

Ejercicio 3: Alineación de Secuencias de Nucleótidos

Algoritmo de Needleman-Wunsch

Fabrizio Denardi

Diciembre 2025

1. Parte 1: Conceptos Teóricos

1.1. Secuencias de Nucleótidos y su Importancia

Las secuencias de nucleótidos se refieren a cadenas de bases que forman el ADN o ARN de cualquier organismo vivo. En el ADN se encuentran cuatro bases nitrogenadas principales: adenina (A), timina (T), guanina (G) y citosina (C). Por su parte, el ARN es bastante parecido, solo que la timina es sustituida por uracilo (U). Estas secuencias no son solo combinaciones aleatorias, sino que guardan toda la información genética necesaria para producir proteínas y mantener funcionando las células.

La comparación de estas secuencias resulta fundamental en biología molecular por varios motivos:

- Cuando dos organismos tienen secuencias parecidas, es muy probable que compartan un ancestro común. Esto permite armar árboles evolutivos que muestran cómo están relacionadas las especies.
- Si se encuentran genes similares en organismos distintos, se puede predecir qué función tienen incluso sin haberlos estudiado directamente.
- En medicina, comparar secuencias ayuda a detectar mutaciones que causan enfermedades hereditarias.
- Para la biotecnología moderna, esto es clave: desde diseñar los primers que se usan en PCR hasta desarrollar nuevas vacunas o terapias génicas.

1.2. Alineación de Secuencias

Alinear secuencias significa organizarlas de tal manera que se puedan identificar qué partes se parecen entre sí. No es simplemente ponerlas una al lado de la otra, sino encontrar la mejor forma de disponerlas para ver patrones que indiquen algo sobre su relación evolutiva o funcional.

1.2.1. Tipos de Alineación

Alineación Global: Se comparan las secuencias completas, desde el primer nucleótido hasta el último. Funciona mejor cuando las secuencias tienen tamaños parecidos y se encuentran relacionadas en toda su longitud. El algoritmo de Needleman-Wunsch, que es el que se usa en este trabajo, hace precisamente esto.

Alineación Local: A veces no interesa comparar todo, sino buscar regiones específicas que sean similares dentro de secuencias más largas. Es útil cuando se sabe que solo ciertos dominios o fragmentos son parecidos. Para esto existe el algoritmo de Smith-Waterman.

1.2.2. Elementos de un Alineamiento

- Coincidencias (matches): son posiciones donde los nucleótidos de ambas secuencias son exactamente iguales. Cuando se observan muchas coincidencias, generalmente significa que esa región se ha conservado bastante bien a lo largo de la evolución, probablemente porque es importante funcionalmente.
- Desajustes (mismatches): son posiciones donde los nucleótidos difieren. Podría tratarse de mutaciones puntuales que ocurrieron en algún momento de la historia evolutiva de estas secuencias. No siempre son "malas", a veces estas diferencias son las que generan diversidad.
- Huecos (gaps): representan lugares donde parece que se insertó o eliminó algo en una de las secuencias respecto a la otra. Se introducen estratégicamente durante el alineamiento para que todo encaje mejor. En términos evolutivos, reflejan eventos de inserción o eliminación de nucleótidos que pueden haber ocurrido hace millones de años.

1.3. Modelo de Puntuación

Para decidir qué tan bueno es un alineamiento, se necesita alguna forma de "calificarlo". Aquí es donde entra el modelo de puntuación, que básicamente asigna valores a cada tipo de evento que puede ocurrir en el alineamiento.

1.3.1. Reglas de Puntuación Básicas

- Match: cuando dos nucleótidos coinciden, se asignan puntos positivos. En el caso simple, se usa +1.
- Mismatch: si no coinciden, hay que penalizar. Se resta -1.

Ahora bien, esto es el esquema más básico. Cuando se trabaja con proteínas o se requiere algo más sofisticado, se pueden usar matrices especiales (como BLOSUM o PAM) que consideran cuán probable es evolutivamente cada tipo específico de mutación.

1.3.2. Penalización por Huecos

Los gaps son un tema aparte. Si no se penalizaran lo suficiente, el algoritmo podría llenar el alineamiento de huecos por todas partes, lo cual no tendría mucho sentido biológico:

- Penalización por apertura de gap: es el costo que se paga al crear un nuevo hueco. En el modelo simple, se usa -2.
- Penalización por extensión de gap: algunos algoritmos más avanzados cobran extra por cada posición adicional que se extiende un gap. La fórmula típica es $\text{penalty} = d + e \times k$, donde d es abrir el gap, e es extenderlo, y k es qué tan largo es.

Para simplificar, en la implementación cada gap cuesta -2 fijo, sin importar cuánto mida. Esto hace las cosas más sencillas, aunque se sacrifica un poco de precisión biológica.

1.4. Algoritmo de Needleman-Wunsch

El algoritmo de Needleman-Wunsch apareció en 1970 y fue revolucionario para la bioinformática. Usa una técnica llamada programación dinámica que básicamente significa que va resolviendo el problema grande dividiéndolo en pedacitos más pequeños. Lo importante es que garantiza encontrar el mejor alineamiento global posible entre dos secuencias, no solo uno bueno”, sino el óptimo.

1.4.1. Construcción de la Matriz de Programación Dinámica

Lo primero que hace el algoritmo es crear una matriz M de tamaño $(n + 1) \times (m + 1)$, donde n y m son las longitudes de las dos secuencias.

Inicialización:

- Se coloca $M[0, 0] = 0$, que representa alinear dos secuencias vacías (obviamente cuesta 0)
- La primera fila se llena con: $M[0, j] = j \times \text{gap_penalty}$ para $j = 1, \dots, m$
- Lo mismo con la primera columna: $M[i, 0] = i \times \text{gap_penalty}$ para $i = 1, \dots, n$

Esto tiene sentido: alinear una secuencia con ”nada” solo puede hacerse metiendo gaps, y hay que pagar el precio por cada uno.

1.4.2. Ecuación de Recurrencia

Para llenar cada casilla $M[i, j]$, el algoritmo evalúa tres opciones y se queda con la que dé el mejor puntaje:

$$M[i, j] = \max \begin{cases} M[i - 1, j - 1] + s(x_i, y_j) & \text{(diagonal: match/mismatch)} \\ M[i - 1, j] + \text{gap} & \text{(arriba: gap en seq2)} \\ M[i, j - 1] + \text{gap} & \text{(izquierda: gap en seq1)} \end{cases} \quad (1)$$

donde $s(x_i, y_j)$ es simplemente el puntaje que obtenemos al emparejar x_i con y_j :

$$s(x_i, y_j) = \begin{cases} \text{match} & \text{si } x_i = y_j \\ \text{mismatch} & \text{si } x_i \neq y_j \end{cases} \quad (2)$$

Este enfoque implementa algo conocido como el ”principio de optimalidad de Bellman”, que básicamente dice que si construyes la mejor solución usando las mejores soluciones de problemas más chicos, al final obtienes lo óptimo para el problema completo. Medio recursivo conceptualmente, pero funciona perfecto.

1.4.3. Proceso de Traceback

Una vez construida la matriz completa, el score óptimo se encuentra en $M[n, m]$. Para recuperar el alineamiento que produjo ese score:

1. Inicio: comenzar en la celda $M[n, m]$ (esquina inferior derecha)

2. Recorrido: en cada paso, determinar de qué celda provino el valor actual:
 - Si $M[i, j]$ vino de $M[i - 1, j - 1]$: alinear x_i con y_j (movimiento diagonal)
 - Si $M[i, j]$ vino de $M[i - 1, j]$: alinear x_i con gap (movimiento vertical)
 - Si $M[i, j]$ vino de $M[i, j - 1]$: alinear gap con y_j (movimiento horizontal)
3. Terminación: continuar hasta llegar a $M[0, 0]$
4. Reversión: como el alineamiento se construyó de atrás hacia adelante, invertirlo para obtener el orden correcto

1.4.4. Complejidad Computacional

En términos de eficiencia, el algoritmo funciona así:

- Tiempo de ejecución: $O(n \times m)$ para armar toda la matriz
- Memoria que necesita: $O(n \times m)$ para guardarla

Si se trabaja con secuencias enormes (por ejemplo genomas completos), existen algunos trucos de optimización que permiten reducir el uso de memoria a $O(\min(n, m))$. Se sacrifica un poco de conveniencia pero se ahorra espacio.

2. Parte 2: Implementación del Algoritmo

2.1. repositorio

El repositorio del código puede encontrarse en: https://github.com/denardifabricao/MIA_01c_CAED/tree/main/Ejercicio3a

2.2. Descripción General del Código

El código completo puede revisarse en el archivo `needleman_wunsch.py` en el repositorio. A continuación se describe cómo está organizado:

Se implementó el algoritmo de Needleman-Wunsch en Python de forma modular, con varias funciones que se encargan de diferentes partes del proceso:

2.2.1. Función `needleman_wunsch()`

Esta es el corazón del programa. Aquí es donde ocurre el procesamiento principal del algoritmo:

1. Primero se crea una matriz del tamaño necesario y se inicializa con las penalizaciones de gaps en los bordes, como se explicó anteriormente.
2. Después viene el llenado: se recorre cada celda calculando el mejor puntaje posible según las tres opciones disponibles (diagonal, arriba o izquierda).
3. Una vez llena la matriz, se hace el traceback para recuperar cómo quedó el alineamiento óptimo. Se retrocede desde la última celda hasta el principio.

4. Finalmente se devuelve todo: la matriz completa, las dos secuencias alineadas (con sus gaps incluidos) y el puntaje final.

A continuación se muestran algunos fragmentos del código:

```
1 # Crear matriz de dimensiones (n+1) x (m+1)
2 n_rows = len(seq1)
3 n_cols = len(seq2)
4 matriz = [[0 for _ in range(n_cols + 1)]
5           for _ in range(n_rows + 1)]
6
7 # Inicializar primera fila y columna con penalizaciones
8 for i in range(n_rows + 1):
9     matriz[i][0] = i * gap
10 for j in range(n_cols + 1):
11     matriz[0][j] = j * gap
```

Listing 1: Inicialización de la matriz y condiciones de frontera

Código de llenado de la matriz con programación dinámica:

```
1 for i in range(1, n_rows + 1):
2     for j in range(1, n_cols + 1):
3         # Calcular score diagonal (match o mismatch)
4         if seq1[i-1] == seq2[j-1]:
5             diagonal_score = matriz[i-1][j-1] + match
6         else:
7             diagonal_score = matriz[i-1][j-1] + mismatch
8
9         # Calcular scores con gaps
10        up_score = matriz[i-1][j] + gap        # Gap en seq2
11        left_score = matriz[i][j-1] + gap      # Gap en seq1
12
13        # Tomar el maximo (decision optima)
14        matriz[i][j] = max(diagonal_score, up_score,
15                           left_score)
```

Listing 2: Ecuación de recurrencia - Llenado de la matriz

Código del proceso de traceback:

```
1 alignment1 = []
2 alignment2 = []
3 i = n_rows
4 j = n_cols
5
6 while i > 0 or j > 0:
7     if j == 0:
8         # Solo gaps en seq2
9         alignment1.append(seq1[i-1])
10        alignment2.append('-')
11        i -= 1
12    elif i == 0:
13        # Solo gaps en seq1
14        alignment1.append('-')
```

```

15         alignment2.append(seq2[j-1])
16         j -= 1
17     else:
18         # Determinar de donde vino el valor actual
19         if seq1[i-1] == seq2[j-1]:
20             diagonal_score = matriz[i-1][j-1] + match
21         else:
22             diagonal_score = matriz[i-1][j-1] + mismatch
23
24         up_score = matriz[i-1][j] + gap
25         left_score = matriz[i][j-1] + gap
26
27         # Elegir el camino que produjo el valor
28         if matriz[i][j] == diagonal_score:
29             alignment1.append(seq1[i-1])
30             alignment2.append(seq2[j-1])
31             i -= 1
32             j -= 1
33         elif matriz[i][j] == up_score:
34             alignment1.append(seq1[i-1])
35             alignment2.append('-')
36             i -= 1
37         else:
38             alignment1.append('-')
39             alignment2.append(seq2[j-1])
40             j -= 1
41
42 # Invertir los alineamientos
43 alignment1 = ''.join(reversed(alignment1))
44 alignment2 = ''.join(reversed(alignment2))

```

Listing 3: Traceback para recuperar el alineamiento óptimo

2.2.2. Funciones de Visualización

Para facilitar la visualización, se implementaron algunas funciones auxiliares:

- `print_matrix()`: muestra la matriz de puntuación de una forma clara y ordenada, con los encabezados de las secuencias para que sea más fácil interpretarla.
- `print_alignment()`: muestra el alineamiento con símbolos visuales intuitivos:
 - Un `|` cuando hay un match perfecto
 - Un `*` para los mismatches
 - Un espacio en blanco donde hay gaps

Así de un vistazo se puede ver qué tan bien alinearon las secuencias.

- `analyze_alignment()`: hace el análisis cuantitativo. Cuenta cuántos matches, mismatches y gaps hay, y calcula el porcentaje de identidad entre las secuencias.

Código para análisis de estadísticas:

```

1 num_matches = 0
2 num_mismatches = 0
3 num_gaps = 0
4
5 for i in range(len(alignment1)):
6     if alignment1[i] == '-' or alignment2[i] == '-':
7         num_gaps += 1          # Gap
8     elif alignment1[i] == alignment2[i]:
9         num_matches += 1       # Match
10    else:
11        num_mismatches += 1     # Mismatch
12
13 longitud_total = len(alignment1)
14 porcentaje_identidad = (num_matches / longitud_total) * 100
15
16 print(f"Longitud del alineamiento: {longitud_total}")
17 print(f"Coincidencias (matches): {num_matches}")
18 print(f"Desajustes (mismatches): {num_mismatches}")
19 print(f"Huecos (gaps): {num_gaps}")
20 print(f"Identidad: {porcentaje_identidad:.2f}%")

```

Listing 4: Análisis estadístico del alineamiento

2.2.3. Esquema de Puntuación Implementado

Se usa un modelo de puntuación bastante directo y simple:

- Match: +1 (se asigna un punto cuando coinciden)
- Mismatch: -1 (se resta uno cuando no coinciden)
- Gap: -2 (se penaliza más los huecos)

Con este esquema se indica que se prefiere matchear los nucleótidos cuando sea posible, pero si hay que elegir entre un mismatch y un gap, se prefiere el mismatch. Esto tiene sentido biológicamente porque las mutaciones puntuales (sustituciones) suelen ser más comunes que las inserciones o eliminaciones grandes en secuencias relacionadas evolutivamente.

Código de definición de parámetros:

```

1 def needleman_wunsch(seq1, seq2, match=1, mismatch=-1, gap=-2)
2     :
3     """
4     Implementa el algoritmo de Needleman-Wunsch
5
6     Parametros:
7     - match: +1 (recompensa por coincidencia)
8     - mismatch: -1 (penalizacion por desajuste)
9     - gap: -2 (penalizacion por hueco)
10    """

```

```
10 # ... implementacion del algoritmo ...
```

Listing 5: Parámetros del modelo de puntuación

Ejemplo de uso del programa:

```
1 # Definir parejas de secuencias a analizar
2 parejas_secuencias = [
3     ("GATTACA", "GCATGCU"),      # Caso clasico
4     ("ACGT", "ACCT"),           # Secuencias cortas
5     ("ATGCT", "AGCT"),          # Requiere gaps
6 ]
7
8 # Parametros de puntuacion
9 MATCH = 1
10 MISMATCH = -1
11 GAP = -2
12
13 # Procesar cada pareja
14 for seq1, seq2 in parejas_secuencias:
15     matriz, alineamiento1, alineamiento2, puntaje = \
16         needleman_wunsch(seq1, seq2, MATCH, MISMATCH, GAP)
17
18     # Mostrar resultados
19     print_matrix(matriz, seq1, seq2)
20     print_alignment(alineamiento1, alineamiento2, seq1, seq2)
21     analyze_alignment(alineamiento1, alineamiento2,
22                       MATCH, MISMATCH, GAP)
```

Listing 6: Ejemplo de ejecución con parejas de secuencias

2.3. Casos de Prueba

Para probar el programa, se prepararon 10 parejas diferentes de secuencias. Cada una representa un escenario distinto:

1. ("GATTACA", "GCATGCU"): este es el ejemplo clásico que aparece en la literatura - tiene varios mismatches interesantes
2. (.^ACGT", .^ACCCT"): secuencias cortas y simples, solo un mismatch
3. (.^ATGCT", .^AGCT"): se necesita introducir un gap para obtener un buen alineamiento
4. Se incluyen casos con secuencias de distintos largos, algunas muy parecidas, otras que necesitan varios gaps, etc.

Para cada pareja, el programa muestra información detallada:

- Toda la matriz de puntuación (para poder observar cómo razona el algoritmo)
- El alineamiento óptimo encontrado
- Todas las estadísticas: matches, mismatches, gaps, porcentaje de identidad
- El puntaje final que obtuvo ese alineamiento

2.4. Características Técnicas

- Lenguaje: Python 3
- Complejidad temporal: $O(n \times m)$ donde n y m son las longitudes de las secuencias
- Complejidad espacial: $O(n \times m)$ para almacenar la matriz completa
- Documentación: el código incluye comentarios extensos explicando la teoría detrás de cada paso
- Modularidad: funciones separadas para diferentes aspectos del análisis, facilitando mantenimiento y reutilización

2.5. Ejemplo de Resultado

A continuación se muestra el resultado de alinear ("GATTACA", "GCATGCU"):

Alineamiento óptimo:

Seq1: GATTACA

 |**|*|*

Seq2: GCATGCU

Estadísticas:

- Longitud: 7
- Matches: 3 (+3 puntos)
- Mismatches: 4 (-4 puntos)
- Gaps: 0 (+0 puntos)
- Identidad: 42.86%
- Puntaje final: -1

Se puede observar que aunque comparten algunas posiciones (las que tienen el |), hay bastantes diferencias (los *). El puntaje negativo indica que, en general, estas secuencias no son muy similares. Un 42.86 % de identidad es relativamente bajo - en biología, secuencias con menos del 50 % de identidad generalmente no se consideran homólogos.

3. Referencias

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. Link: https://web.stanford.edu/class/sbio228/public/readings/Bioinformatics_I_Lecture6/Needleman_Wunsch_JMB_70_Global_alignment.pdf?utm_source=chatgpt.com
- [2] Anandakumar, M. (2020). *Needleman-Wunsch algorithm for DNA sequence alignment*. Medium. Link: <https://medium.com/@amithunjha/needleman-wunsch-algorithm-for-dna-sequence-alignment-b103b8454de0>

- [3] Wikipedia contributors. (2025). *Needleman–Wunsch algorithm*. In Wikipedia, The Free Encyclopedia. Link: https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm