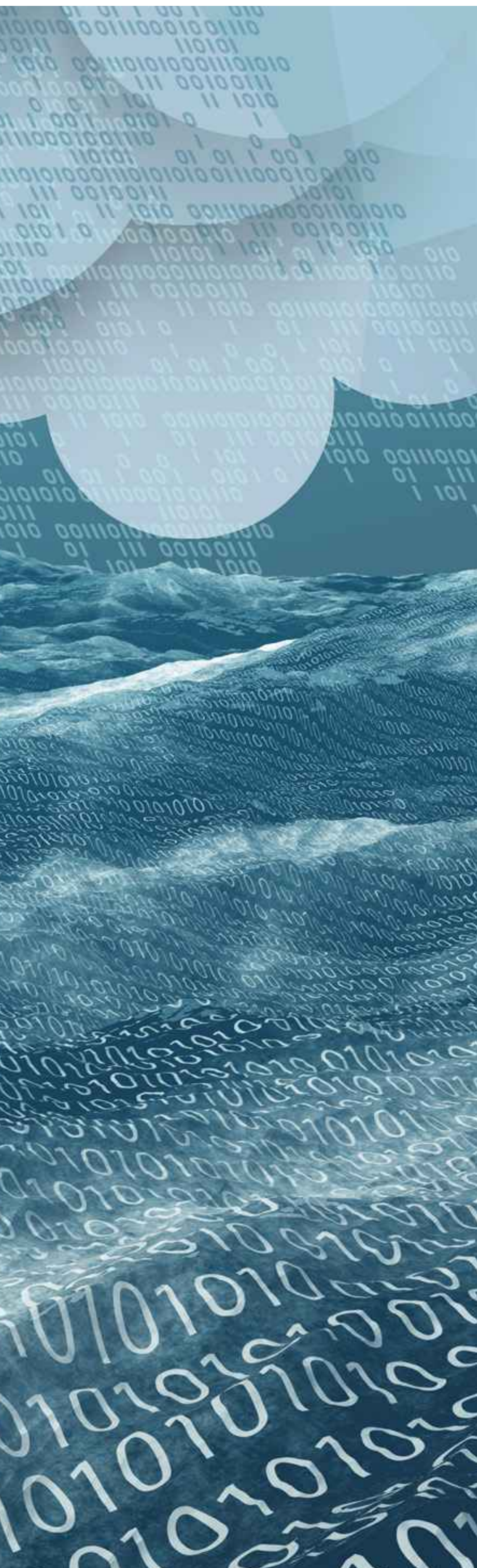
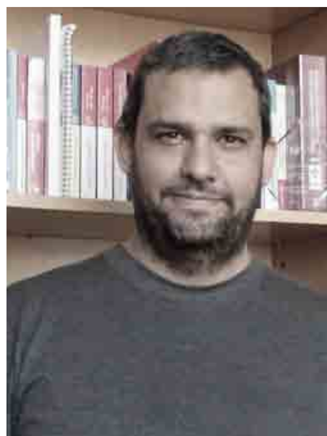
EL DILUVIO DE DATOS Y LA REPETITIVIDAD: NUESTRO MEJOR ENEMIGO*

Uno de los aspectos que mejor caracteriza nuestro desarrollo tecnológico en las últimas décadas es el rápido crecimiento de los datos disponibles digitalmente. En pocos años, hemos pasado de es tar preocupados por la cantidad de cómputo que debíamos realizar a preocuparnos más bien por el volumen de datos que debemos procesar, incluso cuando los cálculos son relativamente simples .

*Financiado por el Proyecto Fondecyt 1-140796





GONZALO NAVARRO
Profesor Titular Departamento de Ciencias de la Computación, Universidad de Chile. Doctor en Ciencias mención Computación y Magíster en Ciencias mención Computación, Universidad de Chile; Licenciatura en Informática, Universidad Nacional de La Plata y ESLAI, Argentina. Líneas de investigación: Diseño y Análisis de Algoritmos, Bases de Datos Textuales, Búsqueda por Similitud, Compresión.
gnavarro@dcc.uchile.cl

EL DILUVIO DE DATOS

Tenemos por un lado los datos que podríamos llamar "científicos": los datos astronómicos, donde Chile es una fuente generadora predilecta; datos de clima obtenidos por sensores de todo tipo, que en varios casos deben ser usados en tiempo real para ser útiles; datos geográficos, incorporados a la vida diaria gracias a los satélites e intermediarios como Google Maps; secuencias biológicas, cada vez más baratas de obtener y por ello más abundantes; etc.

Tenemos también datos que podríamos llamar "contenidos": grabaciones de voz, imágenes, fotos, música, vídeos, libros, revistas, páginas web, emails, blogs, tweets, y un largo etcétera, todos ellos incorporando la necesidad de encontrar el contenido deseado en formatos no siempre amigables para el procesamiento automático.

En una tercera categoría, que podríamos llamar "comportamiento", tenemos evolución de mercados, de compras online, tráfico en redes, *clicks* de usuarios, patrones de navegación en la Web, búsquedas en la Web, "likes" en Facebook, dinámica de redes sociales, etc. Estos datos se analizan permanentemente para predecir el comportamiento económico, mejorar máquinas de búsqueda, ofrecer mejores interfaces, estudiar comportamientos sociales, etc.

Tal cantidad de datos genera inmensas posibilidades de mejorar muchos aspectos de nuestra vida, y ya lo han hecho en gran medida. Dudo que quienes nacieron en la era de los buscadores puedan siquiera imaginar cómo era vivir cuando esa fuente instantánea e inagotable de información no estaba disponible. Mucho más está por venir también. Por ejemplo, la posibilidad de descubrir efectos de medicamentos no porque algún científico se ilumine y lo pruebe sino porque surja automáticamente del análisis y correlación de miles de historias clínicas, la posibilidad de que el automóvil se conduzca solo a través de una ciudad desconocida mediante interactuar con dispositivos inteligentes de la ciudad que lo guíen con información extraída automáticamente de la Web, y muchos otros ejemplos que hace una década eran ciencia ficción hoy están a la vuelta de la esquina.

Con las grandes posibilidades vienen los grandes desafíos. En cada una de las áreas mencionadas, hemos aprendido que es mucho más fácil generar datos que darles un uso útil. La humanidad genera unos cuantos zettabytes de datos al año, de acuerdo a un estudio de 2011 de la International Data Corporation (un zettabyte son 2^{70} bytes, aproximadamente 10^{21}), y el volumen crece a más del doble por año, más rápido que la Ley de Moore, que gobierna (en el mejor de los casos) el crecimiento de las capacidades de hardware. El libro "Algorithms and Data Structu-

res for External Memory" de Jeff Vitter comienza con una sentencia que resume la situación: *The world is drowning in data!* El término *data deluge*, o la inundación en datos, se ha puesto de moda para indicar que tenemos muchos más datos de los que podemos manejar.

Por lo pronto, no parece que nuestras capacidades de mero almacenamiento de datos estén aún comprometidas. Pero como decía, el problema no es realmente almacenar los datos sino darles algún uso. Incluso transmitirlos puede ser un problema: la compañía BGI, que secuencia datos en China, les envía a sus clientes las secuencias en una cinta por correo normal, ya que Internet no tiene el ancho de banda necesario para transmitirlas más rápidamente. ¿Qué esperar entonces de nuestras capacidades de procesar estos datos para obtener información útil? Realizar minería de textos o de datos en general para encontrar patrones relevantes, recorrer grafos para detectar comunidades de interés, buscar similitudes en secuencias genómicas, etc. son procesos que requieren ejecutar sofisticados algoritmos que recorran los datos, una o varias veces, con patrones de recorrido impredecibles. ¿Podemos realizar este procesamiento sobre datos almacenados en memoria secundaria (discos, SSDs) o terciaria (cintas)?

Lamentablemente, no todos los componentes del hardware mejoran según la Ley de Moore (que establece que la capacidad y/o la velocidad se duplica cada dos años aproximadamente). Lo hace la capacidad de la memoria externa (secundaria y terciaria), pero no su velocidad. Desde hace unos años, ya no lo hace la velocidad de los procesadores, aunque a cambio lo ha empezado a hacer la cantidad de procesadores en la CPU. La memoria interna (RAM) crece en capacidad, pero no mucho en velocidad, aunque aparecen cada vez más y mayores memorias caché rápidas entre la CPU y la memoria RAM. La diferencia entre procesar datos en memoria interna o externa es cada vez mayor, lo que ha hecho atractivo usar *clusters* de computadores para simular una gran memoria RAM.

LA REPETITIVIDAD: NUESTRA TABLA DE SALVACIÓN

Muchas de las secuencias, fotos, series de tiempo, etc. de las fuentes más prolíficas de datos son altamente *repetitivas*. Esto significa que es posible obtener cada nuevo elemento mediante combinar unas pocas partes de elementos vistos previamente. La repetitividad parece llamada a ser nuestra tabla de salvación frente al diluvio de datos. Un buen ejemplo son las colecciones de genomas. Existen muchas especies distintas, pero tampoco tantas. Las grandes colecciones de genomas se forman secuenciando muchos individuos de la misma especie. Dos genomas de individuos de la misma especie difieren en un porcentaje muy bajo, cercano al 0.1%. Otro ejemplo son los sistemas de versiones (de software, de documentos, etc.), con el caso públicamente disponible de *Wikipedia* y la *Internet Wayback Machine* (que almacena copias de páginas web a través del tiempo). Otra son las publicaciones de series de tiempo, financieras, climatológicas, etc., que en muchos casos tienen periodicidad con pocas variaciones. Los datos astronómicos suelen incluir "fotos" de la misma zona del cielo en distintos momentos

Se conocen buenas técnicas para tratar con datos muy repetitivos. Los sistemas que manejan versionamiento en edición colaborativa, como *CVS, Subversion,* y otros, almacenan solamente las diferencias de versiones menores con respecto a la versión anterior. Compresores genéricos como la familia Lempel-Ziv buscan, en el texto ya visto, copias del nuevo texto que deben comprimir. Elementos de esta familia como *p7zip* funcionan muy bien en colecciones altamente repetitivas, donde los compresores clásicos fallan en aprovechar la repetitividad.

Pero nuevamente estamos hablando de almacenar los datos eficientemente, permitiendo a lo sumo recrear una versión específica. Si queremos tratar estas grandes colecciones repetitivas en forma eficiente, debemos ser capaces de *operar sobre los datos directamente en forma comprimida*, sin necesidad de descomprimirlos previamente.

ESTRUCTURAS DE DATOS COMPACTAS

Aquí entran en juego las *estructuras de datos compactas*, que combinan compresión y teoría de la información con estructuras de datos, de modo de representar los datos usando espacio cercano a lo que usaría un compresor, pero aún permitiendo navegar, operar y buscar en los datos sin descomprimirlos. En este aspecto van más allá de la compresión, que busca solamente representar los datos en menos espacio de modo de poder recuperarlos, y generalmente requiere descomprimirlos del todo antes de poder usarlos.

Las estructuras de datos compactas se han venido desarrollando cada vez con más fuerza desde fines de los ochenta, y hoy permiten representar una amplia gama de estructuras en espacio reducido: árboles, grafos, textos, grillas de puntos, etc. Pueden llegar a usar, en la práctica, treinta veces menos espacio que la estructura clásica correspondiente, y si eso permite que los datos que normalmente se almacenarían en memoria externa quepan en RAM, entonces la ganancia en tiempo de procesamiento es de órdenes de magnitud, incluso cuando manejar las estructuras compactas requiera bastantes más operaciones que las clásicas. En el caso de una memoria RAM distribuida en *clusters*, las estructuras compactas permiten reducir el número de computadores necesarios, su tiempo de comunicación, y su costo de hardware y de energía.

El desarrollo de estructuras de datos compactas específicas para explotar la repetitividad de los datos es bastante más reciente, y tiene todavía mucho terreno por explorar. Para ejemplificar el

tipo de problemas que enfrentamos al explotar la repetitividad, describiré el desarrollo de índices para colecciones de texto repetitivas, uno de los temas que se abordan en mi actual proyecto Fondecyt. Lo haré maximizando la intuición a través de ejemplos, evitando demostraciones o argumentos complejos.

COLECCIONES REPETITIVAS DE TEXTO

Para permitir búsquedas de diversa complejidad en colecciones de secuencias (por ejemplo secuencias de ADN), los índices favoritos son los *árboles de sufijos*, inventados en 1973 por Peter Weiner. Considere una colección formada por textos T_1, T_2, ..., T_d, sobre un alfabeto Σ. Usaremos un símbolo extra, $, para terminar los textos y los concatenaremos en un único texto $T[1,n] = T_1 \$ T_2 \$ \dots T_d \$$. Los árboles de sufijos se construyen sobre este texto concatenado *T*. Un *sufijo* de *T* es cualquier substring de *T* que termina en un $. El *trie de sufijos* de *T* se obtiene insertando todos sus sufijos en un árbol digital, de modo que cualquier sufijo *T[i...]* se puede leer recorriendo el árbol desde la raíz hasta la hoja asociada al sufijo que empieza en la posición "i" y concatenando las letras que rotulan las aristas.

El árbol de sufijos se obtiene compactando los caminos "unarios" (es decir, donde cada nodo tiene un único hijo) en el trie de sufijos. Ahora cada arista está rotulada con un string y los strings que rotulan las aristas de cada nodo a sus hijos difieren en su primera letra. Como tiene *n* hojas y todo nodo interno tiene al menos dos hijos, el árbol de sufijos tiene menos de 2*n* nodos. También se puede construir en tiempo lineal, *O(n)*. **La Figura 1** muestra un árbol de sufijos para un solo string.

Podríamos dedicar muchas páginas a mostrar las virtudes y potencialidades del árbol de sufijos, pero nos contentaremos con un ejemplo simple e ilustrativo. Considere un problema de análisis de textos (para determinar autoría o plagio en textos de lenguaje natural, para buscar secuencias repetidas de ADN, etc.): encuentre el substring más largo de T que aparece dos veces. En un árbol de sufijos, esto corresponde simplemente a recorrer los nodos y encontrar el nodo interno que representa el string más largo, lo cual puede hacerse fácilmente en tiempo lineal (en nuestro ejemplo, el resultado es "ATA"). Sumado a que este árbol se puede construir en tiempo lineal, tenemos que este problema se puede resolver en tiempo *O*(*n*). Pues bien, antes de que existieran los árboles de sufijos, incluso el legendario Donald Knuth creía que tal cosa no era posible. Esto ilustra la facilidad con la que el árbol de sufijos permite resolver problemas muy complejos.

Asimismo, podemos usar el árbol de sufijos para buscar las ocurrencias de un nuevo patrón, *P*[1,*m*], en *T*. Simplemente hay que bajar por las letras de *P* hasta que (1) no se pueda continuar, en cuyo caso *P* no aparece en *T*; (2) lleguemos a una hoja, en cuyo caso hay que comparar el resto de *P* con esa posición del texto, para ver si *P* aparece allí o no aparece; o (3) se nos acaben las letras de *P*, en cuyo caso el nodo en que estamos, o donde nos lleva la arista en que estamos, corresponde a todas las ocurrencias de *P* en *T*, una por hoja (y las posiciones donde ocurre *P* son los valores asociados a las hojas). En nuestro ejemplo, tenemos búsquedas de tipo (1), (2) o (3) cuando buscamos "TT", "CGA", o "AT", respectivamente. El árbol de sufijos realiza esta búsqueda en tiempo *O*(*m*), independientemente del largo del texto, lo que es notable.

Esta poderosa herramienta tiene, sin embargo, un serio problema: ocupa mucho espacio. En una implementación típica, el árbol de sufijos puede ocupar 20*n* bytes. Si tenemos un genoma humano (unos 3 mil millones de bases), el árbol de sufijos requiere unos 60GB de memoria. Además los recorridos en el árbol siguen caminos impredecibles (no todos los problemas se resuelven con recorridos top-down, por lo que el árbol debe mantenerse en memoria para que los recorridos funcionen eficientemente). Si 60 GB de RAM parece una exigencia considerable, piense en la razonable aspiración de manejar una colección de 1000 genomas: ¡requeriría 60 terabytes de RAM!

Sin embargo, la repetitividad es un poderoso aliado para reducir el espacio de los árboles de sufijos de colecciones repetitivas. La **Figura 2** muestra una colección de varios strings similares.

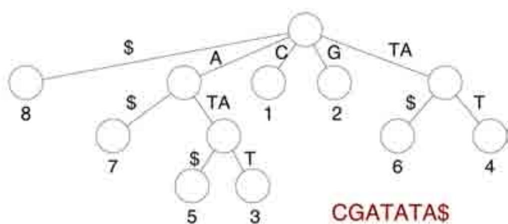


FIGURA 1.
ÁRBOL DE SUFIJOS PARA UN SOLO STRING. COMO LOS STRINGS QUE LLEGAN A LAS HOJAS SON MUY LARGOS, ALMACENAMOS SÓLO SU PRIMERA LETRA.

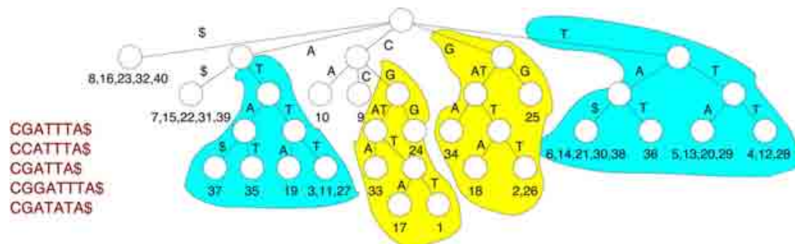


FIGURA 2.
ÁRBOL DE SUFIJOS PARA UNA COLECCIÓN DE VARIOS STRINGS SIMILARES.

Hemos mostrado, en celeste y amarillo, dos zonas del árbol que son topológicamente idénticas. *La repetitividad en los textos induce zonas topológicamente idénticas en el árbol de sufijos.* Veremos luego cómo explotar esa repetitividad. Veamos, por lo pronto, lo que ocurre con las posiciones asociadas a las hojas del árbol, como se muestra en la **Figura 3**.

Las áreas del mismo color son iguales, excepto que sus valores difieren en una unidad (estas áreas se traslapan, como puede verse). Nuevamente, veremos luego cómo explotar este hecho: *La repetitividad en los textos induce largas áreas en las posiciones del árbol de sufijos cuyos valores difieren en una unidad*.

La primera representación compacta de árboles de sufijos fue propuesta por Kunihiko Sadakane en 2007. Consta de tres partes: la topología comprimida, la secuencia de posiciones (que se llama *arreglo de sufijos*) y otro arreglo de prefijos comunes más largos (*LCP, por longest common prefix*). Este arreglo indica el largo del prefijo común que comparten dos hojas consecutivas del árbol. Estos tres componentes son suficientes para simular todas las operaciones en un árbol de sufijos. Ni siquiera se necesita el texto. Incluso se puede extraer de la estructura cualquier substring del texto que se desee, ¡por lo que el texto puede borrarse!

La **Figura 4** muestra el arreglo LCP de nuestra colección. Como puede verse, persiste el mismo tipo de repetitividad que en el arreglo de sufijos, a veces con excepción del primer elemento de las áreas. *La repetitividad en los textos induce largas áreas en el arreglo LCP cuyos valores difieren en una unidad.*

El árbol de sufijos compacto de Sadakane obtiene un espacio cercano a unos 10 *n* bits para el caso de ADN. Esto es 1,25 bytes por símbolo, muy inferior a los 20 de las implementaciones clásicas. Un genoma humano cabe en menos de 4GB, que es muy razonable para una memoria principal. Sin embargo, para nuestro razonable objetivo de 1000 genomas, una memoria principal de 4TB es todavía de muy alta gama. Mostraremos cómo se ha logrado reducir este espacio a unos 2*n* bits para el caso de las colecciones repetitivas, con lo cual los 1000 genomas humanos cabrían en menos de 700GB.

ÁRBOLES DE SUFIJOS PARA COLECCIONES REPETITIVAS

Mostraremos cómo aprovechamos la repetitividad para obtener un árbol de sufijos tan pequeño. La idea es compactar cada uno de los componentes utilizando distintas herramientas. Una idea genérica que es útil es la compresión basada en *gramáticas libres del contexto*. Estas representan una secuencia de símbolos como una serie de *reglas*, donde un símbolo *no terminal* se reescribe como una secuencia de símbolos *terminales* (los de la secuencia) y otros no terminales. Los no terminales, a su vez, se reescriben usando las reglas, hasta que la secuencia contiene sólo símbolos terminales. Las secuencias repetitivas pueden así representarse como la cadena que se deriva de un cierto símbolo *no terminal*, que se llama *símbolo inicial*, y la gramática puede ser mucho menor que la secuencia.

Para ilustrar esta idea, apliquémosla a la versión *diferencial* de nuestro arreglo de sufijos. En la versión diferencial, cada elemento se escribe como la diferencia con el anterior. Esto hace que las zonas largas que se repetían en otra parte con una diferencia de una unidad, ahora se convierten en zonas idénticas.

Nuestro arreglo de sufijos diferencial se ve así tal como se muestra en la **Figura 5**.

Como se puede ver, las repeticiones se han conservado (excepto por los primeros símbolos). Gracias a estas repeticiones, el arreglo se puede representar con una gramática bastante más

FIGURA 3.
ARREGLO DE SUFIJOS PARA NUESTRA COLECCION.

FIGURA 4.
ARREGLO LCP PARA NUESTRA COLECCION.

corta, de 33 símbolos (contando los largos de los lados derechos de las reglas, que es cercano al costo real de almacenarla) en vez de 40, donde *S* es el símbolo inicial:

$$S \rightarrow 8\ B\ \text{-}33\ E\ \text{-}2\ C\ D\ \text{-}17\ \text{-}1\ 24\ C\ 23\ 10\ C\ 24\ \text{-}1\ \text{-}19\ E\ \text{-}31\ A\ \text{-}25\ D$$
$$A \rightarrow 8\ 7\ 9$$
$$B \rightarrow A\ 8$$
$$C \rightarrow \text{-}16\ \text{-}16$$
$$D \rightarrow 8\ 16$$
$$E \rightarrow B\ \text{-}2$$

La diferencia sería más impresionante en un ejemplo más largo, claro. Veremos ahora cómo representar cada componente del árbol de sufijos compacto.

TOPOLOGÍA DEL ÁRBOL: la representación compacta clásica de un árbol de *N* nodos usa 2*N* bits: se recorre el árbol en preorden abriendo un paréntesis al llegar a un nodo y cerrándolo al dejarlo definitivamente. Esta secuencia se parte en bloques, y se construye una jerarquía sobre los bloques. Para cada nodo de esta jerarquía se almacenan datos como el exceso total (suma de paréntesis que abren menos paréntesis que cierran), el mínimo y máximo exceso al recorrer el área de izquierda a derecha, etc. Esta jerarquía es suficiente para permitir navegar los paréntesis eficientemente, como si tuviéramos el árbol. La secuencia de paréntesis correspondiente a nuestro árbol de sufijos sobre la colección repetitiva es lo que se muestra en la **Figura 6.**

Como puede verse, los subárboles idénticos se traducen en substrings idénticos en la secuencia de paréntesis. Podemos entonces representarla usando una gramática, y asociar la información sobre excesos a los no terminales de la gramática. Esto produce una representación mucho menor que los 2*N* bits cuando la secuencia es repetitiva (nótese que nuestro árbol de sufijos tiene hasta *N*=2*n* nodos, por lo que puede ocupar hasta 4*n* bits). Con una representación de gramáticas, el espacio para colecciones repetitivas ocupa unos *n* bits. Una gramática para generar nuestra secuencia de paréntesis podría ser como sigue:

$$S \rightarrow E\ E\ A\ )\ C\ B\ )\ B\ A\ )$$
$$A \rightarrow (\ D\ D\ )$$
$$B \rightarrow (\ E\ D\ )\ (\ )$$
$$C \rightarrow E\ (\ )$$
$$D \rightarrow C\ )$$
$$E \rightarrow (\ (\ )$$

Existen incluso representaciones de árboles de sufijos que no utilizan en absoluto los paréntesis, sino que los deducen de los otros componentes, especialmente del arreglo LCP, el que se almacena en forma de gramática para agilizar las operaciones. Para secuencias muy repetitivas, pueden llegar a necesitar sólo 1.5*n* bits en total, pudiendo almacenar 1000 genomas en casi 500GB. Sin embargo, son también varios órdenes de magnitud más lentos.

Exploramos la vía de eliminar del todo la topología del árbol con mi exalumno memorista Andrés Abeliuk en 2012, obteniendo el menor árbol de sufijos hasta el momento. Con mi alumno de Doctorado Alberto Ordóñez, de la Universidade da Coruña, exploramos la variante de usar gramáticas para comprimir la topología en 2014, resultando en una estructura algo mayor, pero hasta mil veces más rápida que la que prescinde completamente del árbol.

ARREGLO DE SUFIJOS: es posible obtener una representación compacta del arreglo de sufijos usando gramáticas, como vimos en el ejemplo. Probamos esta idea en 2007 con mi alumno de Doctorado Rodrigo González. Si bien el acceso al arreglo de sufijos es muy rápido, la compresión que se alcanza no es competitiva con otras técnicas. Una que funciona mucho mejor, propuesta por Sadakane en 2000, es reemplazar el arreglo de sufijos *A*[1,*n*] por otro que se llama Ψ[1,*n*], que contiene en Ψ[*i*] la posición *j* donde

8 8 7 9 8 -33 8 7 9 8 -2 -2 -16 -16 8 16 -17 -1 24 -16 -16 23 10 -16 -16 24 -1 -19 8 7 9 8 -2 -31 8 7 9 -25 8 16

FIGURA 5.
ARREGLO DE SUFIJOS DIFERENCIAL PARA NUESTRA COLECCIÓN.

(( ) (( ) ((( ) ( )) (( ) ( )))) (( ) ( ) ((( ) (( ) ( ))) ( ))) ((( ) (( ) ( ))) ( )) ((( ) ( )) (( ) ( ))))

FIGURA 6.
REPRESENTACIÓN CON PARÉNTESIS DE LA TOPOLOGÍA DE NUESTRO ÁRBOL DE SUFIJOS.

$A[j]=A[i]+1$ (excepto si $A[i]=n$, en cuyo caso contiene la posición j donde $A[j]=1$). Requeriría mucho espacio explicar cómo este nuevo arreglo puede reemplazar al arreglo de sufijos. En cambio, nos centraremos en sus propiedades. **La Figura 7**, muestra el arreglo Ψ de nuestro ejemplo.

Hay dos cosas importantes que resaltar: primero, el arreglo es creciente en la zona de los sufijos que empiezan con cada letra (las letras iniciales se indican abajo). Con un poco de cuidado eso puede valer para la zona de los $ también. Esto vale para cualquier tipo de texto y permite codificar el arreglo Ψ en un espacio cercano al texto comprimido con un compresor simple, tipo Huffman. Esto representa mucho menos espacio que el arreglo de sufijos original.

El segundo hecho es relevante para las colecciones repetitivas: las zonas coloreadas se convierten en áreas de Ψ donde el arreglo crece de a una unidad, y por lo tanto es muy fácil de codificar en muy poco espacio. Esto ocurre en todas las zonas de cada color excepto en una, donde los números son mayores. Nuevamente, no tenemos suficiente espacio para explicar las razones por las que esto ocurre. Con Veli Mäkinen y sus alumnos, de la University of Helsinki, publicamos en 2008 el Run-Length Compressed Suffix Array, que usa esta idea y se ha convertido en la principal estructura para utilizar en esta situación. En colecciones repetitivas, esta estructura usa unos 0.7n bits.

ARREGLO LCP: como mencionamos, el arreglo LCP se puede comprimir usando una gramática y eso es buena idea cuando no se representa la topología del árbol, pues se necesita accederlo muy frecuentemente. Cuando se tiene la topología, los accesos a LCP son mucho más esporádicos y se puede utilizar una representación más compacta. Sadakane propuso en 2007 utilizar a cambio el arreglo *PLCP* (Permuted LCP), donde $PLCP[i] = LCP[A[i]]$, es decir, almacena los mismos valores pero en orden de texto. Esto es conveniente porque, en ese orden, vale que $PLCP[i+1] >= PLCP[i]-1$. Esto permite marcar los valores $PLCP[i]+2i$ (que es una secuencia estrictamente creciente) como 1s en un vector de 2n bits, y recuperar eficientemente la posición del i-ésimo 1 del vector para encontrar el valor de $PLCP[i]$. Además de esta notable reducción de espacio, se puede ganar aún más en las colecciones repetitivas. El arreglo *PLCP* y el vector de bits para nuestro ejemplo se muestran en la **Figura 8.**

Obsérvese cómo aparecen largas zonas decrecientes en una unidad en PLCP, que se traducen en largas zonas de 1s (y por lo tanto de 0s, ya que hay n de cada uno) en el vector de bits. Esta propiedad, que demostramos con Veli Mäkinen y Johannes Fischer, del KIT Alemania, en 2008, hace al vector de bits mucho menor que 2n bits en colecciones repetitivas, como confirmamos con mi exalumno Andrés Abeliuk: ¡se llega a usar unos 0.2n bits!

Sumando los n bits de los paréntesis, los 0.7 n bits del arreglo de sufijos, y los 0.2n bits del LCP, tenemos los 2n bits mencionados. Sacando los paréntesis y agregando una compresión de gramáticas para LCP, obtenemos los 1,5 n bits ya mencionados también. Cuando se incluye la topología del árbol, la estructura realiza las operaciones en microsegundos, mientras que cuando no, se llega a los milisegundos. Para poner estos resultados en contexto, la representación que usa 2n bits es aún diez veces más lenta que el árbol de sufijos compacto de Sadakane (pero también ocupa cinco veces menos espacio, y va ocupando menos a medida que la repetitividad aumenta). Un árbol de sufijos clásico ¡ocupa 320 veces más espacio que la representación usando paréntesis comprimidos con gramáticas!

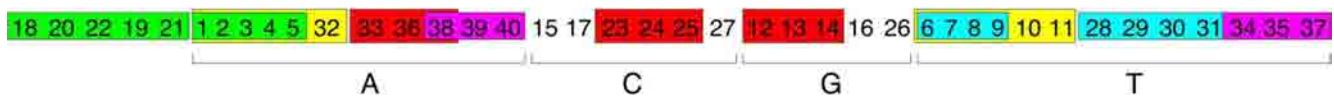


FIGURA 7.
ARREGLO Ψ DE NUESTRA COLECCION.

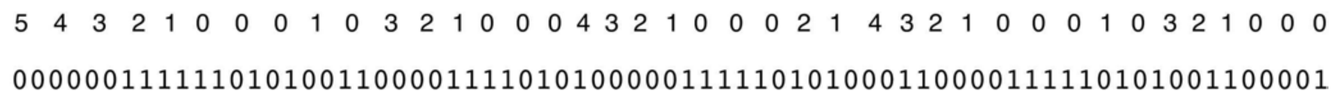
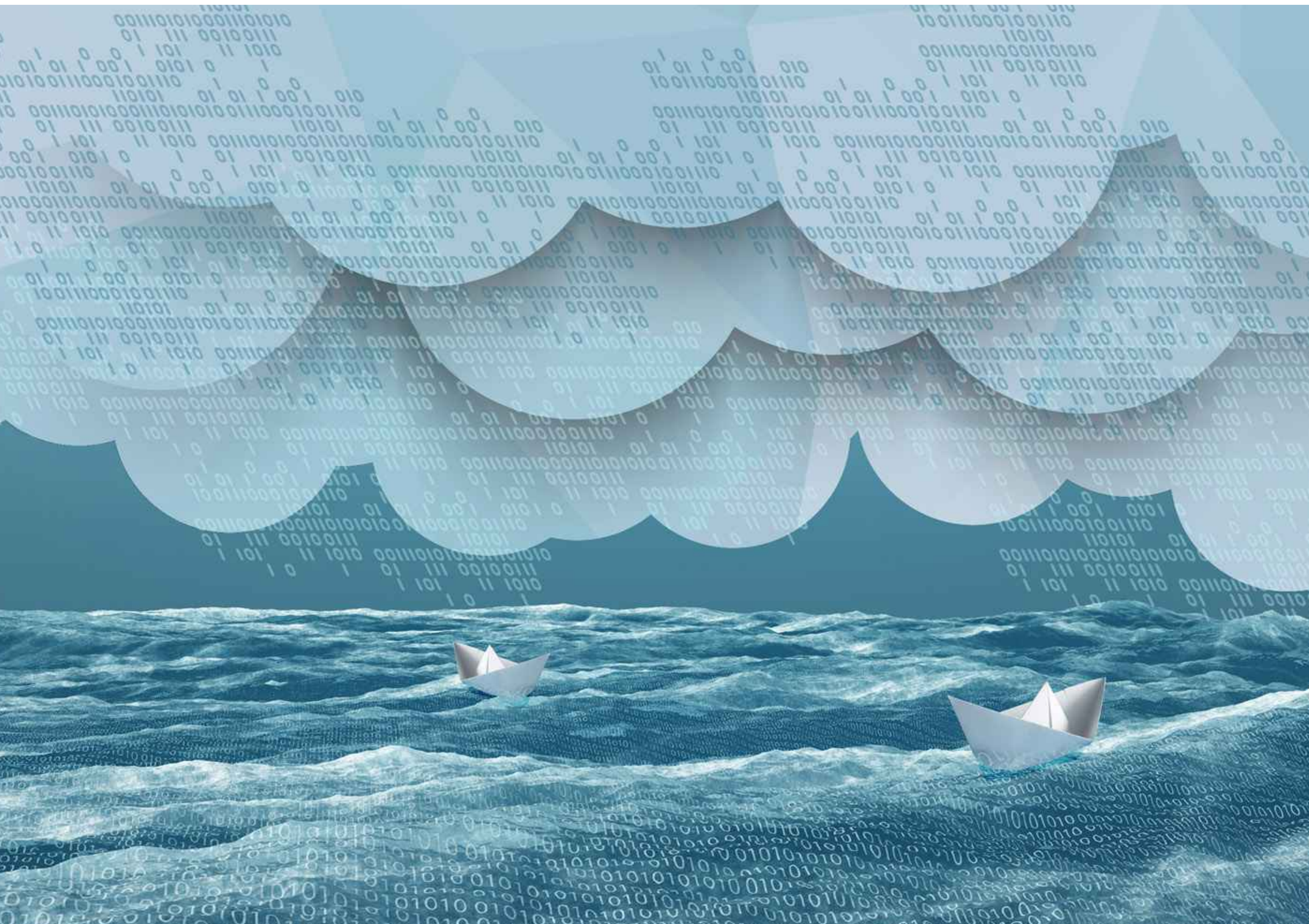


FIGURA 8.
ARREGLO PLCP Y SU REPRESENTACIÓN COMO BITS PARA NUESTRA COLECCIÓN.



CONCLUSIONES

EXPLOTAR LA REPETITIVIDAD DE LOS DATOS PUEDE SER LA CLAVE PARA AFRONTAR EL DILUVIO DE DATOS QUE SE AVECINA RÁPIDAMENTE. EL ENTENDER CÓMO LA REPETITIVIDAD EN LOS DATOS SE MANIFIESTA EN LAS DISTINTAS ESTRUCTURAS DE DATOS QUE USAMOS PARA ACCEDER EFICIENTEMENTE A ELLOS PERMITE DISEÑAR VARIANTES DE ESTAS ESTRUCTURAS QUE REDUCEN SU ESPACIO EN ÓRDENES DE MAGNITUD. MOSTRAMOS CON CIERTO DETALLE EL CASO DE LOS ÁRBOLES DE SUFIJOS, UNA HERRAMIENTA MUY IMPORTANTE PARA EL ANÁLISIS DE SECUENCIAS. EL SEGUIR EL MISMO CAMINO CON OTRAS ESTRUCTURAS Y OTROS TIPOS DE DATOS REQUIERE UN TRABAJO ALGORÍTMICO MUY ESTIMULANTE EN UN ÁREA QUE SE CONSTRUYE SOBRE UNA LARGA TRADICIÓN DE INVESTIGACIÓN EN ALGORITMOS Y ESTRUCTURAS DE DATOS Y TEORÍA DE LA INFORMACIÓN, PERO QUE A LA VEZ ESTÁ CASI INEXPLORADA, ESPERANDO LA CONTRIBUCIÓN DE NUEVOS INVESTIGADORES. ■