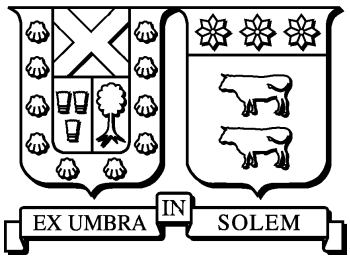
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA

SANTIAGO – CHILE




# "ALGORITMO PARA EL CÁLCULO DE FRAGMENTOS DE PROTEÍNAS EN LOS ORGANISMOS SECUENCIADOS"

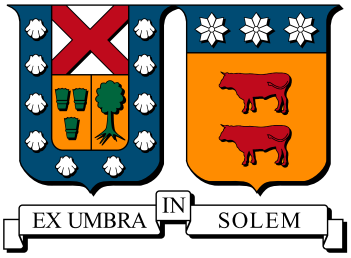
FELIPE NICOLÁS ARAYA BARRERA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: LIOUBOV DOMBROVSKAIA

OCTUBRE 2017

**UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA**
**DEPARTAMENTO DE INFORMÁTICA**
**SANTIAGO – CHILE**




# "ALGORITMO PARA EL CÁLCULO DE FRAGMENTOS DE PROTEÍNAS EN LOS ORGANISMOS SECUENCIADOS"

**FELIPE NICOLÁS ARAYA BARRERA**

**MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INFORMÁTICO**

**PROFESOR GUÍA: LIOUBOV DOMBROVSKAIA**
**PROFESOR CORREFERENTE: DIEGO ARROYUELO BILLIARDI**

**OCTUBRE 2017**

# Agradecimientos

Se escribirán los agradecimientos una vez este documento haya sido terminado.

# Dedicatoria

Se escribirá la dedicatoria una vez este documento haya sido terminado.

# Resumen

Se completará este capítulo una vez que el cuerpo de este trabajo haya finalizado.

# Abstract

Same thing as the previous section.

# Índice de Contenidos

# Índice de cuadros

# Índice de figuras

# Glosario

# Introducción

## Motivación

Para el actual documento, las razones que motivaron al alumno presente a realizar la investigación de su tesis son la de comenzar a entrar en una rama que en los últimos años ha tomado bastante importancia en la informática, conocido como la *Bioinformática* [1], la cual aplica las tecnologías computacionales contemporáneas a datos biológicos que pueden pertenecer a estructuras como ADN, proteínas, entre otras estructuras biológicas complejas y los cuales están a la mano del ser humano como archivos de cadenas de secuencias (en varios formatos) y que pueden ser usados a voluntad.

En el caso puntual de esta memoria, se trabajarán con proteínas (compuestas de combinaciones de 20 aminoácidos) que están distribuidas en un *dataset* cuyo formato del archivo está en **.fasta**, donde cada cadena de polipéptido está compuesto por un ID o código identificador, su nombre taxonómico y su posterior secuencia de aminoácidos.

Hablando de las proteínas, se han realizado numerosos análisis [2, 3] en bases de datos de secuencias con respecto a la cantidad de aminoácidos que se encuentran en total en este tipo de estructuras, distribuyéndolos según su tamaño o para determinar cuál es el aminoácido que más aparece en este tipo de archivos; también según el año de descubrimiento de las proteínas o el tipo de proteína. Todo esto usando fuentes como UniProt y EROP-Moskow.

Como una derivación directa, existe una variabilidad casi infinita de combinaciones llamadas residuos (fragmentos) de aminoácidos (*amino acid residues* o AAR de manera simplificada en inglés) que se determinan según su tamaño $k$ y por las posibles opciones a obtener, que

sigue la regla de combinatoria $20^k$ ya que son 20 los animoácidos base que existen en la actualidad (para dipéptidos serían $20^2 = 400$, para tripéptidos serían $20^3 = 8000$ y así sucesivamente).

# Capítulo 1

# Definición del Problema

## 1.1. Definición

Las proteínas desempeñan un papel fundamental para la vida y son las biomoléculas más versátiles y diversas, las cuales realizan una enorme cantidad de funciones diferentes, tales como enzimáticas, estructurales, inmunológicas, entre otras. Para muchos biólogos y científicos especializados en este tipo de biomolécula resulta crucial investigar sobre tales propiedades anteriormente mencionadas, por lo tanto para ellos es necesario saber o conocer la composición básica de cada una de las proteínas en base a su elemento básico, conocido como el aminoácido (existen 20 diferentes en total). Todas las proteínas se componen de aminoácidos, entregando así una cantidad inmensa de polipéptidos que existen y que van apareciendo gracias al trabajo de investigaciones y proyectos que hacen los encargados de este asunto.

Estas proteínas aparecen registradas en base de datos (como Uniprot, Genbank) con su secuencia, su nombre taxonómico y un código en clave también conocido como ID, las cuales están disponibles en archivos con formato **.fasta** (que pueden abrise usando un editor de texto básico) de la siguiente forma:

```
>sp|Q98957|3SA1A_NAJAT Cytotoxin 1a OS=Naja atra PE=3 SV=1
MKTLLLTLVVVTIVCLDLGYTLKCNKLPPIASKTCPAGKNLCYKMFMMSDLTIPVKRGCI
DVCPKNSLLVKYVCCNTDRCN
>sp|P79810|3SA1C_NAJAT Cytotoxin 1c OS=Naja atra PE=3 SV=1
MKTLLLTLVVVTIVCLDLGYTLKCNKLIPIASKTCPAGKNLCYKMFMMSDLTIPVKRGCI
DVCPKNSHLVKYVCCNTDRCN
>sp|P85429|3SA1F_NAJAT Cytotoxin 1f (Fragment) OS=Naja atra PE=1 SV=1
LKCNKLVPLFYKTCP
>sp|P60304|3SA1_NAJAT Cytotoxin 1 OS=Naja atra PE=1 SV=1
MKTLLLTLVVVTIVCLDLGYTLKCNKLIPIASKTCPAGKNLCYKMFMMSDLTIPVKRGCI
DVCPKNSLLVKYVCCNTDRCN
```

Figura 1.1: Secuencias de varias proteínas en formato **.fasta**

Una de las principales tareas en la investigación de proteínas consiste es buscar residuos de aminoácidos (secuencias de aminoácidos o *aminoacid residues (AAR)* en inglés) en el conjunto universo de las polipéptidos para poder localizar cuáles son los residuos de ciertos tamaños que más aparecen en las bases de datos. El problema principal radica en buscar cuántas veces aparece cierto residuo de una base de datos con tamaños bastante considerables (por ejemplo, buscar una determinada subsecuencia de 2 aminoácidos en una base de datos de 500 GB o superior) sin saber si este residuo está presente o no en aquella base de datos, esto podría provocar gastos innecesarios de tiempo a la hora de realizar este tipo de búsqueda, por consiguiente se desea ocupar el menor tiempo posible para realizar esta tarea.

## 1.2. Objetivos

Lo que se pretende realizar para esta memoria consiste en obtener de un conjunto predeterminado de proteínas el número máximo de fragmentos de péptidos (AAR) que existen asociado a un valor $k$ determinado, donde $k$ se ubicará en el intervalo entre 2 hasta 50, y en base a aquello obtener y determinar para cada $k$ cuáles son los fragmentos de aminóácidos que más se repiten para posteriormente realizar un análisis tanto matemático y biológico de los resultados obtenidos. Para realizar esta tarea se usará como *dataset* la base de datos de proteínas de SwissProt (550100 proteínas) y de TrEMBL (88032926 proteínas).

A partir de esto las implementaciones de código para trabajar estos archivos serán realizados en los lenguajes de programación **C++** y **Python**, con diferentes finalidades que serán explicados a medida que se avance en el documento.

# Capítulo 2

# Estado del Arte

## 2.1. Información previa a considerar

Para entrar de lleno en la tema, es necesario conocer de antemano varios aspectos básicos de la biología.

### 2.1.1. Biomoléculas

Cada vez que se habla de la biología, este concepto se relaciona directamente con la ciencia que estudia a los seres vivos. Ahora bien, las estructuras o compuestos que constituyen una parte esencial de los seres vivos son conocidas como **biomoléculas**. Estas biomoléculas están principalmente constituidas por elementos químicos como el carbono (C), hidrógeno (H), oxígeno (O), nitrógeno (N), fósforo (P) y azufre (S) [4] y se pueden clasificar en biomoléculas inorgánicas, que se encuentran tanto en seres vivos como en los cuerpos inertes, no obstante son imprescindibles para la vida; y las biomoléculas orgánicas, que son sintetizadas por los seres vivos y tienen una estructura con base en carbono. Estas biomoléculas orgánicas se pueden separar en 4 grandes grupos:

1. Glúcidos (hidratos de carbono o carbohidratos): son la fuente de energía primaria que

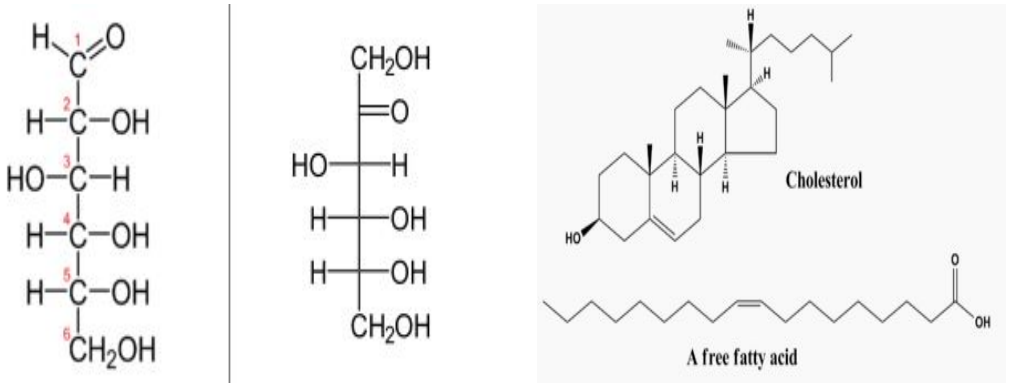
utilizan los seres vivos para realizar sus funciones vitales. Los ejemplos más conocidos son la glucosa, el almidón y el glucógeno.

2. Lípidos: conforman el principal almacén de energía de los animales y desempeñan funciones reguladores de enzimas y hormonas.

3. Ácidos nucleicos: El ácido desoxirribonucleico y el ácido ribonucleico, mayormente conocidos como ADN (DNA) y ARN (RNA y sus derivados) desarrollan posiblemente la función más importante para la vida: contener, de manera codificada, las instrucciones necesarias para el desarrollo y funcionamiento de la célula. El ADN tiene la capacidad de replicarse, transmitiendo así dichas instrucciones a las células hijas que heredarán la información.

4. Proteínas: poseen la mayor diversidad de funciones que realizan en los seres vivos; prácticamente todos los procesos biológicos dependen de su presencia y/o actividad. Son proteínas casi todas las enzimas, catalizadores de reacciones metabólicas, hemoglobina, anticuerpos, entre otros. Su unidad base es el *aminoácido*, por el cual se van formando los péptidos según la cantidad de unidades bases enlazadas.

Dentro de estas biomoléculas, el análisis detallado de los carbohidratos y los lípidos depende en demasía de su estructura química (elementos químicos asociados y tipo de enlaces entre ellos), por lo mismo es una materia más ligada a los químicos (ver Figura 1.1).

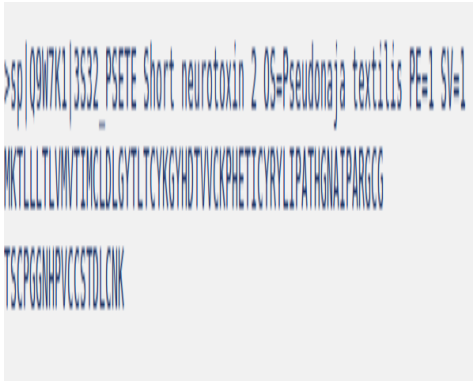



(a) Glúcidos (glucosa y fructosa) (b) Lípidos (colesterol y un ácido graso)

Figura 2.1: Estructura química de los carbohidratos y lípidos.

Sin embargo, el ADN y los polipéptidos poseen unidades base que pueden ser codificadas como letras, por consiguiente pueden ser secuenciados como *cadenas de strings* y en donde los avances computacionales y la evolución informática toman una importante relevancia (ver Figura 1.2).


(a) Cadena de ADN aleatoria



(b) Cadenas $\alpha$ y $\beta$ de hemoglobina bovina

Figura 2.2: Biomoléculas de ADN y péptidos llevadas a cadenas de strings.

Con respecto a estas 2 últimas estructuras, la diferencia visual más notoria radica en la cantidad de diferentes letras (strings) que las componen, para el ADN son 4 [8] y son denominadas **bases nitrogenadas** que son las siguientes:

1. Adenina
2. Timina
3. Citosina
4. Guanina

Para las proteínas, su elemento básico, como ya se mencionó anteriormente es el **aminoácido**, pero ahora se adentrará en más detalle sobre esta molécula.

### 2.1.2. Aminoácidos

Los aminoácidos tienen diferentes funciones en el organismo [5] pero ante todo sirven como **las unidades básicas de los péptidos y de las proteínas.** A nivel orgánico el aminoácido es una molécula compuesta con un grupo amino (-NH2) y un grupo carboxilo (-COOH) y que pueden tener distintas distribuciones. Para el caso de los que componen las proteínas se consideran como alfa-aminoácidos:

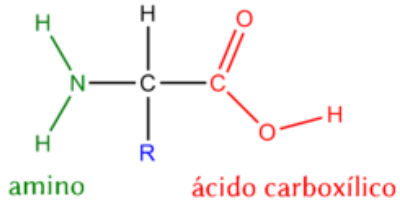



Figura 2.3: Estructura general de un alfa-aminoácido

En la imagen anterior se puede identificar el carbono central (alfa) unido al grupo carboxilo (rojo), grupo amino (verde), un hidrógeno (imagen superior color negro) y el grupo radical (azul) o R. Este grupo radical es el que determina la identidad y las propiedades de cada uno de los diferentes aminoácidos.

El primer aminoácido fue descubierto a principios del siglo XIX, y a partir de ese entonces hasta la actualidad son miles los aminoácidos que han sido descubiertos, pero solo 20 se consideran como los componentes esenciales para las proteínas (y los que se considerarán como parte de esta memoria) que se presentarán a continuación en conjunto con su respectiva abreviación utilizada en las cadenas de proteínas de los archivos FASTA:

| Aminoácido - Abreviatura | | | |
|---|---|---|---|
| Alanina - A | Cisteína - C | Ácido aspártico - D | Ácido glutámico - E |
| Fenilalalina - F | Glicina - G | Histidina - H | Isoleucina - I |
| Lisina - K | Leucina - L | Metionina - M | Aspargarina - N |
| Prolina - P | Glutamina - Q | Arginina - R | Serina - S |
| Treonina - T | Valina - V | Triptófano - W | Tirosina - Y |

Cuadro 2.1: Los 20 aminoácidos existentes y considerados

Existen otras abreviaturas en las cadenas como B, X o J, pero para el alcance de esta memoria no serán considerados como objeto de estudio y análisis posterior.

A partir de este pequeño elemento se forman las macromoléculas que se identifican según la cantidad de aminoácidos (a partir de ahora se mencionarán como aa.) que lo compongan:

| Tamaño | Tipo de estructura |
|---|---|
| 2 aa. | Dipéptido |
| 3 aa. | Tripéptido |
| Entre 2 y 8 aa. | Oligopéptido |
| Menos de 100 aa. | Péptido |
| Mayor o igual de 100 aa. | Proteína o polipéptido |

Cuadro 2.2: Identificación de macromoléculas según cantidad de aminoácidos

## 2.2. Secuencias de proteínas

Desde el momento en que se descubrieron los elementos componentes del ADN y las proteínas, se han investigado sobre las posibles combinaciones que se pueden encontrar entre las bases que los conforman y en que cantidad se encuentran. Para el ADN y sus 4 elementos básicos existen millones de seres vivos, parásitos, virus, protozoos y entre otros que se definen por su código genético, por lo cual encontrar los diversos residuos de bases según un determinado tamaño. En el caso puntual para la finalidad de este escrito y según lo mencionado por [6], es posible estudiar de manera teórica y con fórmulas matemáticas la cantidad máxima de fragmentos que puede formar una proteína. Considerando como base que el número posible de estructuras peptídicas naturales $P$ están compuestas de diferentes residuos de aminoácidos (incluyendo repeticiones en cadenas de aminoácidos) sigue la siguiente fórmula:

$$P = A^n \tag{2.1}$$

Donde $A$ es el número de diferentes aminoácidos existentes, y $n$ es la cantidad de aminácidos correspondientes a la estructura estudiada. Por lo mismo y siguiendo esta fórmula (considerando $A = 20$) la cantidad de diferentes combinaciones péptidos de tamaño k que se pueden obtener se aprecian en la siguiente tabla:

| Tamaño péptido (k) | Combinaciones posibles ($A^k$) |
|---|---|
| 2 aa. | 400 |
| 3 aa. | 8000 |
| 4 aa. | 160000 |
| 5 aa. | 3200000 |
| 10 aa. | $1.024\times10^{13}$ |
| 20 aa. | $1.049\times10^{26}$ |
| 50 aa. | $1.126\times10^{65}$ |

Cuadro 2.3: Combinaciones posibles a obtener según el tamaño del péptido

Según lo observado en esta tabla se identificar que a medida que el valor de k va en aumento, las posibles combinaciones que se pueden obtener de fragmentos de proteínas pueden llegar a tener valores inimaginables para el ser humano corriente; no obstante, no todas estas estructuras existen o son capaces de ser encontradas en la naturaleza [3], aun así la diversidad de la búsqueda de estos residuos sigue siendo gigantesca, y por ende difícil de solucionar, y este será uno de los problemas que se intentará solucionar en esta memoria.

Ahora bien, para un polipéptido de tamaño $n$ aminoácidos, el máximo número posible de fragmentos de tamaño $k$ que teóricamente se podrían obtener (considerando las posibles repeticiones de fragmentos) es descrita mediante la siguiente expresión:

$$N_k^{teorica} = n - k + 1 \tag{2.2}$$

Por consecuencia, el máximo número posible de fragmentos (incluye posibles repeticiones) que teóricamente se pueden obtener para una molécula de tamaño $n$, partiendo desde $k = 2$ (dipéptidos) hasta $k = n - 1$, viene dado por:

$$N_{suma}^{teorica} = \sum_{2}^{n-1} \frac{k(k-1)}{2} - 1 \tag{2.3}$$

Mediante estas fórmulas, se han calculado la cantidad de posibles fragmentos que se pueden obtener en diferentes oligopéptidos y proteínas:

| Número | Oligopéptido/Proteína | $n$ | $N_{suma}^{teorica}$ |
|---|---|---|---|
| 1 | Encefalina (varios tipos biológicos) | 5 | 9 |
| 2 | Bradiquinina (mamíferos) | 9 | 35 |
| 3 | ACTH (humanos) | 39 | 740 |
| 4 | Cadena $\alpha$ hemoglobina (humanos) | 141 | 9869 |
| 5 | Cadena $\beta$ hemoglobina (humanos) | 146 | 10584 |

Cuadro 2.4: Número máximo posible de fragmentos que se pueden formar en 5 proteínas

Considerando que para un polipéptido de largo $n$ aminoácidos, si este valor de $n$ es muy alto, se puede obtener una cantidad muy alta de fragmentos de dipéptidos, pero muchos de estos dipéptidos se pueden repetir varias veces en la cadena, por lo tanto, cuando se desea obtener **el máximo número de fragmentos diferentes** asociado a un valor $k$ determinado, este puede tener un valor muy bajo en comparación con la cantidad total de fragmentos obtenidos. Por medio de las fórmulas descritas anteriormente, se puede obtener el número máximo de diferentes fragmentos (o fragmentos esperables) asociado a un tamaño $k$:

$$N_k^{diff} = N_k^{teorica} - R_k \tag{2.4}$$

Este valor $R_k$, se obtiene introduciendo nuevos parámetros $i$ (que es el número de estructuras idénticas para determinado $k$) y $m$ (el número de diferentes estructuras para el determinado $k$):

$$R_k = \sum_{1}^{m} (i-1) \tag{2.5}$$

Por lo tanto, el número máximo de fragmentos diferentes que se pueden obtener en una proteína sigue la siguiente fórmula:

$$N_{suma}^{diff} = \left[ \sum_{2}^{n-1} \frac{k(k-1)}{2} - 1 \right] - \sum_{2}^{n-1} \left[ \sum_{1}^{m} (i-1) \right] \tag{2.6}$$

Tomando la información de la base de datos de oligopéptidos EROP-Moscow, para mostrar los valores obtenidos con estas fórmulas, se usará como ejemplo la caseína bovina (proteína proveniente de la vaca). Esta proteína se compone de 4 subunidades, $\alpha - s1$, $\alpha - s2$, $\beta$ y $\kappa$. La siguiente tabla muestra las cantidades teóricas y diferentes de fragmentos obtenidos como dipéptidos y sus sumas totales:

| Número | Caseína bovina (subunidad) | $n$ | $N_2^{teorica}$ | $N_2^{diff}$ | $N_{suma}^{teorica}$ | $N_{suma}^{diff}$ |
|---|---|---|---|---|---|---|
| 1 | $\alpha - s1$ | 199 | 198 | 134 | 19700 | 19621 |
| 2 | $\alpha - s2$ | 207 | 206 | 131 | 21320 | 21216 |
| 3 | $\beta$ | 209 | 208 | 124 | 21735 | 21641 |
| 4 | $\kappa$ | 169 | 168 | 118 | 14195 | 14138 |
| 5 | $\alpha - s1 + \alpha - s2 + \beta + \kappa$ | 784 | 780 | 260 | 76950 | 76304 |

Cuadro 2.5: Número máximo posible de fragmentos que se pueden obtener en una proteína de caseína bovina.

Se puede identificar que para los fragmentos de dipéptidos, la cantidad de diferentes fragmentos es bastante menor que la cantidad total de fragmentos obtenidos para las 4 subunidades, pero aún así la cantidad de fragmentos diferentes totales obtenidos es prácticamente la misma que la cantidad de fragmentos totales sin diferenciar. Esto es notorio ya que si $k$ va en progresivo aumento, el universo combinatorio de posibles fragmentos formados se acorta drásticamente, lo que también favorece a la baja formación de fragmentos que se repiten.

## 2.3. Técnicas utilizadas en el problema

En [2] se menciona que buscar secuencias de proteínas en un predeterminado archivo (puede ser de texto o .fasta) es una tarea muy compleja, ya que se formaría un escenario similar al buscar una *aguja en un pajar* y recorrer millones de secuencias en cada búsqueda no sería lo más conveniente considerando que la cantidad de proteínas que existen el día de hoy son muchas, por consiguiente una herramienta recomendable sería preprocesar la base de datos de proteínas con alguna técnica conocida o implementada. A continuación se hablará de forma general de algunos algoritmos conocidos que han tratado este problema.

### 2.3.1. Algoritmo de fuerza bruta

Este es el algoritmo más simple posible[8], ya que dado un texto de tamaño $n$ se revisan todas las posiciones posibles de un patrón de tamaño $k < n$ desde el comienzo hasta el final del texto (izquierda a derecha). Para el caso puntual de los archivos .fasta de las proteínas es necesario extraer únicamente las cadenas de secuencias respectivas y adjuntarlas línea a línea (de esa forma es más fácil trabajarlas). Luego, y siguiendo la presunción matemática del número máximo de fragmentos de tamaño $k$ cada una de las cadenas se revisa desde la posición 0 hasta la posición *n-1* (el valor de $n$ varía según el largo de la cadena de cada proteína) yendo de caracter a caracter un total de *n-k+1* veces para cada secuencia. Lo siguiente muestra una implementación simple en lenguaje C++ acomodada para obtener los diferentes substrings de largo $k$ en un texto de proteínas:

```
1 set <string> unicos;
2 ifstream proteinas("proteinas.txt"); //nombre generico
3 string secuencia;
4 for (int j = 0; j < cantidad_ss; j++)
5 {
6     getline(proteinas, secuencia);
7     int n = secuencia.size();
8     int maximo = n-k+1; // k entre 1 a 50
```

```
9       for (int i = 0; i < maximo; i++)
10      {
11          string pruebas = secuencia.substr(i,k);
12          if (pruebas.find_first_of("BOUXZ")==std::string::npos)
13          {
14              unicos.insert(pruebas);
15          }
16      }
17  }
18  cout << unicos.size() << endl;
```

Listing 2.1: Búsqueda utilizando fuerza bruta en C++

Haciendo un resumen de este código lo que hace es agregar TODOS los substrings totales de tamaño *k* que encuentra en una proteína moviéndose letra por letra en un total de *n-k+1* ocasiones(el getline toma la cadena y la pasa al parámetro *string*, lo hace hasta tomar todas las cadenas de proteínas en el texto), y se verifica si en el substring no posee algún aminoácido que no se encuentre dentro de los 20 aa. mencionados anteriormente, en caso afirmativo el substring se ingresa al set (cuya cualidad es no repetir elementos en su conjunto). Una vez realizado todo esto se puede obtener el número de diferentes substrings de tamaño *k* que se encuentran en el texto, que es la gran ventaja que posee esta implmentación, por lo que puede servir como un "verificador de resultados" cuando se analicen los resultados de implementaciones más óptimas.

Como desventajas importantes resalta el hecho de que el tiempo de la implementación es muy lento ya que al saltar letra por letra y realizando todas las comparaciones posibles de los substrings se pierde bastante tiempo (la complejidad es de $O(n-k+1)$ por cadena, si se tiene que $N$ es la cantidad total de proteínas la complejidad en el texto sería de $O(N(n-k+1))$), que también es negativamente afectado por el tamaño del texto a trabajar (usar un texto de 40 GB como el "uniprot_trembl.fasta" demoraría casi 20 horas en obtener los diferentes substrings para un $k = 20$). Además para guardar un "set<string>" en C++ requiere de una gran capacidad de memoria RAM, la cual va peligrosamente en aumento si *k* sube su valor. Por otra parte esta implementación tampoco entrega la factibilidad de encontrar una manera

de extraer cuáles son los substrings de tamaño $k$ que más se repiten. Por consiguiente esta implementación básica no ayuda a realizar por completo la tarea esperada para este trabajo.

### 2.3.2. Algoritmos de búsqueda de strings

Esta clase de algoritmos (en inglés conocidos como *string searching algorithm*) tratan de localizar si uno o varios strings solicitados (que también se llaman patrones) aparecen en un string más largo o simplemente un texto como tal, considerando el alfabeto que por el cual está compuesto el string de destino o texto. En la mayoría de las ocasiones este alfabeto ($\Sigma$) es el que determina el rendimiento de determinado algoritmo (una variación de este alfabeto puede ayudar o perjudicar la eficiencia de un algoritmo en particular), como también el texto a analizar. Una clasificación básica de estos algoritmos se puede realizar según la cantidad de strings a encontrar:

1. Algoritmos de búsqueda de un único patrón (*Single pattern algorithms*)
2. Algoritmos de búsqueda de múltiples patrones (*Multiple pattern algorithm*)

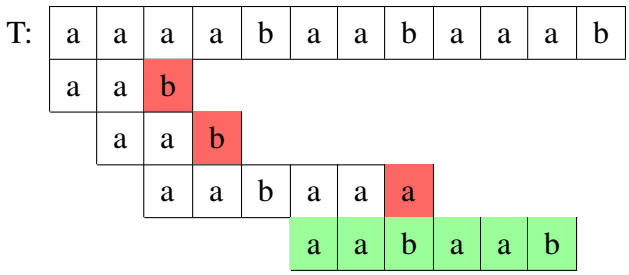
#### Algoritmos de búsqueda de un único patrón

Como lo dice el mismo título y hablando de manera más formal, esta clase de algoritmo de búsqueda consiste en encontrar las ocurrencias de un determinado patrón[9] $p = p_1 p_2 \dots p_m$ en un texto largo $T = T_1 T_2 \dots T_n$, donde $p$ y $T$ son secuencias de caracteres que provienen de set finito de caracteres $\Sigma$. Para este caso se describirán de manera sencilla los algoritmos más reconocidos que han sido desarrollados para este problema, que son el algoritmo de **Knuth-Morris-Pratt** y el algoritmo de **Boyer-Moore**.

*a*) **Algoritmo de Knuth-Morris-Pratt**

Este algoritmo desarrollado el año 1977 por Donald E. Knuth, James H. Morris, y Vaughan R. Pratt busca como objetivo minimizar la cantidad de comparaciones del patrón $p$ con el

texto $T$ manteniendo una pista de información obtenida en informaciones previas, valiéndose de la ayuda de una función de fallo (preproceso del patrón) que indica cuando la última comparación se puede reusar si existe un fallo (revisar [10] para mayores detalles). Las comparaciones de caracteres se realizan de izquierda a derecha buscando el prefijo más largo posible y usarlo como información importante en las iteraciones siguientes (pseudocódigo en sección Apéndice, algoritmo 1). Se puede tomar como ejemplo el texto “aaaabaabaaab” y el patrón “aabaaa”.




Cuadro 2.6: Ejemplo de uso del algoritmo de Knuth-Morris-Pratt

Apreciando la imagen se puede identificar que el algoritmo KMP realiza un total de 14 comparaciones hasta encontrar que el patrón aparece en el texto, en el caso de la fuerza bruta hubiesen sido necesarias 21 comparaciones hasta identificar que el patrón está en el texto. El tiempo de preprocesamiento del texto va del orden $O(m)$ donde $m$ es el largo del patrón, y el tiempo que demora el patrón en ser ubicado en el texto es aproximadamente del orden $O(n)$ donde $n$ es el largo del texto.

*b*) **Algoritmo de Boyer-Moore**

Este algoritmo desarrollado por Bob Boyer y J. Strother Moore el año 1977 se desmarca del algoritmo KMP ya que para Boyer-Moore se realiza la comparación entre patrón y texto de derecha a izquierda, por ejemplo si hubiera una discrepancia en el último carácter del patrón y el carácter del texto no aparece en todo el patrón, entonces éste se puede deslizar $m$ posiciones sin realizar ninguna comparación extra. En particular, no fue necesario comparar los primeros $m - 1$ caracteres del texto, lo cual indica que podría realizarse una búsqueda en el

texto con menos de $n$ comparaciones; sin embargo, si el carácter discrepante del texto se encuentra dentro del patrón, éste podría desplazarse en un número menor de espacios[11]. Esto le entrega una gran ventaja de tiempo y espacio en comparación al algoritmo KMP, en especial cuando el patrón analizado es grande (compuesto por más de 10 caracteres). Para ello se vale de la ayuda de un preprocesamiento del patrón para obtener la posición de ocurrencia de cada uno de los diferentes caracteres involucrados y un localizador de sufijos (pseudocódigo en sección Apéndice, algoritmo 2). Ahora bien, volviendo a realizar la comparación tomando el texto "aaaabaabaaab" y el patrón "aabaaa":

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T: | a | a | a | a | b | a | a | b | a | a | a | b |
| P1: | a | a | b | a | a | a | | | | | | |
| P2: | | | a | a | b | a | a | a | | | | |
| P3: | | | | | | a | a | b | a | a | a | |

Cuadro 2.7: Ejemplo de uso del algoritmo de Boyer-Moore

Para este ejemplo se puede apreciar que con Boyer-Moore se realizan 9 comparaciones totales hasta finalmente encontrar que el patrón está ubicado en el texto, mejorando el resultado de Knuth-Morris-Pratt. Este algoritmo tiene un tiempo de preprocesamiento del orden de $O(m + k)$ donde $m$ es el largo del patrón y $k$ es cantidad de elementos que componen el alfabeto utilizado; el tiempo que demora en encontrar el patrón en el texto varpia entre el orden $O(n/m)$ para el mejor caso y el orden $O(nm)$ para el peor caso, considerando $n$ como el largo del texto en cuestión.

Mirando de una visión macro, pareciera ser que estos algoritmos tuvieran un mejor comportamiento a nivel de tiempo y rendimiento que el algoritmo de fuerza bruta, ¿pero son realmente convenientes para aplicarlos a archivos grandes de proteínas en formatos .fasta? Por una parte, estos algoritmos requieren de un **patrón**, que en este caso serían los posibles residuos de aminoácidos que existen para un tamaño $k$, por consiguiente, hay que ponerse en el caso de que se buscan los diferentes residuos de proteínas de largo 6, que equivalen a comparar las 64000000 combinaciones posibles y buscarlas en el archivo, que si es grande

(por ejemplo superior a los 100 MB) demoraría mucho tiempo porque en ciertos casos podría ocurrir de que se revise para un solo residuo todo el archivo y no encontrarlo en el texto, en consecuencia revisar a cada rato el archivo desde el principio es un objetivo que se desea evitar para este caso del trabajo.

Casi 40 años han pasado desde que estos 2 algoritmos aparecieron para trabajarlos, sin embargo y con el paso del tiempo han aparecido nuevas mejoras de estas implementaciones (como Horsepool) o simplemente nuevos algoritmos que utilizan acercamientos en base a factores intermedios (como el algoritmo BOM - *Backward Oracle Matching*), aunque estas nuevas implementaciones también caen en lo mismo, volver a utilizar como patrones a todos los potenciales residuos de aminoácidos de largo $k$, y usando valores de $k$ muy largos puede ser una tarea infinita de realizar.

**Algoritmos de búsqueda de múltiples patrones**

Esta clase de algoritmos es una extensión del punto anterior, ya que aquí se buscan de manera simultánea si un conjunto de patrones $P = \{p_1, p_2, \ldots, p_3\}$ aparece en un texto $T$ dado un set de caracteres $\Sigma$ [9]. Las implementaciones más reconocidas para este caso son el algoritmo de **Aho-Corsack** (una extensión de algoritmo de Knuth-Morris-Pratt), el algoritmo de **Commentz-Walter** (una extensión del algoritmo de Boyer-Moore) y el algoritmo de **Set-BOM** (una extensión del algoritmo *Backward Oracle Matching*).

Como es lógico, aumentar la cantidad de palabras (patrones) a comparar en el texto determinado toma un proceso más caro a nivel de tiempo y espacio, pero considerando que se tuviera un gran rango de patrones a buscar, como el caso de todos los potenciales péptidos de tamaño $k$ a encontrar, tomaría una visión mucho más optimista que usando un algoritmo de búsqueda con un solo patrón. No obstante, se repetiría el mismo fenómeno con los archivos de gran tamaño, requiriendo de un patrón para localizar y que puede no estar en el texto, supondría un gasto de tiempo innecesario e infructuoso para este trabajo, si esto se traduce en tener muchos substrings para analizarlos como patrones en este caso, por consiguiente utilizar estos algoritmos no es conveniente.

### 2.3.3. Estructuras de datos para texto

Los casos vistos anteriormente tienen como principal problema que para cada patrón dado, es necesario **revisar el texto desde el inicio hasta llegar a la ubicación donde el patrón existe o simplemente recorrer el texto completo sin ubicar al patrón**, ejemplificando, sería como tener que llevar 20 ovejas de La Serena a Santiago y se va trasladando de una a una generando una pérdida inmensa de tiempo y costo, por ende tiene mucho más sentido agrupar estas ovejas y luego llevarlas todas juntas a su destino. Por lo tanto lo que se busca en definitiva es **organizar todos los patrones en una única estructura de datos**, y para lograr esta tarea, no será necesario tener a mano a los patrones a buscar, sino que al **texto en sí** el cual puede ser agrupado como una larga "cadena de strings".

Llevando a un ejemplo más concreto y sencillo, se tiene la palabra MISSISSIPPI$ que está compuesta por 12 letras o caracteres. Cada caracter se podría representar por medio de un índice (indexar la palabra) y quedaría de la siguiente forma:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| M | I | S | S | I | S | S | I | P | P | I | $ |

El concepto **sufijo** corresponde para este contexto al sector de la palabra desde un punto medio (que puede ser incluso el comienzo de la palabra) hasta el final (el prefijo recurre un sector desde el inicio hasta el punto intermedio), lo cual será útil para el siguiente punto. A partir de la imagen superior es posible obtener los "sufijos" de esta palabra, para posteriormente agrupar todos estos "sufijos" e ir ordenándolos alfabéticamente (observar Cuadro 2.8 en la siguiente página).

La clave de este ordenamiento está en el **nuevo orden que tienen los índices**, los cuales tendrán una vital importancia a la hora del desarrollo de esta memoria, y que pueden ser implementados a partir de la palabra o texto entregado para luego guardarlos y registrarlos en un vector.

Los algoritmos encargados de trabajar con este tipo de indexación son conocidos como **Estructuras de datos para textos** (en inglés se conocen como *Indexed Text Searching*) y de los cuales se describirán los más conocidos.

| | | | |
|---|---|---|---|
| 0 | MISSISSIPPI$ | 11 | $ |
| 1 | ISSISSIPPI$ | 10 | I$ |
| 2 | SSISSIPPI$ | 7 | IPPI$ |
| 3 | SISSIPPI$ | 4 | ISSIPPI$ |
| 4 | ISSIPPI$ | 1 | ISSISSIPPI$ |
| 5 | SSIPPI$ | 0 | MISSISSIPPI$ |
| 6 | SIPPI$ | 9 | PI$ |
| 7 | IPPI$ | 8 | PPI$ |
| 8 | PPI$ | 6 | SIPPI$ |
| 9 | PI$ | 3 | SISSIPPI$ |
| 10 | I$ | 5 | SSIPPI$ |
| 11 | $ | 2 | SSISSIPPI$ |

Cuadro 2.8: Sufijos e índices ordenados según orden alfabético (sector derecho).

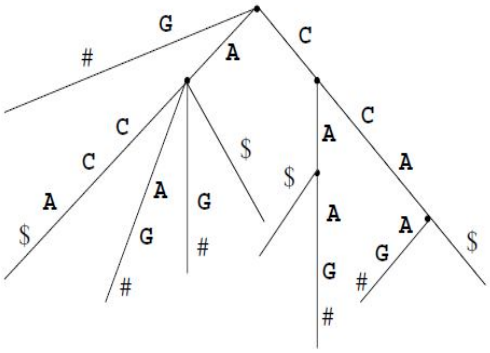
### 2.3.4. Árbol de sufijos

Para la lectura de las secuencias de proteínas y ADN, actualmente se considera el uso de los árboles de sufijos [12] o también conocido como el *suffix tree*, el cual es muy usado para leer cadenas de strings, ya que es ideal para desarrollar diversos problemas relativos al uso de strings como hallar substrings en común, comparación de estadísticas, proyectos relacionados con el genoma, entre otros.

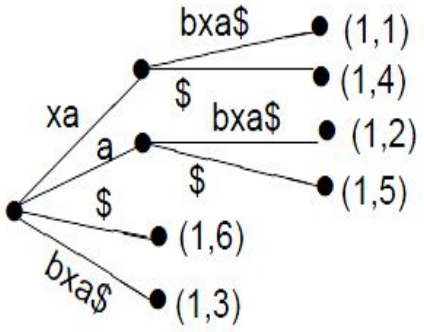
Formalmente hablando, el árbol de sufijos [13] es una estructura de datos comprimida que sirve para almacenar una cadena de caracteres con "información pre-procesada" sobre su estructura interna. Según lo definió Weiner en 1973 [14] el árbol de sufijos $\tau$ para un string $S$ de largo $m = s_1 s_2 ... s_m$ posee un nodo de inicio, y nodos hoja o hijos (que serán al menos 2). Comienza leyendo $s_1$, luego lee $s_2$ (que es $s_1$ quitándole la primera letra respectiva desde izquierda a derecha) identificando si hay strings que ya formaron una descendencia similar para continuar por la hoja y no crear otra y en caso contrario crear una nueva hoja, hasta así leer todo el string. En ciertas oportunidades, un sufijo de $S$ podría coincidir con el prefijo de algún otro sufijo de $S$, por lo tanto el camino para el primer sufijo no terminaría en una hoja. Por ello que la solución creada fue añadir un carácter terminador, siendo usado comúnmente

el $ para este caso.

Esto es muy útil, por ejemplo, para resolver el problema de la subcadena en tiempo lineal ya que se tiene este texto $S$ de longitud $m$ el cual se pre-procesa (se construye el árbol) en tiempo de orden $O(m)$ donde se busca una subcadena $P$ de longitud $n$ (cuyo tiempo es de $O(n)$).



(a) Árbol de sufijos de una cadena de ADN aleatoria




(b) Árbol de sufijos de la palabra *xabxa*

Figura 2.4: Ejemplos de árboles de sufijos.

El problema para la secuencia de ADN considera que habrá un conjunto de textos $S_i$ pertenecientes a la cadena de nucleóticos, y en el cual poder verificar si $P$ es subcadena de algún $S_i$, lo que es llamado *el árbol de sufijos generalizado*:

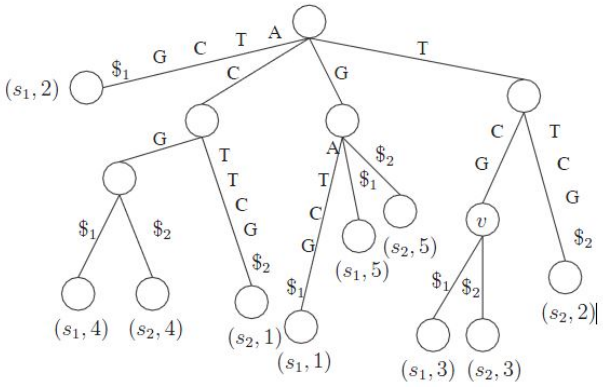



Figura 2.5: El árbol de sufijo generalizado para las secuencias *GATCG$* y *CTTCG$* [17]

### 2.3.5. Arreglo de sufijos

Los arreglos de sufijos fueron introducidos por Udi Mander y Gene Myers el año 1990 [15] y corresponden a una variante del árbol de sufijos que resulta mucho más eficiente en cuanto al uso de la memoria [16]. Hablando de manera más formal, sea $S = s_1, s_2, \ldots, s_n$ una cadena y sea $S[i, j]$ la subcadena que va del índice $i$ hasta $j$. El arreglo de sufijos $A$ de la cadena $S$ va a ser un arreglo de enteros brindando las posiciones iniciales de los sufijos en $S$ en órden lexicográfico. Esto significa que $A[i]$ contiene la posicion inicial del $i$-ésimo sufijo más pequeño en $S$ y por tanto se cumple que para todo $1 < i \leq n : S[A[i-1], n] < S[A[i], n]$. Reutilizando el ejemplo de la palabra MISSISSIPPI$, el arreglo de sufijos que contiene las posiciones iniciales sería el siguiente:

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A[i]$ | 11 | 10 | 7 | 4 | 1 | 0 | 9 | 8 | 6 | 3 | 5 | 2 |

Cuadro 2.9: Arreglo de sufijos $A[i]$ de la palabra MISSISSIPPI$

Para este tipo de estructura, y tal como menciona [18], encontrar un patrón $P$ de tamaño $m$ (donde $m < n$) en el texto consiste en buscar el intervalo $[j, k]$ en el arreglo que contiene a todos los sufijos que comienzan con el patrón mencionado. Esto se logra mediante una búsqueda binaria sobre $A$, buscando lexicográficamente el menor sufijo que parte con $P$, y otra búsqueda binaria buscando el mayor sufijo que parte con $P$. En cada comparación de esta búsqueda binaria se compara un sufijo con el patrón, lo que toma tiempo $O(m)$ en el peor caso, luego la búsqueda completa toma $O(m \log n)$. Junto con el arreglo de sufijos es necesario guardar el texto, y ambos juntos ocupan $O(n \log n)$ bits.

#### Diferencias entre árbol de sufijos y arreglo de sufijos

Ambas estructuras representan (entregan como salida) el mismo resultado, pero con varias discrepancias. La más obvia, es que el *suffix tree* es un árbol donde al recorrer cada una de sus hojas se obtiene el arreglo de sufijos. No obstante, la diferencia importante entre ambos algoritmos radica en **la cantidad de espacio utilizada**. La construcción del árbol de sufijos toma tiempo lineal en el largo del texto. El árbol de sufijos tiene $O(n)$ nodos. Luego, dado que

los substrings de cada rama pueden ser guardados como la posición y el largo de un substring del texto $S$, un árbol de sufijos de $n$ nodos ocupa $O(n \log n)$ bits. Esto permite realizar varias operaciones sobre los nodos del *suffix tree*. Sin embargo esto se convierte en un problema porque implica un tamaño de almacenamiento de varias veces el texto original [18].

Por el contrario el arreglo de sufijos pierde varias de estas funciones a cambio de una importante mejora en cuanto a los requerimientos en espacio de los árboles de sufijos porque el *suffix array* guarda $n$ **enteros**. Por lo cual si se tiene un arreglo donde un entero requiere 4 *bytes* (32 bits), un arreglo de sufijos en ese caso requeriría un total de implementación de $4n$ *bytes* (si el entero necesitara 8 *bytes* (64 bits), el arreglo tendría un tamaño de $8n$ *bytes*). Y esto es significativamente menor que los $20n$ bytes requeridos en la implementación del árbol de sufijos [19].

### 2.3.6. Arreglo LCP

# Capítulo 3

# Propuesta

# Capítulo 4

# Implementación

# Conclusiones

# Apéndice

**Algorithm 1** Knuth-Morris-Pratt

```
1:  n = length[T];
2:  m = length[P];
3:  F = compute_prefix_function(P);
4:  q = 0;
5:  for i = 1 to n do
6:      while q > 0 and P[q + 1] ≠ T[i] do
7:          q = F[q]
8:      end while
9:      if P[q + 1] == T[i] then
10:         q = q + 1
11:     end if
12:     if q == m then
13:         print "patron ocurre con shift" i − m
14:         q = F[q]
15:     end if
16: end for
```

**Algorithm 2** Boyer-Moore

**Require:** String $T$ (texto de largo $n$ caracteres) y $P$ (patron de largo $m$ caracteres)

**Ensure:** Indicacion de que si $P$ es un substring de $T$, o si $P$ no es un substring de $T$

```
1:  i = m − 1
2:  j = m − 1
3:  repeat
4:      if P[j] == T[i] then
5:          if j == 0 then
6:              return i
7:          else
8:              i = i − 1
9:              j = j − 1
10:         end if
11:     else
12:         i = i + m − j − 1
13:         i = i + max(j − last(T[i]), match(j))
14:         j = m − 1
15:     end if
16: until i > n − 1
17: return "No hay substring en T que sea igual a P"
```

# Bibliografía


[1] Rafael Lahoz-Beltrá. *BIOINFORMÁTICA, simulación, vida artificial e inteligencia artificial*. Ediciones Díaz de Santos S.A., Madrid, España, 2004.

[2] Bruce C. Orcutt, Winona C. Barker. *Searching the protein sequence database*. National Biomedical Research Foundation, Georgetown University Medical Center, Washington, D.C., *Bulletin of Mathematical Biology*, 46(4):545-552, 1984.

[3] Alexander A. Zamyatnin. *The Features of an Array of Natural Oligopeptides*. Bakh Institute of Biochemistry, Russian Academy of Sciences, Moskow, Russia; Departamento de Informática, Universidad Técnica Federico Santa María, Valparaíso, Chile, *Neurochemical Journal*, 10(4):249-257, 2016.

[4] Wikipedia, Definición de biomolécula. `https://es.wikipedia.org/wiki/Biomolécula`.

[5] Jan Koolman, Klaus-Heinrich Röhm. *Bioquímica: Texto y Atlas*. Editorial Médica Panamericana, Madrid, España; Georg Thieme Verlag, Stuttgart, Germany, 2004.

[6] A. A. Zamyatnin. *Fragmentomics of Natural Peptide Structures*. Bach Institute of Biochemistry, Russian Academy of Sciences; Universidad Técnica Federico Santa María, Departamento de Informática, Valparaíso, 2009, pp. 405-428.

[7] A. A. Zamyatnin. *Fragmentomics of Oligopeptides and Proteins*. Bach Institute of Biochemistry, Russian Academy of Sciences; Universidad Técnica Federico Santa María, Departamento de Informática, Valparaíso, 2nd Asia-Pacific International Peptide Symposium, 2007.

[8] Búsqueda secuencial de texto, Algoritmo Fuerza Bruta. https://sites.google.com/site/busquedasecuencialdetexto/home.

[9] Gonzalo Navarro, Mathieu Raffinot. *Flexible Pattern Matching Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Universidad de Chile; Centre Nationale de Recherche Scientifique, Marne-La-Vallee, Francia, 2002.

[10] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. *Fast pattern matching in strings*. *SIAM Journal on Computing*, 6(2):323-350, 1977.

[11] Boyer, Robert S. and Moore, J. Strother. *A Fast String Searching Algorithm*. *Communications of the ACM*, Nueva York, Estados Unidos, 20(10):762-772, 1977.

[12] Pekka Kilpelanen. *Lecture 8: Applications of Suffix Trees*. University of Kuopio, Finland; Department of Computer Science, Spring 2005, pp. 1-20.

[13] Introducción a la biología computacional, árboles de sufijos. http://webdiis.unizar.es/asignaturas/TAP/material/4.2.sufijos.pdf, pp. 1-16.

[14] Giulio Pavesi, Giancarlo Mauri, Graziano Pesole. *An algorithm for finding signals of unknown length in DNA sequences*. Department of Computer Science, Systems and Communication, University of Milan-Bicocca, Via Bicocca degli Ancimboldi; Department of General Phisiology and Biochemistry, University of Milan, Via Celoria, April 3rd, 2001, Vol. 17, pp. 207-214.

[15] Manber, Udi and W. Myers, Eugene. *Suffix Arrays: A New Method for On-Line String Searches*. San Francisco, California, USA, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, 319-327, 1990.

[16] Alejandro Deymonnaz. *Arreglos de sufijos para alineamiento de secuencias de ADN con memoria acotada*. Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Departamento de Computación, Argentina, 2012.

[17] Pang Ko, Srinivas Aluru. *Suffix Tree Applications in Computational Biology*. Iowa State University, *Handbook of Computational Molecular Biology*, Chapter 6, 2001.

[18] Andrés Abeliuk Kimmerman. *Árboles de sufijos comprimidos para textos altamente repetitivos*. Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, Departamento de Ciencias de la Computación, Chile, 2012.

[19] Kurtz, Stefan. *Reducing the space requirement of suffix trees*. *Software: Practice and Experience*, 29(13):1149-1171, 1999.