Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Patrones de repetición para clasificación e identificación de proteínas

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Pablo Diego Rago
LU: 239/10

Director: Pablo Turjanski

Buenos Aires, 2017

PATRONES DE REPETICIÓN PARA CLASIFICACIÓN E IDENTIFICACIÓN DE PROTEÍNAS

La proteínas son grandes biomoléculas formadas por cadenas lineales de aminoácidos. Una abstracción posible de esta estructura permite pensar a las mismas como cadenas de caracteres, donde cada aminoácido se corresponde con un caracter, y así es posible representar su estructura primaria.

Algunas proteínas naturales presentan patrones estructurales recurrentes. Estas moléculas pueden ser clasificadas de acuerdo al largo de la mínima unidad de repetición (fibrilares, repetitivas y globulares). Las proteínas repetitivas a pesar de ser muy similares a nivel de estructura terciaria, pueden tener repeticiones extremadamente variables a nivel de estructura primaria.

En su trabajo del año 2016, Turjanski et al proponen una definición matemática de repetición para buscar ocurrencias de estas secuencias en diferentes familias de proteínas. En este trabajo el grupo describe que cadenas largas de repeticiones exactas son infrecuentes en proteínas particulares, incluso para aquellas que se conoce que se pliegan en estructuras de motivos estructurales recurrentes. Por otro lado, también detallan que estas proteínas son repetitivas dentro de sus familias, exhibiendo cadenas de aminoácidos que son repeticiones exactas provenientes de la familia de referencia. Como producto de esto, proponen un algoritmo para cuantificar las chances de que una proteína particular pertenezca a una cierta familia.

En el presente trabajo se estudian y proponen modificaciones al algoritmo mencionado para mejorar su eficiencia a nivel performance computacional y calidad de resultados obtenidos. Para ello se exploran varias opciones. La principal variante se basa en utilizar la noción de supermaximalidad de repeticiones en reemplazo de maximalidad. Adicionalmente se estudia incorporar al algoritmo el uso de otras características de la repetición, como ser la cantidad de instancias y su longitud, entre otras. Los resultados obtenidos han sido comparados con los obtenidos en el trabajo de Turjanski et al. En algunos casos, los resultados obtenidos han sido superiores a los existentes, arrojando indicios de cuáles serían las características que permiten la mejora.

Adicionalmente, se realizó una búsqueda de subcadenas únicas minimales dentro de una familia, con el objetivo de identificar unívocamente una proteína dentro de una familia. Se implementó un algoritmo y se analizaron los resultados obtenidos en base a la familia de proteínas ankyrin. La intención es, a futuro, poder combinar dicha información con la de repeticiones y así poder obtener un algoritmo más preciso aún.

Palabras clave: proteínas, subcadenas únicas minimales, repeticiones súpermaximales, clasificación

Índice general

1. INTRODUCCIÓN Y DEFINICIONES

Las proteínas son grandes biomoléculas, las cuales están formadas por concatenaciones de aminoácidos. Un aminoácido es una molécula orgánica con un grupo amino (-NH2) y un grupo carboxilo (-COOH) [1]. Los elementos principales que forman un aminoácido son carbono, hidrógeno, oxígeno y nitrógeno, aunque otros elementos pueden ser encontrados en cadenas laterales de ciertos aminoácidos. Hay alrededor de 500 aminoácidos conocidos y pueden ser clasificados de muchas formas [2]. Sin embargo, en la mayoría de los organismos, sólo 20 aparecen en el código genético. Existe un grupo de aminoácidos, denominados *aminoácidos esenciales*, que son fundamentales para la nutrición humana [3]. Dos aminoácidos se combinan en una reacción de condensación entre el grupo amino de uno y el carboxilo del otro, liberándose una molécula de agua y formando un enlace amida que se denomina enlace peptídico; estos dos residuos de aminoácido forman un dipéptido. Si se une un tercer aminoácido se forma un tripéptido y así, sucesivamente, hasta formar un polipéptido. Estos polipéptidos se denominan *proteínas* cuando la cadena polipeptídica súpera una cierta longitud (entre 50 y 100 residuos aminoácidos) o la masa molecular total súpera las 5.000 *uma* o, especialmente, cuando tienen una estructura tridimensional estable definida.

Las proteínas tienen una gran cantidad de funciones dentro de los organismos, incluyendo catalizar reacciones metabólicas, replicar ADN, comunicación celular y transporte de moléculas. Difieren entre sí principalmente en su secuencia de aminoácidos la cual está especificada por la secuencia de nucleótidos del gen que la codifica y usualmente resulta en un plegamiento en una estructura tridimensional específica que determina su comportamiento. Algunas de estas proteínas naturales presentan patrones estructurales recurrentes. Estas moléculas están ampliamente clasificadas de acuerdo al largo de la mínima unidad de repetición [4]. Aquellas donde estos patrones de repetición son cortos, menores o iguales a 5 aminoácidos, se denominan *fibrilares*, donde están entre 5 y 60 *repetitivas*, y para mayores o iguales a 60 *globulares*. Las *repetitivas* a pesar de ser muy similares a nivel de estructura terciaria, pueden tener repeticiones que sean extremadamente variables a nivel de estructura primaria. Los niveles de estructura que nos van a resultar de interés en este trabajo son la estructura primaria y la estructura terciaria. La estructura primaria esta dada por la secuencia de aminoácidos que conforman la proteína, la estructura terciaria está representada por su estructura tridimensional [5].

Como proponen Turjanski et al. [6] utilizaremos una definición matemática de repeticiones (se describe más adelante) para buscar ocurrencias de estas secuencias en diferentes familias de proteínas. En su trabajo el grupo describe que cadenas largas de repeticiones exactas son infrecuentes en proteínas particulares, incluso para aquellas que se conoce que se pliegan en estructuras de motivos estructurales recurrentes. Por otro lado, también detallan que estas proteínas son repetitivas dentro de sus familias, exhibiendo cadenas de aminoácidos que son repeticiones exactas provenientes de la familia de referencia. Como producto de esto, proponen un algoritmo para cuantificar las chances de que una proteína particular pertenezca a una cierta familia.

Uno de los propósitos del trabajo aquí presentado es profundizar sobre la idea de este algoritmo, introduciéndole ciertas modificaciones. Utilizaremos una definición similar pero no exactamente igual a la hora de definir patrones de repetición. Esto resultará en un algo-

ritmo algo más sencillo de comprender e implementar, potencialmente, más performante. También evaluaremos la precisión de los resultados obtenidos a la hora de cuantificar si una proteína pertenece a una cierta familia, contrastándolos con los del algoritmo propuesto por Turjanski et al. [6]. Estudiaremos las diferencias entre los resultados obtenidos por ambos algoritmos, y así analizaremos si uno ofrece mejores resultados que el otro, y por qué se produce esta diferencia. Luego presentaremos distintas alternativas sobre este algoritmo que contrastaremos en busca de una mejor precisión en los resultados.

Por otro lado, así como nos interesará identificar cuán similar es una cierta proteína a una familia, vamos a introducir una definición matemática para distinguir unívocamente a esa proteína del resto de su familia. Se van a buscar subpatrones minimales tal que se presenten sólo en esa proteína y no tengan ninguna otra ocurrencia dentro de las proteínas de la familia.

Primero vamos a presentar ciertas definiciones básicas sobre cadenas de caracteres, repeticiones maximales y súper maximales, subcadenas únicas minimales y los algoritmos y estructuras utilizados para buscar patrones en cadenas de caracteres. Haremos una analogía entre cadenas de caracteres y las secuencias de aminoácidos que representan a las protreínas a estudiar. Luego presentaremos una forma de distinguir a una proteína del resto de su familia, aplicando lo visto. Más adelante detallaremos cómo funciona el algoritmo que cuantifica las chances de que una proteína pertenezca a cierta familia. A continuación mostraremos un caso de estudio con datos reales. Contrastaremos los resultados obtenidos con los del trabajo de Turjanski et al. [6] mencionado anteriormente. Y por último propondremos alternativas novedosas en el cálculo de $familiaridad$ de una proteína en búsqueda de una mayor precisión.

1.1. Definiciones básicas sobre cadenas

Sea w una cadena de caracteres de largo $|w| = n$. Para cada $1 \leq i \leq n$, $w[i]$ es el i-ésimo caracter de w. Una *subcadena* de w es una cadena de la forma $w[i..j] = w[i]w[i+1]...w[j]$, para algún $1 \leq i \leq j \leq n$. En particular $w = w[1..n]$. El $reverso$ de w es denotado $w_r = w[n]...w[1]$. El $sufijo$ de w que empieza en la posición i es denotado $suf[i] = w[i..n]$.

Por ejemplo para la cadena $w = abaababa$ tendríamos $w[1] = a$, $w[2] = b$, $w[3..6] = aaba$, $w_r = ababaaba$ $suf[5] = baba$, $suf[1] = abaababa$, $suf[n] = a$

El *suffix array* (arreglo de sufijos) SA contiene las posiciones $1, 2, ..., n$ ordenadas lexicográficamente en forma ascendente de los correspondientes sufijos $suf[i]$, $i = 1, 2, ..., n$. Es decir, $SA[i] = j$ significa que $suf[j]$ es el i-ésimo menor sufijo en orden lexicográfico. El LCP (Longest Common Prefix array, o arreglo de prefijos comunes más largos) contiene en la i-ésima posición el largo del prefijo común más largo entre $suf[SA[i]]$ y $suf[SA[i-1]]$. Notemos que $LCP[1]$ no está definido.

No lo escribiremos en ningún momento explícitamente, pero vamos a considerar de manera implícita que siempre al final de la cadena w que estemos considerando vamos a agregar un caracter que llamaremos \$. Este caracter tendrá la particularidad de: *i)* no formar parte del alfabeto sobre el que esté definida w, *ii)* ser lexicográficamente menor que todos los caracteres de dicho alfabeto. Esto nos va a traer distintas ventajas implementativas y evita evaluar casos particulares en distintos momentos. Entre otras cosas, si un sufijo $suf_a = abb$ estuviera enteramente contenido como prefijo en otro sufijo $suf_b = abbaa$

necesitaríamos definir explícitamente que el menor lexicográficamente es el de menor largo. Sin embargo al agregar \$ al final de w, los sufijos nos quedan $suf_a = abb\$$ y $suf_a = abbaa\$$. Luego suf_a ya no es prefijo de suf_b y es menor, pues \$ es menor que todos los caracteres del alfabeto por definición. Al agregar \$, ningún sufijo puede ser prefijo de ningún otro.

Continuando con el ejemplo anterior, todos los sufijos de w son:

$$\begin{array}{ll} suf[1] & abaababa \\ suf[2] & baababa \\ suf[3] & aababa \\ suf[4] & ababa \\ suf[5] & baba \\ suf[6] & aba \\ suf[7] & ba \\ suf[8] & a \end{array}$$

Si los ordenamos lexicográficamente obtenemos:

$$\begin{array}{ll} SA[1] = 8 & suf[8] = a \\ SA[2] = 3 & suf[3] = aababa \\ SA[3] = 6 & suf[6] = aba \\ SA[4] = 1 & suf[1] = abaababa \\ SA[5] = 4 & suf[4] = ababa \\ SA[6] = 7 & suf[7] = ba \\ SA[7] = 2 & suf[2] = baababa \\ SA[8] = 5 & suf[5] = baba \end{array}$$

Entonces $SA[1] = 8 = n$ pues a es el ultimo sufijo, $SA[2] = 3$, $SA[3] = 6$, etc. Si tomamos el sufijo correspondiente a la posición 5 del *suffix array*, *ababa*, podemos notar que tiene en común con el anterior, *abaababa*, el prefijo *aba*, pues a la siguiente letra el primero contiene una a mientras que el último una b. Este es el prefijo común más largo entre estos dos sufijos. Notemos que a y ab también son prefijos comunes pero *aba* es el más largo. Esto es lo que representa el LCP, que en su posición 5 nos indica que el sufijo en la posición 5 del SA, *ababa*, y el sufijo en la posición 4 del SA, *abaababa*, tienen como prefijo común más largo a una subcadena de largo 3, que es *aba*. Es decir $LCP[5] = 3$. Recordemos que, como no podemos comparar el sufijo de la primera posición del SA contra el anterior dado que no tiene, el primer valor de LCP no esta definido.

En la *Tabla 1.1* podemos ver el ejemplo completo y más detallado. La columna $SA[i]$ muestra el valor del *suffix array*, la columna $suf[SA[i]]$ el sufijo correspondiente, y la columna $LCP[i]$ muestra el valor del *longest common prefix array*. En la columna de sufijos podemos ver subrayados los caracteres prefijo que comparten con el sufijo anterior para mayor claridad.

i	SA[i]	suf[SA[i]]	LCP[i]
1	8	a	-
2	3	aababa	1
3	6	aba	1
4	1	abaababa	3
5	4	ababa	3
6	7	ba	0
7	2	baababa	2
8	5	baba	2

Tabla 1.1: SA y *LCP* arrays para la cadena $w =$ **abaababa**. La tercera columna, correspondiente al sufijo asociado a $SA[i]$, muestra los sufijos con las posiciones del *LCP* subrayadas.

1.2. Suffix Array y Longest Common Prefix array

Estas estructuras fueron introducidas por primera vez en 1989 por Manber et al. [7] como una mejora sobre los algoritmos que existían en ese momento para búsqueda *on-line* de cadenas en textos. Hasta ese momento la estructura más utilizada para este propósito eran los *suffix trees* que, en líneas generales, consisten en organizar todos los sufijos de un texto en una estructura de árbol. Esto da lugar a luego poder identificar rápidamente si una cadena dada podría encontrarse como parte del texto o no. Construir el suffix tree cuesta un tiempo lineal respecto del tamaño del texto que representa y, contando con la estructura, buscar una determinada palabra cuesta un tiempo lineal respecto del tamaño de la palabra. Los suffix trees fueron propuestos por Weiner [8] con importantes aportes subsiguientes de McCreight [9] y Ukkonen [10]. El primero introduciendo una mejora de alrededor del 25 % en el espacio requerido para almacenar la estructura. El segundo introduciendo un importante cambio que consistió en presentar un algoritmo para armar el suffix tree que tiene la propiedad de que el árbol se puede actualizar dinámicamente a medida que se agregan nuevos caracteres al texto que representa.

Tanto el suffix array como el suffix tree, se pueden construir con una complejidad temporal de O(n) siendo n el largo del texto. La mejora que introducen los suffix arrays se encuentra en la complejidad espacial que requiere la estructura. Si bién para ambas estructuras la complejidad espacial requerida es lineal respecto al largo del texto, los suffix arrays ocupan de 3 a 5 veces menos espacio en la práctica. Esta mejora en una constante en principio puede no parecer muy significativa, pero en la práctica, para muchas aplicaciones que manejan grandes volumenes de datos, puede hacer la diferencia entre que sea factible usar la estructura o no. El LCP array es una estructura auxiliar del suffix array y se puede construir al mismo tiempo que se construye éste, sin modificar la complejidad temporal requerida.

Cuando fueron propuestos por primera vez los suffix arrays requerían un tiempo $O(n + log(n))$ en su algoritmo de construcción directa. Si bien se podía armar el suffix array a partir de los sufijos obtenidos en el suffix tree, requiriendo así un tiempo lineal, se perdía la complejidad espacial $O(n)$. Sin embargo eventualmente fueron presentados algoritmos de construcción directa de suffix arrays que requieren una complejidad tanto espacial como temporal lineales como son los de Ko et al. [11], Kärkkäinen et al. [12] y Nong et al. [13]. En Puglisi et al. [14] podemos encontrar una recopilación de distintos algoritmos de construcción de suffix array y un análisis más detallado de sus propiedades.

1.3. Repeticiones maximales y súper maximales

A partir de lo propuesto por Gusfield [15] definimos:

- MR [repetición maximal] (a partir de *maximal repeat*, por sus siglas en inglés):
*"Dada una secuencia s, un **MR** es una subsecuencia que ocurre más de una vez en s, y cada una de sus extensiones ocurre menos veces."*

- SMR [repetición súper maximal] (a partir de *súper maximal repeat*, por sus siglas en inglés):
*"Dada una secuencia s, un **SMR** es una subsecuencia que ocurre más de una vez en s, y cada una de sus extensiones ocurren una única vez."*

Para ilustrar esto con un ejemplo tomemos la cadena *catarata*. Esta tiene como repeticiones maximales a *ata* que tiene dos ocurrencias en las posiciones 2 y 6, y a *a* que tiene 4 ocurrencias; más que cualquiera de sus extensiones propias. La única repetición súpermaximal de *catarata* es *ata*, ya que sus otras repeticiones (en este caso sólo *a*) tienen extensiones propias que ocurren más de una vez. Es decir, si intentamos extender la repetición *a*, en cualquiera de sus ocurrencias, podemos hacerlo sin que deje de ser repetición. Observemos que toda repetición súpermaximal de una cadena también es repetición maximal.

En este trabajo utilizaremos **SMR**. Mas adelante, contrastaremos lo aquí desarrollado con lo propuesto por Turjanski et al. [6] donde se utilizan **MR**.

1.4. Repeticiones y subcadenas únicas

Una *repetición* de w está representada por un intervalo $[i..j]$ tal que la subcadena $w[i..j]$ ocurre al menos dos veces en w. Un subintervalo de una repetición, también representa una repetición y por lo tanto nos va a interesar considerar solo las *repeticiones súper maximales*. Una *repetición súper maximal* (**SMR**) está representada por un intervalo $[i..j]$ tal que la cadena $w[i..j]$ ocurre al menos dos veces en w pero las cadenas $w[i-1..j]$ y $w[i..j+1]$, en caso de estar definidas, ocurren una sola vez. Las **SMR** incluyen a todas las demás repeticiones como subintervalos. Por ejemplo para la cadena *abaababa* la subcadena *ba* es una repetición, pues se encuentra en las posiciones 2, 5 y 7. Sin embargo *ba* no es súper maximal, pues podemos extenderla considerando la subcadena *aba*, que se repite en las posiciones 1, 4 y 6. Observemos que dada la definición, las **SMR** pueden súperponerse como lo hacen la segunda y tercera ocurrencias de *aba*, que comparten una *a*. La subcadena *aba* sí es súper maximal. Fijémosnos que para cualquiera de las 3 ocurrencias, si la extendemos en un caracter, a derecha o a izquierda (siempre que sea posible, dado que la primera, por ejemplo, no se puede extender a izquierda), tomando así una subcadena de 4 caracteres, no vamos a encontrar otra igual en la cadena original w. Es decir, si las extendiéramos dejarían de ser repeticiones.

Una *subcadena única* de w está representada por un intervalo $[i..j]$ tal que la subcadena $w[i..j]$ ocurre exactamente una vez en w. Nos van a interesar solo las subcadenas únicas que sean minimales, ya que todas las subcadenas de w que las contengan también van a ser a su vez únicas. Notemos que por ejemplo w entera es subcadena de sí misma y es única pero lo que nos interasa realmente es saber si hay subcadenas únicas más chicas.

Una *subcadena única minimal* de w está representada por un intervalo $[i..j]$ tal que o $i = j$ y el caracter $w[i]$ ocurre solo una vez en w o $i < j$ y cada uno de los intervalos $[i + 1..j]$ y $[i..j - 1]$, son repeticiones de w. En el ejemplo, *abaababa*, podemos ver que *aa* y *bab* son subcadenas minimales únicas, ambas ocurren una sola vez en w. Si las redujéramos en un caracter, tanto considerando *a* en lugar de *aa* o considerando ya sea *ba* o *ab* en lugar de *bab* obtendríamos subcadenas que son repeticiones. Por eso podemos asegurar que *aa* y *bab* son minimales. Si consideramos las subcadenas *baa* o *baba* notaremos que también son únicas, pero no son minimales, por lo tanto no van a ser de interés.

Cuando definamos los algortimos para buscar **SMR** y subcadenas únicas minimales vamos a definir la salida como los intervalos que representen estas subcadenas. Para las subcadenas únicas, ya sean minimales o no, se dá que tenemos una biyección entre la subcadena y el intervalo que la representa. Un intervalo representa siempre unívocamente una subcadena (esto es siempre así más allá de si la subcadena es única o no). Y una subcadena única a su vez determina unívocamente un intervalo ya que existe una sola ocurrencia de la misma. Sin embargo, como dijimos, vamos a dar la respuesta en términos de intervalos. La diferencia es importante desde un punto de vista algorítmico, ya que toma tiempo lineal encontrar un intervalo a partir de la correspondiente subcadena y tiempo constante proveer la cadena a partir del intervalo. Como ejemplo para *abaababa*, al buscar las subcadenas únicas minimales, la respuesta tendrá la forma: [3..4], [5..7].

En el caso de las **SMR** no existe la biyección de la que hablamos antes. Se va a devolver también un intervalo por cada ocurrencia de cada repetición. Sin embargo dejaremos esto sujeto a ciertas restricciones que vamos a definir en breve. Así, por ejemplo para *abaababa*, cuando busquemos las **SMR**, la respuesta tendría la forma [1..3], [4..6], [6..8].

Como ya mencionamos antes, nos interesan solo las repeticiones que son súper maximales, por lo tanto vamos a reportar solo los *intervalos* que representen subcadenas *súper maximales*. Estos intervalos son las ocurrencias de las repeticiones no extendibles tales que no sean una subcadena propia contenida en un intervalo que representa alguna otra ocurrencia de alguna repeticion no extendible de w. En el ejemplo que dimos anteriormente las 3 repeticiones encontradas son intervalos que representan cadenas *súper maximales*. Sin embargo si tomamos el caso $w = abaababaabaab$ lo que el algortimo va a reportar son los intervalos [1..6], [6..11], y [9..13] correspondientes a las repeticiones *abaaba*, *abaaba* y *abaab*. Todas son repeticiones *súper maximales*, pues ninguna puede ser extendida, pero mientras las 2 ocurrencias de *abaaba* están siendo reportadas, solo nos interesa 1 de las 3 ocurrencias de *abaab*. Esto es porque tanto [1..5], como [6..10], son subintervalos propios de [1..6], [6..11], es decir, no representan subcadenas súper maximales. Remarquemos que de esta manera vamos a estar reportando *una* sola ocurrencia de la repetición *abaab*. No por esto deja de ser una *repetición súper maximal*.

1.5. Algoritmo para buscar repeticiones súper maximales

A continuación se describe el algoritmo para encontrar *repeticiones súper maximales* (**SMR**) en tiempo lineal respecto del largo de la cadena, propuesto por Ilie et al. [16]. Es importante destacar que si bien en el trabajo de Ilie et al. las llaman *repeticiones maximales*, dada la definición, estas se corresponden con las que aquí denominamos *repeticiones súper maximales*. La idea principal en la que se basa dicho algoritmo se encuentra en la siguiente propiedad:

Cualquier repetición súper maximal $[i..j]$ *es el más largo entre todos los prefijos repe-*

tidos que terminan en j de todos los sufijos de w.

Veamos más en detalle como se interpreta esta propiedad. Dada una repetición súper maximal $[i..j]$, sobre una cadena w de largo n, consideremos todos los sufijos de w: $suf[1]$, $suf[2]$, ... , $suf[n]$. Ahora consideremos todos los sufijos que tienen prefijos repetidos: los $suf[k]$ tales que existe al menos un $suf[k']$ (con $k \neq k'$) y $suf[k][1..x] = suf[k'][1..x]$. A su vez observemos que puede haber otros pares de sufijos con prefijos repetidos tales que esos prefijos terminan en la misma posición en w. Por ejemplo tomando $suf[k_2][1..x+d] = suf[k'_2][1..x+d]$ de manera que $k+x = k_2+x+d$ (no necesariamente $k'+x = k'_2+x+d$). Si tomamos todos los prefijos de sufijos que tienen alguna repetición, tales que todos terminan en la misma posición en w, el más largo de ellos será una repetición súper maximal. Por ejemplo si $suf[k][1..x]$ y $suf[k_2][1..x+d]$ son los únicos prefijos de sufijos que terminan en la posición $k+x$ en w y $k_2 < k$ entonces $i = k_2$ y $j = k+x = k_2+x+d$ y $w[i..j]$ es repetición súper maximal. En la *Figura 1.1* podemos ver una representación gráfica de este ejemplo.

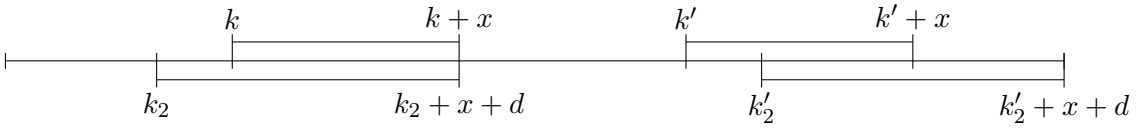


Figura 1.1: Ejemplo de repetición súper maximal a partir de los sufijos de una cadena.

A continuación explicaremos cómo implementar la idea expuesta por la propiedad mencionada anteriormente utilizando las estructuras previamente definidas: SA y LCP. El array SA contiene todos los sufijos de una cadena w ordenados lexicográficamente. Dos sufijos que se encuentren contiguos en SA son propensos a compartir un prefijo común, y el largo de este prefijo nos lo va a indicar la estructura LCP. Al encontrar un valor distinto de 0 en LCP esto quiere decir que encontramos un sufijo que tiene un prefijo (de largo mayor o igual a 1) en común con el sufijo correspondiente a la posición anterior en SA. Este prefijo común es una repetición. No estamos buscando cualquier repetición si no las repeticiones súper maximales, pero vamos a llegar a eso más adelante. Es importante en este momento observar que *toda* repetición posible que existe en w la vamos a poder encontrar reflejada como un prefijo común de dos sufijos que se encuentren en posiciones consecutivas del SA. Por eso, la idea del algoritmo va a ser evaluar exclusivamente estos casos.

Detengámosnos en ciertos casos que, si bién son sencillos, al tratar de entender el algoritmo por primera vez podrían prestar a confusión. Veamos por qué nos alcanza con evaluar los prefijos de sufijos consecutivos en SA y no hace falta comparar otras subcadenas. Para eso contestemos las siguientes preguntas:

¿Es posible que una repetición se encuentre como subcadena de un sufijo pero que no sea prefijo de ese sufijo?

La respuesta es sí, pero en ese caso va a haber otro sufijo que va a tener como prefijo a esa subcadena. Recordemos que al considerar todos los sufijos de w estamos considerando todas las posiciones posibles. *Cualquier* subcadena de w se puede representar como el prefijo de alguno de sus sufijos. Si tomamos el ejemplo de $w = abaababa$, ambos sufijos $suf[1] = abaababa$ y $suf[3] = aababa$ contienen la repetición aba, pero esta no es prefijo de

$suf[3]$. Sin embargo existe $suf[4] = ababa$ que contiene a la subcadena aba como prefijo.

¿Podría darse que una repetición se encuentre como prefijo común de dos sufijos **no** consecutivos en SA?

Sí, pero para cualquier repetición dada como prefijo común entre dos sufijos no consecutivos de SA va a haber otra repetición de largo mayor o igual. Recordemos que SA representa los sufijos de w, *ordenados lexicográficamente*. Es por eso que el prefijo común más largo que un sufijo cualquiera pueda compartir con otro lo va a hacer con uno tal que esté en la posición anterior o siguiente de SA. Cualquier otro sufijo no va a tener un prefijo común o va a tener uno de un largo menor o igual. De lo contrario, no se estaría respetando el orden. Veamos un ejemplo para entender mejor:

$$\begin{array}{ll} SA[1] = a_1 & suf[a_1] = \alpha_1 \\ SA[2] = a_2 & suf[a_2] = \alpha_2 \\ ... & ... \\ SA[p] = a_p & suf[a_p] = \alpha_p \\ SA[p+1] = a_{p+1} & suf[a_{p+1}] = \alpha_{p+1} \\ ... & ... \\ SA[q] = a_q & suf[a_p] = \alpha_q \\ ... & ... \\ SA[n] = a_n & suf[a_n] = \alpha_n \end{array}$$

Supongamos que α_p comparte un prefijo común con α_{p+1}, que llamaremos $\gamma_{p::p+1}$, de manera que $\alpha_p[1..p] = \alpha_{p+1}[1..p] = \gamma_{p::p+1}$. Además, α_p tiene otro prefijo común con α_q, que llamaremos $\gamma_{p::q}$ de manera que $\alpha_p[1..p] = \alpha_q[1..p] = \gamma_{p::q}$. Ahora asumamos también que $\gamma_{p::p+1}$ tiene un largo menor que $\gamma_{p::q}$. Es decir α_p comparte un prefijo más largo con α_q que con α_{p+1}. Pero de ser esto así, α_{p+1} debería ser lexicográficamente menor que α_p o mayor que α_q. Sin embargo esto no puede ser porque se rompería el orden lexicográfico de los sufijos en SA, con lo cual llegamos a una contradicción.

Pasemos a describir el funcionamiento del algoritmo de búsqueda de repeticiones súper maximales propuesto por Ilie et al. [16]. En *Algoritmo1.1* podemos ver su pseudocódigo. Se evalúa en orden creciente cada sufijo en SA desde la primer posición hasta la última. Para cada uno de ellos se evalúa el valor de LCP en el índice actual, obteniendo así el largo del prefijo común entre el sufijo actual y el anterior; y el valor de LCP en el índice igual al actual más uno, obteniendo así el largo del prefijo común entre el sufijo actual y el siguiente. Destaquemos que como esto resultaría en algunas posiciones indefinidas en LCP vamos a definir $LCP[1] = LCP[n+1] = 0$ para evitar contemplar casos especiales y simplificar la implementación. Así, si nos encontramos en la iteración i vamos a estar comparando la longitud de prefijos comunes entre $suf[i-1]$ y $suf[i]$; y entre $suf[i]$ y $suf[i+1]$. Por lo expuesto anteriormente, sabemos que el prefijo común más largo entre estos dos, es el prefijo común más largo entre este sufijo ($suf[i]$) y cualquier otro de w. Este prefijo es una repetición, pero podría no ser súper maximal. Al ser el más largo de todos los prefijos comunes que tiene este sufijo lo que podemos asegurar es que si lo extendiéramos en un caracter a derecha la subcadena que representa este prefijo dejaría de ser una repetición y pasaría a ser una subcadena única. Lo que sí se podría dar es que exista otro prefijo común de otro sufijo que termine en la misma posición que este, pero empiece más a la izquierda. Lo único que sabemos es que la repetición que encontramos no se puede extender a derecha porque deja de ser repetición. Sin embargo, si pudiéramos tomar todas

las repeticiones posibles que terminan en esta posición, la que empiece primero va a ser la que es súper maximal pues además de no poder extenderse a derecha, tampoco va a poder extenderse a izquierda porque es la que empieza primero, es decir,la más larga. Esta es la propiedad fundamental en la que se basa el algoritmo.

Se cuenta con un array que denominaremos $MaxRep$ donde se almacenan las repeticiones que se encuentran indexadas por la posición en que terminan. Así se representa la repetición $[i..j]$ guardando en $MaxRep[j] = i$. A cada iteración se guarda el prefijo común más largo $[i..j]$ del sufijo actual en $MaxRep$ asignando $MaxRep[j] = i$, si en una iteración posterior resulta que se encuentra que el prefijo común más largo de otro sufijo termina también en j pero su posición inicial i' es menor que i, es decir, es una repetición que contiene a la anterior y es más larga, se actualiza $MaxRep$ asignando ahora $MaxRep[j] = i'$. En caso de que $i < i'$ se deja $MaxRep[j] = i$ como estaba originalmente.

```
1  MaxRepeat(w) {
2          computar SA, LCP
3          n <- length(w)
4          for idx from 1 to n do
5              MaxRep[idx] <- n+1
6          for idx from 1 to n do
7              lcp <- max(LCP[idx], LCP[idx+1])
8              j <- SA[idx] + lcp - 1
9              i <- min(MaxRep[j], SA[idx])
10             MaxRep[j] <- i
11         return MaxRep
12 }
```

Algoritmo 1.1: Algoritmo para buscar repeticiones súper maximales

Como se mencionó antes, la complejidad temporal de construir las estructuras SA y LCP es lineal respecto del tamaño de w, luego es fácil ver que el algoritmo propuesto toma también tiempo lineal. Respecto a la complejidad espacial del algoritmo, también es lineal, pues SA y LCP tienen el mismo tamaño que w y lo único que agregamos es la estructura auxiliar MaxRep que tambien tiene tamaño $n = length(w)$. Todas las complejidades son indepentientes del tamaño del alfabeto.

A continuación se presenta un ejemplo sencillo de la ejecución del algoritmo para ilustrar cómo funciona. Vamos a buscar las repeticiones súper maximales de la cadena $w = abaababa$. Usaremos como referencia la *Tabla 1.2* que nos muestra a cada iteración el valor que toma la estructura MaxRep.

Recordemos que inicializamos la estructura $MaxRep$ con todas sus posiciones en 9, es decir, el largo de w más 1. En la primera iteración $idx = 1$, detectamos que el primer sufijo analizado a, tiene un prefijo común con el siguiente, de largo 1, el caracter a, luego anotamos en la posición correspondiente a $SA[1] = 8$ de $MaxRep$ el inicio de esa repetición, que como tiene un largo de 1 es el mismo que el final, así asignamos $MaxRep[8] = 8$. Cuando llegamos a la iteración $idx = 3$, el sufijo considerado, aba, tiene un prefijo común con el siguiente de largo 3, que es a su vez aba. Lo interesante es notar que esta repetición también termina en la posición 8 de w, pero empieza en la 6, por lo tanto cuando vayamos a actualizar $MaxRep$ vamos a cambiar el valor que había previamente en la posición 8 por un 6, $MaxRep[8] = 6$. Y así podemos ver como el algoritmo va a ir actualizando

idx	SA[idx]	suf[SA[idx]]	LCP[idx]	MaxRep (al finalizar la idx-ésima iteración)
1	8	a	0	[9, 9, 9, 9, 9, 9, 9, 8]
2	3	aababa	1	[9, 9, 3, 9, 9, 9, 9, 8]
3	6	aba	1	[9, 9, 3, 9, 9, 9, 9, 6]
4	1	abaababa	3	[9, 9, 1, 9, 9, 9, 9, 6]
5	4	ababa	3	[9, 9, 1, 9, 9, 4, 9, 6]
6	7	ba	0	[9, 9, 1, 9, 9, 4, 9, 6]
7	2	baababa	2	[9, 9, 1, 9, 9, 4, 9, 6]
8	5	baba	2	[9, 9, 1, 9, 9, 4, 9, 6]

Tabla 1.2: Ejemplo de ejecución del algoritmo con los valores que va tomando el array MaxRep

las repeticiones que va detectando con otras más largas en caso de que las encuentre. Al finalizar, toda posición de $MaxRep$ de la forma $MaxRep[j] = i$ con $i \neq n + 1$ va representar una repetición súper maximal $[i..j]$.

1.6. Algoritmo para buscar subcadenas únicas minimales

El algoritmo que vamos a describir para buscar las subcadenas únicas minimales, también presentado en Ilie et al. [16], es muy similar al que describimos para buscar repeticiones súper maximales. En *Algoritmo1.2* podemos ver su pseudocódigo. Se basa en la siguiente propiedad: cualquier subcadena única minimal correspondiente a un intervalo $[i..j]$ es la más corta entre todos los prefijos únicos que terminan en j de todos los sufijos de w. Las subcadenas candidatas a ser únicas minimales van a ir siendo guardadas, de acuerdo a los intervalos $[i..j]$ que las representan, en un array $MinUnique$ asignando $MinUnique[j] = i$. Esto es de alguna manera análogo a lo que hacíamos con el array $MaxRep$. Como lo que vamos a buscar son las subcadenas más cortas, esta vez inicializamos $MinUnique$ con valores más chicos que cualquier valor válido posible. De vuelta vamos a computar el prefijo común más largo de cada sufijo. La diferencia está en que ahora, para obtener subcadenas únicas en lugar de repeticiones, vamos a considerar un caracter más que el prefijo común más largo. Deberíamos considerar un caso especial cuando el prefijo común más largo termina en el último caracter de w. Sin embargo, para evitar verificar esta condición a cada iteración, vamos a agregar una posición más, $n + 1$, a $MinUnique$ que vamos a terminar ignorando. En contraposición a lo que hacíamos antes, vamos a actualizar $MinUnique$ si encontramos una subcadena única más corta, en lugar de una más larga. Al finalizar, toda posición de $MinUnique$ de la fomra $MinUnique[j] = i$ con $i \neq 0$ va representar una subcadena única minimal $[i..j]$. Como se puede apreciar, este algoritmo también corre en tiempo lineal.

```
1  MinUnique(w) {
2          computar SA, LCP
3          n <- length(w)
4          for idx from 1 to n+1 do
5              MinUnique[idx] <- 0
6          for idx from 1 to n do
7              lcp <- max(LCP[idx], LCP[idx+1])
8              j <- SA[idx] + lcp
9              i <- max(MinUnique[j], SA[idx])
10             MinUnique[j] <- i
11         return MinUnique
12 }
```

Algoritmo 1.2: Algoritmo para buscar subcadenas únicas minimales

1.7. Extensiones

1.7.1. Repeticiones súper maximales en un conjunto de cadenas

Vamos a extender la definición de *repeticiones súper maximales* a conjuntos de cadenas. Sea $S = w_1, w_2, ..., w_m$ un conjunto de cadenas, una *repetición súper maximal* de S representa una cadena $w_k[i..j]$, para un $1 \leq k \leq m$, que ocurre al menos dos veces entre los $w_q \in S$, mientras que las cadenas $w_k[i-1..j]$ y $w_k[i..j+1]$, en caso de estar definidas, ocurren una sola vez entre todos los w_q. Es decir $w_k[i..j]$ ocurre, por definición, en w_k y además ocurre al menos una vez más en w_k o en algún otro $w_q \in S$.

Ilustremos esto con un ejemplo. Si tenemos el conjunto de cadenas: S={*panama, bana, pan, ena, xyzpanxyz*}. Tendríamos las siguientes repeticiones súper maximales:

$$w_1 = \text{p}\underline{\text{an}}\underline{\text{a}}\text{ma}$$
$$w_2 = \text{b}\underline{\text{ana}}$$
$$w_3 = \underline{\text{pan}}$$
$$w_4 = \text{e}\underline{\text{na}}$$
$$w_5 = \dot{\text{x}}\dot{\text{y}}\dot{\text{z}}\underline{\text{pan}}\dot{\text{x}}\dot{\text{y}}\dot{\text{z}}$$

O sea: $w_1[1..3]$, $w_1[2..4]$, $w_2[2..4]$, $w_3[1..3]$, $w_4[2..3]$, $w_5[1..3]$, $w_5[4..6]$, $w_5[7..9]$. Estos intervalos se corresponden con las subcadenas $\underline{pan}$ subrayadas con guiones, las subcadenas $\underline{ana}$ y $\underline{an}$ subrayadas con líneas completas y las subcadenas $\dot{x}\dot{y}\dot{z}$ subrayadas con líneas de puntos.
Notemos que no importa si las repeticiones se encuentran todas en la misma cadena como xyz en $w_5[1..3]$ y $w_5[7..9]$, o en distintas como pan en $w_1[1..3]$, $w_3[1..3]$ y $w_5[4..6]$. Sin embargo sí nos va a seguir interesando que cada ocurrencia reportada de las repeticiones sea *súper maximal*. Por eso, si bien consideramos na en $w_4[2..3]$, no consideramos na en $w_1[3..4]$ ni $w_2[3..4]$

1.7.2. Extensión del algoritmo

Para poder reutilizar el algoritmo definido anteriormente para una única cadena, ahora en un conjunto de cadenas, vamos a realizar ciertas modificaciones tanto sobre el algoritmo

como sobre la entrada.

Nuestra entrada ahora es un conjunto de cadenas, sin embargo el algoritmo originalmente tomaba una sola, de modo que lo que vamos a hacer es tomar todas nuestras cadenas y transformarlas en una única cadena. Para esto las vamos a concatenar separándolas con un caracter delimitador, que no pertenece al alfabeto utilizado, que denotaremos como #. Consideramos # lexicográficamente menor que todos los caracteres del alfabeto. Continuando con el ejemplo anterior tendríamos w_s=*panama#bana#pan#ena#xyzpanxyz#*. El orden en el que concatenemos las cadenas no es relevante, explicaremos por qué más adelante.

Recordemos que estábamos considerando implícitamente un caracter $ (que no pertenece al alfabeto utilizado) al final de cada cadena. Necesitaremos redefinir esto con pequeños cambios al utilizar la concatenación de cadenas de un conjunto y el caracter delimitador #. Definimos $ como menor lexicográficamente que #. Dado el conjunto de cadenas, cada una de ellas individualmente no será considerada con $ implícitamente como último caracter. El caracter # actuará como delimitador y lo denotaremos siempre explícitamente. El caracter $ se considerará implícitamente como último caracter de la cadena concatenación. Siguiendo el ejemplo anterior obtenemos: w_s=*panama#bana#pan#ena#xyzpanxyz#$*. Estas definiciones no son arbitrarias, la función de $ es la explicada anteriormente: que ningún sufijo sea prefijo de ningún otro. Por este motivo es que no podemos usar $ mismo como caracter delimitador. Sin embargo, utilizar un caracter delimitador es necesario y este no debe formar parte del alfabeto, por eso designamos un nuevo caracter # con este próposito. Es muy útil para entender los ejemplos ver claramente donde empieza una palabra del conjunto original y termina otra, dentro de la cadena concatenación, por eso # siempre lo denotamos explícitamente.

Ahora solo nos falta un pequeño cambio en el algoritmo para que, al aplicarlo sobre esta nueva cadena, obtengamos el resultado deseado. Este cambio tiene por objetivo que el algoritmo no considere el caracter delimitador # cuando evalúa las repeticiones súper maximales. De lo contrario obtendríamos por ejemplo *na#* como repetición súper maximal. Tengamos en cuenta que no solo podrían hallarse repeticiones súper maximales con # al final de la misma, sino en cualquier posición. Por ejemplo, si tuvíeramos S={*abc, ddrt, ffjbc, ddmgh*} al concatenarlas obtendríamos *abc#ddrt#ffjbc#ddmgh#*, de lo cual resultaría que *bc#dd* es una repetición. Esto no es lo que buscamos. Recordemos que dada la definición, una repetición súper maximal no puede estar parte en una cadena del conjunto y parte en otra. Lograr adaptar el algoritmo para evitar esta situción no es complicado. De hecho no hay que modificar el algoritmo en sí mismo sino como armamos las estructuras de las que se vale para funcionar, en particular el LCP array. Veamos por qué esto es suficiente. En la línea 7 el *Algoritmo1.1* usa el LCP array para obtener el largo de las subcadenas que va a considerar como candidatas a repeticiones súper maximales. Este largo funciona como *desplazamiento* sobre una posición en la cadena de entrada dada por $SA[idx]$ (*Algoritmo1.1* línea 8). Si logramos asegurarnos que esta subcadena que está siendo considerada como candidato a repetición súper maximal no contenga # entonces evitaremos toda posibilidad de que # aparezca en las repeticiones súper maximales de la respuesta. Para esto vamos a restringir el largo del prefijo común más largo que se guarda en una posición del LCP array considerando conceptualmente que el caracter # siempre es distinto de sí mismo. Veamos esto con un ejemplo a partir de la cadena *bc#dr#fbc#dm#*. Si calculamos el LCP array sin ninguna modificación se obtiene:

i	SA[i]	suf[SA[i]]	LCP[i]
1	13	#	-
2	10	#dm#	1
3	3	#dr#fbc#dm#	2
4	6	#fbc#dm#	1
5	8	bc#dm#	0
6	1	bc#dr#fbc#dm#	4
7	9	c#dm#	0
8	2	c#dr#fbc#dm#	3
9	11	dm#	0
10	4	dr#fbc#dm#	1
11	7	fbc#dm#	0
12	12	m#	0
13	5	r#fbc#dm#	0

En cambio si aplicamos la restricción mencionada se obtiene:

i	SA[i]	suf[SA[i]]	LCP[i]
1	13	#	-
2	10	#dm#	0
3	3	#dr#fbc#dm#	0
4	6	#fbc#dm#	0
5	8	bc#dm#	0
6	1	bc#dr#fbc#dm#	2
7	9	c#dm#	0
8	2	c#dr#fbc#dm#	1
9	11	dm#	0
10	4	dr#fbc#dm#	1
11	7	fbc#dm#	0
12	12	m#	0
13	5	r#fbc#dm#	0

que es exactamente lo que necesitamos para que al aplicar el algoritmo nos dé el resultado que buscamos. En cuanto a la implementación, se hizo una pequeña modificación en la condición que evalúa el algoritmo utilizado para calcular el LCP (una implementación basada en Kasai et al. [17]), chequeando además de la condición de si los caracteres son o no iguales, que a su vez sean ambos distintos de #.

Observemos que, a partir de estos cambios, lo que logramos es considerar los prefijos comunes más largos que están contenidos enteramente dentro de una única cadena del conjunto de cadenas original. Es precisamente por el hecho de que no consideramos nunca los prefijos comunes más largos que se encuentren parcialmente en una cadena del conjunto y parcialmente en otra (u otras) que el orden en el cual concatenemos las cadenas del conjunto no es relevante. Dicho de otra manera, al considerar cada sufijo sólo hasta el primer # encontrado, el efecto conseguido es el mismo que si tomaramos cada uno de los sufijos de cada una de las cadenas del conjunto de cadenas original, sin concatenarlas en una sola, con eso armaramos un gran conjunto general de sufijos y a estos los ordenaramos

para formar el SA.

Es necesario hacer algunas observaciones sobre el formato de la respuesta. Cuando concatenamos todas las cadenas del conjunto en una única cadena y ejecutamos el algoritmo sobre ella, las repeticiones súper maximales de la respuesta las vamos a obtener en forma de intervalos sobre la concatenación. Si necesitáramos saber a qué cadena del conjunto de entrada corresponde cada repetición súper maximal, deberíamos de alguna manera armar un mapeo entre los índices de la concatenación y las cadenas individuales. Más adelante, cuando una cierta aplicación lo requiera, profundizaremos sobre cómo implementar este mapeo.

1.7.3. Subcadenas únicas minimales en un conjunto de cadenas

De manera análoga, extendemos la definición de *subcadenas únicas minimales* a un conjunto de cadenas. Dado $S = w_1, w_2, ..., w_m$ un conjunto de cadenas, una *subcadena única minimal* de S está dada por una cadena $w_k[i..j]$, para un $1 \leq k \leq m$ que ocurre una única vez entre los $w_q \in S$ mientras que las cadenas $w_k[i+1..j]$ y $w_k[i..j-1]$ tienen al menos una ocurrencia más en w_k o bien en algún otro $w_q \in S$.

Utilizaremos las mismas modificaciones que se explicaron para extender el algoritmo de búsqueda de *repeticiones súper maximales* para extender el algoritmo de búsqueda de *subcadenas únicas minimales*. Recordemos que la primera consistía en adaptar la entrada, transformando el conjunto de cadenas en una sola cadena formada por su concatenación separada por un caracter delimitador. Esto lo hacemos porque contamos con un conjunto de cadenas y el algoritmo toma una sola cadena como entrada. La segunda modificación consistía en cambiar la forma en la que se calculan los valores del LCP array. Esto va a funcionar de la misma manera. Después de todo, lo que estamos haciendo en este algoritmo es también buscar repeticiones (en forma de prefijos comunes en el LCP array), sólo que las extendemos en un caracter para que se vuelvan *subcadenas únicas minimales*. Sin embargo, hace falta una alteración adicional para que esto funcione. Como se explicó anteriormente, en el *Algoritmo1.2*, se agregó una posición adicional al array *MinUnique* para considerar el caso especial que se da cuando un prefijo común más largo termina en el último caracter de la cadena. Ahora esto ya no es suficiente. De la misma manera que nos ocupamos de que el caracter delimitador # no forme parte de los prefijos comunes más largos calculados en LCP, debemos encargarnos de que no sea tenido en cuenta en candidatos a *subcadena única minimal*. Estos candidatos se consideran a partir prefijos comunes más largos, extendidos en un caracter a la derecha. Como ya mencionamos, los prefijos comunes más largos no pueden contener el caracter delimitador #, la posibilidad de que sea introducido está en la extensión a derecha. Para esto, basta agregar en el *Algoritmo1.2* una línea inmediatamente después de la 8 donde se verifique si $w[j]$ es igual a #. De ser así, ignoramos las líneas 9 y 10 y proseguimos con la siguiente iteración. De esta manera evitamos por completo considerar candidatos a *subcadena única minimal* que contengan el caracter delimitador.

2. APLICACIONES SOBRE PROTEÍNAS

En este capítulo vamos a aplicar los algoritmos explicados anteriormente a un problema de sumo interés: la clasificación e identificación de proteínas. Para esto, modelaremos a una proteína como una secuencia de caracteres, donde cada caracter representa cada uno de los aminoácidos que conforman la proteína en el orden correspondiente. De esta manera podremos aplicar los algoritmos sobre secuencias definidos anteriormente a las proteínas, que son nuestro objeto de estudio.

Las proteínas que usaremos para nuestros experimentos se agrupan en familias. Nuestra hipótesis de trabajo es que las proteínas pertenecientes a una misma familia, comparten ciertas cualidades y similitudes. En este trabajo nos enfocaremos en particular en las similitudes que se encuentran en su nivel de estructura primaria. El nivel de estructura primaria de una proteína es precisamente la secuencia de aminoácidos que la conforman. Otras técnicas de clasificación de proteínas se basan en parte en su nivel de estructura secundaria [18] o en técnicas de auto-alineamiento de la estructura primaria [19]. La estructura secundaria de las proteínas ocurre cuando los aminoácidos de la secuencia interactúan a través de puentes de hidrógeno. La estructura terciaria de las proteínas ocurre cuando ciertas atracciones están presentes entre hélices alfa y hojas beta. La estructura cuaternaria de las proteínas consiste en una proteína que posee más de una cadena de aminoácidos. En la *Figura 2.1* se pueden apreciar los distintos niveles estructurales.

En nuestro trabajo vamos a realizar dos estudios distintos. Para el primero usaremos *subcadenas únicas minimales* y buscaremos una forma de distinguir unívocamente a una proteína del resto de su familia. En el caso del segundo utilizaremos *repeticiones súper maximales* y, tomando como base el trabajo realizado en Turjanski et al. [6], buscaremos cuantificar las chances de que una proteína, a priori desconocida, pertenezca o no a una cierta familia.

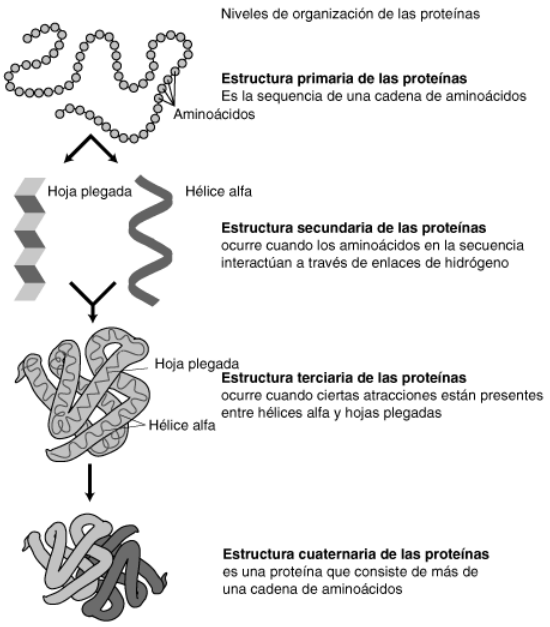


Figura 2.1: Niveles estructurales de las proteínas, fuente: [20]

2.1. Minimal tags

Vamos a llamar *minimal tag* a una subsecuencia minimal de aminoácidos que nos permita distinguir unívocamente una proteína dentro de un conjunto de proteínas. En particular nos va a interesar obtener los minimal tag de cada una de las proteínas que conforman una familia. Definimos formalmente *minimal tag* de una proteína p dentro de un conjunto de proteínas $\mathcal{F}$ con $p \in \mathcal{F}$, como una subsecuencia de aminoácidos de p tal que no ocurre otra subsecuencia igual dentro de ninguna proteína de $\mathcal{F}$ y, si consideráramos un aminoácido menos de la subsecuencia, ya sea al principo o al final, dicha secuencia pertenece a $\mathcal{F}$. Observemos que esto es precisamente lo que definimos anteriormente como *subsecuencia única minimal* sobre un conjunto de cadenas, en este caso, aplicado a proteínas. El objetivo de esto es, dada una familia, encontrar para cada proteína el (o los) *minimal tags* mínimos, es decir, los de menor largo. A partir de esto tendremos una noción de cuál es la mínima cantidad de aminoácidos que hace falta para identificar unívocamente a una proteína dentro de su familia.

Como ya mencionamos, para implementar la búsqueda de *minimal tags* usaremos el algoritmo propuesto en Ilie et al. [16], extendido a un conjunto de secuencias como fue expuesto anteriormente. De esta manera, a partir de un conjunto de secuencias queremos obtener, para cada una de ellas, el conjunto de todas sus *subcadenas únicas minimales*. Recordemos que el método propuesto como extensión del algoritmo consistía en concatenar todas las cadenas del conjunto separadas por un caracter delimitador y luego aplicar el algoritmo sobre esta cadena resultante con una leve modificación sobre la manera en la que se calcula el LCP array. Finalmente esto nos otorgaba un conjunto de *subcadenas únicas minimales* como resultado.

Aquí es importante observar que en este conjunto resultado, así como está definido, perdemos la capacidad de asociar qué *subcadena única minimal* pertenece a qué cadena de las que originalmente conformaban el conjunto. Vamos a necesitar implementar una

forma de identificación que no sea excesivamente costosa. Esto lo vamos a basar en un dato que tenemos para cada *subcadena única minimal*: la posición dentro de la cadena concatenación. Lo que vamos a hacer es armar un *índice* que nos permita buscar, dada una posición en la cadena concatenación, cuál era la cadena del conjunto a la que pertenecía. Esto lo implementamos con un árbol binario autobalanceado que vamos a armar en simultáneo con la concatenación de las cadenas del conjunto. Cada vez que agregamos una cadena a la cadena concatenación, tomamos la posición final actual y la agregamos al árbol asociándola con la cadena (proteína) correspondiente. Luego, dada una *subcadena única minimal* basta tomar su posición inicial y buscar en el árbol el nodo que representa la posición inmediata súperior para saber a qué proteína pertenece. El costo adicional de este procedimiento en cuanto a su complejidad temporal es, si consideramos $|S|$ a la cantidad de cadenas (proteínas) en el conjunto, $|S|log(|S|)$ para armar el índice y luego $log(|S|)$ por cada *subcadena única minimal* en el resultado para saber a qué cadena (proteína) pertenece. La complejidad espacial adicional es $|S|$ que es el tamaño que ocupa el índice.

A continuación se muestran los resultados de calcular todos los *minimal tags* para la familia de proteínas *Ankyrin*. A lo largo de este trabajo, cuando nos refiéramos a la familia *Ankyrin*, nos referiremos a un subconjunto de la misma que es el utilizado por Turjanski et al. en [6]. Esto nos facilitará hacer ciertas comparaciones de resultados con otros obtenidos en ese trabajo. El procedimiento para obtener este subconjunto es el utilizado en Turjanski et al. [6] y se decribe allí de la siguiente manera:

"Sobre Uniprot Uniref90 se corrio hmmsearch desde el hmmer ajustado para una familia HMM específica tomada de PFAM. Se incluyeron sólo las secuencias que contienen al menos una correspondencia con una familia específica hmm. Se excluyeron secuencias de proteínas que contuvieran residuos indefinidos o ambiguos (X, B, Z, J)"

En esta familia hay 38.051 proteínas (es una de las más grandes de nuestro set de datos), las cuales tienen en su totalidad 24.018.635 aminoácidos. En la *Figura 2.2* mostramos para el total de los *minimal tags* encontrados para todas las proteínas, cuántos hay de cada largo. Observemos que, para los largos menores, la cantidad de *minimal tags* encontrados es varios ordenes de magnitud mayor que para los largos mayores. Para largos mayores a 40 esta cantidad se mantiene por debajo de los 250 y decrece rápidamente. Notemos que a pesar de haber muy pocos casos que se podrían considerar como *outliers*, llega a haber *minimal tags* de hasta alrededor de 600 caracteres. Recordemos que una proteína puede tener más de un *minimal tag*.

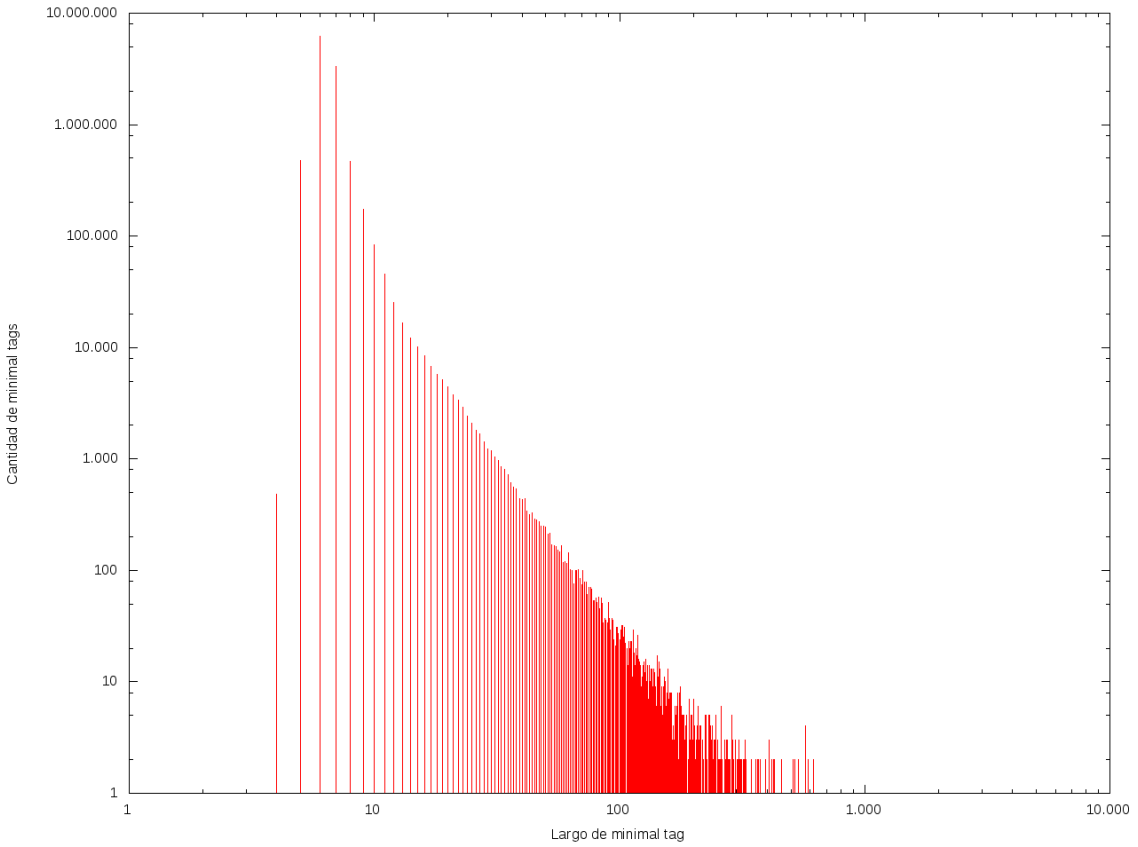


Figura 2.2: Cantidad de *minimal tags* encontrados, en función de su largo, para todas las proteínas de la familia *Ankyrin*

En la figura anterior vimos una descripción general de la distribución de los largos de los *minimal tags* encontrados para toda la familia *Ankyrin*. Podemos especular con que al encontrar una gran concentración de *minimal tags* de poca cantidad de caracteres esto nos va a servir para poder identificar muchas proteínas. Pero antes de poder llegar a esa conclusión necesitamos poder ver los datos discriminados por proteína, después de todo es cada proteína particular lo que queremos poder distinguir de las demás. Es por eso que calculamos ciertos datos agrupados por proteína. En las siguientes figuras mostraremos para cada una de las 38.051 proteínas de la familia *Ankyrin* un valor particular que nos ayuda a caracterizar los *minimal tags* correspondientes a esa proteína en concreto.

En la *Figura 2.3* se muestra, para cada proteína, la cantidad de *minimal tags* encontrados. Al tener más *minimal tags* una misma proteína podemos conjeturar que posee más chances de tener alguno considerablemente chico en cantidad de caracteres. O incluso tener varios *minimal tags* suficientemente chicos y que todos sirvan para identificarla en una forma eficiente. Tengamos en cuenta que si un *minimal tag* fuera demasiado grande, por más que identifique a una proteína unívocamente, no sería adecuado para tal propósito.

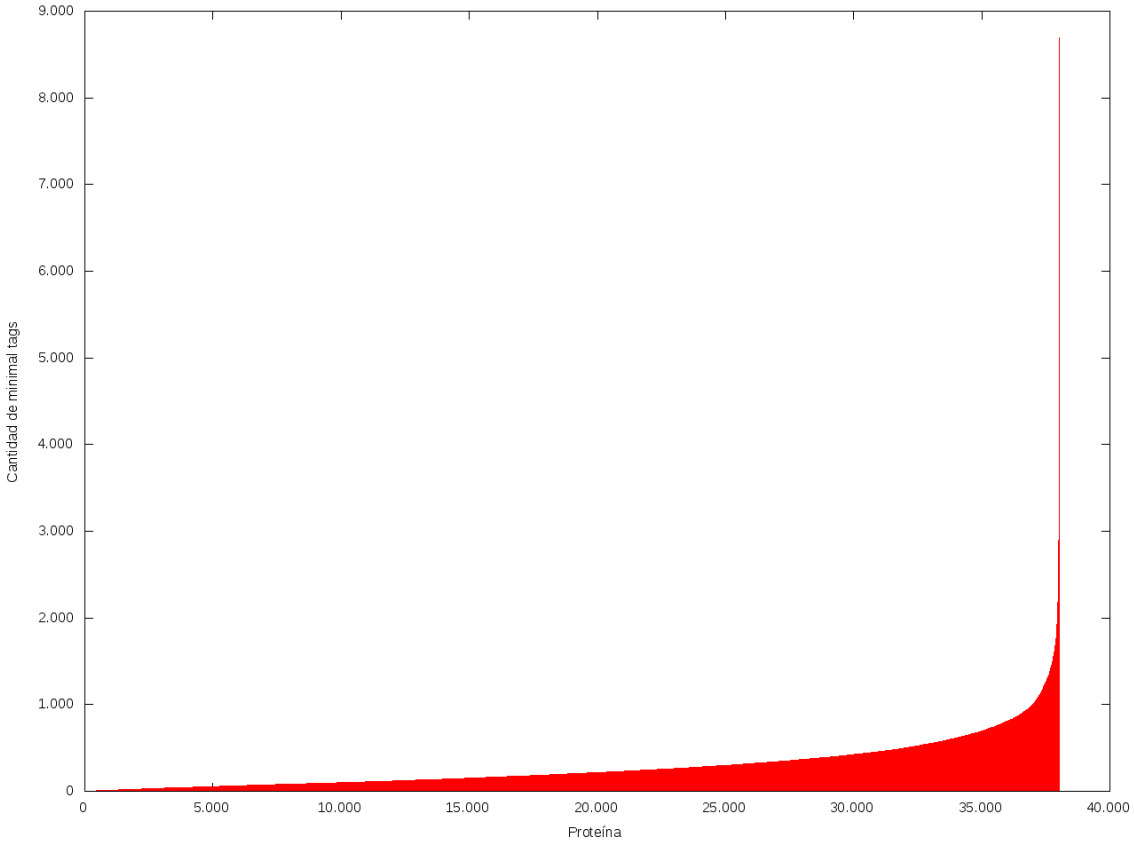


Figura 2.3: Cantidades de *minimal tags* para cada una de las 38.051 proteínas de la familia *Ankyrin*, ordenadas en forma ascendente según cantidad de *minimal tags*

En la *Figura 2.4* mostramos para cada proteína el largo mínimo, promedio y máximo, entre los largos de sus *minimal tags*. Para una mejor apreciación en las *Figuras 2.5, 2.6 y 2.7* se muestran cada uno de estos valores por separado ordenando las proteínas según su valor, en orden ascendente. Podemos ver que, si bien algunas proteínas tienen *minimal tags* de hasta 100 aminoácidos y unas pocas incluso más, la mayoría se mantiene considerablemente por debajo de este número, entre unos 10 y 40 aminoácidos. El promedio nos da una idea de que si bien cada proteína puede tener algún *minimal tag* de tamaño considerable, la mayoría tiene un tamaño inferior, alrededor de 10 aminoácidos o menor. Observemos que los promedios se mantienen, en general, entre 6 y 7. Finalmente la gran mayoría de los mínimos es 5. Calculamos exactamente cuántos y hay un 88,01 % de las proteínas que pueden ser identificadas unívocamente por al menos un *minimal tag* de exactamente 5 caracteres, mientras que un 10,21 % por un *minimal tag* de exactamente 6.

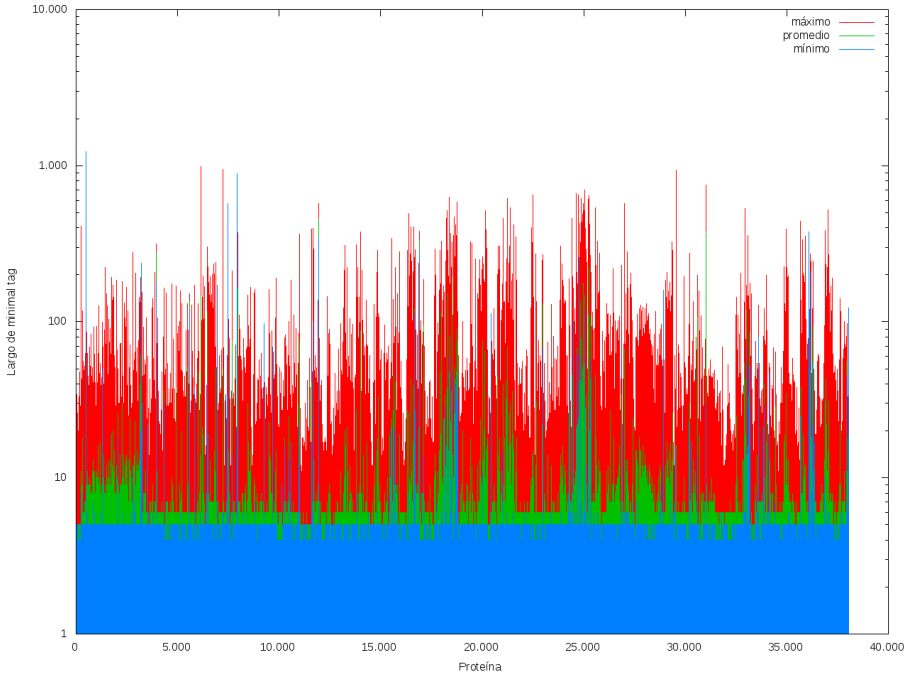


Figura 2.4: Longitud máxima, mínima y promedio de *minimal tags* para cada una de las 38.051 proteínas de la familia *Ankyrin*

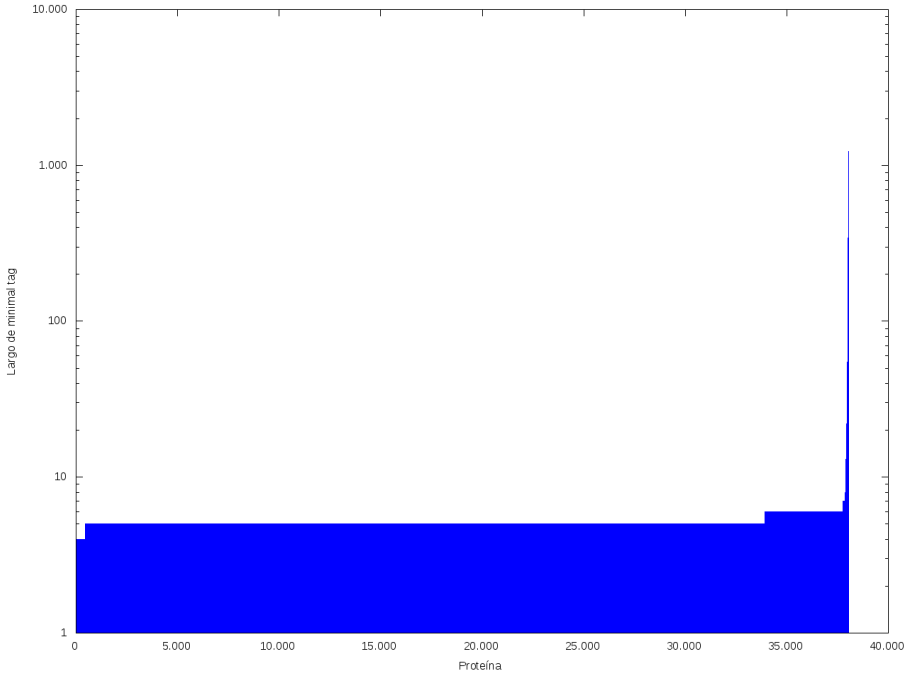


Figura 2.5: Mínimo largo de los *minimal tags* encontrados para cada una de las 38.051 proteínas de la familia *Ankyrin* ordenados en forma ascendente según el largo del *minimal tag*

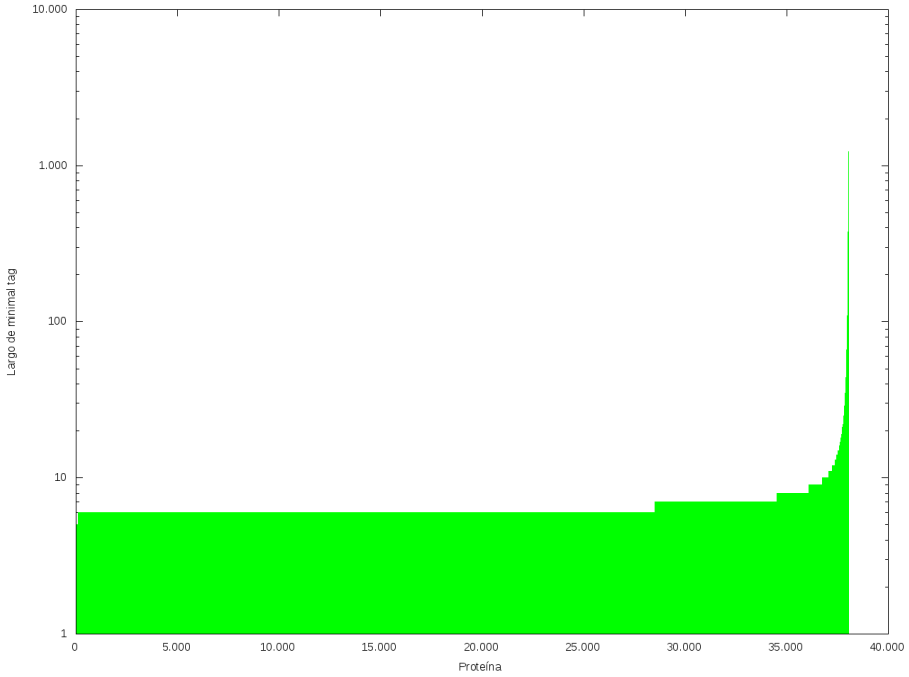


Figura 2.6: Largo promedio de los *minimal tags* encontrados para cada una de las 38.051 proteínas de la familia *Ankyrin* ordenados en forma ascendente según el largo del *minimal tag*

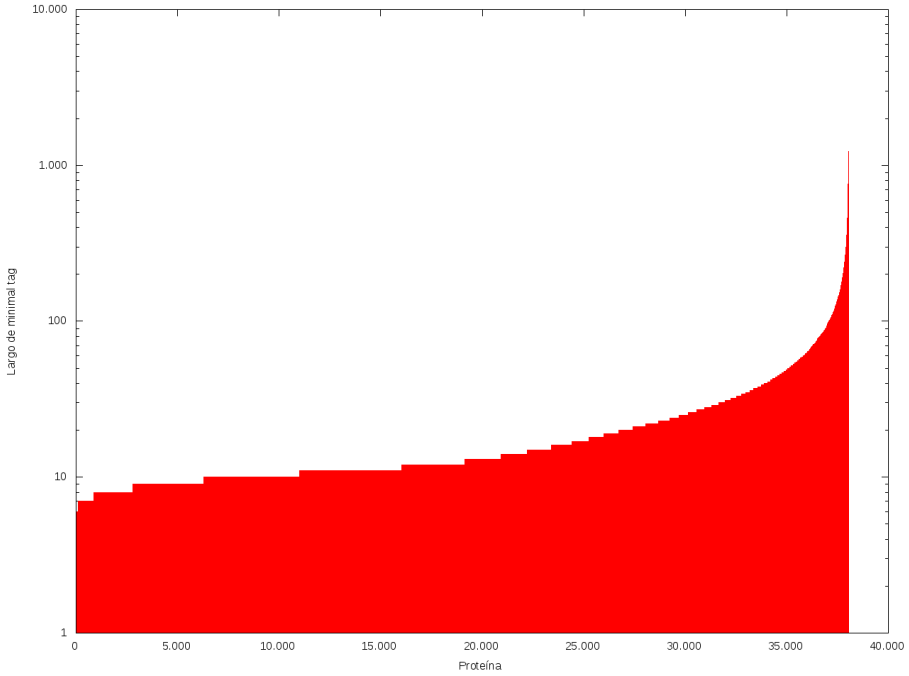


Figura 2.7: Máximo largo de los *minimal tags* encontrados para cada una de las 38.051 proteínas de la familia *Ankyrin* ordenados en forma ascendente según el largo del *minimal tag*

2.2. Familiaridad

2.2.1. Cálculo de familiaridad

Llamaremos *algoritmo de familiaridad* al algoritmo que usaremos para cuantificar las chances de que una proteína pertenezca a una cierta familia. Este algoritmo tomará como entrada un conjunto de proteínas de una misma familia y una proteína a evaluar y retornará un valor, que representará la afinidad de dicha proteína con esa familia. A este valor lo llamaremos *familiaridad*. El algoritmo de familiaridad consiste de 2 etapas.

En una primera etapa busca los patrones de repetición con los que caracteriza a una familia. Como explicamos anteriormente, dada la representación de las proteínas como cadenas de caracteres, podemos usar el algoritmo propuesto por Ilie et al. [16] para buscar repeticiones súper maximales dentro de una única proteína. Y dada la extensión que propusimos en el capítulo anterior, que permite utilizar el algoritmo en base a un conjunto de cadenas, también podemos buscar repeticiones súper maximales en un conjunto de proteínas. Vamos a buscar repeticiones súper maximales dentro de una familia, que luego usaremos para caracterizarla. A partir de esto podremos calcular la $familiaridad$ de una proteína de test dada, con dicha familia.

Una segunda etapa consiste en una implementación de la *función de familiaridad* propuesta por Turjanski et al. [6]. La función de familiaridad mide qué proporción de una determinada proteína (representada por una secuencia) está *cubierta* por las repeticiones de una cierta familia. La hipótesis es que cuanto mayor es el valor de la familiaridad más probable es que la proteína pertenezca a la familia.

Función de familiaridad

La *cobertura* de una cierta secuencia a evaluar, que llamaremos *secuencia-test*, por un conjunto de secuencias, es una medida de la cantidad de posiciones en la *secuencia-test* que están *cubiertas* por dichas secuencias en el conjunto. Una secuencia particular del conjunto *cubre* una serie de posiciones de la *secuencia-test* si la subsecuencia que se encuentra en esas posiciones es igual a la secuencia del conjunto. Vamos a notar $\mathcal{A}$ como nuestro alfabeto, $\mathcal{A}^*$ como el conjunto de todas las secuencias posibles sobre $\mathcal{A}$, y $\mathcal{P}(\mathcal{A}^*)$ como el conjunto de partes de $\mathcal{A}^*$ que representa la colección de todos los diferentes conjuntos de secuencias sobre $\mathcal{A}$. Usaremos $\mathbb{N}$ para el conjunto de números naturales y $\mathbb{Q}$ para el de racionales.

Definimos formalmente a la *función de cobertura*: $\mathcal{A}^* \times \mathcal{P}(\mathcal{A}^*) \to \mathbb{Q}$ tal que para una secuencia dada s y un conjunto de secuencias R

$$cobertura(s, R) = \frac{\#\{j : \exists i \in \mathbb{N}, \exists r \in R, s[i..i + |r| - 1] = r\}}{|s|}$$

Así, $cobertura(s, R)$ es un número racional entre 0 y 1. Por ejemplo si tomamos $s = MKPSLVSFSEKKLVVS$ y $R = \{VS, LV, SEK, MK\}$ vamos a obtener:

$$\underline{MK}PS\underline{LVS}F\underline{SEK}K\underline{LVVS}$$

Entonces tendremos $cobertura(s, R) = 12/16 = 0{,}75$

Observemos que no es relevante cuántas secuencias del conjunto R *cubren* una determinada posición de s, sólo importa si la posición está *cubierta* o no.

Vamos a denotar, dados una secuencia s y un número natural n, $\mathcal{M}(s, n)$ al conjunto de todas las repeticiones súper maximales de s de largo mayor o igual a n. Por definición diremos que $\mathcal{M}(t, 0) = \mathcal{P}(\mathcal{A}^*)$

A partir de esto definimos la *función de familiaridad*: $\mathcal{A}^* \times \mathcal{A}^* \to \mathbb{Q}$ para una secuencia s y una secuencia t:

$$familiaridad(s,t) = \frac{cobertura(s,\mathcal{M}(t,0))+cobertura(s,\mathcal{M}(t,|s|))}{2} + \sum_{i=1}^{|s|-1} cobertura(s, \mathcal{M}(t,i))$$

Notemos que dada esta fórmula, la $familiaridad$ va a ser un número entre 0 y $|s|$. Dada la aplicación que le queremos dar a la noción de $familiaridad$ vamos a definir la $familiaridad$ entre una proteína p y una familia de proteínas $\mathcal{F}$ como la $familiaridad$ entre la secuencia con la que representamos a la proteína p y las repeticiones súper maximales resultantes de la concatenación de todas las secuencias con las que representamos cada una de las proteínas de la familia $\mathcal{F}$. Consideramos esta concatencación de secuencias separándolas por un caracter delimitador de la misma manera que fue explicado en la sección Extensiones.

En el trabajo de Turjanski et al. [6] se observó que, para valores chicos de i, $cobertura(p, \mathcal{M}(\mathcal{F}, i))$ arroja valores de cobertura mayormente altos, sin embargo para valores levemente mayores de i la cobertura baja drásticamente y se mantiene en valores tan bajos que no aportan información significativa a la hora de calcular la $familiaridad$ entre una determinada proteína y una cierta familia. Estas observaciones empíricas fueron las que se hicieron en el trabajo de Turjanski et al. [6] y llevaron a que se utilice para el cálculo de $familiaridad$ sólo las coberturas para valores de i en el rango de 0 a 10. Por estos motivos y para poder comparar nuestro trabajo con éste, vamos a utilizar, en una primera instancia, esta misma restricción. De este modo definimos la fórmula de $familiaridad$ restringida a una cierta cota $\kappa \in \mathbb{N}$ como:

$$familiaridad_\kappa(s,t) = \frac{cobertura(s,\mathcal{M}(t,0))+cobertura(s,\mathcal{M}(t,\kappa))}{2} + \sum_{i=1}^{\kappa-1} cobertura(s, \mathcal{M}(t,i))$$

Como ya mencionamos, vamos a utilizar una cota $\kappa = 10$

Ilustremos la idea con un ejemplo sencillo.
Sean la proteína:
$p = MKPSLVSFSEKKLVVS$
y la familia:
$\mathcal{F} = \{LVVS, LVKKLV, VSSEK, KKLVSEK\}$
la secuencia con la que representaremos a la familia $\mathcal{F}$ es:
$t = LVVS\#LVKKLV\#VSSEK\#KKLVSEK\#$.
Si calculamos el conjunto de repeticiones súper maximales de t obtenemos

SMR(t)= $\{LV, VS, SEK, KKLV\}$. Luego:

$$\mathcal{M}(t,0) = \mathcal{P}(t) \text{ (por definición)}$$
$$\mathcal{M}(t,1) = \{LV, VS, SEK, KKLV\}$$
$$\mathcal{M}(t,2) = \{LV, VS, SEK, KKLV\}$$
$$\mathcal{M}(t,3) = \{SEK, KKLV\}$$
$$\mathcal{M}(t,4) = \{KKLV\}$$
$$\mathcal{M}(t,i) = \emptyset \ \forall i \geq 5$$

Si calculamos la cobertura de p, se puede ver que:

$cobertura(p, \mathcal{M}(t,0)) = 1$	$\underline{MKPSLVSFSEKKLVVS}$
$cobertura(p, \mathcal{M}(t,1)) = 11/16 = 0{,}6875$	$MKPS\underline{LVS}F\underline{SEKKLVVS}$
$cobertura(p, \mathcal{M}(t,2)) = 11/16 = 0{,}6875$	$MKPS\underline{LVS}F\underline{SEKKLVVS}$
$cobertura(p, \mathcal{M}(t,3)) = 6/16 = 0{,}375$	$MKPSLVSF\underline{SEKKLV}VS$
$cobertura(p, \mathcal{M}(t,4)) = 4/16 = 0{,}25$	$MKPSLVSFSE\underline{KKLV}VS$
$cobertura(p, \mathcal{M}(t,5)) = 0/16 = 0$	$MKPSLVSFSEKKLVVS$
$cobertura(p, \mathcal{M}(t,6)) = 0/16 = 0$	$MKPSLVSFSEKKLVVS$
$cobertura(p, \mathcal{M}(t,7)) = 0/16 = 0$	$MKPSLVSFSEKKLVVS$
$cobertura(p, \mathcal{M}(t,8)) = 0/16 = 0$	$MKPSLVSFSEKKLVVS$
$cobertura(p, \mathcal{M}(t,9)) = 0/16 = 0$	$MKPSLVSFSEKKLVVS$
$cobertura(p, \mathcal{M}(t,10)) = 0/16 = 0$	$MKPSLVSFSEKKLVVS$

Así, cuando calculemos la $familiaridad$ resultará:

$$familiaridad_{10}(p, \mathcal{F}) = \frac{cobertura(p,\mathcal{M}(t,0)) + cobertura(p,\mathcal{M}(t,10))}{2} + \sum_{i=1}^{9} cobertura(s, \mathcal{M}(t,i)) =$$

$$= \frac{1+0}{2} + 0{,}6875 + 0{,}6875 + 0{,}375 + 0{,}25 + 0 + 0 + 0 + 0 + 0 + 0 = 2{,}5$$

Es importante observar que al momento de calcular la $familiaridad$ no es relevante dónde se encuentran las ocurrencias de las repeticiones súper maximales que resultan de la familia de proteínas $\mathcal{F}$, ni tampoco *cuántas* ocurrencias hay de cada repetición súper maximal en la familia. Lo único que nos interesa es la subsecuencia que las representa (el patrón). Este es un detalle importante implementativamente ya que vamos a necesitar menos espacio en memoria para implementar el algoritmo.

En la *Figura 2.8* vemos una representación gráfica de la interacción entre *cobertura* y *familiaridad* para el cálculo de coberturas realizado por Turjanski et al. en [6] sobre la proteína IκBα *(Uniprot ID: P25963)*, perteneciente a la familia *Ankyrin*, a partir de distintos conjuntos de repeticiones. Estos conjuntos fueron computados a partir de las familias Ankyrin, DEH, WD, HET, y de la proteína IκBα por separado.

Podemos ver como los valores de cobertura decrecen a medida que aumenta i. El caso de la cobertura sobre el conjunto de repeticiones generado por IκBα aislada es particular, pues al ser una única proteína tiene una longitud de secuencia varios órdenes de magnitud menor que una familia entera, por eso es esperable que la cobertura que genera sea mucho menor.

Por otro lado, es importante notar como para valores de i hasta 5 el resto de la coberturas se mantienen muy similares y luego difieren marcadamente. La familia *Ankyrin*, que es la familia a la que pertenece la proteína de test, continúa generando valores de cobertura altos para i mayores o iguales a 6, mientras que en las otras familias se observa como las coberturas bajan drásticamente hasta quedar en 0.

Para entender mejor como se genera la noción de $familiaridad$, es de gran importancia destacar la relación que esta mantiene con la *cobertura*. Teniendo presente la función de familiaridad podemos ver como el valor de $familiaridad$ se ve representado por el área bajo la curva generada por los valores de cobertura. Es a partir de esta intuición que se forma la idea de familiaridad.

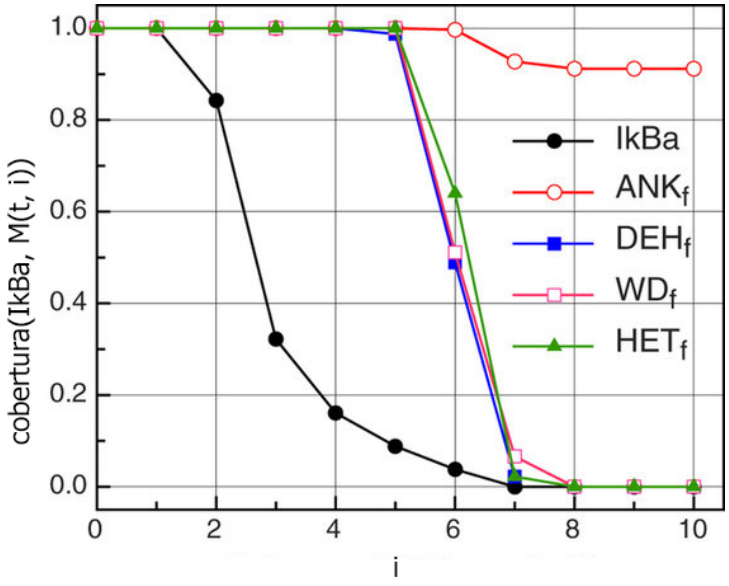


Figura 2.8: Valores de $cobertura(I\kappa B\alpha, \mathcal{M}(t,i))$ para t=Ankyrin, DEH, WD, HET, IκBα y $i = 0, ..., 10$. Imagen obtenida del trabajo de Turjanski et al. [6].

2.2.2. Reformulación del cálculo y comparación con trabajo previo

En esta sección exponemos la fundamental diferencia que proponemos sobre el trabajo previo realizado por Turjanski et al. [6], y luego contrastamos los resultados obtenidos.

Como se explicó anteriormente tanto la función de $familiariadad$, como la de *cobertura* fueron obtenidas del trabajo de Turjanski et al. [6]. La alteración que aquí proponemos es sobre la noción de *repetición*. Recordemos que tenemos por un lado *repeticiones maximales* (MR) y por otro *repeticiones súper maximales* (SMR).

En el trabajo realizado por Turjanski et al. [6], cuando se define $\mathcal{M}(s, n)$, para luego ser usado en la *función de familiaridad*, se lo define respecto de MR. La reformulación propuesta en este trabajo es definir $\mathcal{M}(s, n)$ a partir del concepto de SMR. De esta manera las *coberturas* calculadas sobre las proteínas de test serán a partir de SMR en lugar de MR.

Cabe recordar que, a partir de las definiciones de MR y SMR dadas, tanto el algoritmo propuesto por Ilie et al. [16] como la extensión propuesta sobre el mismo anteriormente nos permiten buscar SMR, y no MR. El concepto de SMR es más sencillo y se requiere una complejidad computacional menor para buscarlos. Buscar los SMR de una cadena de tamaño n tiene una complejidad de $O(n)$, mientras que buscar los MR de $O(n\ log\ n)$. Además, el algoritmo con el cuál buscamos SMR es más sencillo de comprender e implementar. Es por esto que consideramos una potencial mejora el hecho de poder usar SMR en lugar de MR en la *función de familiaridad*.

El algoritmo utilizado por Turjanski et al. [6] para computar MR es el presentado por Becher et al. [21]. Es importante destacar que el algoritmo que uilizamos para buscar SMR no significa una mejora sobre este, si no que tiene un propósito muy diferente. Si bien SMR y MR son muy similares en cuanto a su definición, buscar MR conlleva tener que manipular una cantidad de información mucho mayor sobre las posibles repeticiones ya que se debe tener en cuenta la cantidad de ocurrencias de cada una. Es en la posibilidad de que los SMR sean más adecuados que los MR para calcular familiaridad que esperamos encontrar una ventaja.

Antes de proseguir con los resultados de aplicar SMR sobre el cálculo de familiaridad y de contrastarlos con los del trabajo de Turjanski et al. [6] realizaremos un pequeño análisis sobre las diferencias entre MR y SMR en sí mismos, al ser buscados sobre una secuencia de estudio particular: Ankyrin.

Análisis sobre diferencias entre SMR y MR en Ankyrin

Buscaremos SMR y MR sobre una misma secuencia y realizaremos un análisis sobre como difieren los conjuntos de SMR y MR encontrados. A partir de esto intentaremos conjeturar sobre el posible impacto de las diferencias encontradas en los cálculos de *cobertura* y $familiaridad$. Para esto utilizaremos las proteínas de la familia Ankyrin. La familia de proteínas Ankyrin contiene motivos repetitivos en estructura a simple vista, lo que motivó al grupo de Turjanski et al. [6] a analizar si esa repetitividad se mantenía a nivel de secuencia (estructura primaria). Se analizó la ocurrencia de repeticiones exactas en miembros de la familia de proteínas repetitivas Ankyrin (ANK) para la cual muchas de sus estructuras terciarias ya han sido resueltas. Las Ankyrins son una de las clases más abundantes de proteínas repetitivas naturales y han sido extensivamente estudiadas [22].

Más allá de la cobertura que generan los MR o SMR en una proteína particular, nos va

a interesar analizar estos conjuntos y ver cómo difieren. Para esto vamos a tomar los MR y SMR, esta vez no de una proteína particular, sino de toda la familia, después de todo es esto lo que vamos a utilizar luego para calcular la *familiaridad*. Obtuvimos los MR y SMR de la familia Ankyrin utilizando para cada conjunto el correspondiente método. Llamaremos a cada uno de estos conjuntos de secuencias ANK_{MR} y ANK_{SMR} respectivamente. Recordemos que si bien la definición de MR y SMR difieren, ambos conjuntos son patrones de repetición sobre la misma secuencia base: la concatenación de todas las proteínas de la familia Ankyrin, por lo tanto, como se explica a continuación, van a tener una gran cantidad de elementos en común.

Primero es importante destacar que no va a haber secuencias en ANK_{SMR} que no estén en ANK_{MR}, es decir $ANK_{SMR} \subseteq ANK_{MR}$. Esto es fácil de ver observando ambas definiciones. Es en la noción de extensibilidad que las definiciones difieren. Una repetición se considera MR, si al extender la subsecuencia que la reresenta, la cantidad de ocurrencias de esa subsecuencia disminuye. Una repetición se considera SMR, si al extender la subsecuencia que la reresenta, esta pasa a ser la única ocurrencia. Puesto de otra manera, la cantidad de ocurrencias pasa a ser una. De este modo queda claro por qué una definición abarca a la otra.

Veamos esto con un ejemplo (*Figura 2.9*). Consideremos la cadena *TAGATGATAGAATCTGAGTTCAGAGTAGAGATAGAA* y observemos la subcadena *GA* marcada en columnas amarillas. Como podemos ver, hay 8 ocurrencias de *GA* y cada una de sus extensiones posibles, ya sea a izquierda o a derecha, tiene menos cantidad de ocurrencias: *AGA*: 6, *GAT*: 3, *TGA*: 2, *GAA*: 2, *GAG*: 3. Al mismo tiempo todas estas extensiones siguen siendo repeticiones. Por eso es que *GA* es MR pero no SMR.

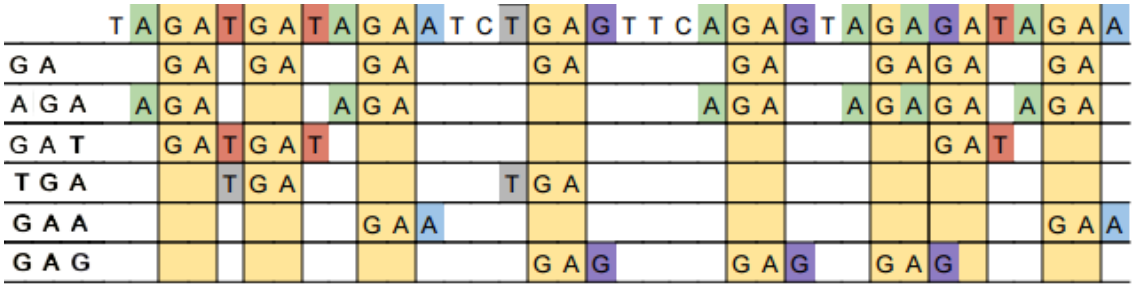


Figura 2.9: Ejemplo de subsecuencia que es MR pero no SMR: *GA*

Recordemos que si una repetición es SMR, entonces también es MR. Nos va a interesar analizar $ANK_{MR} - ANK_{SMR} = ANK_{diff}$. En principio podemos conjeturar que, al contar con menor cantidad de SMR que de MR las coberturas generadas en proteínas de test van a ser menores y también así los valores de familiaridad obtenidos. Contamos además, con la especulación intuitiva de que repeticiones muy cortas podrían cubrir trivialmente gran parte de una cadena cualquiera dada. Esto llevaría, a su vez, a que estas repeticiones más cortas no sean buenas para poder estimar la familiaridad entre una proteína de test y una familia. Siguiendo este razonamiento podemos especular con que, en caso de encontrar una cantidad considerable de repeticiones cortas en ANK_{diff}, las estimaciones de familiaridad realizadas usando SMR podrían ser mejores.

La noción que consideramos de mayor importancia para caracterizar la composición del conjunto ANK_{diff} fue analizar la frecuencia de las repeticiones según su largo. De este modo contamos cuántas secuencias encontramos de largo 1, cuántas de largo 2, etc. En

la *Figura 2.10* podemos observar que la gran mayoría de los elementos del conjunto son secuencias de largos relativamente chicos. Encontramos secuencias de 462 largos diferentes en ANK_{diff}. Aquí mostramos qué proporción del total de elemetos del conjunto (eje y) representan las secuencias de un largo determinado (eje x). Se puede ver cómo, para secuencias de largo mayor a 6, rápidamente a medida que aumenta el largo de las secuencias la proporción que representan disminuye. Como las secuencias de largos mayores a 30 no representaban individualmente una proporción significativa del total, no fueron tenidas en cuenta en los gráficos. En su total las secuencias de largos mayores a 30 representan entre un 3 % y 4 %. En la *Figura 2.11* mostramos la proporción acumulada del total de elementos del conjunto que representan las secuencias de un largo menor o igual a n, para los n representados en el eje x. A partir de estos resultados podemos observar que hay una mayor cantidad de secuencias que son MR pero no SMR de tamaños pequeños. Es importante destacar que para largos de secuencias menores a 4, no es que se hayan encontrado pocos elementos de esos largos porque los restantes eran SMR, por el contrario, hay esa cantidad porque esa es la máxima cantidad de elementos de ese largo que puede haber dado el tamaño del alfabeto utilizado, que es de 20 símbolos. En la *Tabla 2.1* mostramos, para las secuencias de largos de 1 a 6, cuántas de ellas encontramos en ANK_{diff} efectivamente comparado contra cuántas podría haber como máximo dado el tamaño del alfabeto.

Columna 1	*Columna 2*	*Columna 3*	*Columna 4*
$\|s\|$	# elem en ANK_{diff}	Máximo teórico ($20^{\|s\|}$)	$\frac{Columna2}{Columna3} * 100\,\%$
1	20	20	100 %
2	400	400	100 %
3	7.982	8.000	99.77 %
4	102.166	160.000	63.85 %
5	178.290	3.200.000	5.57 %
6	201.108	64.000.000	0.31 %

Tabla 2.1: Comparación con el máximo teórico de secuencias $s \in ANK_{diff}$ de largo de 1 a 6.

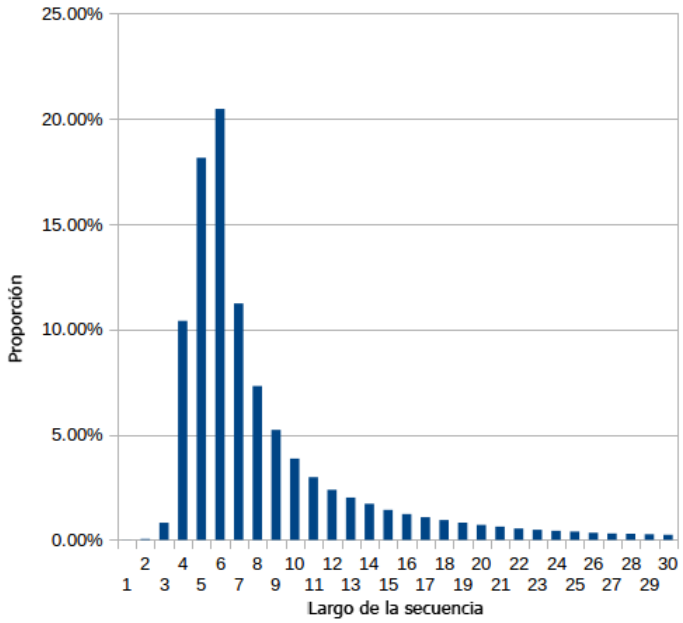


Figura 2.10: Proporción de secuencias de ANK_{diff}, según su largo, respecto al tamaño del conjunto. (Sólo se muestran los primeros 30 largos)

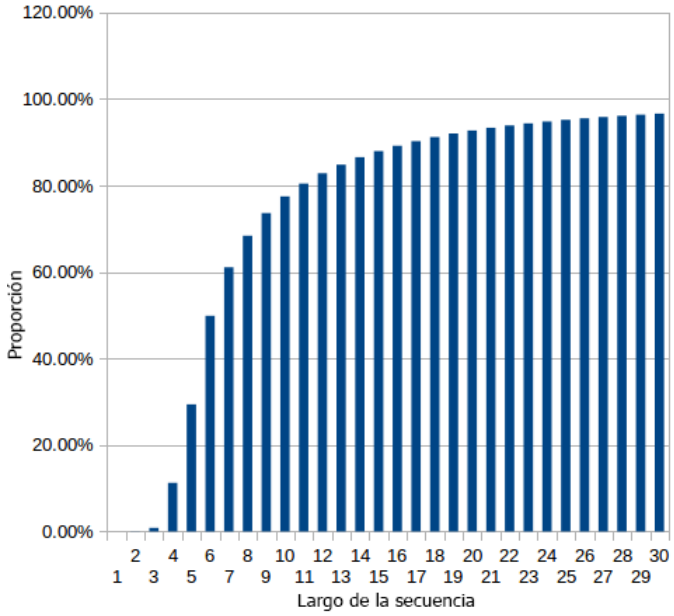


Figura 2.11: Proporción acumulada de secuencias de ANK_{diff}, según si su largo es menor o igual a un determinado valor, respecto al tamaño del conjunto.

En términos nominales podemos ilustrar la reducción en cantidad de cadenas que significó, para el caso de la familia *Ankyrin*, pasar de usar MR a SMR. El conjunto ANK_{MR} cuenta con 6.469.794 elementos, mientras que el conjunto ANK_{SMR} con 5.487.234. En consecuencia. ANK_{diff} tiene 982.560, lo cual significa una reducción del 15.18 % de ANK_{SMR} respecto de ANK_{MR}.

Recordemos que la idea es que el conjunto de SMR sea más o a lo sumo igual de representativo de la familia de proteínas que el conjunto de MR. Conjeturamos que cadenas cortas (3 o 4 caracteres) no son buenas para discriminar entre familias, pues tienen altas chances de generar una alta cobertura en una proteína de test, ya sea que esta pertenezca a dicha familia o no. Suponemos que en el caso general se va a dar que justamente las cadenas que dejemos de tener en cuenta al pasar a usar SMR, en lugar de MR, son las más cortas. A partir de estas nociones, en principio, podemos especular que el reemplazo de MR por SMR, nos podría llevar a tener valores de familiaridad más precisos. Más adelante se exponen distintos análisis puntuales donde se intenta mostrar evidencia acerca de esta conjetura.

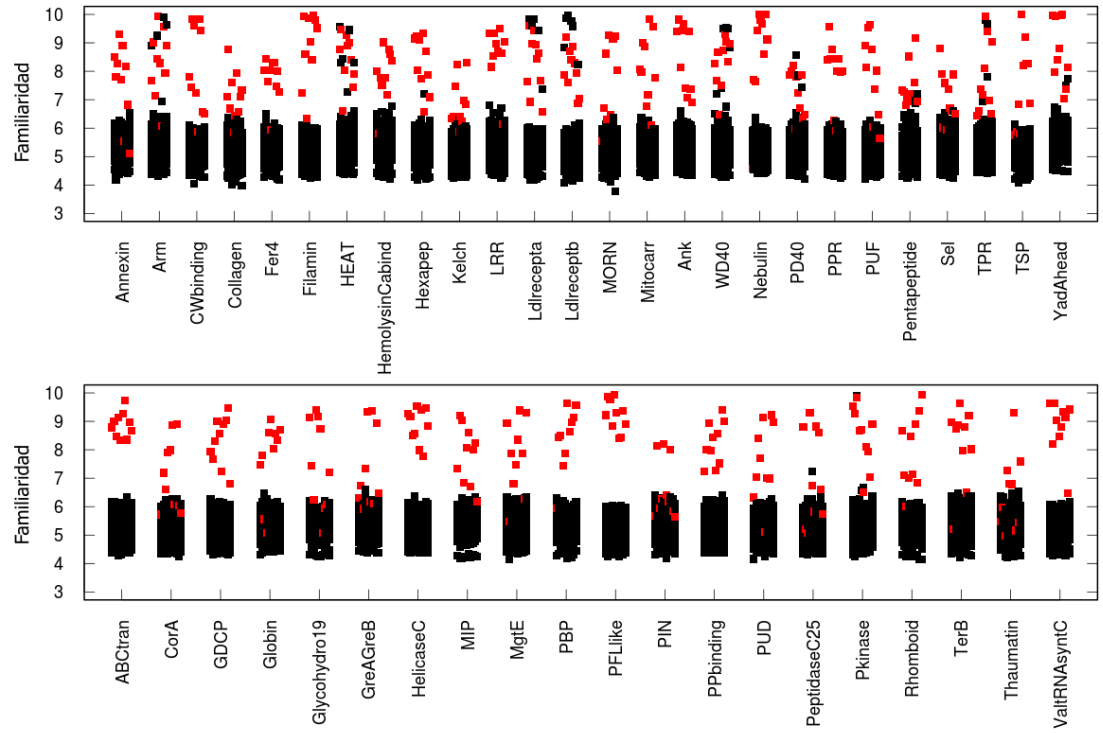
Resultados de familiaridad sobre un dataset de proteínas

En esta sección aplicaremos las reformulaciones propuestas anteriormente, sobre el cálculo de familiaridad, sobre un dataset de proteínas. El mismo fue obtenido del trabajo realizado por Turjanski et al. [6] donde se puede ver detalladamente como está conformado. Utilizar exactamente el mismo dataset nos permitirá poder comparar los resultados obtenidos, con los que previamente obtuvieron Turjanski et al. [6]. Este dataset contiene 26 familias de proteínas repetitivas y 20 familias de proteínas globulares. La cantidad de proteínas que conforma una familia varía de una familia a otra. Así mismo también varía la cantidad de aminoácidos entre distintas proteínas. De cada una de estas familias fueron excluídas 10 proteínas para utilizar como proteínas de test. De esta manera contamos con un conjunto de proteínas de test que sabemos a priori a qué familia pertenecen. Luego, idealmente eperaremos obtener valores de familiaridad mayores cuando calculemos la familiaridad de una proteína de test contra la familia a la cual pertenece y menores cuando calculemos la familiaridad contra otras familias. Cuando hablamos de la familiaridad entre una proteína de test y una familia nos referimos a la familia con sus respectivas proteínas de test excluidas.

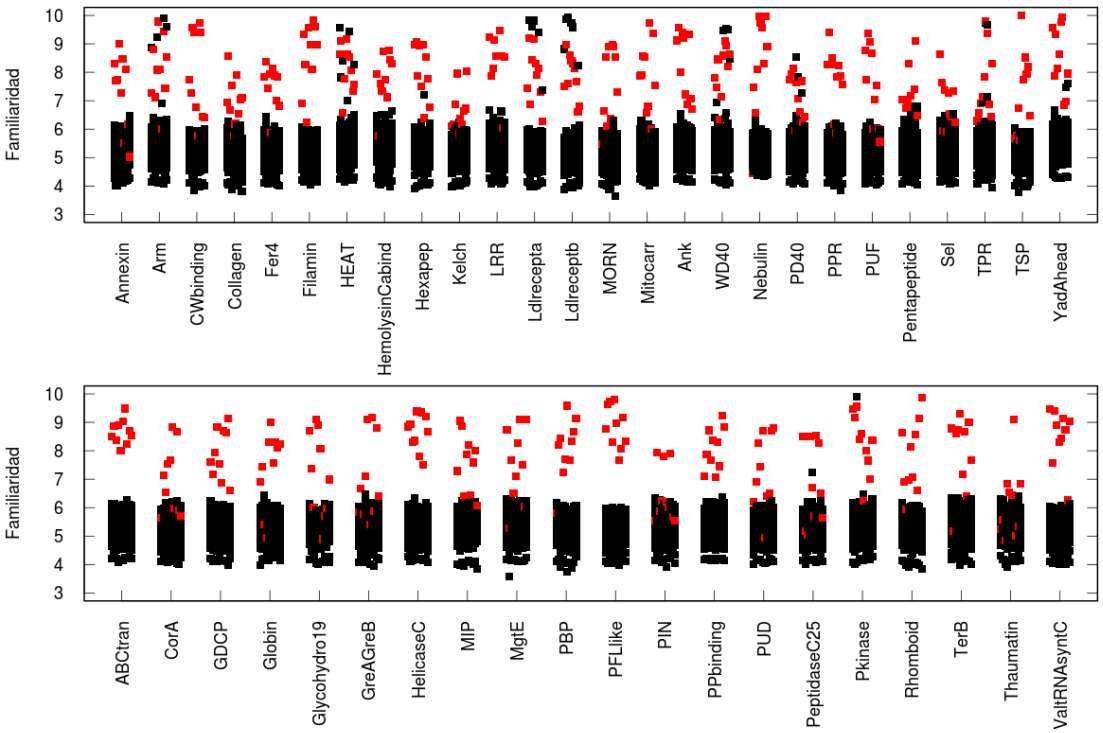
En la *Figura 2.12* se pueden ver los valores de familiaridad usando primero MR (*Figura 2.12a*), obtenidos del trabajo de Turjanski et al. [6], y luego usando SMR (*Figura 2.12b*), como fue explicado anteriormente. En los gráficos podemos ver, por cada familia, la familiaridad de cada proteína de test contra esa familia. Representamos este valor con un punto rojo si la proteína de test pertenece a dicha familia y negro en caso contrario.

Se puede observar que efectivamente, en la mayoría de los casos, la familiaridad cuando la proteína de test pertenece a la familia (puntos rojos) es notablemente mayor que cuando no (puntos negros). Se pueden ver también ciertas anomalías: *i)* algunas proteínas de test, que sin pertenecer a una familia, dan valores de familiaridad altos; *ii)* otras que, incluso perteneciendo a la familia dan valores de familiaridad bajos.

Es importante destacar que cuando comparamos las familiaridades usando MR y usando SMR se ve que, cuando se usan SMR, los valores suelen ser más bajos. Esto tiene sentido ya que recordemos que, como se expuso anteriormente, usar los SMR genera una cobertura, en principio, menor lo cual genera valores de familiaridad a su vez menores. En base a lo observado, e hipotetizando que esto es extrapolable a otros casos, podemos decir que como los valores de familiaridad dan menores tanto cuando la proteína de test pertenece a la familia como cuando no, la relación entre el promedio de los valores para cada uno de estos dos casos se mantiene. Esto hace que la precisión del algoritmo para decidir si una proteína de test pertenece a una familia o no, sea similar tanto al usar MR como SMR. Más adelante exponemos las diferencias entre los valores encontrados utilizando cada uno de los métodos.



(a) Familiaridad usando MR, tomado de Turjanski et al. [6]



(b) Familiaridad usando SMR.

Figura 2.12: Comparación de valores familiaridad usando *a)* MR y *b)* SMR. Puntos rojos representan familiaridades para proteínas de test *vs* su propia familia, puntos negros representan familiaridades para proteínas de test *vs* otras familias.

A continuación, en la *Figura 2.13* mostramos una medida de la diferencia de valores de familiaridad al usar el algoritmo utilizando MR y SMR. Para cada familia mostramos el promedio y el desvío estándar de las diferencias entre los valores al usar cada método, para cada una de las poteínas de test.

Como se puede observar, el promedio se mantiene reducido teniendo en cuenta que los valores de familiaridad oscilan entre 4 y 9, y el desvío estándar es pequeño.

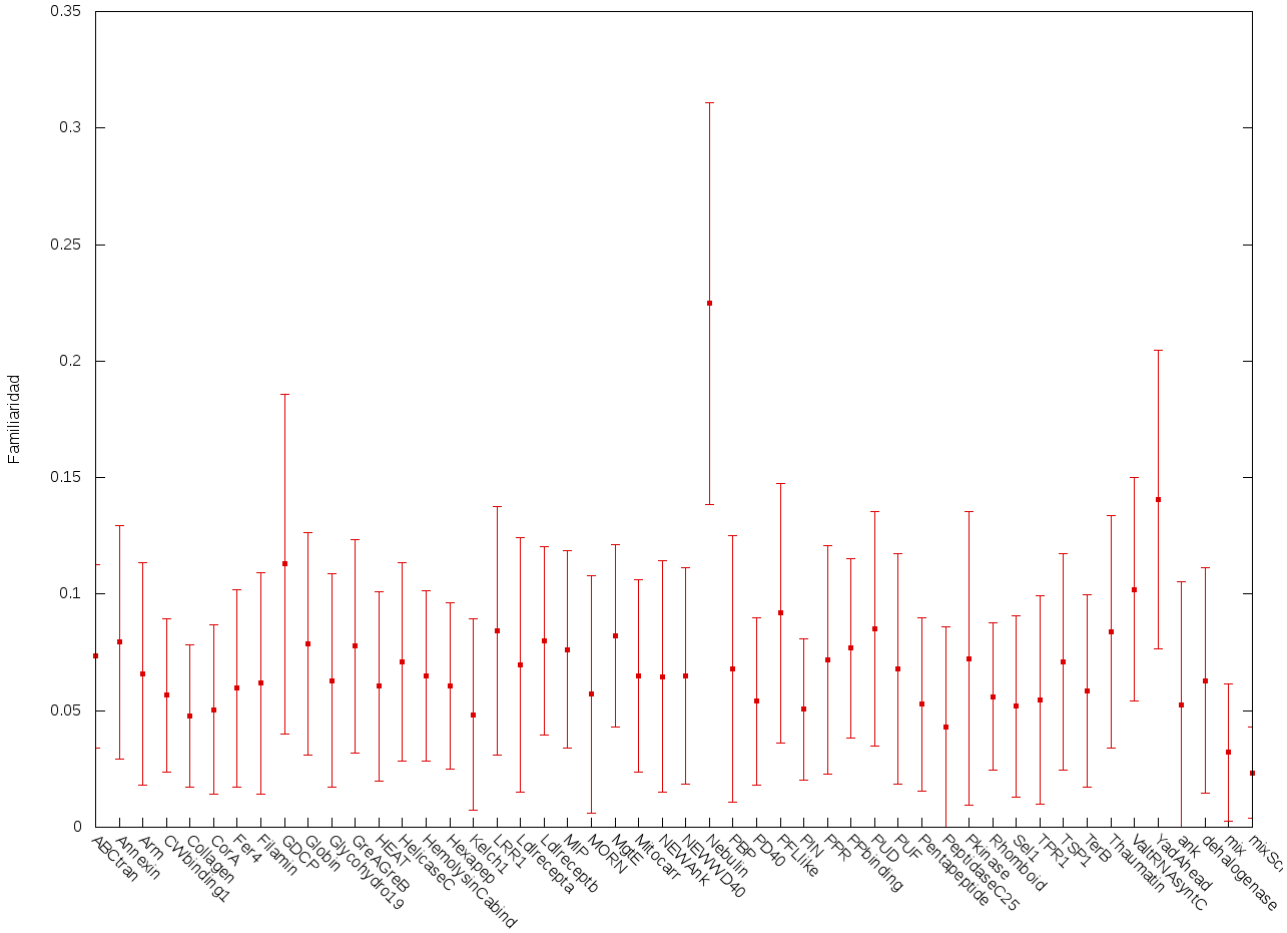


Figura 2.13: Promedio y desvío estándar de la diferencia entre valores de familiaridad utilizando, para cada una de las proteínas de test, MR y SRM

2.2.3. Familiaridad: Explorando alternativas

En la sección anterior usamos la noción de SMR aplicada al cálculo de familiaridad para obtener nuevos resultados que contrastamos contra los expuestos en Turjanski et al. [6] en donde se utiliza el concepto de MR. Ahora vamos a explorar distintas alteraciones en los cálculos de familiaridad y cobertura para evaluar cómo impacta esto en los resultados obtenidos. El objetivo es intentar aumentar la brecha entre valores de familiaridad para proteínas de test contra su propia familia y contra otras. Esto nos llevaría idealmente, a poder identificar más categoricamente cuándo una cierta proteína de test pertenece a una cierta familia y cuándo no.

Eliminación de la cota κ y redefinición del conjunto de cobertura

En primer lugar redefiniremos la fórmula de familiaridad de manera de no necesitar restringirla a una cota arbitraria κ. Recordemos que en el caso anterior usamos $\kappa = 10$, que estaba relacionado con el uso de MR y la necesidad de tratar de salvar las diferencias entre proteínas de test de distintos largos. Una proteína de test muy larga podría ser cubierta por MRs muy largos que en una proteína de test más corta no serían tenidos en cuenta. La hipótesis que dió lugar al uso de la cota era que incorporar las coberturas de largos mayores a 10, al cálculo de familiaridad, no aportaba demasiada información.

La segunda alteración sobre el cálculo de familiaridad que haremos es en la manera de calcular la cobertura. Recordemos que la fórmula de familiaridad consistía en la suma de distintos términos que tenían la forma $cobertura(p, \mathcal{M}(t, j))$ para distintos j. Esto era la cobertura de la proteína de test p por los elementos del conjunto $\mathcal{M}(t, j)$ que estaban definidos como los SMR de largo mayor o igual a j encontrados en la cadena t que representaba a una cierta familia. Esta idea de tomar, para cada cobertura, los SMR de largo **mayor o igual** a j, contribuía a de alguna manera otorgarles mayor *peso* a los SMR más largos, pues serían tenidos en cuenta en mayor cantidad de términos cobertura en la función de familiaridad. Esta noción puede surgir a partir de la intuición de pensar que los SMR más largos podrían ser más significativos a la hora de representar a una familia, pues cabe pensar que hay menos chances de encontrarlos que a SMR de largo menor. A su vez, al utilizar SMR de largo **mayor o igual** a j, la existencia de la cota κ no restringe el hecho de que se usen todos los SMR disponibles en el cálculo de cobertura. Observemos que un SMR de largo mayor a κ se estaría usando para todas las coberturas calculadas sobre un j menor a κ. Mientras que si usáramos solo los SMR de largo igual a j, todos los SMR de largo mayor a κ se descartarían. Redefiniremos el cálculo de cobertura sobre un nuevo conjunto $\mathcal{E}(t, j)$ que va a representar a todos los SMR de largo exáctamente igual a j. No descartaremos por completo la idea de *pesar* en distinta manera SMR de distinto largo, sino que la implementaremos de otra manera más adelante.

Es importante comprender la fuerte relación que existe entre utilizar una cota κ y considerar los SMR de largo mayor o igual a un j en la manera en como se define en $\mathcal{M}(t, j)$. Es por esto que aplicaremos simultaneamente ambas modificaciones sobre la fórmula de familiaridad: eliminar la cota κ y utilizar $\mathcal{E}(t, j)$ en lugar de $\mathcal{M}(t, j)$.

La fórmula resultante:

$$familiaridad_{\mathcal{E}}(s,t) = \frac{cobertura(s,\mathcal{E}(t,0))+cobertura(s,\mathcal{E}(t,|s|))}{2} + \sum_{i=1}^{|s|-1} cobertura(s, \mathcal{E}(t,i))$$

En la *Figura 2.14* podemos ver los resultados de aplicar esta nueva fórmula. Se puede observar que los valores de familiaridad son menores que usando la fórmula presentada anteriormente, a la que denominaremos $familiaridad_{\mathcal{M}}$ (*Figura 2.12b*). Es esperable que cada vez que calculamos una cobertura, al usar conjuntos de SMR de menos elementos (SMR de tamaño exactamente igual en vez de mayor o igual) esta sea menor o a lo sumo igual. Adicionalmente, hemos removido la cota $\kappa = 10$ impuesta anteriormente sobre las coberturas usadas. A partir de los resultados podemos especular que, si bien al momento de calcular cada familiaridad contamos con más coberturas (más términos en la sumatoria), cada una de estas coberturas sería sustancialmente menor que las obtenidas usando $\mathcal{M}(t, j)$. Otra posibilidad también sería que no se encuentren demasiados SMRs de largos mayores a 10 que generen coberturas.

Si bien en la *Figura 2.14* se puede ver esta tendencia general de valores menores, también se puede notar que la relación entre familiaridades de proteínas de test con su propia familia y con otras (puntos rojos y puntos negros) se mantiene similar, de modo que en principio no obtenemos una mejora sustancial perceptible sobre el uso de la fórmula anterior.

Una observación importante es el hecho de que los valores de familaridad se mantienen en un rango similar al anterior, a pesar de estar acotados de manera diferente. El valor máximo de $familiaridad_{\mathcal{M}}$ era 10 debido a la cota. En el caso de $familiaridad_{\mathcal{E}}$, para cada proteína de test, el máximo valor teórico de familiaridad que puede alcanzar contra cualquier familia es el largo de la propia proteína de test. Sin embargo, este valor no es alcanzado en ninguno de los casos estudiados.

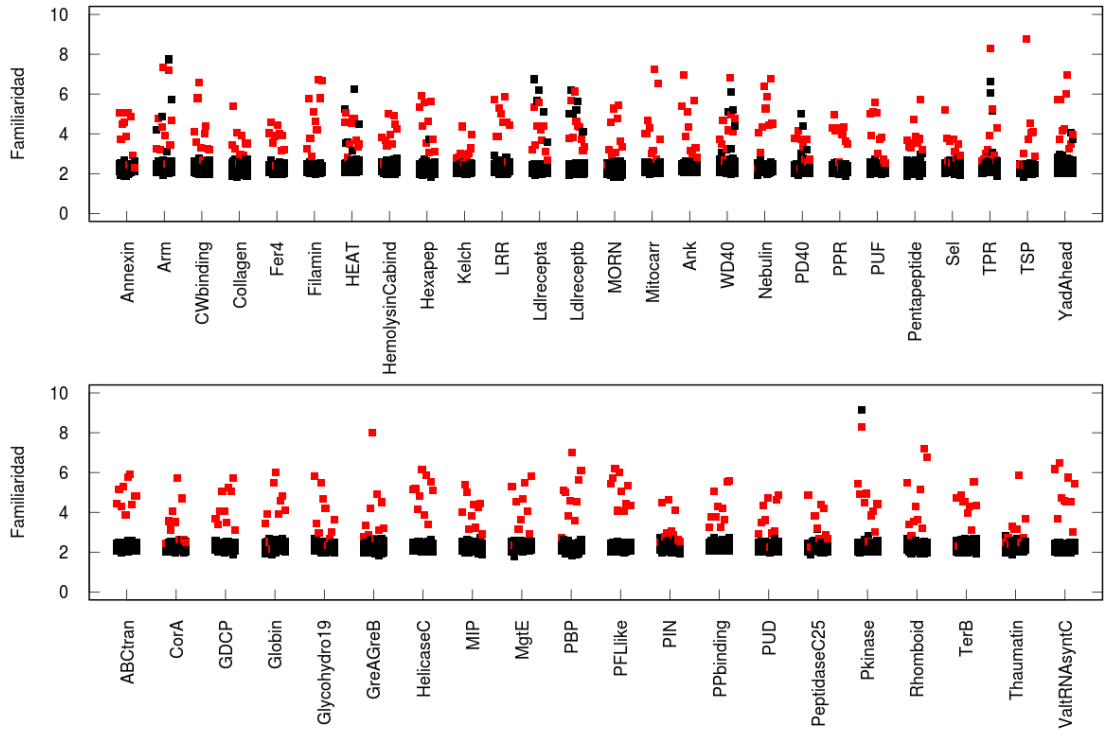


Figura 2.14: Familiaridad usando la fórmula $familiaridad_{\mathcal{E}}$. Puntos rojos representan familiaridades para proteínas de test *vs* su propia familia, puntos negros representan familiaridades para proteínas de test *vs* otras familias.

Cobertura pesada de acuerdo a la cantidad de ocurrencias de los SMR

En esta sección vamos a presentar una variación sobre la fórmula $familiaridad_{\mathcal{E}}$ presentada anteriormente. La idea es seguir sin retringir la familiaridad por una cota ($\kappa = 10$) y seguir calculando la cobertura a partir de los SMR de cada largo exacto pero modificar la manera en la que la calculamos.

La noción de cobertura que venimos manejando consiste en calcular qué proporción de posiciones de la secuencia a cubrir son *cubiertas* por al menos una secuencia del conjunto de cobertura. En particular, cada posición de la secuencia a cubrir tiene solo dos estados posibles: o está cubierta o no lo está.

Para nuestro cálculo de familiaridad los conjuntos de cobertura están formados por los SMR, de un determinado largo, pertenecientes a una familia de proteínas. Recordemos que los SMR van a tener varias repeticiones dentro de la cadena que representa a la familia. Esta cantidad de repeticiones podría variar sustancialmente entre distintos SMR. Podríamos conjeturar que un SMR que tiene más repeticiones que otro dentro de la familia, proveería de un mayor afinidad a una proteína de test. En base a esto surge la idea de otorgar un peso a cada SMR basado en su cantidad de ocurrencias en la familia.

La cantidad de ocurrencias de un SMR es contabilizada como se explica en la sección *Repeticiones y subcadenas únicas*: cada *intervalo* que representa una ocurrencia de un SMR es una repetición no extendible tal que no es una subcadena propia contenida en un intervalo que representa alguna otra ocurrencia de alguna repetición no extendible. Volviendo al ejemplo de esa sección, si contamos con la cadena $w = abaababaabaab$, sus SMR se encuentran en [1..6], [6..11] y [9..13], y corresponden a las cadenas *abaaba*, *abaaba* y *abaab* respectivamente. De esta manera, contamos con 2 SMR: *abaaba* que tiene 2 ocurrencias y *abaab* que tiene 1 ocurrencia. El SMR *abaab* cuenta con una sola ocurrencia pues sus otras apariciones son subcadenas propias contenidas en el SMR *abaaba*.

Una primer intuición para implementar esta idea de pesado por cantidad de ocurrencias consiste en darle un valor a cada posición cubierta equivalente a la cantidad de ocurrencias del SMR que la cubrió. Una posición que se encuentre cubierta por un SMR que en su familia ocurre 152 veces tendría un valor de 152 (en vez de 1 como se venía haciendo con la estrategia anterior). Una posición no cubierta tendría un valor de 0. Luego se sumarían los valores de cada una de las posiciones para continuar con el cálculo de la *cobertura*.

El inconveniente que surge es que ahora la cobertura no se encuentra más acotada en el intervalo [0,1]. Este es uno de los motivos por el cual se necesitan algunos ajustes adicionales para poder llevar esta idea a la práctica. En primer lugar, una posición dada de la secuencia a cubrir puede estar cubierta por más de un SMR, que a su vez podrían tener distintinta cantidad de ocurrencias. Definimos el valor que va a tomar la posición cubierta como la máxima cantidad de ocurrencias entre las cantidades de ocurrencias de los SMR que cubren esa posición.

En segundo lugar, recordemos que luego de contabilizar la cantidad de posiciones que cubren una secuencia, calculamos la cobertura como la sumatoria de dichas posiciones, dividido el largo de la secuencia a cubrir. Esto no tiene el mismo efecto cuando pesamos cada posición cubierta. El propósito original es mantener el valor de la cobertura entre 0 y 1, sin embargo al pesar cada posición y darle un valor potencialmente mayor a 1, la cobertura podría tender a quedar por encima de este rango. Esto es algo no deseado, por un lado, porque va a hacer que sea más complicado acotar los valores de familiaridad y por lo tanto definir un umbral para discriminar entre distintas secuencias. Por otro lado, podríamos obtener mayores valores de cobertura para secuencias más largas. Esto podría

conllevar una cierta desigualdad al momento de calcular la familiaridad. Es por eso que vamos a apuntar a mantener los valores de cobertura en el rango entre 0 y 1 de manera de estandarizar estos valores para cualquier largo de la secuencia a cubrir. A partir de esto la idea es dividir la sumatoria de los valores que toma cada posición por el largo de la secuencia a cubrir, pero ajustado por un cierto factor, de manera que el divisor sea mayor que el máximo valor posible que pueda tomar la sumatoria. Al mismo tiempo se desea acotar ese valor de manera que sea lo menor posible y no usar un factor que sea arbitrariamente grande, simplemente, para asegurarnos de que sea mayor.

La primer propuesta fue tomar como factor de ajuste el máximo valor entre los valores que tomaron cada una de las posiciones. Es decir la máxima cantidad de ocurrencias entre los SMR que cubrieron alguna posición. Esta idea además de mantener el factor de ajuste en una escala similar a la de la sumatoria, tiene sentido desde un punto de vista semántico, pues estamos *pesando* el largo de la secuencia a cubrir con el que dividimos, de la misma manera en que pesamos cada una de las posiciones. Denotaremos a esta alternativa $cobertura_\alpha$. Sin embargo en este punto surge un cuestionamiento: ¿por qué no ajustar el largo de la secuencia a cubrir, en lugar de por la máxima cantidad de ocurrencias entre los SMR que cubrieron alguna posición, por la máxima cantidad de ocurrencias entre *todos* los SMR (todos los SMR que se utilizan para calcular esa cobertura particular)? Después de todo, si la intención es tomar la máxima cantidad de ocurrencias entre los SMR, descartar algunas porque no curbieron a la secuencia a cubrir podría no ser *justo* y terminar desbalanceando el cálculo. A esta alternativa la denotaremos $cobertura_\beta$.

Se decidió implementar ambas alternativas para poder contrastarlas. Los resultados se muestran más adelante. Antes de continuar, incluimos un ejemplo y una definición formal de la nueva cobertura.

Consideremos la secuencia a cubrir s = abcabbaabcccbab y el multiconjunto de SMR, T = {abc, abc, abc, abc, cab, cab, bbb, bbb, bbb, bbb, bbb, bbb}. Es decir que, para cada elemento, contamos con las siguientes cantidades de ocurrencias:

$$\#\mathrm{T}(\mathrm{abc}) = 4$$
$$\#\mathrm{T}(\mathrm{cab}) = 2$$
$$\#\mathrm{T}(\mathrm{bbb}) = 6$$

Podemos ver que cada elemento de T cubre las siguientes posiciones:

```
  s:    abcabbaabcccbab
abc:    abc----abc-----
cab:    --cab----------
bbb:    ---------------
```

Tomando la cantidad de ocurrencias de cada $t \in T$, podemos ver el valor que se asigna a cada posición según el SMR que la afecta y la cantidad de instancias del mismo en el multiconjunto T:

```
  s:    abcabbaabcccbab
abc:    444000044400000
cab:    002220000000000
bbb:    000000000000000
```

Considerando la mayor cantidad de ocurrencias para cada posición se obtiene:

```
s:    abcabbaabcccbab
      444220044400000
```

Observemos que la tercer posición de s, que contiene el caracter c, toma el valor 4 pues tanto abc ∈ T como cab ∈ T la cubren, pero la mayor cantidad de ocurrencias es la de abc que es 4.

Considerando la alternativa $cobertura_\alpha$ la cobertura sería:

$(4+4+4+2+2+4+4+4)/(15*4) = 28/60 = 0{,}466666$

Si consideráramos la alternativa $cobertura_\beta$:

$(4+4+4+2+2+4+4+4)/(15*6) = 28/90 = 0{,}311111$

Notemos que 15 corresponde a la longitud de s y 4 es la máxima cantidad de ocurrencias entre los elementos de T que cubrieron alguna posición de s (correspondiente a abc), pero 6 es la máxima cantidad de ocurrencias entre todos los elementos de T (correspondiente a bbb).

Definimos formalmente $cobertura_\alpha$, para una secuencia s y un multiconjunto de secuencias T como:

$$cobertura_\alpha(s,T) = \frac{\sum_{i=1}^{|s|} nivel(i,s,T)}{max_{1\leq i\leq |s|}(nivel(i,s,T))}$$

donde definimos:

$$nivel(i,s,T) = \begin{cases} max(\#T(t)) : t \in cobertores(i,s,T) & \text{si } cobertores(i,s,T) \neq \emptyset \\ 0 & \text{si } cobertores(i,s,T) = \emptyset \end{cases}$$

$$cobertores(i,s,T) = \{t \in T : \exists j \in \mathbb{N} \wedge j \leq i \leq j + |t| - 1 \wedge s[j..j+|t|-1] = t\}$$

Por otro lado $cobertura_\beta$ quedará definida como:

$$cobertura_\beta(s,T) = \frac{\sum_{i=1}^{|s|} nivel(i,s,T)}{max_{t\in T}(\#T(t))}$$

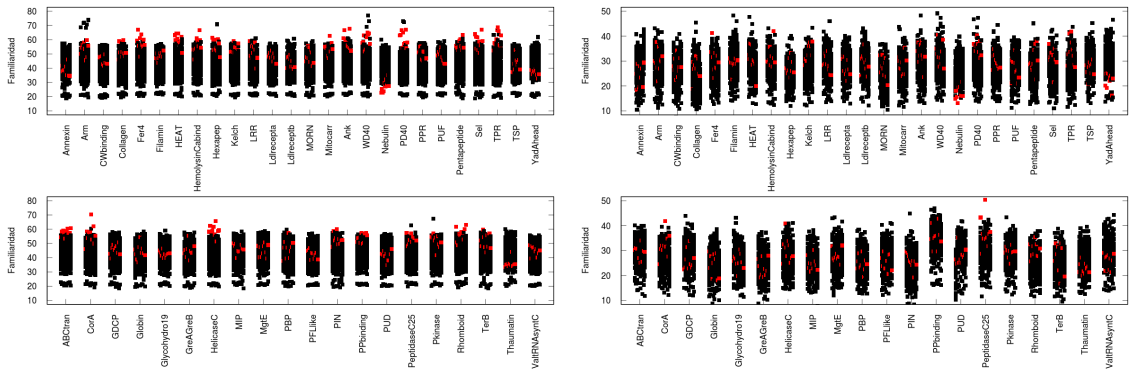
Dada una cadena w, y un $j \in \mathbb{N}$, definimos el multiconjunto $\mathcal{E}_\mu(w,j)$ como el multiconjunto de todos los SMR sobre la cadena w, de largo exactamente igual a j. Es decir, cada SMR tiene en el multiconjunto tantas ocurrencias como tiene w. Luego, las nuevas alternativas de familiaridad quedan definidas de la siguiente manera:

$$familiaridad_{\mathcal{E}\mu\alpha}(s,t) = \frac{cobertura_\alpha(s,\mathcal{E}_\mu(t,0)) + cobertura_\alpha(s,\mathcal{E}_\mu(t,|s|))}{2} + \sum_{i=1}^{|s|-1} cobertura_\alpha(s,\mathcal{E}_\mu(t,i))$$

$$familiaridad_{\mathcal{E}\mu\beta}(s,t) = \frac{cobertura_\beta(s,\mathcal{E}_\mu(t,0)) + cobertura_\beta(s,\mathcal{E}_\mu(t,|s|))}{2} + \sum_{i=1}^{|s|-1} cobertura_\beta(s,\mathcal{E}_\mu(t,i))$$

A continuación, en la *Figura 2.15* podemos ver los resultados de calcular la familiaridad para nuestro dataset mediante las nuevas fórmulas. Lamentablemente, más allá del potencial de la idea los resultados no fueron alentadores. Como se puede ver, los valores de familiaridad para una proteína de test contra su misma familia y contra otras son demasiado similares. Esto queda representado en el gráfico con puntos rojos y puntos negros muy juntos. Lo cual hace que los valores obtenidos no sean buenos para intentar estimar si una proteína de test dada pertenece a una cierta familia o no. Se puede observar que, con la fórmula $familiaridad_{\mathcal{E}\mu\alpha}$, se obtuvieron valores levemente mejores que con $familiaridad_{\mathcal{E}\mu\beta}$. Aunque siguen siendo peores que los de las alternativas anteriores: la *familiaridad* propuesta por Turjanski et al. [6] y $familiaridad_{\mathcal{E}}$, propuesta anteriormente en este trabajo.

Se puede notar en la *Figura 2.15a* que para algunas familias de proteínas particulares se dá que los valores de familiaridad para proteínas de test de esa misma familia son levemente más altos que los valores de familiaridad para proteínas de test de otras familias. Esto se ve reflejado en el gráfico como puntos rojos, en su mayoría por encima de los puntos negros. Si bien estos casos son una minoría, se puede notar una tendencia a que se den en las familias más grandes, es decir, las que en su concatenación de proteínas cuentan con más cantidad de aminoácidos. Mientras que para varias de las familias más pequeñas dentro del dataset se puede notar lo opuesto: los valores de familiaridad para las proteínas de test de la misma familia dan particularmente bajos. Esto nos llevó a especular que se podría estar produciendo una desigualdad debido al pesado de la cobertura por cantidad de ocurrencias que haría que las proteínas de test pertenecientes a familias más grandes obtengan mayores valores de familiaridad, sin importar si pertenecen a la familia. Esta idea se basa en que se espera que los SMR generados en familias más grandes tengan en general bastantes más ocurrencias que en familias chicas. Queda como trabajo futuro intentar definir cómo escala la cantidad de ocurrencias de SMR respecto al tamaño de las familias para así poder ajustar el pesado de acuerdo a la cantidad de ocurrencias por un cierto factor relativo al tamaño de la familia.



(a) Resultados usando $familiaridad_{\mathcal{E}\mu\alpha}$ *(b)* Resultados usando $familiaridad_{\mathcal{E}\mu\beta}$

Figura 2.15: Comparación de valores de familiaridad usando $familiaridad_{\mathcal{E}\mu\alpha}$ y $familiaridad_{\mathcal{E}\mu\beta}$. Puntos rojos representan familiaridades para proteínas de test *vs* su propia familia, puntos negros representan familiaridades para proteínas de test *vs* otras familias.

Cobertura pesada de acuerdo al tamaño de los SMR

Mas allá de que los resultados de la sección anterior no fueron satisfactorios, nos llevaron a retomar la idea de pesar a cada SMR de acuerdo a su largo. Además de tratar de generar un balance entre los valores obtenidos a partir de familias muy dispares en cuanto a su tamaño, tenemos un motivo adicional. Este motivo tiene que ver con la especulación de que ciertos SMR de tamaños más cortos podrían tener chances arbitrariamente altas de encontrarse en una proteína de test más allá de si el SMR fue generado a partir de la propia familia de la proteína o no. Si logramos darle mayor peso a las coberturas generadas con SMR más largos estaríamos de alguna manera súperando esta situación.

A partir de esta idea vamos a definir la siguiente alternativa a la fórmula de familiaridad:

$$familiaridad_{fg}(s,t) = g(\sum_{i=1}^{|s|-1}(cobertura(s,\mathcal{E}(t,i))*f(i))) + \frac{cobertura(s,\mathcal{E}(t,0))+cobertura(s,\mathcal{E}(t,|s|))}{2}$$

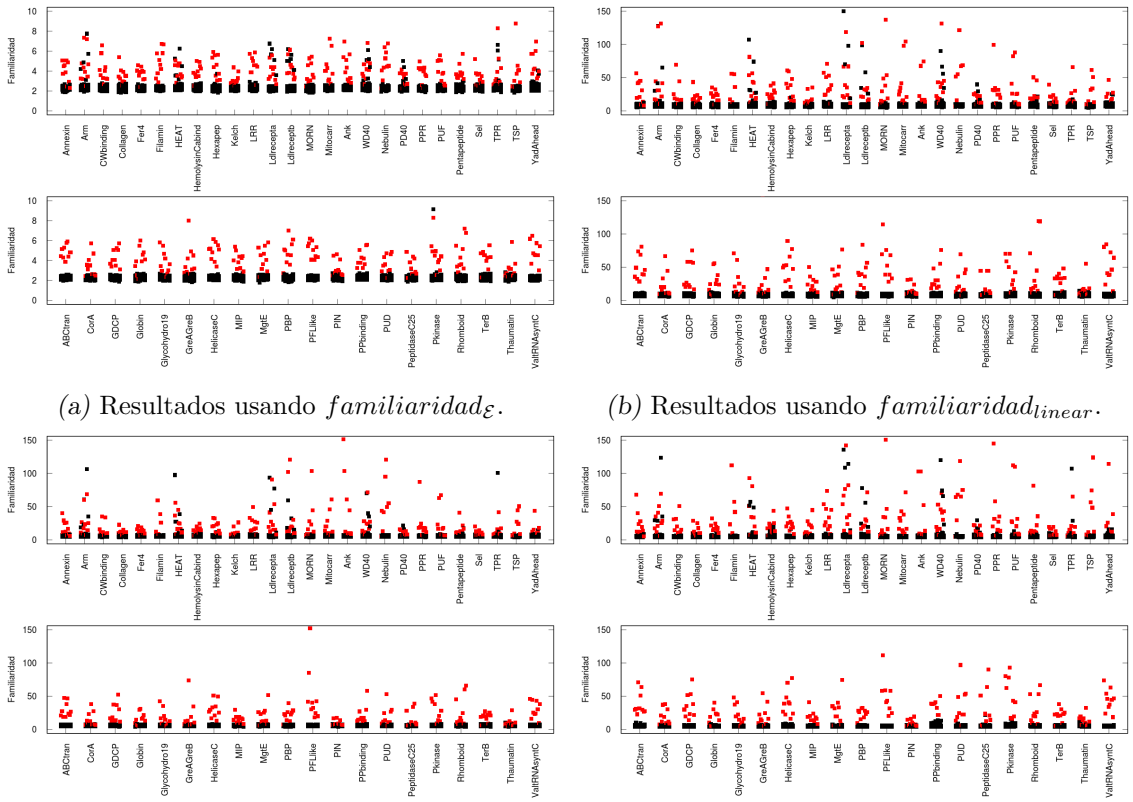
Esto nos va a permitir pesar la cobertura generada a partir de los SMR de cada largo dado por una función f sobre el mismo. La idea es usar siempre funciones estrictamente crecientes de manera de asegurarnos de otorgar mayor peso a las coberturas generadas por SMR de mayor tamaño. La intención de la función g es reducir el resultado final de la sumatoria que puede haberse hecho arbitrariamente grande dependiendo de la función f usada; al mismo tiempo, para que tenga sentido, deberá guardar una cierta relación con f. La idea intuitiva que aplicaremos es que g sea la inversa de f para que los valores se vean reducidos en el mismo orden de magnitud que fueron incrementados. De esta manera definimos las siguientes fórmulas:

$$familiaridad_{linear}(s,t) = \sum_{i=1}^{|s|-1}(cobertura(s,\mathcal{E}(t,i))*i) + \frac{cobertura(s,\mathcal{E}(t,0))+cobertura(s,\mathcal{E}(t,|s|))}{2}$$

$$familiaridad_{quad}(s,t) = \sqrt{\sum_{i=1}^{|s|-1}(cobertura(s,\mathcal{E}(t,i)) * i^2)} + \frac{cobertura(s,\mathcal{E}(t,0))+cobertura(s,\mathcal{E}(t,|s|))}{2}$$

$$familiaridad_{exp}(s,t) = \log_{10}(\sum_{i=1}^{|s|-1}(cobertura(s,\mathcal{E}(t,i)) * 10^i)) + \frac{cobertura(s,\mathcal{E}(t,0))+cobertura(s,\mathcal{E}(t,|s|))}{2}$$

En la *Figura 2.16* vemos una comparación de los valores de familiaridad obtenidos utilizando las distintas fórmulas. En la *Figura 2.16a* agregamos, a modo de referencia, los resultados utilizando $familiaridad_{\mathcal{E}}$, ya discutidos previamente. Para poder representar una mayor cantidad de valores sin distorsionar la escala, en las *Figuras 2.16b, 2.16c y 2.16d*, truncamos los valores a un máximo de 150. Si bien el objetivo de la función g, es mantener los valores en una cierta escala, para ciertos casos estos resultaron demasiado grandes. Esta es una desventaja a destacar sobre el uso de estas fórmulas: encontrar una cota práctica para los valores de familiaridad se vuelve todavía más complejo. Más allá de esto, podemos observar que, en general, dada una familia, la diferencia entre valores de familiaridad contra proteínas de test de esa misma familia (rojo), y de otras (negro) se agranda. Para analizar mejor esto generamos otros gráficos.



(a) Resultados usando $familiaridad_{\mathcal{E}}$.

(b) Resultados usando $familiaridad_{linear}$.

(c) Resultados usando $familiaridad_{quad}$.

(d) Resultados usando $familiaridad_{exp}$.

Figura 2.16: Comparación de valores de familiaridad usando distintas funciones. Puntos rojos representan familiaridades para proteínas de test *vs* su propia familia, puntos negros representan familiaridades para proteínas de test *vs* otras familias. Los valores del eje y han sido acotados en un máximo de 150 en las figuras b, c y d

En las *Figuras 2.17, 2.18, 2.19, 2.20*, representamos, para cada familia, el promedio y desvío estándar de los valores de familiaridad para proteínas de test; las de esa misma famila en rojo y las de otras familias en negro; utilizando las fórmulas $familiaridad_{\mathcal{E}}$, $familiaridad_{linear}$, $familiaridad_{quad}$ y $familiaridad_{exp}$ respectivamente. Si bien el promedio no es una medida definitiva de la distribución de los datos, nos sirve para poder representar una idea de los valores esperados. De este modo, podemos ver fácilmente la diferencia entre el promedio de valores de proteínas de test de esa misma familia y de otras. Notemos que el desvío estándar para proteínas de test de la misma familia (rojo) es siempre mayor que para proteínas de test de otras familias (negro). Podemos ver que, cuanto más rápido crece la función f utilizada, mayor es el desvío estándar. De hecho, algunos valores crecen tanto que no pueden ser representados y por eso no figuran en los gráficos. En la implementación se utilizaron números de punto flotante de doble precisión.

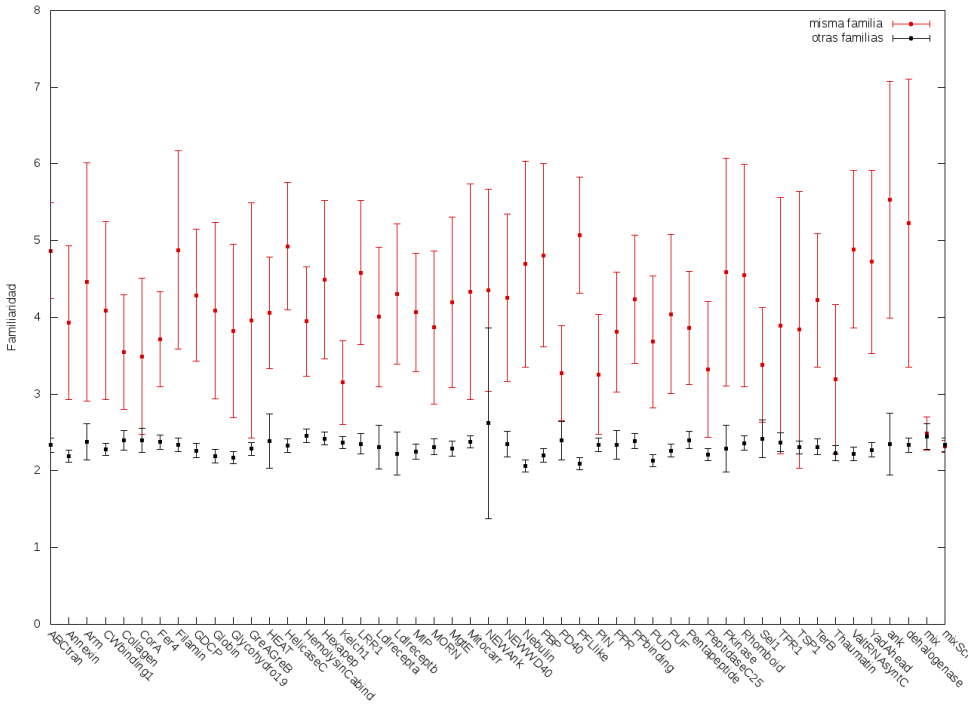


Figura 2.17: Promedio y desvío estándar sobre resultados de la *Figura 2.16a* ($familiaridad_{\mathcal{E}}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

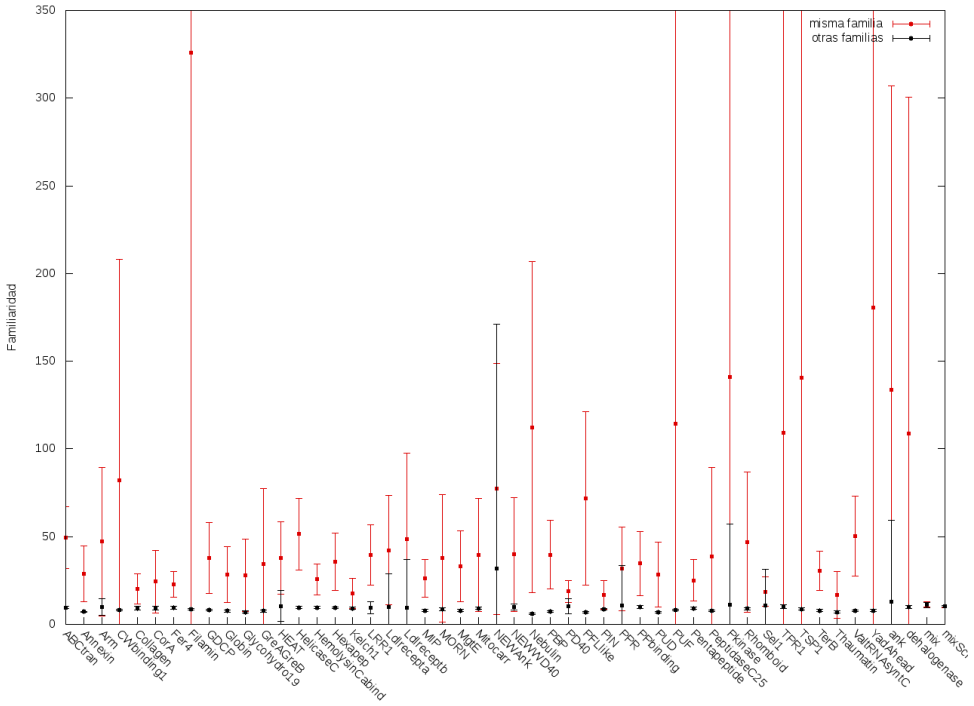


Figura 2.18: Promedio y desvío estándar sobre resultados *Figura 2.16b* ($familiaridad_{linear}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

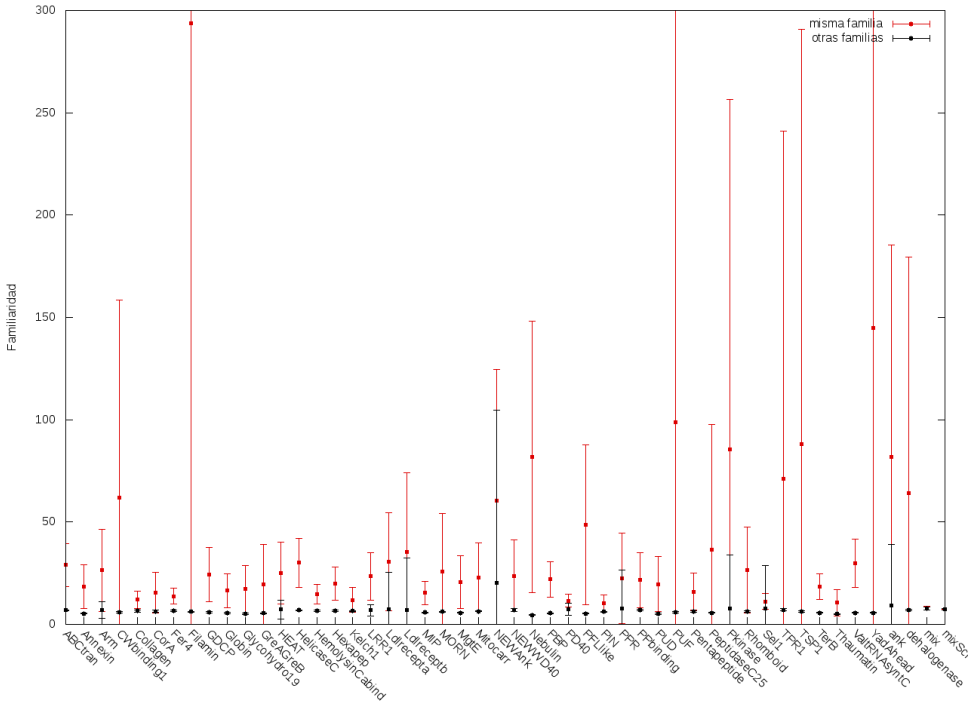


Figura 2.19: Promedio y desvío estándar sobre resultados *Figura 2.16c* ($familiaridad_{quad}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

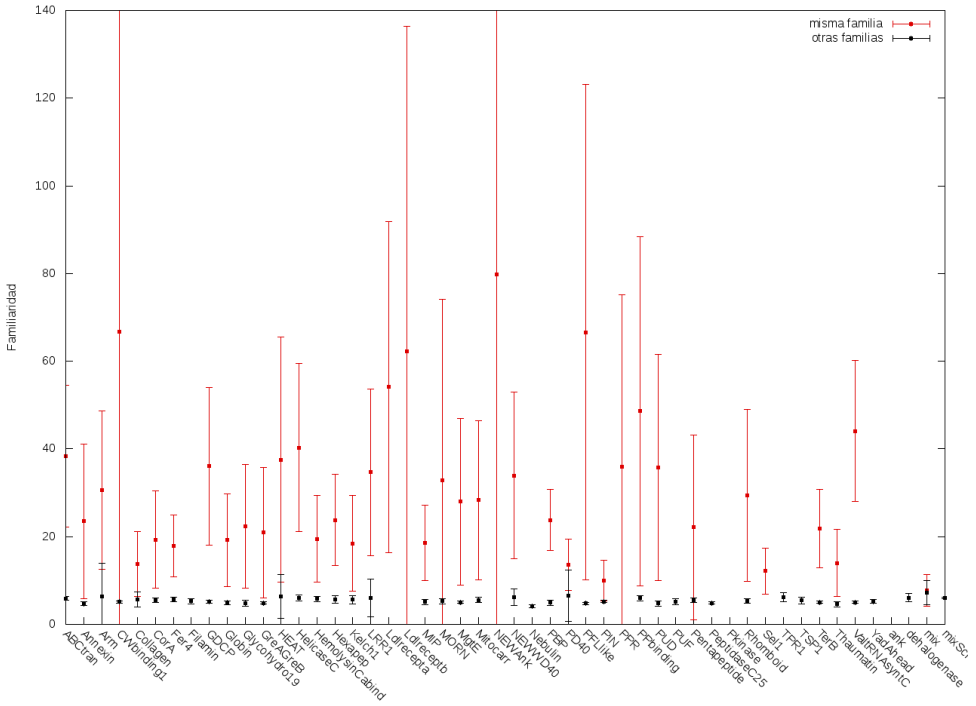


Figura 2.20: Promedio y desvío estándar sobre resultados *Figura 2.16d* ($familiaridad_{exp}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

Por último intentamos representar comparativamente una aproximación de las diferencias entre estas cuatro alternativas. En la *Figura 2.21* se muestra, para cada familia y para cada fórmula de $familiaridad$, la proporción entre el promedio de valores de familiaridad para proteínas de test de otras familias y el promedio de valores de familiaridad para proteínas de test de esa misma familia. En otras palabras, dada un familia, llamaremos $\mu_=$ al promedio de los valores de familiaridad entre todas las proteínas de test de esa familia y $\mu_{\neq}$ al promedio de valores de familiaridad entre todas las proteínas de test correspondientes a otras familias. Luego definimos $P = \frac{\mu_{\neq}}{\mu_=}$. Si bien esta proporción entre promedios no es más que una aproximación de la precisión, nos sirve para obtener una intuición de cuán precisa es una determinada fórmula y poder compararla con otras. Utilizamos la proporción entre promedios y no la diferencia entre promedios, para salvar las dificultades que traería tratar de comparar valores que se encuentran en escalas diferentes, entre distintas fórmulas.

Si por ejemplo contamos con un $P = 0,10$ esto quiere decir que $\mu_{\neq}$ es 10 veces menor que $\mu_=$. Con lo cual, si contamos con un desvío estándard entre valores de familiaridad suficientemente bajo, dado un valor de familiaridad cualquiera debería resultarnos fácil decidir si la proteína de test que lo generó pertenecía a la familia o no. Esto debido a que los valores que esperamos encontrar para familiaridad de proteínas de test de otras familias estarán alrededor de $\mu_{\neq}$, mientras que los de la misma familia alrededor de $\mu_=$, el cual será 10 veces mayor y consecuentemente facilitará en gran manera la disitinción entre ambos grupos. Si por el contrario contáramos con un P cercano a 1 sería muy difícil distinguir cuando una proteína de test pertenece o no a una familia puesto que $\mu_{\neq}$ y $\mu_=$ serían muy similares.

De este modo, podemos ver que la idea de usar funciones crecientes para pesar los

valores de cobertura de acuerdo al tamaño de los SMR conllevó una considerable mejora en la precisión de la fórmula. Sin embargo, como ya mencionamos los valores llegan a volverse arbitrariamente altos y son difíciles de acotar en la práctica, lo cual es una gran desventaja. En la siguiente sección, partiendo de esta misma idea, tomaremos otro enfoque para tratar de llegar a resultados más satisfactorios en este aspecto.

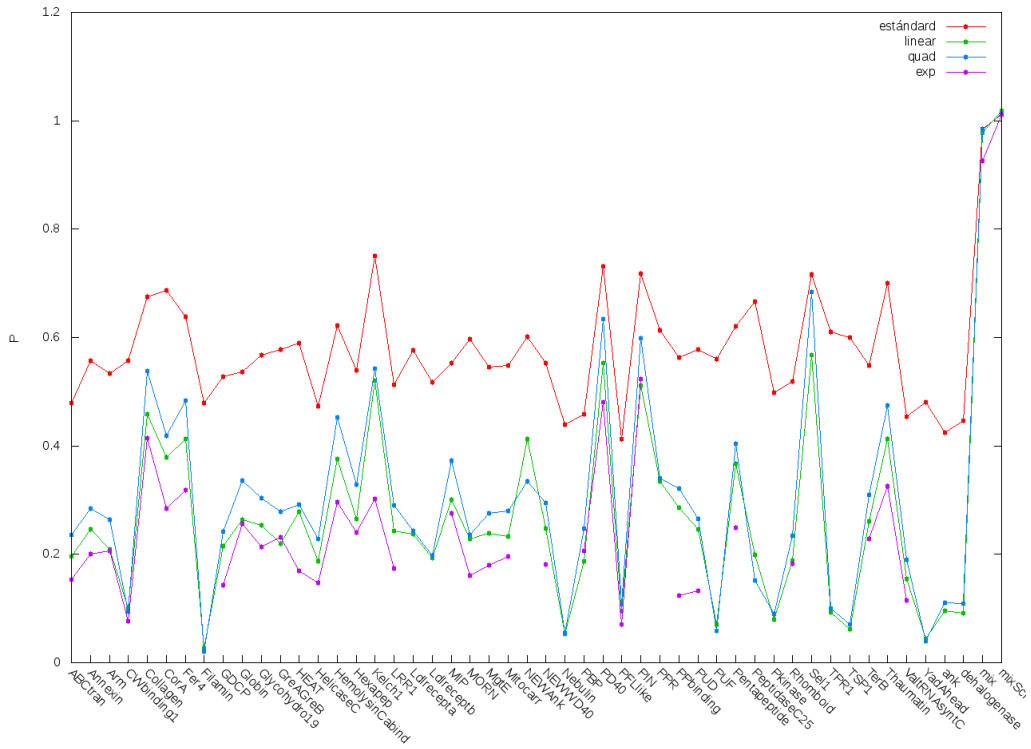


Figura 2.21: Precisión expresada como $P = \frac{\mu_{\neq}}{\mu_{=}}$ para $familiaridad_{fg}$ para distintas funciones f y g de pesado de cobertura. Ciertos valores para $familiaridad_{exp}$ no se encuentran representados, pues las correspondientes familiaridades no pueden ser representadas con la implementación de números de punto flotante de doble precisión.

Eliminación de valores de cobertura para las menores longitudes de repeticiones

Una variación sobre la idea de pesar los SMR de acuerdo a su longitud fue, en lugar de aplicar una función que varíe el peso de cada cobertura, utilizar un predicado para decidir si utilizar esa cobertura o no en el cálculo de familiaridad. Lo que hicimos fue tomar los valores de cobertura calculados sobre SMR de largo mayor o igual a un n dado. De esta manera definimos la familiaridad con un desplazamiento de n como:

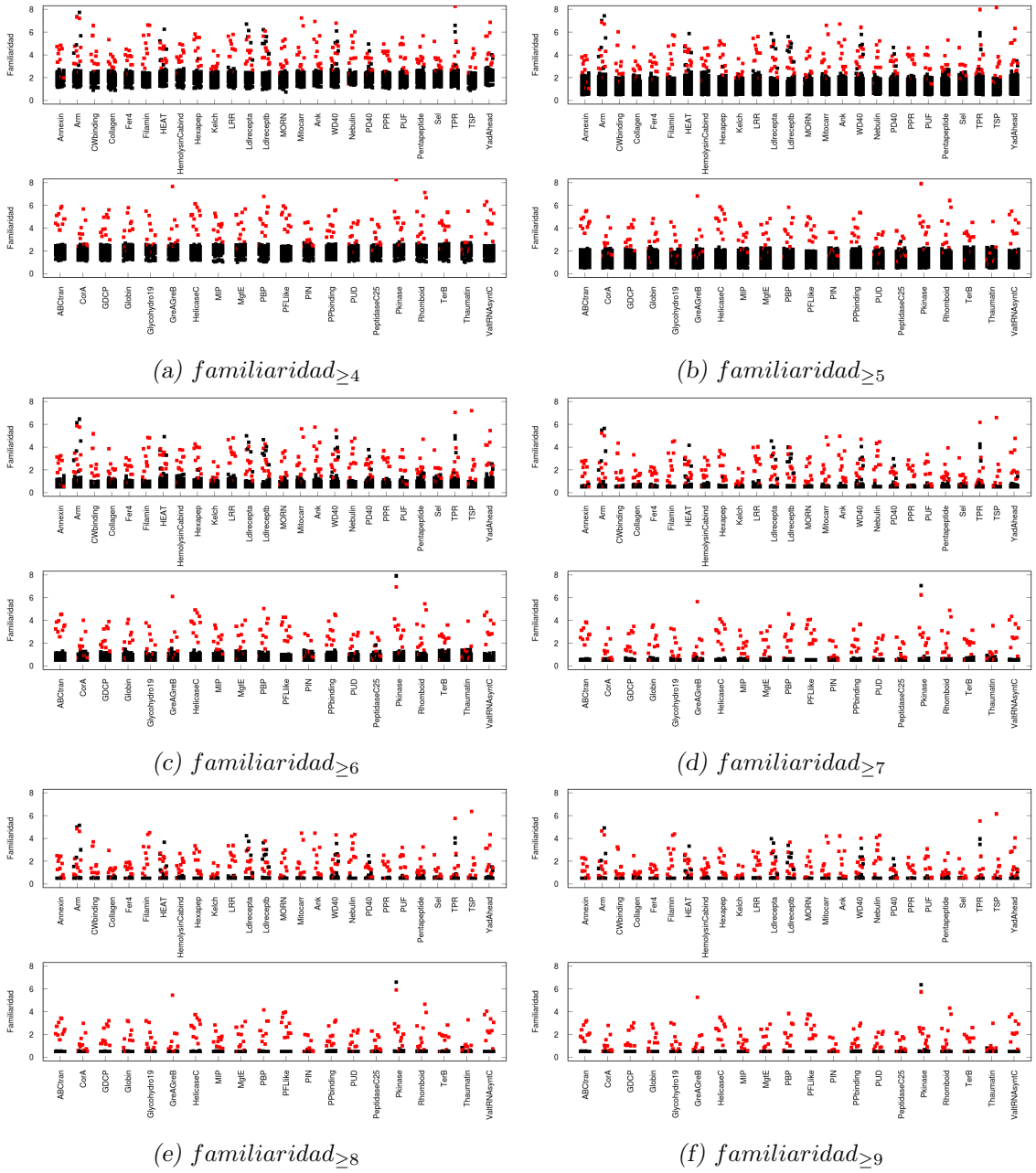
$$familiaridad_{\geq n}(s,t) = \sum_{i=n}^{|s|-1}(cobertura(s,\mathcal{E}(t,i))) + \frac{cobertura(s,\mathcal{E}(t,0))+cobertura(s,\mathcal{E}(t,|s|))}{2}$$

La intención detrás de esta nueva forma de cálculo es contemplar la posibilidad de que, si bien distintas coberturas para distintos largos de SMR tienen distintas importancias en la constitución de la familiaridad, una función estrictamente creciente como las presentadas anteriormente no se adapta adecuadamente a modelar esta variación. Con esta nueva forma de cálculo representamos una variación brusca y repentina en la importancia de los valores de cobertura. De este modo si encontramos un n tal que podamos detectar que la cobertura para SMR de largo $n-1$ no aporta información significativa al cálculo, mientras que la cobertura para SMR de largo n sí, vamos a obtener una mayor precisión en los valores de familiaridad obtenidos.

Esta idea surge de la especulación de que para ciertos tamaños de cadenas chicos, dado el tamaño de nuestro alfabeto (20) y la gran cantidad de aminoácidos que se cuentan en total entre todas las proteínas de una familia (incluso para las familias más chicas), la probabilidad de encontrar SMR de estos tamaños sea muy grande. Para tamaños suficientemente chicos la probabilidad podría llegar a ser tan grande que genere que se encuentre prácticamente toda, o una gran parte, de la combinatoria de posibles cadenas de ese tamaño, dado el alfabeto. Esto haría que cualquier cadena, en particular cualquier proteína de test, quede casi completamente cubierta por cadenas de este tamaño. De ser así esto no aportaría información significativa pues cualquier proteína de test, perteneciente a la familia o no, se comportaría por igual. Por otro lado el tamaño del alfabeto es suficientemente grande para pensar que las cadenas del tamaño inmediatamente súperior podrían comportarse en una forma muy diferente. Observemos que dado un n, la diferencia entre la cantidad de cadenas posibles de ese largo y el siguiente en un alfabeto de 20 caracteres es $20^{n+1} - 20^n$.

A continuación en la *Figura 2.22* presentamos los resultados de calcular la familiaridad con esta fórmula para valores de n de 4 a 9. Como se puede observar, la variación en los resultados entre $familiaridad_{\geq 4}$ y $familiaridad_{\geq 5}$ no es grande. Entre $familiaridad_{\geq 5}$ y $familiaridad_{\geq 6}$ se distingue que los valores de familiaridades para proteínas de test de otras familias (puntos negros) tienden a estar concentrados en rangos levemente más chicos. Esto se ve en forma de puntos negros más juntos entre sí, en cada familia. A su vez la distancia entre puntos negros y rojos, en su conjunto, aumenta. Es entre $familiaridad_{\geq 6}$ y $familiaridad_{\geq 7}$ que encontramos una destacable diferencia. Podemos observar que los puntos negros se encuentran concentrados en rangos considerablemente más chicos en $familiaridad_{\geq 7}$. Además, en general, la distancia entre puntos rojos y negros también parece aumentar respecto a $familiaridad_{\geq 6}$. A partir de aquí, se vuelve difícil distinguir variaciones en los resultados. Aunque entre $familiaridad_{\geq 8}$ y $familiaridad_{\geq 9}$ podemos ver una leve disminución en la diferencia entre puntos rojos y negros, que es algo no desea-

do. Para representar de forma más precisa estas variaciones utilizaremos otros gráficos.



(a) $familiaridad_{\geq 4}$ (b) $familiaridad_{\geq 5}$

(c) $familiaridad_{\geq 6}$ (d) $familiaridad_{\geq 7}$

(e) $familiaridad_{\geq 8}$ (f) $familiaridad_{\geq 9}$

Figura 2.22: Comparación de valores de $familiaridad_{\geq n}$ usando distintos valores de n. Puntos rojos representan familiaridades para proteínas de test *vs* su propia familia, puntos negros representan familiaridades para proteínas de test *vs* otras familias.

En las *Figuras 2.23 a 2.28* representamos promedio y desvío estándar de valores de familiaridad para cada familia -como se explicó en la sección anterior- pero esta vez para $familiaridad_{\geq n}$ con $n = 4, 5, 6, 7, 8, 9$. Aquí se observa cómo, en la *Figura 2.26*, para $familiaridad_{\geq 7}$, los promedios para valores de familiaridad de proteínas de test de otras familias (en negro) se alínean tomando valores muy similares a través de distintas familias. Este es un efecto deseado. A su vez los, desvíos estándar se reducen notablemente respecto

de los representados en las *Figuras 2.23, 2.24, 2.25* (correspondientes a $familiaridad_{\geq 4}$, $familiaridad_{\geq 5}$, $familiaridad_{\geq 6}$ respectivamente). En la *Figura 2.27* podemos ver que los valores que toma $familiaridad_{\geq 8}$ son muy similares a los de $familiaridad_{\geq 7}$. En la *Figura 2.28* se puede notar, para $familiaridad_{\geq 9}$, una leve disminución en la diferencia entre los valores promedio de familiaridad para proteínas de la misma familia y de otras familias, lo cual va en contra de lo deseado.

Para poder comparar la precisión de las distintas fórmulas, en la *Figura 2.29*, representamos la proporción entre promedios de valores de familiaridad P, como se explicó en la seccion anterior. Recordemos que lo deseado es obtener valores lo más cercanos posibles a cero. Podemos ver cómo, para la mayoría de las familias, P es menor al utilizar $familiaridad_{\geq 6}$ y $familiaridad_{\geq 7}$. Al analizar estos datos vemos que la precisión de $familiaridad_{\geq 6}$ es muy buena, algo que no era del todo visible al observar la *Figuras 2.22c y 2.25*. Por otro lado, como se vió en la *Figura 2.26*, al utilizar la fórmula $familiaridad_{\geq 7}$, los valores de familiaridad para proteínas de test de otras familias quedan restringidos a intervalos convenientemente pequeños.

Utilizar estas fórmulas, $familiaridad_{\geq n}$, genera una gran ventaja sobre las fórmulas $familiaridad_{fg}$ definidas anteriormente, ya que los valores de familiaridad son considerablemente más fáciles de acotar. Además, el hecho de no utilizar SMR de tamaño menor a 6 o 7 podría potencialmente hacer la implementación del algoritmo más eficiente, ya que no sería neceserio buscar cómo esos SMR cubren una proteína de test. Recordemos que los SMR de tamaños pequeños son muchos. Si bien las proporciones entre promedios al usar $familiaridad_{fg}$, en los mejores casos, son levemente mejores que al usar $familiaridad_{\geq 6}$ o $familiaridad_{\geq 7}$, se obtienen desvíos estándar muy altos entre los valores, lo cual las hace menos preferibles.

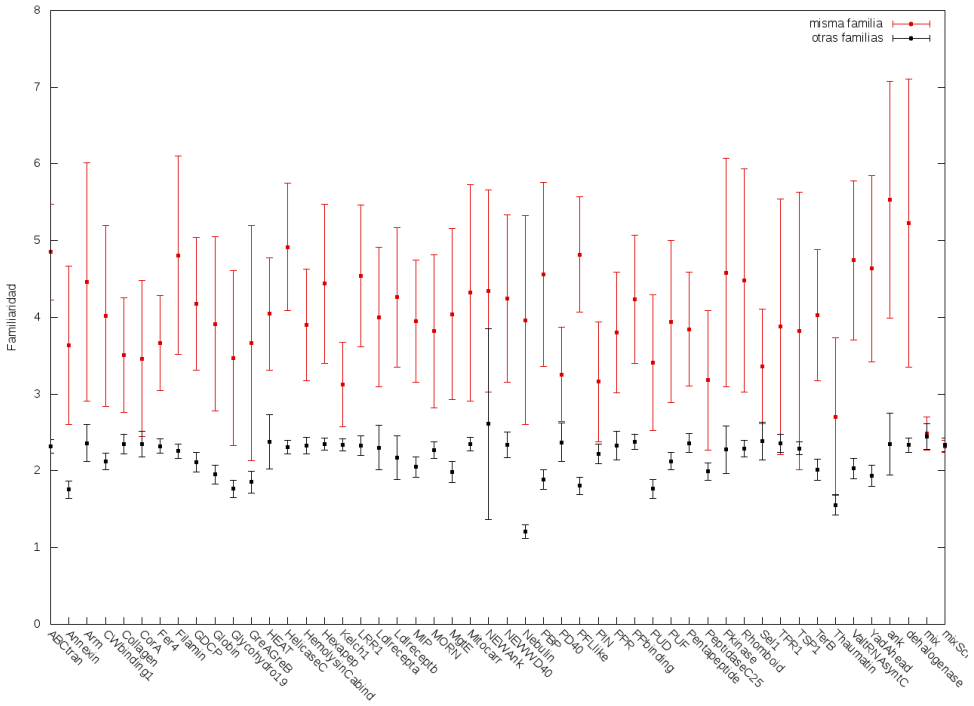


Figura 2.23: Promedio y desvío estándar sobre resultados de la *Figura 2.22a* ($familiaridad_{\geq 4}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

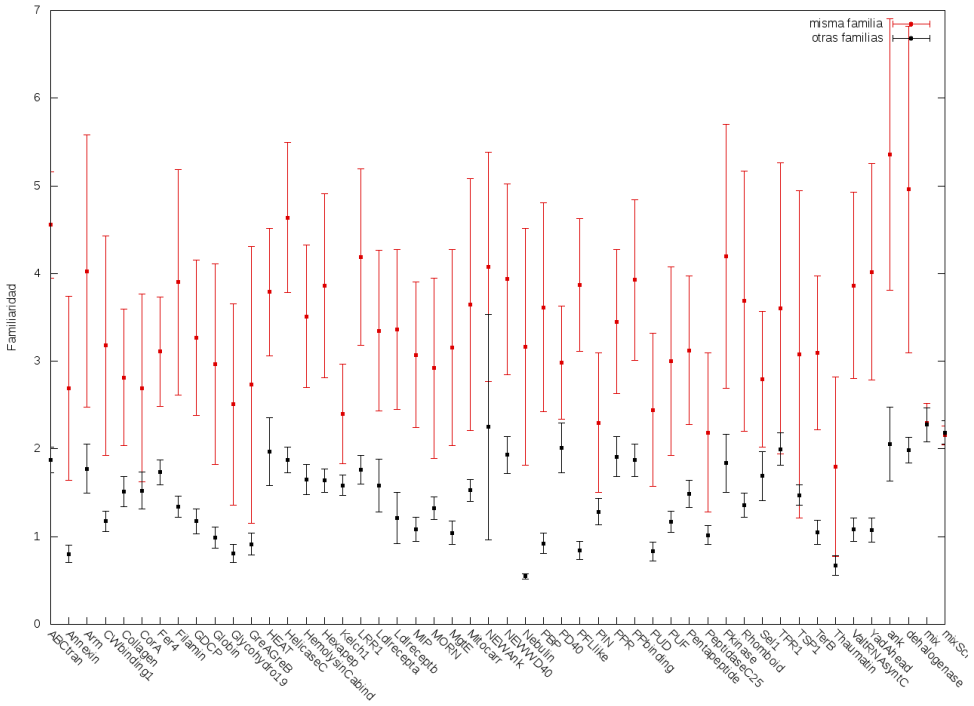


Figura 2.24: Promedio y desvío estándar sobre resultados de la *Figura 2.22b* ($familiaridad_{\geq 5}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

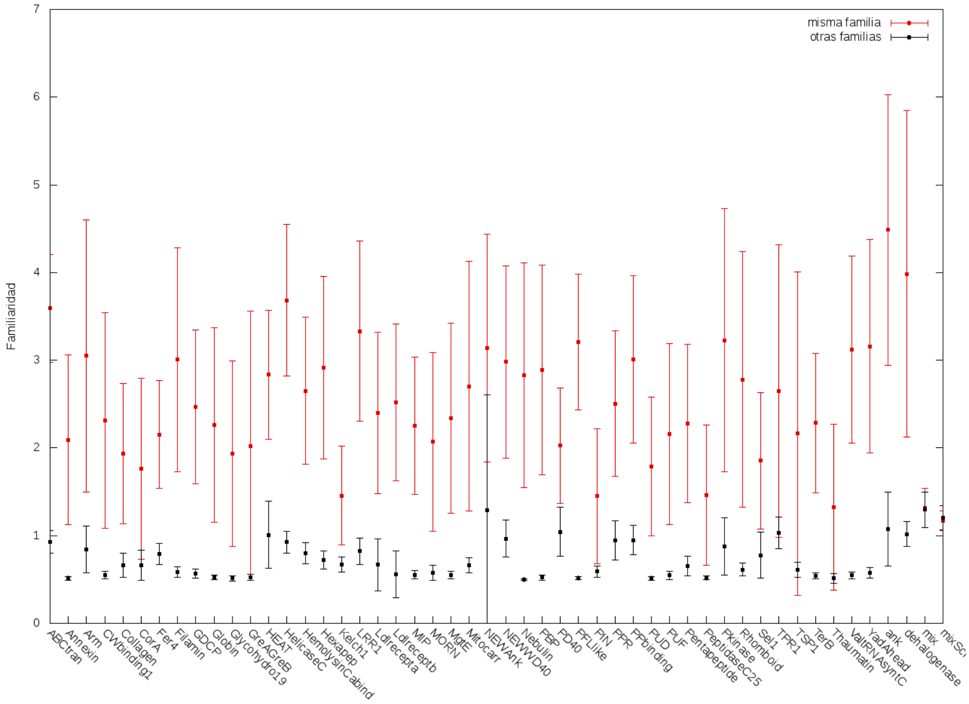


Figura 2.25: Promedio y desvío estándar sobre resultados de la *Figura 2.22c* ($familiaridad_{\geq 6}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

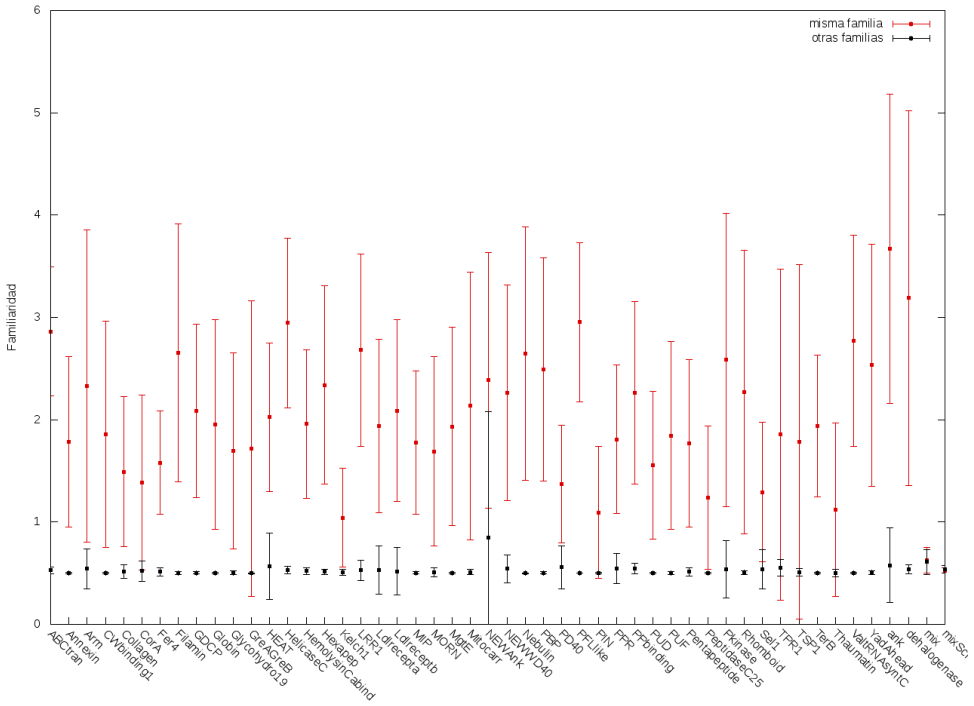


Figura 2.26: Promedio y desvío estándar sobre resultados de la *Figura 2.22d* ($familiaridad_{\geq 7}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

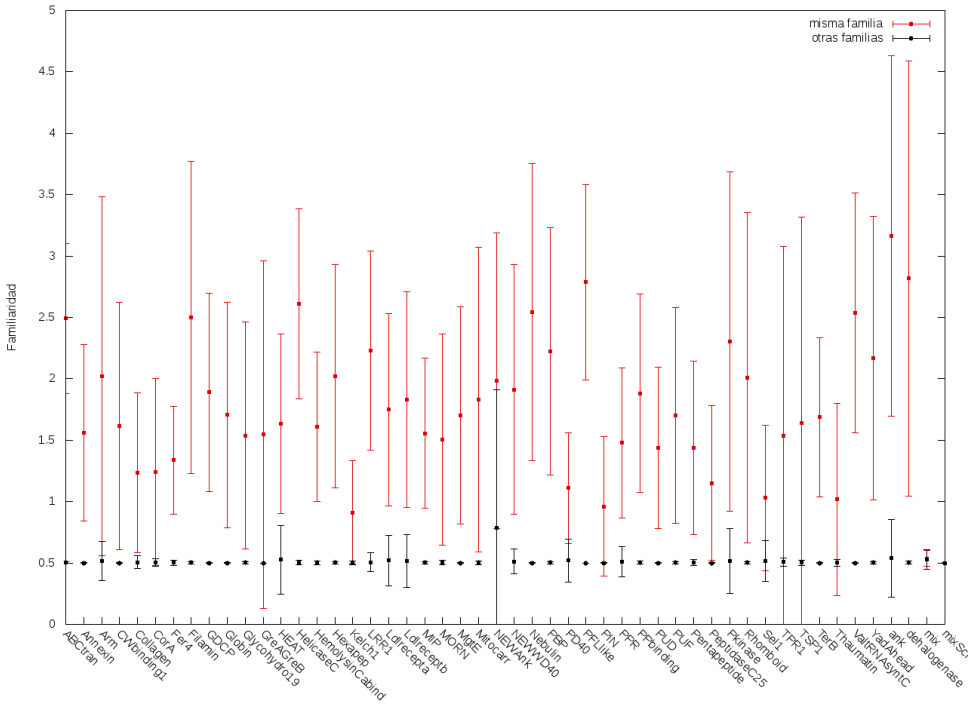


Figura 2.27: Promedio y desvío estándar sobre resultados de la *Figura 2.22e* ($familiaridad_{\geq 8}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

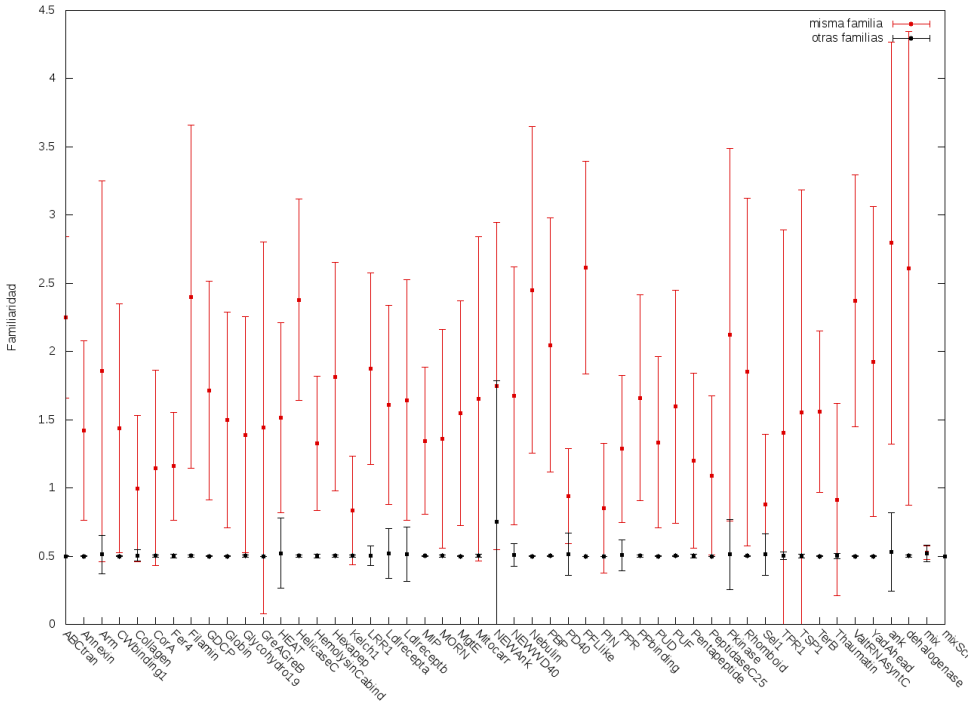


Figura 2.28: Promedio y desvío estándar sobre resultados de la *Figura 2.22f* ($familiaridad_{\geq 9}$). Rojo para proteínas de test de la misma familia, negro para proteínas de test de otras familias.

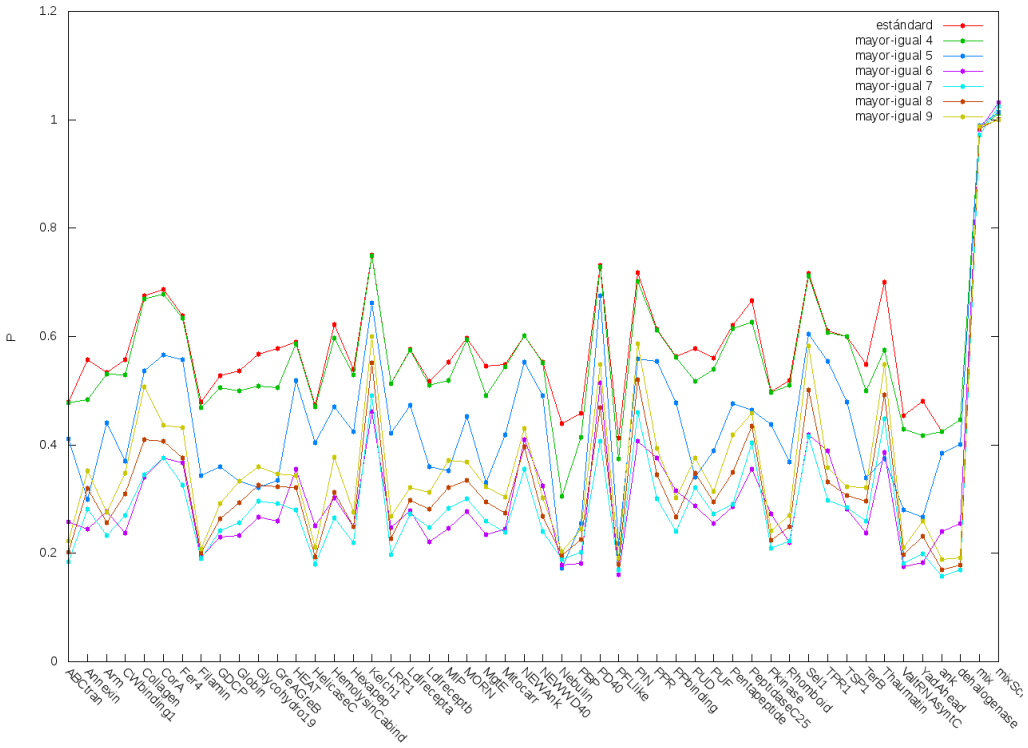


Figura 2.29: Precisión expresada como $P = \frac{\mu_{\neq}}{\mu_{=}}$ para $familiaridad_{\geq n}$ para distintos valores de n. Recordemos que llamamos *"estándard"* a la $familiaridad_{\mathcal{E}}$ la cual también podríamos considerar como $familiaridad_{\geq 0}$.

3. CONCLUSIONES Y TRABAJO FUTURO

A lo largo de este trabajo expusimos distintas técnicas para la clasificación e identificación de proteínas. Ambos propósitos pueden tener utilidades muy distintas en la práctica, sin embargo, los algortimos que usamos a estos efectos son muy similares. Es por esto que el análisis de uno y de otro se ven fuertemente relacionados. Los algoritmos utilizados se basan en los propuestos por Ilie et al. en [16], sobre los cuales en este trabajo propusimos ciertas extensiones para su uso aplicado a conjuntos de cadenas de caracteres. Los algoritmos usados para clasificación de proteínas están basados en los propuestos por Turjanski et al. en [6]. Sobre estos realizamos distintas modificaciones buscando mejorar la precisión de los resultados obtenidos. De esta forma, distinguimos dos etapas en el trabajo: la primera consiste en identificación de proteínas y la segunda en clasificación de proteínas.

La primera etapa consistió en buscar subcadenas de aminoácidos, dentro de una proteína, que no se repitieran en ninguna proteína perteneciente a la familia de la misma. De esta manera, al encontrar esta subcadena de aminoácidos, que llamamos *minimal tag*, podemos identificar unívocamente a la proteína en cuestión del resto. Los resultados obtenidos empíricamente fueron satisfactorios puesto que encontramos que, para la familia *Ankyrin* (compuesta por 38.051 proteínas), un 88.01 % de las proteínas pueden ser identificadas unívocamente por al menos un *minimal tag* de exáctamente 5 aminoácidos y un 10.21 % por al menos un *minimal tag* de exacatmente 6. Los tamaños 5 y 6 son relativamente pequeños respecto del tamaño total de una proteína y son convenientes para que un *minimal tag* sea utilizable como tal.

Queda como trabajo futuro profundizar más en la identificación de proteínas utilizando *minimal tags*, en particular realizar pruebas sobre otras familias. Sería de especial interés, realizar estudios sobre familias de proteínas de tamaños marcadamente diferentes que *Ankyrin*. Incluso se podría analizar la posibilidad de identificar una proteína mediante un *minimal tag* dentro de un conjunto de proteínas mayor, que comprenda varías familias. Un posible estudio sería tratar de discernir como escala el tamaño mínimo de *minimal tag* necesario para identificar un cierto alto porcentaje predefinido de proteínas del conjunto, respecto del tamaño total del conjunto. Otra posible línea de investigación sería tratar de identificar en qué partes de las proteínas se encuentran los *minimal tags*. Buscar si existe alguna relación entre sectores que contienen *minimal tags* para distintas proteínas, para *minimal tags* de distintos largos. También se podría buscar una correlación entre grandes areas de *minimal tags* para una cierta proteína de test, al inentar compararla contra una familia a la que no pertenezca y un bajo valor de familiridad al evaluarla contra esa familia. Y de la misma manera, el caso inverso.

La segunda etapa consistió en, dadas una familia de proteínas y una proteína de test, poder decidir si esta pertenece a la familia o no. Con este propósito utilizamos distintas alternativas de una fórmula de *familiaridad*, basadas en lo propuesto por Turjanski et al. en [6]. Estas fórmulas toman como parámetros la cadena que representa la proteína de test y el conjunto de SMR que representa a la familia, obtenido a partir de los algoritmos propuestos en la sección de *extensiones*. Al aplicar una fórmula obtenemos un valor de *familiaridad* que representa la afinidad de la proteína de test con la familia. De esta

manera, cuanto más marcadamente diferentes sean los valores de familiaridad obtenidos cuando la proteína de test pertenece a la familia y cuando no, diremos que mayor es la precisión de la fórmula usada.

Con el propósito de aumentar la precisión de la *fórmula de familiaridad* experimentamos con distintas alternativas. El primer cambio sobre la propuesta original de Turjanski et al. [6] consistió en utilizar SMR en lugar de MR. Como mencionamos esto conlleva una mejora en la complejidad algoritmica, pues al representar un concepto más sencillo, los SMR son más fáciles de obtener. Buscar los MR en una cadena de largo n tiene un costo de $O(n\ log\ n)$ mientras que buscar SMR un costo de $O(n)$. Donde radica la ventaja de la idea es en el hecho de que los SMR resultan muy similares a los MR para calcular familiaridad. Incluso aplicando ciertas modificaciones en el cálculo de familiaridad este se vuelve más preciso. Luego, redefinimos el conjunto de *cobertura* utilizado para *cubrir* una proteína de test. A continuación probamos distintas maneras de pesar los cálculos de *cobertura*. En una primera instancia, pesar la *cobertura* de acuerdo a la cantidad de ocurrencias de cada SMR, otorgó resultados poco satisfactorios, considerablemente menos precisos que los obtenidos hasta el momento. Por otro lado, pesar cada *cobertura* de acuerdo al tamaño de los SMR utilizando distintas funciones de pesado llevó a una cosiderable mejora en la precisión de los resultados. Sobre esta última idea implementamos la variante de aplicar una condición sobre el tamaño de los SMR para decidir si utilizabamos la *cobertura* computada en el cálculo de *familiaridad* o no. Esta fue la última alternativa implementada y fue la que otorgó mejores y más precisos resultados. Esto se debe a distintos motivos. En primer lugar, los valores de familiaridad obtenidos se mantienen en rangos acotables en función del tamaño de las proteínas de test. En la alternativa anterior, al usar $familiaridad_{fg}$ para dos funciones f y g dadas, los valores de $familiaridad$, de querer ser acotados, deberían acotarse en función del tamaño de la proteína de test *y también* de las funciones f y g. Además, los valores de familiaridad cuando una proteína de test no pertenece a una familia son fácilmente acotables, pues toman valores muy similares independientemente de las familias y proteínas de test utilizadas. Por último, la diferencia entre valores de familiaridad cuando una proteína de test pertenece a una familia y cuando no, aumenta. Lo cual es precisamente el efecto buscado para poder clasificarlas más fácilmente.

Si analizamos la mejora en la precisión para $familiaridad_{fg}$ y $familiaridad_{\geq n}$, respecto de todas las versiones anteriores del cálculo, podemos especular con que, si bien ambas actúan de distintas maneras, el motivo a partir del cual se origina la mejora es el mismo para ambas. Podemos pensar que los SMR de longitudes pequeñas, aproximadamente de 2 a 5, que representan una familia, tienen muchas chances de encontrarse en una proteína de test cualquiera, más allá de si esta pertenece a la familia o no. Por este motivo, las coberturas calculadas a partir de SMR de los tamaños mencionados contribuyen al cálculo de familiaridad desfavorblemente. Estas coberturas estarían generando un *"ruido"* al incrementar el valor de familiaridad por igual para proteínas de test pertenecientes y no pertenecientes a la familia. Y lo que es peor, el valor de este *"ruido"* representa una proporción muy significativa del valor total de familiaridad. Esto lo intuímos a partir de que los SMR de longitudes pequeñas tienen altas chances de encontrarse en cualquier proteína e incluso de cubrir una proporción alta de ella, por eso el alto valor de *cobertura* generado. Teniendo en cuenta esto, sería interesante estudiar las chances de generar, a partir de una cierta familia, todos los SMR posibles, para longitudes pequeñas. Con "todos los SMR posibles" nos referimos a toda la combinatoria de cadenas de esa longitud, dado el alfabeto. Si se generaran todos o muchos de los SMR posibles, podemos ver trivialmente como el

cubrimiento de una proteína de test cualquiera seria total o casi total. A partir de esto se podría llegar a contar con datos que respalden en una forma más precisa la decisión de utilizar $familiaridad_{\geq n}$ y que definan mejor el n ideal a elegir.

Como se mencionó, intuímos que la gran ventaja de $familiaridad_{\geq n}$ sobre $familiaridad_{fg}$, se encuentra en que ciertos SMR de una longitud n tienen muy altas chances de encontrarse en una proteína cualquiera, mientras que para los de longitud $n+1$ esas chances decrementan drásticamente. Por eso la idea que modela $familiaridad_{\geq n}$ sería mas adecuada que la de pesar los valores con una función estrictamente creciente y que los pesos asignados a los valores de cobertura generados por SMR de distintas longitudes se incrementen de manera gradual. A partir de esto, surge la posible línea de estudio de obtener datos estadísticos sobre las chances de que un cierto SMR de longitud n se encuentre en una proteína para distintos valores de n. A su vez, buscar algún tipo de relación entre esas chances y la precisión de $familiaridad_{\geq n}$. Si bien, en principio, utilizar $familiaridad_{\geq n}$ es mejor que utilizar $familiaridad_{fg}$, sería de interés evaluar un uso conjunto, donde se tomen SMR de longitudes mayores o iguales a un cierto n pero a su vez se pesen las coberturas en función del tamaño del SMR a partir de funciones f y g.

Como potencial ventaja de performance adicional de esta última alternativa, destaquemos que una gran cantidad de cálculos de *cobertura* para los SMR de tamaños menores que n, siendo $familiaridad_{\geq n}$ la fórmula usada, son innecesarios. Recordemos que los SMR de menor tamaño son justamente los que se encuentran en mayor cantidad en una familia, con lo cual la cantidad de cómputos a realizar para calcular la *cobertura* correspondiente es considerable.

Un análisis interesante también sería buscar si existe una relación entre el tamaño de una familia en cuanto a cantidad total de aminoácidos de todas sus proteínas y la precisión de los valores de familiaridad obtenidos. Así mismo, buscar si existe una relación entre la precisión de valores de familiaridad y el tamaño de las proteínas de test. Especulamos que podría existir una relación que afecte negativamente a la precisión. De ser así, sería de interés buscar una manera de salvar esta dificultad. Esta intuición surge a partir de considerar que proteínas de test que sean considerablemente más largas que otras tienen más chances de generar mayores valores de cobertura y a su vez mayor familiaridad. Esto debido a que, dada su extensión, las chances de que un cierto SMR las cubra podrían ser mayores. De encontrarse que esta especulación se vea reflejada en la práctica se podría buscar una forma de introducir la longitud de la proteína de test como parámetro en el cálculo de familiaridad y así, de alguna manera, contemplar también este factor.

4. ANEXO: EJECUCIÓN

A continuación se describe como compilar los fuentes y ejecutarlos sobre los datasets utilizados en el trabajo. La implementación está desarrollada en java 8 con lo cual es portable a distintas plataformas. Por simplicidad, sólo se detallan las intrucciones para un sistema operativo linux (probado para Ubuntu 15.10).

- Descomprimir el proyecto

```
tar -xzvf project.tar.gz
cd project
```

- Estructura del proyecto

```
project
    sources
        src
        target
    data
        proteins
            testGroupDataset
            familyDataset
            squashed_data
    results
```

- Instalar java 8

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer -y
sudo apt install oracle-java8-set-default
```

- Corroborar que quedo bien instalado ejecutando

```
java -version
```

(La versión se mostrará con el formato **java version 1.8.XXXXXXX**)

- Intalar maven para poder compilar el proyecto y descargar dependencias

```
sudo apt-get install maven
```

- Corroborar que quedo bien instalado ejecutando

```
mvn -version
```

- Hacer un *build* del proyecto

```
cd sources
mvn clean package
```

- El jar generado con el codigo y todas las dependencias incluidas quedará en target/prago-1.0-jar-with-dependencies.jar

- Ejecutar el paso de pre-procesamiento que se encarga de generar un único archivo *.family por cada familia de proteínas usado para el cálculo de familiaridad y un *.index usado para el cálculo de minimal tags

```
java -cp target/prago-1.0-jar-with-dependencies.jar
ar.uba.dc.prago.executable.SquashData
../data/proteins/familyDataset
ABCtran,Annexin,Arm,CWbinding1,Collagen,CorA,Fer4,Filamin,GDCP,Globin,
Glycohydro19,GreAGreB,HEAT,HelicaseC,HemolysinCabind,Hexapep,Kelch1,LRR1,
Ldlrecepta,Ldlreceptb,MIP,MORN,MgtE,Mitocarr,NEWAnk,NEWWD40,Nebulin,PBP,
PD40,PFLlike,PIN,PPR,PPbinding,PUD,PUF,Pentapeptide,PeptidaseC25,Pkinase,
Rhomboid,Sel1,TPR1,TSP1,TerB,Thaumatin,ValtRNAsyntC,YadAhead,ank,dehalogenase,
mix,mixScrambled
../data/proteins/squashed_data
```

- Para ejecutar el cálculo de minimal tags correr:

```
java -cp target/prago-1.0-jar-with-dependencies.jar -Xms2g -Xmx4g -XX:NewRatio=8
ar.uba.dc.prago.executable.MinimalUniqueMain
../data/proteins/squashed_data/ank.family
../data/proteins/squashed_data/ank.index
../results
```

- Para ejecutar el cálculo de familiaridad correr:

```
java -cp target/prago-1.0-jar-with-dependencies.jar -Xms2g -Xmx5g -XX:NewRatio=8
ar.uba.dc.prago.executable.CrossExecMain
<familiarity-alternative>
<families-directory>
<test-group-dataset>
<output-directory>
```

Donde

familiarity-alternative es uno de los siguientes:
–cap10 –standard –occurrences-applied –occurrences-global
–linear –quad –exp
–start4 –start5 –start6 –start7 –start8 –start9
families-directory es el directorio donde están los archivos *.family (los que fueron generados en el pre-procesamiento)
test-group-dataset el directorio 'testGroupDataset'
output-directory el directorio donde los archivos *.csv de resultados de familiaridad y de coverage van a ser generados.

El argumento de la jvm -Xmx5g indica 5 gigabytes de memoria que se asignan al proceso. Si se le asigna menos tardaría demasiado porque el volumen de datos es demasiado

grande y podría procesar menos cantidad de datos en paralelo. El tiempo de ejecución para este dataset con un procesador Intel i5-3570 CPU 3.40GHz x 4 es de varias horas.

- Ejemplo

```
java -cp target/prago-1.0-jar-with-dependencies.jar -Xms2g -Xmx5g -XX:NewRatio=8
ar.uba.dc.prago.executable.CrossExecMain
--start9
../data/proteins/squashed_data
../data/proteins/testGroupDataset
../results
```

- Reutilizar cobertura ya calculada para nuevos cálculos de familiaridad. Con esta opción se pueden utilizar coberturas generadas previamente para calcular mucho más rápido otras alternativas de familiaridad. Tener en cuenta que las alternativas –cap10 –occurrences-applied y –occurrences-global utilizan coberturar calculadas de diferente manera que todas las demás (y diferentes entre sí) todas las demás alternativas de familiaridad pueden utilizar las mismas coberturas para su cálculo. Ejemplo:

```
java -cp target/prago-1.0-jar-with-dependencies.jar -Xms2g -Xmx4g -XX:NewRatio=8
ar.uba.dc.prago.executable.FamiliarityExecMain
--standard
../results/coverage2017-03-12T14:20:09.016.csv
../results
```

Bibliografía

[1] D Nelson and M Cox. Lehninger principles of biochemistry. W. H. Freeman; 4th edition, 2004.

[2] Ingrid Wagner and Hans Musso. New naturally occurring amino acids. Angewandte Chemie International Edition in English, 22(11):816–828, 1983.

[3] Peter J. Reeds. Dispensable and indispensable amino acids for humans. The Journal of Nutrition, 130(7):1835S–1840S, 2000.

[4] Andrey V. Kajava. Tandem repeats in proteins: From sequence to structure. Journal of Structural Biology, 179(3):279 – 288, 2012. Structural Bioinformatics.

[5] Carl Ivar Branden and John Tooze. Introduction to protein structure. Garland Science; 2nd edition, 1998.

[6] Pablo Turjanski, R. Gonzalo Parra, Rocío Espada, Verónica Becher, and Diego U. Ferreiro. Protein repeats from first principles. Scientific Reports, 6:23959, Apr 2016. doi: 10.1038/srep23959.

[7] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[8] Peter Weiner. Linear pattern matching algorithms. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973), SWAT '73, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[9] Edward M. McCreight. A space-economical suffix tree construction algorithm. J. ACM, 23(2):262–272, April 1976.

[10] E. Ukkonen. On-line construction of suffix trees. Algorithmica, 14(3):249–260, 1995.

[11] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching, CPM'03, pages 200–210, Berlin, Heidelberg, 2003. Springer-Verlag.

[12] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Proceedings of the 30th International Conference on Automata, Languages and Programming, ICALP'03, pages 943–955, Berlin, Heidelberg, 2003. Springer-Verlag.

[13] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear time suffix array construction using d-critical substrings. In Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching, CPM '09, pages 54–67, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. ACM Comput. Surv., 39(2), July 2007.

[15] Dan Gusfield. Algorithms on strings, trees, and sequences: Computer science and computational biology. Cambridge University Press, 1997.

[16] Lucian Ilie and William F. Smyth. Minimum unique substrings and maximum repeats. Fundam. Inf., 110(1-4):183–195, January 2011.

[17] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, CPM '01, pages 181–192, London, UK, UK, 2001. Springer-Verlag.

[18] L. Marsella, F. Sirocco, A. Trovato, F. Seno, and S. C. Tosatto. REPETITA: detection and discrimination of the periodicity of protein solenoid repeats by discrete Fourier transform. Bioinformatics, 25(12):i289–295, Jun 2009.

[19] H. Luo and H. Nijveen. Understanding and identifying amino acid repeats. Brief. Bioinformatics, 15(4):582–591, Jul 2014.

[20] National Human Genome Research Institute. Glosario de términos genéticos: proteína. https://www.genome.gov/glossarys/index.cfm?id=169. Accedido: 2016-12-07.

[21] Verónica Becher, Alejandro Deymonnaz, and Pablo Heiber. Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. Bioinformatics, 25(14):1746, 2009.

[22] Leila K. Mosavi, Tobin J. Cammett, Daniel C. Desrosiers, and Zheng-yu Peng. The ankyrin repeat as molecular architecture for protein recognition. Protein Science, 13(6):1435–1448, 2004.