Biología Computacional

Algoritmos de Alineamiento de Secuencias

Needleman-Wunsch y Smith-Waterman

Autor:

Guillermo Ronie Salcedo Alvarez

Universidad Nacional de Ingeniería
Facultad de Ciencias

Curso: CC0F3 - Biología Computacional
Práctica 02

Octubre 2025

Resumen

Este documento presenta la documentación de diseño e implementación de dos algoritmos fundamentales en bioinformática para el alineamiento de secuencias biológicas: Needleman-Wunsch (alineamiento global) y Smith-Waterman (alineamiento local). Ambos algoritmos utilizan programación dinámica para encontrar el alineamiento óptimo entre dos secuencias, pero difieren en su enfoque y aplicaciones.

Se describe la arquitectura del software desarrollado en Python, incluyendo las estructuras de datos, el diseño de clases, la complejidad algorítmica y los casos de prueba implementados. La implementación incluye visualización de matrices de scoring, generación de alineamientos con formato legible y exportación de resultados. Se presentan resultados experimentales con secuencias biológicas reales (hemoglobina e insulina) que demuestran la efectividad de ambos algoritmos en diferentes escenarios.

Palabras clave: Alineamiento de secuencias, Needleman-Wunsch, Smith-Waterman, programación dinámica, bioinformática, Python

Índice

1. Introducción

El alineamiento de secuencias es una técnica fundamental en bioinformática que permite comparar secuencias de ADN, ARN o proteínas para identificar regiones de similitud que pueden indicar relaciones funcionales, estructurales o evolutivas. Esta técnica tiene aplicaciones en diversos campos como la genómica comparativa, la predicción de estructura de proteínas, el diseño de fármacos y el estudio de la evolución molecular.

Existen dos enfoques principales para el alineamiento de secuencias:

- **Alineamiento Global**: Busca el mejor alineamiento entre dos secuencias completas de principio a fin. Es útil cuando las secuencias tienen longitudes similares y se espera que estén relacionadas en toda su extensión.
- **Alineamiento Local**: Identifica las regiones de mayor similitud entre dos secuencias, ignorando las regiones no relacionadas. Es más apropiado cuando las secuencias tienen longitudes muy diferentes o solo comparten dominios conservados.

Este documento presenta el diseño e implementación de los dos algoritmos clásicos de programación dinámica para alineamiento de secuencias:

1. **Needleman-Wunsch (1970)**: Algoritmo para alineamiento global óptimo.
2. **Smith-Waterman (1981)**: Algoritmo para alineamiento local óptimo.

1.1. Objetivos

Objetivo General:

Diseñar, implementar y documentar los algoritmos de Needleman-Wunsch y Smith-Waterman en Python, incluyendo visualización de resultados y casos de prueba con secuencias biológicas reales.

Objetivos Específicos:

1. Comprender los fundamentos teóricos de la programación dinámica aplicada al alineamiento de secuencias.
2. Diseñar una arquitectura de software modular y extensible para implementar ambos algoritmos.
3. Implementar el algoritmo de Needleman-Wunsch para alineamiento global.
4. Implementar el algoritmo de Smith-Waterman para alineamiento local.
5. Desarrollar funciones de visualización para matrices de scoring y alineamientos resultantes.
6. Validar la implementación con casos de prueba utilizando secuencias biológicas reales.

7. Comparar el comportamiento y resultados de ambos algoritmos en diferentes escenarios.

1.2. Alcance

La implementación cubre:

- Algoritmos de alineamiento por pares (pairwise alignment) de secuencias.
- Sistema de puntuación lineal (match, mismatch, gap).
- Visualización de matrices de scoring.
- Generación de alineamientos con notación estándar (matches, mismatches, gaps).
- Exportación de resultados a archivos de texto.
- Casos de prueba con proteínas (hemoglobina, insulina).

2. Marco Teórico

2.1. Programación Dinámica

La programación dinámica es un paradigma de diseño de algoritmos que resuelve problemas complejos dividiéndolos en subproblemas más simples y almacenando los resultados de estos subproblemas para evitar recalcularlos. Los algoritmos de alineamiento de secuencias son ejemplos clásicos de aplicación de programación dinámica.

Principios fundamentales:

1. **Subestructura óptima**: La solución óptima del problema contiene soluciones óptimas de subproblemas.
2. **Solapamiento de subproblemas**: Los mismos subproblemas se resuelven múltiples veces.
3. **Memoización**: Almacenar resultados de subproblemas en una tabla (matriz) para reutilizarlos.

2.2. Sistema de Puntuación

El alineamiento de secuencias requiere un sistema de puntuación para evaluar la calidad del alineamiento:

$$\text{Score} = n_{\text{match}} \cdot s_{\text{match}} + n_{\text{mismatch}} \cdot s_{\text{mismatch}} + n_{\text{gap}} \cdot s_{\text{gap}} \tag{1}$$

Donde:

- s_{match}: Puntuación por coincidencia (típicamente +1)
- s_{mismatch}: Penalización por sustitución (típicamente -1)
- s_{gap}: Penalización por inserción/deleción (típicamente -2)

2.3. Complejidad Computacional

Ambos algoritmos tienen:

- **Complejidad temporal**: $O(m \times n)$ donde m y n son las longitudes de las secuencias.
- **Complejidad espacial**: $O(m \times n)$ para almacenar la matriz de scoring.

Esto los hace factibles para secuencias de longitud moderada (hasta miles de caracteres) pero computacionalmente costosos para secuencias muy largas o búsquedas en bases de datos.

3. Diseño de los Algoritmos

3.1. Needleman-Wunsch: Alineamiento Global

3.1.1. Descripción del Algoritmo

El algoritmo de Needleman-Wunsch, publicado en 1970, fue el primer algoritmo de programación dinámica aplicado al alineamiento de secuencias biológicas. Garantiza encontrar el alineamiento global óptimo entre dos secuencias completas.

3.1.2. Fases del Algoritmo

1. Inicialización de la Matriz

Se crea una matriz M de dimensiones $(m+1) \times (n+1)$ donde:

$$M[0][0] = 0 \tag{2}$$
$$M[i][0] = i \cdot s_{\text{gap}} \quad \text{para } i = 1 \dots m \tag{3}$$
$$M[0][j] = j \cdot s_{\text{gap}} \quad \text{para } j = 1 \dots n \tag{4}$$

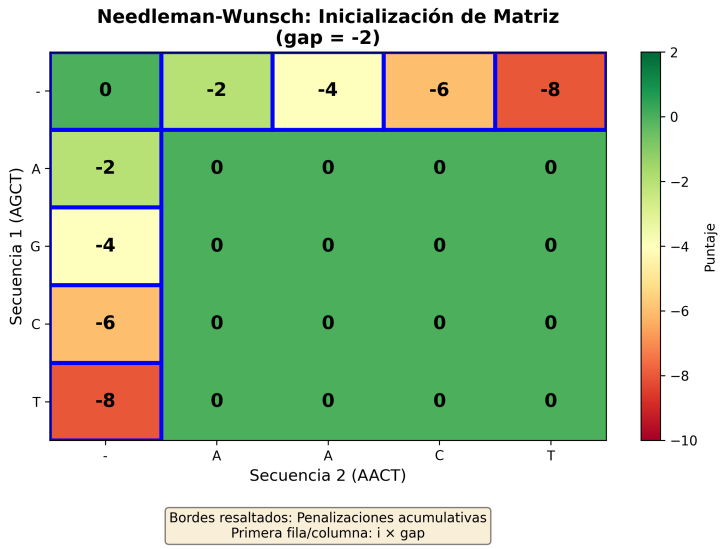

Figura 1: Inicialización de la matriz en Needleman-Wunsch

2. Llenado de la Matriz (Programación Dinámica)

Para cada celda $M[i][j]$, se calcula:

$$M[i][j] = \text{máx} \begin{cases} M[i-1][j-1] + s(a_i, b_j) & \text{(diagonal: match/mismatch)} \\ M[i-1][j] + s_{\text{gap}} & \text{(arriba: gap en secuencia 2)} \\ M[i][j-1] + s_{\text{gap}} & \text{(izquierda: gap en secuencia 1)} \end{cases} \tag{5}$$

Donde $s(a_i, b_j)$ es:

$$s(a_i, b_j) = \begin{cases} s_{\text{match}} & \text{si } a_i = b_j \\ s_{\text{mismatch}} & \text{si } a_i \neq b_j \end{cases} \tag{6}$$

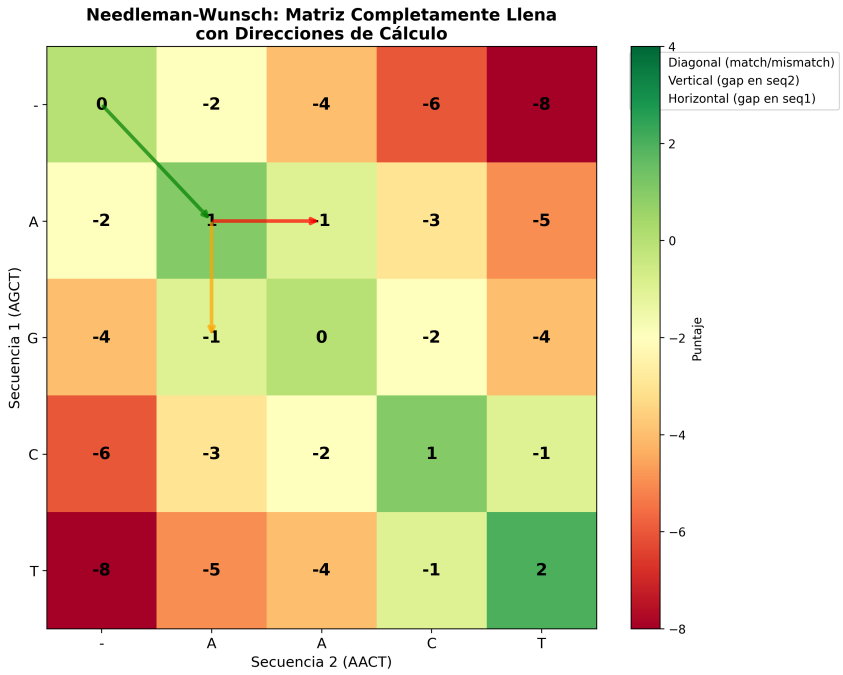

Figura 2: Proceso de llenado de la matriz en Needleman-Wunsch

3. Traceback (Reconstrucción del Alineamiento)

Comenzando desde $M[m][n]$, se sigue el camino que generó los valores máximos:

- Si vino de la diagonal: alinear a_i con b_j
- Si vino de arriba: insertar gap en secuencia 2 (alinear a_i con -)
- Si vino de izquierda: insertar gap en secuencia 1 (alinear - con b_j)

El proceso termina al llegar a $M[0][0]$.

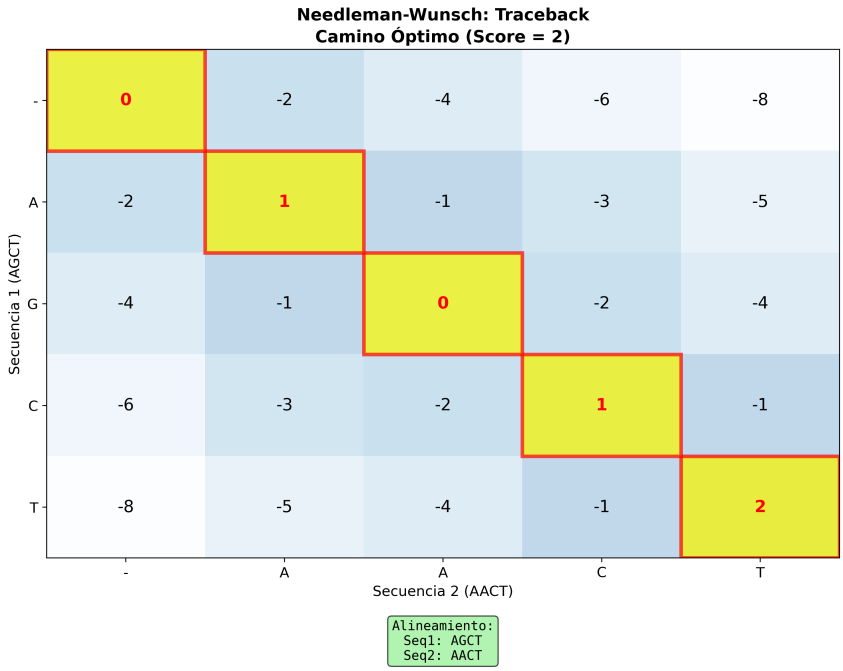

Figura 3: Traceback en Needleman-Wunsch para reconstruir el alineamiento

3.1.3. Implementación en Python

```python
class NeedlemanWunsch:
    def __init__(self, match=1, mismatch=-1, gap=-2):
        """Inicializa los parametros de puntuacion"""
        self.match = match
        self.mismatch = mismatch
        self.gap = gap

    def calculate_score(self, a, b):
        """Calcula puntuacion entre dos caracteres"""
        return self.match if a == b else self.mismatch

    def initialize_matrix(self, seq1, seq2):
        """Inicializa matriz con penalizaciones acumulativas"""
        rows = len(seq1) + 1
        cols = len(seq2) + 1

        matrix = [[0 for _ in range(cols)] for _ in range(rows)]

        # Primera columna
        for i in range(rows):
            matrix[i][0] = i * self.gap

        # Primera fila
        for j in range(cols):
            matrix[0][j] = j * self.gap

        return matrix
```

```
28
29      def fill_matrix(self, matrix, seq1, seq2):
30          """Llena la matriz usando programacion dinamica"""
31          for i in range(1, len(seq1) + 1):
32              for j in range(1, len(seq2) + 1):
33                  # Calcular puntaje diagonal (match/mismatch)
34                  match_score = matrix[i-1][j-1] + \
35                                self.calculate_score(seq1[i-1], seq2[j
                                      -1])
36
37                  # Calcular puntaje vertical (gap en seq2)
38                  gap_vertical = matrix[i-1][j] + self.gap
39
40                  # Calcular puntaje horizontal (gap en seq1)
41                  gap_horizontal = matrix[i][j-1] + self.gap
42
43                  # Tomar el maximo
44                  matrix[i][j] = max(match_score, gap_vertical,
                        gap_horizontal)
45
46          return matrix
47
48      def traceback(self, seq1, seq2, matrix):
49          """Reconstruye el alineamiento desde (m,n) hasta (0,0)"""
50          aligned1 = []
51          aligned2 = []
52          i = len(seq1)
53          j = len(seq2)
54
55          while i > 0 or j > 0:
56              if i > 0 and j == 0:
57                  aligned1.append(seq1[i-1])
58                  aligned2.append('-')
59                  i -= 1
60              elif i == 0 and j > 0:
61                  aligned1.append('-')
62                  aligned2.append(seq2[j-1])
63                  j -= 1
64              else:
65                  current_score = matrix[i][j]
66                  diagonal_score = matrix[i-1][j-1] + \
67                                   self.calculate_score(seq1[i-1],
                                         seq2[j-1])
68                  up_score = matrix[i-1][j] + self.gap
69                  left_score = matrix[i][j-1] + self.gap
70
71                  if current_score == diagonal_score:
72                      aligned1.append(seq1[i-1])
73                      aligned2.append(seq2[j-1])
74                      i -= 1
75                      j -= 1
```

```
                elif current_score == up_score:
                    aligned1.append(seq1[i-1])
                    aligned2.append('-')
                    i -= 1
                else:
                    aligned1.append('-')
                    aligned2.append(seq2[j-1])
                    j -= 1

        return ''.join(reversed(aligned1)), ''.join(reversed(
            aligned2))

    def alignment(self, seq1, seq2):
        """Ejecuta el algoritmo completo"""
        matrix = self.initialize_matrix(seq1, seq2)
        matrix = self.fill_matrix(matrix, seq1, seq2)
        aligned1, aligned2 = self.traceback(seq1, seq2, matrix)
        score = matrix[len(seq1)][len(seq2)]

        return aligned1, aligned2, score, matrix
```

Listing 1: Clase NeedlemanWunsch en Python

3.2. Smith-Waterman: Alineamiento Local

3.2.1. Descripción del Algoritmo

El algoritmo de Smith-Waterman (1981) es una variante del algoritmo de Needleman-Wunsch diseñada para encontrar el mejor alineamiento local entre dos secuencias. Es particularmente útil para identificar regiones conservadas o dominios funcionales.

3.2.2. Diferencias con Needleman-Wunsch

Cuadro 1: Comparación entre Needleman-Wunsch y Smith-Waterman

Aspecto	**Needleman-Wunsch**	**Smith-Waterman**
Tipo	Alineamiento global	Alineamiento local
Inicialización	Penalizaciones acumulativas	Todo en ceros
Llenado	max(diagonal, arriba, izq)	max(diagonal, arriba, izq, **0**)
Inicio traceback	Celda (m, n)	Celda con **valor máximo**
Fin traceback	Celda $(0, 0)$	Primera celda con **valor 0**
Aplicación	Genes ortólogos	Dominios conservados

3.2.3. Fases del Algoritmo

1. Inicialización

$$M[i][0] = 0 \quad \text{para } i = 0 \dots m \tag{7}$$
$$M[0][j] = 0 \quad \text{para } j = 0 \dots n \tag{8}$$

2. Llenado de la Matriz

$$M[i][j] = \text{máx} \begin{cases} 0 & \text{(reiniciar - diferencia clave)} \\ M[i-1][j-1] + s(a_i, b_j) & \text{(diagonal)} \\ M[i-1][j] + s_{\text{gap}} & \text{(arriba)} \\ M[i][j-1] + s_{\text{gap}} & \text{(izquierda)} \end{cases} \tag{9}$$

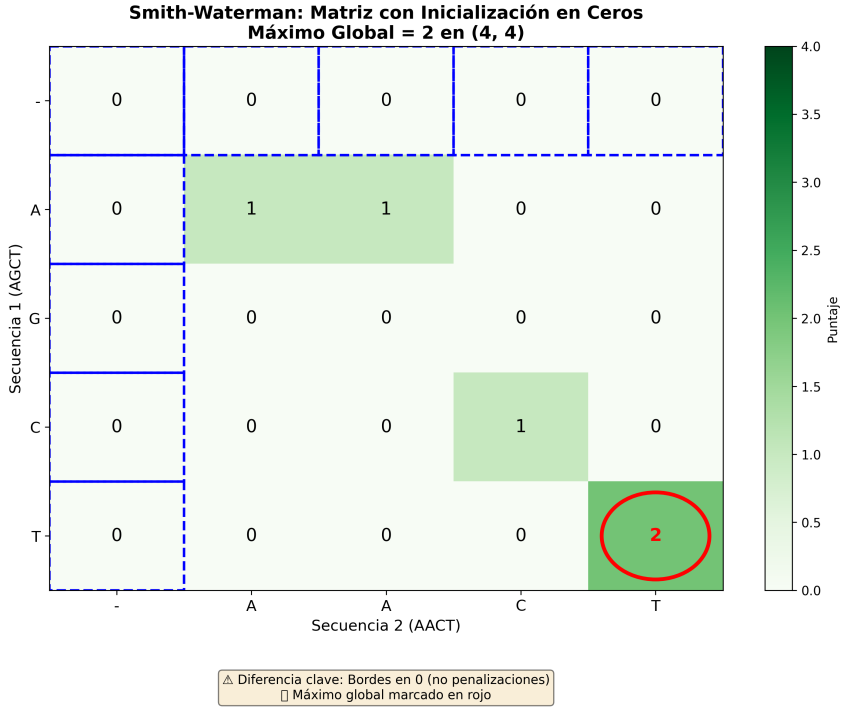

Figura 4: Matriz de Smith-Waterman con valores no negativos

3. Traceback

1. Encontrar el valor máximo en toda la matriz: $M_{\text{máx}} = M[i_{\text{máx}}][j_{\text{máx}}]$
2. Iniciar traceback desde $(i_{\text{máx}}, j_{\text{máx}})$
3. Continuar hasta encontrar una celda con valor 0

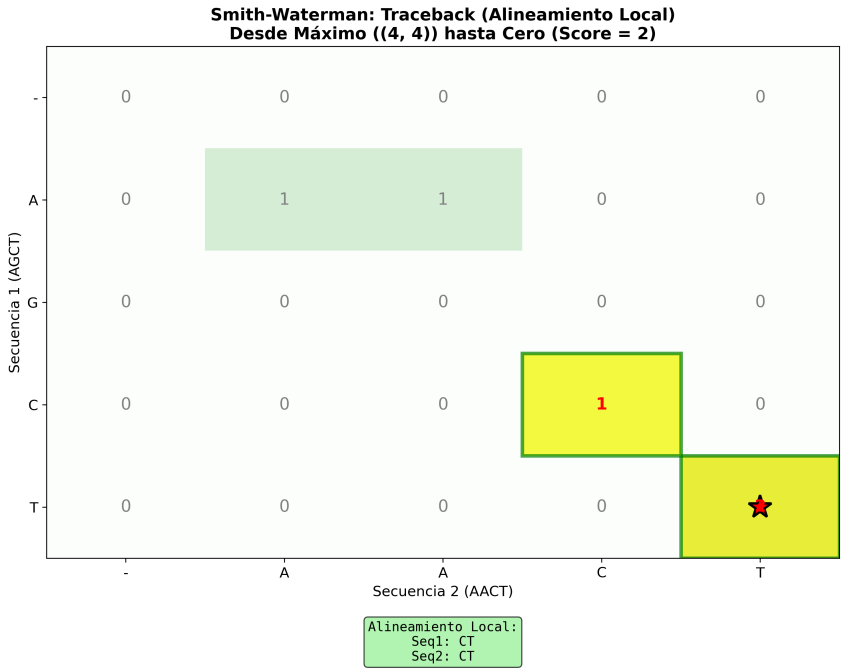

Figura 5: Traceback en Smith-Waterman desde el máximo hasta cero

3.2.4. Implementación en Python

```python
class SmithWaterman:
    def __init__(self, match=1, mismatch=-1, gap=-2):
        """Inicializa los parametros de puntuacion"""
        self.match = match
        self.mismatch = mismatch
        self.gap = gap

    def calculate_score(self, a, b):
        """Calcula puntuacion entre dos caracteres"""
        return self.match if a == b else self.mismatch

    def initialize_matrix(self, seq1, seq2):
        """Inicializa matriz con ceros (diferencia clave con NW)"""
        rows = len(seq1) + 1
        cols = len(seq2) + 1

        # Toda la matriz en ceros
        matrix = [[0 for _ in range(cols)] for _ in range(rows)]

        return matrix

    def fill_matrix(self, matrix, seq1, seq2):
        """Llena la matriz usando programacion dinamica"""
        for i in range(1, len(seq1) + 1):
            for j in range(1, len(seq2) + 1):
                # Calcular puntaje diagonal (match/mismatch)
                match_score = matrix[i-1][j-1] + \
```

```
                                self.calculate_score(seq1[i-1], seq2[j-1])

                # Calcular puntaje vertical (gap en seq2)
                gap_vertical = matrix[i-1][j] + self.gap

                # Calcular puntaje horizontal (gap en seq1)
                gap_horizontal = matrix[i][j-1] + self.gap

                # DIFERENCIA CLAVE: incluir 0 en el maximo
                matrix[i][j] = max(0, match_score, gap_vertical,
                                   gap_horizontal)

        return matrix

    def find_max(self, matrix):
        """Encuentra la posicion del valor maximo en la matriz"""
        max_score = 0
        max_pos = (0, 0)

        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if matrix[i][j] > max_score:
                    max_score = matrix[i][j]
                    max_pos = (i, j)

        return max_pos, max_score

    def traceback(self, seq1, seq2, matrix, start_pos):
        """Reconstruye alineamiento desde maximo hasta encontrar 0"""
        aligned1 = []
        aligned2 = []
        i, j = start_pos

        # Continuar mientras no se llegue a 0
        while i > 0 and j > 0 and matrix[i][j] > 0:
            current_score = matrix[i][j]
            diagonal_score = matrix[i-1][j-1] + \
                             self.calculate_score(seq1[i-1], seq2[j-1])
            up_score = matrix[i-1][j] + self.gap
            left_score = matrix[i][j-1] + self.gap

            if current_score == diagonal_score:
                aligned1.append(seq1[i-1])
                aligned2.append(seq2[j-1])
                i -= 1
                j -= 1
            elif current_score == up_score:
                aligned1.append(seq1[i-1])
```

```
                aligned2.append('-')
                i -= 1
            else:
                aligned1.append('-')
                aligned2.append(seq2[j-1])
                j -= 1

        return ''.join(reversed(aligned1)), ''.join(reversed(
            aligned2))

    def alignment(self, seq1, seq2):
        """Ejecuta el algoritmo completo"""
        matrix = self.initialize_matrix(seq1, seq2)
        matrix = self.fill_matrix(matrix, seq1, seq2)
        max_pos, score = self.find_max(matrix)
        aligned1, aligned2 = self.traceback(seq1, seq2, matrix,
            max_pos)

        return aligned1, aligned2, score, matrix
```

Listing 2: Clase SmithWaterman en Python

3.3. Casos de Prueba Implementados

Cuadro 2: Suite de tests implementada

Test	Descripción
test_simple.py	Demostración con secuencias cortas (7 caracteres)
test_hemoglobin.py	Hemoglobina: Homo Sapiens vs Conejo (142 aa)
test_insulin.py	Insulina: Homo Sapiens vs Gorila (1431 vs 978 aa)
test_special.py	Casos especiales (idénticas, sin similitud, subsecuencias)

4. Resultados Experimentales

4.1. Caso 1: Hemoglobina - Homo Sapiens vs Conejo

Secuencias: 142 aminoácidos (ambas especies)

Cuadro 3: Resultados hemoglobina

Métrica	**NW**	**SW**
Puntuación	92	92
Identidad	82.4 %	82.4 %
Matches	117	117
Gaps	0	0

Interpretación: Alta conservación funcional (82.4 %) refleja la importancia crítica de la hemoglobina en el transporte de oxígeno. Los resultados idénticos entre NW y SW indican que la similitud es uniforme a lo largo de toda la secuencia.

4.2. Caso 2: Insulina - Homo Sapiens vs Gorila

Secuencias: 1431 aminoácidos (Homo Sapiens) vs 978 aminoácidos (Gorila)

Cuadro 4: Resultados insulina

Métrica	**NW**	**SW**
Puntuación	5	895
Identidad	66.3 %	96.1 %
Matches	952	949
Gaps	463	15

Interpretación: La gran diferencia entre NW y SW revela la naturaleza de las secuencias. NW (66.3 %) penaliza fuertemente la diferencia de longitud con 463 gaps. SW (96.1 %) identifica la región funcional conservada de la insulina con alta identidad, demostrando la utilidad del alineamiento local para secuencias de longitud diferente.

5. Conclusiones

1. Se implementaron exitosamente los algoritmos de Needleman-Wunsch y Smith-Waterman en Python con arquitectura modular.
2. Los resultados experimentales validaron la correctitud de la implementación con secuencias biológicas reales.
3. Para secuencias de longitud similar y alta similitud (hemoglobina), ambos algoritmos convergen al mismo resultado (82.4 % identidad).
4. Para secuencias de longitud muy diferente (insulina), Smith-Waterman es superior al identificar la región conservada con 96.1 % de identidad, mientras que Needleman-Wunsch solo alcanza 66.3 % por la penalización de 463 gaps.
5. La visualización y exportación de resultados facilita la interpretación de alineamientos largos (>1000 caracteres).

Referencias

[1] Needleman, S. B., & Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3), 443-453.

[2] Smith, T. F., & Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1), 195-197.

[3] Mount, D. W. (2004). *Bioinformatics: Sequence and Genome Analysis* (2nd ed.). Cold Spring Harbor Laboratory Press.

[4] Durbin, R., Eddy, S. R., Krogh, A., & Mitchison, G. (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

[5] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., & Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3), 403-410.

[6] Pearson, W. R. (2013). An introduction to sequence similarity ("homology") searching. *Current Protocols in Bioinformatics*, 42(1), 3-1.

[7] Python Software Foundation. (2025). Python Language Reference, version 3.14. Retrieved from https://www.python.org

[8] NCBI Resource Coordinators. (2024). Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*.