BioSeqAligner: Optimización en Python del algoritmo de Needleman-Wunsch

Máster en Bioinformática y Ciencia de Datos en Medicina Personalizada de Precisión y Salud

Autores

Álvaro García Barragán
Pablo Fernández Lagos

Profesor

Daniel Báscones García

Noviembre, 2025

Índice

1. Introducción **1**

2. Fundamentos teóricos **1**
2.1. Conceptos básicos de alineamiento de secuencias . . . 1
2.2. Algoritmos clásicos de alineamiento . . . 2
2.3. Detalle del algoritmo de Needleman–Wunsch . . . 2

3. Implementación del algoritmo **3**
3.1. Algoritmo básico . . . 3
3.2. Algoritmo por diagonales . . . 3
3.3. Algoritmo con banda diagonal . . . 5
3.4. Algoritmo optimizado con Numba . . . 6

4. Evaluación de optimizaciones **6**
4.1. Criterios de comparación . . . 6
4.2. Características del sistema . . . 6
4.3. Benchmarks . . . 7
4.4. Resultados . . . 7
4.4.1. Tiempo de ejecución . . . 7
4.4.2. Uso de memoria . . . 8
4.4.3. Calidad del alineamiento . . . 9

5. Desarrollo de la aplicación **9**
5.1. Funcionalidades . . . 10
5.2. Arquitectura . . . 11

6. Conclusiones y trabajo futuro **12**

7. Accesibilidad **12**

1. Introducción

El alineamiento de secuencias constituye una de las herramientas fundamentales en bioinformática para el estudio de la información biológica contenida en el ADN, ARN y proteínas [1]. Su objetivo principal es identificar regiones de similitud que puedan reflejar relaciones funcionales, estructurales o evolutivas entre biomoléculas. Estas comparaciones permiten inferir homologías, predecir funciones de genes desconocidos, y comprender mecanismos de conservación a lo largo de distintas especies. Dentro de este contexto, el algoritmo de **Needleman–Wunsch** [2] resulta especialmente relevante, ya que proporciona un método sistemático para obtener el *alineamiento global óptimo* entre dos secuencias completas. Esta característica lo convierte en una herramienta adecuada para analizar secuencias de tamaño moderado donde se busca maximizar la correspondencia total.

Sin embargo, su complejidad computacional —proporcional al producto de las longitudes de las secuencias $\mathcal{O}(n \cdot m)$ — representa un desafío al aplicarlo a secuencias de gran tamaño. En este proyecto, se propone analizar la eficiencia del algoritmo de Needleman–Wunsch mediante implementaciones en `Python`, evaluando su rendimiento frente a diferentes longitudes de secuencias.

Este proyecto tiene dos objetivos principales:

1. Evaluar y optimizar el algoritmo, aplicando distintos niveles y técnicas de optimización para analizar su eficiencia.
2. Desarrollo de una interfaz intuitiva que permita visualizar resultados y obtener métricas básicas.

Este documento está organizado de la siguiente manera: la Sección 2 presenta los fundamentos teóricos del alineamiento de secuencias y la formalización del algoritmo de Needleman–Wunsch. La Sección 3 describe la implementación del algoritmo, desde el algoritmo básico hasta la versión con diagonales, el algoritmo con banda diagonal y su optimización usando Numba. A continuación, la Sección 4 detalla la evaluación de las optimizaciones. Finalmente, se describe el diseño de la interfaz en la Sección 5.

2. Fundamentos teóricos

2.1. Conceptos básicos de alineamiento de secuencias

El alineamiento de secuencias consiste en disponer dos o más secuencias biológicas (ADN, ARN o cadenas de aminoácidos) de manera que las regiones homólogas queden en la misma columna. Los elementos clave que definen un alineamiento son la definición de puntuaciones para coincidencias y desajustes, y la penalización por inserciones/borrados (gaps).

- **Tipos de alineamiento:**
  - *Global:* busca el mejor alineamiento que cubra la totalidad de ambas secuencias.
  - *Local:* encuentra la subcadena de mayor similitud entre las dos secuencias.
  - *Múltiple:* alinea tres o más secuencias simultáneamente, objetivo más complejo que requiere heurísticas o algoritmos especializados; p. ej: MUSCLE [3].
- **Matrices de sustitución:** describen la puntuación para emparejar pares de residuos (aminoácidos o bases) [4]. Ejemplos clásicos:
  - *PAM:* basadas en modelos evolutivos de sustitución a corto alcance (PAM_k).

- *BLOSUM:* derivadas de bloques de alineamientos conservados; p. ej. BLOSUM62 es de uso frecuente para proteínas.

Estas matrices proporcionan la función de similitud $s(a, b)$ usada en el calculo de puntuación del algoritmo.

- **Penalizaciones por huecos (gaps):** determinan el coste de insertar una alineación con huecos. Modelos habituales:
  - *Penalidad lineal:* cada gap de longitud k tiene coste $-d \times k$ (un solo parámetro d).
  - *Penalidad afín (gap open/extend):* coste $-(g + e \times (k - 1))$, donde g es el coste de apertura del gap, e el coste de extensión por cada posición adicional y k la longitud total del gap.

2.2. Algoritmos clásicos de alineamiento

- **Needleman–Wunsch (alineamiento global) [2]:** algoritmo de programación dinámica que encuentra el alineamiento global de coste máximo (o puntuación máxima) entre dos secuencias completas. Utiliza una matriz dinámica que se rellena por filas/columnas y luego realiza un *traceback* para obtener el alineamiento.
- **Smith–Waterman (alineamiento local) [5]:** variante de programación dinámica orientada a encontrar subsegmentos de alta similitud; la recurrencia incluye 0 como alternativa para permitir reiniciar el conteo cuando la puntuación se vuelve negativa.

2.3. Detalle del algoritmo de Needleman–Wunsch

Fundamentos matemáticos Sea $A = a_1 a_2 \dots a_n$ y $B = b_1 b_2 \dots b_m$ las dos secuencias. Definimos una función de similitud $s(a_i, b_j)$ y una penalización por gap d. La matriz de puntuación F de dimensiones $(n + 1) \times (m + 1)$ se inicializa como:

$$\begin{aligned} F(0,0) &= 0, \\ F(i,0) &= -i \cdot d \quad (1 \le i \le n), \\ F(0,j) &= -j \cdot d \quad (1 \le j \le m). \end{aligned}$$

La puntuación para $1 \le i \le n$, $1 \le j \le m$ es:

$$F(i,j) = \text{máx} \begin{cases} F(i-1, j-1) + s(a_i, b_j), \\ F(i-1, j) - d, \\ F(i, j-1) - d. \end{cases}$$

El valor $F(n, m)$ da la puntuación del mejor alineamiento global; el alineamiento se recupera mediante *traceback* desde (n, m) hacia $(0, 0)$, eligiendo en cada paso la opción que produjo el máximo.

Especificación formal El algoritmo se formaliza tal como se muestra en el Algoritmo 1. Sus etapas principales son las siguientes:

1. *Inicialización:* se asignan penalizaciones acumulativas por *gaps* en la primera fila y la primera columna.
2. *Relleno de la matriz:* cada celda se calcula mediante la el calculo de puntuación correspondiente con la función de similaridad s y el coste de *gap* (d).
3. *Traceback:* a partir de la celda (n, m) se reconstruye el camino que genera la puntuación óptima, recuperando el alineamiento (coincidencias, desajustes, inserciones o deleciones).

Algorithm 1 Needleman–Wunsch Alignment

Input: Sequences $A[1..n]$, $B[1..m]$, scoring function $s(a, b)$, gap penalty d
Output: Alignment score matrix F and optimal alignment
procedure NeedlemanWunsch(A, B, s, d)
    Initialize matrix F of size $(n+1) \times (m+1)$
    for $i = 0$ to n **do**
        $F[i, 0] \leftarrow -i \cdot d$
    end for
    for $j = 0$ to m **do**
        $F[0, j] \leftarrow -j \cdot d$
    end for
    for $i = 1$ to n **do**
        for $j = 1$ to m **do**
            $match \leftarrow F[i-1, j-1] + s(A[i], B[j])$
            $delete \leftarrow F[i-1, j] - d$
            $insert \leftarrow F[i, j-1] - d$
            $F[i, j] \leftarrow \max(match, delete, insert)$
        end for
    end for
    Traceback from (n, m) down to $(0, 0)$ to recover optimal alignment.
end procedure

Complejidad computacional La versión básica requiere tiempo $\mathcal{O}(n \cdot m)$ y espacio $\mathcal{O}(n \cdot m)$, donde n y m son las longitudes de las dos secuencias. Para secuencias largas esto puede ser inasumible.

3. Implementación del algoritmo

La implementación del algoritmo se realizó de manera progresiva: se comenzó con una implementación básica; se optimizó el relleno de la tabla de puntuaciones y se aplicó Numpy [6]; y, por último, se aplicó el paquete Numba [7].

3.1. Algoritmo básico

En la primera implementación, la matriz de puntuación se rellena de manera secuencial, fila a fila, siguiendo la estructura tradicional de un doble bucle *for* tal como se ha descrito en 1. El cálculo del máximo para la casilla de la matriz a rellenar se realiza también de manera no paralelizada. La Figura 1 muestra de manera visual el orden de generación.

Para ver el código de la implementación en Python consultar Sección 7.

3.2. Algoritmo por diagonales

Una de las principales limitaciones del relleno de la matriz es su dependencia espacial con las casillas que se encuentran tanto encima, a su izquierda, como finalmente en la diagonal superior izquierda. Esto obliga a que su relleno no pueda paralelizarse completamente. La única manera de paralelizar el cálculo es recorriendo de manera diagonal la matriz, en vez de manera secuencial por filas y columnas. Al recorrer de manera iterativa cada diagonal, todas las casillas pueden paralelizar su cómputo, dado que no hay dependencia entre ellas. Un ejemplo visual de como se recorre la matriz en diagonales se encuentra en la Figura 2.

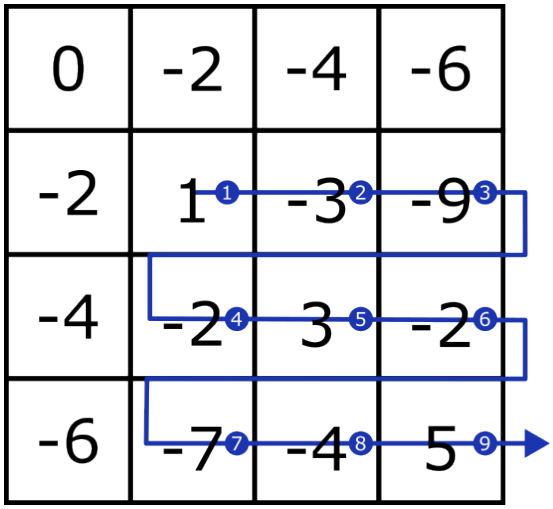

Figura 1: La imagen muestra la matriz de puntuación rellenada. La línea azul indica el orden de generación. Los círculos azules indican la iteración en la cual se asigna el valor.

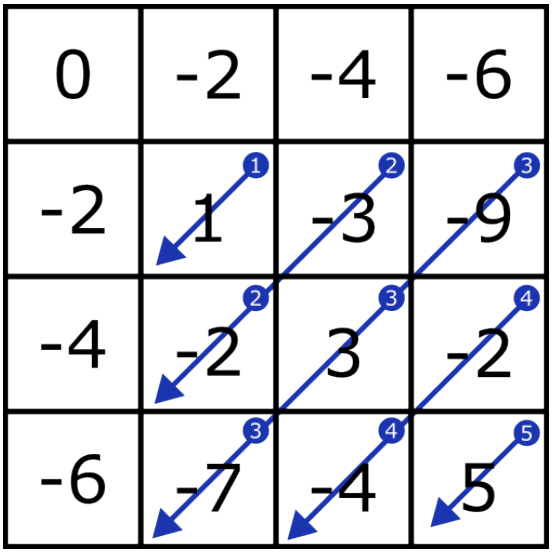

Figura 2: La imagen muestra la matriz de puntuación rellenada en base a diagonales. Las líneas azules indican las diagonales. Los círculos azules indican la iteración en la cual se asigna el valor.

Para obtener la reducción en cantidad de operaciones con el nuevo algoritmo, comparamos la cantidad de iteraciones entre el método secuencial y el método basado en diagonales. Podemos comprobar que, mientras uno es $N \times M$, el otro es $N + M - 1$, siendo N y M el tamaño de cada secuencia.

Para demostrar por qué la cantidad de iteraciones es $N + M - 1$ en el segundo caso, debemos entender como se generan las diagonales. Comenzaremos creando las diagonales desde la primera fila. Por cada casilla, obtenemos la diagonal siguiendo movimientos en el sentido de la diagonal inferior izquierda, hasta llegar a la última fila o alcanzar la primera columna. Una vez que hemos generado las diagonales para todas las casillas de la primera fila, recorremos la última columna siguiendo el mismo patrón hasta llegar al final, tal como se hace en la Figura 2. Si hiciésemos ambos procesos de manera independiente, dejando de lado que la matriz no se rellenara completamente, podríamos comprobar que, para el primer caso, se generan N diagonales y, para el segundo, M, coincidiendo ambas en una diagonal, tal como se muestra en la Figura 3. Es por ello que podemos concluir que la cantidad total de diagonales será $N + M - 1$.

Una vez realizada la implementación de la obtención de las diagonales, podemos paralelizar el cálculo del valor nuevo para cada casilla de la diagonal iterada. Utilizando Numpy, obtenemos primero los índices de la diagonal, realizamos el cálculo de cada movimiento de manera escalar y, finalmente, obtenemos el valor correspondiente del movimiento con valor máximo para cada casilla.

Para la obtención de los índices de cada diagonal tendremos que tener en cuenta: la iteración en la que nos encontramos, la cantidad de filas, la cantidad de columnas y, por último, la relación de índices en las diagonales. Empezando por la relación de índices, podemos comprobar que al bajar por la diagonal mientras que el índice de las columnas disminuye su valor en una unidad, el índice de las filas aumenta en una unidad. De manera que, si estamos en la tercera iteración de una comparación de 2 secuencias de tamaño 3, los índices resultantes serán (3,1), (2,2) y (1,3). Teniendo en cuenta todos los factores, el proceso de obtención de índices es el siguiente:

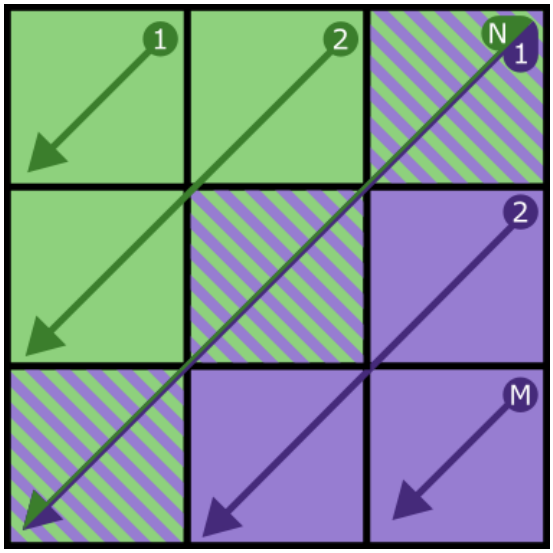

Figura 3: La imagen muestra la generación de diagonales desde la primera fila (en verde) y desde la última columna (en morado). Ambas generaciones comparten una diagonal.

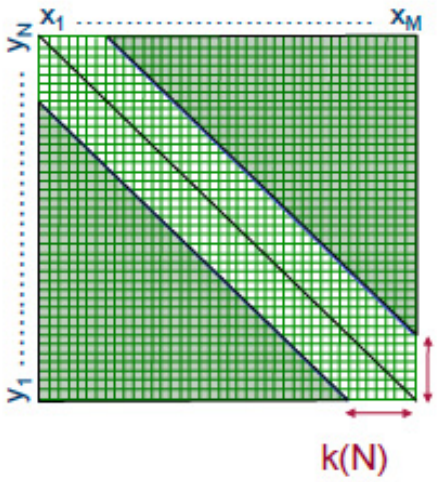

Figura 4: En caso de limitar la búsqueda a una serie de casillas alrededor de la diagonal ahorraremos cómputo evitando rellenar toda la matriz, la cual en condiciones habituales no tenderá a explorar los bordes.

1. Generamos una lista en Numpy para la dimensión de las filas desde 1 hasta la cantidad de filas. Seguimos de esta manera el aumento ascendente en esta dimensión.
2. Para la obtención de la lista de las columnas, necesitamos obtener índices de manera descendiente, y desplazadas según la iteración en la que nos encontremos. La manera más rápida de realizar esto es tomar la lista anteriormente creada y restársela al valor de la iteración actual. Al ser los valores de la segunda lista cada vez mayores, el resultado de la resta es cada vez menor, siguiendo un orden descendente. Si volvemos al ejemplo que comentamos al inicio, con valores (1 2 3) para la lista de las filas y encontrándonos en la tercera iteración (índice real 4), obtendremos la lista (3 2 1).
3. En el inicio y final de la matriz, así como en aquellas matrices cuya cantidad de filas sea mayor que la de columnas, obtendremos índices fuera del rango. Por ejemplo, en una matriz 6 x 3 en su cuarta iteración obtendremos la lista para las filas (1 2 3 4 5 6) y para las columnas (4 3 2 1 0 -1). Dado que las filas nunca se saldrán de la matriz (empezarán en 1 omitiendo el gap y acabarán en el máximo de filas) y teniendo en cuenta que las columnas dependen del valor de la iteración, pudiendo quedar casillas dentro o fuera, aplicaremos una máscara en Numpy, asegurando que los índices seleccionados para las columnas son mayores de 1 y menores que la cantidad de columnas.

Como se comentaba anteriormente, tras obtener el algoritmo de generación de diagonales aplicamos el cálculo del movimiento de manera escalar. Para ello, realizamos de manera individual y escalar el cálculo de la puntuación para cada movimiento (arriba, izquierda y diagonal superior izquierda) y asignamos en la matriz para el valor correspondiente el máximo de esta operación. De manera que sea cual sea el tamaño de la diagonal, esta tendrá un máximo de 4 operaciones aritméticas.

3.3. Algoritmo con banda diagonal

El algoritmo está diseñado para secuencias similares, y de similar tamaño, es por ello que la exploración por los bordes de la matriz es prácticamente nula. Para evitar el relleno de la matriz completa, se realiza una modificación en la que exploraremos la matriz dado un ancho de banda

por la diagonal. Un ejemplo visual de como se limita se puede visualizar en la Figura 4.

Para aplicar esto, se modificará la función de obtención de diagonales explicada en la Subsección 3.2. Se toma el lado de la matriz de menor tamaño y, dado un rango expresado como porcentaje, calcularemos el máximo de casillas de las cuales podrá estar compuesta la diagonal y descartaremos aquellas que queden fuera del rango en los extremos. Por ejemplo, en una matriz 10 x 10 aplicando un rango de 0,4 obtendremos que la cantidad máxima de casillas de la diagonal será siempre 4, y se tomarán de manera centrada descartando los extremos.

3.4. Algoritmo optimizado con Numba

En la versión básica del algoritmo de Needleman–Wunsch se ha empleado la librería Numba con el objetivo de mejorar el rendimiento durante la fase de alineamiento. Para ello, se ha añadido el decorador @jit a las funciones críticas del algoritmo (relleno de la matriz), lo que permite a Numba compilar dinámicamente el código Python a código máquina optimizado. Esta técnica reduce notablemente el tiempo de ejecución sin modificar la lógica del algoritmo original.

Numba resulta especialmente útil en implementaciones que realizan operaciones numéricas intensivas o bucles anidados, como es el caso del cálculo de la matriz de puntuación en Needleman–Wunsch. La documentación oficial de la librería puede consultarse en: https://numba.pydata.org/ .

Es importante señalar que esta optimización solo se ha aplicado a la versión básica, ya que Numba presenta ciertas limitaciones cuando se trabaja con estructuras de datos más complejas. En particular, Numba no permite el acceso a índices del tipo *M[i, j]* cuando estos pertenecen a objetos que no son arrays compatibles con su modo de compilación. Por este motivo, las versiones avanzadas del algoritmo no pudieron beneficiarse del uso de Numba sin una reestructuración profunda de sus estructuras internas.

4. Evaluación de optimizaciones

4.1. Criterios de comparación

Dadas los algoritmos y los distintos niveles de optimización, los algoritmos son evaluados con 3 criterios distintos.

1. *Velocidad*: tiempo de ejecución del algoritmo.
2. *Uso de memoria*: Almacenamiento necesario en RAM para ejecutar el algoritmo.
3. *Calidad del alineamiento*: Puntuación máxima del alineamiento conseguida.

4.2. Características del sistema

Las pruebas de rendimiento se realizaron en un sistema *VANT MOOVE15* con las siguientes especificaciones:

- **CPU:** Intel(R) Core(TM) 5 120U, 10 núcleos físicos, 12 hilos, caché L3 de 12 MB, soporta SSE, SSE2, SSE4.1, SSE4.2, AVX y AVX2.
- **Memoria:** 32 GB de RAM total, aproximadamente 25 GB disponibles para el sistema, sin uso de swap durante las pruebas.
- **Sistema operativo:** Ubuntu 24.04, kernel 6.14.0-35-generic, arquitectura x86_64.
- **Python:** versión 3.10.19
    - `NumPy` 2.2.6

- `Numba 0.62.1`

Es importante destacar que el computador utilizado tiene un sistema operativo de propósito general, en el cual otras aplicaciones podían ejecutarse mientras se realizaba la evaluación. En Ubuntu, los procesos de usuario evaluados tienen una prioridad baja comparado otros procesos del sistema como es la interfaz gráfica.

4.3. Benchmarks

Para evaluar el rendimiento de los algoritmos, se utilizó el genoma de referencia de *Escherichia coli* descargado de la base de datos *Entrez* [8]. A partir de este genoma se generaron **tres conjuntos de datos simulados** que representan diferentes longitudes y características de lectura:

- *Lecturas pequeñas (**Small**):* secuencias de 50 nucleótidos con un 1 % de errores, simulando lecturas tipo Illumina [9].
- *Lecturas medianas (**Medium**):* secuencias de 1000 nucleótidos con un 2 % de errores, representando genes o exones para alineamiento global [10].
- *Lecturas grandes (**Huge**):* secuencias de 5000 nucleótidos con un 2 % de errores, simulando contigs o scaffolds [11].

Estos conjuntos de datos permitieron evaluar el algoritmo bajo distintas condiciones de longitud y error de secuencia, reflejando escenarios reales de análisis de genomas. Los detalles de los datasets generados se resumen en la **Tabla 1**.

Cuadro 1: Benchmarks de E. coli

Benchmark	#Reads	Read Length	Error Rate
Small	20	50 nt	1 %
Medium	10	1000 nt	2 %
Huge	2	5000 nt	2 %

Para evaluar el algoritmo en condiciones realistas, a cada lectura del genoma de *Escherichia coli* se le generó una contraparte mutada aplicando cinco mutaciones aleatorias. De este modo, cada lectura original cuenta con una versión correspondiente que incorpora variaciones simuladas.

Las mutaciones se introdujeron en posiciones seleccionadas al azar y corresponden a los tres tipos básicos de alteraciones de secuencia: inserciones, deleciones y sustituciones.

Este procedimiento se aplicó a los tres conjuntos de datos descritos previamente (Small, Medium y Huge), garantizando que cada lectura tuviera su respectiva lectura mutada para las pruebas de alineamiento en tiempo de ejecución.

4.4. Resultados

4.4.1. Tiempo de ejecución

En la Tabla 2 se presentan los resultados del tiempo de ejecución y la aceleración (*speedup*) obtenidos para diferentes algoritmos al procesar los diferentes *bechmarks*. Los tiempos se han calculado ejecutando cada benchmark 5 veces con el paquete *timeit*[1] y calculado la media aritmética y desviación típica de todas las ejecuciones.

En cuento a tiempos, el alineamiento de 20 secuencias de 50 nucleótidos requiere menos de 1 segundo en todos los algoritmos. Para el conjunto *Medium*, la diferencia de rendimiento es

[1] https://www.geeksforgeeks.org/python/timeit-python-examples/

Algoritmo	Short		Medium		Huge	
	Tiempo (s)	Speedup	Tiempo (s)	Speedup	Tiempo (s)	Speedup
Basic	0.464 ± 0.090	1.00	73.05 ± 5.30	1.00	407.547 ± 12.92	1.00
Diag	0.081 ± 0.007	5.75	1.70 ± 0.17	42.80	4.91 ± 0.20	82.92
Diag B.	0.096 ± 0.016	4.83	1.55 ± 0.09	46.95	4.02 ± 0.40	**101.18**
Numba	0.015 ± 0.001	**31.77**	1.35 ± 0.13	**54.08**	6.87 ± 0.35	59.26

Cuadro 2: Resultados de los *bechmarks* para diferentes algoritmos. Los tiempos muestran media ± desviación estándar y el *speedup* se calcula respecto al algoritmo básico.

mucho más notable: la versión básica necesita alrededor de 70 segundos para alinear 10 secuencias, mientras que la implementación más eficiente lo hace en aproximadamente 1 segundo. Finalmente, en el *bechmark*. *Huge*, compuesto por 2 secuencias de más de 5000 nucleótidos, la versión sin optimizaciones supera los 400 segundos (alrededor de 6 minutos), mientras que la versión más eficiente completa el proceso en apenas 4 segundos.

Con respecto a *speedup*, según se observa en Figura 5, la implementación con `Numba` muestra un aumento significativo en el rendimiento respecto al método básico, alcanzando un *speedup* de más de 30 veces para *short reads* y más de 50 veces para *medium reads*. Los métodos *Diagonal* y *Diagonal Bounded* también ofrecen mejoras importantes, aunque menores que `Numba`.

Sin embargo, para el benchmark *Huge*, los algoritmos basados en diagonales superan considerablemente a la implementación en Numba. En particular, el algoritmo con banda diagonal alcanza un *speedup* superior a 100, mientras que Numba obtiene un *speedup* de aproximadamente 60 para este mismo conjunto de datos.

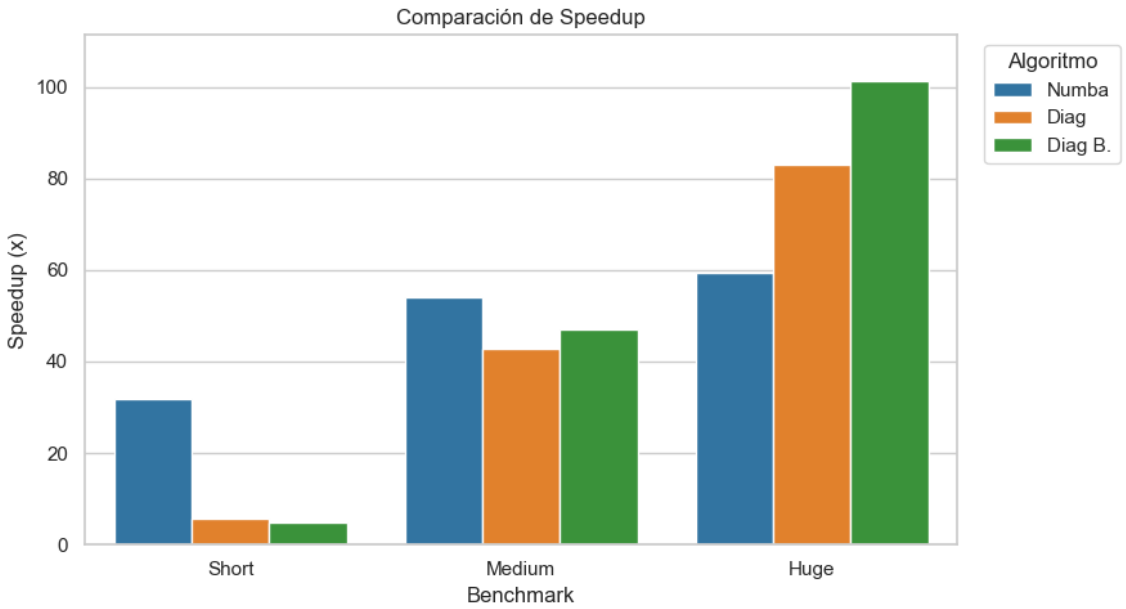

Figura 5: Comparación del *Speedup* por algoritmo según el tamaño de longitud de las secuencias

4.4.2. Uso de memoria

El uso de memoria es similar en todas las implementaciones, ya que en todos los casos la matriz de puntuaciones es la misma.

En el caso peor, para alinear dos secuencias de 5000 nucleótidos, almacenar las puntuaciones en *int64* requiere aproximadamente 0.74 GB de memoria. Esto sugiere que, incluso para

secuencias de este tamaño, cargar las secuencias en memoria de manera individual no representa un problema significativo, considerando que la mayoría de los computadores cuentan con aproximadamente 16GB de memoria *RAM*.

4.4.3. Calidad del alineamiento

Dado que algunas optimizaciones no rellenan completamente la matriz, puede provocar en ciertos casos que no se realice el alineamiento con la mejor puntuación. Es por ello que será necesario explorar de manera exhaustiva este aspecto.

Para verificarlo, generamos una secuencia base de 1000 nucleótidos. Para esta secuencia base, generamos 7 secuencias mutadas, aplicando tantos cambios como el tamaño de la secuencia base por el porcentaje de mutaciones deseado, en este caso (0.8, 0.6, 0.4, 0.2, 0.1, 0.05, 0.01). Por ejemplo para la secuencia base, 0.6 corresponde con 600 operaciones de inserción, eliminación o substitución. De la misma manera, se generan otras siete secuencias en las que se elimina nucleótidos de posiciones aleatorias, siguiendo los mismos porcentajes. Los resultados finales se encuentran en la Tabla 3.

	Basic	Diag	Numba	Diag B.
mut-0.8	48	48	48	48
mut-0.6	158	158	158	158
mut-0.4	338	338	338	338
mut-0.2	603	603	603	603
mut-0.1	791	791	791	791
mut-0.05	885	885	885	885
mut-0.01	978	978	978	978
rem-0.8	-1400	-1400	-1400	-1430
rem-0.6	-800	-800	-800	-808
rem-0.4	-200	-200	-200	-200
rem-0.2	400	400	400	400
rem-0.1	700	700	700	700
rem-0.05	850	850	850	850
rem-0.01	970	970	970	970

Cuadro 3: En esta tabla se refleja las puntuaciones de los alineamientos de los 4 algoritmos. Se evalúan en 2 secciones, mutaciones (*mut)* y eliminaciones (*rem*).

En lo que a mutaciones se refiere, la puntuación permanece la misma para todos los algoritmos, obteniendo una puntuación peor en aquellas secuencias en las que se aplique una mayor cantidad de mutaciones. Por otro lado, en cuanto a las eliminaciones, solo encontramos una menor puntuación entre algoritmos para el *algoritmo Diag B.* en aquellas secuencias cuyo porcentaje de eliminaciones sea muy alto. La diagonal al reducirse en base al lado de menor longitud, deja una diagonal muy pequeña con abundantes eliminaciones, es por ello que se penaliza la exploración por los bordes. En este caso, se aprecia la reducción en cuanto se elimina un 60 % de los nucleótidos o más.

5. Desarrollo de la aplicación

Una aplicación web didáctica se ha desarrollado para visulizar y entender el algoritmo de Needle wunch. Esta visualziación se ha hecho con el framweork web *Streamlit*[2] que permite un desarrollo sin conocimientos profundos de diseño web.

[2]https://streamlit.io/

5.1. Funcionalidades

La funcionalidad principal de la aplicación consiste en recibir dos secuencias de entrada, calcular sus alineamientos y mostrar los resultados de manera clara y visual. Esto permite a los usuarios analizar y comparar secuencias de manera intuitiva, facilitando la interpretación de los resultados de los algoritmos de alineamiento.

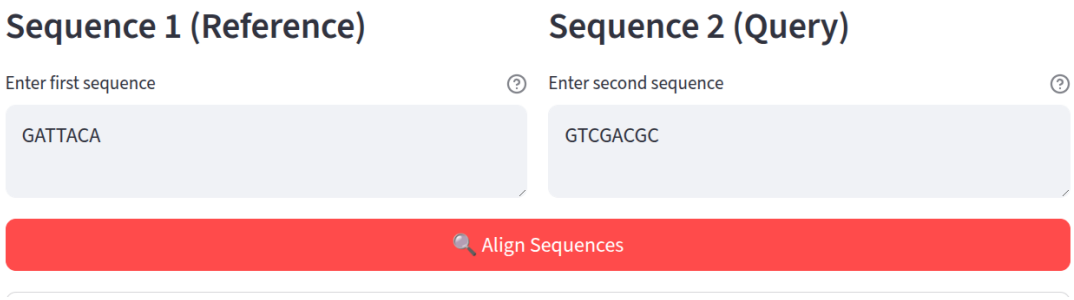

Figura 6: Entrada de secuencias en la interfaz principal. Los usuarios pueden ingresar o cargar las secuencias que desean comparar.

Una vez ingresadas las secuencias, la aplicación realiza el alineamiento y presenta el resultado mediante una visualización que permite identificar coincidencias, sustituciones y gaps de manera sencilla (Figura 7).

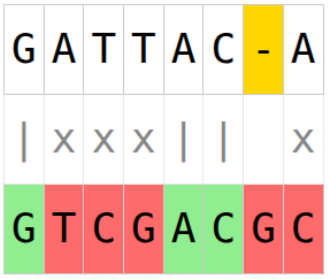

Figura 7: Visualización del alineamiento de secuencias. La interfaz muestra de forma clara los resultados obtenidos tras aplicar el algoritmo de alineamiento.

La aplicación también incluye un menú lateral de configuración que permite a los usuarios ajustar parámetros del algoritmo, como los valores de puntaje para coincidencias, sustituciones y penalizaciones por gaps (Figura 8).

Adicionalmente, se han incorporado secciones extras que permiten al usuario explorar funcionalidades avanzadas y detalles del proceso de alineamiento (Figura 9).

Finalmente, se ha integrado en la aplicación un recurso adicional disponible en la web para comprender el funcionamiento de la matriz de puntuación (Figura 10). Este recurso, disponible en experiments.mostafa.io/needleman-wunsch/, permite a los usuarios ingresar secuencias y visualizar detalladamente la matriz de puntuación utilizada en el algoritmo de Needleman-Wunsch. Además, se puede seguir paso a paso el proceso de *traceback*, mostrando cómo se seleccionan los puntajes para determinar el alineamiento óptimo. Esta herramienta facilita la comprensión de los criterios de coincidencia, sustitución y penalización por gaps, ofreciendo una perspectiva más interactiva y didáctica del funcionamiento interno del algoritmo.

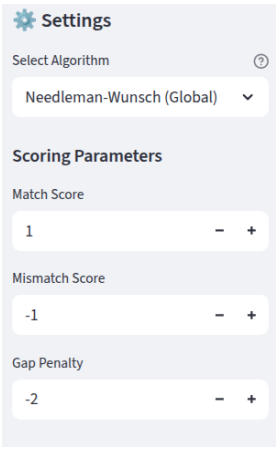

Figura 8: Menú lateral de configuración, donde se pueden ajustar parámetros del algoritmo de alineamiento.

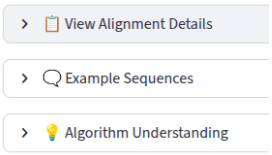

Figura 9: Sección adicional para visualización avanzada de resultados y parámetros del alineamiento.

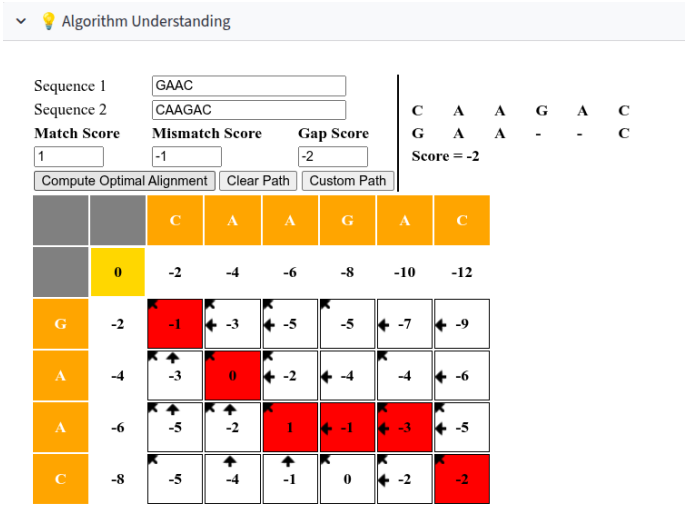

Figura 10: Visualización detallada del funcionamiento del algoritmo, incluyendo la matriz de puntuación utilizada para calcular el alineamiento.

5.2. Arquitectura

La aplicación sigue el patrón de diseño *Modelo-Vista-Controlador (MVC)*, lo que permite separar las responsabilidades de la aplicación y mejorar su mantenibilidad, extensibilidad y facilidad de pruebas. La arquitectura se organiza en cuatro componentes principales:

- *Modelo* (algorithms.py): contiene la lógica principal de los algoritmos de alineamiento, incluyendo la gestión de matrices de puntuación y la ejecución de los procesos de *traceback*.
- *Vista* (visualization.py): se encarga de formatear los resultados, generar visualizaciones y componentes de la interfaz como estadísticas y ejemplos de secuencias.
- *Controlador* (app.py): actúa como intermediario, gestionando la interacción del usuario, validando entradas y coordinando la comunicación entre Modelo y Vista.
- *Configuración* (config.py): centraliza parámetros para configurar títulos, colores y opciones por defecto.

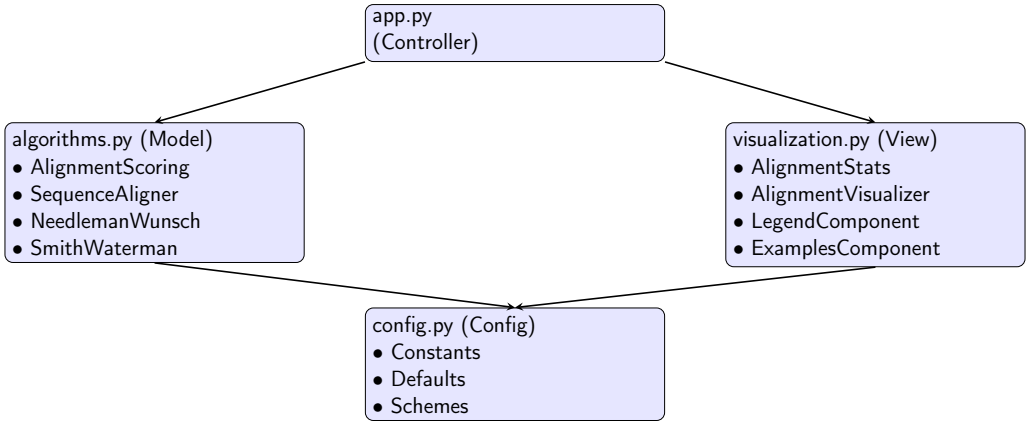

Figura 11: Arquitectura MVC de BioSeqAligner

6. Conclusiones y trabajo futuro

El algoritmo basado en diagonales con una banda del 40 % resulta ser la opción más eficiente para el alineamiento de secuencias superiores a 5000 nucleótidos. Su rendimiento no solo es significativamente mejor en términos computacionales, sino que además mantiene una calidad de alineamiento comparable a la de los algoritmos no optimizados. Solo se observa una degradación apreciable cuando el número de eliminaciones supera el 60 %, un escenario extremo dado que la divergencia entre genomas humanos es cercana al 1 % [12] y el número de mutaciones de novo por generación en humanos es aproximadamente 60 mutaciones por individuo [13] .

Como trabajo futuro, se propone reducir el coste de memoria del algoritmo con banda diagonal, ya que actualmente, aunque parte de la matriz no se utiliza, esta sigue creándose en su totalidad. Además, se pueden probar diferentes niveles de *banding* para evaluar su impacto en el rendimiento. Finalmente, se recomienda repetir las pruebas en un computador con un sistema operativo dedicado o con la interfaz gráfica desactivada, con el fin de obtener mediciones más precisas y estables.

7. Accesibilidad

La herramienta está disponible a través de la URL: https://bioaligner.streamlit.app/.

Además, el código está liberado bajo una licencia de código abierto, por lo que otras personas pueden mejorarlo y contribuir al proyecto a través del siguiente repositorio de Github.

Appendice

Implementaciones en Python

La implementación básica del algoritmo puede verse en Figura 12, usando arrays numpy como estructura de datos.

Needlman Wunsch Basic Algorithm

```
def needleman_wunsch(a, b):
    n, m = len(a), len(b)

    F = np.zeros((n+1, m+1), dtype=np.int64)
    F[1:, 0] = np.arange(1, n+1) * GAP
    F[0, 1:] = np.arange(1, m+1) * GAP

    # Rellenar matriz
    for i in range(1, n+1):
        for j in range(1, m+1):
            diag = F[i-1][j-1] + sim(a[i-1], b[j-1])
            up   = F[i-1][j] + GAP
            left = F[i][j-1] + GAP
            F[i][j] = max(diag, up, left)

    ref, best_align = traceforward(F, a, b)

    return F[n][m], ref, best_align
```

Figura 12: Implementación básica en Python

Referencias

[1] David W. Mount. *Bioinformatics: Sequence and genome analysis*. 2.ª ed. Cold Spring Harbor, NY: Cold Spring Harbor Laboratory Press, 2004.

[2] Saul B. Needleman y Christian D. Wunsch. «A general method applicable to the search for similarities in the amino acid sequence of two proteins». En: *Journal of Molecular Biology* 48.3 (1970), págs. 443-453. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(70)90057-4.

[3] Robert C. Edgar. «MUSCLE: multiple sequence alignment with high accuracy and high throughput». En: *Nucleic Acids Research* 32.5 (mar. de 2004), págs. 1792-1797. ISSN: 0305-1048. DOI: 10.1093/nar/gkh340.

[4] S. Henikoff y J. G. Henikoff. «Amino acid substitution matrices from protein blocks». En: *Proceedings of the National Academy of Sciences of the United States of America* 89.22 (1992), págs. 10915-10919. DOI: 10.1073/pnas.89.22.10915.

[5] T.F. Smith y M.S. Waterman. «Identification of common molecular subsequences». En: *Journal of Molecular Biology* 147.1 (1981), págs. 195-197. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(81)90087-5.

[6] Charles R. Harris et al. «Array programming with NumPy». En: 585.7825 (sep. de 2020), págs. 357-362. DOI: 10.1038/s41586-020-2649-2. arXiv: 2006.10256 [cs.MS].

[7] Siu Kwan Lam, Antoine Pitrou y Stanley Seibert. «Numba: a LLVM-based Python JIT compiler». En: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162.

[8] National Center for Biotechnology Information. *Entrez Molecular Sequence Database System*. NCBI. Bethesda, MD, 2004. URL: https://www.ncbi.nlm.nih.gov/Web/Search/entrezfs.html.

[9] David R. Bentley et al. «Accurate whole human genome sequencing using reversible terminator chemistry». En: *Nature* 456.7218 (2008), págs. 53-59. DOI: 10.1038/nature07517. URL: https://doi.org/10.1038/nature07517.

[10] Wikipedia. *Par de bases*. [Online; accessed 16-November-2025]. 2025. URL: https://es.wikipedia.org/w/index.php?title=Par_de_bases&oldid=162558121.

[11] Wikipedia. *Contig*. [Online; accessed 16-November-2025]. 2025. URL: https://en.wikipedia.org/wiki/Contig.

[12] 1000 Genomes Project Consortium. «A map of human genome variation from population-scale sequencing». En: *Nature* 467.7319 (2010), págs. 1061-1073. ISSN: 0028-0836. DOI: 10.1038/nature09534.

[13] Augustine Kong et al. «Rate of de novo mutations and the importance of father's age to disease risk». En: *Nature* 488.7412 (2012), págs. 471-475. DOI: 10.1038/nature11396.