Prácticas de Algorítmica

Segunda parte del proyecto
coordinado SAR-ALT:
Búsqueda aproximada de cadenas

Curso 2020-2021

Objetivos

Objetivos

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

1. Crear un sistema eficiente para realizar la búsqueda aproximada de una cadena respecto de todas las cadenas del diccionario.
   Como subtareas:
   - A) Aprender la técnica de programación dinámica hacia delante.
   - B) Extender la distancia entre cadenas a distancia cadena-trie.
   - C) Medir experimentalmente qué algoritmo es mejor para unos datos concretos.
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

Una de las mejores referencias recomendadas para estudiar con detalle la búsqueda aproximada de cadenas es el artículo:

> *"Indexing Methods for Approximate Dictionary Searching: Comparative Analysis" (por Leonid Boytsov)*
> *Disponible en* `http://boytsov.info/pubs/jea2011.pdf`

Otra referencia, mucho más rudimentaria, es la página de wikipedia:

> `https://en.wikipedia.org/wiki/Approximate_string_matching`

Vamos a introducir rápidamente el problema situando en contexto cuál es el objetivo concreto en este proyecto:

- 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**.

Búsqueda aproximada de cadenas

- 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.
- Otro aspecto a tener en cuenta cuando utilizamos un diccionario es saber si éste va a ser actualizado de manera frecuente o no. Si casi nunca se actualiza no será importante la eficiencia o el coste de la parte de preprocesado. Vamos a suponer que el diccionario **no se actualiza** (en caso de querer actualizarlo, podemos regenerar las estructuras de datos desde cero).
- También podemos distinguir entre búsquedas que tienen en cuenta el contexto (otras palabras que forman parte de dicha búsqueda) o no. Vamos a tratar con el caso más simple donde **no se tiene en cuenta el contexto**.

Búsqueda aproximada de cadenas

- Otro punto a tener en cuenta es si queremos que el sistema nos sugiera palabras mientras estamos escribiendo el término a buscar (búsqueda **predictiva**). 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**.

Resumiendo: vamos a limitarnos a palabras ya escritas, que pueden contener errores ortográficos. Buscaremos las palabras candidatas 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

El objetivo es diseñar una clase Python que recibe un diccionario de términos/palabras, lo preprocesa y después ofrece un método `suggest` que recibe:

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

Y devuelve:

- Una lista de las palabras del diccionario que estén próximas a la buscada (dentro del umbral indicado).

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

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

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 y 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).

- $\gamma(\lambda \to y_j) = 1$
- $\gamma(x_i \to \lambda) = 1$
- $\gamma(x_i \to y_j) = \begin{cases} 0 & \text{si } x_i = y_j \\ 1 & \text{si } x_i \neq y_j \end{cases}$

Distancia de Levenshtein

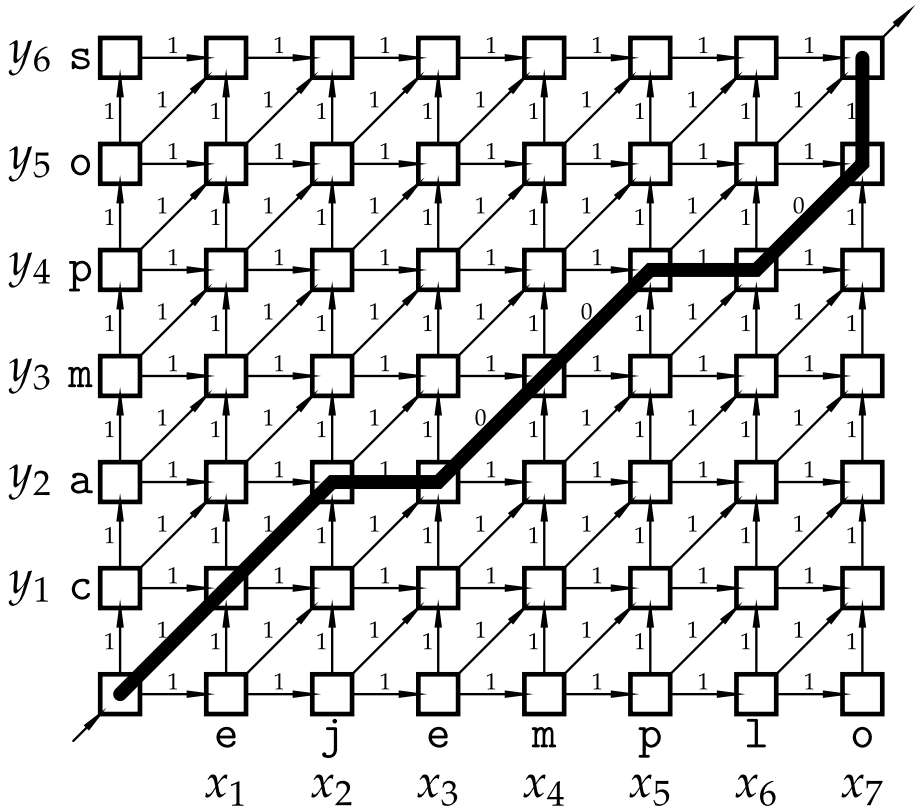
En la siguiente ecuación recursiva $D(i, j)$ denota el coste de convertir el prefijo de longitud i de x en el prefijo de longitud j de y:

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

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 Levenshtein

Es más, con este sencillo ejemplo en Python (utilizando la biblioteca numpy) podemos mostrar los pesos de las diagonales para algunos pares de palabras:

```
def matriz(term, ref):
    vref  = np.array(list(ref))
    return np.vstack([vref != letter for letter in term])+0

for term,ref in [("campos","ejemplo"),
                 ("reconocibles","irreconocible")]:
    print(term,ref,levenshtein_dist(term,ref))
    print(matriz(term,ref))
```

```
campos ejemplo 5.0
[[1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 0 1 1 1]
 [1 1 1 1 0 1 1]
 [1 1 1 1 1 1 0]
 [1 1 1 1 1 1 1]]
```

Distancia de Levenshtein

```
reconocibles irreconocible 3.0
[[1 0 0 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 0 1 1 1 1 1 1 1 1 0]
 [1 1 1 1 0 1 1 1 0 1 1 1 1]
 [1 1 1 1 1 0 1 0 1 1 1 1 1]
 [1 1 1 1 1 1 0 1 1 1 1 1 1]
 [1 1 1 1 1 0 1 0 1 1 1 1 1]
 [1 1 1 1 0 1 1 1 0 1 1 1 1]
 [0 1 1 1 1 1 1 1 1 0 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 0 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 0 1]
 [1 1 1 0 1 1 1 1 1 1 1 1 0]
 [1 1 1 1 1 1 1 1 1 1 1 1 1]]
```

Se puede observar en general que, si no nos interesasen distancias muy altas, bastaría con calcular los valores cercanos a la diagonal principal.
Nota: En la figura, a diferencia de en esta matriz, la cadena vertical está dibujada desde debajo hacia arriba y por eso allí la diagonal no es la principal.

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:

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

Por ejemplo, la distancia de Levenshtein entre "algoritmo" y "algortimo" 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("ba","acb")` es 3 en la versión restringida pero sólo 2 en la versión no restringida.
2. No cumple la desigualdad triangular, por lo que no es una métrica.
   Ejemplo: `d("ca","ac") + d("ac","abc") < d("ca","abc")`
   Esto hace que no pueda utilizarse en algunas circunstancias donde esa propiedad es necesaria.
3. Pero es más sencilla y rápida de calcular.

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:

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

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

Distancia de Damerau-Levenstein no restringida

Si en la versión restringida hemos supuesto que $ab \to ba$ tiene coste 1, para la versión no restringida bastaría con suponer un conjunto de operaciones:

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

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

1. Borrar u en aub, lo cual tiene coste $|u|$ borrados,
2. La transposición $ab \to ba$ 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 $ab \to ba$ sería:

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

Esta alternativa 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 es un poco complejo.
Vamos a basarnos en la *intuición* de que a medida que $|u|$ y $|v|$ crecen es más y más improbable que tenga sentido aplicar una transposición $a \leftrightarrow b$ en $aub \rightarrow bva$ para proponer una solución que llamaremos *intermedia* y que consiste en considerar únicamente los casos donde $|u| + |v| \leq$ cte para una constante cte prefijada. En esta práctica consideraremos 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

Observa...

Observa que no vale la pena contemplar la siguiente operación de edición:

- $acb \leftrightarrow bda$ coste 3

Puesto que ese mismo coste se consigue sin necesidad de una trasposición.

Programación dinámica “hacia delante”

Programación dinámica "hacia delante"

- En clase de teoría se ha estudiado cómo pasar de la ecuación recursiva a una implementación iterativa con memorización de resultados intermedios.
- Normalmente esta versión calcula todos los resultados intermedios que suelen disponerse en un vector o en una matriz en función del número de parámetros con los que indexar esos resultados intermedios.
- En la práctica, en muchos problemas existen muchos resultados intermedios no van a ser necesarios. Veremos con un ejemplo (la mochila discreta) que es posible (en general) mantener únicamente los resultados intermedios necesarios. Esta técnica se denomina "programación dinámica hacia delante" porque, en lugar de plantearnos qué estados hay que consultar para calcular una determinada posición `D[i,j]` (en el caso de depender de 2 parámetros `i,j`) lo que haremos es ver cómo influye una posición *activa* `D[i,j]` en otras posiciones posteriores.

Programación dinámica "hacia delante"

Recordemos el problema de la mochila discreta. Tenemos una mochila con capacidad para cargar W unidades de peso y tenemos N objetos cada uno con peso w_i y valor v_i para $1 \leq i \leq N$.
El mayor beneficio que podemos obtener considerando los i primeros objetos y una capacidad hasta c es:

$$V(i,c) = \begin{cases} 0 & \text{si } \ i = 0 \\ \max\{V(i-1,c), V(i-1,c-w_i) + v_i\} & \text{si } \ i > 0, w_i \leq c \\ V(i-1,c) & \text{si } \ i > 0, w_i > c \end{cases}$$

De manera que la llamada $V(N,W)$ nos calcula el máximo beneficio buscado.

Programación dinámica "hacia delante"

El siguiente código calcula ese beneficio utilizando programación dinámica:

```
def iterative_knapsack(W, v, w):
    N = len(v)
    V = np.zeros(shape=(N+1,W+1))
    for c in range(W+1):
        V[0,c] = 0
    for i in range(1, N+1):
        V[i,0] = 0
        for c in range(1, min(W+1,w[i-1])):
            V[i,c] = V[i-1,c]
        for c in range(w[i-1], W+1):
            V[i,c] = max(V[i-1,c],
                         V[i-1,c-w[i-1]] + v[i-1])
    return V[N, W]
```

Programación dinámica "hacia delante"

En el caso de no querer recuperar la secuencia de objetos podríamos considerar la siguiente versión con **reducción del coste espacial**:

```
def iterative_knapsack_2vec(W, v, w):
    Vcurr = np.zeros(W+1)
    Vnext = np.zeros(W+1)
    for i in range(1, len(v)+1):
        Vcurr[0] = 0
        for c in range(1, w[i-1]):
            Vcurr[c] = Vprev[c]
        for c in range(w[i-1], W+1):
            Vcurr[c] = max(Vprev[c], Vprev[c-w[i-1]] + v[i-1])
        Vcurr, Vprev = Vprev, Vcurr
    return Vprev[W]
```

Programación dinámica "hacia delante"

Para implementar la versión hacia delante vamos a mantener una lista explícita de estados activos. Cada estado vendrá vendrá identificado:

- De manera **explícita** por la capacidad de la mochila y,
- De manera **implícita** por el nº de objetos considerados (todos los estados activos de una etapa comparten ese valor).

En cada etapa de programación dinámica recorremos la lista de estados activos para actualizar los estados que corresponda en la etapa siguiente.
Los estados activos se guardan en un diccionario donde sólo almacenamos aquellos que son alcanzados desde la etapa anterior.
Para recorrer los estados activos de la etapa actual (guardados en un diccionario `current`) basta con:

```
for ... # recorremos las etapas de prog. dinámica
    nxt = {}
    for peso,benef in current.items():
        # utilizar peso,benef para actualizar
        # nxt donde corresponda...
    current = nxt
```

Programación dinámica "hacia delante"

- En esta versión de la mochila hacia delante el peso se refiere al peso acumulado por los objetos metidos en la mochila, con lo que inicialmente (para 0 objetos) sólo está activa la hipótesis de peso 0.
- Al llegar al final, a diferencia de la versión anterior, debemos buscar el valor máximo en la última columna y no únicamente el valor $V[N, W]$ (que podría incluso no existir si ninguna combinación de objetos consigue llenar exactamente la mochila).

```
def iterative_knapsack_delante(W, v, w):
    Vcurr = {0:0}
    for vi,wi in zip(v,w):
        Vnext = {}
        for peso,benef in Vcurr.items():
            Vnext[peso] = max(benef, Vnext.get(peso,0))
            if peso+wi <= W:
                Vnext[peso+wi] = max(benef+vi, Vnext.get(peso+wi,0))
        Vcurr = Vnext
    return max(Vcurr.values())
```

Programación dinámica "hacia delante"

Esta otra versión de la mochila hacia delante es un poco más complicada y se basa en una propiedad que no siempre tendremos a mano cuando hagamos programación dinámica: en cada etapa i, dado un estado de peso *weight*, generamos (a lo sumo) otros 2 en la etapa siguiente:

- Otro del mismo peso *weight*.
- Otro de peso $weight + w[i]$.

Por tanto, si recorremos los estados activos *weight* **ordenados por peso**:

- Los que son del mismo peso *weight* se generan de manera ordenada.
- Los que son de peso $weight + w[i]$ también se generan de menor a mayor.

Una vez tenemos esas 2 listas ordenadas, bastaría con fusionarlas:

```
def sparse_knapsack(W,v,w):
  col = [(0,0)]
  for vi,wi in zip(v,w):
    col = merge(col,
                [(weight+wi,value+vi) for (weight,value) in col if weight+wi<=W])
  return max(v for (w,v) in col)
```

Programación dinámica "hacia delante"

El código anterior utiliza la siguiente función que recibe dos listas de tuplas `(peso,beneficio)` ordenadas por `peso` y las junta en una sola lista, también ordenada por `peso`, quedándose con el mayor beneficio si, por acaso, hay tuplas con el mismo `peso` (cada uno procedente de una de las listas):

```
def merge(list1,list2):
  resul=[]
  i,j,len1,len2=0,0,len(list1),len(list2)
  while i<len1 and j<len2:
    if list1[i][0]==list2[j][0]:
      resul.append((list1[i][0],max(list1[i][1],list2[j][1])))
      i += 1
      j += 1
    elif list1[i][0]<list2[j][0]:
      resul.append(list1[i])
      i += 1
    else:
      resul.append(list2[j])
      j += 1
  resul += list1[i:]+list2[j:]
  return resul
```

Programación dinámica "hacia delante"

Ejercicio

El código `versiones_mochila.py` contiene un generador de instancias y las distintas versiones de la mochila discreta presentadas en este boletín. Se pide modificarlo para:

- Medir el tiempo de ejecución de cada versión.
- Medir el nº estados activos utilizados en las 2 versiones "hacia delante" y calcular el porcentaje de estados activos utilizados respecto al total de estados que contiene la matriz de programación dinámica.

Programación dinámica "hacia delante"

Cómo medir tiempos en Python

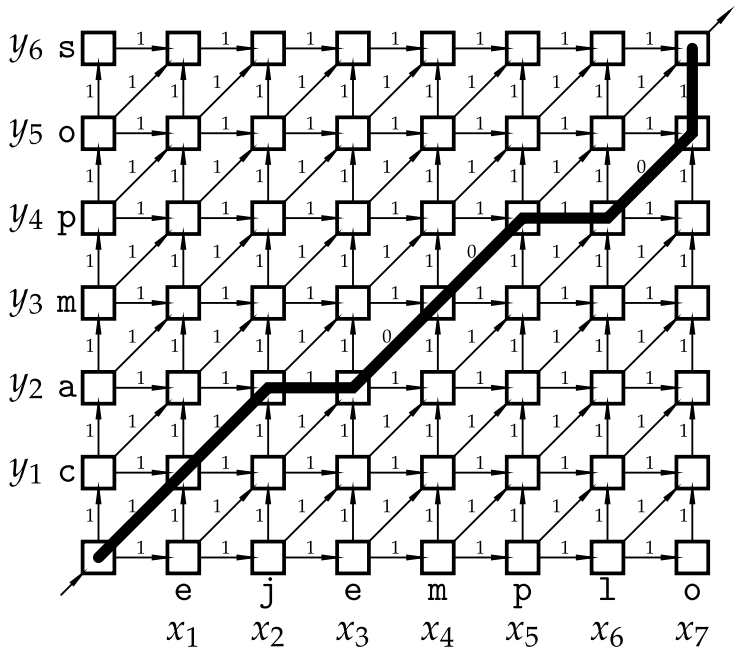
```
def measure_time(function, arguments,
                 prepare=dummy_function, prepare_args=()):
    """ mide el tiempo de ejecutar function(*arguments)

    IMPORTANTE: como se puede ejecutar varias veces puede que sea
    necesario pasarle una función que establezca las condiciones
    necesarias para medir adecuadamente (ej: si mides el tiempo de
    ordenar algo y lo deja ordenado, la próxima vez que ordenes no
    estará desordenado)

    DEVUELVE: tiempo y el valor devuelto por la función"""
    count, accum = 0, 0
    while accum < 0.1:
        prepare(*prepare_args)
        t_ini = time.process_time()
        returned_value = function(*arguments)
        accum += time.process_time()-t_ini
        count += 1
    return accum/count, returned_value
```

Programación dinámica "hacia delante"

El problema de la programación dinámica hacia delante cuando se aplica al problema de la distancia de Levenshtein (y Damerau-Levenstein) es que hay dependencias entre estados de una misma fila y de una misma columna, recordemos el grafo de dependencias:

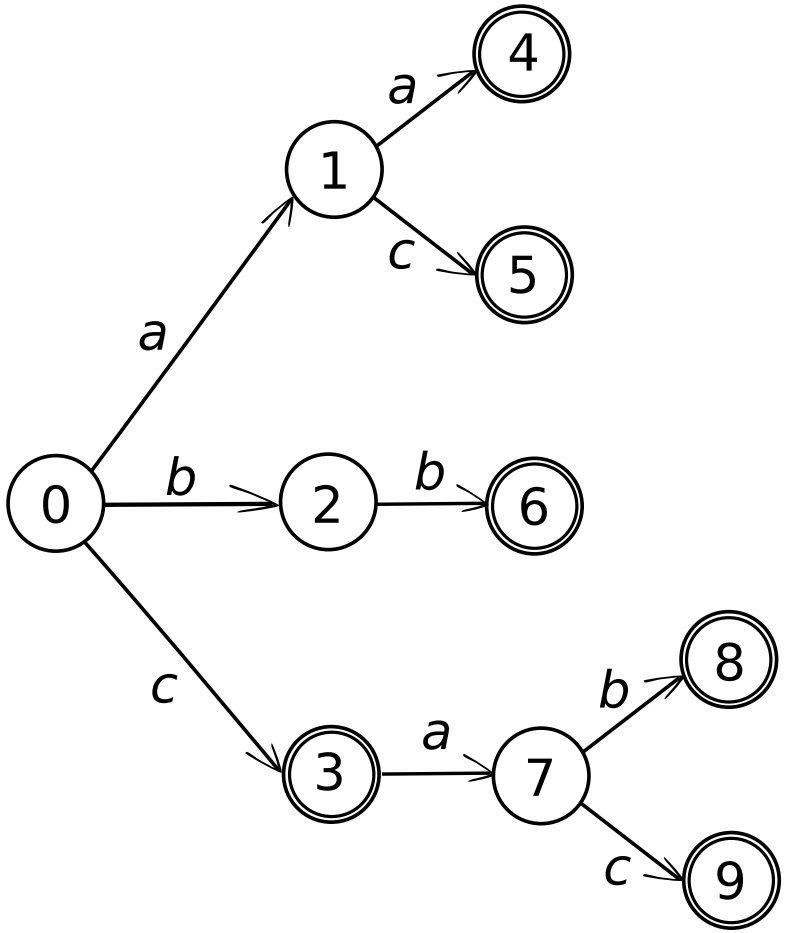


Cosa que no ocurría con el problema de la mochila discreta.
Eso complica la implementación de la versión hacia delante con una lista de estados activos.

La estructura de datos *trie*

La estructura de datos *trie*

Un *trie* es un árbol de prefijos que permite representar un conjunto de cadenas.
El siguiente trie que representa las cadenas `["aa","ac","bb","c","cab","cac"]`:



La estructura de datos *trie*

- Observa que todo prefijo de cualquier cadena aparece como un nodo o estado del *trie* incluyendo el prefijo vacío que es la raíz.
- Salvo el nodo raíz, a todo nodo del trie se llega con una única arista que está etiquetada con un símbolo del alfabeto. De alguna manera podemos asumir que ese símbolo podría ser un atributo de dicho nodo.
- Aquellos estados/prefijos que corresponden a una palabra aparecen marcados como finales. Vemos estados finales internos (no hojas) cuando una palabra puede ser prefijo de otra.
- Un *trie* puede representar un diccionario de una manera posiblemente más compacta que guardando las cadenas por separado.

  ¿Cuánto más compacta? Depende de hasta qué punto esas cadenas comparten prefijos. De hecho, a veces podríamos plantearnos hacer el trie con las cadenas al revés (para que compartan sufijos).

La estructura de datos *trie*

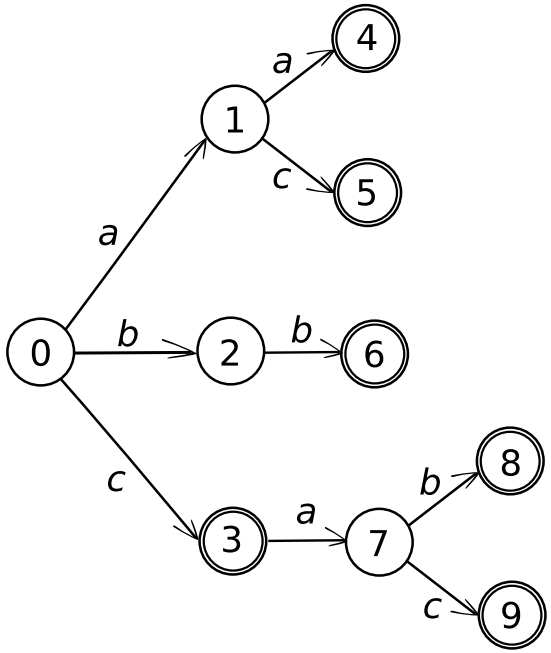
¿Cómo podemos representar una clase `Trie` en Python?
Lo primero que podemos plantearnos sería crear una clase `TrieNode` con campos como:

- Etiqueta o símbolo del alfabeto con el que se llega a dicho nodo.
- Lista de nodos hijos.
- El nodo padre del que proviene.
- Si es un estado terminal o no.

Con esto, la clase `Trie` tendría como principal atributo un `TrieNode` asociado al nodo raíz.
El problema de esta representación es que consumiría una cantidad brutal de espacio en memoria.
Vamos a proponer una representación mucho más eficiente en espacio. Para ello observemos qué sucede si numeramos los nodos del trie siguiendo un **recorrido por niveles**.

La estructura de datos *trie*

El siguiente trie que representa las cadenas `["aa","ac","bb","c","cab","cac"]` y tiene los estados numerados siguiendo un **recorrido por niveles**:



Los hijos de cada nodo ocupan **posiciones contiguas**. Esto nos permite poder recorrer los hijos de cualquier nodo sabiendo únicamente dónde está su primer hijo (observa que hay un estado ficticio nº 10 que sirve de centinela):

`state`	0	1	2	3	4	5	6	7	8	9	10
`first_child`	1	4	6	7	8	8	8	8	10	10	10

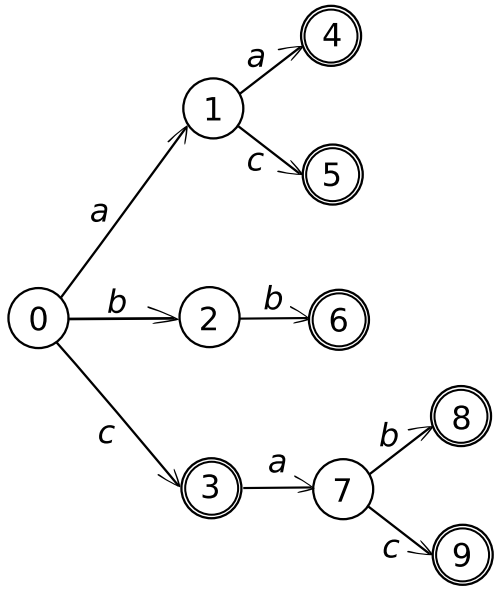
La estructura de datos *trie*

No necesitamos guardar el nº de hijos ni dónde está el último hijo puesto que después de él viene el primer hijo del siguiente nodo.
Para recorrer todos los hijos de un determinado nodo `i` basta con hacer:

```
# recorremos los nodos hijos del nodo i
for child_of_i in range(self.first_child[i],self.first_child[i+1]):
    # process node child_of_i
```

Observa el valor `first_child` cuando un nodo es una hoja:

state	0	1	2	3	4	5	6	7	8	9	10
first_child	1	4	6	7	8	8	8	8	10	10	10



Tareas a realizar en el proyecto

Tareas a realizar en el proyecto

1. Implementar una serie de distancias de edición entre cadenas en su versión básica.
2. Implementar estas distancias de alguna forma "mejorada" que reciba un umbral `threshold` de modo que se pueda dejar de calcular cualquier distancia que supere dicho umbral. Si la distancia calculada es mayor a `threshold` el método podrá simplemente devolver `None`.
3. Evitar la ejecución de la distancia de edición entre aquellas cadenas en las que algún tipo de cota optimista nos permita saber que dicha distancia será mayor al umbral establecido.
4. Extender esta distancia, originalmente definida entre pares de cadenas, para calcular la distancia entre una cadena y un **trie**.
5. Realizar un estudio experimental de medidas de tiempo para determinar qué versiones de las anteriores son las más eficientes para unos datos concretos.
6. Modificar el Indexador y el Recuperador de SAR para permitir la búsqueda aproximada de cadenas utiliznado las versiones más eficientes según el punto anterior.

Tareas a realizar en el proyecto (tarea 1)

1. Implementar una serie de distancias de edición entre cadenas.

Debes implementar, utilizando programación dinámica:

- Distancia de Levenshtein (inserción, borrado y sustitución tienen coste 1).
- Distancia de Damerau-Levenstein restringida.
- Distancia de Damerau-Levenstein "intermedia" (es decir, considerar las operaciones de edición $aub \rightarrow bva$ cuando $|u| + |v| \leq cte$) para $cte = 1$.

Muchos de estos algoritmos ya están disponibles en los apuntes o en wikipedia. El objetivo es tener una referencia para asegurarse de que el resto de las implementaciones dan el mismo resultado y para, posteriormente, comparar la eficiencia con respecto a estas versiones.

Opcional

De manera **opcional** puedes implementar la versión general de Damerau-Levenstein.

Tareas a realizar en el proyecto (tarea 2)

2. Implementar estas distancias de alguna forma "mejorada" que reciba un umbral `threshold` de modo que se pueda dejar de calcular cualquier distancia que supere dicho umbral. Si la distancia calculada es mayor a `threshold` el método podrá simplemente devolver `None`.

Se trata de:

1. Establecer límites en el recorrido para que solamente se calculen aquellas partes del grafo de dependencias que tengan sentido para dicho umbral. Por ejemplo: zonas relativamente cercanas a la diagonal principal de la matriz.
2. Detener el algoritmo si, tras calcular una etapa (fila o columna según sea tu algoritmo) se puede asegurar que el coste superará el umbral.

Debes implementar las mismas variantes que antes:

- Distancia de Levenshtein (inserción, borrado y sustitución tienen coste 1).
- Distancia de Damerau-Levenstein restringida.
- Distancia de Damerau-Levenstein "intermedia" (es decir, considerar las operaciones de edición $aub \rightarrow bva$ cuando $|u| + |v| \leq cte$) para $cte = 1$.

Tareas a realizar en el proyecto (tarea 2)

De manera **optativa**

Se puede implementar las distancias con la variante de programación dinámica **hacia delante** con una lista de **estados activos** y un umbral `threshold` que evite generar hipótesis que superen dicho umbral.

> *Programación dinámica hacia delante: cada etapa utiliza una lista de estados activos y la recorre con un coste lineal con el nº elementos de dicha lista.*

Tareas a realizar en el proyecto (tarea 3)

En esta etapa, para calcular el conjunto de palabras del diccionario que son próximas ortográficamente a un término escrito por el usuario no tendremos más remedio que calcular la distancia entre dicho término y **cada una** de las palabras del diccionario.
Una vez completada la tarea 2 debes implementar una clase Python que reciba una referencia del diccionario y que tenga un método `suggest` con el que obtener aquellas palabras del diccionario a una distancia menor que `threshold` del término buscado. Está claro que este proceso es muy **costoso** porque require recorrer todo el diccionario, por eso se contempla la tarea 3:

3. Evitar la ejecución de la distancia de edición entre aquellas cadenas en las que algún tipo de cota optimista nos permita saber que dicha distancia será mayor al umbral establecido.

Se trata de evitar, cuando se buscan sugerencias a un término dado, el cálculo de algunas distancias de edición. Obviamente, el cálculo que determine si esa distancia puede ser evitada deberá ser mucho más rápido que la propia distancia, ya que en otro caso sería contraproducente.

Tareas a realizar en el proyecto (tarea 3)

De manera **obligatoria** se debe probar:

- La diferencia entre las longitudes de ambas cadenas.
  Ver el siguiente enlace (pinchar aquí).
- De manera **optativa** (ampliación) y únicamente para Levenshtein se puede probar la siguiente cota optimista:
  - Ordenar el alfabeto del diccionario por frecuencia y crear un grupo de 10 letras de manera que, yendo de más a menos frecuente, cada grupo tenga aproximadamente el mismo número de ocurrencias.
  - Dada una palabra, obtener un vector de talla 10 con el nº letras que tiene la palabra en cada grupo.
  - Utilizar la distancia (L1 o Manhattan) entre ese vector y otros generados a partir del diccionario para obtener una cota de la distancia.

Tareas a realizar en el proyecto (tarea 4)

4. Extender esta distancia, originalmente definida entre pares de cadenas, para calcular la distancia entre una cadena y un **trie**.

Se trata de ir, etapa por etapa, recorriendo la cadena y, en cada etapa, mantener un conjunto de estados activos donde cada nodo viene dado por:

- Nodo del trie (número),
- Distancia entre el prefijo de la palabra y el prefijo dado por el nodo del trie.

La salida de dicho método será una lista con aquellas palabras del trie que se encuentren a una distancia menor al umbral dado como parámetro.
De manera **obligatoria** debes implementarlo para Levenshtein utilizando la técnica *hacia atrás* o *backwards* (la que se utiliza normalmente en teoría y que consiste en aplicar la transformación recursiva-iterativa a las ecuaciones de recurrencia). En ese caso puedes tener el conjunto de estados activos de una etapa en forma de un vector de tipo `numpy` de tamaño en nº de estados o nodos del *Trie*. Necesitarás 2 vectores para las etapas previa y actual (que se irán intercambiando).

Tareas a realizar en el proyecto (tarea 4)

Consejos para implementar la distancia de Levenshtein cadena-Trie:

- **Cada etapa corresponde a procesar un carácter de la cadena.**
- Inicialmente distancia es 0 `root` e *infinito* (`threshold+1`) para el resto.
- Las **inserciones** de caracteres en la cadena corresponde a avanzar por el trie **en la misma etapa** por lo que has de utilizar el método `get_parent` y valores del vector que representa el estado actual.

  *Por eso es **imprescindible** recorrer los estados del Trie de menor a mayor, ya que ese recorrido garantiza que se procesan los padres antes de los hijos.*
- Los **borrados** de caracteres en la cadena corresponde a quedarse en el mismo estado del Trie (en lugar de bajar a un hijo), lo que corresponde copiar (más 1) al valor del vector que representa el estado anterior al actual.
- Las sustituciones/aciertos es avanzar al mismo tiempo en la cadena y en el Trie, lo que corresponde a usar `get_parent` y comparar la etiqueta (`get_label`) con el caracter de la etapa actual.
- Si al terminar una etapa el menor valor es `>threshold` se podría parar.
- La salida final se obtiene de los estados finales (método `is_final`) con distancia `<=threshold` y la cadena la da el método `get_output`.

Tareas a realizar en el proyecto (tarea 4)

Ampliaciones opcionales en la tarea 4

- Implementar las versiones *hacia atrás* para Damerau-Levenstein restringida e intermedia.
- Implementar Levenshtein *hacia delante*: en lugar de consultar de dónde vienes (con el método `get_parent` de la clase `Trie`) debes iterar sobre los hijos con:

```
for child_st in trie.iter_children(parent_st):
    # hacer lo que sea con child_st
```

Puedes utilizar un diccionario para guardar la lista de estado activos de cada etapa.

- Alternativamente (si haces esta versión no haría falta implementar la otra) puedes guardar la lista de estados activos en una lista de pares `(estado,distancia)` ordenado por el `estado` **en lugar de** un diccionario Python (hay una versión de la mochila que utiliza la misma técnica).

Tareas a realizar en el proyecto (tarea 5)

5. Realizar un estudio experimental de medidas de tiempo para determinar qué versiones de las anteriores son las más eficientes para unos datos concretos.

Se trata de diseñar un experimento empírico en el que se pruebe:

- Los diversos algoritmos implementados.
- Distintos diccionarios (castellano e inglés con las N palabras más frecuentes, es fácil encontrar los diccionarios de ambos idiomas con las 10K o las 100K palabras más frecuentes). Puedes recortarlo si quieres utilizar valores de N más pequeños o intermedios.
- Forma de generar errores ortográficos: se proporciona un programa que toma palabras al azar del diccionario y las "perturba" con errores de diversos tipos.

Tareas a realizar en el proyecto (tarea 5)

5. Realizar un estudio experimental de medidas de tiempo para determinar qué versiones de las anteriores son las más eficientes para unos datos concretos.

Atención

Se valorará la forma correcta de realizar esta experimentación, en particular:

- Elección de datos y barrido de tallas.
- Metodología:
  - Repetir muestreos para una misma talla.
  - Utilizar el mismo muestreo con todos los algoritmos (datos apareados).
  - Determinar si las diferencias son estadísticamente significativas.

Tareas a realizar en el proyecto (tarea 6)

6. Modificar el Indexador y el Recuperador de SAR para permitir la búsqueda aproximada de cadenas utiliznado las versiones más eficientes según el punto anterior.

Para facilitar esto, debes crear una clase que represente el corrector ortográfico de la que heredar al menos 2 implementaciones:

- La que utiliza una distancia cadena-cadena y recorre el diccionario (utiliznado quizás algún tipo de técnica para evitar algunas distancias).
- La que calcula la distancia cadena-*trie*.

Añade en el recuperador de noticias de SAR la posibilidad de activar la búsqueda aproximada de manera que, cuando uno de los términos a buscar no se encuentre en el diccionario, se recuperen las noticias asociadas a los términos que sugiera el corrector.

Material proporcionado

Material proporcionado

Para la realización de este proyecto se proporciona:

- El código de la mochila del ejercicio de medir tiempos y estados activos.
- Código para crear la estructura de datos de tipo *trie*.
- Código para generar cadenas de texto perturbadas.
- Referencias para averiguar cómo medir tiempos.
- Propuesta de una API para integrar vuestro código en el proyecto de SAR.

Atención

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

- Grupo de mañanas (L1-4CO11) contactar con Jose Angel González-Barba jogonba2@inf.upv.es
- Grupo de tardes (L1-4CO21) contactar con Salvador España-Boquera sespana@dsic.upv.es

Presentación y defensa del proyecto

Presentación y defensa del proyecto

Cada grupo/equipo debe preparar:

- Un **informe** con los resultados obtenidos y justificando las decisiones que ha adoptado. Debe ser en formato pdf (o ir acompañado de una versión pdf si se se utilizan otros formatos).
- El **código** utilizado. Se valorará la organización (incluyendo la utilización de módulos/bibliotecas), estilo, eficiencia, documentación/comentarios, etc.
- Preparar una **presentación muy breve** de los resultados. En dicha presentación deben intervenir todos los miembros del equipo y una pequeña **demo**.