Universdad Nacional de Colombia Sede De La Paz

Alineamiento de secuencias biologicas en R y Python

Programacion en Lenguajes Estadisticos

- Ronald Mateo Ceballos Lozano

---

1. Resolución del problema de alineación de secuencias en Python

Por John Lekberg el 25 de octubre de 2020.

El post de esta semana trata sobre la resolución del problema de .ªlineación de Secuencias".Aprenderás:

1. Cómo crear una solución de fuerza bruta.
2. Cómo crear una solución más eficiente utilizando el algoritmo Needleman-Wunsch
3. La programación dinámica

2. Planteamiento del problema

Como entrada, se le dan dos secuencias.
Por ejemplo

$$"CAT"$$

$$"CT"$$

(Las secuencias pueden ser cadenas u otras matrices de datos).
Como salida, su objetivo es producir un alineamiento, que empareje los elementos de la secuencia.
Por ejemplo

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

Un alineamiento puede tener huecos.
Por ejemplo

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

Aunque un alineamiento puede tener huecos, no puede cambiar el orden relativo de los elementos de la secuencia. Por ejemplo,

$$"CT"$$

no puede cambiarse por

$$"TC"$$

En concreto, su objetivo es producir un alineamiento con la máxima puntuación. Así es como se calcula la puntuación:

1. Puntuación = 0
2. Mire cada par de elementos:

2.1 Si hay una brecha, entonces la puntuación -= 1
2.2 De lo contrario, si los elementos son los mismos, entonces la puntuación += 1.
2.3 Si los elementos son diferentes, la puntuación es -= 1.

Por ejemplo, esta alineación tiene una puntuación de -1:

```
C - C   (same, +1)
A - T   (different, -1)
T -     (gap, -1)
```

Pero esta alineación tiene una puntuación de +1:

```
C - C   (same, +1)
A -     (gap, -1)
T - T   (same, +1)
```

Así que el objetivo es tomar dos secuencias y encontrar un alineamiento con la máxima puntuación.

3. ¿Cómo represento los datos del problema?

Para la entrada, represento las secuencias como cadenas o listas. Por ejemplo, puedo representar la secuencia CAT como

```
"CAT"
```

o como

```
["C", "A", "T"]
```

Realmente, cualquier cosa que implemente *collections.abc.Sequence* (no sólo cadenas y listas) funcionará.

Para la salida, represento un alineamiento como una lista de tuplas de índices (o Ninguno, si hay un hueco.) Por ejemplo, este alineamiento:

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

se representaría como

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

Y esta alineación:

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

se representaría como

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

4. Crear una solución de fuerza bruta

Me gusta empezar con soluciones de fuerza bruta cuando trabajo en problemas. Las soluciones de fuerza bruta tienden a ser más simples de implementar y, con bastante frecuencia, la solución de fuerza bruta es "suficientemente buena"para las entradas reales que se encontrarán.

Empiezo creando una función que toma dos rangos de índices e itera sobre todas las alineaciones posibles:

```
from collections import deque


def all_alignments(x, y):
    """Return an iterable of all alignments of two
    sequences.

    x, y -- Sequences.
    """

    def F(x, y):
        """A helper function that recursively builds the
        alignments.

        x, y -- Sequence indices for the original x and y.
        """
        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:]))

        # NOTE: "xh" and "xt" stand for "x-head" and "x-tail",
        # with "head" being the front of the sequence, and
        # "tail" being the rest of the sequence. Similarly for
        # "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)
```

(Este código utiliza: *collections.deque*, función generadora, range, len, map).
Por ejemplo, aquí están todas las posibles alineaciones de "CAT" y "CT":

```
list(all_alignments("CAT", "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)]]
```

Aquí hay una forma más legible de la salida: (ïndica un hueco)

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

```
CAT
CT-

CAT
C-T

CAT-
C--T

CA-T
C-T-

C-AT
CT--

CAT
-CT

CAT-
-C-T

CA-T
-CT-

CAT-
--CT

CAT--
---CT

CA-T
--CT
```

```
CA-T-
--C-T

CA--T
--CT-

C-AT
-CT-

C-AT
-C-T

C-AT-
-C--T

C-A-T
-C-T-

C--AT
-CT--

-CAT
CT--

-CAT
C-T-

-CAT
C--T

-CAT-
C---T

-CA-T
C--T-

-C-AT
C-T--

--CAT
CT---
```

A continuación, creo una función que toma dos secuencias y un alineamiento para producir una puntuación:

```
def alignment_score(x, y, alignment):
    """Score an alignment.
```

```
    x, y -- sequences.
    alignment -- an alignment of x and 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
```

Considere esta alineación:

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

He aquí un ejemplo de cálculo de la puntuación:

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

```
1
```

Con estas dos funciones y la solución de fuerza bruta buscará todos los alineamientos para encontrar uno con la máxima puntuación:

$$"all_alignments alignment_score"$$

```
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),
    )
```

(Este 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 temporal de esta solución? Para dos secuencias de m y n elementos.

El número de alineaciones posibles viene dado por los números de Delannoy. El número de alineaciones viene dado por la relación de recurrencia.

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

Para que te hagas una idea de lo rápido que 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)
```

Hay 3 posibles alineaciones de dos secuencias de 1 elemento:

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

```
3
```

Hay 8.097.453 alineaciones posibles de dos secuencias de 10 caracteres:

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

```
8097453
```

Hay 2,05e+75 alineaciones posibles de dos secuencias de 100 caracteres:

```
D(100, 100)
```

```
205371683087241577022877800627197112033484312834955058714104727584027414304 1
```

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

(Ver OEIS A001850 para más información sobre los números de Delannoy de la forma (,).)Dnn

–El cálculo de la puntuación de la alineación lleva un tiempo lineal en los tamaños de ambas secuencias: O(+).nm

Como resultado, la complejidad temporal global de la solución de fuerza bruta es

–O(D(n, m) × (n + m))

5. Creación de una solución más eficiente

La solución de fuerza bruta es sencilla, pero no se adapta bien. En la práctica, la alineación de secuencias se utiliza para analizar secuencias de datos biológicos (por ejemplo, secuencias de ácidos nucleicos). Dado que el tamaño de estas secuencias puede ser de cientos o miles de elementos, es imposible que la solución de fuerza bruta funcione para datos de ese tamaño.

En 1970, Saul B. Needleman y Christian D. Wunsch crearon un algoritmo más rápido para resolver este problema: el algoritmo Needleman-Wunsch. (Véase .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 utiliza la programación dinámica para resolver el problema de alineación de secuencias en tiempo O().mn.time

Aquí hay una implementación en Python del algoritmo Needleman-Wunsch, basada en la sección 3 de .ªlgoritmo Needleman-Wunsch paralelo para Grid":

```
from itertools import product
from collections import deque


def needleman_wunsch(x, y):
    """Run the Needleman-Wunsch algorithm on two sequences.

    x, y -- sequences.

    Code based on pseudocode in Section 3 of:

    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)
```

(Este código utiliza: *collections.deque*, *itertools.product*, *zip*, *list*.)

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

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

Y así, el algoritmo más rápido simplemente llamará :

$$"needleman_wunsch"$$

```
def align_fast(x, y):
    """Align two sequences, maximizing the
    alignment score, using the Needleman-Wunsch
    algorithm.

    x, y -- sequences.
    """
    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
```

¿Cuál es la complejidad temporal de esta solución? Para dos secuencias de y elementos:nm
–Crear las tablas y toma $"O()tiempo.FPtrmn"$
–Crear la alineación navegando desde hasta toma $"O(+)time.Ptr[N-1,M-1]Ptr[0,0]nm"$
Como resultado, la complejidad temporal global del algoritmo es

$$O(mn)$$

6. En conclusión...

En el post de esta semana, has aprendido a resolver el problema de .ªlineación de Secuencias". Aprendiste a crear una solución de fuerza bruta que genera todos los alineamientos posibles. Luego aprendió que la fuerza bruta es inviable para secuencias más grandes: ¡dos secuencias de 10 elementos tienen más de 8.000.000 de alineaciones diferentes! Por último, has aprendido a reimplementar el algoritmo de Needleman-Wunsch en Python.

Mi desafío para usted:

-Modificar para tomar parámetros para la puntuación: $"needleman_wunsch"$

– $"score_gap"$ - cómo puntuar un gap. (Por defecto: -1) – $"score_same"$ - cómo puntuar elementos iguales. (Por defecto: +1) – $"score_different"$ - cómo puntuar elementos no iguales. (Por defecto: -1)

Por ejemplo, puede llamar a la nueva función así

```
needleman_wunsch(
    "CAT",
    "CT",
    score_gap=-10,
```

```
        score_same=+3,
        score_different=-1,
    )
```

Si te ha gustado el post de esta semana, compártelo con tus amigos y estate atento al post de la semana que viene. ¡Nos vemos entonces!

7. Bibliografía

Referencias

[1] Solving the Sequence Alignment problem in Python, John Lekberg, Recuperado el 30 de junio de 2022: https://johnlekberg.com/blog/2020-10-25-seq-align.html

[2] Solving the Sequence Alignment problem in Python, Jose Francisco Ruiz Muñoz Recuperado el 30 de junio de 2022: https://colab.research.google.com/drive/1DVvEZFcNkdCq-ArLHJ7EPSiJnlhLSP7iscrollTo=lcM3qVd6k-k