 362-373. LNCS 4957.

[9] Luís Russo, Gonzalo Navarro, and Arlindo Oliveira. Dynamic Fully-Compressed Suffix Trees. *Proc. CPM'08*, pages 191-203. LNCS 5029.

[10] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. An(other) Entropy-Bounded Compressed Suffix Tree. *Proc. CPM'08*, pages 152-165. LNCS 5029.

[11] Gonzalo Navarro and Veli Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys* 39(1), article 2, 61 pages, 2007.