UNIVERSIDAD MAYOR DE SAN ANDRÉS
FACULTAD DE CIENCIAS PURAS Y NATURALES
CARRERA DE INFORMÁTICA

TESIS DE GRADO

ALGORITMOS DE EMPAREJAMIENTO DE SECUENCIAS DE ADN

PARA OPTAR AL TÍTULO DE LICENCIATURA EN INFORMÁTICA
MENCIÓN: INGENIERÍA DE SISTEMAS INFORMÁTICOS

POSTULANTE: FREYSNER NOEL CHAMBI MENDIETA
TUTOR: M.SC. JORGE HUMBERTO TERÁN POMIER

LA PAZ - BOLIVIA

2021

HOJA DE CALIFICACIONES

UNIVERSIDAD MAYOR DE SAN ANDRÉS

FACULTAD DE CIENCIAS PURAS Y NATURALES

CARRERA DE INFORMÁTICA

Tesis de Grado:

ALGORITMOS DE EMPAREJAMIENTO DE SECUENCIAS DE ADN

Presentado por: Freysner Noel Chambi Mendieta

Para optar el grado Académico de Licenciatura en Informática

Mención Ingeniería De Sistemas Informáticos

Nota Numeral: 82

Nota Literal: OCHENTA Y DOS

Ha sido: SOBRESALIENTE

Director a.I. de carrera de Informática: Lic. Nancy Imelda Orihuela Sequeiros

Tutor: M.Sc. Jorge Humberto Terán Pomier

Pdte. Tribunal: Lic. Brigida Alexandra Carvajal Blanco

Tribunal: Lic. Jhonny Roberto Felipez Andrade

Tribunal: Ph.D. Yohoni Cuenca Sarzuri

Tribunal: M.Sc. Aldo Ramiro Valdez Alvarado

UNIVERSIDAD MAYOR DE SAN ANDRÉS
FACULTAD DE CIENCIAS PURAS Y NATURALES
CARRERA DE INFORMÁTICA

LA CARRERA DE INFORMÁTICA DE LA FACULTAD DE CIENCIAS PURAS Y NATURALES PERTENECIENTE A LA UNIVERSIDAD MAYOR DE SAN ANDRÉS, AUTORIZA EL USO DE LA INFORMACIÓN CONTENIDA EN ESTE DOCUMENTO SI LOS PROPÓSITOS SON ESTRICTAMENTE ACADÉMICOS.

LICENCIA DE USO

El usuario está autorizado a:

A) Visualizar el documento mediante el uso de un ordenador o dispositivo móvil.
B) Copiar, almacenar o imprimir si ha de ser de uso exclusivamente personal y privado.
C) Copiar textualmente parte(s) de su contenido mencionando la fuente y/o haciendo la referencia correspondiente respetando normas de redacción e investigación.

El usuario no puede publicar, distribuir o realizar emisión o exhibición alguna de este material, sin la autorización correspondiente.

Dedicado con enorme cariño

Para mis wawas Aglae, (Milagros †), Caleb y Mateo por existir.

Para mi esposa por nadar en la vida a mi lado.

Para mi mamá por inculcarme muchos valores.

Para mi papá por vivir el fútbol con triunfos y derrotas.

Agradecimientos

Agradezco al MSc. Terán por su tiempo, conocimiento y experiencia. Así también le agradezco por aceptar ser el tutor metodológico de este trabajo, gracias por brindarme artículos de investigación, gracias por la predisposición para realizar las revisiones y enormemente agradecido por sugerir los ajustes al desarrollo del tema. Miles de gracias por hacerme notar la diferencia entre un trabajo semestral y un trabajo de fin de Carrera.

Agradezco al Msc. Cuevas por su gran experiencia y orientación en el avance del tema de investigación, además le doy gracias por la motivación e impulso para terminar mi trabajo en la última materia de la carrera.

Un agradecimiento a todos los docentes de las carreras de Informática, Matemática, Estadística y Física que cubren el Pensum de las distintas materias de la Carrera de Informática.

freysner@gmail.com

Resumen

Las máquinas de secuenciación de ADN (ácido desoxirribonucleico) combinan algoritmos de emparejamiento de búsqueda exacta y aproximada para formar algoritmos de ensamblado de secuencias alineando fragmentos pequeños por etapas, hasta conseguir secuencias completas.

En los antecedentes de algoritmos de búsqueda sobre textos existen varios algoritmos, de los cuáles en esta investigación se eligió de búsqueda exacta sin indexación: Knuth-Morris-Pratt, Rabin-Karp, Boyre-Moore y Skip-Search. El desarrollo del tema permitió analizar, describir, experimentar con los algoritmos de emparejamiento sobre secuencias de ADN.

El algoritmo Knuth-Morris-Pratt es comparable con un autómata finito, de este modo es análogo a realizar búsquedas utilizando expresiones regulares en cualquier lenguaje de programación. En implementaciones para secuenciadores se utiliza el algoritmo Rabin-Karp con una estrategia de búsqueda off-line (con indexación). Experimentalmente el algoritmo Boyre-Moore es de mejor rendimiento y el algoritmo Rabin-Karp muestra una menor desviación estándar. El algoritmo Skip-Search alcanza un buen rendimiento haciendo uso de una estructura de datos lista-contenedor.

Los algoritmos de alineamiento son una generalización de los algoritmos aproximados (permitiendo inexactitud) de emparejamiento de secuencias, se introdujo en el tópico eligiendo el algoritmo Needleman-Wunsch que permite alinear dos secuencias con la estrategia de alineamiento global hallando la puntuación óptima de similitud.

Palabras clave: Algoritmos, emparejamiento, búsquedas, ADN, Knuth-Morris-Pratt, Rabin-Karp, Boyre-Moore y Skip-Search.
Metodologia: Investigación científica del tipo descriptivo-experimental.

Abstract

Sequencing DNA (deoxyribonucleic acid) involves exact and approximate search matching algorithms that work aligning small fragments in stages to complete the sequences.

On a review of search algorithms on texts there exist several algorithms, of which the exact search algorithm without indexing: Knuth-Morris-Pratt, Rabin-Karp, Boyer-Moore and Skip-Search were analyzed. The implementation allowed to analyze, describe, experiment with the matching algorithms on DNA sequences.

Knuth-Morris-Pratt algorithm is comparable to a finite automata, thus it is analogous to searching using regular expressions in some programming languages. In implementations for sequencers, the Rabin-Karp algorithm is used with a strategy off-line search (with indexing).

Experimentally, the Boyer-Moore algorithm has a better performance and the Rabin-Karp algorithm shows a lower standard deviation. Skip-Search algorithm achieves good performance using a list-container data structure.

Alignment algorithms are a generalization of the approximate algorithms (allowing inaccuracy) of sequence matching, it was introduced in the topic choosing the Needleman Wunsch algorithm that allows aligning two sequences with the global alignment strategy finding the optimal similarity score.

Keywords: Algorithms, matching, searches, DNA, Knuth-Morris-Pratt, Rabin-Karp, Boyre-Moore and Skip-Search.

Methodology: Scientific research of type descriptive-experimental.

ÍNDICE **Págs.**

ÍNDICE DE FIGURAS **Pág.**

ÍNDICE DE PROGRAMAS **Pág.**

CAPÍTULO I

INTRODUCCIÓN

En **1953**, James Watson y Francis Crick determinaron la **estructura** del ácido desoxirribonucleico (**ADN**, en inglés DNA) y con eso extendieron la ciencia a la **biología molecular**, que no para de evolucionar porque encuentra **aliados** poderosos en las **computadoras**, como sus capacidades de procesamiento e integración para obtener un marco mayor, como el **secuenciamiento** del Genoma Humano.

Un modo de representación de **información** es escribiendo **textos** que **no** pueden descomponerse en **registros independientes**. Este tipo de dato se caracteriza por el hecho de ser una **secuencia lineal** y larga de caracteres, también llamada **cadena**. (Crochemore & Rytter, 1994)

Los algoritmos de **emparejamiento de cadenas** comúnmente los encontramos en **editores de texto** para realizar cambios y/o reemplazos. (Aoe, 1994)

Los algoritmos de emparejamiento de cadenas conducen al **reconocimiento de patrones**, por tanto proveen ideas a diversas áreas como: recuperación de información, motores de búsqueda, reconocimiento de imágenes, minería de datos, bioinformática, detección de intrusos, escaneo de virus, generación de código, análisis sintáctico y otros.

Por lo expuesto anteriormente en esta investigación se desarrolla algoritmos de emparejamiento de secuencias de ADN.

En el **Capítulo I**, se muestra el planteamiento del problema, se construye los objetivos y la justificación y la hipótesis.

En el **Capítulo II**, se construye el marco teórico del tema de investigación donde se aborda los temas: secuencias de ADN, cadenas sobre lenguajes regulares, algoritmos de emparejamiento exacto, autómatas finitos, expresiones regulares y algoritmos de emparejamiento aproximado.

En el **Capítulo III**, se contextualiza el uso de algoritmos en las máquinas de secuenciación. Posteriormente se desarrolla el marco aplicativo del tema de investigación donde se realizo las implementaciones de los algoritmos Knuth-Morris-Pratt, Rabin-Karp, Boyre-More, Skip-Search y Needleman-Wunsch. Por último en este capítulo se realiza un análisis del rendimiento experimental de los algoritmos de emparejamiento.

En el **Capítulo IV**, se muestra las conclusiones del desarrollo de la investigación, para finalmente mostrar las recomendaciones que genero el tema.

1.1 ANTECEDENTES

Aoe (1994) desarrolla el emparejamiento de cadenas por **sub-tópicos** como son: emparejamiento exacto de cadenas, emparejamiento aproximado, emparejamiento por diferentes estructuras de datos y el emparejamiento por hardware.

Gusfield (1997) muestra varios algoritmos de emparejamientos de cadenas aplicados a la biología computacional proporcionando un panorama completo de los fundamentos algorítmicos del **análisis de secuencias** moleculares.

En la Carrera de Informática de la UMSA (Universidad Mayor de San Andrés) se puede encontrar el trabajo de Yañez (2005) de búsqueda en texto, donde se **implementan** los algoritmos de emparejamiento exacto de cadenas como son: KMP (Knuth, Morris y Pratt), BM (Boyer-Moore), RK (Rabin-Karp), Horspool y Shift-Or. Permitiéndole realizar **comparaciones** de rendimiento teóricas y experimentales.

En la Carrera de Informática de la UMSA también existe el trabajo de Fernández (2017) que utiliza la estructura de datos de **árbol** para realizar **búsquedas aproximadas**, que a su vez se nutre del trabajo de Laura (2016) que aplica la búsqueda de palabras en un **diccionario** en aymara.

Nusret, Uzun y Doruk (2017), **implementaron** varios algoritmos de búsquedas exactas en cadenas para buscar etiquetas web en sitios web y así lograr **comparar** los **rendimientos** de varios algoritmos.

Bellekens, Tachtatzis, Atkinson, Renfrew y Kirkham (2014), en su investigación muestran la **importancia** que tiene el emparejamiento de patrones para la **seguridad en redes**, con el fin de encontrar anomalías, detectar intrusiones y reportar incidentes. Para ello implementaron algoritmos de emparejamiento de cadenas utilizando GPU (Graphics Processing Unit) con su capacidad paralela, el conjunto de implementaciones lo denominaron librería GLoP, para la parte **experimental** revisaron **archivos log** en servidores.

Tran, Lee, Hong y Shin (2012), en su trabajo reportan la implementación del algoritmo Aho-Corasick para el **emparejamiento de cadenas múltiples** utilizando GPU, realizando un uso eficiente de las memorias para alcanzar el mejor rendimiento.

1.2 IDENTIFICACIÓN DEL PROBLEMA

La obtención del **genoma humano** significo un costó de unos 3000 millones de dólares y que llevó más de una década en terminarse. Una vez logrado el secuenciamiento del genoma humano, queda por secuenciar el genoma de otras especies como el tulipán, que puede llevar hasta 30 años, debido a que el **genoma de la planta es enorme**, unas 10 veces más que el de un ser humano.

Los datos pueden ser almacenados en diferentes modos, pero el texto es un modo importante de intercambio de información. Además, la cantidad disponible de **datos** en muchos campos **se duplica** cada ocho meses.

Para esta investigación se resalta que las secuencias genéticas de ADN (ácido desoxirribonucleico) , como del **homo sapiens** llegan a pesar 3400Mb en almacenamiento o su equivalente de alrededor de **3 millones de caracteres**.

Es así que realizar el **emparejamiento en secuencias** para contar, encontrar, buscar, alinear ocurrencias de patrones es un **desafío computacional**, incluso el determinar cuánta similitud tiene una secuencia de ADN a otra.

¿La implementación de algoritmos de emparejamiento de secuencias de ADN permiten extraer información exacta o aproximada?

1.3 PLANTEAMIENTO DE LA HIPÓTESIS

La implementación de algoritmos de emparejamiento de cadenas permite el emparejamiento exacto o aproximada de secuencias de ADN.

1.4 OBJETIVOS

1.4.1 Objetivo General

Implementar algoritmos de emparejamiento de cadenas para comparar los rendimientos experimentales

1.4.2 Objetivos Específicos

- ✔ Analizar los algoritmos de emparejamiento exacto de cadenas.
- ✔ Analizar los algoritmos de emparejamiento aproximado de cadenas.
- ✔ Implementar algoritmos de emparejamiento para mostrar su aplicación.
- ✔ Registrar los rendimientos de los programas implementados.

1.5 JUSTIFICACIÓN

En internet, el año 2021, se puede encontrar **software** en el área de **bioinformática** de carácter libre, casi todos desarrollados en universidades del exterior del país. Se justifica investigar algoritmos de emparejamiento de cadenas en la carrera de Informática de la UMSA como un marco teórico descriptivo-experimental y permitir implementar herramientas tecnológicas eficientes de mejor rendimiento y en un futuro **crear** un software con un **propósito específico** en bioinformática.

1.6 MÉTODO DE INVESTIGACIÓN

Con base en la clasificación de Hernández (2010) esta investigación tiene un diseño descriptivo-experimental, para describir los algoritmos de emparejamiento y observar su rendimiento.

CAPÍTULO II

MARCO TEÓRICO

2.1 SECUENCIAS DE ADN (ÁCIDO DESOXIRRIBONUCLEICO)

A partir de los experimentos en vegetales a finales del siglo XIX de **Gregor Mendel** fue posible postular la existencia de **factores heredables**, llamados patrones mendelianos de la herencia, responsables de transmitir las características de una especie de generación en generación. Estos factores heredables fueron posteriormente identificados como genes, secuencias específicas de nucleótidos de ADN ubicadas en los cromosomas de las células. (Flores, 2011)

En 1945 **Watson y Crick** sugirieron un modelo tridimensional para la estructura del ADN y el modo de replicación.

El ADN **vive** en las **células** de todos los **organismos**, pero el ADN no es el mismo en todos los organismos, originando células especializadas en cada organismo.

El ADN es la **molécula** bicatenaria (dos cadenas) de la vida, cada cadena es una **secuencia** de unidades químicas denominadas **nucleótidos**, cada cadena es portadora de la **información genética** y junto al ARN es uno de los dos ácidos nucleicos presentes en todas las células. Enrolladas en un mismo eje, constituyen un doble hélice con una orientación de antiparalelismo.

Un nucleótido es un monómero compuesto por una pentosa (desoxirribosa en el ADN y D-ribosa en el ARN) , un grupo fosfato y una base nitrogenada. Los nucleótidos difieren a nivel de las bases nitrogenadas en dos tipos. Las **purinas** representadas por la guanina (G) y la adenina (A). Las **piramidinas** constituidas por la citosina (C) y la timina (T).

La estructura en doble hélice del ADN ilustrada en la Figura 2.1 (dos hebras o cadenas de nucleótidos se encuentran enrolladas una alrededor de la otra formando una doble hélice), con el apareamiento de bases limitado (A-T; G-C), implica que el ordenamiento lineal de la secuencia de bases de una de las cadenas delimita automáticamente el orden de la otra, por eso se dice que las cadenas son complementarias. Una vez conocida la secuencia de las bases de una cadena, se deduce inmediatamente la secuencia de bases de la complementaria.

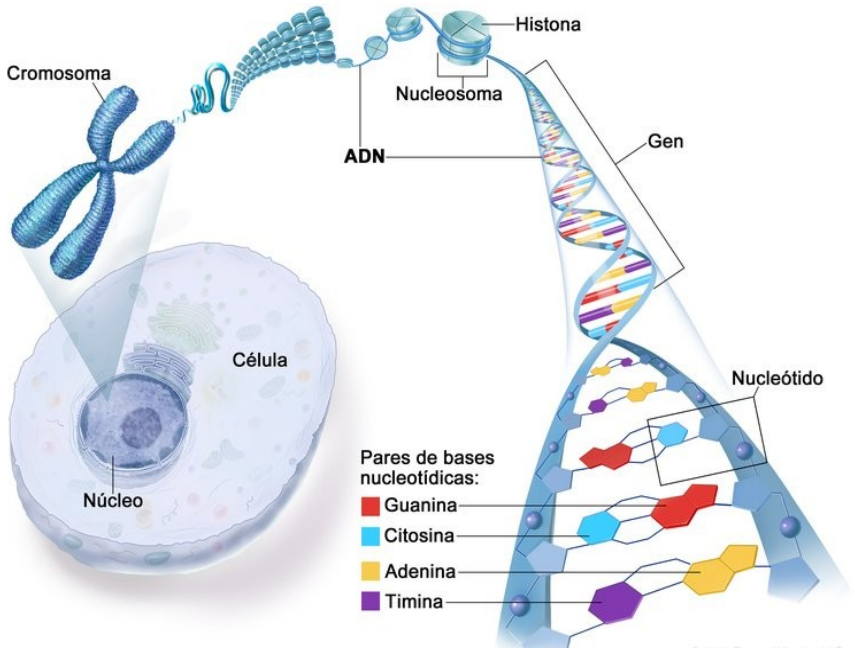

Figura 2.1: Estructura del ADN
Fuente: https://visualsonline.cancer.gov/details.cfm?imageid=9946

No existe una relación directa entre la cantidad de ADN y la complejidad del organismo.

La secuencia de ADN de un gen especifica una proteína en particular mediante los procesos de transcripción y traducción. El ADN se transcribe para obtener una molécula de ARN con un alfabeto similar. En la traducción el ARN se decodifica para obtener una secuencia de aminoácidos en una nomenclatura diferente. El flujo de información de ADN - ARN - proteína se conoce como el **dogma central** de la biología molecular

La organización de la información genética es diferente entre organismos eucariotas y procariotas. En **procariotas** los cromosomas están muy compactados y son traducidos en su totalidad. En **eucariotas** el cromosoma no es tan condensado, los genes aparecen dispersos y son más complejos. Existen, además, zonas de ADN del que no se conoce la función.

En bioinformática se realiza el **análisis de la secuencia de ADN,** para el **descubrimiento de similitudes estructurales y funcionales, y las diferencias entre** múltiples secuencias biológicas**.** Esto puede hacerse **comparando** las nuevas (**desconocidas**) con las bien estudiadas (**conocidas**) secuencias. (Meneses, Rozo, Franco, 2011)

El análisis incluye la alineación de secuencias, la búsqueda en la base de datos de secuencias, el descubrimiento de patrones, la reconstrucción de las relaciones evolutivas, y la formación y la comparación del genoma.

La obtención de la secuencia es a través de complejos procesos en laboratorios. La **secuenciación de ADN** es el proceso que permite obtener la secuencia de bases de los nucleótidos (A, T, C y G) de un fragmento de ADN. Frecuentemente se secuencian regiones de ADN de hasta 900 pares de bases con un método llamado **secuenciación de Sanger** o método por terminación de cadena, en honor al desarrollador del método, el bioquímico británico Fred Sanger y sus colegas en 1977.

Obtenida las secuencias de ADN, en biología molecular utilizando herramientas de bioinformática se pueden realizar investigaciones de:

- ✔ Ensamblado de secuencias de ADN.
- ✔ Búsqueda de genes.

- ✔ Análisis y comparación de secuencias biológicas.
- ✔ Búsqueda de homologías o similitudes.
- ✔ Inferencia filogenética.
- ✔ Simulación de procesos genéticos.
- ✔ Inferencia de funciones en estructuras de proteínas.
- ✔ Diseño de proteínas,Etc.

Las bases de datos que contienen secuencias de ADN, ARN (Ácido ribonucleico) y proteínas se clasifican como: primarias, secundarias y combinadas. (Meneses, Rozo, Franco, 2011)

Primarias: contienen información solamente de la secuencia o la **estructura**, es decir que los datos experimentales son directamente subidos a la base de datos. En esta categoría encontramos las bases de datos GenBank, DNA Data Bank of Japan (DDJB) , UniProtKB/SwissProt, UniProtKB/TrEMBL y Protein Data Bank (PDB)

Secundarias: contienen **información derivada** de las bases de datos primarias. Una base de datos secundaria de secuencia contiene información de la conservación de la secuencia, patrones de secuencia y residuos del sitio activo de familias de proteínas derivados de alineamientos múltiples entre secuencias evolutivamente relacionadas. Una base de datos secundaria organiza las entradas de PDB clasificándolas, por ejemplo, de acuerdo a su estructura como: alfa, beta, alfa-beta, etc. Algunos ejemplos de éstas bases de datos son: CATH y SCOP.

Compuestas: combinan una variedad de fuentes primarias de datos, como por ejemplo, el National Center for Biotechnology Information (NCBI) que alberga un conjunto de bases de datos de secuencia, taxonomía, genomas, mutaciones, entre otras y además herramientas como BLAST para búsquedas por similitud de secuencia.

De la secuenciación del genoma humano se sabe que esta constituido por 3000 megabases ($3x10^9$) con una estructura compleja, que codifican los genes estructurales de ARN

y estas codifican a las proteínas. Las secuencias del genoma humano pueden clasificarse por su grado de repetición o por su significado funcional. Por su grado de repetición se clasifican en: alto grado de repetición, de moderado grado de repetición y secuencias únicas. Por su función se clasifican en: codificantes y no codificantes. (Solari, 2004)

Las secuencias de ADN cuanto más grande es su extensión entonces mayor será el número posible de permutaciones que se pueden formar con las cuatro bases, cómo se muestra en la figura 2.2:

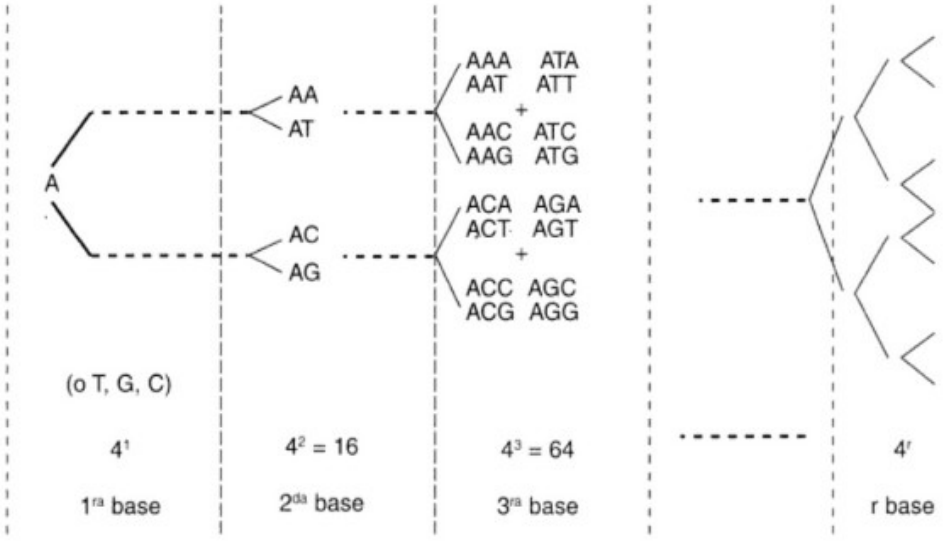

Figura 2.2: Esquema de permutaciones posibles de secuencias de ADN
Fuente: (*Solari*, 2004)

El ratón y la especie humana tienen el mismo número de genes e igual tamaño de ADN (28000 genes y del tamaño de 3000Mb) y el doble de genes respecto de la mosca Drosophila melanogaster (13000 genes y del tamaño de 180Mb).

2.2 CADENAS Y LENGUAJES

Un modo de representación de información es escribiendo **textos** que no pueden descomponerse en registros independientes. Este tipo de dato se caracteriza por el hecho de ser una secuencia lineal y larga de caracteres, también llamada **cadena**. (Sedgewick, 1995)

Las cadenas son el centro de los editores de texto, contienen una variedad de posibilidades para manipular cadenas: búsqueda, cambios, reemplazos,etc.

El **alfabeto** Σ es un conjunto finito de **caracteres**. El tamaño del alfabeto se denota por |Σ|.

Una **cadena** (T) es una serie de caracteres tomadas del **alfabeto** (Σ). El tamaño de la cadena es el número de caracteres que la compone y se denota como |T|.

El termino **lenguaje** se refiere a cualquier conjunto de cadenas de un alfabeto fijo. En teoría del lenguaje, los términos frase y palabra a menudo se utilizan como sinónimo de cadena. (Aho, Sethi & Ullman, 1990)

Las **cadenas** como datos pueden ser de **distinta naturaleza**, siendo archivos completos o partes de un archivo: el texto de un libro, los bits de los gráficos de una computadora, una página HTML (Hyper Text Transfer Protocol) en la web, registros log de un servidor, un genoma representada por su secuencia de ADN, etc. (Sedgewick, 1995)(Nusret, Uzun y Doruk, 2017)(Bellekens, Tachtatzis, Atkinson, Renfrew y Kirkham, 2014)(Navarro y Raffinot, 2000).

La **cadena vacía** es un caso especial donde el tamaño es |T| = 0 y se denota como ϱ.

Dado una cadena $T = t_0 ... t_{n-1}$ de tamaño n, la cadena $T_{pref} = t_{0..} t_j$ es un **prefijo** de T si $j \leq n-1$.

Dado una cadena $T = t_0 ... t_{n-1}$ de tamaño n, la cadena $T_{suf} = t_i ... t_{n-1}$ es un **sufijo** de T si $i \geq 0$.

Dado una cadena $T = t_0 ... t_{n-1}$ de tamaño n, la cadena $T_{sub} = t_i ... t_j$ es una **subcadena** de T si $i \geq 0$ y $j \leq n-1$.

2.3 TÓPICOS DE ALGORITMOS DE BÚSQUEDA

Aoe (1994) desarrolla la búsqueda de cadenas por tópicos como son:

- ✔ Búsqueda exacta de cadenas (un sólo patrón).
- ✔ Búsqueda de múltiples patrones en cadenas.
- ✔ Búsqueda aproximada.
- ✔ Búsqueda multidimensional.
- ✔ Y la búsqueda por hardware.

Charras y Lecroq (2004) describen y muestran 38 implementaciones realizadas en C de algoritmos de búsqueda exacta de cadenas.

Nusret, Uzun y Doruk (2017) realizan estudios experimentales de algoritmos de búsqueda exacta, donde citan los siguientes algoritmos: Apostolico Crochemore, Apostolico Giancarlo, Backward Nondeterministic Dawg Matching, Backward Oracle Matching, Boyer Moore, Brute Force, Colussi, Deterministic Finite Automaton, Forward Dawg Matching, Galil Giancarlo, Horspool, Karp Rabin, KMP Skip Search, Knuth Morris Pratt, Maximal Shift, Morris Pratt, Optimal Mismatch, Quick Search, Reverse Colussi, Reverse Factor, Shift Or, Simon, Skip Search, Smith, String Matching on Ordered Alphabets, Tuned Boyer Moore, Turbo BM, Turbo Reverse Factor y Two Way. La implementaciones de los algoritmos las escribieron en C#.

Kouzinopoulos, Michailidis y Margaritis (2015) hacen notar la importancia de los algoritmos de búsqueda de múltiples claves en aplicaciones biológicas experimentando con los algoritmos: Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber y SOG (Shift-Or extendido).

Navarro (1998) realiza una taxonomía de algoritmos aproximados con la siguiente clasificación: basados en matrices llenadas por programación dinámica, por autómatas, por filtros y bit-paralelismo. Además muestra la importancia de los algoritmos aproximados para búsquedas on-line y para búsquedas en índices.

Keng (2006) elabora una adaptación de Landau y Vishkin para combinar la programación dinámica con árboles sufijos y realizar búsquedas aproximadas.

2.4 ALGORITMOS DE BÚSQUEDA POR EMPAREJAMIENTO EXACTO

El problema de búsqueda de cadenas (del inglés **string matching**) es utilizado para acceder a la información y frecuentemente aparece en muchas aplicaciones, esta operación es comparable al ordenamiento y las operaciones aritméticas.(Crochemore & Rytter, 1994)

Los métodos de programación de los diferentes algoritmos sirven de paradigmas para el diseño de software en otras áreas de las ciencias de la computación. (Charras & Lecroq, 2004)

Los datos pueden ser almacenados en diferentes modos, pero el texto es un modo importante de intercambio de información. En el campo de la literatura los datos se almacenan en grandes corpus y diccionarios, en ciencias de computación los datos son almacenados en enormes archivos lineales. También lo anterior es aplicable al campo de la biología molecular donde instancias de moléculas biológicas se representan mediante aproximación de secuencias de nucleótidos o aminoácidos. Por último se debe subrayar que la cantidad disponible de datos en muchos campos se duplica cada ocho meses.

El emparejamiento de cadenas es un problema clásico en ciencias de la computación, así también en otros campos como reconocimiento de patrones, escaneo de virus, filtración de contenidos, motores de búsquedas y sistemas de detección de intrusos. (Liu y Xu, 2012).

La **cadena** llamada **patrón** de tamaño m se encuentra dentro del **texto** de longitud n donde n es más que m.

o_i = Emparejamiento($T_{i+j,}$ P_j), donde $0 \leq i < n-m$ y $0 \leq j < m$

Cada **ocurrencia** es una emparejamiento dentro del texto del patrón. Una ocurrencia del patrón es el par formado por:

$o = (i = <0, m>, T_{sub} = <b_0, \ldots, b_m>)$

Tal que:

$T = T_{pref}T_{sub}T_{suf}$ donde $|T_{pref}|, |T_{suf}| \geq 0$ $|T_{sub}| > 1$ $T_{pref}, T_{suf} \in \Sigma*$

Cada ocurrencia está compuesta por el intervalo formado por la posición inicial y final donde ocurre la coincidencia dentro del texto, y por la propia coincidencia.

El **número de ocurrencias** (no_k) teóricas de orden k en el texto T, es la cardinalidad media en términos estadísticos del conjunto de ocurrencias de orden k.

Las variantes del problema de emparejamiento de cadenas son: estructuras de texto, compresión de datos, problemas de aproximación, búsqueda de regularidades, extensión a dos dimensiones, extensión a arboles e implementaciones óptimas. (Crochemore & Ryller, 1994)

A continuación se describe el trabajo de los algoritmos de emparejamiento exacto:

✔ Primero se toma una región del texto denominada **ventana (window)**, esta región generalmente lineal es de tamaño m (longitud del patrón).

✔ Luego se alinea la ventana al texto para comparar o emparejar los caracteres de la ventana al patrón, esta tarea especifica se denomina **intento (attempt)**.

✔ Después de un emparejamiento completo o de un fallo se **cambia (shift)** de ventana hacia la derecha, también puede interpretarse como salto cuando el desplazamiento es mayor a uno.

✔ Se repite los procedimientos anteriores hasta alcanzar el final del texto, a la repetición se la denomina **mecanismo de desplazamiento**.

Por el modo que se realiza las comparaciones en la ventana, se puede distinguir cuatro categorías:

✔ De izquierda a derecha: es el más natural, como se acostumbra a realizar una lectura.

✔ De derecha a izquierda: es el que conduce en la práctica al mejor algoritmo.

✔ En un orden específico: con este se alcanza el límite teórico.

✔ Sin orden.

2.4.1 Reconocimiento de patrones

Para describir patrones se desarrolla un “lenguaje” que permite especificar los problemas de búsqueda en cadenas. Además se necesita tres operaciones básicas: concatenación, unión y clausura. (Sedgewick, 1995)

Concatenación: si dos caracteres son adyacentes en el patrón, entonces hay concordancia si y sólo si los dos caracteres son adyacentes en el texto. Por ejemplo, GC significa G seguido de C.

Unión: Si se tiene un or entre dos caracteres, hay concordancia si y sólo si si uno de los caracteres figura en el texto. Por ejemplo G+T significa G o T.

Clausura: Esta operación permite que algunas partes del patrón se pueda repetir arbitrariamente. La clausura se representará poniendo un * después del carácter o grupo entre paréntesis. Por ejemplo, AT* concuerda con las cadenas que consisten en una A seguida de cualquier número de T.

En sistemas reales se hacen varias adiciones:

✔ Excepto (-): todos los caracteres diferentes a la letra precedida.

✔ Cualquiera(?): concuerda con cualquier letra.

Para tratar expresiones regulares es necesario considerar una máquina abstracta que determina totalmente el próximo carácter a pesar de la clausura.

Cuando se parte de un estado inicial, la máquina debe ser capaz de reconocer cualquier cadena descrita por el patrón leyendo caracteres y cambiando de estado de acuerdo a las reglas, hasta llegar a un estado final.

En Unix una de los comandos más útiles es grep, acrónimo de "general regular expression print", su notación para especificar patrones es conocida como expresiones

regulares. Otras herramientas basadas en expresiones regulares son sed y el lenguaje de programación awk (Aho, Weinberger y Kernighan).

Las expresiones regulares fueron estudiadas por primera vez por Kleene (1956), para describir los acontecimientos que se podían representar por el modelo de autómata finito de actividad nerviosa (red neuronal) de McCulloch y Pitts (1943).

2.4.2 Autómatas finitos

Los autómatas finitos se pueden considerar como modelos simplificados de máquinas abstractas y como mecanismos utilizados para especificar lenguajes formales. Como máquinas, su único recuerdo se compone de un conjunto finito de estados. (Crochemore & Ryller , 1994)

Formalmente, un autómata (determinista) G es una secuencia (A, Q, q_0, δ, T), donde A es un alfabeto de entrada, Q es un conjunto finito de estados, q_0 es el estado inicial (un elemento de Q) y δ es la función de transición. El valor δ (q, a) es el estado alcanzado desde el estado q por la transición etiquetada por la entrada símbolo a. La función de transición se extiende de forma natural a todas las palabras y, para la palabra x, δ (q, x) denota, si existe, el estado alcanzado después de leer la palabra x en el autómata de la estado q. El conjunto T es el conjunto de estados de aceptación o estados terminales del autómata.

El autómata G acepta el lenguaje: L (G) = {x: δ (q_0, x) pertenece a T}. El tamaño de G, denotado por tamaño (G) es el número de todas las transiciones de G: cantidad de todos los pares (q,a) (q es un estado, a es un símbolo) para el cual se define δ (q, a).

El formalismo de las expresiones regulares es equivalente al de los autómatas finitos. Este último se utiliza para el reconocimiento de patrones específicos. El algoritmo de emparejamiento de patrones consta de dos fases: primero, la expresión regular es transformado en un autómata equivalente. En segundo lugar, la búsqueda sobre el texto se realiza con la ayuda del autómata.

2.4.3 Algoritmo KMP (Knuth, Morris y Pratt)

El algoritmo KMP de emparejamiento exacto de una cadena T de tamaño **n** y un patrón P de tamaño **m**, utiliza una **ventana** de tamaño m para buscar todas las coincidencias sin realizar **retrocesos** en el desplazamiento por la cadena T, en caso de no coincidir (**por un falso principio**), se **salta un número de posiciones** hacia la derecha, los s_i forman un arreglo S o **tabla de saltos** para cada carácter del patrón P que se obtienen en el momento previo a la búsqueda. (Sedgewick, 1995)(Kouzinopoulos, Michailidis & Margaritis, 2015)

La tabla de saltos es calculada usando la misma técnica descrita en el anterior párrafo, realizando un desplazamiento de la copia del mismo patrón. El algoritmo KMP alcanza un tiempo lineal **O(n)** al recordar que su característica principal es desplazarse sin retroceso. El algoritmo KMP diseñado para un patrón en particular puede ser comparado a un **autómata** finito determinista que acepta la entrada patrón. (Nusret, Uzun y Doruk, 2017)(Navarro y Raffinot, 2000)

El algoritmo KMP tanto en su versión secuencial necesita construir la tabla de saltos NEXT tomando como entrada el patrón P como se muestra en la figura 2.3:

```
Construir_tabla_de_saltos(p)
    x= 1; y = 0; NEXT[1] = 0
    while not at end of pattern
        if  px ≠py
         then
              y := NEXT[y] ;
              x++; y++;
        if px = py then
         then
              NEXT[x] = NEXT[y] ;
         else
              NEXT[x] = y;

    endwhile
```

Figura 2.3: Construcción de la tabla de Saltos del algoritmo KMP Fuente: (Aoe Jun-ichi, 1994)

Una vez construida la tabla de saltos NEXT, continua con la fase de emparejamiento o búsqueda, tomando como entrada el patrón P y la cadena T como se muestra en la figura 2.4:

```
Emparejamiento_KMP(p, t)
        j = 0; k = 0 ;
        while not at end of pattern or string
                if  pj ≠tk
                then
                        j ++;k++;
                else
                        j = NEXT[j]
                        if j=0
                            k++;
        endwhile
```

Figura 2.4: Emparejamiento con el algoritmo KMP
*Fuente: (*Aoe Jun-ichi, 1994*)*

2.4.4 Algoritmo RK (Rabin y Karp)

El algoritmo RK de emparejamiento exacto de cadenas utiliza una **gran memoria** tratando una serie de m caracteres del texto como si fuera una **clave** (un número identificador) de una **tabla de dispersión** estándar (o tabla hash). Se debe calcular la clave en función de los m caracteres para cada una de las series de los m caracteres del texto y verificar si es igual a la función de dispersión del patrón. (Sedgewick, 1995)(Aoe, 1994)

El algoritmo RK en su versión secuencial utiliza una cadena o texto T, un patrón P, una función hash y un número primo, para reportar todas las posiciones donde se producen las coincidencias de P en T como se muestra en la figura 2.5. Donde, m = |P| y n = |T|:

```
Emparejamiento RK(P, T)
    emparejamientos = {}
    hash_patron = hash(P)
    hash_subcadena = hash(T[0 : m])
    for i from 0 to n - m - 1 do
        actualizar hash_subcadena  to hash(T[i : i + m])
        if hash_subcadena = hash_patron then
                adicionar i to emparejamientos
        end
    end
end
```

Figura 2.5: Emparejamientos con el algoritmo RK
Fuente: (Sharma y Singh, 2015)

Rabin y Karp encontraron la **función hash** *h(k)=k mod q*, donde *q* es un **número primo**. Para no almacenar una tabla de dispersión, en el desplazamiento por el texto se utiliza la ecuación de actualizar. (Sedgewick, 1995)

Para resolver la ecuación de **actualizar** el hash de la subcadena para la posición *i se* necesita conocer el valor hash de la posición *i-1*. La ecuación transforma los m caracteres en números agrupándolos en una palabra de lenguaje máquina, que se trata como un entero. Esto equivale a escribir los caracteres como **números en un sistema de base *d***, donde *d* es el número de caracteres posibles, es decir el tamaño del alfabeto. El número que corresponde a *a…[i]…a[i+m-1]* es entonces:

$$x = a[i]d^{m-1} + a[i+1]d^{m-2} + \ldots + a[i+m-1]$$

Aoe (1994) ejemplifica la ecuación anterior, tomando el numero 345 en base 10. Esto es equivalente a escribir:

$$x = 3x10^{2} + 4x10^{1} + 5$$

Aoe (1994) también muestra que si el número primo **q** no es muy grande, pueden ocurrir colisiones. Si el hash del patrón iguala al hash de la subcadena, entonces hay que verificar carácter por carácter para asegurar el emparejamiento.

La ventaja de este algoritmo es que puede ser extendido a búsquedas en dos dimensiones como las imágenes. (Aoe, 1994)

Nunes, Bordim, Ito y Nakano (2018) muestran como puede extenderse el algoritmo RK a búsquedas de un conjunto de claves o patrones, para posteriormente realizar una versión paralela e implementada con el framework CUDA.

Sharma y Singh (2015) realizaron la captura de los paquetes del tráfico de una red con la ayuda de software open source Wireshark (alternativa de Tcpdump) y generar un archivo que es enviado al programa que utilizaron el emparejamiento por Rabin-Karp, donde los patrones fueron generados aleatoriamente.

2.4.5 Algoritmo BM (Boyre y Moore)

El algoritmo BM es el más conocido y utilizado por su alta eficiencia en búsquedas de cadenas, una versión de este algoritmo con frecuencia es implementada en editores de texto para los comandos de "búsqueda" y "reemplazo". (Yañez, 2005)

El algoritmo utiliza tres conceptos claves, **recorrer el patrón de derecha a izquierda**, para el desplazamiento por ventanas se utiliza las funciones de la regla del **buen sufijo** y regla del **carácter nulo**.

La idea es decidir lo que se debe hacer en función del carácter que provocó la discordancia tanto en el texto como en el patrón. La etapa del preprocesamiento consiste en construir una tabla de saltos, para cada carácter que podría figurar en el texto, lo que se haría si dicho carácter hubiese provocado el fallo, en ese caso se mueve el puntero tan lejos como sea posible. Si hubiese más de un carácter en el patrón, se utiliza para el cálculo el que este más a la derecha, de aquí la tabla de saltos se construye explorando de izquierda a derecha. (Sedgewick, 1995)

2.4.6 Algoritmo Shift-Or

El algoritmo Shift Or usa la técnica orientada a bits que resuelven el problema de la búsqueda exacta en texto muy eficientemente para tamaños de alfabeto relativamente pequeños, el alfabeto binario por ejemplo. (Yañez, 2005)

Es un algoritmo semi-numérico desarrollado por Yates y Gonnet (1992) basado en el uso de comparaciones numéricas en vez de la comparación de caracteres.

Sea una cadena y un patrón, se crea un arreglo de dos dimensiones para almacenar la información de ocurrencia/fallo pues cada carácter en el patrón se compara con cada carácter en la cadena. Una ocurrencia se asigna con un valor de 1 y 0 para un fallo.

El método de SO tiene una ventaja sobre la comparación de caracteres, porque las comparaciones de bits se realizan más rápidamente por los procesadores, sin embargo, ya que

el tamaño del arreglo es del tamaño del patrón por el tamaño de la cadena, el método es relativamente útil sólo para cadenas pequeñas.

La eficiencia más alta sería lograda si el número de caracteres en la cadena es menor al número de bits en una palabra de computadora. Así, si una palabra típica de computadora es de 32 bits, entonces un procesador podría acomodar una cadena de 31 letras en una sola operación, eliminando la necesidad de realizar ciclos repetidamente a través de una cadena.

2.4.7 Algoritmo Skip Search

El algoritmo Skip Search utiliza un contenedor de todas las posiciones de un carácter en la cadena. La fase de preprocesamiento del algoritmo Skip Search consiste en calcular los contenedores para todos los caracteres del alfabeto.

En la fase de búsqueda el principal bucle realiza una revisión en cada m-ésimo carácter del patrón P[j]. Para P[j], utiliza cada posición en el contenedor z[P[j]] para obtener un posible inicio en la posición p de T en P. Realiza una comparación de T con P comenzando en la posición p, carácter a carácter, hasta que haya un fallo, o hasta que haya un emparejamiento completo.

2.5 EMPAREJAMIENTO DE UN CONJUNTO DE PATRONES CON UNA CADENA

El problema de emparejamiento de cadenas de **un conjunto de patrones** se define como: dado una cadena T de tamaño n y un **conjunto** P={P_0, P_1,...P_k | k pertenece al conjunto de enteros positivos}, la tarea es encontrar todas las **ocurrencias de los patrones** en la cadena de entrada. (Kouzinopoulos, Michailidis & Margaritis, 2015)

Una solución de fuerza bruta es se separar los patrones y realizar la búsqueda con algoritmos de emparejamiento con solo un patrón.

Con un refinamiento se construyen **algoritmos de dos fases**. Existen algoritmos prácticos y eficientes como: Aho-Corasick, Ser Horspool, Set Backward Oracle Matching, Wu-Manber y SOG.

El **algoritmo AC (Aho y Corasick)** es capaz de identificar múltiples patrones en un texto en un solo paso (sin retroceso). Con el algoritmo se construye una **máquina de estados de emparejamiento** en base al conjunto de patrones para posteriormente desplazarse por el texto y encontrar todas las ocurrencias, invocando tres funciones: goto, failure y output. El algoritmo AC es aplicado a la seguridad en redes y bioinformática, y otros campos. (Bellekens, Tachtatzis, Atkinson, Renfrew y Kirkham, 2014) (Tran, Lee, Hong y Shin, 2015)

Existen dos formas de implementar el algoritmo AC: mediante un autómata finito determinista (AFD) y un autómata finito no-determinista (AFND). (Liu, Li & Sun, 2016)

El algoritmo AC necesita construir la máquina de estados, a continuación se muestra el pseudo-código del algoritmo en la figura 2.6:

```
Emparejamiento AFD_AC(X, n)
    state = 0
    for (int i = 0; i < n; i++)
        state = δ(state, x[i]);
        if (output(state) != empty)
            print i
            print output(state)
        end
    end
end
```

Figura 2.6: Emparejamientos con el algoritmo AFD_AC
Fuente: (Tran, Lee, Hong & Shin, 2015)

2.6 EMPAREJAMIENTO APROXIMADO DE CADENAS

La secuencia genética de dos miembros de una misma especie no son idénticas, pero sin son **similares**. Esto conduce a una búsqueda que permita k errores, es decir una **distancia** entre dos cadenas. Esos errores son muy comunes en secuencias genéticas. La edición de

distancia entre las cadenas P y T es definida como la **mínima edición de operaciones** de transformar uno en otro. (Navarro Gonzalo, 1998)

Navarro define el problema de emparejamiento aproximado de textos como: dado un patrón corto P de longitud m, y un texto largo T de longitud n, y un **número máximo k de errores**; encontrar todas las posiciones j donde los sufijos de T emparejan con P, con un máximo de k errores por inserción, eliminación y reemplazo.

En biología molecular la comparación de varias secuencias se realiza normalmente escribiendo una sobre otra. El resultado se conoce como **alineación** del conjunto de secuencias. Esto puede dar una idea sobre la evolución de las secuencias a través de la mutaciones, inserciones y eliminaciones de nucleótidos. (Crochemore & Ryller, 1994)

La alineación de dos secuencias es el problema de edición de distancia, se realiza mediante algoritmos basados en técnicas de programación dinámica similares al utilizado por el comando diff de Unix. Una herramienta, denominada agrep, desarrollada en la Universidad de Arizona se dedica a problemas relacionados con la coincidencia aproximada de cadenas. El problema de la subsecuencia común más larga es una variación del alineamiento de secuencias.

2.6.1 Programación dinámica

También llamada planificación dinámica es una técnica de programación que también permite resolver problemas mediante una secuencia de decisiones, par lograr el objetivo se necesita producir varias secuencias de decisiones. Solamente al final se sabe cuál es la mejor de todas. (Galve, Gonzalez, Sanchez & Velázquez,1993)

No es fácil establecer una definición de la programación dinámica; una característica es que el programa “aprende” dinámicamente de las decisiones que toma. Además, todo problema resoluble con esta técnica debe satisfacer el principio de optimalidad. Este principio establece que una secuencia óptima de decisiones que resuelve un problema debe cumplir la propiedad de que cualquier subsecuencia de decisiones también debe ser óptima respecto al subproblema que resuelva.

En la programación dinámica todos los subproblemas se resuelven de acuerdo a un criterio de tamaño creciente y los resultados de los subproblemas más pequeños se almacenan en algún tipo de estructura de datos, normalmente tablas, para facilitar la solución de los problemas más grandes.

CAPÍTULO III

MARCO APLICATIVO

3.1 TECNOLOGÍAS DE SECUENCIACIÓN DE ADN

La secuenciación de ADN es un proceso que permite obtener el orden de cada uno de los nucleótidos que conforman la molécula de ADN. Para la secuenciación del genoma humano se utilizó una máquina de primera de generación que fue el ABI PRISM 3700 (figura 3.1) que realizó 500 corridas ininterrumpidas de 2.5 horas para obtener un total de 550 bases por reacción de secuenciación. La obtención del genoma humano significo un costó de unos 3000 millones de dólares y que llevó más de una década en terminarse.

Figura 3.1: ABI PRISM 3700
Fuente 0126,
https://unabiologaenlacocina.files.wordpress.com/2017/01/3abiprism3100.jpg?w=300&h=238

Una vez logrado el secuenciamiento del genoma humano, queda por secuenciar el genoma de otras especies como el tulipán, que puede llevar hasta 30 años, debido a que el genoma de la planta es enorme, unas 10 veces más que el de un ser humano.

De la necesidad de bajar costos y acelerar el tiempo en la secuenciación surgen las tecnologías de nueva generación (NGS, por sus siglas en inglés, Next Generation Sequencing), que permiten el procesamiento masivo de muestras, a continuación se muestra un secuenciador de última generación (figura 3.2).

Figura 3.2: Secuenciador de última generación
Fuente
https://img.medicalexpo.es/images_me/photo-pc/83632-15064083.webp

La "mejor" tecnología para la secuenciación de genomas, y la "mejor" metodología para el ensamblaje de secuencias están en dependencia de las características del microorganismo en cuestión, y del objetivo final de la investigación. (Aguilar-Bultet, Lisandra, & Falquet, Laurent, 2015)

Las máquinas de secuenciación producen millones de cadenas de fragmentos de nucleótidos de entre 35 y 1100 que son cortos en comparación de los genomas. Una manera de reconstruir el genoma a partir de millones de lecturas es mediante el proceso de alineación, el cuál consiste en ubicar cada lectura corta con la referencia de un genoma secuenciado previamente, en este proceso aparecen inserciones, supresiones o mutaciones de nucleótidos,

así como los errores de las máquinas de secuenciación. (Pacheco-Bautista, D., Martínez-Oviedo, J., Carreño-Aguilera, R., Algredo-Badillo, I., & Sánchez-Sánchez, S.., 2019).

3.2 ESTRATEGIAS COMPUTACIONALES DE ENSAMBLE DE SECUENCIAS DE ADN

La estrategia más utilizada se conoce con el nombre de secuenciación Shotgun, En esta técnica la cadena de ADN a secuenciar se clona a través del uso de PCR o mediante una bacteria anfitrión, el número de copias que se obtienen se conoce como cobertura. Posteriormente la muestra resultante se divide aleatoriamente en pequeños fragmentos en forma desordenada mediante alguna de las tecnologías, la división aleatoria crea fragmentos de diferente tamaño por lo que en este paso es necesario elegir únicamente las que se encuentran en un rango apropiado para la tecnología de secuenciación a utilizar. Una vez obtenidas las lecturas, los traslapes entre estas se utilizan para reconstruir mediante técnicas computacionales (grafos), a este último paso se le conoce como ensamble de fragmentos, la figura 3.3 muestra el esquema Shotgun.

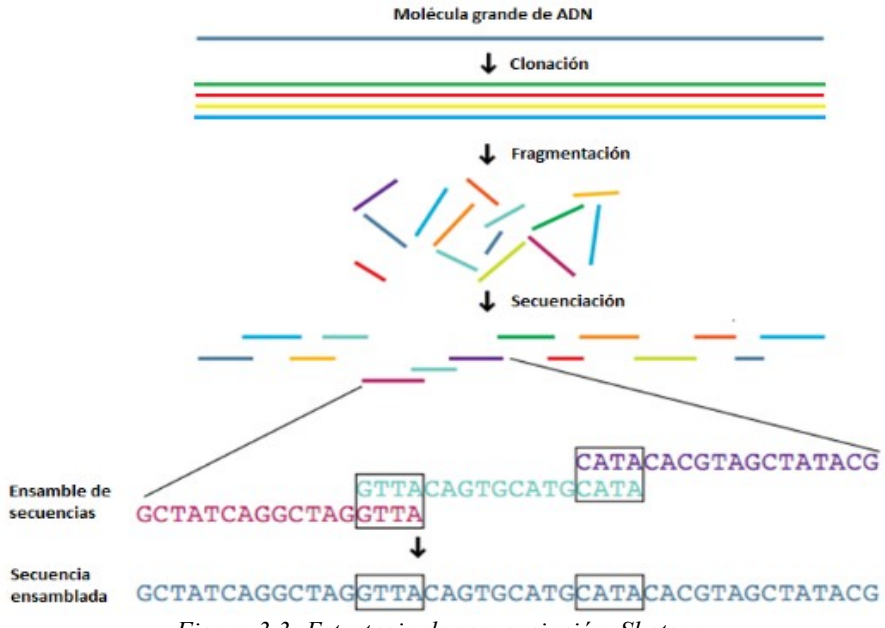

Figura 3.3: Estrategia de secuenciación Shotgun
Fuente (Pacheco Bautista, D., González Pérez, M., & Algredo Badillo, I., 2015)

Otra estrategia para la secuenciación se denomina siembra y extiende, que combina búsquedas exactas y aproximadas para la alineación de secuencias. Durante la etapa de siembra se intenta encontrar una subcadena de la lectura corta (semilla) que se empareje exactamente en uno o más lugares del genoma de referencia. (Pacheco-Bautista, D., Martínez-Oviedo, J., Carreño-Aguilera, R., Algredo-Badillo, I., & Sánchez-Sánchez, S., 2019).

Para la etapa de extensión se intenta extender la semilla en ambas direcciones, en esta etapa toda la lectura corta es alineada respecto a la cadena de referencia en las zonas encontradas durante la siembra. La extensión permite determinar de forma precisa la existencia de mutaciones, inserciones o supresiones en la lectura respecto a la referencia. A continuación se muestra la figura 3.4 de siembra y extiende.

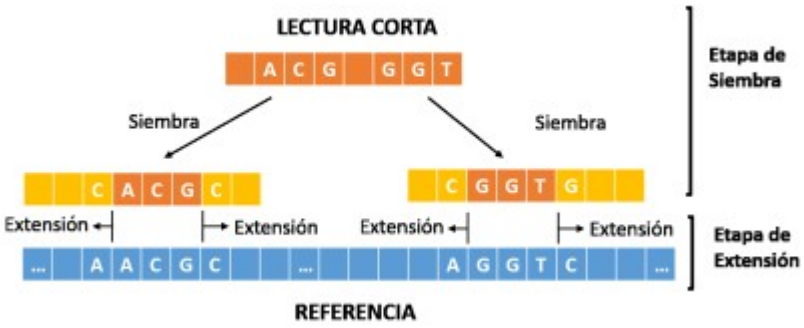

Figura 3.4: Estrategia de siembra y extiende
Fuente (Pacheco-Bautista, D., Martínez-Oviedo, J., Carreño-Aguilera, R., Algredo-Badillo, I., & Sánchez-Sánchez, S., 2019)

Para la etapa de siembra, dichos programas realizan un pre-procesamiento del genoma, obteniendo índices de búsqueda eficientes mediante los algoritmos basados en Tablas Hash, o en la Transformada de Burrows-Wheeler. Posteriormente, para la extensión implementan algoritmos basados en programación dinámica, tales como el Smith-Waterman, o el Needleman-Wunsch.

3.3 ARCHIVOS CON SECUENCIAS DE ADN

Las distintas bases de datos de secuencias de ADN almacenan los archivos con sus propios formatos, existen herramientas informáticas para convertir formatos.

Los archivos de formato FASTA pueden obtenerse de la base de datos del sitio web de National Center for Biotechnology Information (NCBI), estos archivos tienen atributos o propiedades como el nombre, la procedencia e información que describen a la secuencia.

```
Id: NC_045512.2
Name: NC_045512.2
Description: NC_045512.2
Severe acute respiratory
syndrome coronavirus 2
isolate Wuhan-Hu-1, complete
genome
Annotations: {}
```

Figura 3.5: Propiedades de la secuencia extraída de la base de datos NCBI con ID="NC_045512" (Covid-19)

La información en la anterior figura 3.5 corresponde a la secuencia con ID="NC_045512".

```
ATTAAAGGTTTATACCTTCCCAGGTAAC
AAACCAACCAACTTTCGATCTCTTGTAG
ATCTGTTCTCTAAACGAACTTTAAAATC
TGTGTGGCTGTCACTCGGCTGCATGCTT
AGTGCACTCACGCAGTATAATTAATAAC
TAATTACTGTCGTTGACAGGACACGAGT
AACTCGTCTATCTTCTGCAGGCTGCTTA
CGGTTTCGTCCGTGTTGCAGCCGATCAT
CAGCACATCTAGGTTTCGTCCGGGTGTG
ACCGAAAGGTAAGATGGAGAGCCTTGTC
CCTGGTTTCAACGAGAAAACACACGTCC
AACTCAGTTTGCCTGTTTTACAGGTTCG
CGACGTGCTCGTACGTGGCTTTGGAGAC
TCCGTGGAGGAGGTCTTATCAGAGGCAC
GTCAACATCTTAAAGATGGCACTTGTGG
CTTAGTAGAAGTTGAAAAAGGCGTTTTG
CCTCAACTTGAACAGCCCTATGTGTTCA
TCAAACGTTCGGATGCTCGAACTG
```

```
Que trata de la condición y
ejercicio del famoso hidalgo
D. Quijote de la Mancha
En un lugar de la Mancha, de
cuyo nombre no quiero
acordarme, no ha mucho tiempo
que vivía un hidalgo de los
de lanza en astillero, adarga
antigua, rocín flaco y galgo
corredor. Una olla de algo
más vaca que carnero,
salpicón las más noches,
duelos y quebrantos los
sábados, lentejas los
viernes, algún palomino de
añadidura los domingos,
consumían las tres partes de
su hacienda.
```

Figura 3.6: Fragmento de la secuencia de ADN NC_045512 y fragmento del libro Don Quijote de la Mancha

Las secuencias de ADN son cadenas de series de caracteres largas que no pueden leerse o comprenderse a simple vista, en la figura 3.6 se muestran dos fragmentos, el fragmento de la derecha es más fácil de comprender porque tiene palabras del idioma español.

Para comprender o extraer información de una secuencia de ADN se utilizan herramientas de software que permiten el reconocimiento de patrones, análisis estadísticos, traducciones y otros.

3.4 BÚSQUEDA POR EMPAREJAMIENTO EXACTO CON EL ALGORITMO KNUTH, MORRIS Y PRATT

Un autómata finita explica como el algoritmo KMP encuentra una ocurrencia de una instancia en particular. A continuación se muestra el diseño de un autómata que acepta un entrada (patrón) P=”GATGATC”.

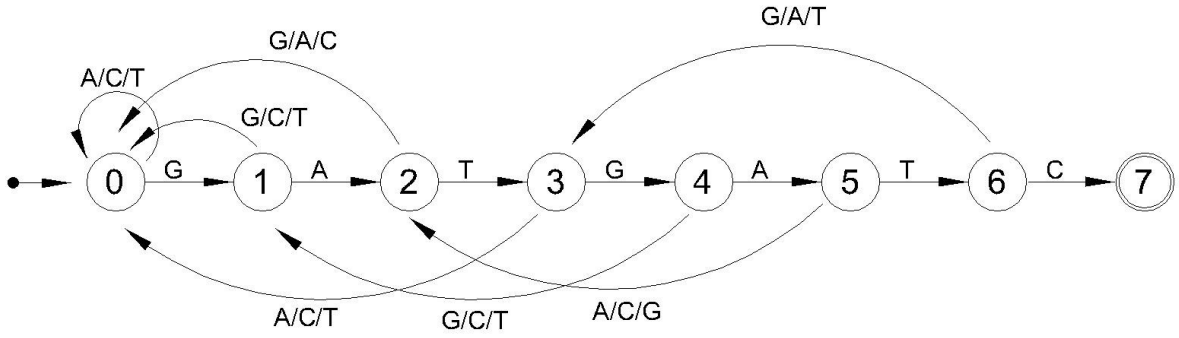

Figura 3.7: Diagrama de transición para encontrar la primera ocurrencia del patrón P=”GATGATC”

En el capítulo dos se presento el pseudo-código del algoritmo KMP de búsqueda de un patrón sobre cadenas y con la idea de construir un autómata finito de la figura 3.7, en esta sección se presenta la implementación de las funciones necesarias: construcción del vector de cambios o saltos por longitud del prefijo y búsqueda por emparejamiento KMP:

```
import re
import time
def KMPSearch(pat, txt):
    resultado=[]
    M = len(pat)
    N = len(txt)
    lps = [0]*M
    j = 0 # indice para pat[]

    computeLPSArray(pat, M, lps)

    i = 0 # indice para txt[]
    while i < N:
       if pat[j] == txt[i]:
            i += 1
            j += 1
       if j == M:
            resultado.append(str(i-j))
            j = lps[j-1]
       elif i < N and pat[j] != txt[i]:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
    return resultado

def computeLPSArray(pat, M, lps):
    len = 0
    lps[0] # lps[0] es siempre 0
    i = 1
    while i < M:
        if pat[i]== pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:
            # Caso engañoso. Consideremos el ejemplo.
            # AAACAAAA y i = 7. La idea es similar
            # a buscar paso a paso(fuerza bruta)
            if len != 0:
                len = lps[len-1]
                # Notemos que la variable i no incrementa aqui
            else:
                lps[i] = 0
                i += 1

txt=''
for _ in range(100):
```

```
    f=open('datos','r')
    for i in f:
        txt+=i.strip()
    f.close()
#print(len(txt))
pat='TTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA'
inicio=time.time_ns()
print('KMP encontro',len(KMPSearch(pat, txt)))
print('tiempo en ns',(time.time_ns()-inicio))
inicio=time.time_ns()
resultado=[]
x=0
while True:
    x=txt.find(pat,x+1)
    if x==-1:
        break
    resultado.append(x)
print('Find Encontro en',len(resultado))
print('tiempo en ns',(time.time_ns()-inicio))
inicio=time.time_ns()
a=re.compile(pat)
b=a.finditer(txt)
#b=a.findall(txt)
resultado=[]
for i in b:
    resultado.append(i.start())
print('RE encontro en',len(resultado))
print('tiempo en ns',(time.time_ns()-inicio))
```

Programa 3.1: Implementación en Python 3.8.6 del algoritmo KMP

A continuación se muestra un ejemplo de ejecución de la implementación del programa 3.1.

```
KMP encontro 100
tiempo en ns 1332959921
Find encontro 100
tiempo en ns 25652840
RE encontro 100
tiempo en ns 31230463
```

Figura 3.8: Ejemplo de ejecución de la implementación del algoritmo KMP mostrando un análisis experimental del tiempo

El algoritmo KMP se comporta como un autómata para el emparejamiento de una instancia de un patrón, de la teoría de lenguajes regulares se conoce que un autómata tiene su equivalencia con una expresión regular, entonces, se puede comparar su ejecución con la búsqueda de un expresión regular. En la figura 3.8 la ejecución muestra una comparación experimental con la librería RE (expresiones regulares) de Python para buscar patrones mediante expresiones regulares, con los tres métodos se verifica el resultado del número de ocurrencias es correcto.

En el anterior programa el arreglo de longitudes de cambio en la posición inicial siempre comienza en 0, Knuth acelero este proceso realizando el cambio en la posición inicial en -1. El programa 3.2 que se obtiene se muestra a continuación.

```
def preKmp(x,m,kmpNext):
    i = 0
    j = -1
    kmpNext[0] = -1
    while (i < m-1):
        while (j > -1  and x[i] != x[j]):
            j = kmpNext[j]
        i+=1
        j+=1
        if (x[i] == x[j]):
            kmpNext[i] = kmpNext[j]
        else:
            kmpNext[i] = j

def KMP(x,y): # x=Pat, y=Cad
    resultado=[]
    n=len(y)
    m=len(x)
    kmpNext=[0]*m
    # Preprocesamiento
    preKmp(x, m, kmpNext)
    #print(kmpNext)
    # Búsqueda
    i = j = 0
    while (j < n):
        while (i > -1 and x[i] != y[j]):
            i = kmpNext[i]
        i+=1
        j+=1
        if (i == m):
```

```
            #OUTPUT(j - i)
 #print("ocurrencia en "+str(j-i))
            resultado.append(str(j-i))
            i = kmpNext[i-1]
      return resultado

 import re
 import time

 txt=''
 for _ in range(100):
     f=open('datos','r')
     for i in f:
         txt+=i.strip()
     f.close()
 #print(len(txt))

pat='TTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAAA
A
AAAAAAAAAAAAAAAAAAAAAAAAA'
 inicio=time.time_ns()
 print('KMP-m encontro',len(KMP(pat,m,txt,n)))
 print('tiempo en ns',(time.time_ns()-inicio))
 inicio=time.time_ns()
 resultado=[]
 x=0
 while True:
     x=txt.find(pat,x+1)
     if x==-1:
         break
     resultado.append(x)
 print('Find Encontro en',len(resultado))
 print('tiempo en ns',(time.time_ns()-inicio))
 inicio=time.time_ns()
 a=re.compile(pat)
 b=a.finditer(txt)
 #b=a.findall(txt)
 resultado=[]
 for i in b:
     resultado.append(i.start())
 print('RE encontro en',len(resultado))
 print('tiempo en ns',(time.time_ns()-inicio))
```

Programa 3.2: Implementación en Python 3.8.6 del algoritmo KMP modificado por Knuth

A continuación se muestra en la figura 3.9 un ejemplo de ejecución del programa 3.2.

```
KMP-m encontro  100
tiempo en ns 972790243
Find Encontro en 100
tiempo en ns 18263862
RE encontro en 100
tiempo en ns 16199968
```

Figura 3.9: Ejemplo de ejecución de la implementación del algoritmo KMP-modificado mostrando un análisis experimental del tiempo

El algoritmo Knuth-Morris-Pratt (y el modificado) es una variante o mejora del algoritmo Morris-Pratt. Otros algoritmos que son comparables con el algoritmo KMP son: algoritmo de Simon (basado en un automata), algoritmo Apostólico-Crochemore, algoritmo Colussi y el algoritmo Galil-Giancarlo.

3.5 BÚSQUEDA POR EMPAREJAMIENTO EXACTO CON EL ALGORITMO RABIN-KARP

Si el patrón tiene una longitud de 4, entonces el algoritmo necesita calcular el hash de las subcadenas (ventanas) de tamaño 4 como sugiere la figura 3.10.

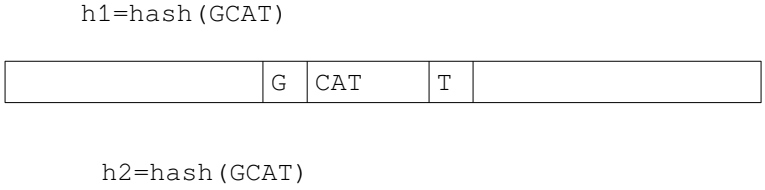

Figura 3.10: Ejemplo de agrupación de subcadenas para calcular el hash de cada subcadena

De la figura 3.10 y del capítulo dos donde se presento el pseudo-código del algoritmo RK de búsqueda de un patrón sobre cadenas, en esta sección se presenta su implementación en Python 3.8.6

```
def rabinKarp(cad,pat):
    n=len(cad)
    m=len(pat)
    resultado=[]
    contador=0
```

```
        if (n>0 and m>0):     # verificar si la cadena esta vacía
            c,p = 0,0         # c = hash de cadena, p = hash de pat
            pm = 4            # nro de caracteres del alfabeto
            q = 33554393      #101# numero primo
            h = 1             # h multiplicador
            for i in range(0,m-1):
                h = (h * pm) % q
            for i in range( 0,m):
                c = (pm * c + ord(cad[i])) % q
                p = (pm * p + ord(pat[i])) % q
            #print('hash %d' %p)
            for i in range (0,n-m+1):
                #print(f'hash en {i} es {c}')
                if (c == p):  # comparando hash de cadena y patrón
                    for j in range(0,m):
                        if(cad[i+j] != pat[j]):
                            break
                    if(j==m-1):
                        resultado.append(str(i))
                if i<n-m:
                    c=(pm*(c-h*ord(cad[i]))+ord(cad[i+m]))%q
                    if c<0:
                        c=c+q
        return resultado

    import time
    import re
     txt=''
     for _ in range(100):
         f=open('datos','r')
         for i in f:
             txt+=i.strip()
         f.close()
     #print(len(txt))

    pat='TTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAA'
     inicio=time.time_ns()
     print('Rabin Karp encontro ',len(rabinKarp(txt,pat)))
     print('tiempo en ns',(time.time_ns()-inicio))
inicio=time.time_ns()
     resultado=[]
     x=0
     while True:
         x=txt.find(pat,x+1)
         if x==-1:
             break
         resultado.append(x)
```

```
print('Find Encontro en',len(resultado))
print('tiempo en ns',(time.time_ns()-inicio))
inicio=time.time_ns()
a=re.compile(pat)
b=a.finditer(txt)
#b=a.findall(txt)
resultado=[]
for i in b:
    resultado.append(i.start())
print('RE encontro en',len(resultado))
print('tiempo en ns',(time.time_ns()-inicio))
```

Programa 3.3: Implementación en Python del algoritmo Rabin-Karp

A continuación en la figura 3.11 se muestra el ejemplo de ejecución de la implementación del algoritmo Karp Rabin, donde se imprime el número de ocurrencias y el tiempo experimental.

```
Rabin Karp encontro  100
tiempo en ns 2686248361
Find Encontro en 100
tiempo en ns 17479708
RE encontro en 100
tiempo en ns 16105069
```

Figura 3.11: Ejemplo de ejecución del algoritmo Rabin-Karp

3.6 BÚSQUEDA POR EMPAREJAMIENTO EXACTO CON EL ALGORITMO BOYRE-MOORE

El algoritmo BM su principal característica es recorrer la ventana de derecha a izquierda como fué descrito en el capítulo dos, a continuación se desarrolla un ejemplo en la figura 3.11.

Indices	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Secuencia	...	...	A	C	C	A	T	C	A	C	A	G	C	T	A	...	...

(1)		C	G	C	A	C

Figura 3.12: Ejemplo de saltos de caracteres del algoritmo Boyre-Moore

En la figura 3.12 se empieza en el estado (1), se salta completamente la longitud del patrón para llegar al estado (2) porque el carácter T no aparece en el patrón y para llegar al estado (3) se salta tres posiciones para alinear el carácter G.

Si existiera un emparejamiento entre el último carácter del patrón y el último carácter de la ventana del texto o secuencia, entonces los caracteres precedentes son comparados en sus correspondientes posiciones hasta que exista un fallo o un completo emparejamiento.

La obtención de un completo emparejamiento implica que la ocurrencia es en la posición inicial de la ventana del texto.

El número de desplazamiento ante un fallo depende de la tabla saltos.

A continuación se muestra la implementación del programa correspondiente al algoritmo Boyre-Moore.

```
ASIZE=256
def preBmBc(x,m,bmBc):
   vector=[m]*ASIZE
   for i in range( m - 1):
      vector[ord(x[i])] = m - i - 1
   return vector


def suffixes(x,m,suff):
   suff[m - 1] = m
   g = m - 1
   for i in range(m - 2,0, -1):
      if (i > g and suff[i + m - 1 - f] < i - g):
         suff[i] = suff[i + m - 1 - f]
      else:
         if (i < g):
            g = i
         f = i
         while (g >= 0 and x[g] == x[g + m - 1 - f]):
            g-=1
```

```
            suff[i] = f - g

def preBmGs(x,m,bmGs):
    suff=[0]*m #m=XSIZE
    suffixes(x, m, suff)
    for i in range(m):
        bmGs[i] = m
    j = 0

    for i in range(m - 1,0,-1):
        if (suff[i] == i + 1):
            for j in range(i, m - 1 - i, 1):
                if (bmGs[j] == m):
                    bmGs[j] = m - 1 - i
    for i in range(m - 1):
        bmGs[m - 1 - suff[i]] = m - 1 - i
    return bmGs

def BM(x, m, y, n):
    bmGs=[m]*m
    bmBc=[m]*ASIZE
    #Preprocesamiento
    bmGs=preBmGs(x, m, bmGs)
    print(bmGs)
    bmBc=preBmBc(x, m, bmBc)
    print(bmBc)
    # Buscando
    j = 0
    while (j <= n - m):
        i=m-1
        while(i >= 0 and x[i] == y[i + j]):
            i-=1
        if (i < 0):
 #OUTPUT(j)
            print("Ocurrencia en "+str(j))
            j += bmGs[0]
        else:
            j += max(bmGs[i], bmBc[ord(y[i + j])] - m + 1 + i)

import time
import re
 txt=''
 for _ in range(100):
     f=open('datos','r')
     for i in f:
         txt+=i.strip()
     f.close()
 #print(len(txt))
```

```
   pat='TTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAA
   AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
    inicio=time.time_ns()
    print('BM encontro ',len(BM(pat,txt)))
    print('tiempo en ns',(time.time_ns()-inicio))
inicio=time.time_ns()
    resultado=[]
    x=0
    while True:
        x=txt.find(pat,x+1)
        if x==-1:
            break
        resultado.append(x)
    print('Find Encontro en',len(resultado))
    print('tiempo en ns',(time.time_ns()-inicio))
    inicio=time.time_ns()
    a=re.compile(pat)
    b=a.finditer(txt)
    #b=a.findall(txt)
    resultado=[]
    for i in b:
        resultado.append(i.start())
    print('RE encontro en',len(resultado))
    print('tiempo en ns',(time.time_ns()-inicio))
```

Programa 3.4: Implementación en Python del algoritmo Boyre-Moore

A continuación en la figura 3.13 se muestra el ejemplo de ejecución de la implementación del programa 3.4 , donde se imprime el número de ocurrencias y el tiempo experimental o empírico.

```
BM encontro  100
tiempo en ns 90914260
Find Encontro en 100
tiempo en ns 18140918
RE encontro en 100
tiempo en ns 16095508
```

Figura 3.13: Ejemplo de ejecución del algoritmo Boyre-Moore

Algunas variantes del algoritmo Boyre-Moore son: algoritmo Turbo-BM, algoritmo Horspool, algoritmo Raita (variante Hoorspol), algoritmo Apostólico-Giancarlo, algoritmo

Reverse-Colussi (simplificación BM), algoritmo Quick-Search (simplificación BM), algoritmo Optimal Mismatch (variante de Quick-Search), algoritmo Maximal Shift (variante de Quick-Search), algoritmo Tuned Boyer-Moore, algoritmo Smith (variante Hoorspol y Quick Search), algoritmo Backward Oracle Matching, algoritmo Zhu-Takoaka y el algoritmo Berry-Ravindran.

Otros algoritmos comparables con el algoritmo Boyre-Moore por utilizar sufijos autómatas son: algoritmo Reverse-Factor y el algoritmo Turbo Reverse-Factor.

3.7 BÚSQUEDA POR EMPAREJAMIENTO EXACTO CON EL ALGORITMO SKIP-SEARCH

El algoritmo Skip-Search principalmente utiliza la estructura de datos lista. Con un patrón P de longitud m y un texto T de longitud n, para cada carácter c del alfabeto A, el algoritmo Skip-search realiza una fase de preprocesamiento formando una lista (bucket) con todas las posiciones del carácter en el patrón, para cada carácter del alfabeto como se muestra en la figura 3.14.

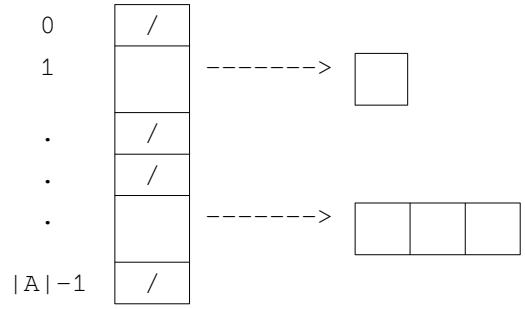

Figura 3.14: Esquema de listas por caracteres del alfabeto del algoritmo Skip-Search

Si el carácter ocurre k veces en el patrón, hay k posiciones almacenadas en la lista del carácter. Hay que notar que cuando el tamaño del patrón es mas corto que el tamaño del alfabeto algunas listas se quedan vacías.

En la fase de búsqueda el algoritmo Skip-search examina todos los caracteres T[j] en las posiciones j=km-1, para k=1,2,…,n/m. Para cada carácter T[j], la lista B[T[j]] permite el

calculo de las posiciones h del texto en la vecindad j en el cuál el patrón puede emparejar completamente.

A continuación se muestra la implementación del algoritmo Skip-Search en python:

```
ASIZE=256

def ArrayCmp(a,aIdx,b,bIdx,Length):
   i = 0
   while(i < Length and aIdx + i < len(a) and bIdx + i < len(b)):
      if (a[aIdx + i] != b[bIdx + i]):
         return 1
      i+=1
   if (i== Length):
      return 0
   else:
      return 1


def SKIP(x,y):
   resultado=[]
   m=len(x)
   n=len(y)
   z=[]
   for i in range(ASIZE):
      z.append([])
   # Preprocesamiento
   for i in range(m):
      z[ord(x[i])].append(i)

   #Búsqueda
   for i in range(m-1,n,m):
      ptr=z[ord(y[i])]
      for j in range(len(ptr)):
         if (ArrayCmp(x,0,y,i-ptr[j],m)==0):
            #print('Ocurrencia en %d\n' % (i-ptr[j]))
            resultado.append(i-ptr[j])
   return resultado

import time
import re
 txt=''
 for _ in range(100):
     f=open('datos','r')
     for i in f:
         txt+=i.strip()
     f.close()
```

```
    #print(len(txt))

   pat='TTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAAAAAAAAAA
   AAAAAAAAAAAAAAAAAAAAA'
    inicio=time.time_ns()
    print('Skip Search encontro ',len(SKIP(pat,txt)))
    print('tiempo en ns',(time.time_ns()-inicio))
inicio=time.time_ns()
    resultado=[]
    x=0
    while True:
        x=txt.find(pat,x+1)
        if x==-1:
            break
        resultado.append(x)
    print('Find Encontro en',len(resultado))
    print('tiempo en ns',(time.time_ns()-inicio))
    inicio=time.time_ns()
    a=re.compile(pat)
    b=a.finditer(txt)
    #b=a.findall(txt)
    resultado=[]
    for i in b:
        resultado.append(i.start())
    print('RE encontro en',len(resultado))
    print('tiempo en ns',(time.time_ns()-inicio))
```

Programa 3.5: Implementación en Python del algoritmo Skip-Search

A continuación en la figura 3.15 se muestra el ejemplo de ejecución del programa 3.5 (algoritmo Skip-Search), donde se imprime las ocurrencias y el tiempo empírico obtenido.

```
Skip Search encontro  100
tiempo en ns 864574965
Find Encontro en 100
tiempo en ns 17598525
RE encontro en 100
tiempo en ns 15913752
```

Figura 3.15: Ejemplo de ejecución del algoritmo Skip-Search

La variante principal del algoritmo Skip-search es el algoritmo Alpha Skip Search.

3.8 ALINEAMIENTO GLOBAL DE SECUENCIAS

El algoritmo de programación dinámica según los antecedentes de la literatura que permite obtener el mejor alineamiento global, sin tener que realizar todos los alineamientos posibles entre dos secuencias, es el algoritmo Needleman-Wunsch, que tienen tres fases:

Primera fase: con un recorrido hacia adelante se inicializa la matriz de coincidencias de las secuencias: M[i,0]=0 y M[0,j]=0

Segunda fase: con un recorrido hacia adelante se forma una matriz donde se hace coincidir las dos secuencias, una en la fila y la otra en la columna.

El camino se optimiza hacia delante mediante la siguiente función, llenando así la matriz con la puntuación que mayor valor le proporcione:

M[i-1,j-1]+S[i,i] emparejamiento/fallo (diagonal)
M[i,j] = MAX M[i,j-1]+w gap en la sec. X (filas)
M[i-1,j]+w gap en la sec. Y (columnas)

Donde: S[i,j]=2 si hay emparejamiento, S[i,j]=-1 si es fallo y w=-2 por penalización gap (hueco).

Tercera fase: con un recorrido hacia atrás (TraceBack) se recorre la matriz desde el extremo inferior derecho y se recorre siguiendo el camino marcado en la primera fase y se pueden obtener alineamientos alternativos A_1, A_2, …, A_k y se elige el MAX(A_k)

Con este algoritmo se construye una matriz de puntuación con el que se puede rastrear el alineamiento óptimo.

Como guía para la construcción de la matriz tomemos dos secuencias, X=”CTGAA” y Y=”TGA”.

	_	C	T	G	A	A
_	0	-2	-4	-6	-8	-10
T	-2	-1	0	-1	-2	-3
G	-4	-2	-1	1	0	-1
A	-6	-3	-2	0	2	3

En este ejemplo el alineamiento óptimo es el siguiente:

C	T	G	A	A	
_	T	G	_	A	
-1	+1	+1	-1	+1	total=+1

A continuación se muestra la implementación del algoritmo:

```
import numpy as np

import time

sequence_1 = "CTGAA"
sequence_2 = "TGA"

inicio=time.time_ns()

#Crear las matrices
main_matrix = np.zeros((len(sequence_1)+1,len(sequence_2)+1))
match_checker_matrix = np.zeros((len(sequence_1),len(sequence_2)))

# Definir la puntuación para emparejamiento,fallo y hueco(vacio)
match_reward = 1
mismatch_penalty = -1
gap_penalty = -2

#Llenar match checker matrix de acuerdo a emparejamiento o fallo
for i in range(len(sequence_1)):
    for j in range(len(sequence_2)):
        if sequence_1[i] == sequence_2[j]:
            match_checker_matrix[i][j]= match_reward
        else:
            match_checker_matrix[i][j]= mismatch_penalty
```

```
#Llenar la matriz usando el algoritmo Needleman_Wunsch
#PASO 1 : Iniciar
for i in range(len(sequence_1)+1):
    main_matrix[i][0] = i*gap_penalty
for j in range(len(sequence_2)+1):
    main_matrix[0][j] = j * gap_penalty

#PASO 2 : Llenar la matriz
for i in range(1,len(sequence_1)+1):
    for j in range(1,len(sequence_2)+1):
       main_matrix[i][j] = max(main_matrix[i-1][j-1]+match_checker_matrix[i-
 1][j-1],main_matrix[i-1][j]+gap_penalty,
 main_matrix[i][j-1]+ gap_penalty)

#print(main_matrix)

# PASO 3 : Traceback

aligned_1 = ""
aligned_2 = ""

ti = len(sequence_1)
tj = len(sequence_2)

while(ti >0 and tj > 0):

    if (ti >0 and tj > 0 and main_matrix[ti][tj] == main_matrix[ti-1][tj-1]+
match_checker_matrix[ti-1][tj-1]):

        aligned_1 = sequence_1[ti-1] + aligned_1
        aligned_2 = sequence_2[tj-1] + aligned_2

        ti = ti - 1
        tj = tj - 1

    elif(ti > 0 and main_matrix[ti][tj] == main_matrix[ti-1][tj] + gap_penalty):
        aligned_1 = sequence_1[ti-1] + aligned_1
        aligned_2 = "-" + aligned_2

        ti = ti -1
    else:
        aligned_1 = "-" + aligned_1
        aligned_2 = sequence_2[tj-1] + aligned_2

        tj = tj - 1
#test
print(aligned_1)
print(aligned_2)
print('Needleman- Wunsch tiempo en ns',(time.time_ns()-inicio))
```

```
from Bio import pairwise2
from Bio.Seq import Seq

seq1 = Seq(sequence_1)
seq2 = Seq(sequence_2)

inicio=time.time_ns()
alignments = pairwise2.align.globalxx(seq1, seq2)

for match in alignments:
    print(match)

print('Biopython tiempo en ns',(time.time_ns()-inicio))
```

Programa 3.6: Implementación en Python del algoritmo Needleman-Wunsch para alineamiento global de pares

A continuación se muestra la ejecución del programa 3.3 y se compara su resultado el módulo Biopython, para finalmente registrar sus rendimientos en la figura 3.16.

```
TGAA
TG-A
Needleman- Wunsch tiempo en ns 174936

Alignment(seqA='CTGAA', seqB='-TG-A',
score=3.0, start=0, end=5)
Alignment(seqA='CTGAA', seqB='-TGA-',
score=3.0, start=0, end=5)
Biopython tiempo en ns 588427
```

Figura 3.16: Ejemplo de ejecución de la implementación Needleman-Wunsch comparado con Biopython

3.9 PROGRAMA DEMOSTRATIVO PARA EMPAREJAMIENTO EXACTO DE SECUENCIAS

La ejecución del programa demostrativo se realizo sobre una máquina virtual con las siguientes características.

- ✔ Sistema Operativo: Ubuntu 20.10 (groovy)
- ✔ Procesadores: 2
- ✔ Memoria base: 2048 MB

✔ Python 3.8.6

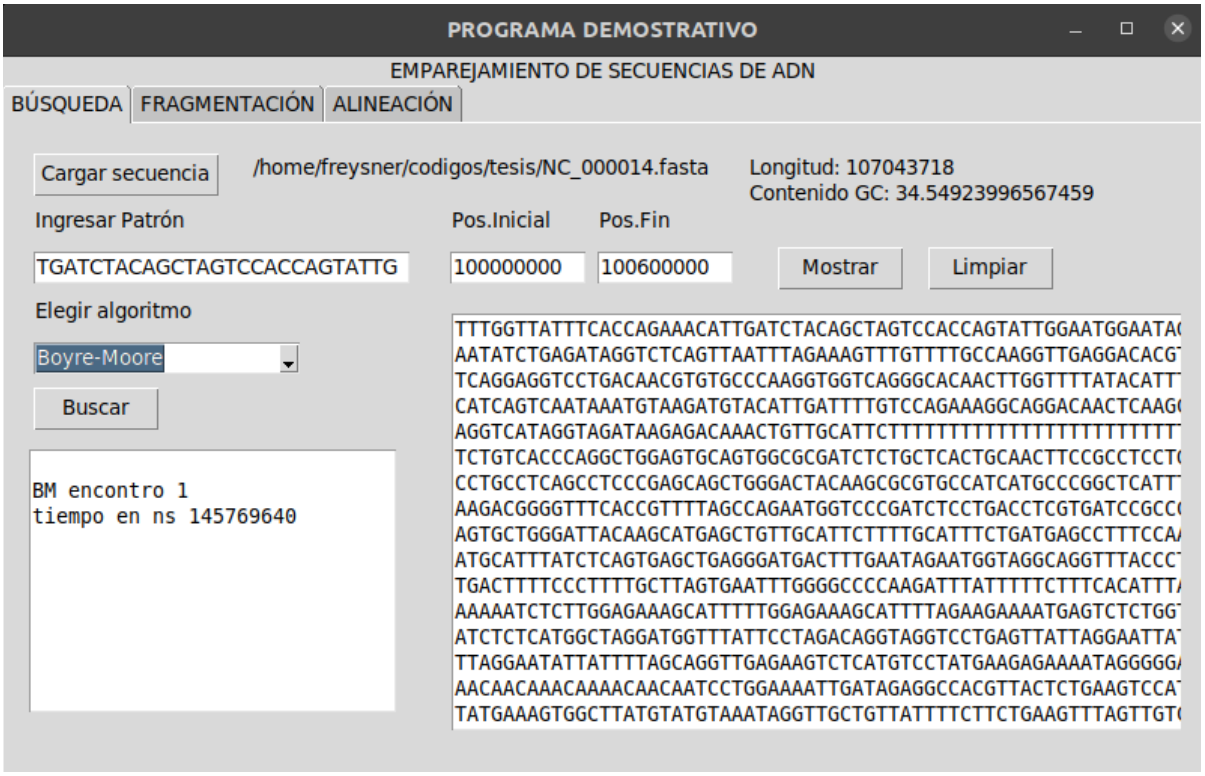

Figura 3.17: Interfaz gráfica para realizar búsquedas de patrones

La figura 3.17 muestra el entorno para realizar las búsquedas de patrones. En secuencias mayores a 600K es recomendable mostrar (de modo similar a un editor de texto) sólo una porción de la secuencia de nucleótidos, porque esta operación ocupa una gran porción de la memoria.

En la ejecución del programa demostrativo se utilizo el archivo fasta del cromosoma 14 del homo-sapiens, este archivo tiene un tamaño de 107MB, pero como se menciono anteriormente sólo se muestra una porción de nucleótidos. Sobre esa porción se realizo búsquedas de patrones con tamaño de 50 nucleótidos para registrar los rendimientos en la tabla 3.1.

Tabla 3.1: Registro de ejecución de las implementaciones de los algoritmos de emparejamiento exacto

PATRÓN de 50 nucleótidos	**Knuth-Morris-Pratt (ns)**	**Karp-Rabin (ns)**	**Boyre-Moore (ns)**	**Skip-Search (ns)**
TTTGGTTATTTCACCAGAAACATTGATCTACAGCTAGTCCACCAGTATTG	137301467	321449374	75828269	122198699
AATATCTGAGATAGGTCTCAGTTAATTTAGAAAGTTTGTTTTGCCAAGGT	285498378	329752644	84056280	122175130
TCAGGAGGTCCTGACAACGTGTGCCCAAGGTGGTCAGGGCACAACTTGGT	169893786	328806062	94835738	98709787
CATCAGTCAATAAATGTAAGATGTACATTGATTTTGTCCAGAAAGGCAGG	170711205	338131784	73735318	118952231
AGGTCATAGGTAGATAAGAGACAAACTGTTGCATTCTTTTTTTTTTTTTT	197692593	351351940	56499216	125929053
TCTGTCACCCAGGCTGGAGTGCAGTGGCGCGATCTCTGCTCACTGCAACT	177220417	335456517	47511629	153234374
CCTGCCTCAGCCTCCCGAGCAGCTGGGACTACAAGCGCGTGCCATCATGC	167702152	353879561	144226656	134612492
AAGACGGGGTTTCACCGTTTTAGCCAGAATGGTCCCGATCTCCTGACCTC	179171263	337272142	151647424	120187580
AGTGCTGGGATTACAAG	189773615	352319297	95282354	117044444

CATGAGCTGTTGCATTCT TTTGCATTTCTGATG				
ATGCATTTATCTCAGTGA GCTGAGGGATGACTTTG AATAGAATGGTAGGC	218402394	350595482	81865952	129666429

De la tabla 3.1 se puede realizar el siguiente análisis para formar la tabla 3.2:

Tabla 3.2: Análisis de los registros de ejecución de las implementaciones de los algoritmos de emparejamiento exacto

ALGORITMO	Media (ns)	Mediana	Desviación Estándar
Knuth-Morris-Pratt	189336727	178195840	39878552
Karp-Rabin	339901480	337701963	11505815
Boyre-Moore	90548884	82961116	33802765
Skip-Search	124271022	122186915	13892174

Los algoritmos de emparejamiento exacto utilizan diferentes estrategias para encontrar todas las ocurrencias de un patrón en las secuencias, como:

- ✔ Knuth-Morris-Pratt de izquierda a derecha.
- ✔ Karp-Rabin por función hash.
- ✔ Boyre-Moore de derecha a izquierda.
- ✔ Skip-Search por listas-contenedores.

Pero los algoritmos con mayor relevancia son Knuth-Morris-Pratt y Boyre-Moore por que la revisión de estos algoritmos permitieron el desarrollo de muchas variantes como se mencionó en anteriores párrafos de este capítulo.

El programa demostrativo con la implementación de los algoritmos, permitió la comparación de los diferentes algoritmos en sus rendimientos experimentales. Experimentalmente el algoritmo Boyre-Moore es el de mejor rendimiento y Rabin-Karp es el de menor desviación estándar como muestra la tabla 3.2.

3.10 PROGRAMA DEMOSTRATIVO PARA ALINEAMIENTO DE SECUENCIAS

Cuando los patrones a buscar sufren cambios por adición, eliminación entonces es necesario realizar búsquedas aproximadas, un modo de generalizar las búsqueda aproximadas es realizar los alineamientos, a continuación se muestra el programa demostrativo en la figura 3.18 para alineamientos.

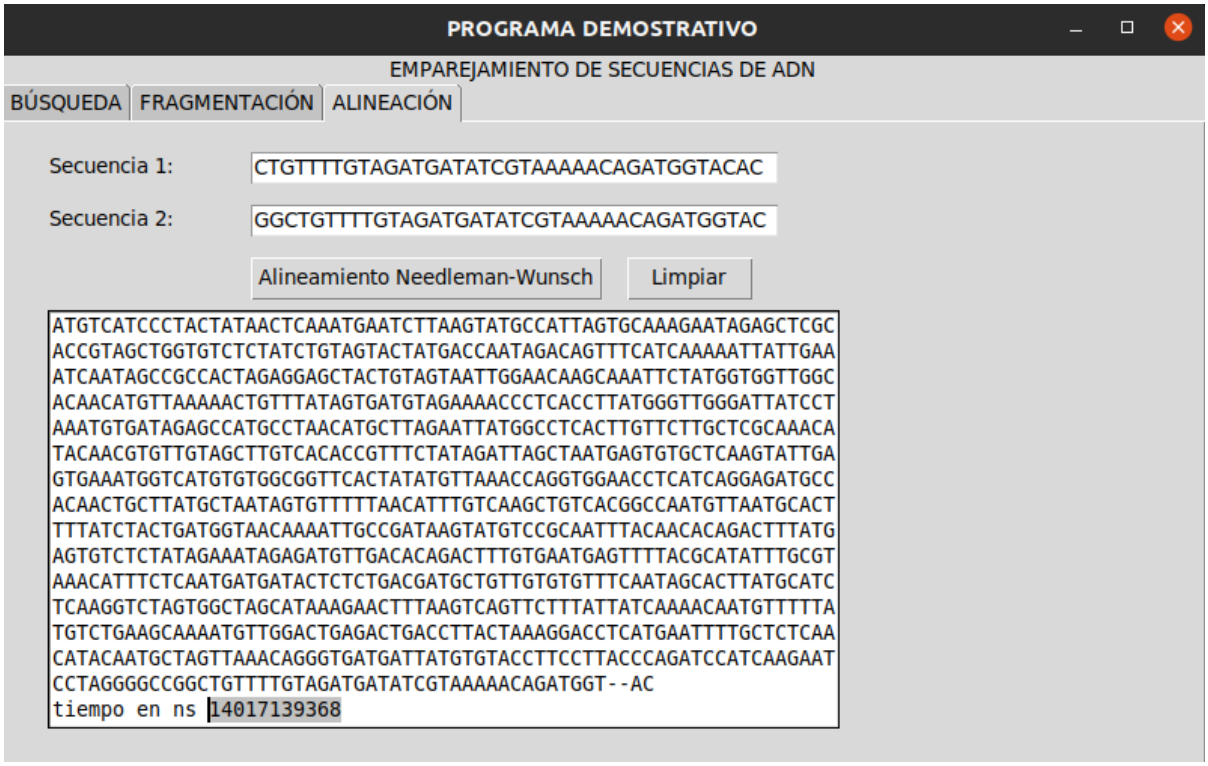

Figura 3.18: Interfaz gráfica para realizar alineamientos

El algoritmo Needleman-Wunsch tiene una complejidad $O(n_1*n_2)$ por tanto experimentalmente para secuencias de similar longitud se comporta como una función cuadrática, en las pruebas de tamaño 500 se ejecuto en 0,5 [seg], para tamaño 1000 en 3 [seg] y para un tamaño 2000 paso a 14 [seg].

CAPÍTULO IV

CONCLUSIONES Y RECOMENDACIONES

4.1 CONCLUSIONES

- ✔ Los algoritmos de emparejamiento exacto muestran varias estrategias para encontrar todas las ocurrencias de un patrón en las secuencias, como: Knuth-Morris-Pratt de izquierda a derecha, Boyre-Moore de derecha a izquierda, Karp-Rabin por función hash y Skip-Search por listas-contenedores. Pero los algoritmos con mayor relevancia son Knuth-Morris-Pratt y Boyre-Moore, la revisión de estos algoritmos permitió observar muchas variantes.
- ✔ El algoritmo Knuth-Morris-Pratt es comparable con un autómata finito, de lo anterior sus búsquedas pueden ser realizadas mediante expresiones regulares en cualquier lenguaje de programación.
- ✔ Experimentalmente el algoritmo Boyre-Moore es el de mejor rendimiento y Rabin-Karp es de menor desviación estándar. En implementaciones para secuenciadores se utiliza el algoritmo Rabin-Karp con una estrategia de búsqueda off-line (con indexación).
- ✔ El algoritmo Skip-Search alcanza un buen rendimiento haciendo uso de una estructura de datos lista-contenedor.

- ✔ Los algoritmos de emparejamiento exacto implementados en el lenguaje interpretado en Python tienen menor rendimiento que los reportados implementados en lenguajes compilados como C o C++.
- ✔ Al construir un programa demostrativo para englobar a los diferentes algoritmos que se desarrollo se experimento que en secuencias mayores a 600K (en un entorno específico mostrado en el capítulo III) es recomendable mostrar (de modo similar a un editor de texto) sólo una porción de la secuencia de nucleótidos, porque esta operación ocupa una gran porción de la memoria.
- ✔ El algoritmo de alineamiento global Needleman-Wunsch de complejidad cuadrática, experimentalmente (en un entorno específico mostrado en el capítulo III) se alcanzo a probar el alineamiento de secuencias menores de 2000 nucleótidos, secuencias más grandes necesitan más recursos computacionales.

4.2 RECOMENDACIONES

- ✔ Investigar las diferentes estrategias de los algoritmos de ensamblado de secuencias.
- ✔ Desarrollar implementaciones de procesamiento paralelo de distintos fragmentos.
- ✔ Realizar alineamientos locales o globales con diferentes estrategias en la construcción de la matriz de puntuación.
- ✔ Construir un modelo de una máquina de secuenciación.
- ✔ Investigar con una perspectiva biológica casos de estudio de inferencia filogenética, inferencia de funciones en estructuras de proteínas y/o genética de poblaciones.

BIBLIOGRAFÍA

Aguilar-Bultet, Lisandra, & Falquet, Laurent. (2015). *Secuenciación y ensamblaje de novo de genomas bacterianos: una alternativa para el estudio de nuevos patógenos*. Revista de Salud Animal, *37*(2), 125-132. Recuperado en 15 de agosto de 2021, de http://scielo.sld.cu/scielo.php?script=sci_arttext&pid=S0253-570X2015000200008&lng=es&tlng=es.

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman (1990). *Compiladores: principios, técnicas y herramientas* Pearson.

Aoe Jun-ichi, (1994). *Computer Algorithms: string pattern matching strategies.* Los Alamitos: IEEE Computer Society Press.

Bellekens, Tachtatzis, Atkinson, Renfrew & Kirkham (2014). *GLoP: Enabling Massively Parallel Incident Response Through GPU Log Processing* University of Strathclyde, Glasglow.

Brassard G. & Bratley P., (1997). *Fundamentos de Algoritmia (Edición en español)*. Madrid: Pearson Educación.

Charras C. & Lecroq T., (2004). Handbook of Exact String-Matching Algorithms. College Publications.

Crochemore M. & Ryller W., (1994). *Text Algorithms*. Oxford University Press.

Fernández (2017). *Árbol de prefijos para la búsqueda aproximada en lenguas originarias indígenas campesinas*, Tesis de Grado, Universidad Mayor de San Andrés, carrera de Informática, La Paz, Bolivia

Flores (2011). *Modelo de identificación de individuos mediante el perfil genético del ADN aplicando algoritmos genéticos*, Tesis de Grado, Universidad Mayor de San Andrés, carrera de Informática, La Paz, Bolivia

Galve, Gonzalez, Sanchez & Velázquez (1993) *Algorítmica diseño y análisis de algoritmos funcionales e imperativos* RA-MA, Madrid, España.

Gusfield D., (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge: University of Cambridge.

Hernández R., Fernández C. & Baptista P. (2010). *Metodología de la Investigación*. México: Mc Graw Hill.

Keng Leng Hui, (2006). *Approximate String Matching With Dynamic Programming and Suffix Trees.* Florida: University of North Florida.

Kouzinopoulos, Michailidis & Margaritis (2015). *Multiple string matching on a GPU using CUDA* University of Macedonia, Grecia.

Laura (2016). *Diseño de algoritmo de búsqueda aproximada de lenguas originarias campesinas*, Tesis de Grado, Universidad Mayor de San Andrés, Carrera de Informática, La Paz, Bolivia

Liu Jiahui & Xu Dongliang (2012). *An Efficient Parallel String Matching Algorithm Based on DFA.* Harbin, China.

Liu, Li & Sun (2016). *Parallel Algorithm of Multiple String Matching Based on Set-Partition in Multi-core Architecture.* International Journal of Security and Its Applications China

Meneses, Rozo & Franco (2011). *Tecnologías bioinformáticas para el análisis de secuencias de ADN* . Universidad Tecnológica de Pereira, Colombia.

Navarro Gonzalo (1998). *Approximate Text Searching.* Universidad de Chile, Santiago, Chile.

Navarro & Raffinot (2000). *Fast and Flexible String Matching by Combining Bit-parallelism and Suffix Automata* ACM Journal of Experimental Algorithmics. New York, United States.

Nunes, L.S., Bordim, J., Ito, Y., & Nakano, K. (2018). *Parallel Rabin-Karp Algorithm Implementation on GPU (preliminary version)*. Bulletin of Networking, Computing, Systems, and Software

Nusret, Uzun & Doruk (2017). *Comparison of string matching algorithms in web documents* Unitech, Gabrovo.

Pacheco Bautista, D., González Pérez, M., & Algredo Badillo, I.. (2015). *De la Secuenciación a la Aceleración Hardware de los Programas de Alineación de ADN, una Revisión Integral*. Revista mexicana de ingeniería biomédica, 36(3), 259-277. *https://doi.org/10.17488/RMIB.36.3.6*

Pacheco-Bautista, D., Martínez-Oviedo, J., Carreño-Aguilera, R., Algredo-Badillo, I., & Sánchez-Sánchez, S.. (2019). *ABPSE: Alineador de ADN Basado en Paralelismo a Nivel de Bit y la Estrategia Siembra y Extiende*. Revista mexicana de ingeniería biomédica, 40(1), e201821. *https://doi.org/10.17488/rmib.40.1.4*

Sedgewick R., (1995). *Algoritmos en C++*. Madrid: Addison-Wesley.

Sharma & Singh (2015). *CUDA based Rabin-Karp Pattern Matching for Deep Packet Inspection on a Multicore GPU.* University Patiala, India.

Solari (2004). *Genética Humana: Fundamentos y aplicaciones en Medicina.* Buenos Aires, Argentina.

Sun & Ma (2009). *Parallel Lexicographic Names Construction with CUDA.* Shenyang, China.

Tran, Lee, Hong & Shin (2012). *Memory Efficient Parallelization for Aho-Corasick Algorithm on A GPU* Myongji University, Korea.

Tran, Lee, Hong & Shin (2015). *Cache Locality-Centric Parallel String Matching on Many-Core Accelarator Chips* Myongji University, Korea.

Yañez Bernal J. F. (2005). *Comparación y clasificación de algoritmos de búsqueda en texto*, Tesis de Grado, Universidad Mayor de San Andrés, Carrera de Informática, La Paz, Bolivia.

La Paz, 6 de septiembre del 2021

Señor:
PhD José María Tapia Baltazar
DIRECTOR DE LA CARRERA DE INFORMÁTICA
Universidad Mayor de San Andrés
Presente.

REF: AVAL PARA DEFENSA DE TESIS DE GRADO

De mi consideración:

Mediante la presente, me dirijo a su Autoridad, en calidad de **Tutor Metodológico** para informar que luego de haber realizado el seguimiento de la estructura y contenido de la Tesis de Grado titulado **"ALGORITMOS DE EMPAREJAMIENTO DE SECUENCIAS DE ADN"**, elaborado por el Universitario **Freysner Noel Chambi Mendieta** con cédula de identidad **4780415 LP**, presento mi **conformidad y aval** respectivo para que el postulante pueda realizar la defensa de la Tesis de Grado, para optar al título de LICENCIADO EN INFORMÁTICA MENCIÓN: INGENIERÍA DE SISTEMAS INFORMÁTICOS, de acuerdo a normas y reglamento vigentes.

Sin otro particular, me suscribo con las atenciones más distinguidas.

M.Sc. Jorge Humberto Terán Pomier
Tutor Metodológico