Recuperación de la secuencia de edición

En esta función se pide obtener, además del coste, la secuencia de operaciones de edicón. Esto equivale a una representación gráfica de qué cambios se realizaron sobre una palabra para obtener otra.

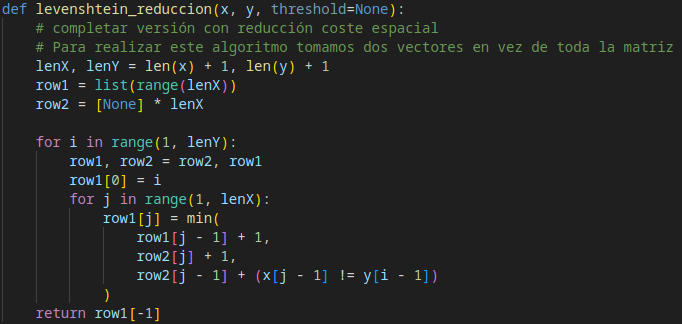
Este resultado lo podemos conseguir recorriendo la matriz de estados en desde el último elemento hasta llegar al primero. La posición a la que debemos de saltar en cada paso ha de ser la que tenga el valor mínimo de entre las posición justo encima, justo detrás o la posición detrás y encima. Saltar una columna hacia arriba supone añadir una tupla (xi-1, “”) a la secuencia, saltar hacia atrás supone (“”, yj-1) y el salto en diagonal supone (xi,yj). En el código esto se implementa con tres bucles. Cabe destacar que en el primer bucle el orden de las condiciones if es relevante para obtener el resultado adecuado: si el valor en las posiciones comprobadas es el mismo nos debemos guiar por la posición que nos acerca más a las coordenadas (0,0). Esto nos lleva a tener que elejir dar un salto en diagonal siempre que se pueda.

A medida que recorremos la matriz debemos de añadir las tuplas conseguidas a una lista. Esta es el objeto que devolveremos que contendrá la secuencia de operaciones de ecición, pero, para que esté ordenada, deberemos darle la vuelta.

Ampliaciones Damerou Restricted

En este caso tendremos que resolver el problema de la distancia Damerou-Levenshtein pero sin poder reutilizar aquellas letras en las que ya hemos hecho un intercambio del tipo (“ab”,“ba”).

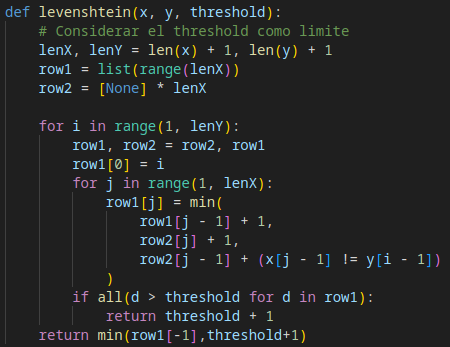
Esto lo implementamos con una matriz de booleanos y, en el caso de reducción de coste espacial, con dos listas de booleanos. Cada posición en estas estructuras de datos representa, para un estado dado, si una letra se puede volver a reutilizar para una trasposición se aplique una trasposición



El objetivo de esta función es aplicar el algoritmo de levenshtein con una reducción del coste espacial, dado que solo es necesario tener en memoria las dos últimas filas para poder realizar las comparaciones necesarias para la realización de este algoritmo.

Primero se inicializan las longitudes de la misma manera que en el Levenshtein normal, y inicializamos las dos primeras filas del algoritmo, **row1** como el rango de **lenX** y **row2** como una lista vacía del la misma longitud.

Después comienza el bucle donde compararemos estas 2 filas. Primero las cambiamos para que **row2** sea la base, hacemos que el primer elemento de la fila donde vamos a hacer las comparaciones sea igual a **i**, y iteramos a lo largo del resto de **row1** haciendo las evaluaciones correspondientes a las operaciones de inserción, borrado y sustitución. Guardamos su mínimo y seguimos hasta acabar el bucle, que es cuando devolveremos el último valor obtenido.



Esta función es igual que la de **levenshtein\_reduccion**, excepto que hacemos un uso de **threshold** para dejar de hacer comparaciones en cuanto la distancia de levenshtein se pasa de este **threshold**.

Respecto al código, esto significa hacer una comprobación al final del cálculo de cada fila, de manera que si todos los elementos de la **row1** son estrictamente mayores que **threshold**, entonces dejamos de iterar y devolvemos el **threshold**.