TJAA Vol. **000**, *Issue* 0, *p.*(0000). *Research Article*

Alineamiento de secuencias en Rstudio

Isaías Pérez Ramírez, Hugo Francisco Rincón López, Rudesindo Acuña Sánchez
Universidad Nacional de Colombia - sede La Paz
isperezr@unal.edu.co, hufrinconlo@unal.edu.co, ruacunas@unal.edu.co

3 July 2022

Abstract
En 1970 los laboratorios Bell crearon un lenguaje de programación llamado S, que tenía como propósito brindar un enfoque alternativo y más interactivo en el campo de análisis de datos. Más adelante, dicho lenguaje derivó a lo que hoy se conoce como len- guaje de programación R. Este entorno de programación es una poderosa herramienta para análisis estadístico y creación de gráficos. R tiene la ventaja de ser gratis, tener numerosos paquetes de mucha ayuda para múltiples cálculos y tener en su página web muchas guías y ayudas para la correcta implementación de sus comandos. Ahora bien, en los últimos años, R ha incursionado fuertemente en el campo de la bioinformática. Dicha tecnología se encarga de la aprovechar el recurso computacional para almacenar, analizar y comparar datos relacionados como ácidos nucleicos, secuencias de proteínas y sobre todo alineamiento de secuencia, temática de interés en este documento

Key words: Bioinformatica– alineación de secuencias– Rstudio.

1 Introducción

Desde hace mucho, la ciencia ha reunido esfuerzos para darle un sentido numérico a componentes como los ácidos nucleicos y las secuencias de las proteínas. A raíz de eso, es necesario una tecnología que ayude al análisis de estos datos biológicos con el fin de predecir formas de proteínas, estudiar las secuencias de ADN y ARN, y alineamiento de secuencias. Es aquí donde surge la bioinformática, la cual se conoce como la aplicación de la tecnología computacional para almacenar, recuperar y análisis de datos biológicos.

El alineamiento de secuencias es una rama de la bioinformática relacionada comparar la similitud entre dos o más secuencias biológicas. Para el alineamiento de secuencias biológicas, el primer lenguaje de programación usado fue C y C++, sin embargo, con el paso del tiempo se emplearon otros lenguajes como Python y R. El alineamiento de secuencias se compone de dos tipos: Global y local. En el global se intenta que el alineamiento cubra las dos secuencias completamente introduciendo los gaps que sean necesarios. En el local se alinean sólo las zonas más parecidas. En la actualidad existen muchos métodos para calcular la similitud entre secuencias. En este documento se analizará un código en Python para alineamiento entre dos secuencias, con el objetivo de crear uno parecido en Rstudio que cumpla con la misma función.

2 Script para alineamiento de secuencias en python

Resolviendo el problema de alineación de secuencias en Python. Tomado de

https://johnlekberg.com/blog/2020-10-25-seq-align.html

Por John Lekberg Octubre 25, 2020. Adaptado a Colab por Jose Francisco Ruiz-Munoz

2.1 Enunciado del problema

Como entrada le son dadas dos secuencias. Ejemplo:“CAT“ y “CT“ (La secuencia puede ser de tipo string u otros arreglos de datos.) Como salida, su objetivo es producir un alineamiento, que se empareje con elementos de la secuencia. Ejemplo:

C - C
A - T
T -

Una alineación puede tener brechas. Ejemplo

C - C
A -
T - T

Si bien el alineamiento puede tener huecos, no puede cambiarse el orden relativo de la secuencia de los elementos. E.g. “CT“ no se puede cambiar por “TC“. Especificamente, su objetivo es producir un alineamiento con el puntaje máximo. A continuación se tiene como calcular el la puntuación:

a. Score = 0
b. Mirar cada par de elementos:
   i. si hay un espacio, entonces score -1.
   ii. si los elementos son los mismos, entonces score +1.
   iii. si los elementos son diferentes, entonces score -1.

Ej: Esta alineación tiene un score de -1:

C - C (igual, +1)
A - T (diferente, -1)
T - (espacio, -1)

Pero esta alineación tiene un score de +1:

C - C (igual, +1)
A - (espacio, -1)
T - T (igual, +1)

La meta es tomar dos secuencias diferentes y encontrar un alineamiento con el score máximo.

2.2 Cómo se representa la información del problema.

Para la entrada, se representan las secuencias como cadenas o listas. Ejemplo las secuencias pueden ser una cadena "CAT" o una lista ["C", "A", "T"]. Cualquier cosa que implemente collections.abc.Sequence (no solo listas y cadenas) servirá.

Para la salida se representa el alineamiento como una lista de tuplas de los indices (o None cuando hay un espacio), por ejemplo este alineamiento

C - C
A - T
T -

sería representado como

[(0, 0), (1, 1), (2, None)]

Y esta alineación:
C - C
A -
T - T

sería representada como

[(0, 0), (1, None), (2, 1)]

2.3 Creando una solucion por fuerza bruta

Me gusta comenzar con soluciones por fuerza bruta cuando trabajo en un problema. Este tipo de soluciones tienden a ser más sencillas de implementar y, muchas veces, es suficiente para las entradas que se van a encontrar. Empiezo creando una funcion que toma dos rangos de indices e itera todos los alineamientos posibles.

```
    from collections import deque


def all_alignments(x, y):


    """Retorna un iterable con todos los
    alineamientos de las dos secuencias.

    x, y -- Secuencias.
    """

    def F(x, y):
        """Una funcion de ayuda que recursivamente
        crea los alineamientos.

        x, y -- Secuencia de indices para los x y
        y originales.
        """
        if len(x) == 0 and len(y) == 0:
            yield deque()

        scenarios = []
        if len(x) > 0 and len(y) > 0:
         scenarios.append((x[0], x[1:], y[0], y[1:]))
        if len(x) > 0:
         scenarios.append((x[0], x[1:], None, y))
        if len(y) > 0:
         scenarios.append((None, x, y[0], y[1:]))

        # NOTA: "xh" y "xt" hacen referencia a
        # "x-head" y "x-tail", con "head" siendo
        # el frente o cabeza de la secuencia, y
        # "tail" siendo el resto de la secuencia.
        # De igual forma para "yh" and "yt".

        for xh, xt, yh, yt in scenarios:
            for alignment in F(xt, yt):
                alignment.appendleft((xh, yh))
                yield alignment

    alignments = F(range(len(x)), range(len(y)))
    return map(list, alignments)
```

(El código usa: collections.deque, generator function, range, len, map.)

Y aquí están todos los arreglos para "'CAT"' y "CT"

```
[[(0, 0), (1, 1), (2, None)],
[(0, 0), (1, None), (2, 1)],
[(0, 0), (1, None), (2, None), (None, 1)],
[(0, 0), (1, None), (None, 1), (2, None)],
[(0, 0), (None, 1), (1, None), (2, None)],
[(0, None), (1, 0), (2, 1)],
[(0, None), (1, 0), (2, None), (None, 1)],
[(0, None), (1, 0), (None, 1), (2, None)],
[(0, None), (1, None), (2, 0), (None, 1)],
[(0, None), (1, None), (2, None),
(None, 0), (None, 1)],
[(0, None), (1, None), (None, 0), (2, 1)],
[(0, None), (1, None), (None, 0), (2, None),
                              (None, 1)],
[(0, None), (1, None), (None, 0), (None, 1),
(2, None)],
[(0, None), (None, 0), (1, 1), (2, None)],
[(0, None), (None, 0), (1, None), (2, 1)],
[(0, None), (None, 0), (1, None), (2, None),
                                (None, 1)],
[(0, None), (None, 0), (1, None), (None, 1),
(2, None)],
[(0, None), (None, 0), (None, 1), (1, None),
(2, None)],
[(None, 0), (0, 1), (1, None), (2, None)],
[(None, 0), (0, None), (1, 1), (2, None)],
[(None, 0), (0, None), (1, None), (2, 1)],
[(None, 0), (0, None), (1, None), (2, None),
(None, 1)],
[(None, 0), (0, None), (1, None), (None, 1),
(2, None)],
[(None, 0), (0, None), (None, 1), (1, None),
(2, None)],
[(None, 0), (None, 1), (0, None), (1, None),
(2, None)]]
```

Y aquí una forma más legible de la salida (los guiones "-" indicas un espacio

```
def print_alignment(x, y, alignment):
    print("".join(
     "-" if i is None else x[i] for i,_ in alignment
```

[h]

```
    CAT    CAT    CAT-   CA-T   C-AT
    CT-    C-T    C–T    C-T-   C-T-
    C-AT   CAT    CAT-   CA-T   CAT-
    CT–    -CT    -C-T   -CT-   –CT
    CA-T-  CA–T   C-AT   CAT–   CAT-
    –C-T   –CT-   -CT-   —CT   –CT
```

```
))
print("".join(
 "-" if j is None else y[j] for _,j in alignment
))

x = "CAT"
y = "CT"
for alignment in all_alignments(x, y):
    print_alignment(x, y, alignment)
    print()
```

Se muestran algunos de los alineamientos obetnidos

A continuación creo una función ue tome las dos secuencias y un alineamiento para producir una score

```
def alignment_score(x, y, alignment):
"""Puntuar un alineamiento.

x, y -- secuencias.
alignment -- un alineamiento de x y y.
"""
score_gap = -1
score_same = +1
score_different = -1

score = 0
for i, j in alignment:
    if (i is None) or (j is None):
        score += score_gap
    elif x[i] == y[j]:
        score += score_same
    elif x[i] != y[j]:
        score += score_different

return score
```

Consideremos un alineamiento.

```
C - C
A -
T - T
```

Se le calcula el score

```
x = "CAT"
y = "CT"
alignment = [(0, 0), (1, None), (2, 1)]
alignment_score(x, y, alignment)
1
```

Con estas dos funciones el método por fuerza bruta buscará el alineamiento con mayor score

```
from functools import partial


def align_bf(x, y):
    """Align two sequences, maximizing the
    alignment score, using brute force.

    x, y -- sequences.
    """
    return max(
        all_alignments(x, y),
        key=partial(alignment_score, x, y),
    )
```

El código utiliza: functools.partial, max.

```
align_bf("CAT", "CT")

[(0, 0), (1, None), (2, 1)]

print_alignment("CAT", "CT", align_bf("CAT", "CT"))

CAT
C-T
```

Cuál es la complejidad en tiempo para esta solución? Para dos secuencias de n y m elementos

a. el número de alineamientos posibles D es dado por la relación de recurrencia

$$D(n,0) = D(0,m) = 1$$
$$D(n,m) = D(n-1,m) + D(n,m-1) + D(n-1,m-1)$$

Para hacerse una idea de cuánto crece este número

```
from functools import lru_cache


@lru_cache(maxsize=None)
def D(n, m):
    if n == 0 or m == 0:
        return 1
    else:
        return D(n - 1, m) + D(n, m - 1)
        + D(n - 1, m - 1)
```

a. Hay 3 alineamientos posibles para 2 secuencias de 1 elemento cada una

```
D(1, 1)
3
```

b. Hay 8,097,453 alineamientos posibles para dos secuencias de 10 elementos cada una

```
D(10, 10)
8097453
```

c. Hay 2.05e+75 alineamientos posibles para dos secuencias de 100 elementos

```
D(100, 100)
205371683087241577022877800627197112O3...
```

D(100,100) se acerca al número de Eddington , 10e+80 –el número estimado de átomos de hidrógeno en el universo observable.

(Consultar OEIS A001850 para más saber information sobre los números de Delannoy de la forma D(n,n).)

El cálculo de la puntuación de la alineación toma un tiempo lineal en los tamaños de ambas secuencias: O(n + m). Como resultado, la complejidad de tiempo total de la solución por fuerza brutal es:

O(D(n, m) × (n + m))

2.4 Creando una solución más eficiente

La solución por fuerza bruta es simple, pero no escala bien. En la práctica, la alineación de secuencias es usada para analizar la secuencias de datos biológicos (ej: secuencias de ácidos nucléicos). Dado que el tamaño de estas secuencias pueden ser de cientos o miles de elementos de longitud, no hay forma que la solución por fuerza bruta funcione para datos de este tamaño.

En 1970, Saul B. Needleman and Christian D. Wunsch crearon un algoritmo más veloz para resolver este problema: el algoritmo de Needleman-Wunsch. (consultar "A general method applicable to the search for similarities in the amino acid sequence of two proteins", https://doi.org/10.1016/0022-2836(70)90057-4.). El algoritmo usa programación dinámica para resolver el problema de alineación de secuencias en tiempo O(mn).

Aquí hay una implementación en Python del algotimo de Needleman-Wunsch, basada en la sección 3 de "Parallel Needleman-Wunsch Algorithm for Grid":

```
from itertools import product
from collections import deque


def needleman_wunsch(x, y):
    """Corre el algoritmo de Needleman-Wunsch
    en dos secuencias.

    x, y -- secuencias.

    Cödigo basado en el pseudocódig en la seccion
    3 de:

    Naveed, Tahir; Siddiqui, Imitaz Saeed;
    Ahmed, Shaftab.
    "Parallel Needleman-Wunsch Algorithm
    for Grid." n.d.
    https://upload.wikimedia.org/wikipedia/
    en/c/c4/ParallelNeedlemanAlgorithm.pdf
    """
    N, M = len(x), len(y)
    s = lambda a, b: int(a == b)

    DIAG = -1, -1
    LEFT = -1, 0
    UP = 0, -1

    # Create tables F and Ptr
    F = {}
    Ptr = {}

    F[-1, -1] = 0
    for i in range(N):
        F[i, -1] = -i
    for j in range(M):
        F[-1, j] = -j

    option_Ptr = DIAG, LEFT, UP
    for i, j in product(range(N), range(M)):
        option_F = (
            F[i - 1, j - 1] + s(x[i], y[j]),
            F[i - 1, j] - 1,
            F[i, j - 1] - 1,
        )
        F[i, j], Ptr[i, j] = max(zip(option_F,
                                     option_Ptr))

    # Work backwards from (N - 1, M - 1) to (0, 0)
    # to find the best alignment.
    alignment = deque()
    i, j = N - 1, M - 1
    while i >= 0 and j >= 0:
        direction = Ptr[i, j]
        if direction == DIAG:
            element = i, j
        elif direction == LEFT:
            element = i, None
        elif direction == UP:
            element = None, j
        alignment.appendleft(element)
        di, dj = direction
        i, j = i + di, j + dj
    while i >= 0:
        alignment.appendleft((i, None))
        i -= 1
    while j >= 0:
        alignment.appendleft((None, j))
        j -= 1

    return list(alignment)
```

El código utiliza: collections.deque, itertools.product, zip, list

```
needleman_wunsch("CAT", "CT")

[(0, 0), (1, None), (2, 1)]
```

Por ende el algoritmo más rápido llamará needleman_wunsch

```
def align_fast(x, y):
"""Alinea dos secuencias maximizando el score
utilizando el algoritmo de Needleman-Wunsch

x, y -- secuencias.
"""
return needleman_wunsch(x, y)


align_fast("CAT", "CT")

[(0, 0), (1, None), (2, 1)]

print_alignment("CAT", "CT", align_fast("CAT", "CT"))

CAT

C-T
```

La complejidad en tiempo de esta solucion, para dos secuencias de n y m elementos, es:

a. Crear tablas **F** y **Ptr** toma O(mn) tiempo
b. Crear el alineamiento al navegar desde **Ptr[N-1,M-1]** hasta **Ptr[0,0]** toma

$$O(n+m)$$

tiempo

2.5 En conclusión

En el post de esta semana aprendieron a resolver el problema de "alineamiento de secuencias". Aprendieron como crear una solución por fuerza bruta que genera todos los alineamientos posibles. Luego aprendieron que esta solucion es inutil para secuencias largas, dos secuencias de 10 elementos tienen 8 millones de alineamientos diferentes!. Finalmente aprendieron a implementar el algoritmo de Needleman-Wunsch en python.

Mi desafío para ti:

Modificar la función needleman_wunsch para que tome los siguientes parametros para encontrar la puntuación

- score_gap cómo puntuar un espacio, por defecto -1
- score_same cómo puntuar elementos iguales, por defencto +1
- score_different cómo puntuar elementos diferentes, por defecto -1

3 Analisis y script para alineamiento en Rstudio

Para la implementación del alineamiento de secuencias en Rstudio se decidió adaptar las funciones alignment_score y print_alignment del cuaderno de Colab a R.

La función alignment_score permite calcular la puntuación de la alineación con el puntaje más alto de acuerdo a las dos secuencias ingresadas. Dicha función recibe cuatro parámetros:

x es un string y es la primera secuencia.

y es un string y es la segunda secuencia.

pxy es la penalización por no coincidir con los caracteres de x e y. En este caso puede ser asignado, pero se determina como 1 por defecto.

pgap es la penalización en el caso de que haya un un espacio (representado por "-") en la alineación.

```
alignment_score = function(x, y, pxy, pgap){

  # inicializando variables
  i = 0
  j = 0

  # Separando los strings por caracteres
  x = (as.vector(str_split_fixed(x, pattern = "",
                                 n = nchar(x))))
  y = (as.vector(str_split_fixed(y, pattern = "",
                                 n = nchar(y))))

  # Tamaño de los strings
  m = length(x)
  n = length(y)

  # Inicializando matriz donde estarán
  # almacenadas las respuestas óptimas
  dp = (matrix(0, m+1, n+1))

  # Se toma toda la fila y se llena
  dp[1, ] =  pgap * c(0:(n))
  # Se toma toda la columna y se llena
  dp[, 1] =  pgap * c(0:(m))

  # Calculando el score más alto
  i = 2
  while (i < m+2){
    j = 2
    while (j < n+2){
      if (x[i-1] == y[j-1]){
        dp[i, j] = dp[(i - 1),(j - 1)]
      }else{
        dp[i, j] = min(dp[(i - 1),(j - 1)] + pxy,
                       dp[(i - 1), j] + pgap,
                       dp[i, (j - 1)] + pgap)
      }
      j = j + 1
    }
    i = i + 1
  }
  score = max(length(x), length(y))
                - 2*dp[m+1, n+1]
  newList <- list("m" = m, "n" = n,
  "score" = score, "list" = dp, "x" = x,
  "y" = y, "pxy" = pxy, "pgap" = pgap)
 return(newList)
}
```

Con el objetivo de poder retornar más de un dato en una función es R es necesario crear un elemento llamado newList para almacenar todos los datos que se requieren tener para ser usados en otras funciones.

En la función anterior es posible tomar el valor almacenado en score que es calculado de acuerdo a las fallas que se han tenido en la alineación al saber cual de las dos secuencias óptimas es la más larga.

Como en la función print_alignment se toma la alineación de acuerdo al Unicode de las cadenas ingresadas, se hace necesario el uso de la siguiente función para conocer el código decimal de cada caracter extraido de las secuencias

```
# Function para convertir binario a decimal
binaryToDecimal = function(n)
{
  n = as.integer(pryr::bits(n))
  num = n
  dec_value = 0

  # Initializing base value to 1, i.e 2^0
  base = 1

  temp = num;
  while (temp > 0) {
    last_digit = temp %% 10
    temp = as.integer(temp / 10)

    dec_value =  dec_value + last_digit * base

    base = base * 2
  }

  return(dec_value)
```

```
}
```

Ahora bien, la función print_alignment ejecuta las instrucciones para determminar la alineación con la puntuación más alta. Dependiendo de los casos en los que la implementación haya ejecutado las intrucciones para conocer la puntuación, print_alignment llenará con "-" o caracter alineado. Por último se manipulan los datos Unicode para poder dar la respuesta final de la óptima alineación según los datos tratados.

```
#Extrayendo el óptimo alineamiento
print_alignment = function(dp, m, n, x, y, pxy, pgap)
{
  l = m + n
  m
  i = m
  j = n

  xpos = l
  ypos = l
  l
  #Los alineamientos óptimos
  xans = replicate(l+1, 0)
  yans = replicate(l+1, 0)
  while (i > 0 | j > 0){
    if (x[i] == y[j]){
      xans[xpos] = binaryToDecimal(x[i])
      yans[ypos] = binaryToDecimal(y[j])
      xpos = xpos -1
      ypos = ypos -1
      i = i - 1
      j = j - 1
    }else if((dp[i, j] + pxy) ==
                (dp[(i+1), (j+1)])){
      xans[xpos] = binaryToDecimal(x[i])
      yans[ypos] = binaryToDecimal(y[j])
      xpos = xpos -1
      ypos = ypos -1
      i = i - 1
      j = j - 1
    }else if((dp[i, (j+1)] + pgap) ==
                dp[(i+1),(j+1)]){
      xans[xpos] = binaryToDecimal(x[i])
      yans[ypos] = binaryToDecimal('-')
      xpos = xpos -1
      ypos = ypos -1
      i = i - 1
    }else if((dp[(i+1),j] + pgap) ==
                dp[(i+1), (j+1)]){
      xans[xpos] = binaryToDecimal('-')
      yans[ypos] = binaryToDecimal(x[j])
      xpos = xpos -1
      ypos = ypos -1
      j = j - 1
    }
  }
  while(xpos > 0){
    if (i > 0){
      i = i - 1
      xans[xpos] = binaryToDecimal(x[i+1])
      xpos = xpos - 1
    }else{
      xans[xpos] = binaryToDecimal('-')
      xpos = xpos - 1
    }
  }
    while(ypos > 0){
      if (j > 0){
        j = j - 1
        yans[ypos] = binaryToDecimal(y[j+1])
        ypos = ypos - 1
      }else{
        yans[ypos] = binaryToDecimal('-')
        ypos = ypos - 1
      }
    }
  id = 1
  i = l
  while (i > 1){
    if ((intToUtf8(yans[i+1]) == '-') &
           intToUtf8(xans[i+i]) == '-'){
      id = i + 1
      break
    }
    i = i - 1
  }
  #Imprimiendo la respuesta final
  print("La alineación óptima es ")
  i = id
  x_seq = rep(NA, m)
  while (i < l-1){
   x_seq[i] = intToUtf8(xans[i+2])
    i = i + 1
  }
  print(x_seq)

  #Y
  i = id
  y_seq = rep(NA, n)
  while (i < l-1){
    y_seq[i] = intToUtf8(yans[i+2])
    i = i + 1
  }
  print(y_seq)
}
```

4 Conclusion

Con el pasar del tiempo se vuelve más relevante estudiar similitudes entre secuencias biológicas. Gracias a la tecnología y a los lenguajes de programación, la labor del análisis secuencial se torna más sencilla. Será necesario entonces la mejora paulatina de los principales algoritmos y la constante creación de paquetes estadísticos que arrojen resultados más certeros sobre los alineamientos de secuencias biológicas de grandes tamaños. En este trabajo se logró la creación de un código en R capaz de estudiar el alineamiento entre dos secuencias. El objetivo fue tomar un código inicial de Python y adaptarlo al lenguaje R. Al momento de comparar los códigos para alineamiento de secuencias biológicas tanto de Python como R, se llega al acuerdo de que es más eficiente y rápido trabajar con Python por su dinamismo.