Approximate string matching
(Búsqueda Aproximada de Cadenas)

Algorítmica (2ª parte de proyecto coordinado SAR - ALT)

versión 1.0

El objetivo de esta segunda parte del proyecto conjunto SAR-ALT es añadir al motor de recuperación de información que se desarrolló en la primera parte el proyecto (prácticas de SAR) la capacidad de hacer búsquedas aproximadas en nuestro índice. Para lograr este objetivo antes deberemos elegir un algoritmo eficiente para hacer la búsqueda aproximada de una cadena respecto de todas las cadenas del diccionario de términos.

1. Búsqueda aproximada de cadenas

En el contexto de este proyecto definiremos la búsqueda aproximada de cadenas como la búsqueda en un diccionario de una o varias cadenas que sean *similares* a una cadena dada. El objetivo de la búsqueda aproximada no es tanto encontrar una coincidencia exacta (exact match) sino un conjunto de cadenas *cercanas* a la cadena de referencia.

1.1. Medidas de distancia

Para poder decidir si dos cadenas son *cercanas* es necesario definir una medida de distancia entre cadenas y un umbral de tolerancia (la distancia máxima que se considera *cercana*).

1.1.1. Distancia de Hamming

La distancia de Hamming entre dos cadenas α y β de igual longitud, $|\alpha| = |\beta|$, viene definida por la Ecuación 1.

$$hamming(\alpha, \beta) = \sum_{i=1}^{|\alpha|} sust(\alpha_i, \beta_i) \tag{1}$$

donde $sust(\alpha_i, \beta_i)$ es el coste de substituir el carácter α_i por β_i. Típicamente $sust$ es 0 cuando $\alpha_i = \beta_i$ y 1 en caso contrario:

$$sust(\alpha_i, \beta_i) = \begin{cases} 0: & \alpha_i = \beta_i \\ 1: & \alpha_i \neq \beta_i \end{cases}$$

La distancia de Hamming se pude definir por tanto como el número de caracteres distintos, posición por posición, entre dos cadenas de igual longitud. La necesidad de que las cadenas que se comparan deban ser de la misma longitud hace inservible esta distancia para el propósito de este proyecto.

1.1.2. Distancia de Levenshtein

También conocida como distancia de edición, la distancia de Levenshtein es la medida de distancia entre cadenas más utilizada. La distancia de Levenshtein entre dos cadenas α y β, no necesariamente de la misma longitud, se define como el número mínimo de operaciones básicas que son necesaria para transformar la cadena α en la cadena β. Las operaciones permitidas son:

- el borrado de un carácter de la cadena α,
- la inserción de un carácter de la cadena β, y
- la substitución de un carácter de la cadena α por otro de la cadena β.

Asumiendo que todas las operaciones de edición tienen la misma penalización, 1, la distancia de Leveshtein se puede calcular por programación dinámica utilizando la ecuación recursiva de la Ecuación 2, donde, $d_{\alpha,\beta}(i,j)$ es la distancia parcial entre las cadenas α y β cuando se han analizados los i primeros caracteres de la cadena α y los j primeros caracteres de la cadena β. La distancia de Levenshtein entre las dos cadenas será por tanto, $d_{\alpha,\beta}(|\alpha|, |\beta|)$.

$$d_{\alpha,\beta}(i,j) = \begin{cases} 0 & : \quad i = j = 0 \\ i & : \quad i > 0 \wedge j = 0 \\ j & : \quad i = 0 \wedge j > 0 \\ min \begin{pmatrix} d_{\alpha,\beta}(i-1, j-1) + 1_{(\alpha_i \neq \beta_j)}, \\ d_{\alpha,\beta}(i, j-1) + 1, \\ d_{\alpha,\beta}(i-1, j) + 1 \end{pmatrix} & : \quad i > 0 \wedge j > 0 \end{cases} \tag{2}$$

El cálculo de la distancia de Levenshtein entre dos cadenas infiere un alineamiento entre esas dos cadenas. Debe tenerse en cuenta que para dos cadenas puede haber múltiples alineamientos de mínima distancia. La Figura 1.1.2 muestra un posible alineamiento entre las cadenas $AACBDC$ y $ACBBBC$ con distancia mínima 3.

El concepto de distancia de Levenshtein se puede ampliar considerando que no necesariamente todas las operaciones tienen el mismo coste (por ejemplo, se puede penalizar más la substitución de una vocal por otra que la substitución de una vocal por la misma vocal acentuada). La Ecuación 3 muestra la función recursiva para el cálculo de la versión *ponderada*

A	A	C	B	D	-	C
=	↓	=	=	↕	↑	=
A	-	C	B	B	B	C

Figura 1: Alineamiento de 2 cadenas.

de la distancia de Levenshtein. En esta ecuación $borr(\alpha_i)$ es el coste de borrar el carácter α_i, $ins(\beta_j)$ es el coste de insertar el carácter β_j y $sust(\alpha_i, \beta_j)$ es el coste de substituir el carácter α_i por el β_j carácter.

$$d_{\alpha,\beta}(i,j) = \begin{cases} 0 & : \quad i = j = 0 \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i) & : \quad i > 0 \wedge j = 0 \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j) & : \quad i = 0 \wedge j > 0 \\ min \begin{pmatrix} d_{\alpha,\beta}(i-1,j-1) + sust(\alpha_i,\beta_j), \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j), \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i) \end{pmatrix} & : \quad i > 0 \wedge j > 0 \end{cases} \tag{3}$$

1.1.3. Distancia de Damerau-Levenshtein

La distancia de Damerau-Levenshtein es una extensión de la distancia de Levenshtein en la cual además de la substitución, borrado e inserción también se permite la transposición de dos caracteres consecutivos.

La Ecuación 4 y la Ecuación 5 muestran respectivamente la función recursiva de la distancia de Damerau-Levenshtein y la de su versión *ponderada*. En el segundo caso, $trans(\alpha_{i-1:i})$ es el coste de transponer la subcadena $\alpha_{i-1:i}$.

$$d_{\alpha,\beta}(i,j) = \begin{cases} 0 & : \quad i = j = 0 \\ i & : \quad i > 0 \wedge j = 0 \\ j & : \quad i = 0 \wedge j > 0 \\ min \begin{pmatrix} d_{\alpha,\beta}(i-1,j-1) + 1_{(\alpha_i \neq \beta_j)}, \\ d_{\alpha,\beta}(i,j-1) + 1, \\ d_{\alpha,\beta}(i-1,j) + 1 \end{pmatrix} & : \quad i = 1 \vee j = 1 \\ min \begin{pmatrix} d_{\alpha,\beta}(i-1,j-1) + 1_{(\alpha_i \neq \beta_j)}, \\ d_{\alpha,\beta}(i,j-1) + 1, \\ d_{\alpha,\beta}(i-1,j) + 1, \\ d_{\alpha,\beta}(i-2,j-2) + 1 : \alpha_i = \beta_{j-1} \wedge \alpha_{i-1} = \beta_j \end{pmatrix} & : \quad i > 1 \wedge j > 1 \end{cases} \tag{4}$$

$$d_{\alpha,\beta}(i,j) = \begin{cases} 0 & : \quad i = j = 0 \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i) & : \quad i > 0 \wedge j = 0 \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j) & : \quad i = 0 \wedge j > 0 \\ min \begin{pmatrix} d_{\alpha,\beta}(i-1,j-1) + sust(\alpha_i,\beta_j), \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j), \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i) \end{pmatrix} & : \quad i = 1 \vee j = 1 \\ min \begin{pmatrix} d_{\alpha,\beta}(i-1,j-1) + sust(\alpha_i,\beta_j), \\ d_{\alpha,\beta}(i,j-1) + ins(\beta_j), \\ d_{\alpha,\beta}(i-1,j) + borr(\alpha_i), \\ d_{\alpha,\beta}(i-2,j-2) + trans(\alpha_{i-1:i}) : \alpha_i = \beta_{j-1} \wedge \alpha_{i-1} = \beta_j \end{pmatrix} & : \quad i > 1 \wedge j > 1 \end{cases} \tag{5}$$

2. Trie

Un trie[1] es una estructura de tipo árbol utilizada para almacenar el diccionario de términos en sistemas de recuperación de información. La Figura 2 muestra un trie construido con los términos *caro*, *cara*, *codo* y *caros*.

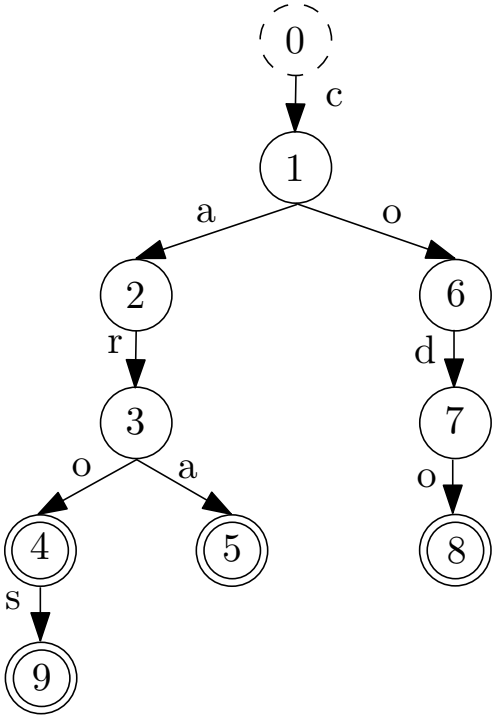

Figura 2: Ejemplo de trie.

[1]El termino trie deriva de la palabra inglesa para recuperación, reTRIEval.

Podemos realizar algunas consideraciones respecto a la estructura del trie:

- La raíz del árbol representa la cadena vacía.
- Las hojas representan los términos del diccionario. Por ejemplo, el nodo 8 representa el término *codo*.
- Cada nodo interno representa una subcadena con tantos caracteres como la profundidad de ese nodo. Esa subcadena será un prefijo de uno o más términos. Por ejemplo, el nodo 3 representa el prefijo *car* que es común a 3 términos
- Cada nodo hijo representa una alternativa diferente para ampliar la longitud del prefijo. Por ejemplo, los dos hijos del nodo 3 representan los dos carácter valido para ampliar el prefijo *car*: *caro* y *cara*.

El trie está íntimamente relacionado con el autómata aceptor de prefijos.

3. Extensión de la distancia de Levenshtein

Es posible realizar una extensión de la distancia de Levenshtein para medir la distancia entre una cadena y todas las cadenas de un diccionario representadas mediante un trie.

La Ecuación 6 muestra la ecuación recursiva de la distancia de Levenshtein entre una cadena α y un trie τ.

$$d_{\alpha,\tau}(i,\eta) = \begin{cases} 0 & : i = 0 \wedge \eta = \eta_0 \\ i & : i > 0 \wedge \eta = \eta_0 \\ depth(\eta) & : i = 0 \wedge \eta \neq \eta_0 \\ min \begin{pmatrix} d_{\alpha,\tau}(i-1,\eta) + 1, \\ d_{\alpha,\tau}(i, parent(\eta)) + 1, \\ d_{\alpha,\tau}(i-1, parent(\eta)) + 1_{(\alpha_i \neq char(\eta))} \end{pmatrix} & : i > 0 \wedge \eta \neq \eta_0 \end{cases} \tag{6}$$

donde, α_i es el carácter de la posición i de la cadena α, η es un nodo del trie τ, η_0 es la raíz del trie y las funciones $char(\cdot)$, $parent(\cdot)$ y $depth(\cdot)$ representan el carácter, el padre y la profundidad de un nodo respectivamente. La distancia $d_{\alpha,\tau}(i,\eta)$ es la mínima distancia de Levenshtein entre la cadena α y el trie τ cuando se ha analizado hasta el carácter i de la cadena y el camino desde la raíz hasta el nodo η en el trie.

La versión *ponderada* de esta ecuación se muestra en la Ecuación 7.

$$d_{\alpha,\tau}(i,\eta) = \begin{cases} 0 & : i = 0 \wedge \eta = \eta_0 \\ d_{\alpha,\tau}(i-1,\eta) + borr(\alpha_i) & : i > 0 \wedge \eta = \eta_0 \\ d_{\alpha,\tau}(i,parent(\eta)) + ins(char(\eta)) & : i = 0 \wedge \eta \neq \eta_0 \\ min \begin{pmatrix} d_{\alpha,\tau}(i-1,\eta) + borr(\alpha_i), \\ d_{\alpha,\tau}(i,parent(\eta)) + ins(char(\eta)), \\ d_{\alpha,\tau}(i-1,parent(\eta)) + sust(\alpha_i, char(\eta)) \end{pmatrix} & : i > 0 \wedge \eta \neq \eta_0 \end{cases} \quad (7)$$

La distancia de Levenshtein entre la cadena α y una cadena β del vocabulario representado por el trie τ se puede calcular utilizando las Ecuaciones 6 y 7 como $d_{\alpha,\tau}(|\alpha|, \eta_\beta)$ donde η_β es la hoja del trie que representa la cadena β.

La Figura 3 puede darte pistas de como abordar la implementación. Esta figura muestra algunos momentos del cálculo de esta distancia para la cadena *casos* y el trie de la Figura 2.

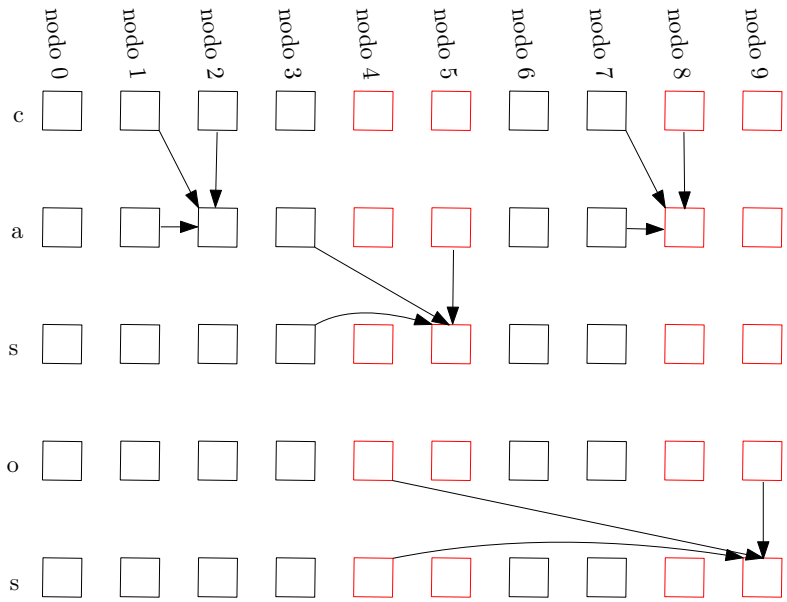

Figura 3: Distancia de Levenshtein extendida.

4. Distancia de cadena vs Trie mediente ramificación de estados

Una alternativa a la programación dinámica para el cálculo de la distancia de Levenshtein entre una cadena y un trie es mediante la ramificación de estados. Para el problema que nos ocupa, un estado vendrá representado por la terna (i, η, d), donde i, $1 \leq i \leq |\alpha|$ es la longitud de una subcadena de α, η es un nodo del trie τ y d es la mejor distancia hasta el momento para esa subcadena y nodo.

En nuestro caso la ramificación se realizará mediante la aplicación *hacia adelante* de las tres operaciones básicas (borrado sobre la cadena, inserción de cualquier hijo del nodo y substitución de una carácter de la cadena por un hijo del nodo) al estado a ramificar.

La poda estará basada en la memorización de resultados intermedios como ocurre en la mayoría de problemas de ramificación y poda abordables también mediante programación dinámica.

¿Cuál será el estado inicial?, ¿Se te ocurre alguna cota optimista?

5. Trabajo a realizar en el proyecto

5.1. Estudio de algoritmos para la búsqueda aproximada de cadenas

En primer lugar realizaremos un estudio empírico para comparar el coste temporal de algunos algoritmos para la búsqueda aproximada de cadenas. Para ello utilizaremos el fichero *"quijote.txt"*.

La tarea en este apartado consistirá en:

- Generar un diccionario en trie con el vocabulario del fichero *"quijote.txt"*. Cargar el fichero, *limpiar* el texto, extraer el vocabulario y generar un trie. Se deberá **guardar en memoria secundaria** para facilitar su uso posterior tanto **el trie** como **una lista con todo el vocabulario**.
- Obligatoriamente implementar y testear:
  - Los algoritmos para el cálculo de la **distancia de Levenshtein** y la **distancia de Damerau-Levenshtein** entre dos cadenas. Se deberá hacer un bucle sobre todo el vocabulario para encontrar los términos cuya distancia respecto de la cadena dada sea menor que un umbral determinado.
  - El algoritmo para el cálculo de la distancia de **Levenshtein entre una cadena y** un conjunto de cadenas representado como **un trie** mediante programación dinámica.
  - El algoritmo para el cálculo de la distancia de Levenshtein entre una cadena y un conjunto de cadenas representado como un trie mediante **la ramificación y poda de estados**.
- Opcionalmente implementar y testear:
  - El algoritmo para el cálculo de la distancia de Levenshtein entre una cadena y un conjunto de cadenas representado como un trie mediante programación dinámica **hacia adelante**.

  - El algoritmo para el cálculo de la distancia de Levenshtein entre una cadena y un conjunto de cadenas representado como un trie mediante programación dinámica **hacia adelante con lista de nodos activos**.
  - El algoritmo para el cálculo de la distancia de Levenshtein entre una cadena y un conjunto de cadenas representado como un trie mediante la ramificación y poda de estados **con cota**.
  - Cualquier otra alternativa para la búsqueda aproximada: **Levenshtein automaton, Burkhard Keller tree, ..**

- Hacer un estudio del coste temporal de todas las soluciones desarrolladas y elegir la mas eficiente.

5.2. Modificar el Indexador y el Recuperador de SAR para permitir la búsqueda aproximada de cadenas

Se deberá:

- Modificar el indexador de noticias de SAR para generar un trie con todo el vocabulario del diccionario de términos que se almacenará en disco con el resto de la información ya contemplada.

  - Opcionalmente se puede substituir la hash por la estructura de trie o utilizar las dos.

- Modificar el recuperador de noticias de SAR para que acepte consultas con búsqueda aproximada. Las peticiones de búsquedas aproximadas tendrán el formato *termino***%***d_max* para búsquedas aproximadas utilizando la distancia de Levenshtein y *termino***@***d_max* para la distancia de Damerau-Levenshtein.

  - Por ejemplo, la consulta **casas %2** debería recuperar todas las noticias donde se encuentre alguna palabra que tenga una distancia de Levenshtein como mucho igual a 2 respecto de la cadena 'casas'. La consulta **casas@2** haría lo mismo pero considerando la distancia Damerau-Levenshtein.
  - Los algoritmos utilizados serán los que hayan demostrado mejor rendimiento en la evaluación previa. Si hay dos algoritmos distintos que se comportan cada uno mejor en un contexto distinto, se podrá utilizar cada uno de ellos en sus contexto más favorable.
  - Los algoritmos de creación del trie y de búsqueda aproximada deberán incluirse en una librería aparte que será importada tanto por el indexador como por el recuperador.

5.3. Presentación y defensa del proyecto

Cada grupo deberá hacer un informe mostrando los resultados obtenidos y justificando las decisiones que ha adoptado. También se podrá realizar una evaluación de eficacia y eficiencia de la solución propuesta.

5.4. Distribución de las sesiones

A continuación se presenta una programación del proyecto para 8 sesiones:

- Sesión 1: Presentación + Levenshtein de cadenas
- Sesión 2: Levenshtein de cadenas
- Sesión 3: Trie + Levenshtein de cadena y trie con PD
- Sesión 4: Levenshtein de cadena y trie con PD
- Sesión 5: Levenshtein de cadena y trie con PD y ramificación
- Sesión 6: Levenshtein de cadena y trie con ramificación
- Sesión 7: Levenshtein de cadena y trie con ramificación + Adaptación código SAR
- Sesión 8: Adaptación código de SAR

Esta programación no contempla el trabajo del grupo fuera del laboratorio ni la evaluación del proyecto.

Appendices

A. Añadir la ruta de anaconda3 al PATH

En **cada sesión** en el laboratorio se debe añadir al PATH la ruta de los binarios de anaconda3. Esto se puede hacer añadiendo la siguiente linea al fichero *'.bashrc'* de vuestro HOME.

```
export PATH=/opt/anaconda3/bin:$PATH
```

B. Librería *NumPy*

La librería Numpy es una librería que permite, entre otras cosa, disponer de matrices n-dimensionales en Python. Recordemos que no hay un tipo de datos matriz en Python de forma nativa.

Importación de la librería y creación de una matriz 2x4 de enteros:

```
import numpy as np

M = np.empty(dtype=np.int8, shape=(2, 4))
print(M)
```

```
[[   0    0 -128   63]
 [   0    0 -128   63]]
```

El método *empty* crea una matriz pero no la inicializa. Podemos consultar información de la matriz **M** utilizando algunos de sus métodos.

```
print("n. dimensions: ", M.ndim)
print("shape:", M.shape)
print("size: ", M.size)
print("type:", M.dtype)
print("size of each cell:", M.itemsize, "bytes")
print("total size:", M.nbytes, "bytes")
```

```
n. dimensions:  2
shape: (2, 4)
size:  8
type: int8
size of each cell: 1 bytes
total size: 8 bytes
```

Podemos crear una matriz e inicializarla a cero:

```
import numpy as np
M = np.zeros(dtype=np.int8, shape=(2, 4))
print(M)
```

```
[[0 0 0 0]
 [0 0 0 0]]
```

O, podemos inicializar todas las celdas a cualquier otro valor de dos formas distintas:

```
import numpy as np

M1 = np.zeros(dtype=np.int8, shape=(2, 4)) + 5  # sumamos 5 a todas las celdas
print(M1)

M2 = np.empty(dtype=np.int8, shape=(3, 7))
M2.fill(7)
print(M2)
```

```
[[5 5 5 5]
 [5 5 5 5]]

[[7 7 7 7 7 7 7]
 [7 7 7 7 7 7 7]
 [7 7 7 7 7 7 7]]
```

Algunas otras operaciones divertidas sobre matrices utilizando NumPy:

```
import numpy as np

M = np.empty(dtype=int, shape=(3,5))
# asignamos el mismo valor a toda la matriz
M[:] = 5

# assignamos el mismo valor a una fila
M[1,:] = 3

# o a una columna
M[:,3] = 4

# o a una unica celda
M[0,4] = -4

print(M)

[[ 5  5  5  4 -4]
 [ 3  3  3  4  3]
 [ 5  5  5  4  5]]

# calculamos la suma de la matriz
print(M.sum())

55

# calculamos la suma colapsando el eje 0
# sumamos por columnas
print(M.sum(axis=0))

[13 13 13 12  4]

# calculamos la suma colapsando el eje 1
# sumamos por filas
print(M.sum(axis=1))
```

```
[15 16 24]
```

```
# que celdas de la matriz son < 5?
print(M < 5)
```

```
[[False False False  True  True]
 [ True  True  True  True  True]
 [False False False  True False]]
```

```
# que indices tienen las celdas < 5?
print(np.argwhere(M < 5))
```

```
[[0 3]
 [0 4]
 [1 0]
 [1 1]
 [1 2]
 [1 3]
 [1 4]
 [2 3]]
```

```
# y en la ultima columna?
print(np.argwhere(M[:, -1] < 5))
```

```
[[0]
 [1]]
```

C. FAQs