

Prácticas de Algorítmica

Búsqueda aproximada de cadenas
(2ª parte proyecto coordinado SAR-ALT)

Curso 2022-2023

Objetivos

El objetivo de esta segunda parte del proyecto conjunto SAR-ALT es:

1. Estudiar e implementar la búsqueda aproximada de una cadena respecto de todas las cadenas de un diccionario.
2. Utilizar esa búsqueda aproximada para ampliar el motor de recuperación de información desarrollado en SAR de forma que acepte consultas con búsqueda aproximada.

Búsqueda aproximada de cadenas

Búsqueda aproximada de cadenas

- La búsqueda puede ser **online** o bien **offline**: se llama **online** cuando el diccionario no puede ser preprocesado para generar algún tipo de índice que permita mejorar la eficiencia. Cuando sí podemos realizar algún tipo de preproceso se llama **offline** o **indexing**. Vamos a suponer que sí se puede realizar un preproceso, estamos en el caso **offline**.
- Es posible orientar la búsqueda a **nivel de secuencia o de palabras**:
 - A nivel **de secuencia** alguien podría escribir “wiki pedia” en lugar de “wikipedia” y recuperarse del error (se consideran errores llamados *split* y *merge*).
 - A nivel **de palabras** se separa el texto por sus separadores y luego se limita a buscar las palabras más cercanas. En el ejemplo anterior buscaría las palabras cercanas a “wiki” y a “pedia”. En este caso se trabaja con un diccionario de palabras.

Nosotros vamos a limitarnos a trabajar a nivel de palabra.

Búsqueda aproximada de cadenas

- Hay que considerar si el diccionario se actualiza de manera frecuente. Si casi nunca se actualiza, la eficiencia del preproceso no será crítica. Vamos a suponer que el diccionario **no se actualiza** (para actualizar regeneramos de cero)
- También podemos distinguir entre búsquedas que tienen en cuenta el contexto (otras palabras que forman parte de dicha búsqueda) o no. Consideramos el caso más simple: **no se considera el contexto**.
- También se puede contemplar la **búsqueda predictiva**: el sistema sugiere palabras mientras escribimos término a buscar. El sistema debería completar una palabra a partir de su prefijo *incluso si el usuario se equivoca al escribir ese prefijo*. Únicamente buscaremos sugerencias de manera **no predictiva**.

Búsqueda aproximada de cadenas

Resumiendo: Buscaremos palabras ya escritas. Las candidatas están en un diccionario que podremos haber preprocesado y que normalmente no va a ser modificado. Esta suele ser la tarea de un corrector ortográfico:

https://en.wikipedia.org/wiki/Spelling_suggestion

Existen bibliotecas Python con esta funcionalidad, por ejemplo:

<http://pyenchant.github.io/pyenchant/api/enchant.html>

```
>>> import enchant
>>> d = enchant.Dict("en_US")    # create dictionary for US English
>>> d.suggest("enchnt")
['enchant', 'enchants', 'enchanter', 'penchant', 'incant',
 'enchain', 'enchanted']
```

Búsqueda aproximada de cadenas

Completaremos una clase Python llamada `SpellSuggester` que recibe en el constructor un diccionario de términos/palabras y lo preprocesa. El objeto creado dispondrá del método `suggest` que recibe:

- La palabra a buscar.
- Un umbral o nivel de tolerancia (para limitar la búsqueda).

Este método devuelve lista que contiene las palabras del diccionario que estén próximas a la buscada (dentro del umbral indicado). Esto se verá con más detalle posteriormente.

Para ello necesitamos evaluar la distancia entre dos palabras. Vamos a limitarnos a estudiar **distancias de edición**:

La distancia de edición entre dos cadenas $x, y \in \Sigma^$ es el número mínimo de operaciones de edición para convertir la primera en la segunda (o viceversa, la distancia es simétrica).*

Las distancias de edición más utilizadas son:

- Distancia de Levenshtein.
- Distancia de Damerau-Levenstein.

Distancias de edición

Distancias de edición

Las operaciones de edición consideradas para calcular la distancia de Levenshtein entre x e y son:

- Insertar un carácter (denotado $\lambda \rightarrow y_j$)
- Borrar un carácter (denotado $x_i \rightarrow \lambda$)
- Sustituir un carácter por otro (denotado $x_i \rightarrow y_j$)

Aunque hemos definido la distancia de edición como el número de operaciones de edición, es posible considerar el caso **ponderado** donde las operaciones de edición pueden tener costes diferentes. Por ejemplo, tiene sentido que sustituir una consonante por otra diferente tenga un coste mayor a sustituir una vocal por la misma vocal acentuada. En el caso ponderado el coste de cada operación vendría dado por:

- $\gamma(\lambda \rightarrow y_j)$
- $\gamma(x_i \rightarrow \lambda)$
- $\gamma(x_i \rightarrow y_j)$

Distancias de edición

En este proyecto vamos a limitarnos al caso **no ponderado**. Es decir, aunque se pueda contemplar que algunas operaciones de distancia de edición tienen un coste mayor que otras, el coste de todas las inserciones o borrados es el mismo sin importar el símbolo del alfabeto. Por otra parte, todas las operaciones de sustitución dependen únicamente de si el símbolo que se sustituye es el mismo (un acierto, de coste cero) o es diferente (coste mayor que cero).

$$\blacksquare \gamma(\lambda \rightarrow y_j) = 1$$

$$\blacksquare \gamma(x_i \rightarrow \lambda) = 1$$

$$\blacksquare \gamma(x_i \rightarrow y_j) = \begin{cases} 0 & \text{si } x_i = y_j \\ 1 & \text{si } x_i \neq y_j \end{cases}$$

Distancia de Levenshtein

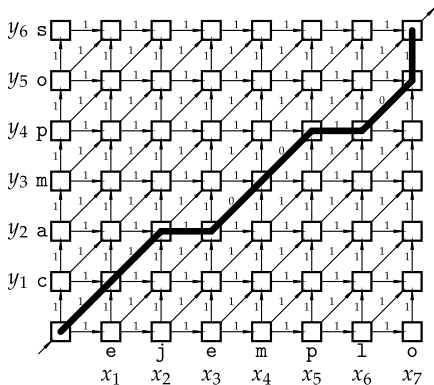
En la siguiente ecuación recursiva $D(i, j)$ denota el coste de convertir el prefijo de longitud i de x en el prefijo de longitud j de y :

$$D(i, j) = \min \begin{cases} 0 & \text{si } i=0 \text{ y } j=0 \\ D(i-1, j) + 1 & \text{si } i>0 \\ D(i, j-1) + 1 & \text{si } j>0 \\ D(i-1, j-1) & \text{si } i > 0, j > 0, x_i = y_j \\ D(i-1, j-1) + 1 & \text{si } i > 0, j > 0, x_i \neq y_j \end{cases}$$

De manera que $d(x, y) = D(|x|, |y|)$.

Distancia de Levenshtein

El siguiente gráfico ilustra la matriz de estados cuando se calcula la distancia de Levenshtein entre las cadenas *campos* y *ejemplo*:



Observa que todas las transiciones tienen coste 1 exceptuando las diagonales donde $x_i = y_j$.

Distancia de Damerau-Levenshtein

Se trata de una extensión de la distancia de Levenshtein en la que se añade un nuevo tipo de operación de edición:

- Trasponer o intercambiar dos símbolos adyacentes $ab \leftrightarrow ba$. Con la notación de editar x para obtener y sería $x_{i-1} = y_j$ y $x_i = y_{j-1}$ para $i > 0, j > 0$.

Una observación **obvia** es que, como incluye las operaciones de edición de la distancia de Levenshtein, se cumplirá siempre que:

$$\text{Damerau-Levenshtein}(x, y) \leq \text{Levenshtein}(x, y) \quad \forall x, y \in \Sigma^*$$

Por ejemplo, la distancia de Levenshtein entre “algoritmo” y “algoritmo” es 2 (dos sustituciones “i” \rightarrow “t”, “t” \rightarrow “i”) mientras que la distancia de Damerau-Levenshtein es 1 (intercambio “it” \leftrightarrow “ti”).

Distancia de Damerau-Levenshtein

Existen 2 variantes de esta distancia:

- La versión **no restringida** es la distancia como se entiende de la definición: buscar la forma de aplicar el menor número de operaciones para pasar de una cadena a la otra.
- En la versión **restringida** vamos a suponer que una vez intercambiados dos símbolos, éstos **no** se pueden utilizar en otras operaciones de edición.

Resulta que la versión restringida:

1. A veces no da la distancia mínima. Por ejemplo, $d(\text{"ba"}, \text{"acb"})$ es 3 en la versión restringida pero sólo 2 en la no restringida.
2. No cumple la desigualdad triangular, por lo que no es una métrica:
Ejemplo: $d(\text{"ca"}, \text{"ac"}) + d(\text{"ac"}, \text{"abc"}) < d(\text{"ca"}, \text{"abc"})$
3. Es más sencilla y rápida de calcular que la no restringida.

Distancia de Damerau-Levenstein restringida

El cálculo de la versión restringida de Damerau-Levenstein mediante programación dinámica es relativamente sencillo, basta con utilizar la siguiente ecuación recursiva:

$$D(i, j) = \min \begin{cases} 0 & \text{si } i=0 \text{ y } j=0 \\ D(i-1, j) + 1 & \text{si } i>0 \\ D(i, j-1) + 1 & \text{si } j>0 \\ D(i-1, j-1) & \text{si } i > 0, j > 0, x_i = y_j \\ D(i-1, j-1) + 1 & \text{si } i > 0, j > 0, x_i \neq y_j \\ D(i-2, j-2) + 1 & \text{si } i > 1, j > 1, x_{i-1} = y_j, x_i = y_{j-1} \end{cases}$$

Observa que simplemente añade un caso más (última línea) a la ecuación recursiva de la distancia de Levenshtein.

Distancia de Damerau-Levenstein no restringida

En la versión restringida se supone que $ab \rightarrow ba$ tiene coste 1. En la versión no restringida suponemos un conjunto de operaciones:

$aub \rightarrow bva$ tiene coste $1 + |u| + |v|$ para cualquier $u, v \in \Sigma^*$.

Explicación: para obtener bva a partir de aub **condicionado a que exista una transposición** $ab \rightarrow ba$ deberíamos:

1. Borrar u en aub , lo cual tiene coste $|u|$ borrados,
2. La transposición $ab \rightarrow ba$ con coste 1 y, finalmente,
3. Insertar v en ba , con coste $|v|$ inserciones.

Observa que una alternativa a pasar de aub a bva sin $ab \rightarrow ba$ sería:

1. Sustituir $a \rightarrow b$ (i.e. $aub \rightarrow bub$),
2. Obtener v a partir de u (i.e. $bub \rightarrow bvb$),
3. Sustituir $b \rightarrow a$ (i.e. $bvb \rightarrow bva$).

Esto tiene coste $2 + d(u, v)$, por lo que sólo conviene la transposición si $1 + |u| + |v| < 2 + d(u, v)$ (i.e. si $d(u, v) > |u| + |v| - 1$). A medida que crecen $|u|$ y $|v|$ será más improbable que valga la pena considerar las transposiciones.

Distancia de Damerau-Levenstein *intermedia*

Es posible implementar la distancia de Damerau-Levenstein no restringida con contadores de talla $|\Sigma|$, pero resulta complejo. A medida que $|u|$ y $|v|$ crecen es más y más improbable que tenga sentido aplicar una transposición $a \leftrightarrow b$ en $aub \rightarrow bva$.

Por eso proponemos una versión llamada *intermedia* que consiste en considerar únicamente los casos donde $|u| + |v| \leq \text{cte}$ para una constante cte prefijada. En esta práctica consideraremos $\text{cte} = 1$ de modo que únicamente consideraremos las siguientes operaciones de edición donde $a, b, c, d \in \Sigma$:

- $ab \leftrightarrow ba$ coste 1
- $acb \leftrightarrow ba$ coste 2
- $ab \leftrightarrow bca$ coste 2

No vale la pena contemplar $acb \leftrightarrow bda$ con coste 3 puesto que ese mismo coste se consigue sin trasposición.

Tareas a realizar en el proyecto

Tareas obligatorias

1. Recuperar la secuencia de operaciones de edición en Levenshtein.
2. Implementar Levenshtein con reducción de coste espacial y con un parámetro umbral o `threshold` de modo que se pueda dejar de calcular cualquier distancia mayor a dicho umbral.
3. Completar la clase `SpellSuggester`.
4. Modificar el Indexador y el Recuperador de SAR para permitir la búsqueda aproximada de cadenas utilizando la clase `SpellSuggester`.

Nota

Se parte de (o de dispone de) una implementación básica del cálculo de la distancia de Levenshtein.

Ver apartado “Material proporcionado”.

Posibles ampliaciones

1. Utilizar una cota optimista para ahorrar evaluaciones de la distancia de Levenshtein e integrarlo en `SpellSuggester` para utilizarlo en el recuperador de SAR.
2. Implementar la versión **restringida** de Damerau-Levenstein (también con un parámetro umbral o `threshold` de modo que se pueda dejar de calcular cualquier distancia mayor a dicho umbral). Es automático que quede integrado en el recuperador.
3. Obtener la secuencia de operaciones de edición para Damerau-Levenshtein restringido.
4. Implementar la versión **intermedio** de Damerau-Levenstein (también con un parámetro umbral o `threshold` de modo que se pueda dejar de calcular cualquier distancia mayor a dicho umbral). Es automático que quede integrado en el recuperador.
5. Obtener la secuencia de operaciones de edición para Damerau-Levenshtein intermedio.
6. Estudio del coste temporal de las distintas funciones de distancia.

Material proporcionado

Material proporcionado

Para la realización de este proyecto se proporciona el siguiente material:

- Versión básica de Levenshtein (fichero `distancias.py`).
- Comprobar los 3 tipos de distancia (fichero `test_distancias.py`).
- Comprobar la recuperación del camino (fichero `test_edicion.py`).
- Plantilla de la clase `SpellSuggester` (fichero `spellsuggester.py`).
- Fichero `test_spellsuggester.py` para probar distancias y la clase `Spellsuggester`.
- Referencia con la salida de `test_spellsuggester.py`.
- Versiones actualizadas `SAR_Indexer.py` y `sar_searcher.py`.
- Referencia para comprobar que la integración con el proyecto de SAR es correcta.

Atención

Si alguien no cursó SAR en los 2 últimos cursos debe ponerse en contacto con su profesor de prácticas.

Material proporcionado

En el fichero `distancias.py` se proporciona la siguiente función que utiliza una matriz `numpy`:

```
def levenshtein_matriz(x, y, threshold=None):
    # no utiliza threshold pero debe permitir recibirlo
    lenX, lenY = len(x), len(y)
    D = np.zeros((lenX+1, lenY+1), dtype=np.int)
    for i in range(1, lenX+1):
        D[i][0] = D[i-1][0] + 1
    for j in range(1, lenY+1):
        D[0][j] = D[0][j-1] + 1
        for i in range(1, lenX+1):
            D[i][j] = min(D[i-1][j] + 1,
                          D[i][j-1] + 1,
                          D[i-1][j-1] + (x[i-1] != y[j-1]))
    return D[lenX, lenY]
```

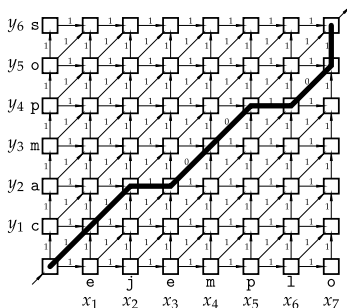
Nota:

Utilizar una matriz `numpy` en este caso puede ser algo más lento que utilizar un diccionario Python.

Tareas obligatorias

1. Recuperar secuencia de operaciones de edición

Es lo que en programación dinámica se denomina “recuperar el camino”:



- **Importante:** tras finalizar el cálculo de la distancia.
- Se parte de la posición $D[\text{len}X, \text{len}Y]$ y se debe ir retrocediendo.
- Ten en cuenta que al retroceder puedes llegar a “una pared” de la matriz y, en ese caso, hay predecesores que no existen.
- Se obtiene la secuencia de operaciones de edición en sentido inverso. Es más eficiente guardarlos con `append` y hacer un solo `reverse` al finalizar (insertarlos al inicio es más ineficiente).

1. Recuperar secuencia de operaciones de edición

La secuencia devuelta debe estar en el formato utilizado por la siguiente función que convierte una cadena en otra utilizando esa secuencia de operaciones. Aquí tienes un ejemplo de cómo convertir la cadena 'ejemplo' en la cadena 'campos' (*es el mismo ejemplo del libro de los apuntes de Andrés Marzal, María José Castro y Pablo Aibar*):

```
aplicar_edicion("ejemplo",  
                [('e', 'c'), ('j', 'a'), ('e', ''), ('m', 'm'),  
                 ('p', 'p'), ('l', ''), ('o', 'o'), ('', 's')])
```

La edición para Levenshtein es una lista de tuplas, cada tupla son 2 letras donde una de ellas puede ser la cadena vacía si se trata de una inserción o de un borrado. Los aciertos también aparecen explícitamente en la secuencia.

1. Recuperar secuencia de operaciones de edición

El fichero `test_edicion.py` contiene código para lanzar las funciones que calculan operaciones de edición sobre una batería de tests. El resultado tiene este aspecto:

```
Comprobando si levenshtein(ejemplo,campos) == 5
operaciones: [('e', ''), ('j', 'c'), ('e', 'a'), ('m', 'm'),
              ('p', 'p'), ('l', 'o'), ('o', 's')]
- e      se aplica en |ejemplo para dar |jemplo coste 1
- j  c   se aplica en |jemplo para dar c|emplo coste 1
- e  a   se aplica en c|emplo para dar ca|mplo coste 1
- m  m   se aplica en ca|mplo para dar cam|plo coste 0
- p  p   se aplica en cam|plo para dar camp|lo coste 0
- l  o   se aplica en camp|lo para dar campo|o coste 1
- o  s   se aplica en campo|o para dar campos| coste 1
CORRECTO!
```

2. Reducción del coste espacial

Para reducir el coste espacial debes reemplazar la matriz por dos vectores. Puedes utilizar vectores numpy o listas Python. A partir del código de `levenshtein_matriz` reemplaza la matriz:

```
D = np.zeros((lenX+1,lenY+1), dtype=np.int)
```

por 2 vectores (numpy o listas Python).

- La función `np.zeros` también puede crear un vector si solamente le pasas un tamaño:

```
>>> np.zeros(10,dtype=np.int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

- Conviene que sean vectores de tamaño `lenX` porque corresponde al bucle más interno.
- La forma *eficiente* de intercambiar los 2 vectores es intercambiar las *referencias* a los mismo:

```
vprev,vcurrent = vcurrent,vprev
```

2. Utilización de un parámetro umbral o `threshold`

- Esta ampliación se debe realizar a partir de la versión con reducción del coste espacial.
- Se trata de añadir un tercer argumento `threshold` de modo que se pueda dejar de calcular una distancia cuando sepamos seguro que el valor final superará dicho umbral.
- **Importante:** Si la distancia calculada es mayor a `threshold` el método deberá devolver `threshold+1` (incluso al final).
- La forma más sencilla consiste en **detener el algoritmo** (vía `return`) si, tras calcular una etapa (columna) se puede asegurar que el coste superará el umbral.

3. Completar la clase `spellSuggester`

- El constructor recibe:
 - Un diccionario que asocia a una cadena una función Python:
 - La cadena sirve para seleccionar qué distancia utilizar.
 - La función (dadas 2 cadenas y un `threshold` devuelve una distancia).
 - Un vocabulario (lista de cadenas) o bien una cadena que se interpreta como filename con el nombre de un fichero (lo lee y tokeniza para construir el vocabulario).
 - El nombre de la función por defecto (normalmente 'Levenshtein').
 - El valor `threshold` a utilizar por defecto (normalmente 3).
- Método `suggest` recibe una cadena en su primer argumento obligatorio y, opcionalmente, el tipo de distancia, un valor `threshold` (ambos por defecto a `None` para indicar que se usa el valor por defecto de la clase) y un argumento `flatten` por defecto a `True`. Devuelve una lista si el argumento `flatten` vale `True` pero devuelve una **lista de listas** de palabras en otro caso: La lista i -ésima contiene las palabras a distancia i , (para i hasta `threshold` inclusive).

3. Completar la clase `SpellSuggester`

Se dispone del fichero `test_spellsuggester.py` para probar las distancias que hayáis implementado. El resultado de la ejecución, con el fichero de prueba `miniquijote.txt` debería de dar:

```
levenshtein_m
- casa [1, 17, 81, 344, 847] 1290
- ancho [0, 2, 13, 75, 511] 601
- ecrvantse [0, 0, 0, 0, 7] 7
- uqijoext [0, 0, 0, 0, 0] 0
...
```

La lista que acompaña a cada palabra de prueba es el nº resultados a distancia 1,2,3,4 y luego el total.

Atención

También se crea un fichero por distancia con las listas de palabras que ha encontrado para cada distancia. Tenéis dichos ficheros de referencia para comprobar, ya que podría estar mal implementado y que por casualidad coincidieran los conteos.

4. Integración del corrector en el proyecto SAR

En el proyecto de SAR se proporcionaban 3 ficheros (`SAR_Indexer.py`, `SAR_Searcher.py` y `SAR_lib.py`) y únicamente se debía modificar `SAR_lib.py`.

- Debéis bajaros un nuevo `SAR_Indexer.py` donde hemos añadido opciones de la línea de comandos:

```
-d --distance nombre_funcion_distancia  
-t --threshold umbral_distancia
```

para poder especificar (opcionalmente) los valores por defecto del tipo de distancia a utilizar y el threshold empleado en el `SpellSuggester` (pasándole estos valores en el constructor).

- En `SAR_Searcher` hemos añadido esas 2 opciones y una más para activar la búsqueda con tolerancia:

```
-s --spell
```

Los valores de `-d` y `-t` se usan para llamar al método `suggest`.

La búsqueda con tolerancia se activa si se usa alguno de los 3 parámetros, en otro caso se busca **sin** tolerancia (sin usar `suggest`).

4. Integración del corrector en el proyecto SAR

- En `SAR_lib` debes añadir el siguiente método a la clase `SAR_Project`:

```
def set_spelling(self, use_spelling, distance, threshold):  
    """  
    self.use_spelling a True se activa la corrección ortográfica  
    EN LAS PALABRAS NO ENCONTRADAS, en caso contrario NO utilizará  
    corrección ortográfica  
    """  
    self.use_spelling = use_spelling  
    self.distance = distance  
    self.threshold = threshold
```

- Y dar valores por defecto (`False`, `None` y `None`) a esos atributos en el constructor.
- En el constructor `SAR_Project` también hay que crear un `SpellSuggester` en el objeto `self.speller`.
- El vocabulario del `SpellSuggester` no se conoce todavía en el constructor, hay que esperar a indexar los ficheros de noticias y utilizar posteriormente el método `set_vocabulary`.

4. Integración del corrector en el proyecto SAR

Es posible que sea casi suficiente modificar el método `get_posting` para que realice la búsqueda con tolerancia.

Para ello, **únicamente si el término no está en `self.index`** hay que usar `self.speller` y utilizar `suggest` con dicho término. Si devuelve una lista no vacía de palabras, hay que buscarlas todas (lo que haría el `get_posting` original) y juntar todos los términos invertidos en una sola lista (como se hacía ya con la conectiva lógica **or**). Una vez obtenido el *posting list*, se usa como antes y da igual si proviene de un término que estaba en el diccionario o del resultado de usar el corrector ortográfico (es transparente al resto).

Entre los recursos disponibles hemos dejado:

- Las noticias del año 2015 que se utilizaron en la asignatura SAR.
- Ficheros referencia: `queries.txt` con consultas, `genera_resultados_SAR.sh` para lanzar batería de pruebas, `resultadosSAR.txt` con la referencia (recopila los diversos ficheros de resultados también disponibles).

Ampliaciones

1. Ampliación cota optimista

Se trata de utilizar una cota optimista para ahorrar evaluaciones de la distancia de Levenshtein (**no sirve** para Damerau-Levenshtein).

La cota optimista propuesta supone olvidar el orden en el que aparecen las letras (como ocurre también con los modelos *bag of words* que puede que vieras en SAR). Veamos unos ejemplos:

- Si se compara 'casa' con 'saca', el nº veces que aparece cada letra es el mismo, así que una cota basada en el conteo devolvería 0.
- Si se compara 'casa' con 'saco', vemos que hay una 'a' de más en la primera y una 'o' de más en la segunda. Una cota optimista basándonos en el conteo (ignorando el orden) sólo podría concluir que la distancia es ≥ 1 por una posible sustitución de la 'a' por la 'o'.
- Si se compara 'casa' con 'abad', vemos que hay una 'c' y una 's' de más en la primera y una 'b' y una 'd' de más en la segunda. Se puede deducir que la distancia es ≥ 2 para dar cuenta con sustituciones o borrados las letras de más de uno de los lado (interesa el máximo entre ambos, aquí había empate).

1. Ampliación cota optimista

■ En general, hay que:

- contar n° veces **de más** que aparece cada letra de una cadena en la otra.
- sumar todas esas cuentas
- hacer lo mismo con la otra cadena.
- devolver el máximo de ambas sumas.

Por ejemplo, si nos pasan las cadenas 'casa' y 'abad' y recorremos la primera sumando 1 obtenemos el diccionario:

{ 'c': 1, 'a': 2, 's': 1 }

Si ahora recorremos la segunda cadena restando 1 nos queda:

{ 'c': 1, 'a': 0, 's': 1, 'b': -1, 'd': -1 }

La suma de valores positivos da 2, la de los negativos da -2. El máximo de ambos en valor absoluto es 2 que sería la estimación optimista de $\text{Levenshtein}('casa', 'abad')$. Donde, por *optimista* se entiende que es menor o igual que la distancia real.

1. Ampliación cota optimista

- El objetivo de esta ampliación es crear una función `levenshtein_cota_optimista` que internamente realice este cálculo y si el resultado es mayor o igual al `threshold` devuelva `threshold+1`. En otro caso ya calcula la distancia de Levenshtein de la forma normal (la versión que puede parar por el `threshold` tras calcular cada columna).
- La forma de integrar esta ampliación en el código de SAR es muy sencilla: al construir el objeto `SpellSuggester` le pasamos en el diccionario de funciones de distancia una entrada con el nombre (por ejemplo) `'levenshtein_o'` asociada a la función que pruebe la cota optimista y, si supera el `threshold`, delegue en el Levenshtein con `threshold` anterior. Así es posible usar también la versión de Levenshtein previa cuando queramos.

2. Ampliación Damerau-Levenshtein *restringido*

Esta ampliación consiste en copiar el código de Levenshtein y modificarlo para dar cuenta de la nueva regla que aparece en la ecuación recursiva (la última línea):

$$D(i, j) = \min \begin{cases} 0 & \text{si } i=0 \text{ y } j=0 \\ D(i-1, j) + 1 & \text{si } i>0 \\ D(i, j-1) + 1 & \text{si } j>0 \\ D(i-1, j-1) & \text{si } i > 0, j > 0, x_i = y_j \\ D(i-1, j-1) + 1 & \text{si } i > 0, j > 0, x_i \neq y_j \\ D(i-2, j-2) + 1 & \text{si } i > 1, j > 1, x_{i-1} = y_j, x_i = y_{j-1} \end{cases}$$

se sugiere tomar como punto de partida la versión de Levenshtein con parada por `threshold` y reducción de coste espacial que utiliza 2 vectores columna. Ahora tendrás que utilizar TRES vectores porque aparece una dependencia $j-2$.

2. Ampliación Damerau-Levenshtein *restringido*

- Para comprobar que funciona se proporciona el programa `test_distancias.py` con un pequeño conjunto de test con ejemplos donde Damerau-Levenshtein restringido da un valor diferente (obviamente inferior) a Levenshtein.
- Para integrarlo en el proyecto de SAR basta con incluir esta función en el diccionario que se le pasa al constructor de `SpellSuggester`.

3. Ampliación Damerau-Levenshtein *restringido*

Si queréis incluir la ampliación de recuperar el camino, hay que:

- Ver cómo especificar las operaciones de edición de tipo transposición $ab \rightarrow ba$ en la secuencia devuelta. Se trata de generar tuplas ('ab', 'ba') donde las letras obviamente serán las adecuadas según el caso.
- Probar que funciona con el fichero test_edicion.py.

```
Comprobando si damerau_r(algortimac,algoritmica) == 3
operaciones: [('a', 'a'), ('l', 'l'), ('g', 'g'), ('o', 'o'),
              ('r', 'r'), ('', 'i'), ('t', 't'), ('im', 'mi'), ('ac', 'ca')]
- a  a  se aplica en |algortimac para dar a|lgortimac coste 0
- l  l  se aplica en a|lgortimac para dar al|gortimac coste 0
- g  g  se aplica en al|gortimac para dar alg|ortimac coste 0
- o  o  se aplica en algo|rtimac para dar algo|rtimac coste 0
- r  r  se aplica en algo|rtimac para dar al|gortimac coste 0
-   i  se aplica en al|gortimac para dar al|gortimac coste 1
- t  t  se aplica en al|gortimac para dar al|gortimac coste 0
- im mi se aplica en al|gortimac para dar al|gortimac coste 1
- ac ca se aplica en al|gortimac para dar al|gortimac coste 1
CORRECTO!
```

4. Ampliación Damerau-Levenshtein *intermedio*

En este caso se tienen en cuenta una serie de operaciones de transposición descritas anteriormente:

- $ab \leftrightarrow ba$ coste 1
- $acb \leftrightarrow ba$ coste 2
- $ab \leftrightarrow bca$ coste 2

donde $a, b, c, d \in \Sigma$. Para ello:

- Inspirándote en la ecuación de recurrencia de la versión restringida y mirando las nuevas operaciones de edición, escribid la ecuación de recurrencia para esta versión.
- Puedes tomar como punto de partida para la implementación la versión Damerau-Levenshtein restringida (seguramente nadie implementará intermedio sin haber hecho restringida) y añadir los nuevos casos que han aparecido en la ecuación de recurrencia.
- Seguramente ahora has de utilizar 4 en lugar de 3 vectores columna.

5. Ampliación Damerau-Levenshtein *intermedio*

- La ampliación de recuperar la secuencia de operaciones de edición requiere considerar tuplas de tipo ('acb', 'ba') y ('ab', 'bca').

Comprobando si `damerau_i(algoritmo, algortximo) == 2`

operaciones: [('a', 'a'), ('l', 'l'), ('g', 'g'), ('o', 'o'),
('r', 'r'), ('it', 'txi'), ('m', 'm'), ('o', 'o')]

- a a se aplica en |algoritmo para dar a|lgoritmo coste 0
- l l se aplica en a|lgoritmo para dar al|goritmo coste 0
- g g se aplica en al|goritmo para dar alg|oritmo coste 0
- o o se aplica en algo|ritmo para dar algo|ritmo coste 0
- r r se aplica en algo|ritmo para dar algor|itmo coste 0
- it txi se aplica en algor|itmo para dar algortxi|mo coste 2
- m m se aplica en algortxi|mo para dar algortxim|o coste 0
- o o se aplica en algortxim|o para dar algortximo| coste 0

CORRECTO!

6. Estudio del coste temporal de las distintas funciones de distancia

- Se trata de medir el tiempo de utilizar el método suggest.
- No tiene mucho sentido comparar distintos tipos de distancia porque calculan cosas distintas, sería centrarse por tanto en las variantes de Levenshtein.
- Ver la influencia del tamaño del vocabulario.
- No se trata de limitarse medir y plotear, también hay que interpretar y sacar alguna conclusión.

Presentación, defensa y evaluación

Presentación y defensa del proyecto

Cada grupo/equipo debe preparar:

- Una pequeña **memoria** o informe con una descripción de las tareas desarrolladas, el reparto de trabajo y decisiones adoptadas. Debe ser en formato txt o pdf (o ir acompañado de una versión pdf si se utilizan otros formatos tipo docx u odt).
- El **código** realizado. Se valorará la organización (incluyendo la utilización de módulos/bibliotecas), estilo, eficiencia, documentación/comentarios, etc.

Pueden entregarse estos ficheros (puede ser en un zip, evitad rar):

- La memoria.
- Fichero SAR_lib.py.
- Fichero distancias.py.
- Según ampliaciones, scripts para medir distancias, resultados, etc.
- NO subáis los ficheros que no se han modificado (especialmente NO subáis miniquijote.txt ni carpeta 2015).

Evaluación

- Hay un día dedicado a la evaluación: 18 de noviembre de 2022.
- La memoria y el código se habrán entregado previamente (os avisaremos de la tarea para la entrega) para que nos lo hayamos podido mirar.
- El día de la evaluación os haremos preguntas y probaremos el código.
- El proyecto vale un 10% de la nota de la asignatura:
 - La implementación+memoria perfecta de la parte obligatoria vale un 60% de la nota.
 - Las ampliaciones sirven para compensar deficiencias y el 40% restante, cada una vale 10%.
 - *Con la parte obligatoria perfecta*, bastarían 4 de las 6 ampliaciones para llegar al 100%, pero puede convenir hacer más para compensar cualquier fallo en la memoria, etc.