

**Centro de Investigación Científica y de Educación
Superior de Ensenada, Baja California**



**Maestría en Ciencias
en Ciencias de la Computación**

**Análisis de transcriptomas para el descubrimiento
de péptidos antimicrobianos**

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de
Maestro en Ciencias

Presenta:

Germán Meléndrez Carballo

Ensenada, Baja California, México

2018

Tesis defendida por

Germán Meléndrez Carballo

y aprobada por el siguiente Comité

Dr. Carlos Alberto Brizuela Rodríguez

Codirector de tesis

Dr. Gabriel Del Río Guerra

Codirector de tesis

Dra. Patricia Juárez Camacho

Dr. Andrey Chernykh

Dr. Israel Marck Martínez Pérez

M.C. Jorge Enrique Luna Taylor



Dr. Jesús Favela Vara

Coordinador del Posgrado en Ciencias de la Computación

Dra. Rufina Hernández Martínez

Directora de Estudios de Posgrado

Germán Meléndrez Carballo © 2018.

Queda prohibida la reproducción parcial o total de esta obra sin el permiso formal y explícito del autor y director de la tesis.

Resumen de la tesis que presenta Germán Meléndrez Carballo como requisito parcial para la obtención del grado de Maestro en Ciencias en Ciencias de la Computación.

Análisis de transcriptomas para el descubrimiento de péptidos antimicrobianos

Resumen aprobado por:

Dr. Carlos Alberto Brizuela Rodríguez

Codirector de tesis

Dr. Gabriel Del Río Guerra

Codirector de tesis

Ante la aparición cada vez más frecuente de enfermedades infecciosas producidas por bacterias resistentes a múltiples fármacos, el descubrimiento de antibióticos se está volviendo una necesidad urgente. Dada la complejidad del problema, se han empleado varios enfoques para abordarlo, siendo uno de ellos la identificación de péptidos antimicrobianos (*AMP*, **AntiMicrobial Peptides**); motivado principalmente por el uso frecuente observado en la naturaleza para combatir con ellos a diferentes microorganismos y porque presentan un amplio espectro de mecanismos de acción, lo que los convierte en una alternativa potencial a los antibióticos convencionales. Los péptidos antimicrobianos comúnmente se derivan de polipéptidos que son proteolizados y que se producen en organismos vivos cuando estos deben defenderse de microorganismos que son una amenaza para su sobrevivencia. Para abordar la identificación de péptidos antimicrobianos que pudieran producirse en la naturaleza para atacar a microorganismos, en este trabajo usamos datos de expresión de genes (transcriptomas) para identificar el posible proteoma que esté siendo expresado en un momento dado, y proponemos un nuevo enfoque basado en la proteólisis *in silico*, aplicando para esto un ventaneo a las secuencias del proteoma, descartando las ventanas redundantes por medio de algoritmos de ordenamiento y comparación lexicográfica. Las ventanas de secuencias resultantes son evaluadas usando técnicas de aprendizaje de máquina para su clasificación como antimicrobianas o no antimicrobianas. Manejar grandes cantidades de datos en un tiempo aceptable, como la demandada al trabajar con proteomas, requiere de la explotación de los recursos ofrecidos por los procesadores modernos. En este trabajo se realizó una implementación en paralelo del flujo de trabajo previamente descrito. Experimentos computacionales muestran que el diseño es capaz de analizar un proteoma de 16,037 secuencias, obteniendo un total de 243,643,492 *k*-meros únicos de entre 10 y 60 aminoácidos de longitud, en aproximadamente 7 horas.

Palabras clave: transcriptomas, RNA-Seq, péptidos antimicrobianos, programación paralela, aprendizaje de máquina

Abstract of the thesis presented by Germán Meléndrez Carballo as a partial requirement to obtain the Master of Science degree in Computer Science.

Analysis of transcriptomes for antimicrobial peptides discovery

Abstract approved by:

Dr. Carlos Alberto Brizuela Rodríguez

Thesis Co-Director

Dr. Gabriel Del Río Guerra

Thesis Co-Director

Faced with the increasingly frequent occurrence of infectious diseases caused by bacterias resistant to multiple drugs, the discovery of antibiotics is becoming an urgent need. Given the complexity of the problem, several approaches have been used to address it. The identification of antimicrobial peptides (*AMP*, **AntiMicrobial Peptides**) is one of these approaches, motivated mainly by the frequent use of these molecules to fight different microorganisms and because they present a wide spectrum of mechanisms of action, which makes them a potential alternative to conventional antibiotics. Antimicrobial peptides are commonly derived from polypeptides that are proteolyzed and that are produced in living organisms when they must defend themselves against microorganisms that are a threat to their survival. To identify antimicrobial peptides that could occur in nature, in this work we analyze expression data (transcriptome) to search in the translated proteome that is being expressed at a given time and propose a new *in silico* proteolysis-based approach. To this aim we apply a sliding window to the proteome sequences, discarding redundant fragments of amino acids by using sorting algorithms and lexicographic comparison. The resulting non redundant sequences are evaluated using machine learning techniques to classify them as antimicrobial or non-antimicrobial. Handling large amounts of data in an acceptable time, such as the one demanded when working with proteomes, requires the exploitation of the resources offered by modern processors. In this work, a parallel implementation of the previously described workflow was carried out. Computational experiments show that the design is capable of analyzing a proteome of 16,037 sequences, obtaining a total of 243,643,492 unique k-mers between 10 and 60 amino acids of length, in approximately 7 hours.

Keywords: transcriptomes, RNA-Seq, antimicrobial peptides, parallel programming, machine learning

Dedicatoria

A mis padres y hermana por todo su apoyo, amor, consejos, y por siempre estar a mi lado. A mi mentor, M.C. Jorge Enrique Luna Taylor por haberme iniciado en la investigación científica, haber confiado en mí para concluir mi posgrado, y motivarme a seguir superándome.

Agradecimientos

A mis asesores de tesis, Dr. Carlos Alberto Brizuela Rodríguez y Dr. Gabriel Del Río Guerra por todo su apoyo y enseñanzas. Así mismo, a los miembros del comité de tesis, Dra. Patricia Juárez Camacho, Dr. Andrey Chernykh, Dr. Israel Marck Martínez Pérez y M.C. Jorge Enrique Luna Taylor por su tiempo dedicado, observaciones y sugerencias durante mi posgrado.

A CICESE por permitirme realizar mis estudios de posgrado, y a CONACyT por brindarme el apoyo económico para realizar mis estudios.

Tabla de contenido

	Página
Resumen en español	ii
Resumen en inglés	iii
Dedicatoria	iv
Agradecimientos	v
Lista de figuras	x
Lista de tablas	xv
Lista de algoritmos	xvi
Capítulo 1. Introducción	
1.1. Antecedentes y motivación	1
1.2. Definición del problema	3
1.3. Trabajos previos relevantes	3
1.4. Objetivos de la investigación	5
1.4.1. Objetivo general	5
1.4.2. Objetivos específicos	5
1.5. Estrategia de solución propuesta	6
1.6. Organización de la tesis	6
Capítulo 2. Marco teórico	
2.1. Conceptos biológicos	7
2.1.1. Péptidos	7
2.1.2. Péptidos antimicrobianos (AMPs)	7
2.1.3. Transcriptomas	8
2.1.4. RNA-Seq	9
2.1.5. FastQ	11
2.1.6. Trinity	12
2.2. Conceptos computacionales	13
2.2.1. Paralelización y concurrencia	13
2.2.1.1. SIMD - Vectorización	15
2.2.2. Aprendizaje de máquina	16
2.2.2.1. Selección de características	17
Capítulo 3. Estructuras de datos y algoritmos para la extracción y conteo eficiente de k-meros en proteomas	
3.1. Descripción general de la propuesta de solución	18
3.2. Motivación	20
3.3. Tablas Hash	21
3.3.1. Función hash	23
3.3.2. Hashing universal	24
3.3.3. Hashing perfecto	24
3.3.3.1. Implicaciones	24

Tabla de contenido

3.4.	Propuesta de solución	27
3.4.1.	Ventaneo del proteoma	27
3.4.2.	Ordenamiento de k -meros	28
3.4.3.	Comparación de k -meros para remover duplicados	28
3.4.4.	Algoritmo propuesto	30
3.4.4.1.	Lectura del proteoma	31
3.4.4.2.	Extracción de k -meros totales y eliminación de k -meros redundantes	32
3.4.4.3.	Cálculo de descriptores moleculares	32
3.4.4.4.	Evaluación y depuración de k -meros predichos como no-AMP	32
3.4.4.5.	Agrupación de k -meros predichos como AMP provenientes de una misma secuencia (podado de proteoma)	32
3.4.4.6.	Escritura de proteoma podado	33
3.4.5.	InProt - <i>In silico</i> Proteolysis	33
3.4.6.	CAMPRED - CAMP AMP PREDiction	33
Capítulo 4. Experimentos y resultados		
4.1.	Depuración del conjunto de casos negativos con CAMP	37
4.2.	Cálculo de descriptores moleculares y entrenamiento de una máquina de aprendizaje	40
4.3.	Pruebas de cálculo descriptores moleculares y evaluación de la máquina de aprendizaje	41
4.3.1.	Comparación de la implementación propia de una máquina de aprendizaje y libsvm	42
4.3.2.	k -meros predichos	48
4.4.	k -meros predichos - CAMP	50
4.5.	Evaluación del tiempo de cómputo y consumo de memoria RAM del algoritmo propuesto	60
4.5.1.	Evaluación del tiempo de cómputo para reducción de k -meros duplicados	60
4.5.2.	Evaluación del tiempo de cómputo y consumo de memoria RAM con diferentes hilos de CPU	62
4.6.	Conteo de k -meros predichos por secuencia	64
4.6.1.	Podado del proteoma	73
4.7.	Discusión	74
Capítulo 5. Conclusiones y trabajo futuro		
5.1.	Sumario	76
5.2.	Conclusiones	76
5.3.	Trabajo futuro	78
Literatura citada		79
Anexo 1 - Tabla ASCII y codificaciones formato FASTQ		85

Tabla de contenido

Anexo 2 - Distribución de la base de datos CAMP y total de <i>k</i>-meros del proteoma	87
Anexo 3 - InProt - <i>In silico</i> Proteolysis	89
.1. Descarga del código fuente	89
.2. Requerimientos	89
.3. Selección de implementación de la máquina de aprendizaje	90
.4. Uso de punto flotante sencillo o punto flotante doble	90
.5. Instalación	90
.6. Opciones de comportamiento	91
.6.1. Escritura de <i>k</i> -meros predichos (opción “-w” ó “--write”)	92
.6.2. Modo “consciente” del consumo de memoria RAM (opción “-a” ó “--aware”)	93
.6.2.1. Modo informativo (opción “-v” ó “--verbose”)	93
.6.3. Ejemplos de uso	93
.6.3.1. <i>k</i> -meros desde 10 hasta 60	94
.6.3.2. <i>k</i> -meros desde 10 a 20, modo informativo y 2 hilos	94
.6.3.3. <i>k</i> -meros desde 10 hasta 60, escritura de <i>k</i> -meros predichos (AMP y no-AMP), modo informativo, y modo consciente del uso de memoria RAM	94
Anexo 4 - CAMPRED - CAMP AMP PREDiction	95
.1. Descarga del código fuente	95
.2. Requerimientos e instalación	95
.3. Opciones de comportamiento	96
.3.1. Guardar en disco duro la respuesta del servidor (opción “-k” ó “--keep”)	98
.3.2. Número de secuencias de cada conjunto (opción “-n” ó “--nseqs”)	98
.3.3. Número de reintentos (opción “-s” ó “--send”)	98
.3.3.1. Modo informativo (opción “-v” ó “--verbose”)	98
.3.4. Ejemplos de uso	99
.3.4.1. Particionado en subconjuntos de 10 secuencias, 2 hilos de CPU y SVM como algoritmo	99
.3.4.2. Particionado en subconjuntos de 50 secuencias, todos los algoritmos y guardando la respuesta de CAMP	99
Anexo 5 - Uso de SLURM, Trinity y TransDecoder	100
.1. SLURM - Manejador de procesos del cluster OMICA del CICESE	100
.1.1. Nodos y particiones	100
.1.2. Mandar procesos	100
.1.2.1. Plantilla básica para proceso de SLURM	101
.2. Trinity - Uso en el cluster OMICA del CICESE	103
.2.1. Instalación	103
.2.2. Ensamblar de forma paralela en un solo nodo	104
.2.3. Ensamblar de forma distribuída en varios nodos	105
.3. TransDecoder	106

Tabla de contenido

.3.1. Instalación	107
.3.2. Fase 1 - Predicción de las secuencias codificantes	107
.3.3. Fase 2 - Identificación de ORFs homólogos a proteínas conocidas	108
.3.3.1. Búsqueda con Blastp	108
.3.3.2. Búsqueda con HMMER	109
.3.3.3. Fase 3 - Predicción de las regiones codificantes probables . .	110
Anexo 6 - Descriptores moleculares	111

Lista de figuras

Figura	Página
1. Taxonomía de Flynn	14
2. Repetición de un 4-mero (QQQQ) en tres secuencias del proteoma del camarón (Ghaffari <i>et al.</i> , 2014).	19
3. Representación de la extracción de diferentes k -meros y los valores de n descriptores moleculares.	21
4. Representación de extracción de un k -mero y el cálculo de sus n descriptores moleculares y su posterior clasificación por medio de una SVM previamente entrenada	21
5. Representación de una función hash, h , mapeando las llaves a una tabla hash, T , donde las llaves k_3 y k_4 son mapeadas a la misma ranura de la tabla, generando una colisión. <i>Figura inspirada de (Cormen, 2009)</i>	22
6. Tiempo de cómputo para la extracción, ordenamiento y eliminación de k -meros duplicados del proteoma del camarón, para $k = 10$ hasta $k = 60$, con diferentes números de hilos y compiladores. La extracción se realizó en un nodo del cluster OMICA del CICESE (<i>ver Capítulo 4</i>).	29
7. Pasos llevados a cabo por el algoritmo propuesto.	31
8. (a) Distribución del conjunto de casos negativos predichos por la SVM de CAMP (165 predichos como AMP y 3,845 predichos como no-AMP). (b) Probabilidad de predicción del conjunto de casos negativos por la SVM de CAMP. Datos procesados el 19/06/2017.	38
9. (a) Distribución del conjunto de casos negativos predichos por RF (Random Forest) de CAMP (75 predichos como AMP y 3,935 predichos como no-AMP). (b) Probabilidad de predicción del conjunto de casos negativos por RF de CAMP. Datos procesados el 19/06/2017.	38
10. (a) Distribución del conjunto de casos negativos predichos por DA (Discriminant Analysis) de CAMP (245 predichos como AMP y 3,765 predichos como no-AMP). (b) Probabilidad de predicción del conjunto de casos negativos por DA de CAMP. Datos procesados el 19/06/2017.	39
11. Distribución del conjunto de casos negativos predichos por ANN (Artificial Neural Network) de CAMP (438 clasificados como AMP y 3,572 clasificados como no-AMP). Datos procesados el 19/06/2017.	39
12. Tiempo de cómputo promedio (de 5 ejecuciones) para el cálculo de 51 descriptores moleculares (<i>ver Tabla 13</i>) de los k -meros únicos para $k = 10$ hasta $k = 60$ utilizando una implementación propia y la de libsvm de la máquina de aprendizaje con el compilador G++.	45
13. Tiempo de cómputo promedio (de 5 ejecuciones) para el cálculo de 51 descriptores moleculares (<i>ver Tabla 13</i>) de los k -meros únicos para $k = 10$ hasta $k = 60$ utilizando una implementación propia y de libsvm de una máquina de aprendizaje con el compilador Intel C++.	45

Lista de figuras

Figura	Página
14. Memoria RAM consumida por InProt con 1, 2, 4, 8 16 y 24 hilos, para el cálculo de 51 descriptores moleculares (<i>ver Tabla 13</i>) de los k -meros únicos para $k = 10$ hasta $k = 60$ utilizando una implementación propia y de libsvm de la máquina de aprendizaje con el compilador G++. La memoria RAM es la reportada por SLURM (comando “sacct”) en las 5 ejecuciones del algoritmo.	47
15. Memoria RAM consumida por InProt con 1, 2, 4, 8 16 y 24 hilos, para el cálculo de 51 descriptores moleculares (<i>ver Tabla 13</i>) de los k -meros únicos para $k = 10$ hasta $k = 60$ utilizando una implementación propia y libsvm como máquina de aprendizaje con el compilador Intel C++. La memoria RAM es la reportada por SLURM (comando “sacct”) en las 5 ejecuciones del algoritmo.	47
16. Distribución de la cantidad de k -meros totales, únicos y predichos como AMP por el algoritmo propuesto.	48
17. Total de k -meros predichos como AMPs por InProt y predichos como AMP por los cuatro algoritmos de CAMP. En el eje “X” se muestra el tamaño de k -mero y en el eje “Y” se muestra la cantidad de k -meros predichos como AMP por cada uno de los cuatro algoritmos y por los cuatro algoritmos en conjunto.	51
18. Total de k -meros predichos como No-AMPs por InProt y predichos como AMP por los cuatro algoritmos de CAMP. En el eje “X” se muestra el tamaño de k -mero y en el eje “Y” se muestra la cantidad de k -meros predichos como AMP por cada uno de los cuatro algoritmos y por los cuatro algoritmos en conjunto. Los valores para k igual a 43 y 48, se encuentran ausentes.	52
19. Tiempo de cómputo para la eliminación de k -meros redundantes entre 10 y 60 aminoácidos de longitud para el proteoma del camarón (Ghaffari <i>et al.</i> , 2014), utilizando diferentes números de hilos de CPU (1 a 24) y los compiladores G++ e Intel C++.	61
20. Memoria RAM consumida para la eliminación de k -meros redundantes entre 10 y 60 aminoácidos de longitud para el proteoma del camarón (Ghaffari <i>et al.</i> , 2014), utilizando diferentes números de hilos de CPU (1 a 24) y los compiladores G++ e Intel C++. La memoria RAM es la reportada por SLURM (comando “sacct”) de 5 ejecuciones.	61
21. Tiempo de cómputo para InProt con k -meros entre 10 y 60 aminoácidos de longitud para el proteoma del camarón (Ghaffari <i>et al.</i> , 2014), utilizando diferentes números de hilos de CPU (1 a 24) y los compiladores G++ e Intel C++.	63

Lista de figuras

Figura	Página
22. Memoria RAM consumida para InProt con k -meros entre 10 y 60 aminoácidos de longitud para el proteoma del camarón (Ghaffari <i>et al.</i> , 2014), utilizando diferentes números de hilos de CPU (1 a 24) y los compiladores G++ e Intel C++.	63
23. Histograma del proteoma del camarón (Ghaffari <i>et al.</i> , 2014) conformado por 16,037 secuencias únicas entre 99 y 8,876 aminoácidos de longitud.	65
24. Distribución de los k -meros predichos como AMP por secuencia de las 16,020 secuencias del proteoma del camarón con k -meros predichos como AMP por el InProt. En el eje "X" se encuentra el total de k -meros predichos como AMP por secuencia, y en el eje "Y" el total de secuencias del proteoma con ese número de predicciones.	68
25. Las 10 secuencias con mayor número de k -meros predichos por el algoritmo. Ordenadas de mayor a menor número de k -meros predichos, de arriba hacia abajo. (a) Densidad de los k -meros predichos de las secuencias de acuerdo al tamaño de los k -meros. (b) Densidad de los k -meros predichos de las secuencias de acuerdo a la posición de los k -meros dentro de la secuencia. <i>Gráfica con escala = 0.9.</i>	69
26. Densidad de los k -meros predichos de 30 secuencias tomadas de forma aleatoria. Ordenadas de mayor a menor número de k -meros predichos, de arriba hacia abajo, por número de k -meros predichos por secuencia. Siendo "m.70564" la secuencia con mayor número de k -meros predichos por un total de 51,257 k -meros (desde 10-meros hasta 60-meros) con una longitud de 8,830 aminoácidos. <i>Gráfica con escala = 0.9.</i>	69
27. Las 10 secuencias más cortas de diferentes tamaños entre ellas. Ordenadas de mayor a menor número de k -meros predichos como AMP, de arriba hacia abajo. Densidad de los k -meros predichos de las secuencias de acuerdo al tamaño de los k -meros.	71
28. Las 10 secuencias más cortas de diferentes tamaños entre ellas. Ordenadas de mayor a menor número de k -meros predichos como AMP, de arriba hacia abajo. Densidad de los k -meros predichos de las secuencias de acuerdo a la posición de los k -meros dentro de la secuencia. <i>Gráfica con escala = 0.9.</i>	72
29. Las 10 secuencias entre 900 y 1,000 aminoácidos de longitud con mayor cantidad de k -meros predichos como AMP. Ordenadas de mayor a menor número de k -meros predichos como AMP, de arriba hacia abajo. Densidad de los k -meros predichos de las secuencias de acuerdo al tamaño de los k -meros. <i>Gráfica con escala = 0.9.</i>	72

Lista de figuras

Figura	Página
30. Las 10 secuencias entre 900 y 1,000 aminoácidos de longitud con mayor cantidad de k -meros predichos como AMP. Ordenadas de mayor a menor número de k -meros predichos como AMP, de arriba hacia abajo. Densidad de los k -meros predichos de las secuencias de acuerdo a la posición de los k -meros dentro de la secuencia. <i>Gráfica con escala = 0.9</i>	73
31. Histograma del proteoma (podado) del camarón (Ghaffari <i>et al.</i> , 2014), conformado por 63,583 secuencias únicas entre 10 y 1,858 aminoácidos de longitud. En el eje “X” se muestra la longitud (en aminoácidos) de las secuencias, y en el eje “Y” se muestra el total de secuencias con esa longitud.	74
32. Tabla del estándar ASCII. Figura tomada de https://wikimedia.org	85
33. Codificaciones del formato FASTQ. Figura tomada de https://en.wikipedia.org/wiki/FASTQ_format	86
34. Histograma de la base de datos de CAMP extraída el 23 de enero del 2017 empleando los filtros descritos en (Verdugo, 2014). Conformada por un total de 2,258 péptidos antimicrobianos entre 55 y 99 aminoácidos. Las líneas punteadas verticales representan los valores de 10 y 60, intervalo en donde se encuentran concentrados la mayoría (90.43%) de los péptidos que conforman la base de datos.	87
35. k -meros (totales y únicos) entre 10 y 60 aminoácidos del proteoma del camarón. <i>Cifras representadas en millones</i>	88
36. Lista de particiones y nodos asignados a las mismas. Las columnas representan el nombre de la partición, disponibilidad, tiempo límite de cómputo, nodos asignados, estado de la partición y la lista de nodos que conforman la partición.	101
37. Ejecución del comando “sbatch” de SLURM para ejecutar la plantilla especificada en la Subsección .1.2.1 y despliegado de los procesos ejecutándose en el cluster, mostrando el identificador, partición, nombre, usuario, estado, tiempo de cómputo, tiempo límite de cómputo y nodo asignado del proceso. La opción “-l” del comando “squeue” es para indicarle a SLURM que muestre los nombres de los campos completos, debido a que por omisión los muestra abreviados.	103
38. Progreso de la segunda fase del ensamblado del Transcriptoma de (Yoo <i>et al.</i> , 2014) de forma distribuída con Trinity.	106
39. Primera fase de TransDecoder utilizando el transcriptoma de (Ghaffari <i>et al.</i> , 2014).	107

40. Tercera fase de TransDecoder utilizando el transcriptoma de (Ghaffari *et al.*, 2014) y los resultados obtenidos en la segunda fase. 110

Lista de tablas

Tabla		Página
1.	<i>k</i> -meros y su $Radix_2$	26
2.	Comparativa de la implementación propia y de libsvm de la máquina de aprendizaje (compilador G++)	43
3.	Comparativa de la implementación propia y de libsvm de la máquina de aprendizaje (compilador Intel C++)	44
4.	<i>k</i> -meros totales, únicos y predichos como AMP	49
5.	<i>k</i> -meros predichos como AMP por InProt y como AMP por los cuatro algoritmos de CAMP con CAMPRED	53
6.	<i>k</i> -meros predichos como No-AMP por InProt y como AMP por los cuatro algoritmos de CAMP con CAMPRED	57
7.	Total de <i>k</i> -meros predichos de las 10 secuencias con mayor número de <i>k</i> -meros predichos como AMP	65
8.	Las 10 secuencias con mayor proporción de <i>k</i> -meros predichos como AMP	66
9.	Las 10 secuencias con menor proporción de <i>k</i> -meros predichos como AMP	66
10.	Total de <i>k</i> -meros predichos de 30 secuencias seleccionadas al azar.	67
11.	Las 10 secuencias más cortas de diferentes tamaños, longitud y cantidad de AMPs predichos por secuencia.	70
12.	Las 10 secuencias entre 900 y 1,000 aminoácidos de longitud con mayor cantidad de AMPs predichos por secuencia.	70
13.	Lista de descriptores moleculares calculados	111

Lista de algoritmos

Algoritmo	Página
1. Extracción de k -meros totales	29
2. Extracción de k -meros únicos	30
3. Comparación de k -meros	30
4. Agrupación de k -meros (podado del proteoma)	35
5. Evaluar si el k -mero k_j se encuentra dentro del k -mero k_i	36
6. Evaluar si el k -mero k_j se encuentra conectado con k_i	36
7. Evaluar si el k -mero k_i intersecta con k_j	36

Capítulo 1. Introducción

1.1. Antecedentes y motivación

En los años 80s, las compañías farmacéuticas estaban convencidas de tener suficientes antibióticos, declarando de forma anticipada la victoria contra los agentes patógenos resistentes a antibióticos, por lo que se enfocaron en otras enfermedades como cáncer, diabetes y enfermedades cardiovasculares, dejando de lado la investigación en los antibióticos (Tegos *et al.*, 2012). Sin embargo, debido al uso excesivo y descontrolado de estos en diferentes sectores como agricultura y ganadería, las bacterias han generado resistencia a múltiples antibióticos, motivando a los investigadores a buscar nuevos métodos, aumentando la investigación enfocada a los péptidos antimicrobianos (**AntiMicrobial Peptides**) y su potencial uso terapéutico (Giuliani *et al.*, 2010).

La resistencia a los antibióticos aparece en un tiempo menor que el tiempo necesario para el descubrimiento y comercialización de un antibiótico nuevo, haciendo el descubrimiento de antibióticos más difícil y costoso. Por ejemplo, el tiempo promedio para que un fármaco sea lanzado al mercado, es de 10 años, mientras que, toma menos de 10 años en aparecer una bacteria resistente a dicho antibiótico (Tegos *et al.*, 2012).

De acuerdo con O'Neill (2014), entre Europa y Estados Unidos, hay 50,000 muertes anuales atribuibles a infecciones resistentes a los antibióticos. Estimándose para el 2050, un total de 10 millones de muertes por año en todo el mundo.

Dentro de las técnicas empleadas para el descubrimiento de fármacos potenciales, se encuentra el análisis de transcriptomas (Prashanth y Lewis, 2015; Lavergne *et al.*, 2013; Prashanth *et al.*, 2012; Kim *et al.*, 2016; Wei *et al.*, 2016; Yoo *et al.*, 2014; Robertson y Cornman, 2014). El análisis de transcriptomas de glándulas de venenos, es una técnica con mucho auge en los últimos años, y empleada en el área de venómica (Prashanth y Lewis, 2015; Lavergne *et al.*, 2013; Prashanth *et al.*, 2012). Dado que los genes transcritos pueden o no producir un péptido o proteína (Heck y Wilusz, 2018; Lodish *et al.*, 2016), el estudio de transcriptomas ha sido de gran interés para la búsqueda de AMPs.

Otras técnicas para el descubrimiento de fármacos potenciales, hacen uso únicamente de descriptores moleculares, calculados a partir de proteínas, sin importar el origen de las mismas (como puede ser de transcriptomas), utilizándolos en conjunto con máquinas de soporte vectorial (*SVM*, **Support Vector Machine**) (Verdugo, 2014; Veltri y Shehu, 2013; Jamali *et al.*, 2016; Pilkington *et al.*, 2012; Kim *et al.*, 2016; Lata *et al.*, 2007), redes neuronales profundas (*DNN*, **Deep Neural Network**) (Unterthiner *et al.*, 2015; Wallach *et al.*, 2015; Ma *et al.*, 2015), redes neuronales artificiales (*ANN*, **Artificial Neural Network**) (Byvatov *et al.*, 2003; Lata *et al.*, 2007), entre otras.

A excepción de Torrent *et al.* (2012), los trabajos previos a éste, se centran en secuencias de aminoácidos de una longitud determinada, descartando aquellas secuencias mayores o iguales a dicha longitud (Yoo *et al.*, 2014; Kim *et al.*, 2016). Descartando el hecho de que algunos AMPs son producidos por medio de la proteólisis de proteínas más largas (Giuliani *et al.*, 2010). En Lata *et al.* (2007) hacen uso de secuencias de 15 aminoácidos (a lo mucho) de longitud para construir su modelo de SVM, estando imposibilitado a buscar péptidos antibacterianos en genomas o proteínas.

En Torrent *et al.* (2012) se centran en la identificación de la región antimicrobiana de las proteínas por medio de un deslizamiento de ventana mayor a 12 aminoácidos, calculando el índice antimicrobiano (*AI*, **Antimicrobial Index**) al fragmento de aminoácidos cubiertos por la ventana. Donde el índice antimicrobiano es calculado de acuerdo a los valores del índice IC_{50} obtenidos previamente del AMP *bactenicin 2A* (Hilpert *et al.*, 2005). Para ello, se debe fijar un umbral para el *AI* y un tamaño de ventana, donde si el fragmento de aminoácidos cubierto por la ventana se encuentra debajo del umbral, dicha región se considera un AMP putativo. Siendo la selección del tamaño de ventana y del umbral una de las desventajas principales de dicho enfoque.

Los métodos computacionales se han vuelto una herramienta imprescindible para el diseño de nuevos fármacos; esto es debido a la gran capacidad de cómputo de los sistemas de hoy en día. Nuestra propuesta consiste en implementar un algoritmo para la detección de fragmentos contiguos de aminoácidos con propiedades antimicrobianas, por medio de un deslizamiento de ventanas entre 10 y 60 aminoácidos de longitud, extendiendo los trabajos previos de Torrent y colaboradores.

1.2. Definición del problema

El problema del descubrimiento de péptidos antimicrobianos en transcriptomas se puede definir de la siguiente manera: dado un conjunto de cadenas de aminoácidos A (derivados de un transcriptoma), dos enteros positivos, K_{min} y K_{max} , y un modelo de aprendizaje, MOD , encontrar todas las secuencias S_i , que sean predichas por MOD como antimicrobianas tal que S_i sea un fragmento contiguo de aminoácidos de A , con $K_{min} \leq \|S_i\| \leq K_{max}$.

1.3. Trabajos previos relevantes

Para el descubrimiento de genes y péptidos antimicrobianos se han propuesto varias técnicas, desde técnicas de QSAR (Verdugo, 2014), hasta las más recientes donde analizan transcriptomas (Kim *et al.*, 2016; Yoo *et al.*, 2014; Robertson y Cornman, 2014; Wei *et al.*, 2016).

Verdugo (2014) hace uso de técnicas de QSAR con algoritmos evolutivos y máquinas de soporte vectorial, tomando como entrada los descriptores moleculares de dos conjuntos de datos: 1) compuesto por AMPs como casos positivos y no-AMPs como casos negativos, y 2) péptidos antibacterianos como casos positivos y péptidos con actividad biológica antiviral, antifúngica o sin actividad antimicrobiana como casos negativos. Como primera fase, se aplica un filtrado para reducir la dimensionalidad de los conjuntos de datos (prueba y entrenamiento), logrando disminuir la carga computacional y ruido para el algoritmo genético, obteniendo así mejores resultados en la predicción de péptidos antimicrobianos y antibacterianos. Para ello, se necesita tener buenos conjuntos de datos, tanto de prueba como de entrenamiento, los cuales fueron contruídos juntando una gran cantidad de péptidos con una longitud entre 10 y 100 residuos de aminoácidos; posteriormente se calcula los descriptores moleculares de los péptidos; finalmente, se eligen aquellos descriptores moleculares que se pudieron calcular para todos los péptidos. El enfoque logra una exactitud de alrededor de 91 % para péptidos antimicrobianos en entrenamiento y un 90 % en entrenamiento para péptidos antibacterianos.

Yoo *et al.* (2014); Kim *et al.* (2016); Robertson y Cornman (2014); Wei *et al.* (2016) hacen uso de transcriptomas para el descubrimiento de nuevos péptidos antimicrobianos, siguiendo un enfoque de prefiltrado de aquellos péptidos que se encuentran en bases de datos como CAMP y UniprotKB, así como en el transcriptoma.

Yoo *et al.* (2014) realizan descubrimiento de AMPs a través de un análisis del transcriptoma del ciempiés "*Scolopendra subspinipes mutilans*". Primero se realiza un *pipeline* de prefiltrado con base a 8 descriptores moleculares calculados, filtrando aquellas secuencias de proteínas cuyos descriptores no cumplan con los criterios asignados. Posteriormente, se realizan dos filtrados con BLAST (Altschul *et al.*, 1990): 1) se eliminan aquellas secuencias de proteínas que se predijeron como no antimicrobianas de acuerdo al algoritmo de análisis de discriminante de la base de datos CAMP (<http://www.camp.bicnirrh.res.in/>), 2) se eliminan aquellas secuencias de proteínas que tengan similitud con aquellas proteínas de la base de datos UniProt (<http://www.uniprot.org/>) a través de BLASTp. Por último, se realiza otro filtrado más, eliminando aquellas proteínas involucradas que tengan similitud con el proteoma humano y con aquellas involucradas en la escisión proteolítica. Una vez que se termina el proceso de filtrado, utilizan AMPA (Torrent *et al.*, 2012) para predecir la región con actividad antimicrobiana. Obteniéndose un total de 10 péptidos validados experimentalmente con una amplia actividad biológica.

Kim *et al.* (2016) siguen el mismo enfoque que Yoo *et al.* (2014), a diferencia que Kim *et al.* (2016) agregan un descriptor más (la propensión alérgica) calculada con Allerdicator (Dang y Lawrence, 2014), analizando el transcriptoma de la cucaracha "*Periplaneta americana (Linnaeus)*". Obteniéndose un total de 11 péptidos validados experimentalmente con una amplia actividad biológica.

Robertson y Cornman (2014) analizan el transcriptoma de las ranas "*Lithobates clamitans*" y "*Pseudacris regilla*". A diferencia de los anteriores, el *pipeline* llevado a cabo está conformado por la generación de "*clusters*" de aquellos *reads* identificados en la base de datos GenBank (<http://www.ncbi.nlm.nih.gov/>) por medio de una búsqueda con BLAST. Llegando a identificar 11 clusters de AMPs para *Lithobates clamitans* y 7 para *Pseudacris regilla*.

Wei *et al.* (2016) realizan el análisis del transcriptoma del aparato reproductor sexual de la mosca "*Bactrocera dorsalis*", siguiendo un *pipeline* de varias búsquedas locales con varias bases de datos de acceso público. Obteniendo un total de 20,419 unigenes anotados funcionalmente a proteínas conocidas.

1.4. Objetivos de la investigación

En éste trabajo, se propone la implementación de un algoritmo similar al propuesto por (Torrent *et al.*, 2012) con la diferencia de que nuestra propuesta se basará en la proteólisis *in silico*, con lo que se utilizarán diferentes tamaños de ventanas (entre 10 y 60 aminoácidos de longitud) en lugar de un solo tamaño de ventana; donde los tamaños de ventana fueron definidos de acuerdo a la distribución de longitudes de los AMPs de la base de datos de CAMP que cumplen con los filtros especificados por Verdugo (2014) (*ver Figura 34*). También se utilizarán los descriptores moleculares y el modelo de la SVM propuestos por Verdugo (2014). Por último, se utilizará el transcriptoma del camarón (Ghaffari *et al.*, 2014) como caso de estudio.

1.4.1. Objetivo general

Diseñar e implementar un algoritmo para el descubrimiento de péptidos con propiedades antimicrobianas en secuencias, de 10 a 60 amino ácidos de longitud, provenientes de experimentos de transcriptómica.

1.4.2. Objetivos específicos

- Diseñar e implementar en paralelo (CPU) técnicas de filtrado de secuencias y *k*-meros con propiedades antimicrobianas.
- Implementar en paralelo (CPU), rutinas para el cálculo de los descriptores moleculares.
- Evaluar el algoritmo propuesto en cuanto tiempo de cómputo y memoria RAM empleada para analizar un transcriptoma.

1.5. Estrategia de solución propuesta

Para abordar la problemática, se siguieron los siguientes procedimientos: primero se tradujo el transcriptoma por medio de TransDecoder (Haas y Papanicolaou, 2012), generando el proteoma. Una vez generado el proteoma, se procedió a hacer un deslizamiento de ventanas (k -meros) de diferentes tamaños (de 10 a 60 aminoácidos), seguido de la eliminación de aquellos k -meros redundantes por medio de algoritmos de ordenamiento y comparación lexicográfica. Los k -meros no redundantes (únicos) son procesados por medio de una máquina de aprendizaje, la cual predice si dichos k -meros cuentan con propiedades antimicrobianas (AMP) o no, guardando los péptidos predichos como antimicrobianos y descartando al resto. Posteriormente, todos aquellos péptidos predichos como antimicrobianos y provenientes de una misma secuencia, son agrupados si estos se traslapan entre si o son contiguos, creando agrupaciones de péptidos antimicrobianos, las cuales, finalmente son escritas en disco duro para análisis posteriores. Todos estos pasos se realizan explotando la arquitectura multi-núcleos de los procesadores actuales.

1.6. Organización de la tesis

El presente trabajo está organizado de la siguiente manera:

En el Capítulo 2 se exponen los conceptos biológicos básicos tales como péptidos, péptidos antimicrobianos, RNA-Seq y transcriptomas. Por otra parte, se abordan conceptos computacionales básicos para la comprensión del problema tratado en éste trabajo.

En el Capítulo 3 se define el problema a resolver en el presente trabajo. Posteriormente, se presentan estructuras de datos para abordar la problemática planteada. Finalmente, se presenta la propuesta de solución a la problemática presentada.

En el Capítulo 4 se presentan los experimentos y resultados obtenidos con la solución propuesta. Seguidamente se hace una discusión de los mismos.

En el Capítulo 5 se exponen las conclusiones a las que se llegaron, así como algunas propuestas para la continuación de éste trabajo de investigación.

Capítulo 2. Marco teórico

2.1. Conceptos biológicos

2.1.1. Péptidos

Los péptidos son polímeros formados por secuencias de aminoácidos unidos por enlaces peptídicos. La distinción entre péptido y proteína no se encuentra bien definida, por ejemplo, de acuerdo con Banga (2015), un péptido normalmente contiene menos de 20 aminoácidos, mientras que una proteína contiene 50 o más aminoácidos, sin embargo, de acuerdo con Wang (2010), un péptido contiene menos de 100 aminoácidos.

2.1.2. Péptidos antimicrobianos (AMPs)

Los péptidos antimicrobianos (AMPs) son péptidos con actividades antimicrobianas que se encuentran en prácticamente todos los organismos como plantas, hongos y animales, incluyendo organismos unicelulares como bacterias. De acuerdo con Rivas-Santiago *et al.* (2006), los AMPs normalmente tienen carga positiva y una longitud entre 15 y 45 aminoácidos. Muchos de ellos presentan una gran variedad de actividades contra bacterias, hongos, parásitos y virus, llegando a ser de gran interés como alternativa para antibióticos (Castañeda-Casimiro *et al.*, 2009; Rivas-Santiago *et al.*, 2006). Los AMPs naturales, normalmente son traducidos como proteínas precursoras más largas, generando el AMP activo por medio de la proteólisis de las mismas (Banga, 2015). Por ejemplo, durante la fermentación del vino, la levadura *Saccharomyces cerevisiae* genera varios AMPs mediante la proteólisis de su proteína "GAPDH", causando la muerte de otras levaduras y bacterias que también se encuentran involucradas en la fermentación del vino (Branco *et al.*, 2014).

A pesar de diferirse y de agruparse de acuerdo a su tamaño, secuencia de aminoácidos y estructura, estos comparten dos propiedades principales: 1) tienen carga positiva, y 2) aproximadamente el 50% de sus aminoácidos son hidrofóbicos (Rivas-Santiago *et al.*, 2006).

Al principio se pensaba que los AMPs tenían únicamente propiedades antimicrobianas, sin embargo, se ha descubierto que también tienen propiedades inmunomodula-

doras, anticancerígenas, entre otras (Banga, 2015; Castañeda-Casimiro *et al.*, 2009). A diferencia de los antibióticos, los cuales se dirigen a proteínas específicas de bacterias, los AMPs tienen diferentes objetivos y mecanismos de acción, haciéndolos menos propensos a imponer presión selectiva sobre las bacterias, haciéndolos una alternativa viable ante los antibióticos convencionales (Banga, 2015).

2.1.3. Transcriptomas

Al conjunto de todos los transcritos presentes (genes expresados) en una muestra biológica, para una condición específica, se le denomina transcriptoma (Wang *et al.*, 2009; Sung, 2017; Stadtländer, 2017). Hay dos tipos de transcritos: transcritos codificantes (ARN mensajero), que codifican a una proteína, y no codificantes (ARN no codificante), los cuales no codifican a una proteína. Los transcritos no codificantes pueden ser categorizados en ARN ribosomal (rRNA), ARN de transferencia (tRNA), ARN corto no codificante (short ncRNA) y ARN largo no codificante (long ncRNA).

En células procariotas, la codificación de una proteína normalmente se realiza después de la transcripción, debido a que regularmente los genes codificantes a una proteína se encuentran de manera contigua. Mientras que en la mayoría de las células eucariotas, las regiones codificantes (exones), se encuentran intercaladas por regiones no codificantes (intrones), por lo que el proceso de codificación de una proteína es aún más complejo en este tipo de células. Durante la transcripción, tanto exones como intrones son transcritos, conformando el pre-ARN mensajero, posteriormente, un juego diferente de intrones son removidos, generando diferentes variantes del transcrito, ARN mensajeros. Finalmente, los ARN mensajeros son traducidos para generar una proteína. Al proceso de remover diferentes juegos de intrones, se conoce como *alternative splicing*, y a las diferentes variantes del transcrito se les conoce como *isoformas* del gen. Dependiendo de la condición en que se encuentre la célula, y del gen, va a depender el número de transcritos a traducirse (Stadtländer, 2017).

Debido a que se pensaba que el ARN mensajero conformaba la mayor parte del transcriptoma, los estudios de transcriptomas se enfocaban en estos. Sin embargo, debido a los avances de la transcriptómica y al descubrimiento de ARN no codificante, se sabe que el ARN mensajero en realidad conforma una pequeña proporción del

transcriptoma (Sung, 2017). Al rededor del 5% del ARN es ARN mensajero, y entre el 85% y 90% es ARN ribosomal. Hoy en día se sabe que una gran cantidad de ARN no codificantes juegan un papel muy importante en la regulación de genes codificantes, por ejemplo, algunos ARN no codificantes pueden silenciar la expresión de un gen mediante diversos mecanismos como la inhibición de la transcripción o traducción del mismo (Stadtländer, 2017).

A la generación de un transcriptoma se le conoce como “*ensamblado de transcriptoma*”. Existen principalmente dos estrategias para dicho ensamblado: basada en un genoma de referencia y sin genoma de referencia (también conocida como “*de novo*”), de las cuales, surge una última “híbrida” como combinación de las dos (Martin y Wang, 2011).

Para el ensamblado de transcriptomas se requiere equipo de cómputo de altas prestaciones y *software* especializado. Existe una gran variedad de *software* para los ensamblados de transcriptomas basados en referencia y sin referencia. Para el ensamblado basado en referencia, la *suite* más usada es “*cufflinks*” (Trapnell *et al.*, 2012) y su sucesora “*HISAT*” (Pertea *et al.*, 2016), mientras que para el enfoque *de novo*, existen diferentes *softwares-suites* *Oases* (Schulz *et al.*, 2012) y *Trinity* (Grabherr *et al.*, 2011a; Haas *et al.*, 2013; Grabherr *et al.*, 2011b), por mencionar algunos, siendo *Trinity* el más usado. Éste trabajo se limitará a transcriptomas *de novo* a través de *Trinity*, ya que no se cuenta con genoma de referencia para nuestro caso de estudio.

2.1.4. RNA-Seq

RNA-Seq (RNA-Sequencing) es un conjunto de técnicas computacionales y experimentales que hacen uso de tecnologías de secuenciación, con el objetivo de determinar la estructura primaria y la abundancia de los transcritos de un transcriptoma (Martin y Wang, 2011; Korpelainen *et al.*, 2014).

La mayoría de las aplicaciones de RNA-Seq se dividen en dos:

- **Secuenciado de transcriptoma:** Tiene como objetivo caracterizar el transcriptoma. Involucra el secuenciado de todos los transcritos. Permite poder determinar los niveles globales de expresión de cada transcrito, identificar exones e intrones

y sus límites, sitios poly-A y de inicio de transcripción (TSS, Transcriptional Start Sites).

- **Secuenciado de ARN mensajeros (mRNA-Seq):** Involucra el secuenciado de los transcritos poliadenilados (poly-A) del transcriptoma (ARN mensajero). Permite medir las diferencias de expresión, isoformas, entre otras.

El objetivo de muchos estudios de RNA-Seq es encontrar genes o transcritos que cambien entre condiciones experimentales. Este cambio se conoce como expresión diferencial. RNA-Seq es la técnica empleada para estudios de expresión de genes que sucede a los microarreglos, los cuales tienen la limitante de cuantificar únicamente a genes conocidos (Van Verk *et al.*, 2013), al basarse en un genoma o transcriptoma de referencia (Hrdlickova *et al.*, 2017). Esto último limita el descubrimiento de nuevos genes o isoformas, aspecto que sí es posible con RNA-Seq (Korpelainen *et al.*, 2014; Stadtländer, 2017; Sung, 2017). Durante los últimos años, un número de tecnologías de secuenciación se han ido desarrollando, en éste trabajo nos limitaremos a la segunda generación (de las tres existentes), también conocida como secuenciación de siguiente generación, NGS (**N**ext **G**eneration **S**equencing). Existen diferentes plataformas de secuenciación, dentro de las más conocidas se encuentran: Illumina, Oxford Nanopore, ThermoFisher Scientific, entre otras.

Un experimento de RNA-Seq involucra la creación de una colección de fragmentos de ADN complementario (cDNA), creados a través de la transcripción reversa del ARN obtenido de la muestra biológica a analizar. A dicha colección se le conoce como librería. Posteriormente, los fragmentos de ADN complementario son marcados con secuencias de nucleótidos conocidas como adaptadores. Finalmente, la librería es procesada en alguna plataforma de secuenciación, arrojando millones de secuencias cortas, producto de las lecturas realizadas a los fragmentos de cDNA. El flujo de trabajo de un experimento de RNA-Seq consiste de los siguientes pasos:

1. **Extracción de ARN:** Una vez obtenida la muestra biológica, se procede a extraer todo el ARN. Este paso involucra dos procedimientos: 1) Aislamiento del ARN, desechando las demás moléculas y 2) Análisis de la calidad del ARN. Para ello, existen diferentes *kits* y plataformas.

2. **Preparación de librería:** Debido a la limitante del tamaño del fragmento de ARN que pueden manejar las plataformas de secuenciación, y por cuestiones de compatibilidad, las moléculas de ARN tienen que ser fragmentadas en piezas más pequeñas (Wang *et al.*, 2009; Hrdlickova *et al.*, 2017; Stadtländer, 2017). Dicha fragmentación se puede realizar de dos formas: 1) directamente a la molécula de ARN, antes de generar el ADN complementario ó 2) directamente al ADN complementario originado de la transcripción reversa. Una vez que se tienen los fragmentos, se les agregan unas secuencias de nucleótidos fijas llamadas adaptadores, en ambos extremos. Estos adaptadores son específicos de la plataforma de secuenciación. Existen diferentes protocolos para la preparación de librerías, siendo “Illumina RNA ligation” y “dUTP second-strand” los que se han desempeñado mejor (Levin *et al.*, 2010).
3. **Secuenciación:** Se hace lectura de los fragmentos, llamados *reads*, de forma paralela y en el orden de millones. Estas lecturas pueden obtenerse por un solo extremo del fragmento (single-end) o por ambos extremos (paired-end).
4. **Análisis bioinformático:** Finalmente, se procesan las lecturas, *reads*, por medio de herramientas bioinformáticas.

El producto final del secuenciado, son las lecturas de los fragmentos de la muestra biológica, que normalmente se encuentran en formato FASTQ.

2.1.5. FastQ

Las plataformas de secuenciación guardan las señales sin procesar en un formato crudo, independiente de la plataforma, y a las lecturas procesadas, en formato FASTQ (Stadtländer, 2017). *FASTQ* es el formato empleado para guardar los *reads*, esto es, las lecturas realizadas de todos aquellos fragmentos de una corrida en alguna de las plataformas de secuenciación. Las lecturas están representadas en formato de texto, donde cada lectura (read) está conformada por cuatro líneas llamadas *spots*:

- **Línea 1:** También conocida como *define sequence (línea de definición de la secuencia)* que es la que describe la secuencia.

- **Línea 2:** La secuencia de bases nitrogenadas leídas por la plataforma de secuenciación.
- **Línea 3:** Usualmente contiene el símbolo + como un separador entre la segunda y la cuarta línea.
- **Línea 4:** La calidad de las bases de la línea 2 donde cada valor de calidad está codificada en un carácter del estándar ASCII (*Ver Figura 32*).

Dentro del formato *FASTQ* existen principalmente dos estándares para la codificación de la calidad de las bases: Phred33 y Phred64, utilizando los valores desde 0 hasta 41 para representar la calidad de las bases. En Phred33 se usa el símbolo número 33 del estándar ASCII (!) para representar la calidad más baja, mientras que en Phred64 se usa el símbolo número 64 del estándar ASCII (@) para representar la calidad más baja (*ver Figura 33*).

En el caso de que las lecturas de los fragmentos hayan sido por ambos extremos (paired-end), la plataforma de secuenciación proporcionará dos archivos como resultado de la lectura de los fragmentos. A las lecturas del extremo izquierdo, normalmente se le denomina como *left reads* o “/1 reads”, mientras que a las lecturas del extremo derecho, se les conoce como *right reads* o “/2 reads”.

2.1.6. Trinity

Trinity (Grabherr *et al.*, 2011a; Haas *et al.*, 2013; Grabherr *et al.*, 2011b) es uno de los ensambladores *de novo* de transcriptomas existentes que se caracteriza por ser eficiente, robusto, y de código abierto, además de contar con una gran documentación y soporte. A diferencia de otros ensambladores, Trinity se basa en el uso de grafos *de Bruijn* para el ensamblado de transcriptomas. Se encuentra compuesto por tres módulos: *Oruga (Inchworm)*, *Crisálida (Chrysalis)* y *Mariposa (Butterfly)*, que se ejecutan de forma consecutiva. A continuación se explica brevemente cada uno de los módulos y su papel en el proceso de ensamblado:

1. **Oruga:** Ensambla los *reads* en secuencias únicas de transcritos. Basado en un enfoque voraz para la búsqueda de posibles rutas en un grafo de *k*-meros, re-

cuperando únicamente un representante (el mejor) de un conjunto de posibles rutas alternativas y diferentes (*contigs*). Regresa una colección de *contigs* donde cada *k*-mero está representado una sola vez en los *contigs*. El valor de *k* es 35 por omisión.

2. **Crisálida:** Agrupa los *contigs* si estos comparten al menos un *k*-mero de longitud $k-1$ y si los *reads* abarcan la unión entre los *contigs*, construye un grafo de *Bruijn* por cada agrupación de *contigs*.
3. **Mariposa:** Recorre los grafos de *Bruijn* de *Crisálida* regresando una secuencia única por cada transcrito representado en el grafo.

Como único requisito, Trinity requiere los *reads* como datos de entrada para el ensamblado, los cuales son archivos en formato FASTQ (ver Sección 2.1.5). En el Anexo Anexo 5 - Uso de SLURM, Trinity y TransDecoder se muestra la manera de instalarlo y su uso en el cluster OMICA del CICESE, de forma paralela en un solo nodo, y distribuída en múltiples nodos.

2.2. Conceptos computacionales

2.2.1. Paralelización y concurrencia

Durante años, el rendimiento de los procesadores fue incrementando de manera acelerada. Dicho incremento permitió a los desarrolladores y usuarios de software, disfrutar una mejoría de rendimiento sin necesidad de hacer cambio alguno en el software. Dadas las problemáticas que se encontraron los fabricantes de procesadores, dicho incremento tuvo sus límites pronto, ocasionando que los fabricantes de procesadores vieran otra alternativa. Como consecuencia de ello, empezaron a enfocarse en arquitecturas de *hyperthreading* y multi-núcleos (Sutter, 2005). Dicho cambio tuvo consecuencias para los desarrolladores de software, debido a que el software desarrollado estaba diseñado para sistemas de un solo núcleo. Requiriendo la reestructuración del software para aprovechar el incremento del rendimiento ofrecido por los procesadores modernos a través de múltiples núcleos, dando inicio a la programación multi-hilos.

Cómputo paralelo, concurrente, y distribuído, son términos que a menudo se usan indistintamente, sin embargo, no son lo mismo. En este trabajo abordaremos los primeros dos:

- **Cómputo paralelo:** Cuando un problema de cómputo es resuelto en menos tiempo por medio de la partición del mismo en particiones que son computadas simultáneamente. Por ejemplo, convertir un arreglo de caracteres, de minúsculas a mayúsculas simultáneamente por medio de múltiples hilos, cada hilo puede procesar un carácter o un grupo de estos.
- **Cómputo concurrente:** Cuando en un programa, dos o más tareas pueden estar realizando progreso en cualquier instante. Por ejemplo, una aplicación con interfaz gráfica, un hilo puede estar atendiendo las solicitudes del usuario, como clics y navegación del menú, mientras otro hilo puede estar descargando archivos al mismo tiempo.

Los sistemas paralelos se han utilizado por años, y varias alternativas de arquitecturas de computadoras han sido propuestas y usadas. Clasificar dichas arquitecturas puede ser difícil debido a la gran cantidad de detalles a considerar, como el número y complejidad de los elementos de procesamiento. Un modelo (*ver Figura 1*) que hace una clasificación de las arquitecturas es la taxonomía de Flynn (Flynn, 1966), en donde se caracterizan los sistemas paralelos de acuerdo al flujo de los datos e instrucciones que pueden manejar simultáneamente.

		Flujo de instrucciones	
		único	múltiple
Flujo de datos	único	SISD	MISD
	múltiple	SIMD	MIMD

Figura 1. Taxonomía de Flynn

En cómputo paralelo, hay dos enfoques para el particionamiento del problema original en problemas más pequeños: paralelismo de tareas y paralelismo de datos. En paralelismo de tareas, el problema principal se particiona en subproblemas más pequeños, los cuales se asignan entre los núcleos del procesador para ser computadas, mientras en paralelismo de datos, los datos son particionados entre los núcleos, donde cada núcleo realiza la misma (o similar) operación en su partición de datos asignada. En este trabajo abordaremos la programación paralela de datos del modelo SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata), también conocida como vectorización.

2.2.1.1. SIMD - Vectorización

Los sistemas SIMD son sistemas paralelos capaces de aplicar una sola instrucción en múltiples datos simultáneamente. Tales tipos de operaciones están disponibles en varias arquitecturas de procesadores modernos. La idea básica es aplicar la misma operación (aritmética) a múltiples datos sin dependencia entre estos. Para ello, un conjunto de instrucciones vectorizadas (**I**nstruction **S**et **A**rchitecture) son implementadas en los procesadores. Ejemplo de este tipo de operaciones son suma, multiplicación, resta, operaciones booleanas, entre otras. Dentro de los conjuntos de instrucciones más usados se encuentran SSE (SSE, SSE 2, SSE 3, SSE 4.1 y SSE 4.2) y AVX (AVX, AVX2 y recientemente AVX-512). Este tipo de cómputo tiene la ventaja de poder realizar el cómputo con varios operandos en una sola instrucción vectorizada, contrario a una instrucción escalar, en donde el cómputo se realiza con un operando a la vez.

La vectorización es el proceso para diseñar los programas de tal forma que hagan uso del conjunto de instrucciones disponible en el procesador con el objetivo de hacer que el programa corra más rápido. Esta se puede llevar a cabo de dos maneras:

- **Manual:** Requiere de la intervención del programador para que realice las modificaciones necesarias al código fuente. Estas pueden ser realizadas en lenguaje ensamblador ó por medio de funciones integradas (built-in) en el lenguaje de programación, esto es, funciones intrínsecas¹. Esto da mayor control del programa a un nivel bajo, aunque es difícil de implementarlo, dificultando el mantenimiento del programa.

¹<https://software.intel.com/sites/landingpage/IntrinsicsGuide>

- **Automática:** También conocida como “auto vectorización”, y es realizada por el compilador (GCC por ejemplo) o por la librería de ejecución (runtime library), Glibc por ejemplo. Da menor control sobre el programa, pero requiere poca o nula intervención del programador hacia el código fuente.

Dentro de las principales desventajas de la autovectorización se encuentran: 1) el compilador no siempre es capaz de vectorizar el código, 2) requiere de una versión reciente del compilador y 3) la vectorización realizada por el compilador no siempre es la óptima (Blair-Chappell y Stokes, 2012; Alaa Eltablawy, 2017).

En éste trabajo, se ha diseñado la implementación propia de la máquina de aprendizaje, así como InProt (*ver Sección 3.4.5*), de forma que pueda ser vectorizado por el compilador, aprovechando el conjunto de instrucciones más adecuado por el procesador.

2.2.2. Aprendizaje de máquina

El aprendizaje de máquina (Machine Learning) es una rama interdisciplinaria que surge de la combinación de ciencias de la computación, estadística, ingeniería, entre otras; siendo una herramienta que puede aplicarse en diferentes áreas (Harrington, 2012).

Los problemas de aprendizaje de máquina se pueden clasificar en dos categorías:

- **Aprendizaje supervisado:** El algoritmo es entrenado con un conjunto (conjunto de entrenamiento) de instancias con sus correspondientes características y sus etiquetas (clases), aprendiendo de este modo, una relación entre las características y las clases de las instancias. Posteriormente, el entrenamiento del algoritmo es probado con un conjunto (conjunto de prueba) de instancias sin sus etiquetas.
- **Aprendizaje no supervisado:** A diferencia del aprendizaje supervisado, en ésta última el conjunto de prueba carece de las etiquetas de las instancias, por lo que el algoritmo tendrá que “*aprender por si mismo*”.

En ambas categorías, el algoritmo es alimentado con conjuntos de instancias con

sus características (conjunto de entrenamiento y conjunto de prueba). Las características (o atributos) son mediciones individuales, como peso, color, entre otras, que juntas con otras características, conforman el conjunto de entrenamiento o prueba (Harrington, 2012).

2.2.2.1. Selección de características

El problema se centra en encontrar aquel conjunto de características (descriptores moleculares) que más favorezcan a los algoritmos de predicción en su tarea.

El algoritmo de selección de características recibirá un conjunto de características (descriptores moleculares) y dará como salida aquel subconjunto de este que más favorezcan al algoritmo de predicción, es decir, aquel subconjunto con el cual el algoritmo logre una mejor predicción.

Capítulo 3. Estructuras de datos y algoritmos para la extracción y conteo eficiente de k -meros en proteomas

3.1. Descripción general de la propuesta de solución

La solución propuesta en éste trabajo para el descubrimiento de péptidos antimicrobianos es a través de la extracción de k -meros de aminoácidos de diferentes tamaños, a partir de las secuencias que conforman el proteoma. Posteriormente, se realiza el cálculo de descriptores moleculares para cada k -mero, el cual se evalúa con una máquina de aprendizaje, con el objetivo de determinar si este tiene o no propiedades antimicrobianas. Un problema que se presenta con esta solución, es la selección de los algoritmos y estructuras de datos adecuados a utilizar para realizar dicho proceso de la manera más eficiente posible. La estructura de datos adecuada para almacenar todos los k -meros existentes, debe ayudar a descartar aquellos duplicados de tal forma que se eviten cálculos redundantes, logrando así, una reducción de los recursos computacionales necesarios. Tomando como ejemplo el proteoma del camarón (Ghafari *et al.*, 2014), el cual se encuentra conformado por un total de 18,694 secuencias, de las cuales 2,657 secuencias son repetidas pero con diferentes identificadores, dando un total de 16,037 secuencias únicas. Del mismo modo que con las secuencias del proteoma, hay k -meros repetidos entre diferentes secuencias del proteoma, o inclusive, dentro de una misma secuencia (*ver Figura 2*), con un total de 513,477,690 k -meros con una longitud entre 10 y 60 aminoácidos, de los cuales, 243,643,492 k -meros son únicos, esto es, aproximadamente el 53% del total de k -meros son repetidos.

Realizar las tareas previamente descritas requiere dos componentes principales: 1) una estructura de datos que represente los cientos o miles (incluso millones) de secuencias de la forma más compacta posible y que a su vez permita el procesamiento de las mismas de la forma más rápida posible, 2) un algoritmo de extracción que permita consultar la estructura de datos de la forma más eficiente posible, evitando tener que hacer consultas redundantes.

La obtención y contabilización de k -meros es un problema que se ha abordado desde hace unos años, y para el cual, se han propuesto varias soluciones (Deorowicz

```

>m.311
KVRVYVDCAIPANNKGKNAIGLMYWLLAREILYMRGAVTCNRQKPWDTMVDLFIYRDPEEIE
KDEQEQLQFQQQQQQQADAAAEYAGEGEDWNADGTAAAPAAGGADWNGT
>m.3429
QQQQQQQLANVGQAGVAHVHFSRGMVGNVGGVGNKQYSGFTAEVTYGVTGNLGEPE
VAGPSGVSGVVVVKQEPLYHDSANLAEYNQSTSKGHEILSQVYQQSPLPLKLLPVKPRKYP
NRPSKTPVHERPYACPVENCRRFRSDELTRHIRIHTGQKPFQCRICMRSFSRSDHLTTHI
RTHTGKPFSCDVC SRKFARSDEKRRHAKVHLKQKLLKVVSGSSPSATAPTATTSGVSL
SAAGTSTQQQQQPQAHLQAPGQQQQQQQQ
>m.4930
QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
QMMPQALQPQNNVPEFDLPSCFNIVGGLGPDATLMDMLEVADDQLAFESGLIGNSDIKAD
YGGKMDSKKKQKDQAVSSGQDVDELTKMISNVRVDDSRGNRNPQSQVDSMNQHPTQS
VDVAFKVAISAAECLQAYAATGDISLLLATHRYLLAVQNNQGD TALHTAVSNKNIEAFNKILK
ACEKIRPQDLLNAQNFARETALHQAVRGNETIMVRRLVAMPGCDVSIVDAQGNTPVHCAA
EMQSIQCLEALLTRPVNGVRSVAVTQAINVYNYQGETPLHLAVINGNLD SVRMLVDAGA QV
HLCERKRGANPLHLAVMHGHHEIARYLLDLTSVTIEAGLFDGNTALHLAAQQRDSEMCKIL
LRHNADPNAKNMLQKRKKLSESEEEEEEEISSRKVEEEEDDSEEEEPDCYTPLDYAGND
AEILSILRGEVIQEEEEEQGAQAQEEVVGTKAGPLHSALDSGIDISVTDIQYSDGPGEDDSV
GELSERVRERLASHLSGDTWRHLAQLLDLDYMPCLAGEPSPASTLLHPDNMKSVSLEKL
RECLTVLGLKECVAVLDQA

```

Figura 2. Repetición de un 4-mero (QQQQ) en tres secuencias del proteoma del camarón (Ghaffari *et al.*, 2014).

et al., 2015; Kokot *et al.*, 2017b; Marçais y Kingsford, 2011; Erbert *et al.*, 2017; Rizk *et al.*, 2013), siendo en su mayoría soluciones implementadas en paralelo para CPU, a excepción de (Erbert *et al.*, 2017), que también ofrece una implementación para la GPU con la tecnología CUDA. Cabe mencionar que las soluciones propuestas hasta la fecha trabajan únicamente para ADN, esto es, no abordan el problema de la obtención y contabilización eficiente de k -meros de secuencias de aminoácidos.

Dentro de los algoritmos más usados en la literatura y su correspondiente estructura de datos para dicha tarea en ADN (Deorowicz *et al.*, 2015; Kokot *et al.*, 2017b; Marçais y Kingsford, 2011; Erbert *et al.*, 2017; Rizk *et al.*, 2013) se encuentran principalmente los algoritmos de ordenamiento y tablas hash. En éste trabajo estudiamos tres estructuras de datos que fueron evaluadas para atacar el problema de obtención y contabilización de k -meros mencionado previamente: árbol de sufijos, un árbol con exactamente m hojas, teniendo sus aristas etiquetadas con subcadenas (no vacías) de la cadena C de longitud m (Gusfield, 1997); arreglo de sufijos, arreglo de números enteros enumerados del 1 al m , especificando el orden lexicográfico de los sufijos de la cadena C de longitud m (Gusfield, 1997); y tablas hash (ver Sección 3.3). En la Sec-

ción 3.4 presentamos nuestra propuesta de solución, la cual se basa en algoritmos de ordenamiento y un arreglo lineal como estructura de datos.

3.2. Motivación

Una de las problemáticas principales que se tiene al momento de analizar el proteoma para el descubrimiento de péptidos antimicrobianos, es el tiempo llevado para realizar dicho proceso, debido a que se tienen que realizar los siguientes pasos:

1. Extraer los k -meros únicos
2. Calcular los descriptores moleculares
3. Predecir la actividad biológica de los k -meros por medio de una máquina de aprendizaje.

En el primer paso, se deslizan ventanas de diferentes tamaños en todas las secuencias del proteoma a interrogar (*ver Figura 3*); a los fragmentos de aminoácidos cubiertos por las mismas (k -meros) se les calcula un conjunto de descriptores moleculares propuestos en (Verdugo, 2014) (*ver Figura 3*). Posteriormente, por medio de una máquina de aprendizaje se evalúa cada k -mero (*ver Figura 4*) para determinar si cuenta con propiedades antimicrobianas o no. Realizar el ventaneo en forma de “*fuerza bruta*” deja la posibilidad de contar con k -meros repetidos dentro de varias secuencias o incluso, dentro de una misma secuencia. En la Figura 2 se muestra un ejemplo de ambos casos, donde un mismo k -mero se encuentra repetido varias ocasiones dentro de una misma secuencia y a su vez, en el resto de las secuencias. Realizar los tres pasos mencionados anteriormente es una tarea que demanda una gran capacidad de cómputo, es por ello que se requiere de algoritmos y estructuras de datos que permitan realizarlos de una manera eficiente. A continuación se presenta una estructura de datos bien conocida, la cual sería la elección natural para realizar de manera eficiente el almacenamiento de k -meros, descartando los duplicados, y evaluar los k -meros únicos con una máquina de aprendizaje, preservando aquellos predichos como AMP, descartando aquellos predichos como no-AMP. Sin embargo, como mostraremos a continuación, esta estructura no es la más conveniente para nuestro problema

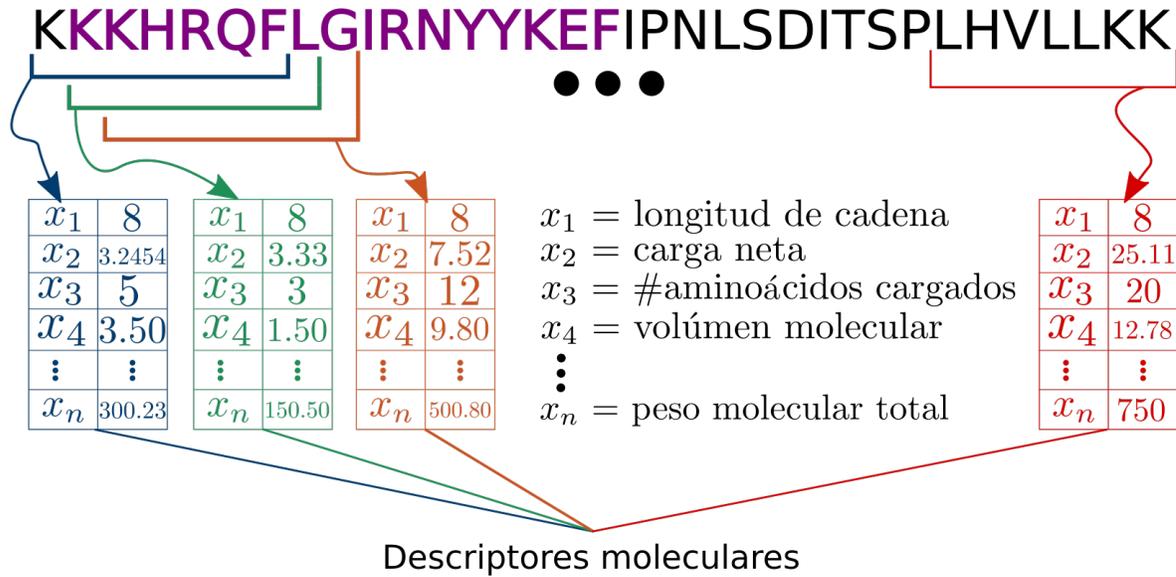


Figura 3. Representación de la extracción de diferentes k -meros y los valores de n descriptores moleculares.

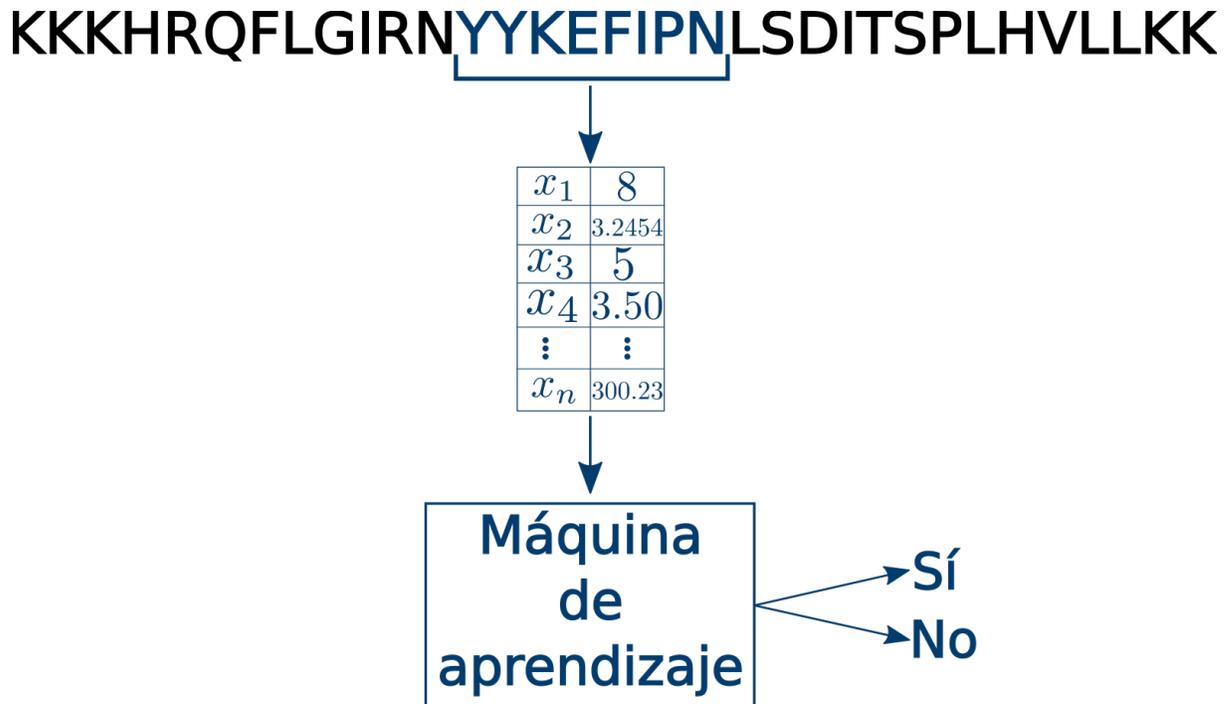


Figura 4. Representación de extracción de un k -mero y el cálculo de sus n descriptores moleculares y su posterior clasificación por medio de una SVM previamente entrenada

3.3. Tablas Hash

Las tablas Hash (Cormen, 2009) son estructuras de datos, las cuales son una implementación de arreglos asociativos o diccionarios (otra implementación son los árboles)

(Mehlhorn y Sanders, 2008), y por medio de los elementos contenidos en éstos, permiten hacer inserción de nuevos datos y búsqueda de los elementos almacenados en ellos de forma rápida, por ejemplo, usualmente la búsqueda se hace en tiempo constante.

El funcionamiento básico de *hashing* es transformar el identificador (k) de un dato de entrada (el elemento a insertar), e , del conjunto de elementos, S , en un valor entero (su llave), $h(k)$, que será una de las ranuras de la tabla hash T .

Debido a que el universo de posibles llaves U es mayor al tamaño de la tabla hash T , dos o más datos de entrada distintos serán asignados a la misma ranura de la tabla hash, a esta situación se le conoce como colisión (*ver Figura 5*). Existen técnicas para la resolución de las colisiones, entre ellas se encuentran: resolución de colisiones por encadenamiento, doble hashing y hashing perfecto; siendo hashing perfecto la técnica que se llegó a probar en este trabajo. En resolución de colisiones por encadenamiento, cada elemento de la tabla hash es una lista, de modo que cuando dos elementos son mapeados a la misma ranura (el primer nivel de la tabla hash), estos son almacenados en una lista (segundo nivel). En *doble hashing*, cuando un elemento es mapeado a una ranura ocupada, una segunda función hash es empleada para determinar la nueva ubicación. En *hashing perfecto*, el esquema es similar a *resolución de colisiones por encadenamiento*, la diferencia principal se encuentra en que cada *lista* es reemplazada por una segunda tabla hash.

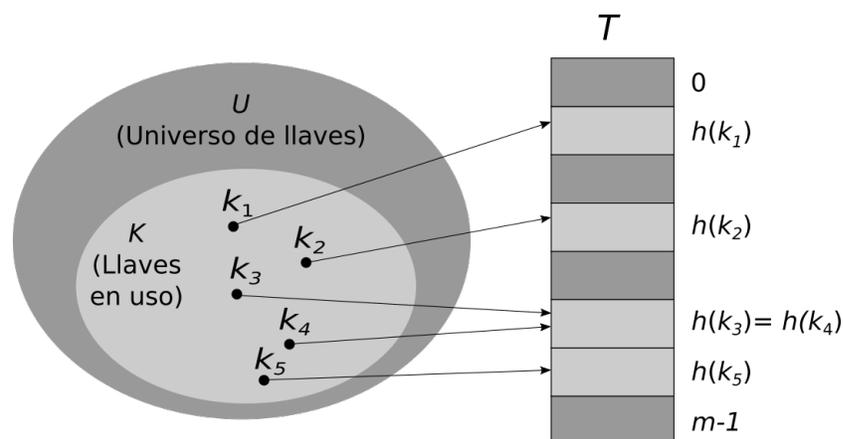


Figura 5. Representación de una función hash, h , mapeando las llaves a una tabla hash, T , donde las llaves k_3 y k_4 son mapeadas a la misma ranura de la tabla, generando una colisión. *Figura inspirada de (Cormen, 2009).*

La calidad de una función hash normalmente se determina por su calidad de gene-

rar llaves “únicas” para cada elemento distinto y distribuir las uniformemente dentro de las ranuras de la tabla hash.

Una característica importante en tablas hash es lo que se conoce como factor de carga. Si en promedio se desean insertar n elementos, en una tabla de tamaño m , el factor de carga está determinado por:

$$\alpha = \frac{n}{m}. \quad (1)$$

3.3.1. Función hash

Dentro de las funciones hash, existen dos categorías de las mismas de acuerdo a su seguridad: las no criptográficas y las criptográficas; siendo las no criptográficas las que se llegaron a probar en este trabajo. Las funciones hash no criptográficas normalmente son empleadas en aplicaciones genéricas (como un traductor de idiomas, compilador, etc.). Dentro de las funciones hash no criptográficas, estas mismas se encuentran categorizadas de acuerdo a su especialización en el tipo de datos de los elementos a insertar en la tabla hash. Esto es, que algunas se encuentran diseñadas especialmente para trabajar con cadenas de texto, mientras que otras se encuentran especializadas para valores numéricos. Por último, existen las funciones genéricas, las cuales trabajan con cualquier tipo de dato. Una de las funciones hash para cadenas de texto es la función de *hashcode* (Kernighan y Pike, 1999), la cual consiste en la sumatoria del producto del valor numérico de cada carácter (de la cadena de texto a insertar) por un multiplicador M , donde $M = \{31, 37\}$, finalmente, se utiliza el método de la división (Cormen, 2009) para determinar el índice (en la tabla hash) del elemento a insertar. Las funciones hash genéricas más usadas son Murmur2/3¹, la familia de funciones de Bob Jenkins², xxhash³ y Cityhash⁴.

¹<https://github.com/aappleby/smhasher>

²<http://www.burtleburtle.net/bob/hash/doobs.html>

³<https://github.com/Cyan4973/xxHash>

⁴<https://github.com/google/cityhash>

3.3.2. Hashing universal

Debido a que las funciones hash no criptográficas no están exentas de colisiones, un atacante puede seleccionar cuidadosamente el conjunto de elementos a insertar, S , pudiendo ocasionar colisiones y por ende, que el *software* tenga un comportamiento no deseado. Para manejar dicha situación, al momento de iniciar el *software*, se elige de forma aleatoria una función hash del conjunto de funciones hash, H , construido por funciones hash seleccionadas cuidadosamente, de modo que cada ejecución del *software* usará una función hash distinta, lo que ocasionará que las llaves generadas sean distintas entre las ejecuciones del mismo.

3.3.3. Hashing perfecto

Aunque el *hashing* puede dar buenos resultados para la mayoría de los casos, también puede proporcionar el peor rendimiento cuando el conjunto de elementos a insertar es estático. A la técnica de *hashing* que en el peor de los casos requiere tiempo constante, $O(1)$, para realizar una búsqueda, se le conoce como *hashing perfecto*. Esta técnica hace uso de *hashing universal* para cada nivel de la tabla hash, la cual consiste de dos niveles de *hashing*, donde cada nivel es una tabla hash con *hashing universal*. Para garantizar que en el segundo nivel de la tabla hash no ocurran colisiones, el tamaño de la segunda tabla hash para la ranura j , debe ser el cuadrado del número de elementos asignados a la ranura j .

3.3.3.1. Implicaciones

Dentro de las implicaciones encontradas a la hora de la implementación de las tablas hash (con hashing perfecto) se encuentran:

- **Determinación del tamaño de la tabla principal (primer nivel):** Determinar el tamaño de la tabla principal (primer nivel). Determinar el valor adecuado para el tamaño es una tarea no trivial (Cormen, 2009), por lo que para ello, se eligió un factor de carga de 8 para el total de k -meros posibles dentro del proteoma (ver Ecuación 3).

- **Determinación del tamaño de la llave (32 bits o 64 bits):** Determinar el tamaño (en bits) de la llaves a usar para los elementos a insertar, fue un proceso difícil debido a la poca documentación encontrada para realizar dicha decisión. Por lo que se llegaron a probar los dos tamaños: 32 y 64 bits para las funciones mencionadas anteriormente. En el caso de Spookyhash, que es una función que genera llaves de 128 bits, las llaves generadas se truncaban a 32 o 64 bits.
- **Determinación del índice de la tabla:** Una vez calculada la llave del elemento a insertar (k -mero), se requiere de una función (función de compresión) que calcule el índice dentro de la tabla en donde se insertará el elemento deseado, de modo que los elementos a insertar sean distribuidos uniformemente dentro de ésta. Se eligió la siguiente función (Cormen, 2009):

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m), \quad (2)$$

donde p es un número primo lo suficientemente grande de modo que cada llave k se encuentre dentro del intervalo $[0, p - 1]$, a un número dentro del conjunto $\{1, 2, \dots, p\}$ y b un número dentro del conjunto $\{0, 1, \dots, p - 1\}$.

- **Determinación de la función hash:** Seleccionar (o diseñar) una función hash que genere una llave (valor hash) “única” de modo que para cualquier par de k -meros distintos, genere dos llaves iguales con la probabilidad más baja posible, no es un proceso trivial. Para esto, se probaron las funciones hash no criptográficas mencionadas anteriormente y se implementó la función de *hashcode* (Kernighan y Pike, 1999) y el algoritmos *radix₂* como funciones hash alternas. El problema con las funciones implementadas (*hashcode* y *radix₂*) radica en que el cálculo sobrepasa las limitantes de los tipos de datos primitivos del lenguaje C++ usados (entero sin signo de 32 y 64 bits), por lo que ocasionaba que al sobrepasar la capacidad de la variable impuesta por la limitante, esta tiende a “reiniciarse”, comportamiento definido (en el estándar de C++) para los enteros sin signo (Prata, 2011). En cambio, cuando se utilizaban una de las funciones no criptográficas para llaves de 32 o 64 bits, la llave estaba dentro de los valores aceptados por los tipos de datos de C++, pero al realizar el cálculo para determinar el índice de la tabla hash, sucedía lo mencionado anteriormente (reseteo de la variable), lo que dificultaba aún más el cálculo mencionado abajo.

Tabla 1. k -meros y su $Radix_2$

k-mero	$Radix_2^a$	$Radix_2$ (uint8) ^b
GHTEL	2,262 (1136, 1712, 2048, 2186, 2262)	76 (112, 64, 80, 138, 76)
HTELK	2,327 (1152, 1824, 2100, 2252, 2327)	75 (128, 160, 20, 152, 75)
TELKA	2,415 (1344, 1896, 2200, 2350, 2415)	65 (64, 40, 48, 150, 65)
ELKAV	2,228 (1104, 1712, 2012, 2142, 2228)	86 (80, 96, 44, 130, 86)
LKAVT	2,332 (1216, 1816, 2076, 2248, 2332)	84 (192, 88, 4, 172, 84)

^a $Radix_2$ del k -mero.

^b $Radix_2$ utilizando un entero sin signo de 8 bits

Los valores entre paréntesis representan el valor de $Radix_2$ durante el cálculo

- Determinación de las constantes a , b y p** (ver Ecuación 2): Esta fue la mayor limitante de todas, siendo en realidad una limitante tecnológica. Dado que los números a y b son números elegidos al azar, el valor más grande de estas constantes es una unidad menor a p , por lo que al realizar la operación para determinar el índice dentro de la tabla hash, se presenta el comportamiento de la limitante por los tipos de datos de C++, mencionada en el punto anterior.

Suponiendo que contamos con una computadora de 8 bits, al manejar enteros en su variante sin signo, tendríamos un total de $2^8 - 1 = 256$ valores en el intervalo $[0,255]$. En la Tabla 1 se muestran cinco k -meros con sus respectivos valores de $Radix_2$ y el valor resultante utilizando un entero sin signo de 8 bits. En la primera columna se muestra el 5-mero, en la segunda columna se muestra el valor en $Radix_2$, mientras que en la tercera columna, el primer número representa el valor de $Radix_2$ calculado con un entero sin signo utilizando 8 bits, y los números en paréntesis, representan los valores obtenidos durante el proceso, donde se observa cómo dicho valor va incrementando y decrementando, esto es debido a la naturaleza de los enteros sin signo, donde una vez que se sobrepasa de su valor máximo, este se “reinicia”. Por ejemplo, el cálculo de $Radix_2$ del 5-mero de la primera fila, “GHTEL”, sería de la siguiente manera: $71(G) * 2^4 + 72(H) * 2^3 + 84(T) * 2^2 + 69(E) * 2^1 + 76(L) * 2^0 = 1,336 + 576 + 336 + 138 + 76 = 2,262$, pero debido a que desde el primer valor obtenido, 1,336 sobrepasa el rango de posibles valores para un entero de 8 bits, 255, éste se “resetea” al pasarse dicho límite, ocasionando que empiece desde cero nuevamente. Los valores 71, 72, 84, 69 y 76 del cálculo, es la representación numérica de los caracteres que representan el 5-mero (ver Figura 32).

Además de las tablas hash, también se probaron árboles de sufijos y arreglos de

sufijos con el uso de la librería de C++, SeqAn (Reinert *et al.*, 2017), encontrándose con la problemática principal de que el peso de los índices generados era de aproximadamente 200 MB para el proteoma del camarón (Ghaffari *et al.*, 2014), el cual tiene un peso aproximado de 13 MB, generando archivos de gran tamaño para el preprocesado del proteoma del camarón.

3.4. Propuesta de solución

Dado los inconvenientes obtenidos con las estructuras de datos mencionadas en las secciones anteriores, como propuesta de solución, se tiene la extracción de k -meros en forma de “*fuera bruta*” de tal forma que, se hace un deslizamiento de ventana a partir del inicio de cada secuencia del conjunto de secuencias (proteoma), la cual tiene un tamaño inicial definido, k , y que se va recorriendo una posición posterior por vez a lo largo de la secuencia, guardándose en un arreglo lineal (*ver Algoritmo 1*). Posteriormente, se realiza la eliminación de los k -meros duplicados por medio de un ordenamiento de los mismos y su posterior comparación. En el Algoritmo 2 se muestra el pseudocódigo propuesto para la extracción de k -meros únicos.

El total de k -meros presentes en una secuencia s de longitud l , está dado por:

$$\text{total } k\text{meros} = l - k + 1. \quad (3)$$

Esto es, en una secuencia s de 100 aminoácidos de longitud, resultarían un total de $100 - 10 + 1 = 91$ k -meros de longitud 10.

3.4.1. Ventaneo del proteoma

El ventaneo del proteoma para la extracción de los k -meros se realiza de la siguiente manera: se “*posiciona*” la ventana (de tamaño k) al inicio de una secuencia determinada; el fragmento de aminoácidos cubiertos por la ventana, conformaría el primer k -mero. Posteriormente, se recorre la ventana un aminoácido hacia la derecha (siempre y cuando la ventana esté cubierta por completo); el fragmento de aminoá-

cidos cubiertos por la ventana, conformaría el segundo k -mero de la secuencia. Esta operación se realiza de forma similar para el resto de las secuencias y para los diferentes tamaños de k -meros. En el Algoritmo 1 se muestra el pseudocódigo para la extracción de los k -meros totales por medio del ventaneo del proteoma.

A pesar de haber usado “ *fuerza bruta* ”, el tiempo de cómputo necesario para la extracción de los k -meros del proteoma del camarón (Ghaffari *et al.*, 2014), es relativamente bajo (ver Figura 6).

3.4.2. Ordenamiento de k -meros

Una vez extraídos todos los k -meros del proteoma del camarón (Ghaffari *et al.*, 2014), hace falta deshacerse de aquellos k -meros duplicados. Una de las técnicas empleadas es por medio del ordenamiento lexicográfico de los mismos (Deorowicz *et al.*, 2015; Kokot *et al.*, 2017b).

El algoritmo de ordenamiento empleado en este trabajo es el algoritmo implementado por omisión en *Intel TBB*, **QuickSort** (Cormen, 2009).

A pesar de que QuickSort no es el algoritmo más rápido para ordenar k -meros reportado en la literatura actualmente (Kokot *et al.*, 2017a), como se muestra al ser aplicado para ordenar k -meros de nucleótidos (Kokot *et al.*, 2017b), en nuestros experimentos con aminoácidos, éste se ha desempeñado bien dando tiempos aceptables (ver Figura 6).

3.4.3. Comparación de k -meros para remover duplicados

Una vez que se tienen los k -meros ordenados, realizar la comparación de los mismos, para la eliminación de los duplicados, de manera general, consiste en un recorrido del arreglo de los k -meros e irlos comparando lexicográficamente por pares. El Algoritmo 3 muestra el pseudocódigo para dicho proceso. En la Figura 6 se muestra el tiempo de cómputo llevado a cabo para la extracción, ordenamiento y eliminación de k -meros duplicados del proteoma del camarón para $k = 10$ hasta $k = 60$ con diferentes números de hilos y compiladores.

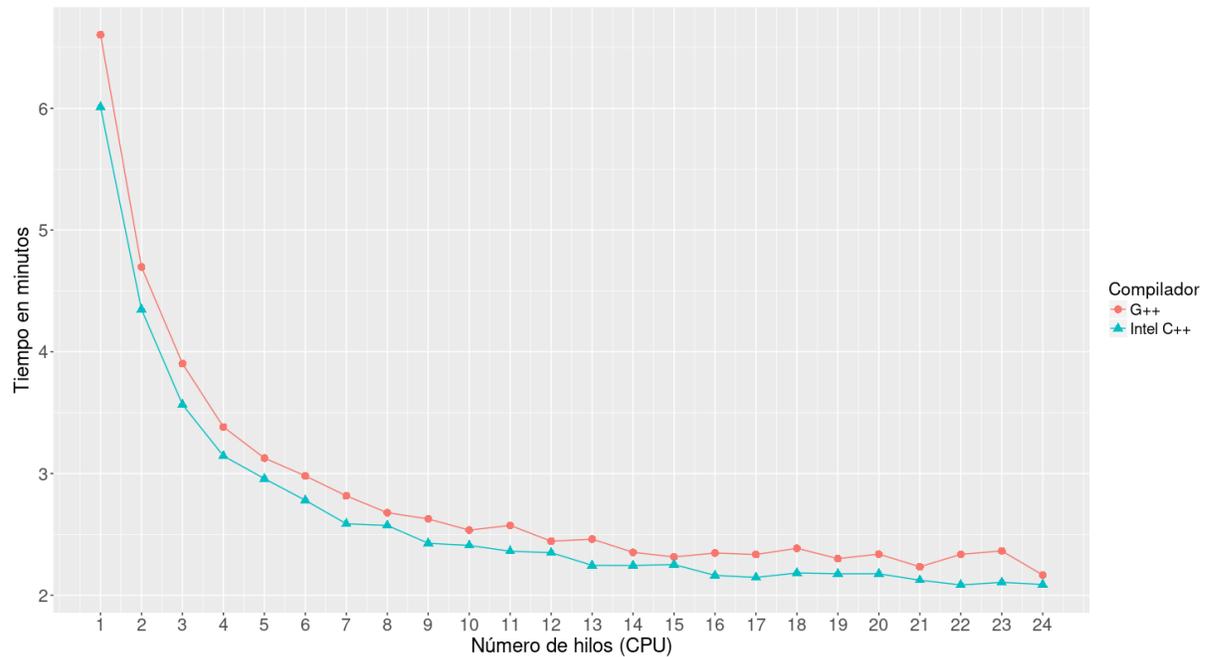


Figura 6. Tiempo de cómputo para la extracción, ordenamiento y eliminación de k -meros duplicados del proteoma del camarón, para $k = 10$ hasta $k = 60$, con diferentes números de hilos y compiladores. La extracción se realizó en un nodo del cluster OMICA del CICESE (ver Capítulo 4).

Algoritmo 1: Extracción de k -meros totales

```

Entrada: proteoma // Proteoma a interrogar
            k // Tamaño de los  $k$ -meros a extraer

Salida :  $k$ -meros // Arreglo de  $k$ -meros totales
1 inicio
2    $kmeros \leftarrow RESERVAR()$  // reservar espacio
3    $totalseqs \leftarrow length(proteoma)$  // total de secuencias
4   para  $i = 1$  a  $totalseqs$  hacer
5      $seq \leftarrow proteoma[i]$  // secuencia actual
6      $seqlen \leftarrow length(seq)$ 
7     si  $k \leq seqlen$  entonces // Si contiene al menos un  $k$ -mero
8        $ultimo \leftarrow seqlen - k + 1$ 
9       para  $j = 1$  a  $ultimo$  hacer
10         $kmeros.insertar(seq[j : j + k])$ 
11         $j \leftarrow j + 1$ 
12      fin
13    fin
14  fin
15 fin

```

Algoritmo 2: Extracción de k -meros únicos

```

Entrada: proteoma // Proteoma a interrogar
          kmin // Tamaño mínimo de  $k$ -mero ( $kmin \geq 10$  y  $kmin < kmax$ )
          kmax // Tamaño máximo de  $k$ -mero ( $kmax > kmin$  y  $kmax \leq 60$ )

Salida : k-meros // Arreglo de  $k$ -meros únicos
1 inicio
2    $kmeros \leftarrow RESERVAR(kmin, kmax)$  // reservar espacio
3   para  $i = kmin$  a  $kmax$  hacer
4      $AGREGAR(kmeros, i, proteoma)$  // Agrega los  $i$ -meros totales
     // ordena los  $i$ -meros por posición de inicio y
     // almacena en último el índice
     // del último  $i$ -mero único
5      $ORDENAR(i, proteoma)$  // ordena los  $i$ -meros
6      $ultimo \leftarrow COMPARAR(kmeros, i)$ 
     // Elimina los  $i$ -meros duplicados
      $ELIMINAR\_DUPLICADOS(kmeros, i, ultimo)$ 
7   fin
8 fin

```

Algoritmo 3: Comparación de k -meros

```

Entrada: k-meros //  $k$ -meros a comprar
1 inicio
2    $total \leftarrow length(kmeros)$  // Total de  $k$ -meros
3   para  $j = 0, i = 1$  a  $i < total$  hacer
4     si  $kmeros[i] \neq kmeros[j]$  entonces // Comparación lexicográfica
5        $j \leftarrow j + 1$ 
6        $kmeros[j] = kmeros[i]$ 
7     fin
8   fin
9 fin

```

3.4.4. Algoritmo propuesto

En esta sección se describe de forma detallada el algoritmo propuesto para la extracción de k -meros únicos, cálculo de 51 descriptores moleculares, evaluación de los

k -meros únicos con una máquina de aprendizaje, agrupamiento de k -meros predichos como AMP provenientes de la misma secuencia del proteoma de la cual fueron extraídos (podado del proteoma), y escritura de los k -meros agrupados. En la Figura 7 se muestra el flujo de los pasos llevados a cabo por el algoritmo propuesto.

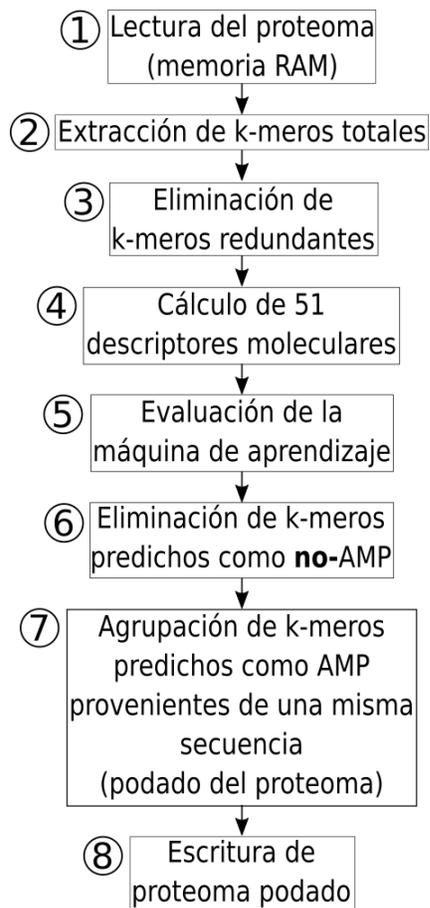


Figura 7. Pasos llevados a cabo por el algoritmo propuesto.

3.4.4.1. Lectura del proteoma

En este proceso, como primer paso de los ocho (*ver Figura 7*) se hace lectura de las secuencias del proteoma, una por una. Al mismo tiempo se valida que estas contengan únicamente aminoácidos estándares, esto último por la limitante de los descriptores moleculares, los cuales trabajan únicamente con aminoácidos estándares. Las secuencias leídas se cargan en memoria RAM.

3.4.4.2. Extracción de k -meros totales y eliminación de k -meros redundantes

Una vez que se tienen cargadas en memoria RAM las secuencias únicas del proteoma, como segundo y tercer pasos del algoritmo propuesto (*ver Figura 7*), se realiza la extracción de los k -meros totales (*ver Algoritmo 1*) para posteriormente, eliminar aquellos k -meros redundantes por medio de un ordenamiento y comparación lexicográfico (*ver Algoritmo 2*). Esto se hace con el motivo de evitar el cálculo redundante de los 51 descriptores moleculares, debido a que una gran mayoría de estos k -meros son duplicados (*ver Figura 35*).

3.4.4.3. Cálculo de descriptores moleculares

Habiendo removido los k -meros duplicados y dejando los k -meros únicos en el segundo y tercer pasos, respectivamente, como cuarto paso (*ver Figura 7*) se realiza el cálculo de los 51 descriptores moleculares. En la Tabla 13 se muestran los 51 descriptores calculados para cada k -mero.

3.4.4.4. Evaluación y depuración de k -meros predichos como no-AMP

Una vez calculados los 51 descriptores moleculares a cada k -mero, como quinto y sexto pasos (*ver Figura 7*), se procede a escalar los descriptores moleculares, evaluar los descriptores moleculares escalados con la máquina de aprendizaje y finalmente, aquellos k -meros predichos como no-AMP, se descartan, manteniendo aquellos k -meros predichos como AMP por la máquina de aprendizaje.

3.4.4.5. Agrupación de k -meros predichos como AMP provenientes de una misma secuencia (podado de proteoma)

Como séptimo paso (*ver Figura 7*), una vez que se tienen los k -meros predichos como AMP, para cada secuencia, se procede a obtener todos los k -meros provenientes de dicha secuencia para agrupar aquellos k -meros que se intersectan o se conectan, es decir, k -meros que se traslapen o se encuentren contiguos. Como paso previo al

agrupamiento de los k -meros de una misma secuencia que se intersectan o se conectan, estos se ordenan por posición de inicio dentro de la secuencia. En el Algoritmo 4 se muestra el pseudocódigo que explica cómo realizar dicha agrupación de k -meros (podado del proteoma).

3.4.4.6. Escritura de proteoma podado

Como último paso (*ver Figura 7*), se escriben todos aquellos k -meros “agrupados” que presenten traslapes o sean contiguos.

3.4.5. InProt - *In silico* Proteolysis

In silico Proteolysis es una implementación en lenguaje C++ 14 del algoritmo propuesto, el cual hace uso de la librería Intel TBB (Reinders, 2007) para la paralelización de los pasos descritos anteriormente (*ver Figura 7*). En el Anexo 3 - InProt - *In silico* Proteolysis se muestran ejemplos de uso de dicha implementación.

3.4.6. CAMPRED - CAMP AMP PREDiction

Con el objetivo de automatizar la prueba de control realizada en la Sección 4.4, se programó CAMPRED (**CAMP AMP PRED**iction), una herramienta desarrollada en lenguaje de programación Go (Doxsey, 2016; Kennedy *et al.*, 2015) que automatiza de forma concurrente el proceso de predicción de unas secuencias dadas con los algoritmos de CAMP (SVM, ANN, RF, DA), extrayendo únicamente aquellas secuencias predichas como AMP por todos los algoritmos especificados por el usuario. Los pasos realizados por CAMPRED son los siguientes:

- **Particionamiento de las secuencias:** Las secuencias se pueden subir todas juntas ó se pueden particionar por conjuntos de “ n ” secuencias, esto es debido a que CAMP cuenta con una cota de tiempo o de tamaño de archivo a procesar, originando que muchas veces la respuesta recibida se encuentre incompleta o incluso vacía.

- **Predicción de las secuencias:** Se mandan a predecir las secuencias con los algoritmos especificados por el usuario. Para ello, de forma concurrente se suben de dos en dos los conjuntos de “n” secuencias (o las secuencias completas en caso de no ser particionadas) por medio de una petición HTTP POST al servidor de CAMP.
- **Procesado de la respuesta de la petición HTTP POST:** Se procesa la respuesta de la petición HTTP POST recibida por el servidor, guardando únicamente aquellas secuencias predichas como AMP por todos los algoritmos especificados por el usuario, sin importar el porcentaje de precisión (para el caso de SVM, RF y DA). Posteriormente, si se especifica por el usuario, como paso opcional, de forma concurrente se escribe en el disco duro la respuesta de la petición HTTP POST.
- **Extracción de secuencias:** Una vez guardados los índices de aquellas secuencias predichas como AMP por todos los algoritmos especificados por el usuario, estos se extraen del archivo original, según los índices regresados por CAMP en los archivos creados en la partición previamente explicada. Esto es debido a que en caso de particionar 100 secuencias en conjuntos de 10 secuencias cada uno, CAMP reconocerá cada secuencia de cada conjunto de 10, como las secuencias de la 1 a la 10, por lo que la primera secuencia del segundo conjunto llegaría a representar la secuencia 11 del conjunto original de secuencias.

Algoritmo 4: Agrupación de k -meros (podado del proteoma)

```

Entrada: kmeros // k-meros predichos como AMP
Salida : grupos // Grupos de k-meros que intersectan o conectan
1 inicio
2 ORDENAR(kmeros) // Ordenar por posición de inicio
3  $k_{\text{primero}} \leftarrow k_{\text{meros}}[0]$ 
    $k_{\text{ultimo}} \leftarrow k_{\text{meros}}[0]$ 
   // Arreglo de par de enteros, (inicio,fin)
   // donde se guardarán la posición de inicio
   // y la posición de terminación de aquellos grupos de k-meros
   // que intersecten o se conecten
    $\text{grupos} \leftarrow \text{RESERVAR}()$  // Reservar espacio para el arreglo
4  $\text{total} \leftarrow \text{LENGTH}(k_{\text{meros}})$  // Total de k-meros
5  $\text{kgc} \leftarrow \text{par}(\text{inicio}, \text{fin})$  // Posiciones del k-mero actual
6  $j \leftarrow 1$ 
7 mientras j a total hacer
8 |  $\text{kgc.ultimo} \leftarrow k_{\text{primero.fin}}$  // Se guarda la posición final
9 |  $\text{kj} \leftarrow k_{\text{meros}}[j]$  // Se obtiene el k-meros en la posición j
10 | si !kj.DENTRO(kgc)5 entonces
11 | | si kj.CONECTADO(kgc) || kj.INTERSECTA(kgc)6,7 entonces
12 | | |  $\text{klast} \leftarrow \text{kj}$ 
13 | | en otro caso
14 | | |  $\text{grupos.INSERTAR}(\text{kgc.inicio}, \text{kgc.fin})$ 
15 | | |  $k_{\text{primero}} \leftarrow \text{kj}$ 
16 | | |  $k_{\text{ultimo}} \leftarrow \text{kj}$ 
17 | | |  $\text{kgc.inicio} \leftarrow \text{kj.inicio}$ 
18 | | fin
19 | fin
20 | |  $j \leftarrow j + 1$ 
21 fin
22 si  $k_{\text{primero.inicio}} \neq k_{\text{ultimo.inicio}} \ \&\& \ k_{\text{primero.fin}} \neq k_{\text{ultimo.fin}}$ 
   entonces
23 | |  $\text{grupos.INSERTAR}(k_{\text{primero.inicio}}, k_{\text{ultimo.fin}})$ 
24 en otro caso
25 | |  $\text{grupos.INSERTAR}(k_{\text{ultimo.inicio}}, k_{\text{ultimo.fin}})$ 
26 fin
27 fin

```

Algoritmo 5: Evaluar si el k -mero kj se encuentra dentro del k -mero ki

```

Entrada: ki // Primer k-mero
           kj // Segundo k-mero
Salida : encuentra // Si se encuentra dentro o no
1 inicio
2 | encuentra ← falso
3 | si  $ki.inicio \geq kj.inicio \ \&\& \ ki.fin \leq kj.fin$  entonces
4 | | encuentra ← verdadero
5 | fin
6 fin

```

Algoritmo 6: Evaluar si el k -mero kj se encuentra conectado con ki

```

Entrada: ki // Primer k-mero
           kj // Segundo k-mero
Salida : conectado // Si se encuentra conectado o no
1 inicio
2 | conectado ← falso
3 | si  $kj.inicio < ki.inicio \ \&\& \ kj.fin == ki.inicio \ || \ (kj.fin + 1) == ki.inicio$ 
   | entonces
4 | | conectado ← verdadero
5 | fin
6 fin

```

Algoritmo 7: Evaluar si el k -mero ki intersecta con kj

```

Entrada: ki // Primer k-mero
           kj // Segundo k-mero
Salida : intersecta // Si intersecta o no
1 inicio
2 | intersecta ← falso
3 | si  $kj.inicio \leq ki.inicio \ \&\& \ kj.fin < ki.fin$  entonces
4 | | intersecta ← verdadero
5 | fin
6 | intersecta ←  $ki.inicio \geq kj.inicio \ \&\& \ ki.fin \leq kj.fin$ 
7 fin

```

Capítulo 4. Experimentos y resultados

En este capítulo se describen los experimentos computacionales realizados para la identificación péptidos antimicrobianos en el transcriptoma del camarón (Ghaffari *et al.*, 2014). Los experimentos se enlistan a continuación:

- Depuración de casos negativos (péptidos no antimicrobianos).
- Cálculo de descriptores y entrenamiento de una máquina de aprendizaje.
- Comparación de la implementación propia de una máquina de aprendizaje y una equivalente de libsvm (Chang y Lin, 2011).
- Evaluación del tiempo de cómputo y consumo de memoria RAM para el cálculo de descriptores moleculares en paralelo con CPU, para los 243,643,492 k -meros únicos.
- Evaluación del tiempo de cómputo y consumo de memoria RAM del algoritmo propuesto en paralelo con CPU.
- Visualización y análisis de los k -meros predichos como AMP dentro de las secuencias del transcriptoma (proteoma) del camarón.

4.1. Depuración del conjunto de casos negativos con CAMP

Partiendo del subconjunto de casos negativos, el cual originalmente se encontraba conformado por 4,010 secuencias (Verdugo, 2014), se procedió a analizar dicho conjunto con los cuatro algoritmos proporcionados por CAMP para la predicción de actividad antimicrobiana: *Support Vector Machine (SVM)*, *Artificial Neural Networks (ANN)*, *Discriminant Analysis (DA)* y *Random Forest (RF)*. En las figuras 8 a la 11 se muestran las distribuciones de las predicciones de los AMPs negativos realizada por los cuatro algoritmos, así como una medida de certeza de dichas predicciones.

Cada figura consta de dos partes, la parte superior, parte (a), muestra el número de secuencias predichas como no-AMP (NAMP) ó AMP (AMP). La parte inferior, parte (b), muestra la probabilidad de que la predicción de las secuencias sea acertada.

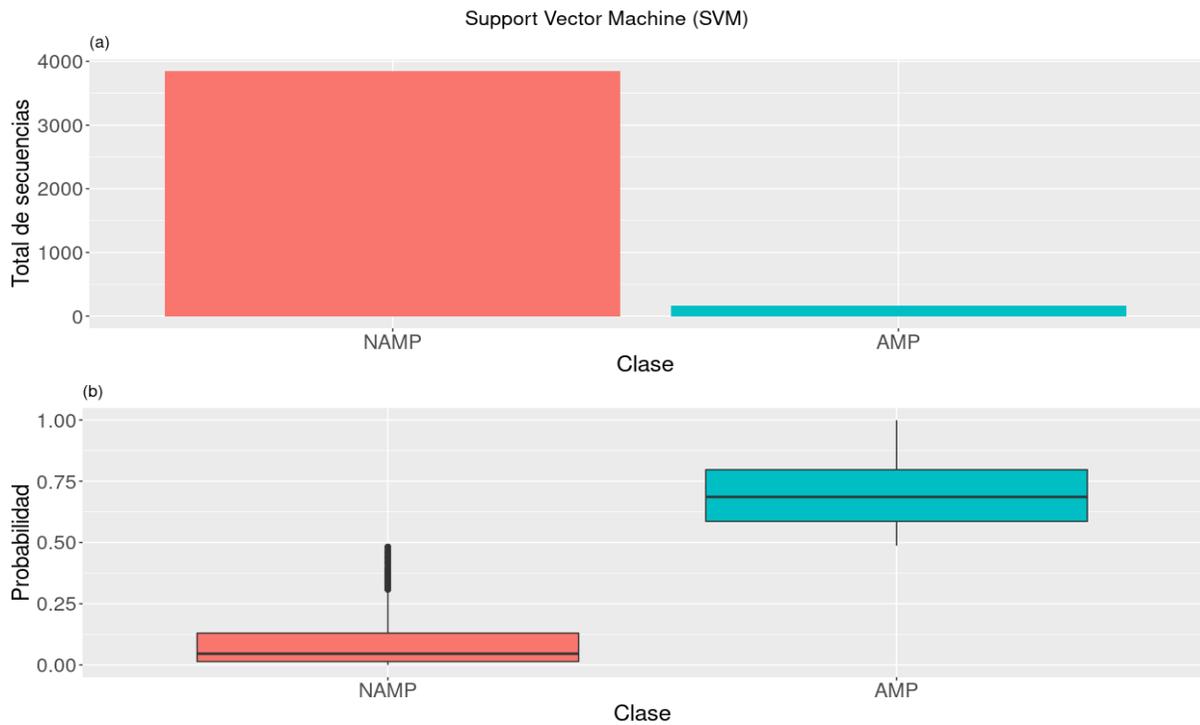


Figura 8. (a) Distribución del conjunto de casos negativos predichos por la SVM de CAMP (165 predichos como AMP y 3,845 predichos como no-AMP). (b) Probabilidad de predicción del conjunto de casos negativos por la SVM de CAMP. Datos procesados el 19/06/2017.

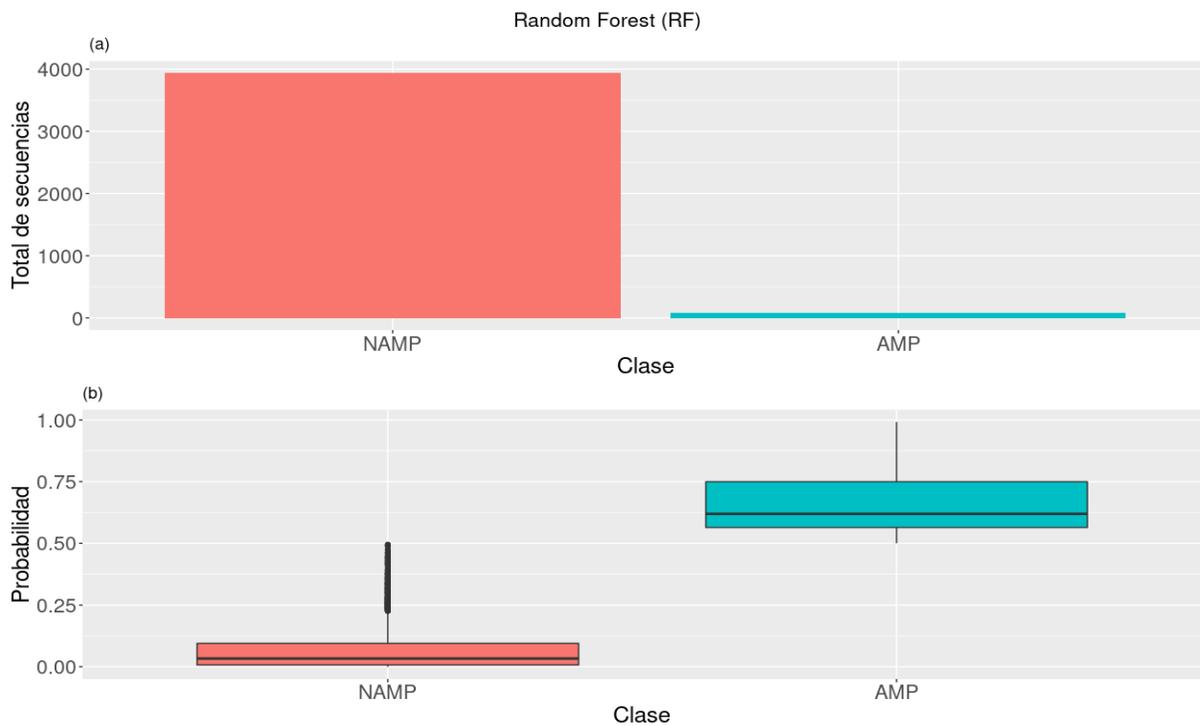


Figura 9. (a) Distribución del conjunto de casos negativos predichos por RF (Random Forest) de CAMP (75 predichos como AMP y 3,935 predichos como no-AMP). (b) Probabilidad de predicción del conjunto de casos negativos por RF de CAMP. Datos procesados el 19/06/2017.

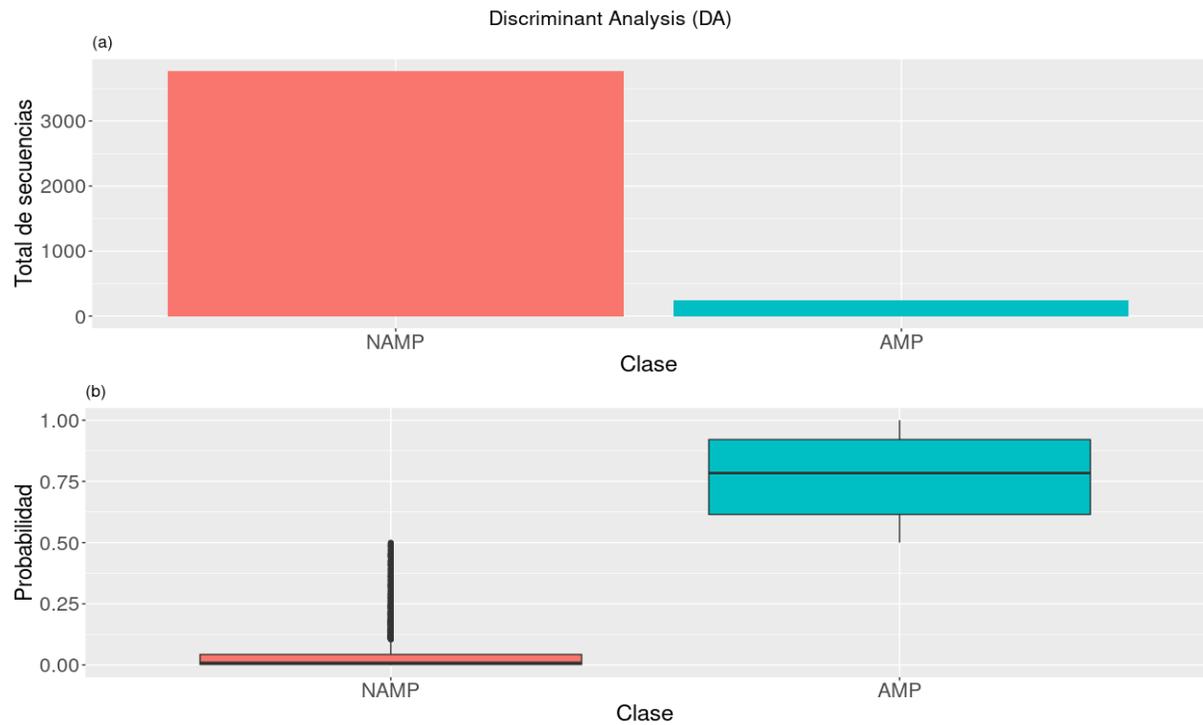


Figura 10. (a) Distribución del conjunto de casos negativos predichos por DA (Discriminant Analysis) de CAMP (245 predichos como AMP y 3,765 predichos como no-AMP). (b) Probabilidad de predicción del conjunto de casos negativos por DA de CAMP. Datos procesados el 19/06/2017.

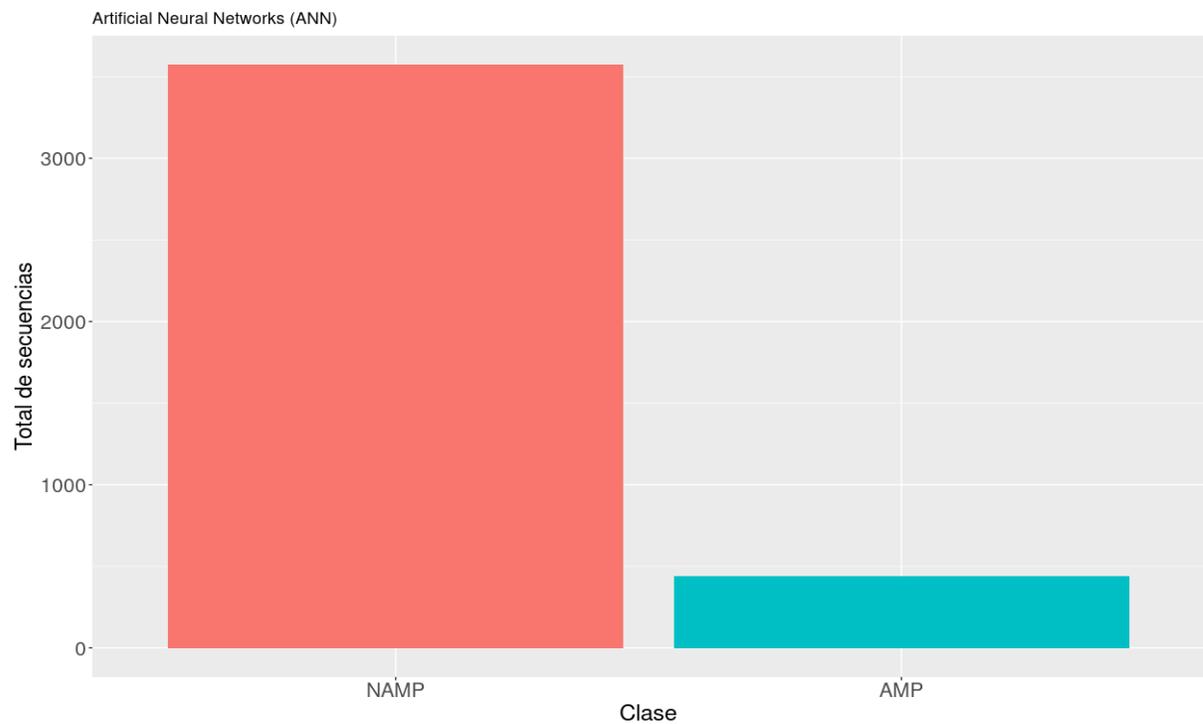


Figura 11. Distribución del conjunto de casos negativos predichos por ANN (Artificial Neural Network) de CAMP (438 clasificados como AMP y 3,572 clasificados como no-AMP). Datos procesados el 19/06/2017.

De las 4,010 secuencias del conjunto original de casos negativos, 3,845 (95.88 %) secuencias fueron predichas como No-AMP por la SVM, 3,935 (98.12 %) como No-AMP por RF, 3,765 (93.89 %) por DA y 3,572 (89.07 %) por ANN. Obteniendo un total de 3,472 (86.58 %) casos negativos en común con los cuatro algoritmos, resultando el conjunto depurado de casos negativos. Esto es, que un 86.58 % del conjunto de casos negativos, fue predicho como no-AMP por los cuatro algoritmos de CAMP en conjunto, mientras que el 13.42 % del conjunto contenía casos que fueron predichos como AMP por al menos uno de los cuatro algoritmos, los cuales, fueron predichos como AMP con una probabilidad superior o igual al 50 %.

4.2. Cálculo de descriptores moleculares y entrenamiento de una máquina de aprendizaje

Se ejecutó el algoritmo de Verdugo (2014) utilizando el conjunto de casos negativos depurados (*ver Sección 4.1*), obteniendo un total de 51 descriptores moleculares de los 276 descriptores que se obtenían anteriormente para la BD que contenía los casos negativos sin depurar. Se realizó el cálculo de los 51 descriptores moleculares al conjunto de casos positivos de Verdugo (2014) y conjunto de casos negativos depurados, obteniendo la salida en formato csv, a la cual se le agregó (en lenguaje R) su clase como la característica número 52, teniendo como 1 para AMP y -1 para no-AMP. Posteriormente, se convirtieron las 52 características, del conjunto de casos positivos y negativos a formato libsvm (Chang y Lin, 2011). Finalmente, se procedió a hacer el escalado de los datos en el intervalo $[-1, 1]$ y el entrenamiento de la máquina de aprendizaje con libsvm utilizando la siguiente configuración:

- **Tipo de kernel (argumento “-t”):** lineal
- **Tipo de máquina de soporte vectorial (argumento “-s”):** C-SVC
- **Costo (argumento “-c”):** 0.526315789
- **Epsilon (argumento “-e”):** 0.001
- **Epsilon de la función de pérdida (argumento “-p”):** 0.1
- **Probabilidad de estimación (argumento “-b”):** 1.0

Obteniendo un porcentaje de clasificación de 91.49% de precisión en entrenamiento con validación cruzada de diez pliegues.

4.3. Pruebas de cálculo descriptores moleculares y evaluación de la máquina de aprendizaje

En esta sección se muestran los resultados obtenidos de diferentes pruebas realizadas sobre el cálculo de 51 descriptores moleculares utilizando diferentes números de hilos de CPU con el uso de la librería Intel TBB 2018.0 (Reinders, 2007), así como la evaluación de la máquina de aprendizaje, en lenguaje de programación C++ 14. Las pruebas fueron realizadas con dos compiladores, G++ e Intel C++ Compiler en el cluster *OMICA* del *CICESE*, el cual utiliza el sistema de archivos paralelos *LUSTRE* y a *SLURM* como sistema manejador de tareas, conformado por 31 nodos, incluyendo el nodo maestro, con las siguientes características por nodo:

- **Sistema operativo:** RedHat Enterprise Linux 6.7 (kernel 2.6.32-573.el6.x86_64)
- **Compilador G++:** G++ 7.2.0
- **Compilador Intel C++:** 18.0.0 20170811
- **Glibc (GNU libc):** 2.12
- **Librería LibSVM:** 3.22
- **Librería Intel TBB:** 2018.0
- **Memoria RAM:** 128 GB.
- **Conjunto de instrucción (ISA) más alto:** AVX-2
- **Opciones de compilación (GCC):** -Wall -ftree-vectorize -march=native -fopt-info-vec-optimized -mfpmath=sse -funroll-loops -mtune=generic -O3 -fdiagnostics-color=auto
- **Opciones de compilación (Intel C++ Compiler):** -O3 -xHOST -ipo -Wall -axCORE-AVX2 -mtune=core-avx2 -march=core-avx2
- **Procesadores:** Dos Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz. Cada uno con 12 núcleos (12 núcleos por 2 procesadores = 24 núcleos).

4.3.1. Comparación de la implementación propia de una máquina de aprendizaje y libsvm

Se comparó el tiempo de ejecución y consumo de memoria de dos implementaciones de una máquina de aprendizaje: una propia desarrollada en C++ 11 y una equivalente de libsvm (Chang y Lin, 2011). Las principales ventajas de la implementación propia de la máquina de aprendizaje son:

- **Representación más compacta:** Debido a que los 51 descriptores moleculares a evaluar siempre tendrán algún valor, estos se representan en un arreglo de números flotantes contiguos en lugar de una lista simple (libsvm). Lo que implica un ahorro en el consumo de memoria RAM.
- **Autovectorización:** Aprovechando la representación de arreglos de los 51 descriptores moleculares, se utiliza el conjunto de instrucción más alto de la arquitectura del procesador (ISA, **I**nstruction **S**et **A**rchitecture) viable para la operación del kernel por medio de la autovectorización, que es la vectorización (Deilmann, 2012) realizada por el compilador. Cabe resaltar que la autovectorización no siempre puede ser empleada por todos los compiladores o en todas las versiones de los mismos.

La comparativa consistió en:

1. **Lectura del proteoma:** Se lee en memoria RAM las 18,694 secuencias del proteoma del camarón, una por una. Al mismo tiempo se valida que estas contengan únicamente aminoácidos estándares, esto último por la limitante de los 51 (ver *Tabla 13*) descriptores moleculares, los cuales trabajan únicamente con aminoácidos estándares; donde no se llegaron a descartar secuencias por dicha validación. Finalmente, se eliminan 2,657 secuencias duplicadas de forma similar a como se hace para eliminación de k -meros duplicados (ver *Algoritmo 2*), dejando 16,037 secuencias para la extracción de los k -meros totales para k igual a 10 hasta 60.
2. **Extracción de k -meros únicos:** Se extraen los k -meros totales de las 16,037 secuencias únicas del proteoma del camarón, y se eliminan aquellos k -meros duplicados por medio del Algoritmo 2.

3. **Cálculo de descriptores moleculares:** Se realiza el cálculo de 51 descriptores moleculares de los k -meros únicos (ver *Tabla 13*).
4. **Evaluación de la máquina de aprendizaje:** Se evalúa la máquina de aprendizaje en ambas implementaciones (propia y libsvm), descartando aquellos k -meros predichos como **no-AMP** y reportando el total de k -meros únicos predichos como AMP.

Se realizaron los pasos descritos arriba un total de 5 ejecuciones para diferentes números de hilos de CPU y compiladores. En la Figura 12 se muestran los resultados obtenidos con el compilador G++ (ver *Tabla 2*), mientras que en la Figura 13 se muestran los resultados obtenidos para el compilador Intel C++ (ver *Tabla 3*). En la columna “Implementación” se especifica si se utilizó la implementación de libsvm (Chang y Lin, 2011) ó la implementación propia; en la columna “# hilos” se especifica el número de hilos utilizados durante la prueba; en la columna “Tiempo” se muestra el tiempo promedio de las 5 ejecuciones. Del mismo modo, en la columna “Memoria RAM (Kilobytes)” se muestra la memoria RAM consumida (comando “sacct” de SLURM) en el nodo del cluster *OMICA* donde se realizó la prueba.

Tabla 2. Comparativa de la implementación propia y de libsvm de la máquina de aprendizaje¹ (compilador G++)

Implementación	# hilos	Tiempo (hh:mm)	Memoria RAM (Kilobytes)
LibSVM	1	15:45	1,334,712 (~1,334.71 MB)
Propia	1	13:05	1,366,424 (~1,366.42 MB)
LibSVM	2	08:10	1,337,744 (~1,337.74 MB)
Propia	2	06:49	1,316,580 (~1,316.58 MB)
LibSVM	4	04:06	1,252,912 (~1,252.91 MB)
Propia	4	03:25	1,350,572 (~1,350.57 MB)
LibSVM	8	02:04	1,312,192 (~1,312.19 MB)
Propia	8	01:43	1,406,104 (~1,406.1 MB)
LibSVM	16	01:03	1,429,472 (~1,429.47 MB)
Propia	16	00:53	1,348,624 (~1,348.62 MB)

Continúa en la siguiente página

Tabla 2 – Continuación

Implementación	# hilos	Tiempo (hh:mm)	Memoria RAM (Kilobytes)
LibSVM	24	00:43	1,361,128 (~1,361.13 MB)
Propia	24	00:36	1,275,636 (~1,275.64 MB)

Tabla 3. Comparativa de la implementación propia y de libsvm de la máquina de aprendizaje² (compilador Intel C++)

Implementación	# hilos	Tiempo (hh:mm)	Memoria RAM (Kilobytes)
LibSVM	1	16:17	1,361,908 (~1,361.91 MB)
Propia	1	06:57	1,359,924 (~1,359.92 MB)
LibSVM	2	08:25	1,316,952 (~1,316.95 MB)
Propia	2	03:53	1,314,588 (~1,314.59 MB)
LibSVM	4	04:13	1,393,976 (~1,393.98 MB)
Propia	4	01:54	1,378,496 (~1,378.5 MB)
LibSVM	8	02:07	1,427,396 (~1,427.4 MB)
Propia	8	00:57	1,350,340 (~1,350.34 MB)
LibSVM	16	01:05	1,258,824 (~1,258.82 MB)
Propia	16	00:30	1,344,288 (~1,344.29 MB)
LibSVM	24	00:44	1,446,108 (~1,446.11 MB)
Propia	24	00:20	1,300,736 (~1,300.74 MB)

¹El tiempo reportado es el promedio de las 5 ejecuciones del algoritmo reportado por el algoritmo con el compilador G++. El consumo de memoria RAM es el reportado por SLURM (comando "sacct").

²El tiempo reportado es el promedio de las 5 ejecuciones del algoritmo reportado por el algoritmo con el compilador Intel C++. El consumo de memoria RAM es el reportado por SLURM (comando "sacct").

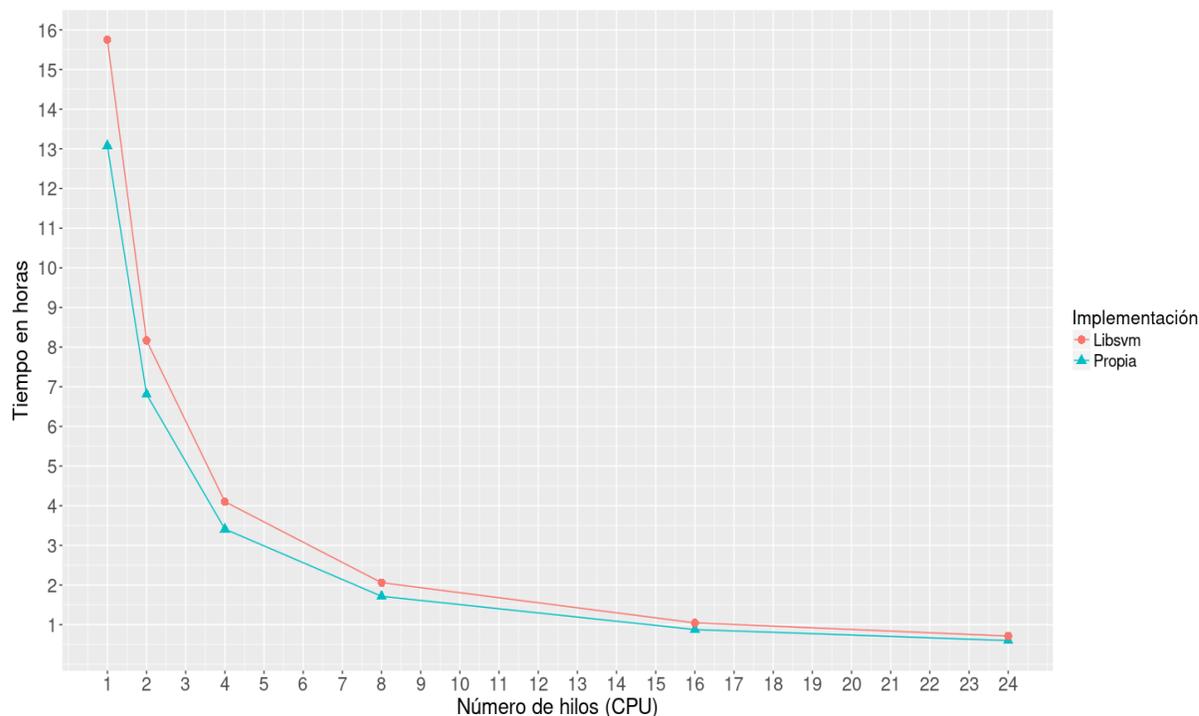


Figura 12. Tiempo de cómputo promedio (de 5 ejecuciones) para el cálculo de 51 descriptores moleculares (ver *Tabla 13*) de los k -meros únicos para $k = 10$ hasta $k = 60$ utilizando una implementación propia y la de libsvm de la máquina de aprendizaje con el compilador G++.

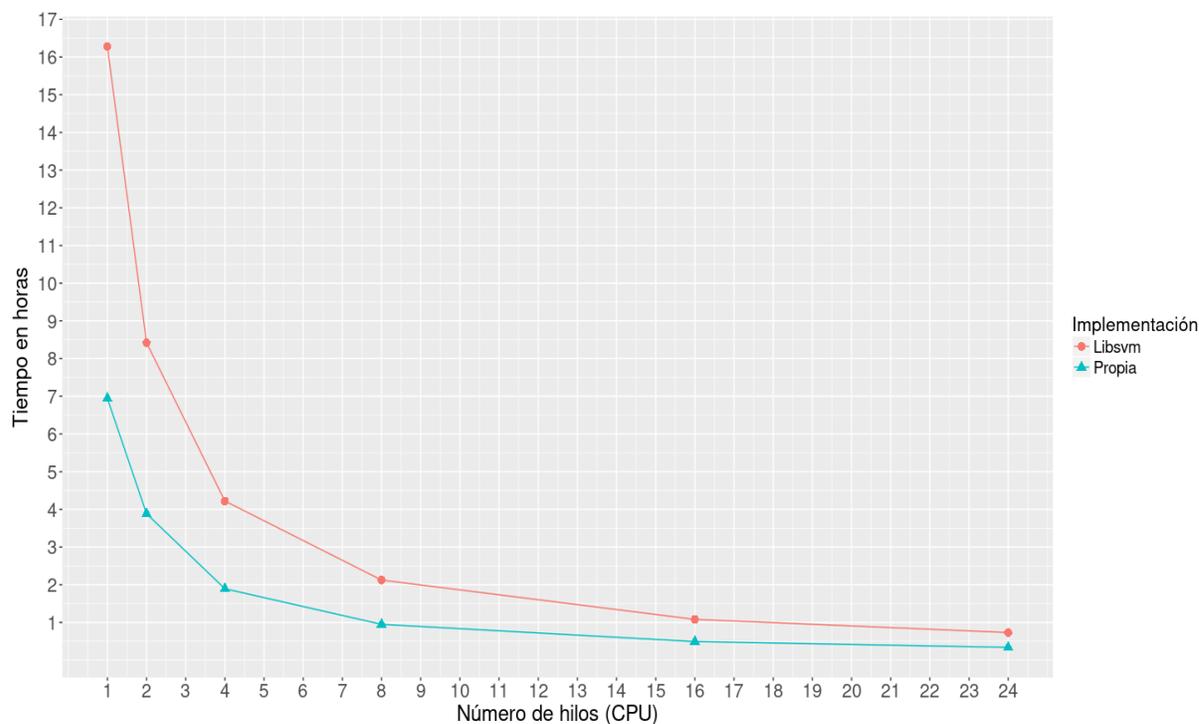


Figura 13. Tiempo de cómputo promedio (de 5 ejecuciones) para el cálculo de 51 descriptores moleculares (ver *Tabla 13*) de los k -meros únicos para $k = 10$ hasta $k = 60$ utilizando una implementación propia y de libsvm de una máquina de aprendizaje con el compilador Intel C++.

Debido a que la variación entre las 5 ejecuciones para ambos compiladores (G++ e Intel C++) es prácticamente nula, el tiempo fue representado por medio de gráficos de líneas y puntos en las figuras 12 y 13 en lugar de gráficos “boxplot”.

En las figuras 12 y 13 se aprecia que conforme incrementa el número de hilos, el tiempo de “libsvm” converge al mismo valor de tiempo que con la implementación propia. Esto puede ser debido al alineamiento de los datos realizada por el compilador. Sin embargo, con las nuevas arquitecturas como “Knights Landing” y “Skylake-X” de los procesadores Intel, se espera que el rendimiento para el cálculo de descriptores sea aún mayor en la implementación propia de la máquina de aprendizaje, esto es gracias a que cuentan con el conjunto de instrucciones “AVX-512”, el cual puede realizar más operaciones en un solo ciclo del CPU (Alaa Eltablawy, 2017). Se aprecia también, que en ambas implementaciones, el binario generado por el compilador de Intel lleva menos tiempo para realizar los cálculos.

En las figuras 14 y 15 se muestra el consumo de memoria RAM reportado por SLURM (comando “sacct”) en las 5 ejecuciones del algoritmo para 1, 2, 4, 8, 16 y 24 hilos, tanto para la implementación propia de la máquina de aprendizaje, como la implementación de libsvm. En la Figura 14 se muestra el consumo de memoria obtenido con el compilador G++, mientras que en la Figura 15 se muestra el consumo de memoria RAM obtenido con el compilador Intel C++. En las figuras 14 y 15 se aprecia que para ambos compiladores, en algunas configuraciones de hilos, la implementación propia demanda mayor uso de memoria RAM, mientras que para otras configuraciones, demanda menor uso, aunque la diferencia entre ambas implementaciones es relativamente pequeña (alrededor de 200 MB). La diferencia se puede deber al hecho de que las secuencias y k -meros únicos a procesar no son siempre los mismos, esto es debido a las estructuras de datos concurrentes de la librería Intel TBB.

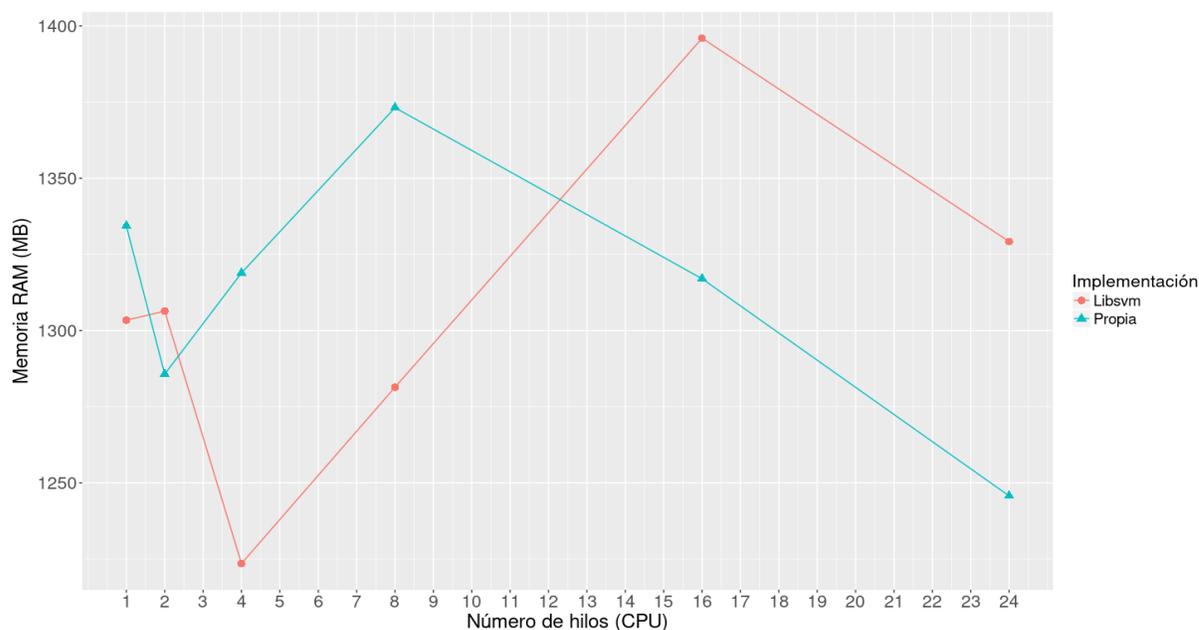


Figura 14. Memoria RAM consumida por InProt con 1, 2, 4, 8, 16 y 24 hilos, para el cálculo de 51 descriptores moleculares (ver Tabla 13) de los k -meros únicos para $k = 10$ hasta $k = 60$ utilizando una implementación propia y de libsvm de la máquina de aprendizaje con el compilador G++. La memoria RAM es la reportada por SLURM (comando “sacct”) en las 5 ejecuciones del algoritmo.

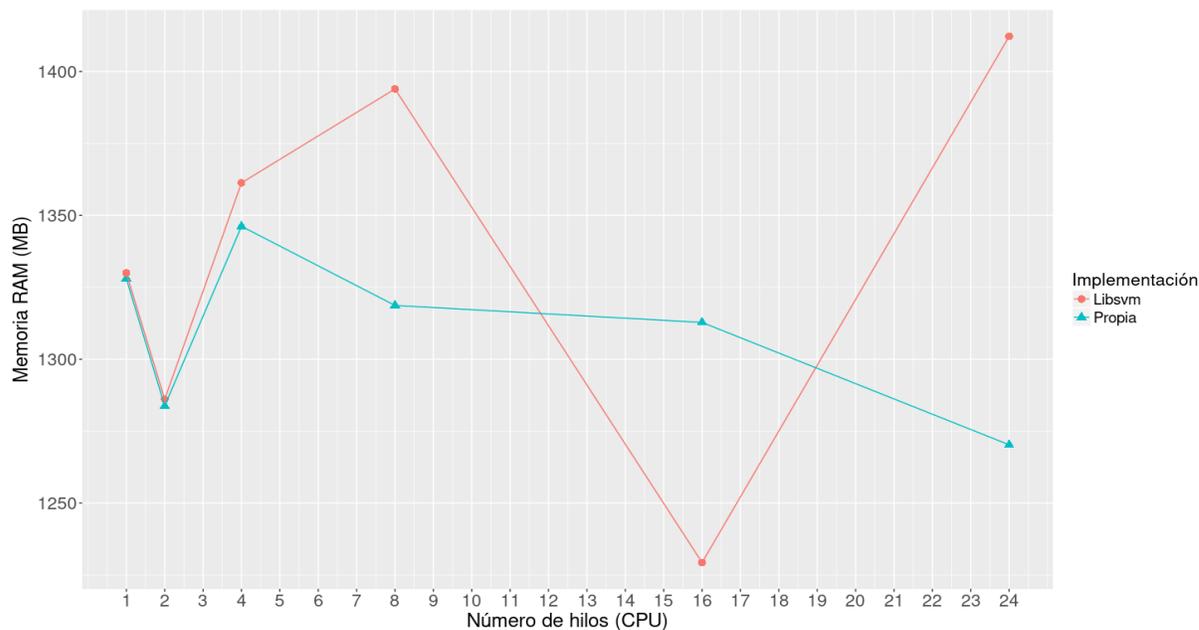


Figura 15. Memoria RAM consumida por InProt con 1, 2, 4, 8, 16 y 24 hilos, para el cálculo de 51 descriptores moleculares (ver Tabla 13) de los k -meros únicos para $k = 10$ hasta $k = 60$ utilizando una implementación propia y libsvm como máquina de aprendizaje con el compilador Intel C++. La memoria RAM es la reportada por SLURM (comando “sacct”) en las 5 ejecuciones del algoritmo.

4.3.2. *k*-meros predichos

De los *k*-meros únicos, una proporción de estos fueron predichos como AMPs por InProt. En la Figura 16 se muestra la cantidad de *k*-meros totales, *k*-meros únicos y de *k*-meros únicos predichos como AMP por InProt (ver Tabla 4), donde aproximadamente el 47% de *k*-meros totales son únicos, y menos de la cuarta parte de los *k*-meros únicos (aproximadamente 17%), son predichos como AMP. Se puede observar en la Figura 16 que los *k*-meros más pequeños son mayoría para los *k*-meros totales, únicos y predichos como AMP, y estos a su vez van decreciendo de forma gradual conforme el tamaño aumenta. En el caso de los *k*-meros totales y únicos, es algo que se esperaría de acuerdo a la Ecuación 3.

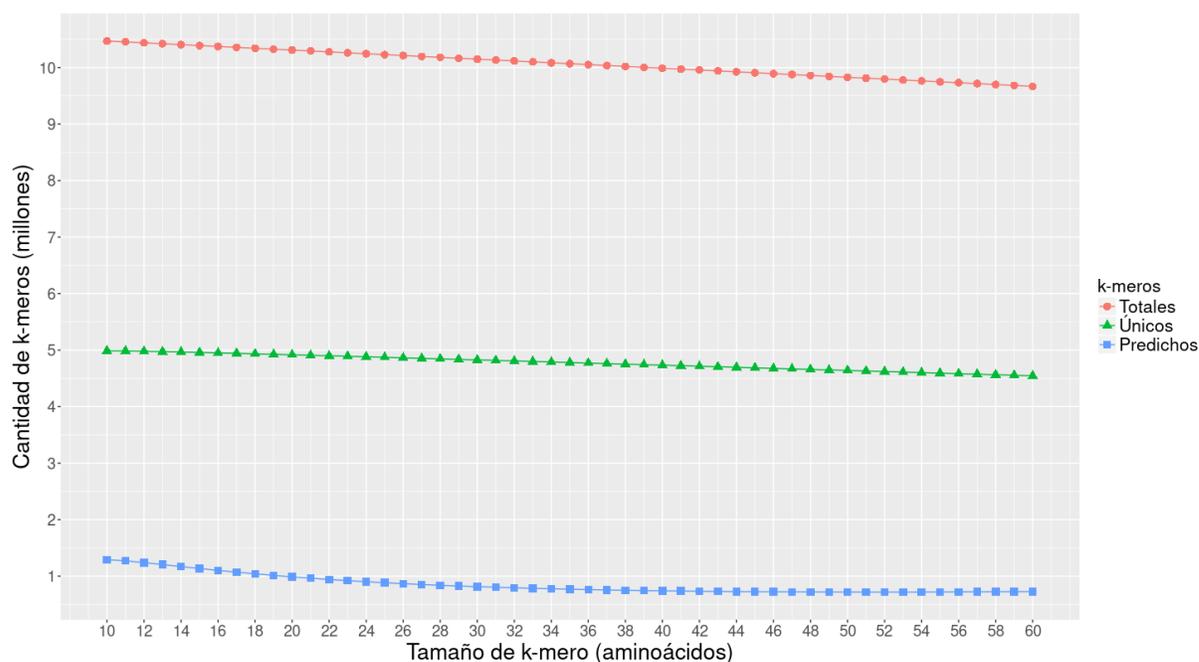


Figura 16. Distribución de la cantidad de *k*-meros totales, únicos y predichos como AMP por el algoritmo propuesto.

Tabla 4. *k*-meros totales, únicos y predichos como AMP

k-mero	Totales	Únicos	Predichos (AMP)
10	10,469,115	4,986,845	1,290,587
11	10,453,078	4,984,019	1,272,101
12	10,437,041	4,979,166	1,239,784
13	10,421,004	4,972,979	1,205,523
14	10,404,967	4,966,022	1,169,946
15	10,388,930	4,958,553	1,135,570
16	10,372,893	4,950,687	1,102,114
17	10,356,856	4,942,543	1,071,083
18	10,340,819	4,934,175	1,041,625
19	10,324,782	4,925,639	1,013,984
20	10,308,745	4,916,962	987,807
21	10,292,708	4,908,187	963,736
22	10,276,671	4,899,313	941,166
23	10,260,634	4,890,359	920,520
24	10,244,597	4,881,348	901,509
25	10,228,560	4,872,301	883,666
26	10,212,523	4,863,208	867,468
27	10,196,486	4,854,084	852,385
28	10,180,449	4,844,937	838,156
29	10,164,412	4,835,753	826,068
30	10,148,375	4,826,544	813,964
31	10,132,338	4,817,306	803,852
32	10,116,301	4,808,040	794,020
33	10,100,264	4,798,745	784,917
34	10,084,227	4,789,436	776,794
35	10,068,190	4,780,113	769,082
36	10,052,153	4,770,780	762,532
37	10,036,116	4,761,436	755,831
38	10,020,079	4,752,085	750,284

Continúa en la siguiente página

Tabla 4 – Continuación

k-mero	Totales	Únicos	Predichos (AMP)
39	10,004,042	4,742,726	745,505
40	9,988,005	4,733,362	740,912
41	9,971,968	4,723,994	736,503
42	9,955,931	4,714,621	732,657
43	9,939,894	4,705,244	729,445
44	9,923,857	4,695,866	726,869
45	9,907,820	4,686,485	724,907
46	9,891,783	4,677,103	723,080
47	9,875,746	4,667,717	721,671
48	9,859,709	4,658,327	720,083
49	9,843,672	4,648,932	719,126
50	9,827,635	4,639,537	718,543
51	9,811,598	4,630,140	718,152
52	9,795,561	4,620,741	717,748
53	9,779,524	4,611,337	717,911
54	9,763,487	4,601,934	718,552
55	9,747,450	4,592,528	719,724
56	9,731,413	4,583,116	720,841
57	9,715,376	4,573,695	722,479
58	9,699,339	4,564,268	724,033
59	9,683,302	4,554,841	725,818
60	9,667,265	4,545,413	726,944
Total	513,477,690	243,643,492	43,487,577

4.4. k-meros predichos - CAMP

Dado que se obtuvieron un gran número de *k*-meros predichos como AMP (ver *Tabla 4*), se procedió a verificar si es posible reducir esta cantidad mediante el uso de predictores adicionales. Para esto, se procedió a analizar con los cuatro algoritmos de

CAMP (Support Vector Machine, Random Forest, Discriminant Analysis, Artificial Neural Network) los k -meros predichos como AMP y como No-AMP por InProt (ver Anexo 3 - InProt - In silico Proteolysis) utilizando CAMPRED (ver Sección 3.4.6). Con este programa se contabilizaron aquellos k -meros predichos como AMPs por los cuatro algoritmos de CAMP a partir de los k -meros predichos como AMP así como los predichos como no-AMP por InProt. En la Figura 17 se muestra la cantidad de k -meros predichos como AMP por InProt (ver Tabla 5), así como los k -meros predichos como AMP por cada uno de los cuatro algoritmos de CAMP de forma individual y en conjunto. Del mismo modo que en la Figura 17, en la Figura 18 se muestra para los k -meros (de 10 a 50) predichos como No-AMP por InProt (ver Tabla 6), así como los k -meros predichos como AMP por cada uno de los cuatro algoritmos de CAMP de forma individual y en conjunto. La columna “Algoritmo (AMPs)” representa los k -meros predichos como AMP por InProt, mientras que la columna “CAMP (AMPs)” representa los k -meros predichos como AMP por InProt y que a su vez fueron predichos como AMP por CAMP.

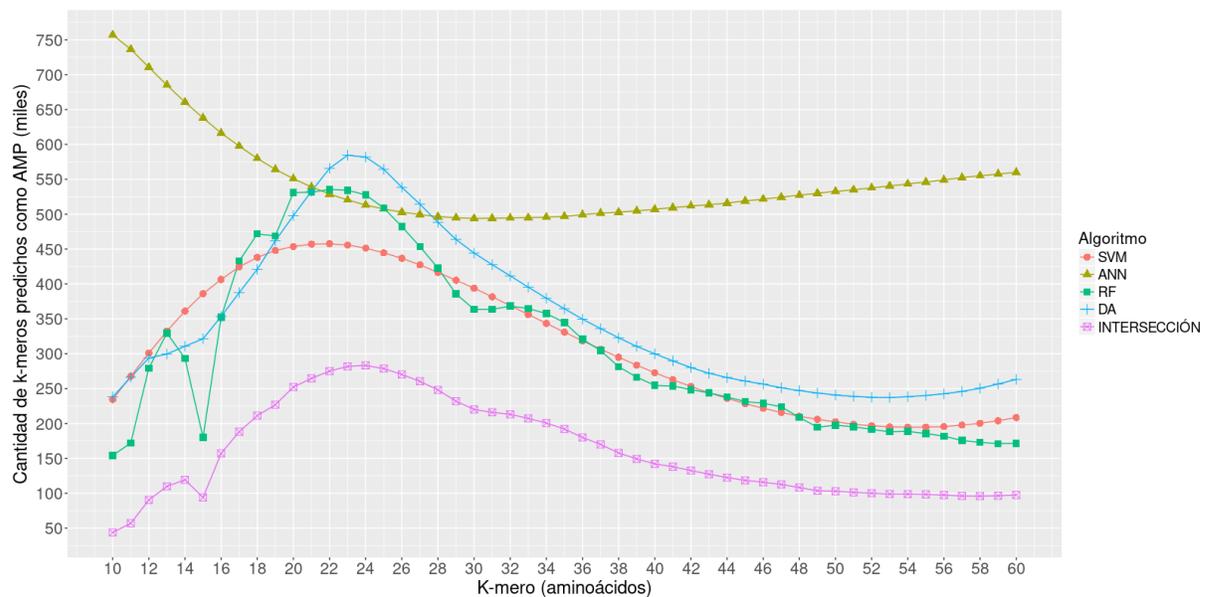


Figura 17. Total de k -meros predichos como AMPs por InProt y predichos como AMP por los cuatro algoritmos de CAMP. En el eje “X” se muestra el tamaño de k -mero y en el eje “Y” se muestra la cantidad de k -meros predichos como AMP por cada uno de los cuatro algoritmos y por los cuatro algoritmos en conjunto.

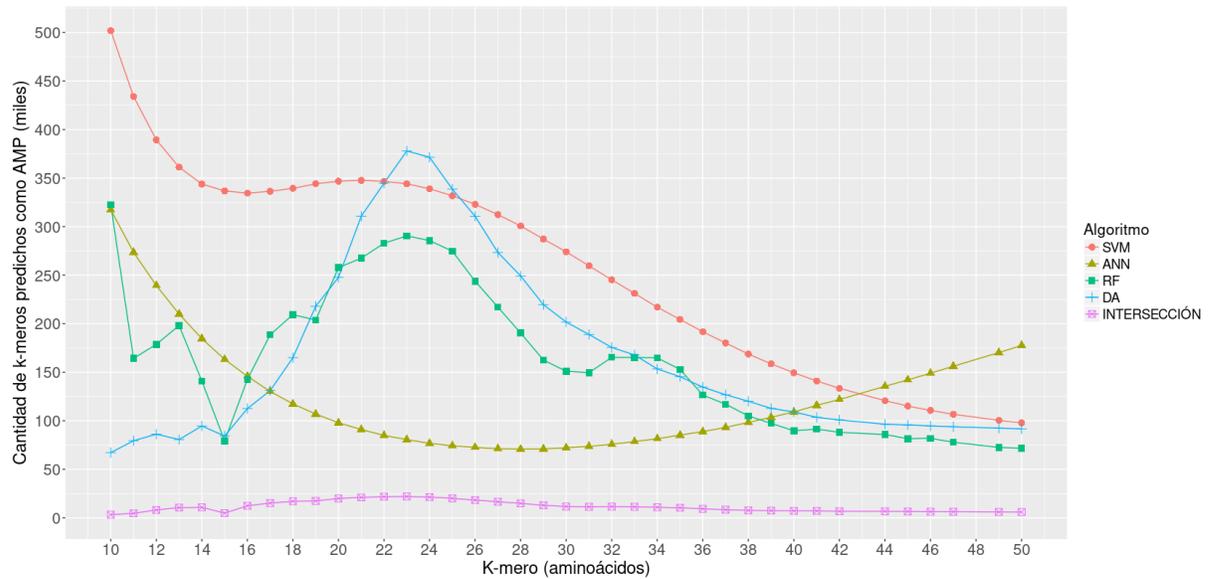


Figura 18. Total de k -meros predichos como No-AMPs por InProt y predichos como AMP por los cuatro algoritmos de CAMP. En el eje "X" se muestra el tamaño de k -mero y en el eje "Y" se muestra la cantidad de k -meros predichos como AMP por cada uno de los cuatro algoritmos y por los cuatro algoritmos en conjunto. Los valores para k igual a 43 y 48, se encuentran ausentes.

Tabla 5. k-meros predichos como AMP por InProt y como AMP por los cuatro algoritmos de CAMP con CAMPRED

k	Algoritmo (AMPs)	CAMP (AMPs)				
		SVM	ANN	RF	DA	Comunes (intersección)
10	1, 290, 587	234, 267	757, 052	154, 176	238, 028	43,737 (3.39%)
11	1, 272, 101	267, 731	736, 416	171, 913	266, 269	56,942 (4.47%)
12	1, 239, 784	300, 806	710, 620	279, 121	293, 820	90,294 (7.28%)
13	1, 205, 523	332, 430	684, 990	329, 616	299, 557	109,756 (9.10%)
14	1, 169, 946	360, 980	660, 605	293, 154	310, 700	119,290 (10.19%)
15	1, 135, 570	385, 929	637, 991	179, 849	321, 358	93,880 (8.26%)
16	1, 102, 114	406, 425	616, 269	352, 102	353, 274	157,116 (14.25%)
17	1, 071, 083	424, 516	597, 512	432, 742	387, 475	188,148 (17.56%)
18	1, 041, 625	437, 994	579, 915	471, 700	420, 992	211,180 (20.27%)
19	1, 013, 984	447, 866	564, 264	468, 964	461, 897	226,834 (22.37%)
20	987, 807	453, 574	550, 910	530, 857	497, 890	252,232 (25.53%)
21	963, 736	457, 121	538, 996	531, 484	532, 872	264,577 (27.45%)
22	941, 166	457, 527	528, 962	535, 443	565, 663	275,030 (29.22%)
23	920, 520	455, 803	520, 353	534, 280	584, 552	281,714 (30.60%)
24	901, 509	451, 350	513, 158	527, 668	581, 862	283,210 (31.41%)
25	883, 666	444, 575	507, 446	508, 858	564, 395	278,613 (31.52%)

Continúa en la siguiente página

Tabla 5 – Continuación

k	Algoritmo (AMPs)	CAMP (AMPs)					Comunes (intersección)
		SVM	ANN	RF	DA	Comunes	
26	867,468	436,674	502,854	482,318	538,536	270,236 (31.15%)	
27	852,385	427,367	499,493	453,617	514,426	260,405 (30.55%)	
28	838,156	416,393	496,475	422,591	488,345	247,997 (29.59%)	
29	826,068	405,311	494,821	385,687	463,780	231,925 (28%)	
30	813,964	393,906	493,942	363,517	444,255	220,029 (27%)	
31	803,852	381,511	494,215	363,441	427,656	215,940 (26.86%)	
32	794,020	368,985	494,461	368,277	411,565	213,115 (26.84%)	
33	784,917	356,127	494,991	364,767	395,344	207,230 (26.40%)	
34	776,794	343,471	495,572	357,483	379,475	200,575 (25.82%)	
35	769,082	330,726	496,916	344,693	364,365	192,051 (24.97%)	
36	762,532	318,511	499,320	320,602	349,524	179,898 (23.59%)	
37	755,831	306,453	501,387	304,445	335,931	169,956 (22.49%)	
38	750,284	294,830	502,871	281,292	322,719	157,648 (21%)	
39	745,505	283,585	504,727	266,310	310,408	149,093 (20%)	
40	740,912	272,673	507,110	254,602	299,795	142,066 (19.17%)	
41	736,503	262,601	509,417	253,626	289,631	137,947 (18.73%)	
42	732,657	252,992	511,729	248,465	280,075	132,483 (18%)	

Continúa en la siguiente página

Tabla 5 – Continuación

k	Algoritmo (AMPs)	CAMP (AMPs)					Comunes (intersección)
		SVM	ANN	RF	DA	Comunes (intersección)	
43	729,445	243,970	513,514	243,775	271,962	127,166 (17.43%)	
44	726,869	235,933	515,995	237,975	265,816	122,455 (16.85%)	
45	724,907	228,457	518,906	231,351	260,725	118,377 (16.33%)	
46	723,080	221,651	521,616	228,987	256,402	115,750 (16%)	
47	721,671	215,836	524,414	223,815	251,130	112,538 (15.59%)	
48	720,083	210,253	527,232	208,546	247,110	108,160 (15%)	
49	719,126	2059,67	530,044	194,622	243,584	103,438 (14.38%)	
50	718,543	202,267	532,638	197,564	240,834	102,786 (14.30%)	
51	718,152	198,852	535,064	195,103	238,778	101,181 (14%)	
52	717,748	196650	537661	191556	237416	100077 (13.94%)	
53	717,911	195187	540457	187979	237099	98779 (13.76%)	
54	718,552	194598	543329	188605	238267	98677 (13.73%)	
55	719,724	194810	546125	185531	240168	98266 (13.65%)	
56	720,841	195513	549166	181751	242473	97360 (13.51%)	
57	722,479	197666	552446	175534	245912	96128 (13.31%)	
58	724,033	200238	555018	172933	250542	95938 (13.25%)	
59	725,818	203896	557642	171102	256567	96394 (13.28%)	

Continúa en la siguiente página

Tabla 5 – Continuación

<i>k</i>	Algoritmo (AMPs)	CAMP (AMPs)					Comunes (intersección)
		SVM	ANN	RF	DA		
60	726,944	208372	559880	171360	263139	97421 (13.40%)	
Total	43,487,577	15,921,126	27,866,907	15,725,749	17,784,358	8,152,038 (18.75%)	

Tabla 6. k-meros predichos como No-AMP por InProt y como AMP por los cuatro algoritmos de CAMP con CAMPRED

k	Algoritmo (No-AMPs)	CAMP (AMPs)					Comunes (intersección)
		SVM	ANN	RF	DA		
10	3, 696, 258	501, 969	317, 459	322, 758	67, 158	3,220 (0.08%)	
11	3, 711, 918	434, 201	273, 611	164, 237	79, 392	4,565 (0.12%)	
12	3, 739, 382	389, 410	239, 333	178, 742	86, 155	8,028 (0.21%)	
13	3, 767, 456	361, 286	209, 970	198, 073	80, 605	10,489 (0.28%)	
14	3, 796, 076	343, 894	184, 619	140, 955	94, 605	10,792 (0.28%)	
15	3, 822, 983	336, 787	163, 238	79, 217	84, 248	4,796 (0.12%)	
16	3, 848, 573	334, 569	145, 822	142, 599	11, 2542	12,356 (0.32%)	
17	3, 871, 460	336, 410	130, 180	188, 603	131, 023	15,160 (0.39%)	
18	3, 892, 550	339, 471	117, 337	209, 352	164, 977	17,028 (0.43%)	
19	3, 911, 655	344, 111	106, 900	203, 726	217, 918	17,303 (0.44%)	
20	3, 929, 155	346, 858	97, 900	257, 771	247, 685	19,913 (0.50%)	
21	3, 944, 451	347, 688	90, 925	267, 506	310, 750	20,938 (0.53%)	
22	3, 958, 147	346, 703	84, 860	282, 907	344, 755	21,713 (0.55%)	
23	3, 969, 839	344, 131	80, 562	290, 517	377, 991	21,986 (0.55%)	
24	3, 979, 839	338, 991	76, 696	285, 608	371, 497	21,324 (0.54%)	
25	3, 988, 635	331, 834	74, 252	274, 717	338, 660	20,058 (0.50%)	

Continúa en la siguiente página

Tabla 6 – Continuación

k	Algoritmo (No-AMPs)	CAMP (AMPs)					Comunes (intersección)
		SVM	ANN	RF	DA		
26	3, 995, 740	322, 960	72, 554	243, 837	310, 667	18,245 (0.46 %)	
27	4, 001, 699	312, 388	71, 220	217, 101	273, 402	16,534 (0.41 %)	
28	4, 006, 781	300, 683	70, 768	190, 420	249, 160	14,882 (0.37 %)	
29	4, 009, 685	287, 200	70, 954	162, 550	219, 560	12,893 (0.32 %)	
30	4, 012, 580	274, 042	72, 180	151, 007	201, 679	11,586 (0.29 %)	
31	4, 013, 454	259, 619	73, 641	149, 551	188, 872	11,354 (0.28 %)	
32	4, 014, 020	245, 236	75, 730	165, 353	175, 512	11,516 (0.29 %)	
33	4, 013, 828	231, 198	78, 556	165, 191	167, 791	11,355 (0.28 %)	
34	4, 012, 642	216, 993	81, 501	164, 773	153, 327	10,878 (0.27 %)	
35	4, 011, 031	204, 393	85, 217	152, 838	145, 386	10,325 (0.26 %)	
36	4, 008, 248	191, 682	88, 830	126, 471	134, 595	9,183 (0.23 %)	
37	4, 005, 605	180, 114	93, 099	117, 017	126, 767	8,359 (0.21 %)	
38	4, 001, 801	168, 774	98, 309	104, 923	120, 070	7,661 (0.19 %)	
39	3, 997, 221	158, 580	103, 606	97, 263	112, 739	7,424 (0.19 %)	
40	3, 992, 450	149, 413	109, 030	89, 629	109, 051	7,104 (0.18 %)	
41	3, 987, 491	140, 943	115, 876	91, 415	103, 521	7,053 (0.18 %)	
42	3, 981, 964	133, 375	121, 901	88, 040	100, 799	6,745 (0.17 %)	

Continúa en la siguiente página

Tabla 6 – Continuación

k	Algoritmo (No-AMPs)	CAMP (AMPs)					Comunes (intersección)
		SVM	ANN	RF	DA		
44	3,968,997	120,685	135,481	85,763	96,408	6,668 (0.17%)	
45	3,961,578	115,132	142,104	81,513	95,628	6,471 (0.16%)	
46	3,954,023	110,565	148,975	81,886	94,616	6,317 (0.16%)	
47	3,946,046	106,657	156,158	77,860	93,830	6,247 (0.16%)	
49	3,929,806	100369	170,252	72,501	92,311	6,012 (0.15%)	
50	3,920,994	97810	177,606	71,749	91,621	5,910 (0.15%)	
Total	153,576,061	10,207,124	4807212	6435939	6567273	450391 (0.29%)	

³Los valores para k igual a 43 y 48, se encuentran ausentes.

4.5. Evaluación del tiempo de cómputo y consumo de memoria RAM del algoritmo propuesto

4.5.1. Evaluación del tiempo de cómputo para reducción de k -meros duplicados

Al aplicar un deslizamiento de ventanas (k -meros) en las secuencias del proteoma, con tamaños entre 10 y 60 aminoácidos de longitud, se obtienen millones de k -meros, donde un gran porcentaje de estos (aproximadamente el 53%) son duplicados (ver *Figura 35*). Debido a la gran cantidad de k -meros duplicados, y con el objetivo de evitar cálculo redundante por medio de la depuración de k -meros duplicados, se requiere de una estructura de datos para el almacenamiento de k -meros únicos de una manera eficiente. Una estructura de datos son las tablas hash, pero dada las dificultades encontradas al momento de implementarlas, se recurrió al uso de vectores lineales, que en conjunto con algoritmos de ordenamiento y comparación lexicográfica de los k -meros, los experimentos computacionales realizados demuestran que es una solución viable. En la *Figura 19* se muestra el tiempo de cómputo requerido para: 1) lectura del proteoma, comprobando que todas las secuencias contengan aminoácidos estándares y almacenándolas en memoria RAM, 2) deslizamiento de ventanas entre 10 y 60 aminoácidos de longitud, almacenando todas las secuencias cubiertas por las ventanas, y 3) eliminación de k -meros duplicados; con el uso de diferente número de hilos y compiladores (G++ e Intel C++). En el eje "X" se muestra el número de hilos, y en el eje "Y" se muestra el tiempo de cómputo en minutos.

Se observa que incluso con el uso de un solo hilo, el tiempo requerido es menor a 7 minutos, decrementándose aún más con el aumento del número de hilos, en ambos compiladores, siendo un tiempo aceptable.

Al requerir únicamente el tamaño del vector de k -meros, y variables de control para el acceso simultáneo al vector, como datos satelitales, el consumo de memoria RAM es menor a 420 MB (ver *Figura 20*).

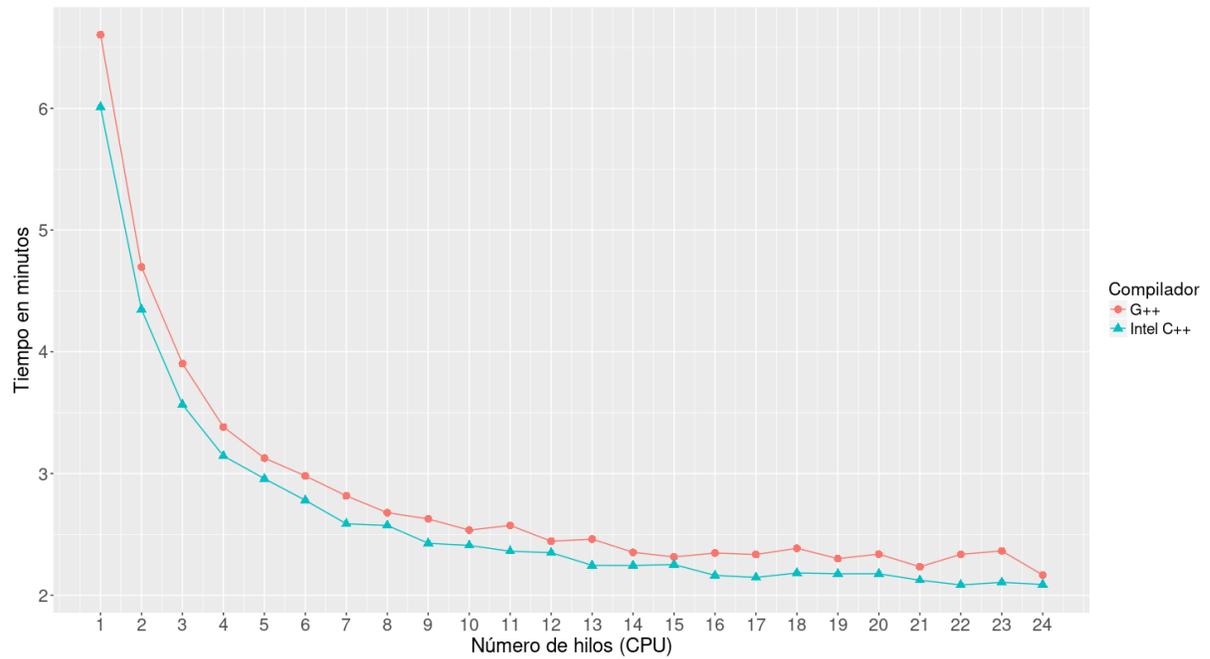


Figura 19. Tiempo de cómputo para la eliminación de k -meros redundantes entre 10 y 60 aminoácidos de longitud para el proteoma del camarón (Ghaffari *et al.*, 2014), utilizando diferentes números de hilos de CPU (1 a 24) y los compiladores G++ e Intel C++.

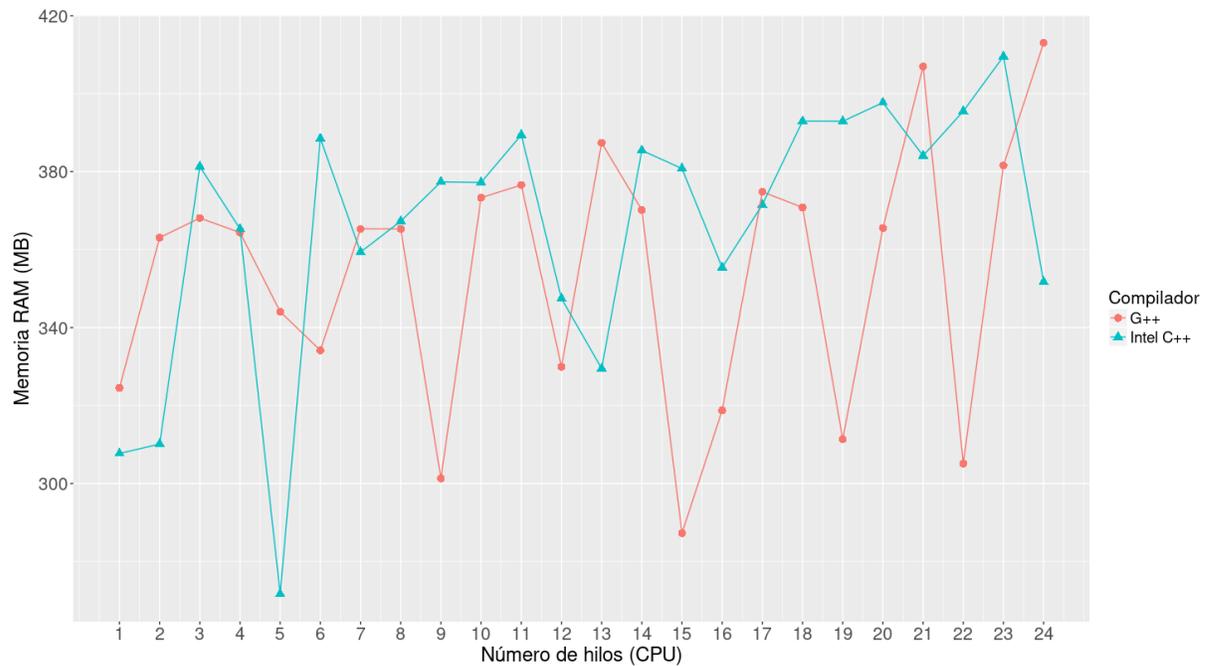


Figura 20. Memoria RAM consumida para la eliminación de k -meros redundantes entre 10 y 60 aminoácidos de longitud para el proteoma del camarón (Ghaffari *et al.*, 2014), utilizando diferentes números de hilos de CPU (1 a 24) y los compiladores G++ e Intel C++. La memoria RAM es la reportada por SLURM (comando "sacct") de 5 ejecuciones.

4.5.2. Evaluación del tiempo de cómputo y consumo de memoria RAM con diferentes hilos de CPU

A pesar del cálculo redundante evitado al eliminar k -meros duplicados, realizar los cálculos involucrados del algoritmo propuesto (ver Sección 3.4.4), a los millones de k -meros únicos, es un cálculo costoso. Por tal motivo, y aprovechando los múltiples núcleos de los procesadores actuales, así como los avances de los compiladores, se recurre al uso de los núcleos ofrecidos por los procesadores a través de la programación paralela, aprovechando al máximo los recursos y reduciendo el tiempo de cómputo de forma significativa, llegando a reducir el tiempo de cómputo a aproximadamente 7 horas con el uso de 24 núcleos, siendo una reducción de más de 18 veces comparado con un solo núcleo. En la Figura 21 se muestra el tiempo de cómputo empleado por el algoritmo propuesto utilizando diferente número de hilos (de 1 a 24) y compiladores. En el eje "X" se muestra el número de hilos, y en el eje "Y" se muestra el tiempo en horas.

Mientras en la Figura 22 se muestra el consumo de memoria por el algoritmo utilizando diferentes números de hilos (de 1 a 24) y compiladores, donde en el eje "X" se muestra el número de hilos, mientras en el eje "Y" se muestra la memoria RAM (en MB) consumida.

Se observa cómo el uso de hilos favorece de forma significativa en la reducción del tiempo de cómputo del algoritmo propuesto, al mismo tiempo que mantiene precisión, con una diferencia pequeña entre ambos compiladores, mostrando el compilador Intel C++ resultados favorables comparado con G++. Esperando una reducción aún mayor con un mayor número de hilos en ambos compiladores.

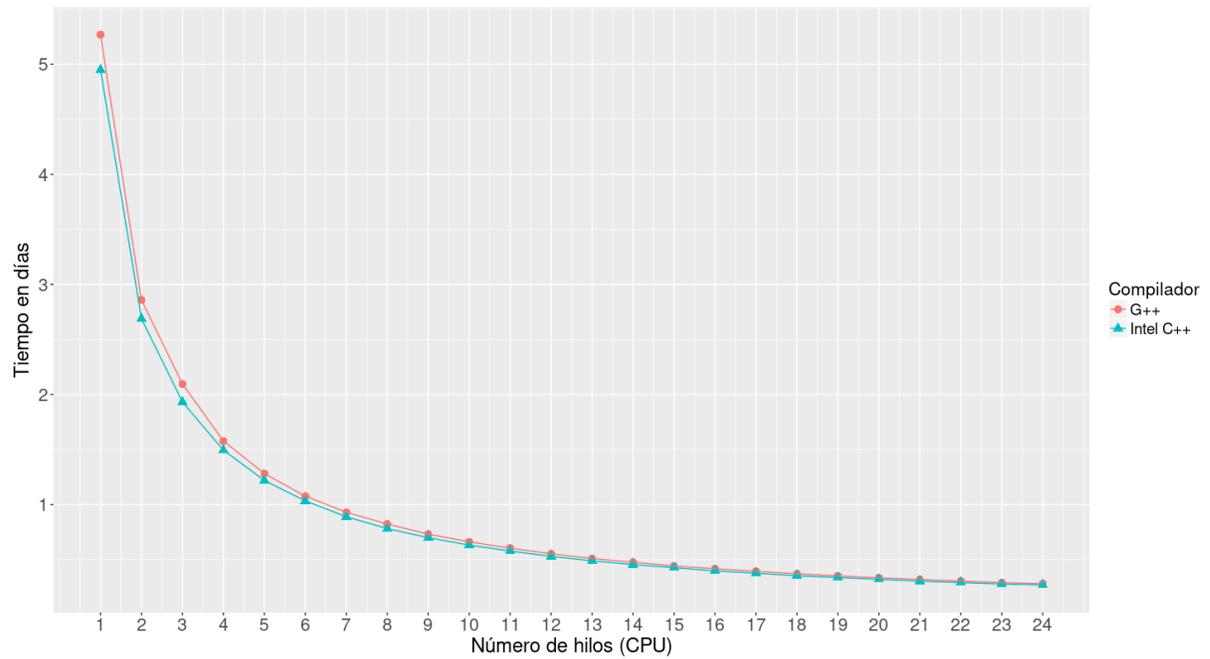


Figura 21. Tiempo de cómputo para InProt con k -meros entre 10 y 60 aminoácidos de longitud para el proteoma del camarón (Ghaffari *et al.*, 2014), utilizando diferentes números de hilos de CPU (1 a 24) y los compiladores G++ e Intel C++.

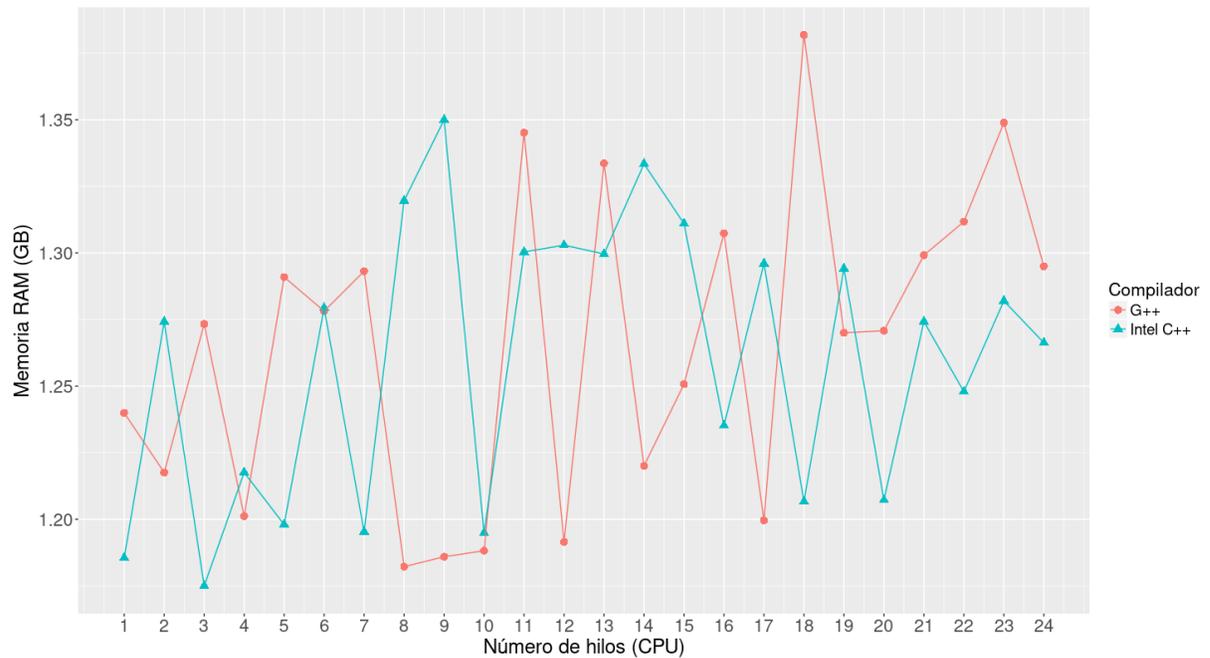


Figura 22. Memoria RAM consumida para InProt con k -meros entre 10 y 60 aminoácidos de longitud para el proteoma del camarón (Ghaffari *et al.*, 2014), utilizando diferentes números de hilos de CPU (1 a 24) y los compiladores G++ e Intel C++.

4.6. Conteo de k -meros predichos por secuencia

Debido a la gran cantidad de secuencias predichas como AMP por InProt (ver Figura 16), se procedió a analizar la distribución de los mismos dentro de las secuencias del proteoma (Ghaffari *et al.*, 2014). Para ello se realizó la contabilización de los residuos de cada secuencia del proteoma que alineaban con los residuos de cada k -mero para los k -meros predichos como AMP por InProt, así como la densidad de los k -meros predichos como AMP de las secuencias del proteoma. De las 16,037 secuencias únicas del proteoma del camarón (ver Figura 23), 16,020 secuencias cuentan con al menos un k -mero predicho como AMP por InProt. La cantidad de k -meros predichos como AMP por secuencia por InProt va de 0 a 83,218. Esto es, hubieron secuencias de las cuales no se obtuvieron k -meros predichos como AMP por InProt, mientras que una secuencia, la “m.66890”, generó 83,218 k -meros predichos como AMPs. En la Figura 24 se muestra la distribución de los k -meros predichos como AMP por secuencia de las 16,020 secuencias con al menos un k -mero predicho como AMP. En la Figura 25 se muestra la distribución de los k -meros dentro de las 10 secuencias con mayor número de k -meros obtenidos y predichos como AMP por InProt (ver Tabla 7). Si hay empates, la selección se hace al azar. Mientras que en la Figura 26 se muestra la distribución de los k -meros obtenidos y predichos como AMP por InProt de 30 secuencias tomadas al azar del total de secuencias con k -meros predichos como AMP (ver Tabla 10). Del mismo modo, en este caso se procedió a obtener aquellas secuencias con al menos un k -mero predicho como AMP, para posteriormente, calcular la proporción de los k -meros predichos ($\# k\text{-meros predichos} / \text{longitud de la secuencia}$). En la Tabla 8 se muestran las 10 secuencias con mayor proporción de k -meros predichos como AMP, y en la Tabla 9 las 10 secuencias de menor proporción, ordenadas de menor a mayor proporción y seleccionadas al azar en caso de empates.

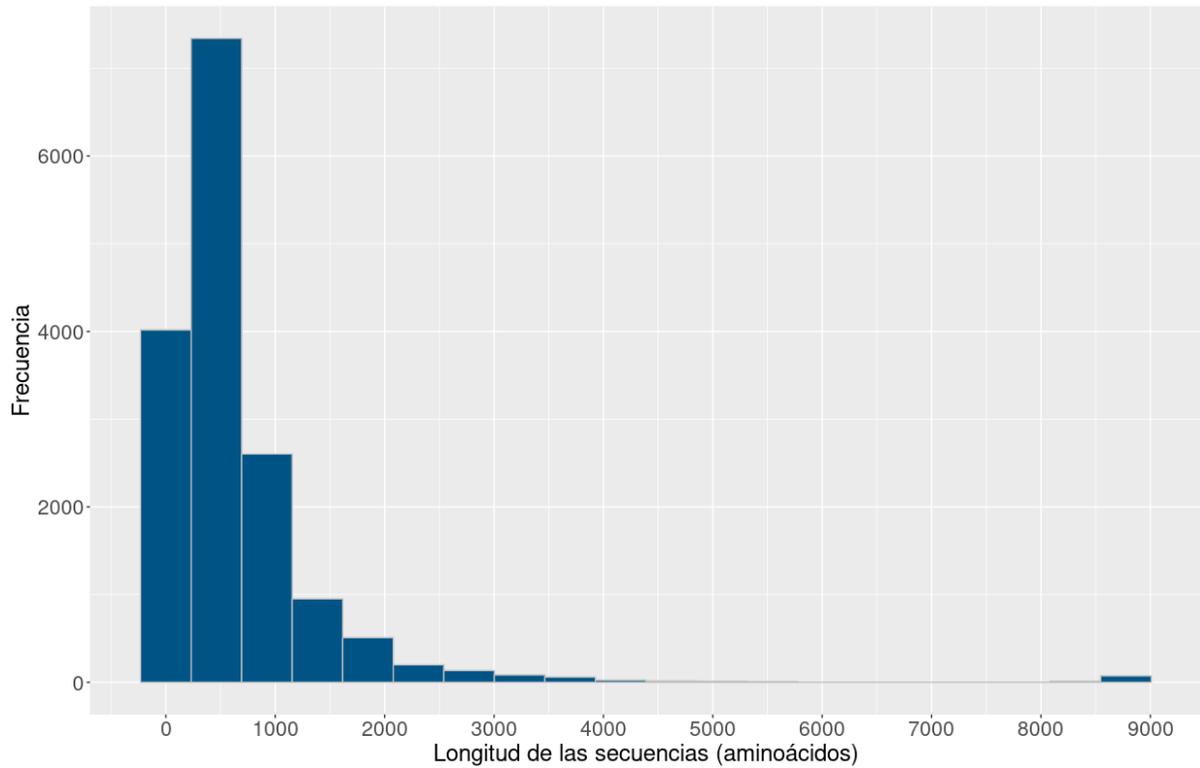


Figura 23. Histograma del proteoma del camarón (Ghaffari *et al.*, 2014) conformado por 16,037 secuencias únicas entre 99 y 8,876 aminoácidos de longitud.

Tabla 7. Total de *k*-meros predichos de las 10 secuencias con mayor número de *k*-meros predichos como AMP

Secuencia	Longitud (aa)	# <i>k</i>-meros predichos (AMP)
m.66890	4,000	83,218
m.51222	3,142	71,901
m.51228	3,126	71,363
m.51226	3,084	69,905
m.51220	3,068	69,367
m.43415	5,111	56,427
m.43420	5,070	56,330
m.66943	2,907	55,800
m.66912	3,437	54,903
m.71441	8,858	51,616

Tabla 8. Las 10 secuencias con mayor proporción de *k*-meros predichos como AMP

Secuencia	Longitud (aa)	# <i>k</i>-meros predichos (AMP)	Proporción
m.36206	406	18,087	44.54926
m.1217	360	16,057	44.60278
m.68427	335	15,146	45.21194
m.68424	335	15,187	45.33433
m.7845	401	18,198	45.38155
m.39483	815	37,138	45.56810
m.39477	818	37,441	45.77139
m.1218	360	16,499	45.83056
m.36212	780	35,883	46.00385
m.36213	780	36,168	46.36923

Tabla 9. Las 10 secuencias con menor proporción de *k*-meros predichos como AMP

Secuencia	Longitud (aa)	# <i>k</i>-meros predichos (AMP)	Proporción
m.17744	148	1	0.006756757
m.57216	143	1	0.006993007
m.16103	112	1	0.008928571
m.89594	109	1	0.009174312
m.87210	109	1	0.009174312
m.93726	105	1	0.009523810
m.24930	103	1	0.009708738
m.57498	103	1	0.009708738
m.69098	167	2	0.011976048
m.5108	151	2	0.013245033

Tabla 10. Total de *k*-meros predichos de 30 secuencias seleccionadas al azar.

Secuencia	Longitud (aa)	# <i>k</i>-meros predichos (AMP)
m.70564	8,830	51,257
m.69294	3,314	22,587
m.51963	490	12,611
m.62678	1,130	11,630
m.38299	533	9,606
m.36458	1,702	8,407
m.72540	333	7,693
m.44505	558	6,614
m.90267	235	6,553
m.59985	1,256	6,434
m.8026	342	5,400
m.11579	706	5,263
m.24780	533	5,122
m.9743	712	3,900
m.89983	131	3,792
m.92744	207	3,749
m.20540	631	3,468
m.55845	884	3,292
m.74390	1,018	3,282
m.56542	314	3,281
m.89509	311	3,209
m.61750	541	2,453
m.63277	483	2,383
m.94676	120	2,094
m.84360	428	1,896
m.98474	102	1,804
m.46944	184	1,396
m.32637	329	1,108
m.9176	200	1,052

Continúa en la siguiente página

Tabla 10 – Continuación

Secuencia	Longitud (aa)	# <i>k</i> -meros predichos (AMP)
m.10811	206	334

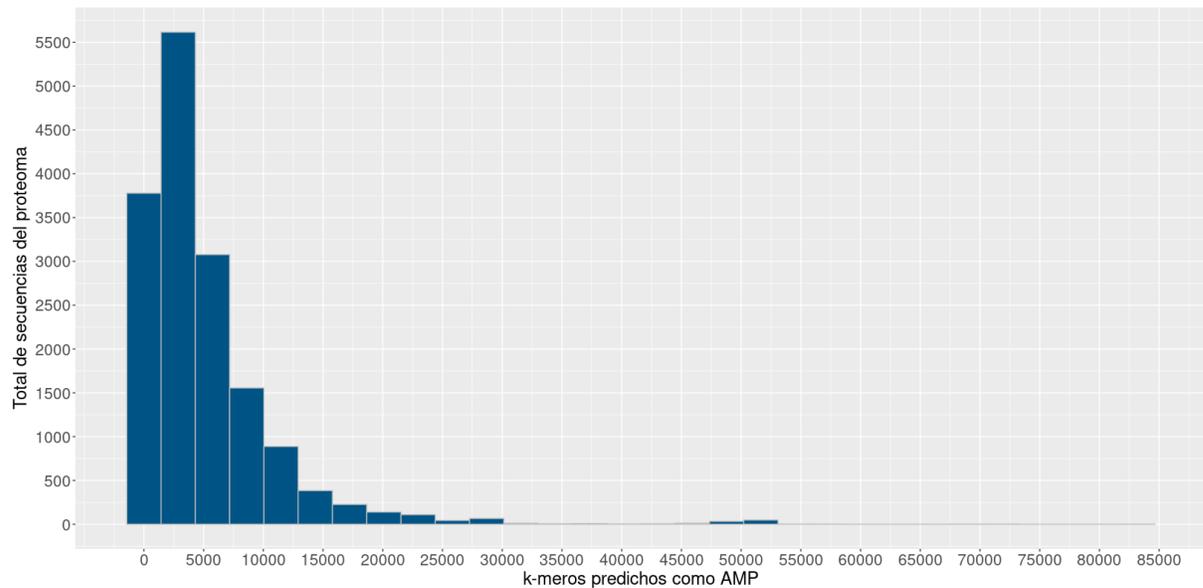


Figura 24. Distribución de los *k*-meros predichos como AMP por secuencia de las 16,020 secuencias del proteoma del camarón con *k*-meros predichos como AMP por el InProt. En el eje “X” se encuentra el total de *k*-meros predichos como AMP por secuencia, y en el eje “Y” el total de secuencias del proteoma con ese número de predicciones.

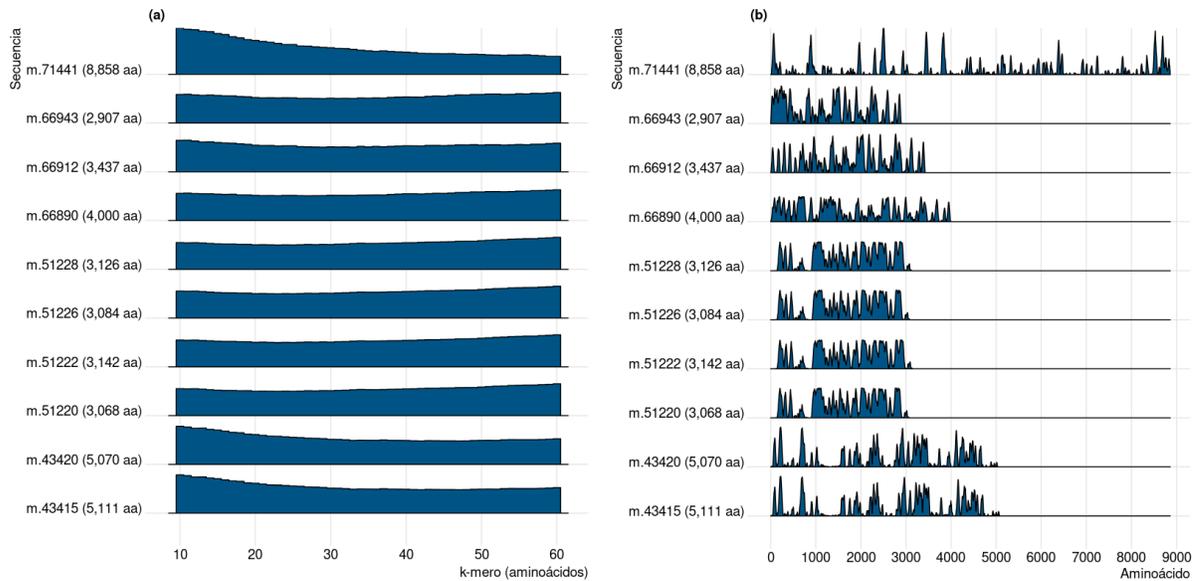


Figura 25. Las 10 secuencias con mayor número de k -meros predichos por el algoritmo. Ordenadas de mayor a menor número de k -meros predichos, de arriba hacia abajo. (a) Densidad de los k -meros predichos de las secuencias de acuerdo al tamaño de los k -meros. (b) Densidad de los k -meros predichos de las secuencias de acuerdo a la posición de los k -meros dentro de la secuencia. *Gráfica con escala = 0.9.*

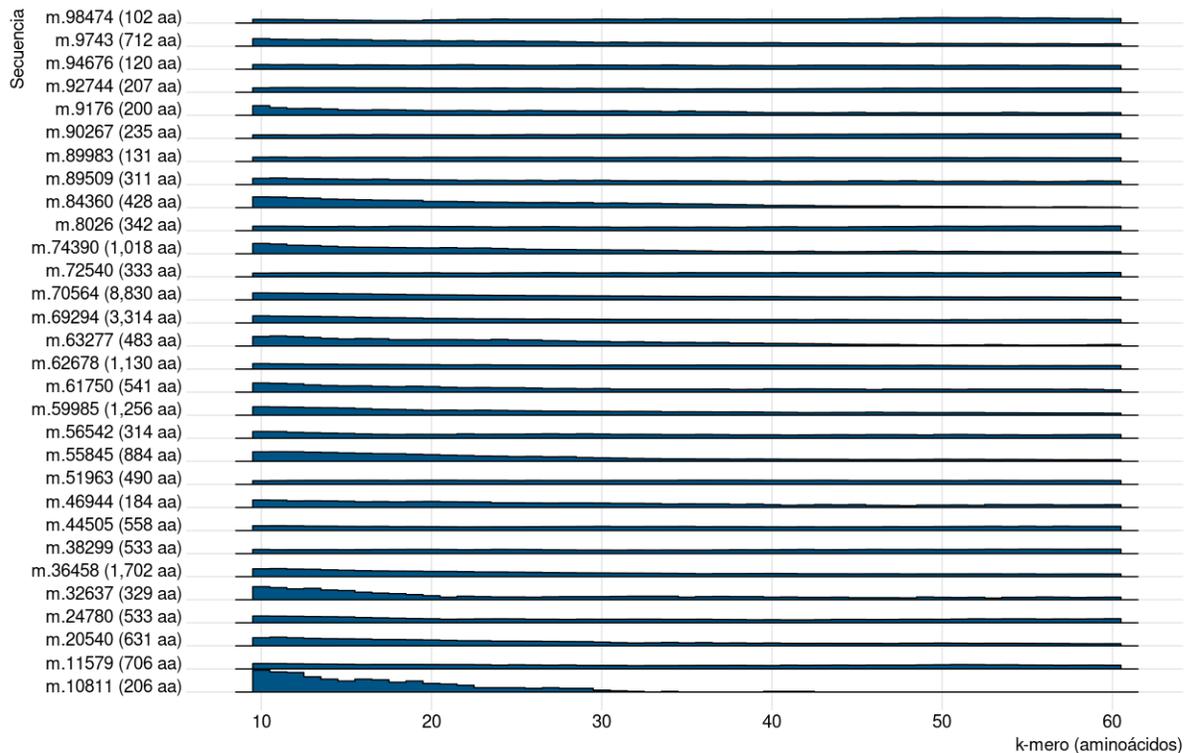


Figura 26. Densidad de los k -meros predichos de 30 secuencias tomadas de forma aleatoria. Ordenadas de mayor a menor número de k -meros predichos, de arriba hacia abajo, por número de k -meros predichos por secuencia. Siendo “m.70564” la secuencia con mayor número de k -meros predichos por un total de 51,257 k -meros (desde 10-meros hasta 60-meros) con una longitud de 8,830 aminoácidos. *Gráfica con escala = 0.9.*

Posteriormente, para responder la pregunta sobre cómo están distribuidos los k -meros dentro de las secuencias cortas, se tomaron al azar las 10 secuencias más cortas de diferentes longitudes (*ver Tabla 11*). En la Figura 27 se muestra la distribución de los k -meros predichos como AMP por InProt para las 10 secuencias más cortas, y en la Figura 28 se muestra la distribución de estos a lo largo de las secuencias. Finalmente, se tomaron las 10 secuencias con mayor número de k -meros predichos como AMP por InProt con una longitud entre 900 y 1,000 aminoácidos (*ver Tabla 12*). En la Figura 29 se muestra la distribución de los k -meros predichos como AMP por InProt de las 10 secuencias entre 900 y 1,000 aminoácidos de longitud y con mayor cantidad de k -meros predichos como AMP, y en la Figura 30 se muestra la distribución de estos a lo largo de las secuencias.

Tabla 11. Las 10 secuencias más cortas de diferentes tamaños, longitud y cantidad de AMPs predichos por secuencia.

Secuencia	Longitud (aa)	# Predichos
m.88224	104	2,614
m.85919	108	2,588
m.3094	102	2,136
m.2946	105	2,060
m.87488	103	1,534
m.94075	99	866
m.86121	106	750
m.44	101	330
m.2975	107	240
m.85784	100	163

Tabla 12. Las 10 secuencias entre 900 y 1,000 aminoácidos de longitud con mayor cantidad de AMPs predichos por secuencia.

Secuencia	Longitud (aa)	# Predichos
m.18150	990	33,082
m.60997	904	28,309

Continúa en la siguiente página

Tabla 12 – Continuación

Secuencia	Longitud (aa)	# Predichos
m.42182	918	23,751
m.55470	960	21,814
m.73829	963	20,259
m.73826	930	19,221
m.82702	980	17,532
m.64925	939	16,810
m.64940	925	16,793
m.16594	950	15,756

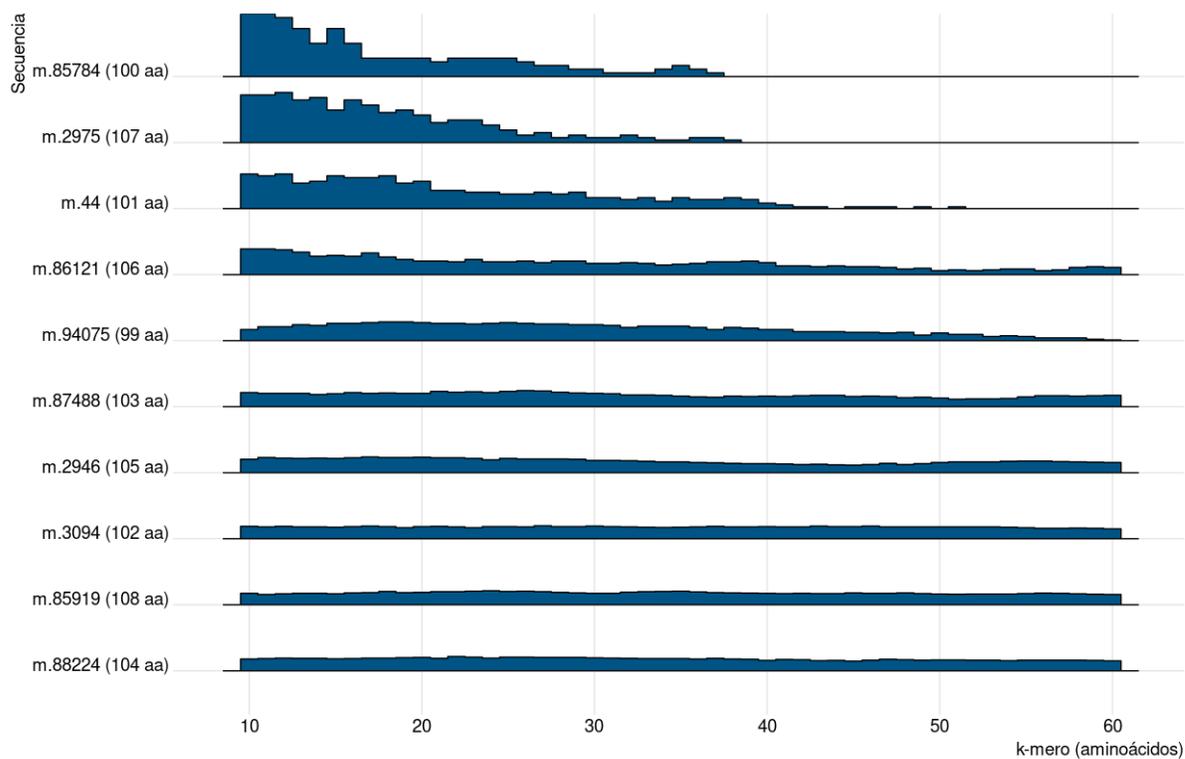


Figura 27. Las 10 secuencias más cortas de diferentes tamaños entre ellas. Ordenadas de mayor a menor número de k -meros predichos como AMP, de arriba hacia abajo. Densidad de los k -meros predichos de las secuencias de acuerdo al tamaño de los k -meros.

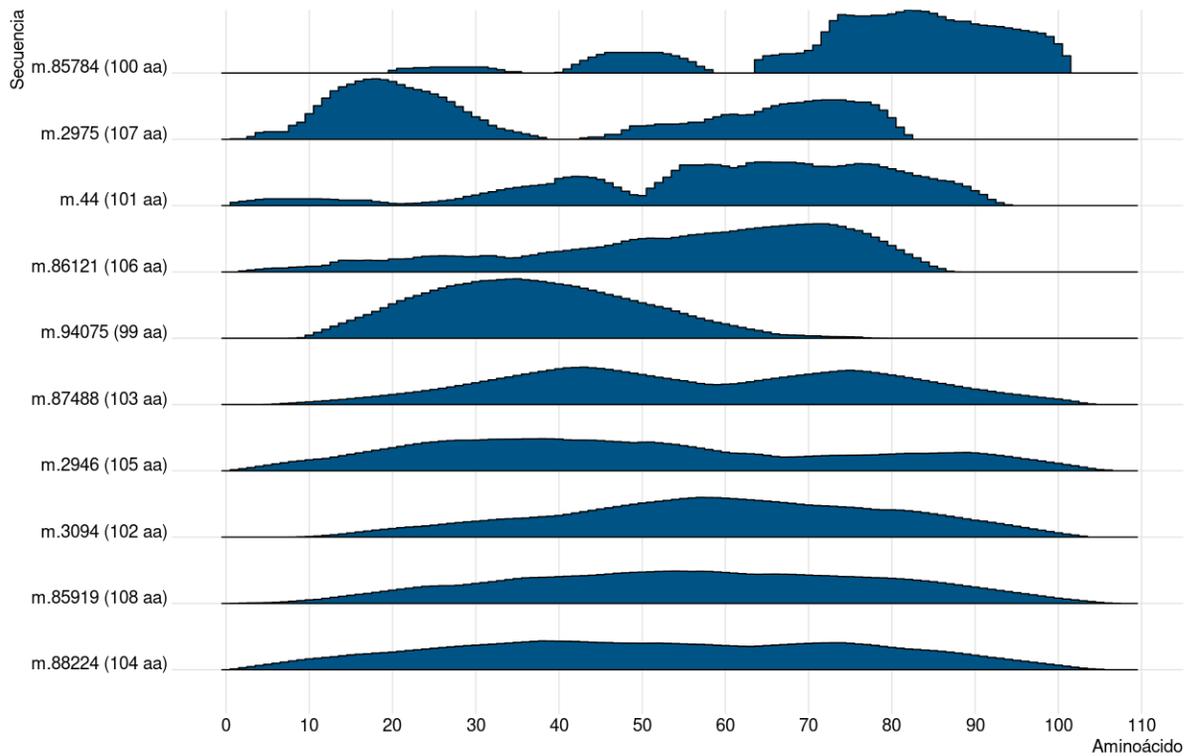


Figura 28. Las 10 secuencias más cortas de diferentes tamaños entre ellas. Ordenadas de mayor a menor número de k -meros predichos como AMP, de arriba hacia abajo. Densidad de los k -meros predichos de las secuencias de acuerdo a la posición de los k -meros dentro de la secuencia. *Gráfica con escala = 0.9.*

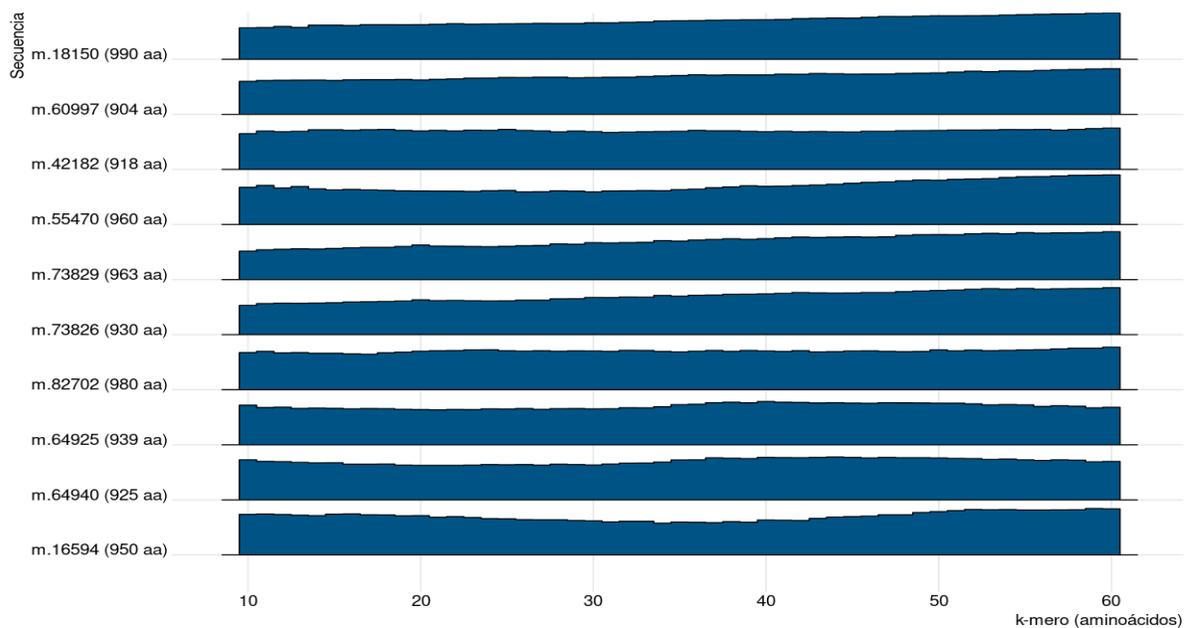


Figura 29. Las 10 secuencias entre 900 y 1,000 aminoácidos de longitud con mayor cantidad de k -meros predichos como AMP. Ordenadas de mayor a menor número de k -meros predichos como AMP, de arriba hacia abajo. Densidad de los k -meros predichos de las secuencias de acuerdo al tamaño de los k -meros. *Gráfica con escala = 0.9.*

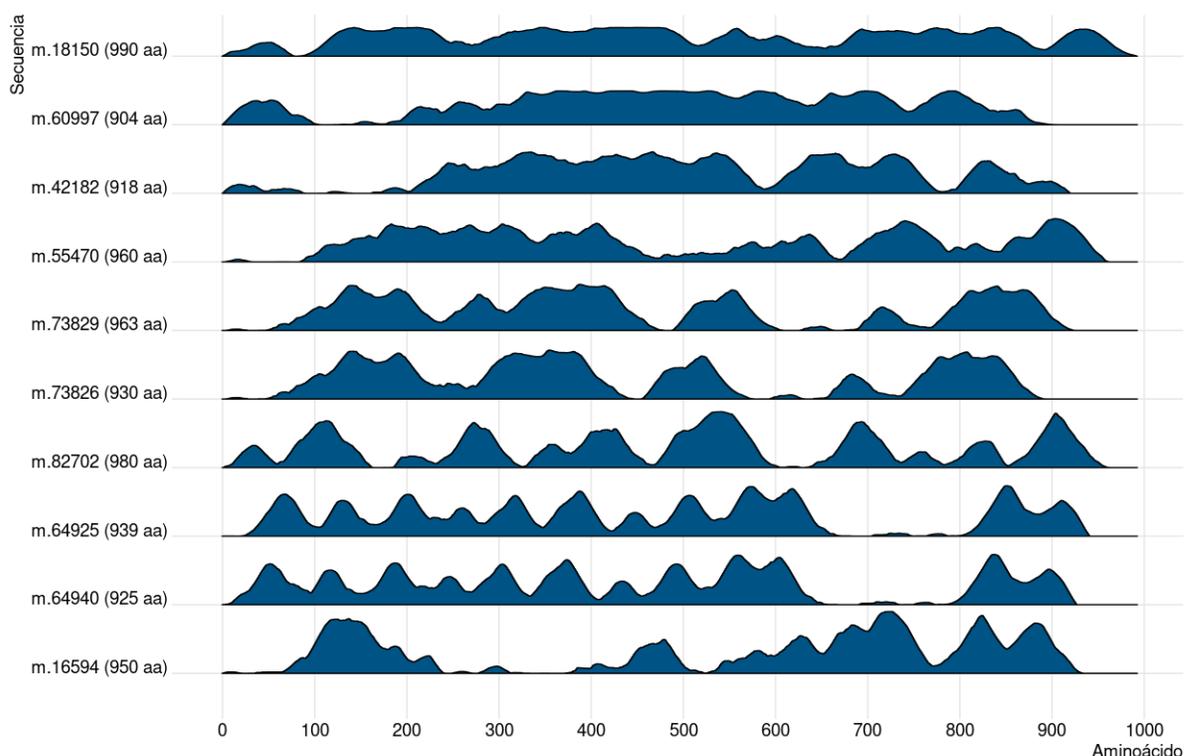


Figura 30. Las 10 secuencias entre 900 y 1,000 aminoácidos de longitud con mayor cantidad de k -meros predichos como AMP. Ordenadas de mayor a menor número de k -meros predichos como AMP, de arriba hacia abajo. Densidad de los k -meros predichos de las secuencias de acuerdo a la posición de los k -meros dentro de la secuencia. *Gráfica con escala = 0.9.*

4.6.1. Podado del proteoma

Dado a que sólo un fragmento (o varios) de las secuencias cuenta con k -meros predichos como AMP por InProt, se procedió a extraer aquellos fragmentos por medio de agrupamiento de los k -meros que se traslapen entre sí ó se encuentren de manera contigua (ver Sección 3.4.4.5). Esto es, se dejaron únicamente aquellas regiones que cuenten con k -meros predichos como AMP por InProt. Cada región con k -meros predichos como AMP representa una nueva secuencia en el nuevo proteoma (proteoma podado). Del total de 16,037 secuencias únicas del proteoma (ver Figura 23), 16,020 secuencias cuentan con al menos un k -mero predicho como AMP (ver Figura 24), los cuales, se agruparon por medio del procedimiento explicado en la Sección 3.4.4.5, creando un total de 63,583 secuencias entre 10 y 1,858 aminoácidos de longitud. En la Figura 31 se muestra la distribución del proteoma podado.

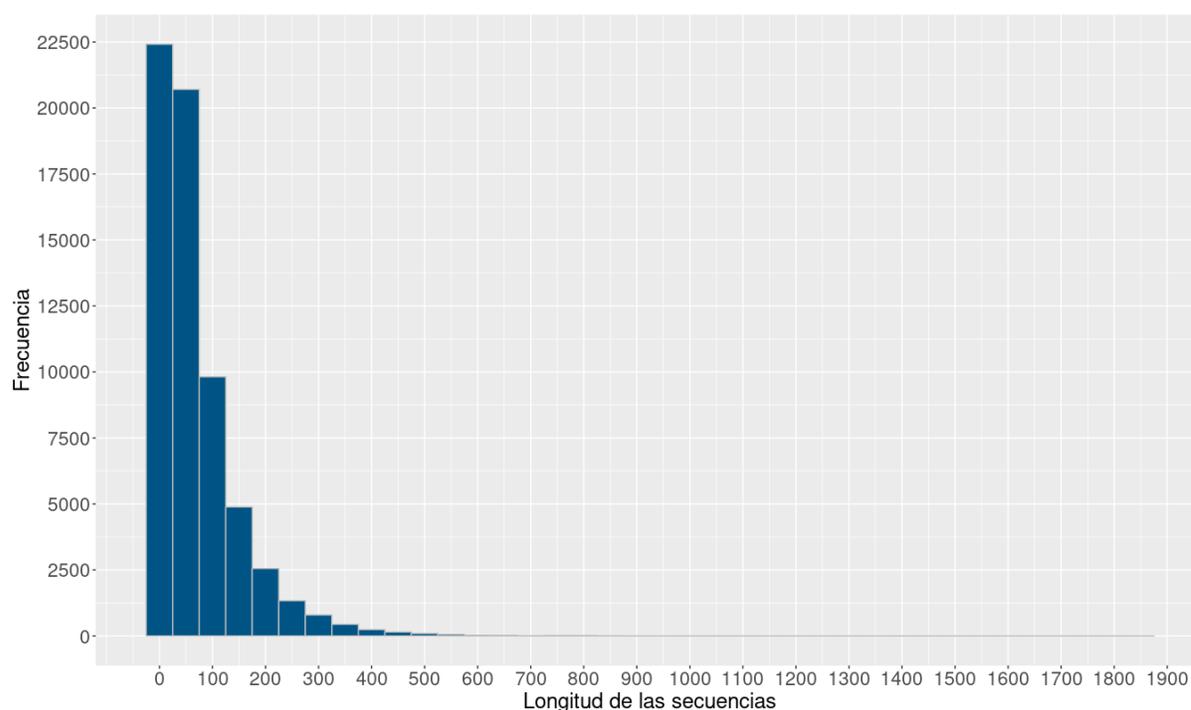


Figura 31. Histograma del proteoma (podado) del camarón (Ghaffari *et al.*, 2014), conformado por 63, 583 secuencias únicas entre 10 y 1, 858 aminoácidos de longitud. En el eje “X” se muestra la longitud (en aminoácidos) de las secuencias, y en el eje “Y” se muestra el total de secuencias con esa longitud.

4.7. Discusión

Las dificultades encontradas al momento de implementar una tabla hash para el almacenamiento de k -meros, nos llevó al uso de vectores lineales para realizar dicho almacenamiento. El ordenamiento y comparación de los k -meros almacenados en estos vectores, ha sido una opción eficiente para descartar k -meros redundantes, reduciendo la cantidad de k -meros a procesar (*ver Figura 16*). Aproximadamente el 53% de los k -meros son duplicados para valores de $k = 10$ hasta $k = 60$. De los k -meros no redundantes (únicos), el 17% fue predicho como AMP por InProt.

Dada la gran cantidad de k -meros predichos como AMP por InProt, se recurrió a otros predictores adicionales como los proporcionados por CAMP, obteniendo una reducción aún mayor como se muestra en la Figura 17. En promedio, el 18% de los k -meros predichos como AMP por InProt, también fueron predichos como AMP por los cuatro algoritmos de CAMP (SVM, ANN, RF y DA) en conjunto (intersección), mientras que, en promedio, de los k -meros predichos como no-AMP por InProt, el 0.29% fueron predichos como AMP por los predictores (en conjunto) previamente mencionados. Esto

nos podrá ayudar en la selección de AMPs candidatos, priorizando o tomando aquellos AMPs candidatos que hayan sido predichos como AMP por todos los predictores.

Los resultados presentados en la Sección 4.6, para el caso del camarón, sugieren que no hay una relación entre la longitud de las secuencias y los AMPs, esto es, no hay un patrón entre la longitud de las secuencias y la posición en donde se encuentren “concentrados” los AMPs; por lo que se pueden conservar aquellos k -meros predichos como AMP por InProt y por los algoritmos de CAMP que se encuentren dentro de las secuencias con mayor frecuencia del proteoma podado (*ver Sección 4.6.1*), reduciendo aún más la cantidad de k -meros predichos como AMP a analizar.

Con el enfoque propuesto de proteólisis *in silico*, todas las secuencias del proteoma son analizadas por completo, evitando así descartar AMPs potenciales, siendo esto último, una de las limitantes de otros métodos del estado del arte. Esto es, al limitar las secuencias a analizar en función de sus longitudes, como parte de los filtros aplicados, se descartan AMPs potenciales que se pueden encontrar más allá del límite de longitud definido, al final de la secuencia o incluso, en ambos lados del límite, aspecto que no sucede con nuestro enfoque. Por ejemplo, de las 16,037 secuencias únicas del proteoma del camarón, un total de 16,020 secuencias contienen al menos un k -mero predicho como AMP, representando el 99.89% de las secuencias únicas donde puede haber un AMP candidato.

Capítulo 5. Conclusiones y trabajo futuro

5.1. Sumario

Este trabajo se enfocó en el diseño e implementación de un algoritmo para el descubrimiento de péptidos antimicrobianos en secuencias de aminoácidos derivadas de transcriptomas. Tomando como entrada un transcriptoma, se construye el proteoma y se emula una proteólisis completa *in silico*, mediante el deslizamiento de ventanas (*k*-meros) de diferentes longitudes, sobre las secuencias en el proteoma construido. De acuerdo a la distribución de las longitudes en la base de datos de CAMP (Waghu *et al.*, 2014), se analizaron *k*-meros con longitudes de 10 a 60 aminoácidos, utilizando un vector lineal como estructura de datos para el almacenamiento de los mismos. Para la eliminación de *k*-meros redundantes, se utilizaron algoritmos de ordenamiento y comparación lexicográfica. A los *k*-meros restantes, se les calcularon 51 descriptores moleculares previamente propuestos por Verdugo (2014) para su posterior evaluación con una máquina de aprendizaje, descartando aquellos *k*-meros que no sean predichos como AMP. Posteriormente, los *k*-meros predichos como AMPs y provenientes de una misma secuencia, se agrupan si estos cuentan con traslapes entre si o son contiguos, escribiéndose, si este es el caso, en un archivo multifasta para su posterior análisis. Finalmente, se realizaron una serie de experimentos computacionales para evaluar el tiempo de cómputo y consumo de memoria RAM del algoritmo propuesto y equipado de una implementación propia de la máquina de aprendizaje y con una equivalente de libsvm (Chang y Lin, 2011).

A continuación se exponen las conclusiones a las que se llegaron en este trabajo de investigación con base en un análisis de los resultados obtenidos.

5.2. Conclusiones

InProt es una herramienta básica para la búsqueda de AMPs en proteomas. Gracias a los recursos ofrecidos por los procesadores multinúcleos de hoy en día, se logró reducir el tiempo de cómputo en al menos 18 veces usando 24 núcleos, comparado con el obtenido al usar un solo núcleo; con la posibilidad de que el tiempo de cómputo se reduzca aún más con el incremento del número de núcleos.

De las 16,037 secuencias únicas del proteoma, un total de 513,477,690 k -meros entre 10 y 60 aminoácidos de longitud fueron obtenidos mediante el enfoque de ventaneo, siendo aproximadamente el 47% (243,643,492) de estos únicos; mientras que aproximadamente el 17% (43,487,577) de los k -meros únicos fueron predichos como AMPs por la máquina de aprendizaje.

El uso de un vector lineal para el almacenamiento de los k -meros, apoyado de algoritmos de ordenamiento y comparación lexicográfica, ha demostrado ser una solución eficiente (*ver Figura 19*) para la búsqueda de AMPs en transcriptomas, evitando la realización de cálculo redundante, reduciendo así el tiempo de cómputo y memoria requeridos.

A pesar de la reducción de k -meros por medio de los algoritmos de ordenamiento y comparación lexicográfica, así como la evaluación de los mismos con la máquina de aprendizaje, el total de k -meros restantes sigue siendo una cantidad inviable de analizar en el laboratorio. Como solución a dicha problemática, se realiza el podado del proteoma, con el cual, se extraen aquellas regiones en donde los k -meros predichos como AMPs, y provenientes de una misma secuencia, presenten traslapes entre si o sean contiguas, llegando a reducir las longitudes de las secuencias de interés a analizar, con un total de 63,583 secuencias únicas entre 10 y 1,858 aminoácidos de longitud, comparado con las 16,037 secuencias únicas entre 99 y 8,876 aminoácidos de longitud del proteoma original. Permitiendo poder enfocarse en secuencias con posibles AMPs.

Dado que la implementación propia de la máquina de aprendizaje fue diseñada de tal manera que aproveche las arquitecturas de los procesadores actuales, así como el conjunto de instrucciones vectorizadas proporcionadas por los procesadores, y los avances de los compiladores, dicha implementación muestra un gran rendimiento comparado con su equivalente de libsvm (*ver Sección 4.3.1*), teniendo como ventaja una disminución en el tiempo de cómputo, al mismo tiempo que mantiene la misma precisión. Sin embargo, esta ventaja desaparece al incrementarse el número de hilos, ya que ambas implementaciones tienden a converger al mismo valor con el incremento de hilos a utilizar, por lo que la diferencia podría llegar a ser nula con un mayor número de hilos.

Para el podado del proteoma, se realiza la búsqueda de aquellos k -meros predichos como AMP que provengan de una secuencia determinada. Dicha búsqueda se realiza de manera exhaustiva a través de la secuencia para todos los k -meros, siendo un cálculo costoso y causando un tiempo de cómputo aún mayor.

CAMPRED ha resultado ser una herramienta útil ya que permite la automatización de la evaluación de las predicciones realizadas por InProt.

5.3. Trabajo futuro

Como trabajo futuro para abordar la problemática de la gran cantidad de k -meros únicos predichos como AMP por InProt, se sugiere explorar los siguientes enfoques: 1) priorizar aquellas secuencias del proteoma podado por su densidad, donde las más densas tengan una prioridad más alta, y por el total de k -meros predichos como AMP obtenidos de las mismas; 2) usar información de la estructura de los AMPs para buscar similitudes en los vecindarios de las secuencias identificadas como AMPs; 3) agrupar las secuencias del proteoma podado de acuerdo a los procesos celulares en los que participen, así como el agrupamiento de las más densas de acuerdo a su distribución en el proteoma.

InProt, al igual que la implementación propia de la máquina de aprendizaje, se pueden vectorizar de forma manual, en lugar de indicarle al compilador que lo haga de forma automática, logrando así un mayor rendimiento. Del mismo modo, se pudiera sacar provecho de otras técnicas de programación como “CPU dispatch” la cual detecta el conjunto de instrucciones vectorizadas más adecuada al problema al momento de iniciar la ejecución, y usar la versión de funciones diseñadas para dicho conjunto, aprovechando al máximo las características del procesador. Por último, se puede hacer uso de memoria alineada, de forma que la vectorización sea más eficiente, al igual que se pueden implementar en GPU (con tecnología CUDA por ejemplo) el cálculo de los descriptores moleculares o distribuir el cálculo en diferentes equipos conectados entre si por medio de MPI.

Como solución a la desventaja mencionada anteriormente para el podado del proteoma, se proponen realizar un índice de aquellas secuencias en donde se encuentre al menos un k -mero predicho como AMP, de modo que al realizar la búsqueda de los k -meros de una secuencia determinada, estas se efectúen más rápido.

Literatura citada

- Alaa Eltablawy, A. V. (2017). Capabilities of intel® avx-512 in intel® xeon® scalable processors (skylake). *Colfax International*. Recuperado de <https://colfaxresearch.com/skl-avx512/>, febrero 2018.
- Altschul, S. F., Gish, W., Miller, W., Myers, E. W., y Lipman, D. J. (1990). Basic local alignment search tool. *Journal of molecular biology*, **215**(3): 403–410.
- Banga, A. K. (2015). *Therapeutic peptides and proteins: formulation, processing, and delivery systems*. CRC press.
- Blair-Chappell, S. y Stokes, A. (2012). *Parallel programming with intel parallel studio XE*. John Wiley & Sons.
- Branco, P., Francisco, D., Chambon, C., Hébraud, M., Arneborg, N., Almeida, M. G., Caldeira, J., y Albergaria, H. (2014). Identification of novel gapdh-derived antimicrobial peptides secreted by *saccharomyces cerevisiae* and involved in wine microbial interactions. *Applied microbiology and biotechnology*, **98**(2): 843–853.
- Byvatov, E., Fechner, U., Sadowski, J., y Schneider, G. (2003). Comparison of support vector machine and artificial neural network systems for drug/nondrug classification. *Journal of chemical information and computer sciences*, **43**(6): 1882–1889.
- Casari, G. y Sippl, M. J. (1992). Structure-derived hydrophobic potential: hydrophobic potential derived from x-ray structures of globular proteins is able to identify native folds. *Journal of molecular biology*, **224**(3): 725–732.
- Castañeda-Casimiro, J., Ortega-Roque, J. A., Venegas-Medina, A. M., Aquino-Andrade, A., Serafín-López, J., Estrada-Parra, S., y Estrada, I. (2009). Péptidos antimicrobianos: péptidos con múltiples funciones. *Alergia, asma e inmunología pediátricas*, **18**(1): 16–29.
- Chang, C.-C. y Lin, C.-J. (2011). Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, **2**(3): 27.
- Charton, M. y Charton, B. (1983a). A parameter of charge transfer capability. *J. Theor. Biol*, **111**: 447.
- Charton, M. y Charton, B. I. (1983b). The dependence of the chou-fasman parameters on amino acid side chain structure. *Journal of theoretical biology*, **102**(1): 121–134.
- Cid, H., Bunster, M., Canales, M., y Gazitúa, F. (1992). Hydrophobicity and structural classes in proteins. *Protein Engineering, Design and Selection*, **5**(5): 373–375.
- Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.
- Dang, H. X. y Lawrence, C. B. (2014). Allerdicator: fast allergen prediction using text classification techniques. *Bioinformatics*, **30**(8): 1120–1128.
- Deilmann, M. (2012). A guide to vectorization with intel c++ compilers. *Intel Corporation*. Recuperado de: <https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers>, febrero 2018.
- Deorowicz, S., Kokot, M., Grabowski, S., y Debudaj-Grabysz, A. (2015). Kmc 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, **31**(10): 1569–1576.

- Doxsey, C. (2016). *Introducing Go: Build reliable, scalable programs*. O'Reilly Media, Inc.
- Dubchak, I., Muchnik, I., Holbrook, S. R., y Kim, S.-H. (1995). Prediction of protein folding class using global description of amino acid sequence. *Proceedings of the National Academy of Sciences*, **92**(19): 8700–8704.
- Eddy, S. (2010). Hmmer3: a new generation of sequence homology search software. URL: <http://hmmer.janelia.org>.
- Eisenberg, D., Weiss, R. M., y Terwilliger, T. C. (1984). The hydrophobic moment detects periodicity in protein hydrophobicity. *Proceedings of the National Academy of Sciences*, **81**(1): 140–144.
- Erbert, M., Rechner, S., y Müller-Hannemann, M. (2017). Gerbil: a fast and memory-efficient k-mer counter with gpu-support. *Algorithms for Molecular Biology*, **12**(1): 9.
- Flynn, M. J. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, **54**(12): 1901–1909.
- Ghaffari, N., Sanchez-Flores, A., Doan, R., Garcia-Orozco, K. D., Chen, P. L., Ochoa-Leyva, A., Lopez-Zavala, A. A., Carrasco, J. S., Hong, C., Briebe, L. G., et al. (2014). Novel transcriptome assembly and improved annotation of the whiteleg shrimp (*Litopenaeus vannamei*), a dominant crustacean in global seafood mariculture. *Scientific Reports*, **4**.
- Giuliani, A., Rinaldi, A. C., et al. (2010). *Antimicrobial peptides. Methods and protocols*, Vol. 618. Humana Press-Springer.
- Grabherr, M. G., Haas, B. J., Yassour, M., Levin, J. Z., Thompson, D. A., Amit, I., Adiconis, X., Fan, L., Raychowdhury, R., Zeng, Q., et al. (2011a). Full-length transcriptome assembly from rna-seq data without a reference genome. *Nature biotechnology*, **29**(7): 644–652.
- Grabherr, M. G., Haas, B. J., Yassour, M., Levin, J. Z., Thompson, D. A., Amit, I., Adiconis, X., Fan, L., Raychowdhury, R., Zeng, Q., et al. (2011b). Trinity: reconstructing a full-length transcriptome without a genome from rna-seq data. *Nature biotechnology*, **29**(7): 644.
- Graham-Cumming, J. (2015). *The GNU make book*. No Starch Press.
- Gregoire, M. (2014). *Professional C++*. John Wiley & Sons.
- Gusfield, D. (1997). *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press.
- Haas, B. y Papanicolaou, A. (2012). Transdecoder.
- Haas, B. J., Papanicolaou, A., Yassour, M., Grabherr, M., Blood, P. D., Bowden, J., Couger, M. B., Eccles, D., Li, B., Lieber, M., et al. (2013). De novo transcript sequence reconstruction from rna-seq using the trinity platform for reference generation and analysis. *Nature protocols*, **8**(8): 1494–1512.
- Harrington, P. (2012). *Machine learning in action*, Vol. 5. Manning Greenwich, CT.

- Heck, A. M. y Wilusz, J. (2018). The interplay between the rna decay and translation machinery in eukaryotes. *Cold Spring Harbor perspectives in biology*, p. a032839.
- Hilpert, K., Volkmer-Engert, R., Walter, T., y Hancock, R. E. (2005). High-throughput generation of small antibacterial peptides with improved activity. *Nature biotechnology*, **23**(8): 1008–1012.
- Hrdlickova, R., Toloue, M., y Tian, B. (2017). Rna-seq methods for transcriptome analysis. *Wiley Interdisciplinary Reviews: RNA*, **8**(1).
- Jamali, A. A., Ferdousi, R., Razzaghi, S., Li, J., Safdari, R., y Ebrahimie, E. (2016). Drugminer: comparative analysis of machine learning algorithms for prediction of potential druggable proteins. *Drug discovery today*, **21**(5): 718–724.
- Kennedy, W., Ketelsen, B., y Martin, E. S. (2015). *Go in Action*. Manning Publications Co.
- Kernighan, B. W. y Pike, R. (1999). *The practice of programming*. Addison-Wesley Professional.
- Kim, I.-W., Lee, J. H., Subramaniyam, S., Yun, E.-Y., Kim, I., Park, J., y Hwang, J. S. (2016). De novo transcriptome analysis and detection of antimicrobial peptides of the american cockroach *periplaneta americana* (linnaeus). *PLoS one*, **11**(5): e0155304.
- Klein, P., Kanehisa, M., y DeLisi, C. (1984). Prediction of protein function from sequence properties: Discriminant analysis of a data base. *Biochimica et Biophysica Acta (BBA)-Protein Structure and Molecular Enzymology*, **787**(3): 221–226.
- Kokot, M., Deorowicz, S., y Debudaj-Grabysz, A. (2017a). Sorting data on ultra-large scale with raduls. En: *International Conference: Beyond Databases, Architectures and Structures*. Springer, pp. 235–245.
- Kokot, M., Długosz, M., y Deorowicz, S. (2017b). Kmc 3: counting and manipulating k-mer statistics. *arXiv preprint arXiv:1701.08022*.
- Korpelainen, E., Tuimala, J., Somervuo, P., Huss, M., y Wong, G. (2014). *RNA-seq data analysis: a practical approach*. CRC Press.
- Kuhn, L. A., Swanson, C. A., Pique, M. E., Tainer, J. A., y Getzoff, E. D. (1995). Atomic and residue hydrophilicity in the context of folded protein structures. *Proteins: Structure, Function, and Bioinformatics*, **23**(4): 536–547.
- Lata, S., Sharma, B., y Raghava, G. (2007). Analysis and prediction of antibacterial peptides. *BMC bioinformatics*, **8**(1): 263.
- Lavergne, V., Dutertre, S., Jin, A.-h., Lewis, R. J., Taft, R. J., y Alewood, P. F. (2013). Systematic interrogation of the conus marmoreus venom duct transcriptome with conosorter reveals 158 novel conotoxins and 13 new gene superfamilies. *BMC genomics*, **14**(1): 708.
- Levin, J. Z., Yassour, M., Adiconis, X., Nusbaum, C., Thompson, D. A., Friedman, N., Gnirke, A., y Regev, A. (2010). Comprehensive comparative analysis of strand-specific rna sequencing methods. *Nature methods*, **7**(9): 709–715.

- Li, Z.-R., Lin, H. H., Han, L., Jiang, L., Chen, X., y Chen, Y. Z. (2006). Profeat: a web server for computing structural and physicochemical features of proteins and peptides from amino acid sequence. *Nucleic Acids Research*, **34**(suppl_2): W32–W37.
- Lodish, H., Berk, A., Kaiser, C., Krieger, M., Bretscher, A., Ploegh, H., Amon, A., y Martin, K. (2016). *Molecular Cell Biology*. W. H. Freeman.
- Ma, J., Sheridan, R. P., Liaw, A., Dahl, G. E., y Svetnik, V. (2015). Deep neural nets as a method for quantitative structure–activity relationships. *Journal of chemical information and modeling*, **55**(2): 263–274.
- Manavalan, P. y Ponnuswamy, P. (1978). Hydrophobic character of amino acid residues in globular proteins. *Nature*, **275**(5681): 673–674.
- Marçais, G. y Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, **27**(6): 764–770.
- Martin, J. A. y Wang, Z. (2011). Next-generation transcriptome assembly. *Nature Reviews Genetics*, **12**(10): 671–682.
- Mehlhorn, K. y Sanders, P. (2008). *Algorithms and data structures: The basic toolbox*. Springer Science & Business Media.
- Murphy, L. R., Wallqvist, A., y Levy, R. M. (2000). Simplified amino acid alphabets for protein fold recognition and implications for folding. *Protein Engineering*, **13**(3): 149–152.
- O'Neill, J. (2014). Antimicrobial resistance: tackling a crisis for the health and wealth of nations. *Review on antimicrobial resistance*, pp. 1–16.
- Pertea, M., Kim, D., Pertea, G. M., Leek, J. T., y Salzberg, S. L. (2016). Transcript-level expression analysis of rna-seq experiments with hisat, stringtie and ballgown. *Nature Protocols*, **11**(9): 1650–1667.
- Pilkington, N. C., Trotter, M. W., y Holden, S. B. (2012). Multiple kernel learning for drug discovery. *Molecular Informatics*, **31**(3-4): 313–322.
- Ponnuswamy, P., Prabhakaran, M., y Manavalan, P. (1980). Hydrophobic packing and spatial arrangement of amino acid residues in globular proteins. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, **623**(2): 301–316.
- Prabhakaran, M. (1990). The distribution of physical, chemical and conformational properties in signal and nascent peptides. *Biochemical Journal*, **269**(3): 691–696.
- Prashanth, J. R. y Lewis, R. J. (2015). An efficient transcriptome analysis pipeline to accelerate venom peptide discovery and characterisation. *Toxicon*, **107**: 282–289.
- Prashanth, J. R., Lewis, R. J., y Dutertre, S. (2012). Towards an integrated venomomics approach for accelerated conopeptide discovery. *Toxicon*, **60**(4): 470–477.
- Prata, S. (2011). *C++ primer plus*. Addison-Wesley Professional.
- Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc.

- Reinert, K., Dadi, T. H., Ehrhardt, M., Hauswedell, H., Mehringer, S., Rahn, R., Kim, J., Pockrandt, C., Winkler, J., Siragusa, E., *et al.* (2017). The seqan c++ template library for efficient sequence analysis: a resource for programmers. *Journal of biotechnology*, **261**: 157–168.
- Rivas-Santiago, B., Sada, E., Hernández-Pando, R., y Tsutsumi, V. (2006). Péptidos antimicrobianos en la inmunidad innata de enfermedades infecciosas. *salud pública de méxico*, **48**(1): 62–71.
- Rizk, G., Lavenier, D., y Chikhi, R. (2013). Dsk: k-mer counting with very low memory usage. *Bioinformatics*, p. btt020.
- Robertson, L. S. y Cornman, R. S. (2014). Transcriptome resources for the frogs *Lithobates clamitans* and *Pseudacris regilla*, emphasizing antimicrobial peptides and conserved loci for phylogenetics. *Molecular ecology resources*, **14**(1): 178–183.
- Schulz, M. H., Zerbino, D. R., Vingron, M., y Birney, E. (2012). Oases: robust de novo rna-seq assembly across the dynamic range of expression levels. *Bioinformatics*, **28**(8): 1086–1092.
- Stadtländer, C. C. T.-H. (2017). Next-generation sequencing data analysis xinkun wan.
- Sung, W. (2017). *Algorithms for Next-Generation Sequencing*. Chapman & Hall/CRC Mathematical and Computational Biology. CRC Press.
- Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's journal*, **30**(3): 202–210.
- Sweet, R. M. y Eisenberg, D. (1983). Correlation of sequence hydrophobicities measures similarity in three-dimensional protein structure. *Journal of molecular biology*, **171**(4): 479–488.
- Tegos, G., Tegos, G., y Mylonakis, E. (2012). *Antimicrobial drug discovery: emerging strategies*. Cabi.
- Tomii, K. y Kanehisa, M. (1996). Analysis of amino acid indices and mutation matrices for sequence comparison and structure prediction of proteins. *Protein Engineering, Design and Selection*, **9**(1): 27–36.
- Torrent, M., Di Tommaso, P., Pulido, D., Nogués, M. V., Notredame, C., Boix, E., y Andreu, D. (2012). Ampa: an automated web server for prediction of protein antimicrobial regions. *Bioinformatics*, **28**(1): 130–131.
- Tossi, A., Sandri, L., y Giangaspero, A. (2002). New consensus hydrophobicity scale extended to non-proteinogenic amino acids. *Peptides*, **27**: 416.
- Trapnell, C., Roberts, A., Goff, L., Pertea, G., Kim, D., Kelley, D. R., Pimentel, H., Salzberg, S. L., Rinn, J. L., y Pachter, L. (2012). Differential gene and transcript expression analysis of rna-seq experiments with tophat and cufflinks. *Nature protocols*, **7**(3): 562–578.
- Unterthiner, T., Mayr, A., Klambauer, G., y Hochreiter, S. (2015). Toxicity prediction using deep learning. *arXiv preprint arXiv:1503.01445*.
- Van Verk, M. C., Hickman, R., Pieterse, C. M., y Van Wees, S. C. (2013). Rna-seq: revelation of the messengers. *Trends in plant science*, **18**(4): 175–179.

- Veltri, D. y Shehu, A. (2013). Physicochemical determinants of antimicrobial activity. En: *Intl. Conf. on Bioinf. and Comp. Biol.(BICoB)*. pp. 1–6.
- Verdugo, J. A. B. (2014). *Métodos para la selección de características y clasificación de péptidos antimicrobianos*. Tesis de maestría, CICESE.
- Waghu, F. H., Gopi, L., Barai, R. S., Ramteke, P., Nizami, B., y Idicula-Thomas, S. (2014). Camp: Collection of sequences and structures of antimicrobial peptides. *Nucleic acids research*, **42**(D1): D1154–D1158.
- Wallach, I., Dzamba, M., y Heifets, A. (2015). Atomnet: a deep convolutional neural network for bioactivity prediction in structure-based drug discovery. *arXiv preprint arXiv:1510.02855*.
- Wang, G. (2010). *Antimicrobial Peptides: Discovery, Design and Novel Therapeutic Strategies*, Vol. 18. CABI.
- Wang, Z., Gerstein, M., y Snyder, M. (2009). Rna-seq: a revolutionary tool for transcriptomics. *Nature reviews genetics*, **10**(1): 57–63.
- Wei, D., Tian, C.-B., Liu, S.-H., Wang, T., Smagghe, G., Jia, F.-X., Dou, W., y Wang, J.-J. (2016). Transcriptome analysis to identify genes for peptides and proteins involved in immunity and reproduction from male accessory glands and ejaculatory duct of *bactrocera dorsalis*. *Peptides*, **80**: 48–60.
- Williams, A. (2012). *C++ concurrency in action: practical multithreading*. Manning Publ.
- Wolfenden, R., Cullis, P., y Southgate, C. (1979). Water, protein folding, and the genetic code. *Science*, **206**(4418): 575–577.
- Yoo, W. G., Lee, J. H., Shin, Y., Shim, J.-Y., Jung, M., Kang, B.-C., Oh, J., Seong, J., Lee, H. K., Kong, H. S., et al. (2014). Antimicrobial peptides in the centipede scolopendra subspinipes mutilans. *Functional & integrative genomics*, **14**(2): 275–283.
- Zimmerman, J., Eliezer, N., y Simha, R. (1968). The characterization of amino acid sequences in proteins by statistical methods. *Journal of theoretical biology*, **21**(2): 170–201.

Anexo 1 - Tabla ASCII y codificaciones formato FASTQ

En este apéndice se muestra los caracteres soportados por el estándar ASCII así como su valor decimal y hexadecimal. Posteriormente, se muestran las diferentes codificaciones para la calidad de las lecturas para las diferentes plataformas de secuenciación.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figura 32. Tabla del estándar ASCII. Figura tomada de <https://wikimedia.org>

Anexo 2 - Distribución de la base de datos CAMP y total de *k*-meros del proteoma

En este apéndice se muestra la distribución de:

- **Base de datos CAMP:** Obtenida el 23 de enero del 2017 empleando los filtros descritos en (Verdugo, 2014), la cual se encuentra conformada principalmente por péptidos entre 10 y 60 aminoácidos de longitud.
- ***k*-meros totales y únicos del proteoma del camarón:** *k*-meros totales y únicos de tamaño de 10 hasta 60 aminoácidos de las 16,037 secuencias únicas del proteoma del camarón.

La base de datos CAMP obtenida el 23 de enero del 2017 empleando los filtros descritos en (Verdugo, 2014), la cual se encuentra conformada principalmente por péptidos entre 10 y 60 aminoácidos, y el total de *k*-meros (entre 10 y 60 aminoácidos) únicos y duplicados del proteoma.

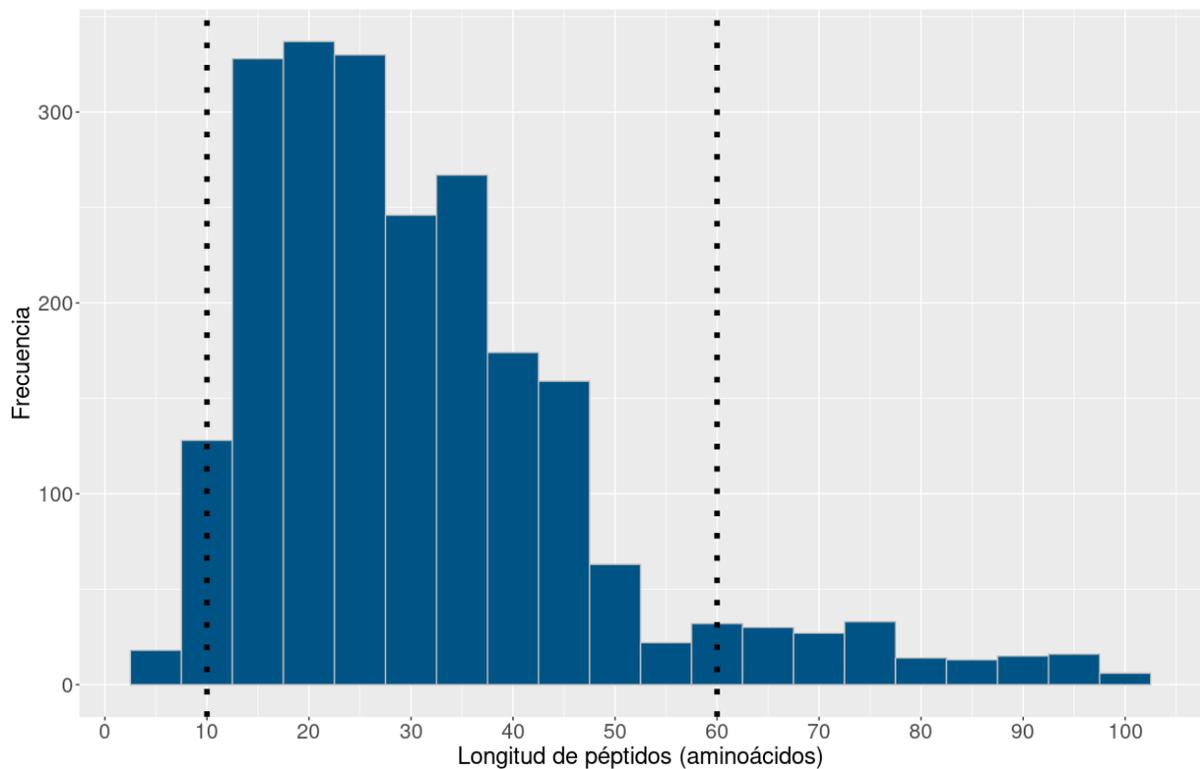


Figura 34. Histograma de la base de datos de CAMP extraída el 23 de enero del 2017 empleando los filtros descritos en (Verdugo, 2014). Conformada por un total de 2,258 péptidos antimicrobianos entre 55 y 99 aminoácidos. Las líneas punteadas verticales representan los valores de 10 y 60, intervalo en donde se encuentran concentrados la mayoría (90.43%) de los péptidos que conforman la base de datos.

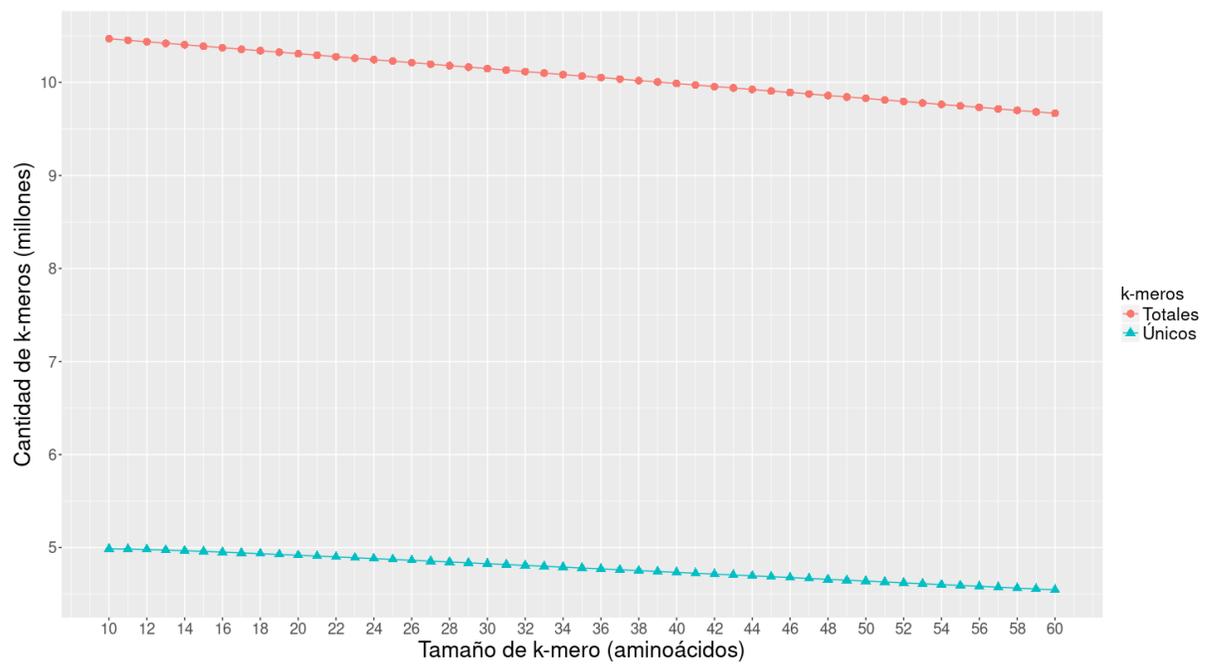


Figura 35. *k*-meros (totales y únicos) entre 10 y 60 aminoácidos del proteoma del camarón. Cifras representadas en millones.

Anexo 3 - InProt - *In silico* Proteolysis

En este apéndice se muestran los requerimientos y pasos necesarios para la instalación de *InProt (In silico Proteolysis)*, sus opciones de uso, así como ejemplos de ejecución del mismo.

.1. Descarga del código fuente

La descarga del código fuente, para su posterior compilación e instalación, se puede realizar de dos maneras:

- **Desde navegador web:** Para ello, deberá ingresar a la sección “Downloads” de la página del repositorio en Bitbucket¹, y hacer clic en la liga “Download repository”, dando inicio la descarga del código fuente en un archivo “.zip”.
- **Desde terminal:** Requerirá de un sistema con terminal y el sistema de control de versiones git², posteriormente, ejecutar el siguiente comando:

```
git clone git@bitbucket.org:germelcar/inprot.git
```

Con lo cual iniciará la descarga del código fuente, creándole un directorio con el nombre “inprot”, donde se encontrará el código fuente.

.2. Requerimientos

Para instalar *InProt* es necesario cumplir con los siguientes requerimientos mínimos:

- **Compilador compatible con C++14:** Compilador con soporte completo a C++ 14, los cuales pueden ser:
 - GCC 5.0³ o superior
 - Intel C++ Compiler 17⁴ o superior
 - Clang 3.4⁵ o superior
 - Microsoft Visual C++ 2017⁶ o superior
- **Cmake:** CMake⁷ versión 3.6 o superior, el generador de archivos GNU Make, así como de GNU Make (Graham-Cumming, 2015).
- **Intel TBB:** Librería Intel TBB⁸ versión 4.4 o superior.
- **Sistema operativo:** GNU/Linux, UNIX (como Mac, BSD) ó Windows.

¹<https://bitbucket.org/germelcar/inprot/downloads/>

²<https://git-scm.com/>

³<https://gcc.gnu.org/>

⁴<https://software.intel.com/en-us/c-compilers>

⁵<http://clang.llvm.org/>

⁶<https://www.visualstudio.com/vs/whatsnew/>

⁷<https://cmake.org/>

⁸<https://software.intel.com/en-us/intel-tbb>

.3. Selección de implementación de la máquina de aprendizaje

InProt utiliza la implementación propia de la máquina de aprendizaje por omisión; en caso de desear la implementación de LibSVM, deberá cambiar la palabra “FALSE” de la línea 44 del archivo “CMakeLists.txt” por “TRUE”, quedando de la siguiente manera:

```
set(USE_LIBSVM TRUE)
```

Una vez realizado el cambio, al compilar InProt, se compilará con la implementación de LibSVM en lugar de la implementación propia.

.4. Uso de punto flotante sencillo o punto flotante doble

InProt utiliza por omisión punto flotante doble (double) para los cálculos de descriptores moleculares y evaluación de la máquina de aprendizaje, que en comparación con cálculo de punto flotante sencillo, hace que el cálculo sea más preciso pero a su vez más lento. InProt fue implementado utilizando plantillas (Gregoire, 2014; Prata, 2011), de forma que cambiando una sola línea de código, antes de ser compilado, se pueda utilizar punto flotante sencillo (float) o doble para las operaciones involucradas en el cálculo de los descriptores moleculares y la máquina de aprendizaje. Cabe aclarar que en caso de utilizar la implementación de la máquina de aprendizaje de libsvm, el tipo de dato de punto flotante doble siempre será utilizado en la evaluación de la máquina de aprendizaje, esto es debido a la forma en que está implementada la máquina en libsvm.

Para especificarle a InProt que utilice punto flotante sencillo, bastará con cambiar la línea 15 del archivo “main.cpp”, cambiando la palabra “double” por “float”, quedando de la siguiente manera:

```
using MD_T = float;
```

una vez realizado el cambio, al compilar InProt, se utilizará punto flotante sencillo en lugar de punto flotante doble al momento de realizar las operaciones para el cálculo de los descriptores moleculares y evaluación de la máquina de aprendizaje, a excepción de que siempre se utilizará punto flotante doble en caso de haber seleccionado la implementación de la máquina de aprendizaje de libsvm.

.5. Instalación

Para su instalación, será necesario contar con alguna terminal y situarse en el directorio del código fuente, una vez dentro del código fuente, se requiere ejecutar los siguientes comandos:

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

Una vez concluidos los comandos anteriores de forma satisfactoria, se generará un binario con el nombre de *inprot*. En caso de desear que el binario se instale junto con

los demás binarios de su sistema, deberá cambiar el comando “make” por “make PREFIX=/usr/local” donde “/usr/local” es el directorio raíz donde se encuentran los binarios del sistema. Posteriormente, deberá finalizar con el comando “make install”, pero para ello, deberá contar con permisos de escritura en el directorio “/usr/local” o el directorio que se haya especificado.

.6. Opciones de comportamiento

“InProt” cuenta con varias opciones de comportamiento, las cuales, dependiendo de su valor o si se encuentran activadas, la aplicación realizará ciertas acciones diferentes a las programadas por omisión. La opción “-h” es un ejemplo de las opciones disponibles en “InProt”, con la cual se muestra un menú de las opciones disponibles y finaliza la ejecución. Por ejemplo:

```
./inprot -h
```

el cual desplegará un menú de opciones similar al siguiente:

```
InProt - In silico Proteolysis
Usage: ./inprot [OPTIONS]

Options:
-h,--help          Print this help message and exit
-i,--input TEXT    Proteome's file where to extract the k-mers
-o,--output TEXT   Output's basename where to put the
                   AMP-predicted k-mers
-s,--scaling TEXT  SVM's scaling factors configuration file
-m,--model TEXT    SVM's model file
-l,--lower UINT    Lower k-mer size (minimum = 10 )
-u,--upper UINT    Upper k-mer size (maximum = 200 )
-t,--threads INT   Number of threads (0 = for automatic
                   determination (default). Maximum = XXX).
-w,--write UINT    Write predicted k-mers (0 = none, 1 = amps, 2
                   = namps, 3 = both; default = none)
-a,--aware         Enable aware mode (low-memory consumption;
                   default false)
-v,--verbose       Enable verbose mode (show extra information)
```

a continuación se detalla cada una de las opciones disponibles en “InProt”:

- **-h**: Muestra el menú de opciones mostrado anteriormente.
- **-i** ó **--input**: Archivo multifasta (proteoma) a procesar.
- **-o** ó **--output**: Archivo multifasta generado (proteoma podado).
- **-s** ó **--scaling**: Archivo de configuración de la máquina de aprendizaje para el escalado de los datos.
- **-m** ó **--model**: Modelo de la máquina de aprendizaje.
- **-l** ó **--lower**: Tamaño mínimo de k -meros a extraer. Mínimo de 10.
- **-u** ó **--upper**: Tamaño máximo de k -meros a extraer. Máximo de 200.
- **-t** ó **--threads**: Número de hilos de CPU a utilizar. 0 para determinar el valor por omisión de acuerdo a la carga de trabajo (valor por omisión). Mínimo igual a 1, máximo igual a “XXX”, el número de hilos de “hardware” (Williams, 2012) del equipo en donde se compiló. El número “XXX” varía de procesador en procesador.
- **-w** ó **--write**: Escribe los k -meros predichos por la máquina de aprendizaje. 0 para no escribir (valor por omisión), 1 para escribir los k -meros predichos como AMP, 2 para escribir los k -meros predichos como No-AMP y 3 para escribir los k -meros predichos como AMP y No-AMP.
- **-a** ó **--aware**: Modo consciente del uso de memoria RAM. Realiza los pasos descritos en la Sección 3.4.4 haciendo uso del disco duro para reducir el consumo de memoria durante la ejecución. *Si se encuentra activada, el tiempo de cómputo será mayor.*
- **-v** ó **--verbose**: Mostrar información detallada sobre el estado de la ejecución. Deshabilitada por omisión.

.6.1. Escritura de k -meros predichos (opción “-w” ó “--write”)

Esta opción es con la finalidad de tener disponibles los k -meros predichos por la máquina de aprendizaje, sean estos predichos como AMP, No-AMP o ambos. De ese modo, los k -meros se pueden procesar con otras herramientas o con otros algoritmos como se hace con CAMPRED (*ver Sección 3.4.6*). Por omisión, no se escribe ningún k -mero, que es el equivalente al valor “0” de dicha opción. El valor “1” es para la escritura de k -meros predichos como AMP, el valor “2” para los k -meros predichos como No-AMP, y el valor “3” para los predichos como AMP y No-AMP, por separado. Los k -meros predichos se escribirán en archivos por separado, por tamaño y por clase (AMP y No-AMP). Por ejemplo, si se especifica la opción “-w” con el valor 3, se escribirían los k -meros predichos como AMP y No-AMP por la máquina de aprendizaje con los siguientes nombres:

- **k -meros predichos como AMP: podado_k-mers_amps.fasta**
- **k -meros predichos como No-AMP: podado_k-mers_namps.fasta**

Donde **podado** sería el nombre del archivo especificado como salida (opción “-o”) para el proteoma podado sin la extensión del archivo (en caso de que se haya especificado alguna), mientras que **k** sería el tamaño de los *k*-meros escritos. Finalmente, se agrega un guión bajo seguido de la palabra *amps* para los *k*-meros predichos como AMP y *namps* para los *k*-meros predichos como No-AMP.

.6.2. Modo “consciente” del consumo de memoria RAM (opción “-a” ó “--aware”)

En caso de encontrarse activada, “InProt” realizará los pasos descritos en la Sección 3.4.4 auxiliándose del disco duro para reducir el uso de memoria RAM. La desventaja principal de esta opción, es el aumento del uso del disco duro que se realiza en comparación del modo normal, el cual, mantiene toda la información procesada en memoria RAM durante la ejecución, recurriendo únicamente al disco duro para la lectura del proteoma original y para la escritura del proteoma podado. Por omisión, el modo normal se encuentra activado.

.6.2.1. Modo informativo (opción “-v” ó “--verbose”)

Por omisión, “InProt” desplegará la configuración realizada para su ejecución y el estado en donde se encuentre, de los dos disponibles:

- **Extracción de *k*-meros:** Involucra los pasos del 1 al 6 de la Figura 7, incluyendo la escritura de *k*-meros predichos si es que se especifica.
- **Podado del proteoma:** Involucra los pasos 7 y 8 de la Figura 7.

En caso de activarse el modo informativo (verbose), “InProt” desplegará información detallada sobre el estado en que se encuentre (avance).

.6.3. Ejemplos de uso

En esta sección se muestran ejemplos de uso de “InProt” con una explicación breve de cada uno. Independiente de cada ejemplo, “InProt” requiere de las siguientes opciones:

- Proteoma a analizar (opción “-i” o “--input”)
- Archivo de salida del proteoma podado (opción “-o” o “--output”)
- Archivo de configuración de la máquina de aprendizaje para el escalado de los datos (opción “-s” o “--scaling”)
- Modelo de la máquina de aprendizaje (opción “-m” o “--model”)
- Tamaño mínimo de *k*-meros a extraer (opción “-l” o “--lower”)
- Tamaño máximo de *k*-meros a extraer (opción “-u” o “--upper”)

.6.3.1. *k*-meros desde 10 hasta 60

Se extraen *k*-meros desde 10 hasta 60 aminoácidos de longitud (opciones “-l 10” y “-u 60”) del proteoma (opción “-i proteoma.fasta”), escribiendo el proteoma podado en el archivo “podado.fasta” (opción “-o podado.fasta”), el número de hilos no se especifica, por lo que “InProt” determinará el número a utilizar de manera automática, se utiliza el modelo de la máquina de aprendizaje del archivo “modelo” (opción “-m modelo”), y la configuración de la máquina de aprendizaje para el escalado de datos del archivo “escalado” (opción “-s escalado”). El comando a ejecutar es el siguiente:

```
./inprot -i proteoma.fasta -o podado.fasta -m modelo -s escalado \  
-l 10 -u 60
```

.6.3.2. *k*-meros desde 10 a 20, modo informativo y 2 hilos

Se extraen los *k*-meros desde 10 hasta 20 aminoácidos de longitud (opciones “-l 10” y “-u 20”) del proteoma (opción “-i proteoma.fasta”), se escribe el proteoma podado en el archivo “podado.fasta” (opción “-o podado.fasta”), se utilizan 2 hilos de CPU, se utiliza el modelo de la máquina de aprendizaje del archivo “modelo” (opción “-m modelo”), la configuración de la máquina de aprendizaje para el escalado de datos del archivo “escalado” (opción “-s escalado”), y se habilita el modo informativo (opción “-v”). El comando a ejecutar es el siguiente:

```
./inprot -i proteoma.fasta -o podado.fasta -m modelo -s escalado \  
-l 10 -u 20 -t 2 -v
```

.6.3.3. *k*-meros desde 10 hasta 60, escritura de *k*-meros predichos (AMP y no-AMP), modo informativo, y modo consciente del uso de memoria RAM

Se extraen los *k*-meros desde 10 hasta 60 aminoácidos de longitud (opciones “--lower 10” y “--upper 60”) del proteoma (opción “--input proteoma.fasta”), se escribe el proteoma podado en el archivo “podado.fasta” (opción “--output podado.fasta”), se utiliza el modelo de la máquina de aprendizaje del archivo “modelo” (opción “--model modelo”), la configuración de la máquina de aprendizaje para el escalado de datos del archivo “escalado” (opción “--scaling escalado”), se especifica de forma explícita que se determine de manera automática el número de hilos de CPU a utilizar (opción “--threads 0”), se habilita el modo informativo (opción “--verbose”), y se especifica que se escriban los *k*-meros predichos como AMP y no-AMP (opción “--write 3”). El comando a ejecutar es el siguiente:

```
./inprot --input proteoma.fasta --output podado.fasta --model modelo \  
--scaling escalado --lower 10 --upper 60 \  
--threads 0 --verbose --write 3
```

Anexo 4 - CAMPRED - CAMP AMP PREDiction

En este apéndice se muestran los requerimientos y pasos necesarios para la instalación de *CAMPRED* (**CAMP AMP PRED**iction), sus opciones de uso, así como ejemplos de ejecución del mismo.

.1. Descarga del código fuente

La descarga del código fuente se puede realizar de dos maneras:

- **Desde navegador web:** Para ello, deberá ingresar a la sección “Downloads” de la página del repositorio en Bitbucket¹, y hacer clic en la liga “Download repository”, dando inicio la descarga del código fuente en un archivo “.zip”.
- **Desde terminal:** Requerirá de un sistema con terminal y el sistema de control de versiones git², posteriormente, ejecutar el siguiente comando:

```
git clone git@bitbucket.org:germelcar/campred.git
```

Con lo cual iniciará la descarga del código fuente, creándole un directorio con el nombre “campred”, donde se encontrará el código fuente.

Cabe mencionar que en caso de instalar “CAMPRED” desde el código fuente a través del binario “go”, el código fuente también será descargado automáticamente, situándose en el directorio “\$GOPATH/src”.

.2. Requerimientos e instalación

Debido a que CAMPRED está desarrollada en lenguaje de programación Go (Doxsey, 2016; Kennedy *et al.*, 2015), no se requiere la instalación de librerías externas; las librerías requeridas son descargadas e instaladas por Go al momento de compilar. La instalación se puede hacer de tres maneras:

- **Compilando desde el código fuente (Internet):** Esta opción requiere que se tenga instalado Go³ y acceso a Internet en el sistema donde se vaya a realizar la instalación. El comando a ejecutar para su instalación es el siguiente:

```
go get -u bitbucket.org/germelcar/campred
```

donde el binario “go” realizará la descarga y compilación del código fuente, dejando un binario con nombre *campred* en el directorio “\$GOPATH/bin”, donde “\$GOPATH” es el directorio especificado durante la instalación de Go, el cual, por convención es “\$HOME/go”, donde “\$HOME” es el directorio raíz de su sistema: /home/USUARIO para sistemas GNU/Linux y /home/Users/USUARIO para sistemas Mac, donde *USUARIO* es el nombre de usuario del sistema.

¹<https://bitbucket.org/germelcar/campred>

²<https://git-scm.com/>

³<https://golang.org/>

- **Compilando desde el código fuente (Local):** Esta opción también requiere que se tenga instalado Go, y el código fuente descargado en el sistema. Una vez descargado el código fuente en el sistema, se requiere posicionarse (con la terminal) en el directorio raíz del código fuente. Posteriormente, se requiere ejecutar el siguiente comando:

```
go build
```

con lo cual se creará un binario con nombre *campred*.

- **Transfiriendo el binario:** En esta opción se requiere únicamente contar con el binario compilado para la misma arquitectura y sistema operativo del sistema donde se desea instalar. Por lo que restaría transferir el binario al sistema deseado.

.3. Opciones de comportamiento

“CAMPRED” cuenta con varias opciones de comportamiento, las cuales, dependiendo de su valor o si se encuentran activadas, la aplicación realizará ciertas acciones diferentes a las programadas por omisión. La opción “-h” es un ejemplo de las opciones disponibles en “CAMPRED”, con la cual se muestra un menú de las opciones disponibles y finaliza la ejecución. Por ejemplo:

```
campred -h
```

el cual desplegará un menú de opciones similar al siguiente:

Usage: campred FLAGS ARGUMENTS

CAMPRED - CAMP AMP PREDiction v0.1

Flags:

-i, --input string	Input filename
-k, --keep	Keep intermediate file (default true)
-n, --nseqs n (default 1)	Split in multiple parts of n parts each one
-o, --output string	Output filename
-s, --send int 10) (default 10)	Max number of times to send each request (max. 10)
-t, --threads int	Number of threads (default "XXX")
-v, --verbose	Show extra information

Arguments:

svm	Support Vector Machine
ann	Artificial Neural Network
rf	Random Forest
da	Discriminant Analysis
all	All the algorithm above

a continuación se detalla cada una de las opciones disponibles en "CAMPRED":

- **-h**: Muestra el menú de opciones mostrado anteriormente.
- **-i ó --input**: Archivo multifasta a mandar al servidor de CAMP.
- **-k ó --keep**: Guardar en disco duro la respuesta del servidor (archivo html). Deshabilitada por defecto.
- **-n ó --nseqs**: Número de secuencias de los conjuntos creados al particionar el archivo de entrada (1 por defecto).
- **-o ó --output**: Archivo multifasta de salida.
- **-s ó --send**: Máximo número de reintentos por cada conjunto de secuencias (por omisión 10, máximo 10).

- **-t ó --threads:** Número de hilos de CPU a utilizar. “XXX” por omisión, el número de hilos de “hardware” (Williams, 2012) del equipo. El número “XXX” varía de procesador en procesador.
- **-v ó --verbose:** Mostrar información sobre el estado de la ejecución. Deshabilitada por omisión.
- **Argumentos:** Algoritmos a utilizar para la predicción de la actividad antimicrobiana de las secuencias.

.3.1. Guardar en disco duro la respuesta del servidor (opción “-k” ó “--keep”)

Al ser activada, la respuesta recibida de la solicitud realizada al servidor de CAMP (HTTP response) será guardada de la forma “XYZ.fasta.camp”, donde “XYZ.fasta” es el conjunto de secuencias en formato multifasta que fue procesado por CAMP. Aunque la respuesta se encuentra en formato HTML, esta se guarda con extensión “.camp”.

.3.2. Número de secuencias de cada conjunto (opción “-n” ó “--nseqs”)

Especifica el total de secuencias que contendrá cada subconjunto creado del archivo de entrada (conjunto original) especificado por el usuario, esto es, por ejemplo, si el archivo de entrada especificado por el usuario contiene 100 secuencias, y el valor de la opción “n” es igual a 10, se crearán 10 subconjuntos con 10 secuencias cada uno. Por omisión tiene el valor de 1, el cual omite el particionado del archivo de entrada especificado por el usuario.

.3.3. Número de reintentos (opción “-s” ó “--send”)

Especifica el total de reintentos por cada subconjunto de secuencias. Esto es, si hubo problemas al procesar una petición, ya sea porque el tiempo de espera (10 minutos) expiró, pérdida de conexión o algún factor externo, se vuelve a procesar la petición hasta el número máximo de veces. El valor máximo es de 10 intentos por petición, en caso de que alguna petición haya fallado, y esta no haya alcanzado el límite, se vuelve a procesar hasta alcanzar el límite máximo del sistema ó el limite especificado por el usuario.

.3.3.1. Modo informativo (opción “-v” ó “--verbose”)

Por omisión, “CAMPRED” desplegará la configuración realizada para su ejecución y el estado, de los dos disponibles, en donde se encuentre:

- **Particionado del archivo de entrada:** Desplegará información detallada de los subconjuntos creados del archivos de entrada original especificado por el usuario, mostrando el total de secuencias contenidas en cada subconjunto y el total de secuencias leídas del archivo de entrada.

- **Procesado de los conjuntos de secuencias:** Desplegará información detallada sobre el archivo procesándose y el total de secuencias predichas como AMPs por los algoritmos especificados por el usuario.

En caso de activarse el modo informativo (verbose), “CAMPRED” desplegará información detallada sobre el avance del estado en que se encuentre.

.3.4. Ejemplos de uso

En esta sección se muestran ejemplos de uso de “CAMPRED” con una explicación breve de cada uno. Independiente de cada ejemplo, “CAMPRED” requiere de las siguientes opciones y argumentos:

- Archivos de secuencias a procesar (opción “-i” o “--input”)
- Archivo de salida de los subconjuntos (opción “-o” o “--output”)
- Algoritmo(s) a utilizar (opciones “svm”, “ann”, “rf”, “da” y “all”)

.3.4.1. Particionado en subconjuntos de 10 secuencias, 2 hilos de CPU y SVM como algoritmo

Se particionan las secuencias a analizar del archivo “secuencias.fasta” (opción “-i secuencias.fasta”), en subconjuntos de 10 secuencias cada uno (opción “-n 10”), donde dichos subconjuntos tendrán el prefijo “subconjuntos” (opción “-o subconjuntos”), usando dos hilos de CPU (opción “-t 2”), y el algoritmo “svm” (argumento “svm”). El comando a ejecutar es el siguiente:

```
campred -i secuencias.fasta -o subconjunto -n 10 -t 2 svm
```

.3.4.2. Particionado en subconjuntos de 50 secuencias, todos los algoritmos y guardando la respuesta de CAMP

Se particionan las secuencias a analizar del archivo “secuencias.fasta” (opción “--input secuencias.fasta”), en subconjuntos de 50 secuencias cada uno (opción “-nseqs 50”), donde dichos subconjuntos tendrán el prefijo “subconjuntos” (opción “-output subconjuntos”), guardando la salida de la respuesta de CAMP (opción “--keep”), y todos los algoritmos de CAMP (argumento “all”). El comando a ejecutar es el siguiente:

```
campred --input secuencias.fasta --output subconjunto -n 50 --keep all
```

Anexo 5 - Uso de SLURM, Trinity y TransDecoder

En este apéndice se describen las siguientes herramientas:

- **SLURM:** Uso básico de SLURM, el manejador de procesos del cluster OMICA del CICESE. Se verán los pasos necesarios para mandar procesos a SLURM y su monitoreo.
- **Trinity:** Uso básico de Trinity, el ensamblador *de novo* de transcriptomas. Se verán los pasos necesarios para mandar a ensamblar un transcriptoma mediante SLURM en un solo nodo, así como en varios nodos.
- **TransDecoder:** Uso básico de TransDecoder, el detector y traductor de ORFs (**O**pen **R**eadin**G** **F**rames), creado por los mismos desarrolladores de Trinity. Se verá el comando básico de uso, así como pasos opcionales.

.1. SLURM - Manejador de procesos del cluster OMICA del CICESE

SLURM¹ es el manejador de procesos del cluster OMICA del CICESE el cual es tolerante a fallas, altamente escalable y de acceso libre mediante el cual los procesos de los usuarios del cluster son manejados; SLURM se encarga de asignar los recursos solicitados por los procesos de los usuarios, administrar dichos procesos y administrar los recursos del cluster.

El cluster OMICA del CICESE cuenta con 30 nodos (*ver Sección 4.3*), además del nodo maestro, el cual, es el encargado de recibir las solicitudes de los procesos de los usuarios y finalmente, asigna dichos procesos a los nodos correspondientes.

.1.1. Nodos y particiones

En SLURM, cada proceso tiene que ser asignado a una partición, la cual, es un conjunto de nodos definidos con un tiempo de cómputo límite. En caso de no especificarse, la partición “cicese” será seleccionada por omisión. Para listar los nodos disponibles y sus respectivas particiones, se debe ejecutar el comando “sinfo”, el cual, deberá mostrar una salida similar a la mostrada en la Figura 36.

Donde “TIMELIMIT” es el tiempo límite de cómputo para un proceso en la partición en la que se ejecutará dicho proceso. Por ejemplo, para la partición “cicese”, hay un tiempo límite de 6 días con cero horas, cero minutos y cero segundos (días:horas:minutos:segundos).

.1.2. Mandar procesos

Para mandar procesos, se debe hacer por medio del comando “sbatch”, el cual, recibe como argumento la ruta del script en formato de SLURM, que se encuentra conformado de directivas de SLURM para el proceso, donde se especifican los requerimientos del proceso a ejecutar, seguido del proceso a ejecutar.

¹<https://slurm.schedmd.com/>

```

gmelendrez@omica:~
File Edit View Search Terminal Help
[gmelendrez@omica ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
cicese*   up 6-00:00:00    1  down* nodo1
cicese*   up 6-00:00:00   29  idle  nodo[2-30]
d15       up 15-00:00:0    1  down* nodo1
d15       up 15-00:00:0   29  idle  nodo[2-30]
d30       up 30-00:00:0    1  down* nodo1
d30       up 30-00:00:0   29  idle  nodo[2-30]
d45       up 45-00:00:0    1  down* nodo1
d45       up 45-00:00:0   29  idle  nodo[2-30]
d60       up 60-00:00:0    1  down* nodo1
d60       up 60-00:00:0   29  idle  nodo[2-30]
mayor     up 30-00:00:0    1  down* nodo1
mayor     up 30-00:00:0   29  idle  nodo[2-30]
large     up 10-00:00:0    1  down* nodo1
large     up 10-00:00:0   29  idle  nodo[2-30]
[gmelendrez@omica ~]$

```

Figura 36. Lista de particiones y nodos asignados a las mismas. Las columnas representan el nombre de la partición, disponibilidad, tiempo límite de cómputo, nodos asignados, estado de la partición y la lista de nodos que conforman la partición.

.1.2.1. Plantilla básica para proceso de SLURM

A continuación se muestra una plantilla básica para SLURM, donde se manda a ejecutar un proceso el cual manda a imprimir la cadena de texto “Hola mundo” en la salida estándar, seguido de una espera de 10 segundos. El proceso se ejecutará en la partición “cicese”, con una duración máxima de 1 día, un máximo de 1 GB de memoria RAM, un nodo y un solo núcleo.

```

1  #!/bin/bash
2
3  #SBATCH -p cicese          # Utilizar la particion 'cicese'
4  #SBATCH -N 1              # Utilizar 1 nodo
5  #SBATCH -c 1              # Utilizar 1 nucleo
6  #SBATCH -J Plantilla      # Nombrar el proceso como 'Plantilla'
7
8  #SBATCH -o plantilla.log   # Redireccionar la salida estandar
9                             # al archivo 'plantilla.log'
10
11 #SBATCH -e plantilla.error # Redireccionar la salida de error
12                             # al archivo 'plantilla.error'
13
14 #SBATCH --time=1-00:00:00 # Tiempo maximo de 1 dia, con cero horas,

```

```

15                                     # cero minutos y cero segundos.
16                                     # dias-horas:minutos:segundos
17
18 #SBATCH --memory=1G                 # Maximo 1 gigabyte de memoria
19                                     # requerida por el proceso
20
21
22 # Comandos a ejecutar
23 echo "Hola mundo"                   # Imprime 'Hola mundo' en la salida estandar
24 sleep 10s                           # Espera 10 segundos antes de terminar el proceso

```

Debido a que la salida estándar (stdout) es redireccionada al archivo “plantilla.log” (línea 8), esta no se muestra en la terminal, del mismo modo, en caso de haber un error o mandarse a imprimir en la salida de error (stderr), este se redireccionará al archivo “plantilla.error” (línea 11). En el archivo “plantilla.log” deberá aparecer la cadena de texto “Hola mundo”.

Las líneas 3 a la 18 son directivas de SLURM que le indican a SLURM los requerimientos para dicho proceso, de modo que pueda reservar los recursos necesarios especificados al inicio del archivo y asignarlos para nuestro proceso. En caso de no especificarse, SLURM asignará el primer nodo disponible que encuentre y que cuente con los recursos solicitados para el proceso. En caso de querer solicitar algún nodo específico, este se puede solicitar mediante la directiva “#SBATCH --nodelist=nodoX” donde “nodoX” es el nodo deseado (del 1 al 30).

Para indicarle a SLURM que asigne un nodo para el proceso especificado en la plantilla, bastará con ejecutar el comando “sbatch plantilla.sh”, donde “plantilla.sh” es el archivo del script plantilla .1.2.1. Una vez asignado uno o varios nodos al proceso, SLURM mostrará el mensaje “Submitted batch job XXX”, donde “XXX” es un identificador numérico del proceso. El tiempo, máximo de memoria, número de nodos, número de núcleos y demás parámetros dependerá del proceso a ejecutar. Una vez asignado un nodo (o varios) al proceso, éste aparecerá en la lista de procesos ejecutándose, el cual se despliega al ejecutar el comando “squeue”. En la Figura 37 se muestra cómo se manda la plantilla .1.2.1 a SLURM y se mandan a desplegar los procesos ejecutándose, donde se observa el proceso de la plantilla.

```

gmelendrez@omica:/LUSTRE/bioinformatica_data/biocomp/gmelendrez
File Edit View Search Terminal Help
[gmelendrez@omica gmelendrez]$ sbatch plantilla.sh
Submitted batch job 104811
[gmelendrez@omica gmelendrez]$ squeue -l
Thu Jan 11 14:21:19 2018
      JOBID PARTITION   NAME     USER    STATE   TIME TIME_LIMI  NODES  NODELIST(REASON)
      104811   cicese Plantill gmelendr RUNNING  0:04 1-00:00:00    1 nodo2
[gmelendrez@omica gmelendrez]$

```

Figura 37. Ejecución del comando “sbatch” de SLURM para ejecutar la plantilla especificada en la Subsección .1.2.1 y desplegado de los procesos ejecutándose en el cluster, mostrando el identificador, partición, nombre, usuario, estado, tiempo de cómputo, tiempo límite de cómputo y nodo asignado del proceso. La opción “-l” del comando “squeue” es para indicarle a SLURM que muestre los nombres de los campos completos, debido a que por omisión los muestra abreviados.

.2. Trinity - Uso en el cluster OMICA del CICESE

.2.1. Instalación

Una vez descargado, la instalación requerirá del comando “make” para proceder con la compilación de los ejecutables necesarios (Inchworm, Parafly, Chrysalis, entre otros) para el funcionamiento correcto de Trinity. Una vez finalizada la compilación, desplegará un mensaje similar al siguiente:

```

~~~~~
Performing Unit Tests of Build
~~~~~

JellyFish:           has been Installed Properly
Inchworm:            has been Installed Properly
Chrysalis:           has been Installed Properly
QuantifyGraph:       has been Installed Properly
GraphFromFasta:      has been Installed Properly
ReadsToTranscripts: has been Installed Properly
parafly:             has been Installed Properly
samtools             has been Installed Properly

```

Por omisión, Trinity utiliza g++ como compilador de C++. En caso de contar con una versión más reciente de g++ de la soportada por Trinity, se puede apoyar de los

pasos descritos en la siguiente página²; por ejemplo, para la versión más reciente actualmente de Trinity, 2.5.1, la última versión soportada de g++ es la 6, mientras que la versión más actual es la 7.3 (a febrero de 2018).

.2.2. Ensamblar de forma paralela en un solo nodo

Para el ensamblaje de transcriptomas *de novo* de forma paralela en un solo nodo, será necesario contar con el total de *reads*, la orientación de los mismos, saber si son *single-end* ó *paired-end* y recursos computacionales suficientes, esto es, en promedio 1GB de memoria RAM para 1 millón de *paired-end reads*³. A continuación se muestra una plantilla básica para el ensamblaje de *paired-end reads* de forma paralela en un solo nodo en el cluster OMICA del CICESE.

```

1 #!/bin/bash
2
3 #SBATCH -p cicese      # Utilizar la particion 'cicese'
4 #SBATCH -N 1          # Utilizar 1 nodo
5 #SBATCH -c 6          # Utilizar 6 nucleos
6 #SBATCH -J Trinity    # Nombrar el proceso como 'Plantilla'
7
8 #SBATCH -o trinity.log # Redireccionar la salida estandar
9                       # al archivo 'plantilla.log'
10
11 #SBATCH -e trinity.error # Redireccionar la salida de error
12                       # al archivo 'plantilla.error'
13
14 #SBATCH --time=1-00:00:00 # Tiempo maximo de 1 dia, con cero horas,
15                           # cero minutos y cero segundos.
16                           # dias-horas:minutos:segundos
17
18 #SBATCH --memory=50G     # Maximo 50 gigabytes de memoria
19                           # requerida por el proceso
20
21
22 # Comandos a ejecutar
23 Trinity --seqType fq --max_memory 50G \
24         --left reads_1.fq.gz --right reads_2.fq.gz --CPU 6 \
25         --no_normalize_reads

```

Donde la opción “--left” representa los *left reads* y “right” representa los *right reads*, “--seqType” formato de los *reads* (fasta ó fastq) y “--CPU” el total de hilos a utilizar. En caso de no especificarse la salida del transcriptoma (opción “--output”), Trinity escribirá los archivos generados durante el ensamblado, así como el ensamblado, en el directorio “trinity_out_dir”. Si se desea algún directorio diferente, éste deberá llevar el prefijo “trinity”. Por omisión, el ensamblado se llamará “Trinity.fasta” en el directorio de salida especificado previamente. Del mismo modo, en versiones recientes, por

²<https://github.com/trinityrnaseq/trinityrnaseq/issues/174>

³<https://github.com/trinityrnaseq/trinityrnaseq/wiki/Trinity-Computing-Requirements>

omisión Trinity realizará una normalización de los *reads*, en caso de no desear dicho proceso, se puede omitir mediante la opción “--no_normalize_reads”.

.2.3. Ensamblar de forma distribuída en varios nodos

Además de contar con los datos necesarios para el ensamblaje paralelo en un solo nodo (ver *Subsección .2.2*), deberá contar con HPC GridRunner⁴ instalado, el cual, distribuirá el trabajo a los diferentes nodos disponibles para el ensamblaje; también deberá contar con el archivo de configuración de HPC GridRunner, en el cual, se especifica cuántos nodos máximos se utilizarán, entre otros parámetros. A continuación se muestra una plantilla básica para el archivo de configuración de HPC GridRunner para SLURM:

```

1 [GRID]
2 # grid type:
3 gridtype=SLURM
4
5 # template for a grid submission
6 cmd=sbatch -p cicese --mem=120G --time=6-00:00:00 -c 24 --exclusive
7
8 # note -e error.file -o out.file are set internally,
9 # so dont set them in the above cmd.
10
11 #####
12 # settings below configure the Trinity job submission system, #
13 # not tied to the grid itself.
14 #
15 #####
16 # number of grid submissions to be maintained at steady state by
17 # the Trinity submission system
18 max_nodes=4
19
20 # number of commands that are batched into a single grid submission job.
21 cmds_per_node=1
22
23
24 ##### IMPORTANT #####
25 # it is important that the node you launch your job on is capable of
26 # submitting new jobs. The parameter that determines this is the nodes
27 # specified in 'AllocNodes' ( http://slurm.schedmd.com/scontrol.html )
28 # for a given slurm partition. An easy way to have this work is to set
29 # that parameter to allow all nodes to submit new jobs:
30 #
31 'AllocNodes=ALL'
32 #
33 #####

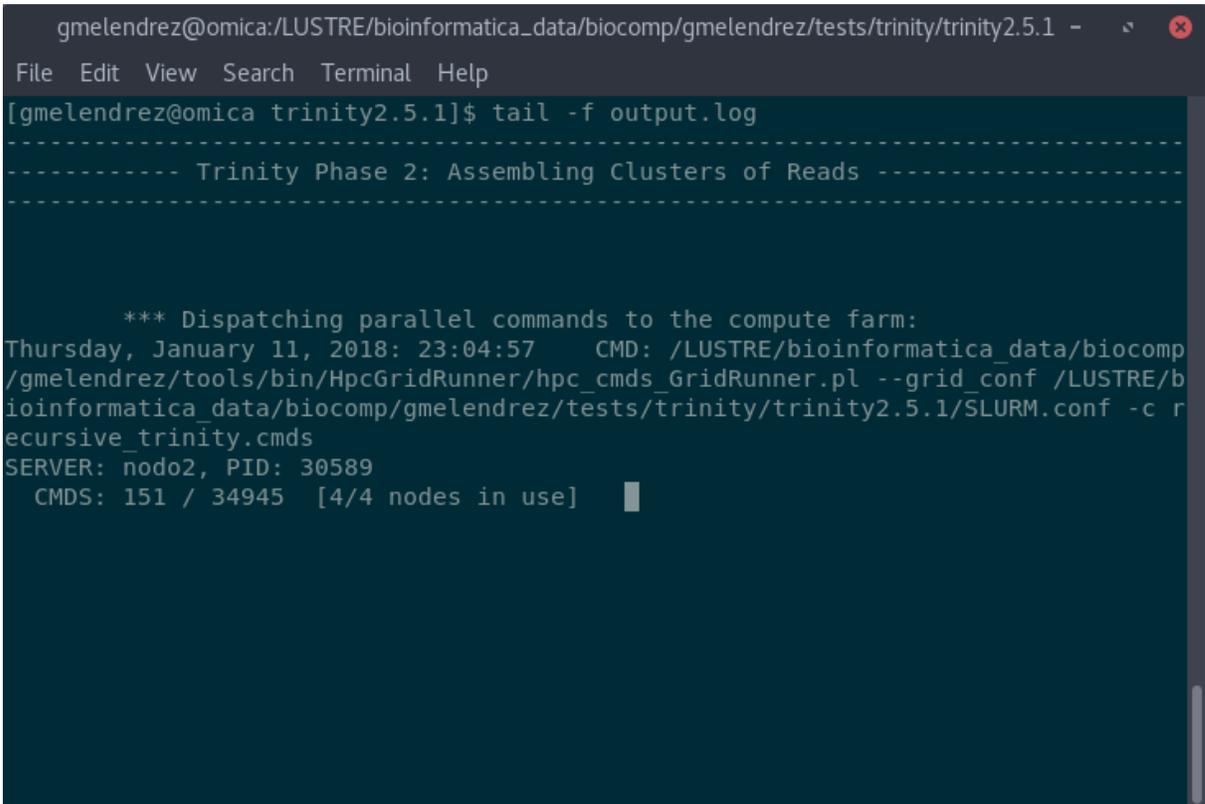
```

⁴<https://hpcgridrunner.github.io/>

De este modo, el comando de Trinity especificado en la Subsección .2.2 cambiará ligeramente por:

```
Trinity --seqType fq --max_memory 50G \
--left reads_1.fq.gz --right reads_2.fq.gz --CPU 6 \
--no_normalize_reads \
--grid_exec "hpc_cmds_GridRunner.pl --grid_conf SLURM.conf -c"
```

Donde “SLURM.conf” es el archivo de configuración de HPC GridRunner para SLURM (ver Subsección .2.3). Cabe aclarar que la única fase (de las 3) que se realiza de forma distribuída es la segunda, el ensamblado de clusters de los *reads*. Una vez llegada a dicha fase, Trinity deberá desplegar el progreso del ensamblado similar al que se muestra en la Figura 38.



```
gmelendrez@omica:/LUSTRE/bioinformatica_data/biocomp/gmelendrez/tests/trinity/trinity2.5.1 -
File Edit View Search Terminal Help
[gmelendrez@omica trinity2.5.1]$ tail -f output.log
----- Trinity Phase 2: Assembling Clusters of Reads -----
*** Dispatching parallel commands to the compute farm:
Thursday, January 11, 2018: 23:04:57    CMD: /LUSTRE/bioinformatica_data/biocomp
/gmelendrez/tools/bin/HpcGridRunner/hpc_cmds_GridRunner.pl --grid_conf /LUSTRE/b
ioinformatica_data/biocomp/gmelendrez/tests/trinity/trinity2.5.1/SLURM.conf -c r
ecursive_trinity.cmds
SERVER: nodo2, PID: 30589
  CMDS: 151 / 34945  [4/4 nodes in use]
```

Figura 38. Progreso de la segunda fase del ensamblado del Transcriptoma de (Yoo *et al.*, 2014) de forma distribuída con Trinity.

.3. TransDecoder

TransDecoder cuenta con dos modos para la detección y traducción de los ORFs: 1) basado en un archivo multifasta de transcritos, y 2) por medio de la estructura de los transcritos basados en genomas (archivo “GTF”). En este trabajo nos limitaremos al primer modo, al no contar con genoma de referencia. El primer modo cuenta con 3 fases, donde la primera fase detecta, extrae y traduce ORFs largos, de al menos 100 aminoácidos de longitud, generando varios archivos, entre ellos, los ORFs, los ORFs traducidos de acuerdo al código genético especificado (universal por omisión) y las posiciones de los ORFs encontrados en el transcriptoma en formato “.gff3”. Los

archivos serán guardados en el directorio “TRANSCRIPTOMA.transdecoder_dir”, donde “TRANSCRIPTOMA” es el nombre del archivo del transcriptoma.

.3.1. Instalación

Debido a TransDecoder se encuentra desarrollado en lenguaje de programación Perl, no se requiere de compiladores y demás pasos necesarios para su instalación. Bastará con la descarga y extracción (en caso de descargarlo comprimido) de TransDecoder.

.3.2. Fase 1 - Predicción de las secuencias codificantes

Como primera fase, está la detección de los ORFs y traducción de los mismos, detectando aquellos ORFs de al menos 100 aminoácidos de longitud. Para ello, TransDecoder requiere del transcriptoma (opción “-t”), teniendo las demás opciones con valores por omisión. En caso de cambiar la longitud de los ORFs, se deberá de especificar su longitud (en aminoácidos) con la opción “-m”, aunque la tasa de falsos positivos para la predicción de ORFs, incrementa drásticamente con un valores más pequeños. El comando a ejecutar para esta primera fase, es el siguiente:

```
TransDecoder.LongOrfs -t Trinity.fasta
```

En la Figura 39 se muestra un ejemplo de la primera fase con el transcriptoma del camarón (Ghaffari *et al.*, 2014).

```

gmelendrez@omica:/LUSTRE/bioinformatica_data/biocomp/gmelendrez/shrimp/transde
File Edit View Search Terminal Help
[gmelendrez@omica transde]$ TransDecoder.LongOrfs -t Trinity.fasta

-first extracting base frequencies, we'll need them later.
CMD: /LUSTRE/bioinformatica_data/biocomp/gmelendrez/tools/bin/TransDecoder-5.0.2/util/compute_base_probs.pl Trin
ity.fasta 0 > Trinity.fasta.transdecoder_dir/base_freqs.dat
CMD: touch Trinity.fasta.transdecoder_dir/base_freqs.dat.ok

- extracting ORFs from transcripts.
-total transcripts to examine: 110602
[110600/110602] = 100.00% done    CMD: touch Trinity.fasta.transdecoder_dir/_longorfs.ok

#####
### Done preparing long ORFs. ###
#####

    Use file: Trinity.fasta.transdecoder_dir/longest_orfs.pep for Pfam and/or BlastP searches to enable hom
ology-based coding region identification.

    Then, run TransDecoder.Predict for your final coding region predictions.

[gmelendrez@omica transde]$

```

Figura 39. Primera fase de TransDecoder utilizando el transcriptoma de (Ghaffari *et al.*, 2014).

Una vez finalizada la primera fase, se crearán varios archivos, entre ellos se encuentra el archivo con nombre “longest_orfs.pep”, en donde se escribirán los ORFs traducidos de acuerdo al código genético especificado.

.3.3. Fase 2 - Identificación de ORFs homólogos a proteínas conocidas

Aunque esta segunda fase es opcional, puede mejorar la sensibilidad de captura de ORFs homólogos a proteínas conocidas, y que puedan tener significancia funcional. Eso se hace por medio de búsquedas con Blastp (BLAST+) o HMMER (Eddy, 2010).

.3.3.1. Búsqueda con Blastp

La búsqueda se puede hacer de dos modos, de manera similar a como se puede hacer el ensamblado de transcriptomas con Trinity, es decir, de forma paralela en un solo nodo o de forma distribuida en varios nodos, requiriendo de HPC GridRunner para esta última. Dicha búsqueda se puede hacer con la base de datos SwissProt⁵ o Uniref90⁶, siendo más rápida la búsqueda con la primera base de datos.

Búsqueda con Blastp de forma paralela en un solo nodo

La búsqueda con Blastp, de forma paralela en un solo nodo, se realiza con el siguiente comando:

```
blastp -query longest_orfs.pep \
  -db uniprot_sprot.fasta -max_target_seqs 1 \
  -outfmt 6 -evalue 1e-5 -num_threads 24 > blastp.outfmt6
```

Donde “longest_orfs.pep” son los péptidos generados en la primera fase de TransDecoder (ver Subsección .3.2). El resultado será escrito de forma tabular (opción “-outfmt 6”), utilizando 24 hilos (opción “-num_threads 24”) y los resultados en la búsqueda serán guardados en el archivo “blastp.outfmt6”.

Búsqueda con Blastp de forma distribuida en varios nodos

Para ello deberá contar con HPC GridRunner descargado. Reutilizando la plantilla básica para mandar procesos a SLURM (ver Subsección .1.2.1), cambiando las líneas 23 y 24 de la plantilla por:

```
perl $HPC_GRIDRUNNER/BioIfx/hpc_FASTA_GridRunner.pl \
  --cmd_template "blastp -query __QUERY_FILE__ -db $DB
  -max_target_seqs 1 -outfmt 6 -evalue 1e-5 -num_threads 24" \
  --query_fasta longest_orfs.pep \
  --grid_conf SLURM.conf \
  -N 1000 -O blastp_search
```

Donde “\$HPC_GRIDRUNNER” es el directorio donde se encuentre HPC GridRunner, “\$DB” es la ruta completa de la base de datos, “longest_orfs.pep” es el archivo generado en la primera fase, “SLURM.conf” es el archivo de configuración de HPC GridRunner para SLURM (ver Subsección .2.3). Con ello, el archivo “longest_orfs.pep” será

⁵<http://www.uniprot.org/>

⁶<http://www.uniprot.org/help/uniref>

particionado en subconjuntos de 1,000 secuencias cada uno (opción “-N 1000”), donde cada subconjunto será procesado de forma paralela en un nodo del cluster, dejando los subconjuntos y sus resultados de búsqueda en el directorio “blastp_search” (opción “-O blastp_search”). Finalmente, para concatenar los resultados de las búsquedas de cada subconjunto, en un solo archivo, se deberá ejecutar el siguiente comando:

```
find blastp_search -name "*.fa.OUT" -exec cat {} \; > all.blast.out
```

Dejando en “all.blast.out” la concatenación de los resultados de todas las búsquedas de los subconjuntos.

.3.3.2. Búsqueda con HMMER

Búsqueda con HMMER de forma paralela en un solo nodo

De forma similar a la búsqueda con Blastp, la búsqueda con HMMER de forma paralela en un solo nodo, se realiza con el siguiente comando:

```
hmmScan --cpu 24 --domtblout pfam.domtblout Pfam-A.hmm longest_orfs.pep
```

Siendo “Pfam-A.hmm” la base de datos Pfam⁷, “longest_orfs.pep” es el archivo generado en la primera fase, utilizando 24 hilos (opción “--cpu 24”), y escribiendo los resultados de la búsqueda en formato tabular en el archivo “pfam.domtblout” (opción “--domtblout”).

Búsqueda con HMMER de forma distribuída en varios nodos

De forma similar a la búsqueda con Blastp de forma distribuída (ver *Subsección .3.3.1*), se requiere de HPC GridRunner. Reutilizando la plantilla básica para mandar procesos a SLURM (ver *Subsección .1.2.1*), el comando a ejecutar es el siguiente:

```
perl $HPC_GRIDRUNNER/BioIfx/hpc_FASTA_GridRunner.pl \
--cmd_template "hmmScan --cpu 24 --domtblout __QUERY_FILE__.domtblout
$DB __QUERY_FILE__" \
--query_fasta longest_orfs.pep \
--grid_conf SLURM.conf \
-N 1000 -O pfam_search
```

De manera similar a la búsqueda distribuída con Blastp (ver *Subsección .3.3.1*), el archivo generado en la primera fase se particiona en subconjunto de 1,000 secuencias, donde cada subconjunto es procesado de forma paralela en un nodo independiente. Finalmente, para concatenar los resultados de las búsquedas de cada subconjunto, en un solo archivo, se deberá ejecutar el siguiente comando:

```
find pfam_search/ -name "*.fa.domtblout" -exec cat {} \; > all.pfam.out
```

Dejando en “all.pfam.out” la concatenación de los resultados de todas las búsquedas de los subconjuntos.

⁷<http://pfam.xfam.org/>

.3.3.3. Fase 3 - Predicción de las regiones codificantes probables

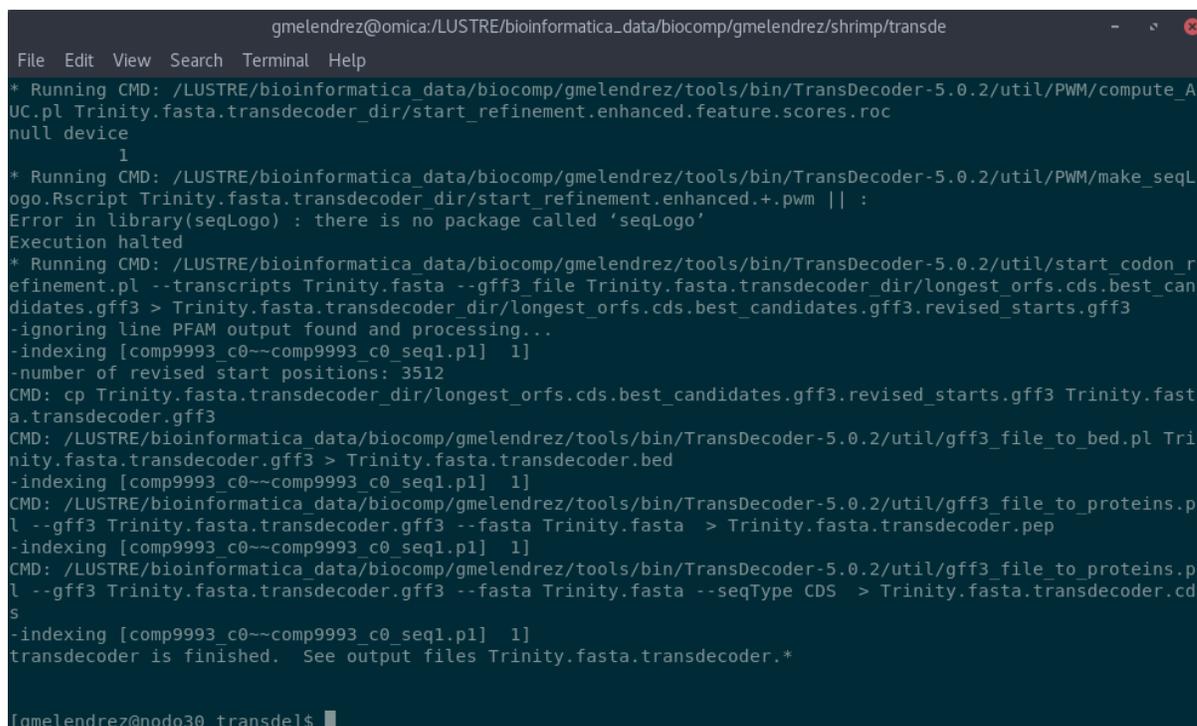
Como tercera fase, se realiza la predicción de regiones codificantes probables. Esta fase se puede realizar de forma independiente a la segunda fase (*ver Subsección .3.3*) o se puede realizar utilizando los resultados de la segunda fase. Para la primera opción, el comando a ejecutar es:

```
TransDecoder.Predict -t Trinity.fasta
```

Mientras que para la segunda opción (utilizando los resultados de la segunda fase), el comando a ejecutar es el siguiente:

```
TransDecoder.Predict -t Trinity.fasta --retain_pfam_hits pfam.domtblout \
--retain_blastp_hits blastp.outfmt6
```

Donde “pfam.domtblout” es el archivo de la búsqueda con HMMER, mientras que “blastp.outfmt6” es el archivo de la búsqueda con BLASTp. Las predicciones finales de las secuencias codificantes (CDS, **CoDing Sequence**) serán guardadas en el mismo directorio donde se guardaron los archivos de la primera fase, las cuales incluirán las 500 regiones codificantes más largas, así como las 5,000 regiones codificantes más largas, y las 5,000 regiones codificantes más largas y no redundantes. Las regiones codificantes finales incluirán aquellas regiones codificantes que hayan tenido un “hit” con BLASTp o HMMER. En la Figura 40 se muestra un ejemplo de la tercera fase ya finalizada, donde se hizo uso de los resultados obtenidos en la segunda fase.



```
gmelendrez@omica:/LUSTRE/bioinformatica_data/biocomp/gmelendrez/shrimp/transde
File Edit View Search Terminal Help
* Running CMD: /LUSTRE/bioinformatica_data/biocomp/gmelendrez/tools/bin/TransDecoder-5.0.2/util/PWM/compute_AUC.pl Trinity.fasta.transdecoder_dir/start_refinement.enhanced.feature.scores.roc
null device
1
* Running CMD: /LUSTRE/bioinformatica_data/biocomp/gmelendrez/tools/bin/TransDecoder-5.0.2/util/PWM/make_seqLogo.Rscript Trinity.fasta.transdecoder_dir/start_refinement.enhanced.+_pwm || :
Error in library(seqLogo) : there is no package called 'seqLogo'
Execution halted
* Running CMD: /LUSTRE/bioinformatica_data/biocomp/gmelendrez/tools/bin/TransDecoder-5.0.2/util/start_codon_refinement.pl --transcripts Trinity.fasta --gff3_file Trinity.fasta.transdecoder_dir/longest_orfs.cds.best_candidates.gff3 > Trinity.fasta.transdecoder_dir/longest_orfs.cds.best_candidates.gff3.revised_starts.gff3
-ignoring line PFAM output found and processing...
-indexing [comp9993_c0--comp9993_c0_seq1.p1] 1
-number of revised start positions: 3512
CMD: cp Trinity.fasta.transdecoder_dir/longest_orfs.cds.best_candidates.gff3.revised_starts.gff3 Trinity.fasta.transdecoder.gff3
CMD: /LUSTRE/bioinformatica_data/biocomp/gmelendrez/tools/bin/TransDecoder-5.0.2/util/gff3_file_to_bed.pl Trinity.fasta.transdecoder.gff3 > Trinity.fasta.transdecoder.bed
-indexing [comp9993_c0--comp9993_c0_seq1.p1] 1
CMD: /LUSTRE/bioinformatica_data/biocomp/gmelendrez/tools/bin/TransDecoder-5.0.2/util/gff3_file_to_proteins.pl --gff3 Trinity.fasta.transdecoder.gff3 --fasta Trinity.fasta > Trinity.fasta.transdecoder.pep
-indexing [comp9993_c0--comp9993_c0_seq1.p1] 1
CMD: /LUSTRE/bioinformatica_data/biocomp/gmelendrez/tools/bin/TransDecoder-5.0.2/util/gff3_file_to_proteins.pl --gff3 Trinity.fasta.transdecoder.gff3 --fasta Trinity.fasta --seqType CDS > Trinity.fasta.transdecoder.cds
-indexing [comp9993_c0--comp9993_c0_seq1.p1] 1
transdecoder is finished. See output files Trinity.fasta.transdecoder.*
[gmelendrez@nodo30 transde]$
```

Figura 40. Tercera fase de TransDecoder utilizando el transcriptoma de (Ghaffari *et al.*, 2014) y los resultados obtenidos en la segunda fase.

Anexo 6 - Descriptores moleculares

En este apéndice se enlistan los 51 descriptores moleculares calculados como requisito previo para la evaluación de los *k*-meros con la máquina de aprendizaje.

Tabla 13. Lista de descriptores moleculares calculados

#	Descriptor
1	Longitud de la secuencia
2	Composición del aminoácido "F" (Dubchak <i>et al.</i> , 1995)
3	Carga neta con $pH = 9$
4	Distribución del grupo "MHKFRYW" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 50% de la clase "[GASTCPD],[NVEQIL],[MHKFRYW]"
5	Composición del aminoácido "M" (Dubchak <i>et al.</i> , 1995)
6	Distribución del grupo "NVEQIL" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 75% de la clase "[GASTCPD],[NVEQIL],[MHKFRYW]"
7	Composición del aminoácido "Q" (Dubchak <i>et al.</i> , 1995)
8	Momento hidrofóbico a un ángulo de 100° utilizando la escala de (Eisenberg <i>et al.</i> , 1984)
9	Distribución del grupo "PATGS" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 0% de la clase "[LIFWCMVY],[PATGS],[HQRKNED]"
10	Distribución del grupo "LIFWCMVY" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 0% de la clase "[LIFWCMVY],[PATGS],[HQRKNED]"
11	Promedio de la composición de la secuencia utilizando la escala de (Klein <i>et al.</i> , 1984)
12	Distribución del grupo "HQRKNED" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 25% de la clase "[LIFWCMVY],[PATGS],[HQRKNED]"
13	Carga neta de la secuencia utilizando la escala de (Charton y Charton, 1983a,b)
14	Promedio de la hidrofilia de la secuencia utilizando la escala de (Kuhn <i>et al.</i> , 1995)
15	Promedio de la hidrofobicidad (CIDH920102) de la secuencia utilizando la escala de (Cid <i>et al.</i> , 1992)
16	Promedio de la hidrofobicidad (CIDH920104) de la secuencia utilizando la escala de (Cid <i>et al.</i> , 1992)
17	Promedio de la hidrofobicidad (CIDH920105) de la secuencia utilizando la escala de (Cid <i>et al.</i> , 1992)
18	Promedio de la hidrofobicidad (MANP780101) de la secuencia utilizando la escala de (Manavalan y Ponnuswamy, 1978)
19	Promedio de la hidrofobicidad (PONP800105) de la secuencia utilizando la escala de (Ponnuswamy <i>et al.</i> , 1980)
20	Distribución del grupo "GASDT" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 75% de la clase "[GASDT],[CPNVEQIL],[KMHFRYW]"
21	Promedio de la hidrofobicidad (PRAM900101) utilizando la escala de (Prabhakaran, 1990)
22	Promedio de la hidrofobicidad (SWER830101) utilizando la escala de (Sweet y Eisenberg, 1983)
23	Distribución del grupo "GASDT" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 100% de la clase "[GASDT],[CPNVEQIL],[KMHFRYW]"
24	Promedio de la hidrofobicidad (ZIMJ680101) utilizando la escala de (Zimmerman <i>et al.</i> , 1968)
25	Promedio de la hidrofobicidad (WOLR790101) utilizando la escala de (Wolfenden <i>et al.</i> , 1979)

Continúa en la siguiente página

Tabla 13 – Continuación

#	Descriptor
26	Promedio de la hidrofobicidad (CASG920101) utilizando la escala de (Casari y Sippl, 1992)
27	Promedio de la hidrofobicidad (TOSSI2002) utilizando la escala de (Tossi <i>et al.</i> , 2002)
28	Distribución del grupo "EALMQKRH" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 50% de la clase "[EALMQKRH],[VIYCWFT],[GNPSD]"
29	Distribución del grupo "EALMQKRH" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 100% de la clase "[EALMQKRH],[VIYCWFT],[GNPSD]"
30	Composición del grupo "CLVIM" utilizando la escala de (Murphy <i>et al.</i> , 2000) de la clase "[FWY],[CLVIM],[H],[AG],[ST],[DENQ],[KR],[P]"
31	Distribución del grupo "DE" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 0% de la clase "[KR],[ANCQGHILMFSTWYV],[DE]"
32	Composición del grupo "FWY" utilizando la escala de (Murphy <i>et al.</i> , 2000) de la clase "[FWY],[CLVIM],[H],[AG],[ST],[DENQ],[KR],[P]"
33	Distribución del grupo "KR" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 100% de la clase "[KR],[ANCQGHILMFSTWYV],[DE]"
34	Composición del grupo "MHKFRYW" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[GASTCPD],[NVEQIL],[MHKFRYW]"
35	Distribución del grupo "ALFCGIVW" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 0% de la clase "[ALFCGIVW],[RKQEND],[MPSTHY]"
36	Distribución del grupo "MPSTHY" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 0% de la clase "[ALFCGIVW],[RKQEND],[MPSTHY]"
37	Distribución del grupo "RKQEND" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 0% de la clase "[ALFCGIVW],[RKQEND],[MPSTHY]"
38	Distribución del grupo "RKQEND" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 25% de la clase "[ALFCGIVW],[RKQEND],[MPSTHY]"
39	Composición del grupo "KMHFRYW" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[GASDT],[CPNVEQIL],[KMHFRYW]"
40	Composición del grupo "DE" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[KR],[ANCQGHILMFSTWYV],[DE]"
41	Composición del grupo "KR" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[KR],[ANCQGHILMFSTWYV],[DE]"
42	Distribución del grupo "VIYCWFT" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 50% de la clase "[EALMQKRH],[VIYCWFT],[GNPSD]"
43	Composición del grupo "ALFCGIVW" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[ALFCGIVW],[RKQEND],[MPSTHY]"
44	Transición del grupo "CLVIMFW" al grupo "RKEDQN" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[RKEDQN],[GASTPHY],[CLVIMFW]"
45	Composición del tripéptido conformado por los grupos "[RKEDQN][CLVIMFW][GASTPHY]" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[RKEDQN],[GASTPHY],[CLVIMFW]"
46	Composición del tripéptido conformado por los grupos "[CLVIMFW][CLVIMFW][GASTPHY]" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[RKEDQN],[GASTPHY],[CLVIMFW]"
47	Transición del grupo "ALFCGIVW" al grupo "RKQEND" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[ALFCGIVW],[RKQEND],[MPSTHY]"
48	Distribución del grupo "GASTPHY" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 0% de la clase "[RKEDQN],[GASTPHY],[CLVIMFW]"

Continúa en la siguiente página

Tabla 13 – *Continuación*

#	Descriptor
49	Distribución del grupo "CLVIMFW" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 0% de la clase "[RKEDQN],[GASTPHY],[CLVIMFW]"
50	Composición del tripéptido conformado por los grupos "[CLVIMFW][CLVIMFW][CLVIMFW]" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) de la clase "[RKEDQN],[GASTPHY],[CLVIMFW]"
51	Distribución del grupo "GASTPHY" utilizando la escala de (Tomii y Kanehisa, 1996; Li <i>et al.</i> , 2006) a un 75% de la clase "[RKEDQN],[GASTPHY],[CLVIMFW]"