REPORTE DIVIDE Y VENCERÁS

LONGEST COMMON SUBSEQUENCE

Centro Universitario de Ciencias Exactas e Ingenierías
Departamento: División de Tecnologías para la Integración Ciber-Humana
Materia: Análisis de algoritmos
Profesor: Jorge Ernesto López Arce Delgado
Alumnos: De la Mora Villaseñor Diego Gabriel 220576197
Lopez Esparza Angel Emanuel 223991772
López Galván Melanie Montserrat 220574046
Carrera: Ingeniería en Computación
Sección: D05
Fecha: 27/11/2025

Contenido

Introducción.....2
Objetivos.....3
Desarrollo.....3
Conclusiones.....9
Referencias.....11

Introducción

El ácido desoxirribonucleico (ADN) constituye la base fundamental de la estructura y funcionamiento de los seres vivos. Sus cadenas, compuestas por miles de millones de nucleótidos, albergan la información necesaria para comprender la diversidad biológica. En el campo de la bioinformática, el análisis de estas estructuras no se limita a la lectura de datos aislados, sino que exige la comparación exhaustiva entre secuencias para identificar similitudes, evoluciones y funcionalidades compartidas.

Para determinar el grado de similitud entre dos cadenas de ADN, el problema de la Subsecuencia Común Más Larga (LCS, por sus siglas en inglés) se ha establecido como un estándar métrico. Históricamente, este problema cobró relevancia en la década de 1970, cuando investigadores como Saul Needleman y Christian Wunsch publicaron uno de los primeros algoritmos de alineamiento global basados en programación dinámica. Su objetivo era superar las limitaciones de los enfoques intuitivos de fuerza bruta, los cuales, al generar todas las combinaciones posibles de subsecuencias, derivaban en una complejidad temporal exponencial O(2^n) intratable para secuencias biológicas reales.

Si bien la programación dinámica y las estrategias de "divide y vencerás" lograron optimizar el tiempo de ejecución al evitar el recálculo de subproblemas (memorización), introdujeron un nuevo desafío: el consumo de memoria. Algoritmos clásicos como Needleman-Wunsch requieren la construcción de matrices de dimensiones proporcionales a las longitudes de las cadenas (N * M). Dado que las secuencias genómicas modernas pueden alcanzar tamaños de gigabytes, el espacio en memoria se convierte en un cuello de botella crítico que impide el procesamiento en equipos convencionales.

Ante esta problemática de almacenamiento y procesamiento de "Big Data" biológico, surge la necesidad de integrar técnicas de compresión de datos en los algoritmos de comparación. El algoritmo de Huffman, conocido por su estrategia voraz para la codificación óptima basada en la frecuencia de caracteres, se presenta como una alternativa viable para reducir la redundancia de la información.

El presente documento propone el desarrollo de una aplicación para la comparación de cadenas de ADN que integre el método de compresión de Huffman con el algoritmo LCS.

Objetivos

- Desarrollar una herramienta de software para el análisis de secuencias de ADN que integre el algoritmo de compresión de Huffman para la solución del problema LCS, con la finalidad de optimizar el consumo de memoria operativa en comparación con los métodos tradicionales.

- Implementar el algoritmo de codificación de Huffman adaptado específicamente para el alfabeto de nucleótidos (A, C, G, T), generando estructuras de datos comprimidas a partir de archivos de secuencias (FASTA o texto plano).
- Desarrollar el algoritmo de Subsecuencia Común Más Larga (LCS) diseñado para procesar las cadenas codificadas, permitiendo el cálculo de similitud entre secuencias.
- Evaluar el rendimiento computacional de la solución propuesta midiendo dos variables críticas: el consumo de memoria RAM (espacial) y el tiempo de ejecución (temporal).
- Validar la exactitud de los resultados verificando que el porcentaje de similitud obtenido con las cadenas comprimidas sea idéntico al obtenido con las cadenas originales, asegurando la integridad de los datos biológicos.
- Diseñar una interfaz gráfica o de línea de comandos que permita la carga masiva de secuencias de ADN y la visualización de las métricas de compresión y similitud de manera clara para el usuario.

Desarrollo

El planteamiento del algoritmo parte de una premisa muy sencilla, la cual es el núcleo de todo el algoritmo, las subsecuencias de elementos. Estas forman parten de una secuencia original, donde se obtienen eliminando algún, o ninguno, de los elementos de la secuencia original; pero sin modificar el orden de estos elementos. No necesitan un orden consecutivo.

Tenemos un arreglo principal ‘A, B, C, D, E’ y a partir de este arreglo se pueden crear subsecuencias, por ejemplo ‘A, C, E’. Es una subsecuencia valida del arreglo principal, ya que los elementos mantienen el mismo orden, la A va primero, luego la C y por último la E. En caso contrario; si por ejemplo creáramos la subsecuencia ‘A, E, C’, entonces ya no seria válida, ya que la letra E se encuentra antes que la C y eso no es real en la cadena principal.

Lo que define si una subsecuencia es válida es cuando ninguno de los elementos antecede al orden de otro elemento en la cadena principal, sin importar si se eliminó cualquier otro elemento en medio de ellos.

Con la necesidad de la evolución del algoritmo nos vimos en la necesidad de optimizar el programa ya que hacer uso de la API de NCBI requería de conocer exactamente los ID´s de cada genoma a

comparar con lo que resultaba poco práctico utilizarlo. Para esto decidimos cambiarlo por la implementación de lectura de archivos FASTA, los cuales son especialmente específicos para nuestro programa ya que es un formato de texto plano para almacenar secuencias de ADN, ARN o proteínas. Para leer estos archivos usamos la librería biopython.

Razones para usar la librería:

- Biopython permite el tratamiento de archivos en diversos formatos utilizados en el ámbito biológico, presentando total compatibilidad con los archivos procedentes de GenBank, PDB, PubMed, ExPASy, etc.
- Permite trabajar con secuencias tanto nucleotídicas como aminoacídicas.
- Herramientas incorporadas. Herramientas para realizar operaciones comunes en secuencias, como traducción, transcripción, longitud de cadenas...
- Herramientas para realizar alineamientos de secuencias.
- Se puede usar la función SeqIO.read() para archivos con una sola secuencia y SeqIO.parse() para archivos con múltiples secuencias.

Todo lo anterior resuelve el conflicto de la dificultad elevada para el usuario que quiera usar el programa, sin embargo, nuestro programa contaba con otro problema que consistía en que al extraer las cadenas y compararlas el tiempo de ejecución era bastante elevado por lo que utilizar un algoritmo de compresión como el de Huffman resulto ser la respuesta a este problema. Dicho algoritmo consta de la compresión de datos sin pérdida que asigna códigos binarios de longitud variable a los símbolos de un mensaje, dando códigos más cortos a los símbolos más frecuentes, esto encaja de forma exacta con las cadenas de ADN a comparar ya que constan únicamente de 4 símbolos por base nitrogenada Adenina (A), Citosina (C), Guanina (G) y Timina (T), con esto sabemos que nuestro diccionario de compresión constara de pocos caracteres y será más sencillo de interpretar como conexiones en un árbol.

¿Cómo usamos la compresión?: Lo primero que hacemos es la unión de las cadenas de ADN a comparar para de esta forma saber que caracteres aparecen de forma total, con esta cadena unificada se crea un diccionario que asigna una cadena binaria a cada carácter. Con el diccionario ya creado se puede comprimir cada una de las cadenas a comparar de forma individual y posteriormente se buscan la subsecuencia de bits más larga entre las dos cadenas a comparar, esto va a depender

mucho de los especímenes que se comparen, ya que en el ejemplo que usamos no hay caracteres similares en las cadenas hasta el carácter 3’600,000 por lo que depende mucho de que exactamente desee ser comparado.

Implementación:

Huffman

Para la fase de compresión, se implementó una variante del algoritmo de Huffman que asegura la consistencia entre las dos cadenas a comparar.

El proceso consta de cuatro etapas:

1. Análisis de Frecuencia Global: Se concatenan ambas secuencias de ADN para realizar un conteo unificado de frecuencias. Esto garantiza que un nucleótido tenga el mismo código binario en ambas cadenas.
2. Construcción del Heap (Cola de Prioridad): Se utiliza una estructura de montículo (heapq) para ordenar los símbolos de menor a mayor frecuencia, lo que constituye la base de la estrategia voraz del algoritmo.
3. Generación del Árbol Binario: Se iteran extracciones de los dos nodos con menor probabilidad para combinarlos en un nuevo nodo padre, asignando '0' a las ramas izquierdas y '1' a las derechas, hasta que todos los nodos convergen en una raíz única.
4. Codificación y Decodificación: Finalmente, las cadenas de texto se transforman en listas de tokens binarios (encode_text), reduciendo el espacio en memoria pero manteniendo la integridad de la información para su decodificación (decode_tokens)".

Fuerza bruta

Como primera aproximación al problema, se implementó el algoritmo basado en una definición recursiva pura. La función evalúa las cadenas desde sus últimos caracteres:

- Caso Base: Si alguno de los índices llega a cero, la recursión se detiene.
- Coincidencia: Si los caracteres en m-1 y n-1 son iguales, se suma 1 al resultado y se realiza una llamada recursiva restando ambos índices (diagonal).

- Divergencia: Si los caracteres difieren, el algoritmo se bifurca en dos nuevas llamadas recursivas (m, n-1 y m-1, n), buscando el máximo valor entre ambas ramas.
- Esta implementación evidencia la naturaleza exponencial del problema, ya que recalcula los mismos subproblemas múltiples veces.

Divide y vencerás

Para la estrategia de Divide y Vencerás, se diseñó una solución híbrida que combina la segmentación recursiva con el procesamiento lineal mediante colas. La implementación se divide en dos componentes principales:

- División: Esta función aplica la lógica recursiva de dividir el problema. Toma la primera cadena (A) y la divide en mitades consecutivas hasta alcanzar un caso base donde la subcadena tiene una longitud de 4 caracteres o menos.
- Conquista (Función LCS con deque): Una vez que la cadena A es lo suficientemente pequeña, se compara contra la cadena B utilizando una estructura de cola (deque) para gestionar las comparaciones optimizando el uso de memoria respecto a la fuerza bruta.
- Combinación: Finalmente, los resultados de las subcadenas izquierda y derecha se suman para obtener la longitud total y se concatenan las subsecuencias encontradas.

Programación dinámica

La implementación final utiliza la técnica de tabulación (bottom-up) para resolver el problema. La función opera en dos fases:

Llenado de la Matriz: Se inicializa una matriz de dimensiones (m+1)*(n+1). Mediante un doble ciclo iterativo, se rellena la matriz almacenando en cada celda la longitud de la subsecuencia común hasta ese punto, utilizando los valores previamente calculados (celdas superior, izquierda o diagonal superior).

Reconstrucción: Una vez obtenido el valor máximo en la posición dp[m][n], se ejecuta un ciclo while inverso. Este recorre la matriz desde la última posición hacia el origen, siguiendo el camino de las decisiones óptimas para reconstruir la cadena de caracteres exacta que conforma la subsecuencia común más larga.

Resultados

En cuanto a complejidad algorítmica, este es un problema polinomial cuando se comparan solo 2 cadenas; su implementación con fuerza bruta tiene una complejidad de $O(2^{n+m})$, donde realiza todas las iteraciones del algoritmo. En la implementación de divide y vencerás este solo toma en cuenta el mínimo de N u M, donde su complejidad sería O(n*log(n)). La programación dinámica, demuestra una complejidad de O(n*m).

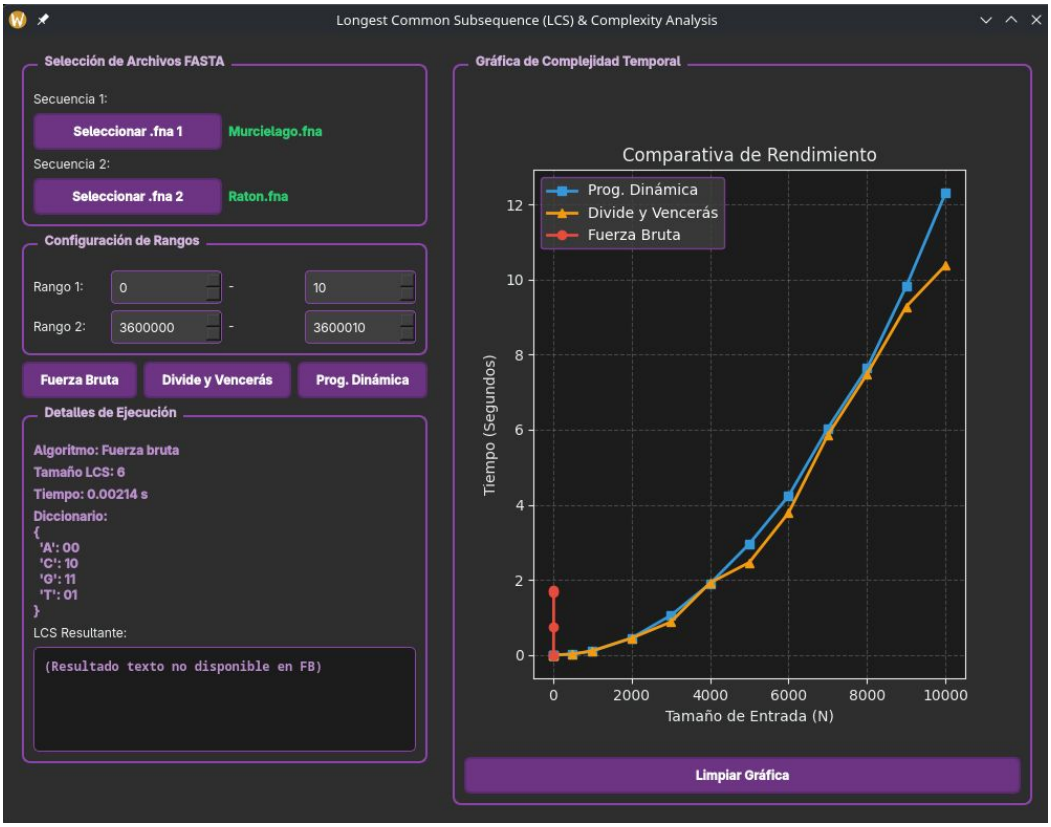

Figure 1: Grafica comparativa de la implementación de cada algoritmo

Conclusiones

En conclusión, la parte más difícil de todo este proyecto fue asimilar si quiera en que partes podríamos usar las técnicas algorítmicas deseadas y en que puntos sería hasta imposible. Por ejemplo, la técnica de divide y vencerás fue especialmente complicada ya que no fue posible integrarlo al proyecto sin hacer una mezcla extraña con programación dinámica, además el backtracking podría ser eliminado ya que las soluciones se pueden encontrar mucho antes de su uso como tal. Creo que lo único que conviene usar en este algoritmo LCS es la programación

dinámica, desde el simple hecho de que se puede representar de forma matricial y almacenar resultados aumentando el rango de comparación de cadenas sin perder tiempo de eficiencia para un mismo resultado ya que no es como que cambiar de técnica altere la cadena final.

Conclusión Melanie: La parte más difícil ha sido integrar todos los algoritmos, aunque considero que hacer una combinación entre LCS y el algoritmo de Huffman permite una comparación computacional más sencilla lo que podría mejorar el tiempo de resolución del algoritmo.

Conclusión Emanuel: Leer información acerca de distintas técnicas e implementaciones de los algoritmos es una practica que no pensaba que sería tan entretenida. Sinceramente no me esperaba que pudiéramos encontrar una implementación tan "reciente" para resolver este algoritmo, porque muchas veces tengo la idea de que los problemas ya tienen la mejor solución, me dio esperanza ver que fue en 2019 que se desarrolló la técnica de divide y vencerás.

Conclusión Diego: Yo tuve mucha confusión en cuando un algoritmo podía resolverse usando una técnica u otra, pero el propio problema te dicta como lo puedes resolver; cambiamos muchas veces los algoritmos para acoplarnos a las técnicas que se nos pedían y al final nos quedamos con un algoritmo interesante, que tenía formas curiosas de resolverse y tenía mucha complejidad en todos los sentidos. Fue un proyecto interesante de realizar y pude observar como distintas técnicas aplican mejor o peor.

Referencias

Wikipedia. (2025, October 26). *Longest common subsequence*. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/wiki/Longest_common_subsequence

R. Bhowmick, M. I. Sadek Bhuiyan, M. Sabir Hossain, M. K. Hossen and A. Sadee Tanim, "An Approach for Improving Complexity of Longest Common Subsequence Problems using Queue and Divide-and-Conquer Method" 2019 1st International Conference on Advances in Science, Engineering and Robotics Technology (ICASERT), Dhaka, Bangladesh, 2019, pp. 1-5, doi: 10.1109/ICASERT.2019.8934638.