

Reconocimiento Automático del Habla

Alineamiento Temporal

María José Castro

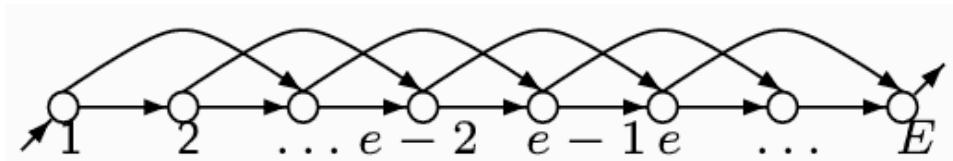
mcastro@dsic.upv.es

Un repaso de programación dinámica

En este tema y el siguiente, la programación dinámica juega un papel fundamental. Repasamos ahora algunos aspectos básicos de programación dinámica.

Un problema ilustrativo

Hay E embarcaderos a lo largo de un río. De un embarcadero e sólo se puede ir en canoa al siguiente ($e + 1$) o al que hay dos posiciones más a la derecha ($e + 2$). El transporte entre dos embarcaderos e_i y e_j cuesta $c(e_i, e_j)$ euros. ¿Cuánto cuesta el trayecto (secuencia de embarcaderos) más barato entre los embarcaderos 1 y E ?



Representación esquemática del río con sus embarcaderos.

Conjunto de posibles soluciones (**soluciones factibles**):

$$\mathcal{S} = \{(e_1, e_2, \dots, e_n) \mid e_1 = 1, e_n = E, 1 \leq e_i - e_{i-1} \leq 2, 2 \leq i \leq n\}$$

Buscamos:

$$\min_{(e_1, e_2, \dots, e_n) \mid e_1 = 1, e_n = E, 1 \leq e_i - e_{i-1} \leq 2, 2 \leq i \leq n} \sum_{i=2}^n c(e_{i-1}, e_i) = \text{coste}(E).$$

Exploración exhaustiva

- Enumerar todos los caminos válidos entre 1 y E .
- Evaluar el coste de cada camino.
- Escoger el de menor coste.

Coste temporal: $O(2^E)$.

Formulación en términos de ecuación recursiva

$$\begin{aligned}\text{coste}(E) &= \min_{(e_1, e_2, \dots, e_n) | e_1 = 1, e_n = E, 1 \leq e_i - e_{i-1} \leq 2, 2 \leq i \leq n} \sum_{i=2}^n c(e_{i-1}, e_i) \\ &= \min_{(e_1, e_2, \dots, e_{n-1}) | e_1 = 1, 1 \leq e_i - e_{i-1} \leq 2, 2 \leq i \leq n-1} \left(\sum_{i=2}^{n-1} c(e_{i-1}, e_i) + c(e_{n-1}, E) \right)\end{aligned}$$

El embarcadero e_{n-1} sólo puede ser $E - 1$ o $E - 2$

$$\text{coste}(E) = \min \left\{ \begin{array}{l} \min_{(e_1, e_2, \dots, E-1) | e_1 = 1, 1 \leq e_i - e_{i-1} \leq 2, 2 \leq i \leq n-1} \sum_{i=2}^{n-1} c(e_{i-1}, e_i) + c(E-1, E) \\ \min_{(e_1, e_2, \dots, E-2) | e_1 = 1, 1 \leq e_i - e_{i-1} \leq 2, 2 \leq i \leq n-1} \sum_{i=2}^{n-1} c(e_{i-1}, e_i) + c(E-2, E) \end{array} \right.$$

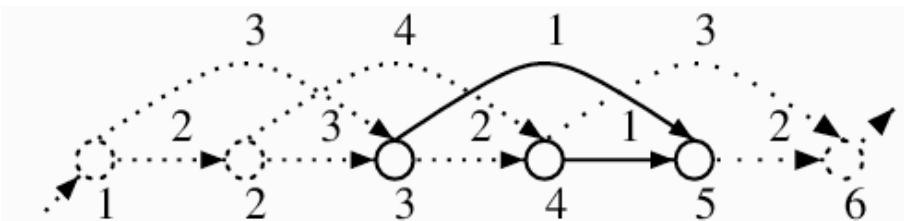
Las minimizaciones internas se pueden expresar como $\text{coste}(E - 1)$ y $\text{coste}(E - 2)$:

$$\text{coste}(E) = \min \left\{ \begin{array}{l} \text{coste}(E - 1) + c(E - 1, E) \\ \text{coste}(E - 2) + c(E - 2, E) \end{array} \right.$$

En general:

$$\text{coste}(e) = \begin{cases} 0, & \text{si } e = 1; \\ c(1, 2), & \text{si } e = 2; \\ \min\{\text{coste}(e - 1) + c(e - 1, e), \text{coste}(e - 2) + c(e - 2, e)\}, & \text{si } e > 2. \end{cases}$$

El coste del trayecto óptimo es $\text{coste}(E)$.



Elementos implicados en la ecuación recursiva de coste(5).

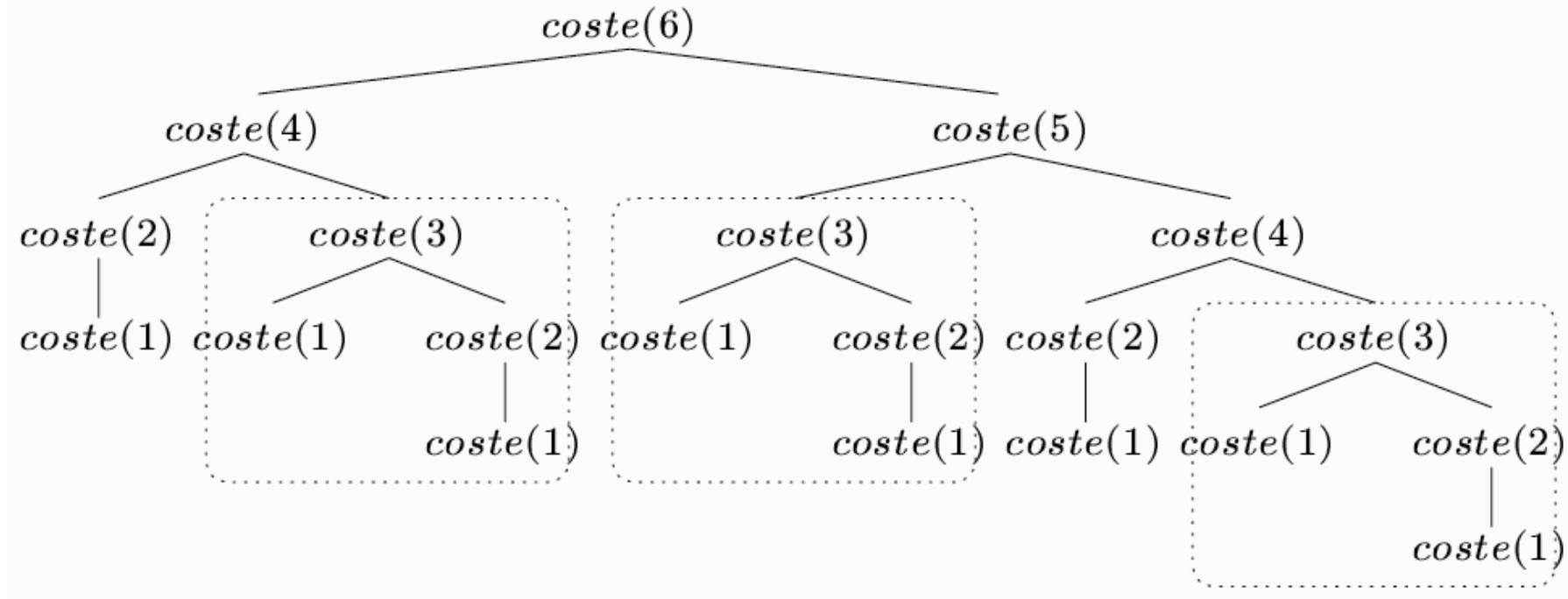
Implementación directa del cálculo recursivo

```
c = {(1,2): 2, (1,3): 3,
      (2,3): 3, (2,4): 4,
      (3,4): 2, (3,5): 1,
      (4,5): 1, (4,6): 3,
      (5,6): 2 }

def coste_rec(e, c):
    if e == 1:
        return 0
    elif e == 2:
        return c[1,2]
    else:
        return min(coste_rec(e-1, c) + c[e-1, e], \
                   coste_rec(e-2, c) + c[e-2, e] )

print 'Coste del trayecto mas barato entre 1 y 6: ', coste_rec(6,c)
```

El coste temporal crece exponencialmente. Problema: la repetición de cálculos es una fuente de ineficiencia.



Árbol de llamadas recursivas. El valor de $\text{coste}(3)$, por ejemplo, se repite tres veces. El de $\text{coste}(2)$, 5 veces.

Versión recursiva con memorización Es absurdo repetir cálculos. Podemos “memorizar” los resultados la primera vez que se piden:

```
c = {(1,2): 2, (1,3): 3,
      (2,3): 3, (2,4): 4,
      (3,4): 2, (3,5): 1,
      (4,5): 1, (4,6): 3,
      (5,6): 2 }
```

```
def coste_memo(e, c, coste = {}):
    if e == 1:
        return 0
    elif e == 2:
        return c[1, 2]
    else:
        if not coste.has_key(e-1): coste[e-1] = coste_memo(e-1, c, coste)
        if not coste.has_key(e-2): coste[e-2] = coste_memo(e-2, c, coste)
        coste[e] = min( coste[e-1] + c[e-1, e], coste[e-2] + c[e-2, e] )
    return coste[e]
```

```
print 'Coste del trayecto mas barato entre 1 y 6: ', coste_memo(6,c)
```

Tiempo: $O(E)$. Espacio: $O(E)$.

Recordar los valores ya calculados evita tener que repetir cálculos.

Transformación recursivo-iterativa

Es habitual efectuar una conversión recursivo-iterativa

```
c = {(1,2): 2, (1,3): 3,
      (2,3): 3, (2,4): 4,
      (3,4): 2, (3,5): 1,
      (4,5): 1, (4,6): 3,
      (5,6): 2 }

def coste_iter(E, c):
    coste = []
    coste[1] = 0
    coste[2] = c[1,2]
    for e in range(3, E+1):
        coste[e] = min( coste[e-1] + c[e-1, e], coste[e-2] + c[e-2, e] )
    return coste[E]

print 'Coste del trayecto mas barato entre 1 y 6: ', coste_iter(6,c)
```

Tiempo: $O(E)$. Espacio: $O(E)$.

En lugar de un diccionario, y gracias a que los nodos del grafo sirven de índice de un vector (son simples enteros), podemos usar un vector:

```
c = {(1,2): 2, (1,3): 3,
      (2,3): 3, (2,4): 4,
      (3,4): 2, (3,5): 1,
      (4,5): 1, (4,6): 3,
      (5,6): 2 }

def coste_iter2(E, c):
    coste = [0] * (E+1)
    coste[1] = 0
    coste[2] = c[1,2]
    for e in range(3, E+1):
        coste[e] = min( coste[e-1] + c[e-1, e], coste[e-2] + c[e-2, e] )
    return coste[E]

print 'Coste del trayecto mas barato entre 1 y 6: ', coste_iter2(6,c)
```

Tiempo: $O(E)$. Espacio: $O(E)$.

Reducción del coste espacial Es innecesario recordar todas las soluciones anteriores: basta con las dos últimas.

```
c = {(1,2): 2, (1,3): 3,
      (2,3): 3, (2,4): 4,
      (3,4): 2, (3,5): 1,
      (4,5): 1, (4,6): 3,
      (5,6): 2 }
```

```
def coste_iter3(E, c):
    if E == 1: return 0
    if E == 2: return c[1,2]
    coste_2= 0
    coste_1= c[1,2]
    for e in range(3, E+1):
        coste = min( coste_1+ c[e-1, e], coste_2+ c[e-2, e] )
        coste_2, coste_1= coste_1, coste
    return coste

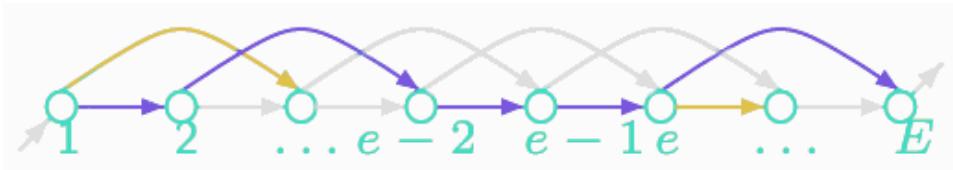
print 'Coste del trayecto mas barato entre 1 y 6: ', coste_iter3(6,c)
```

Tiempo: $O(E)$. Espacio: $O(1)$.

Recuperación del camino

Podemos anotar en cada nodo cuál es el origen de su nodo anterior en el camino óptimo.

```
def camino_optimo(E, c):
    coste = [0] * (E+1)
    anterior = [None] * (E+1)
    coste[1] = 0
    coste[2] = c[1,2]
    anterior[1] = None
    anterior[2] = 1
    for e in range(3, E+1):
        if coste[e-1] + c[e-1, e] <= coste[e-2] + c[e-2, e]:
            coste[e] = coste[e-1] + c[e-1, e]
            anterior[e] = e-1
        else:
            coste[e] = coste[e-2] + c[e-2, e]
            anterior[e] = e-2
    ...
    ...
```



Punteros hacia atrás. En cada nodo se recuerda el arco que proporcionó el peso óptimo hasta él. Si se sigue desde el último nodo el trayecto de estos de estos arcos, se obtiene el camino de peso óptimo.

Y ahora podemos recuperar el camino siguiendo los enlaces de anterior (técnica denominada *backtracing*):

```
c = {(1,2): 2, (1,3): 3,
      (2,3): 3, (2,4): 4,
      (3,4): 2, (3,5): 1,
      (4,5): 1, (4,6): 3,
      (5,6): 2 }
```

```
def camino_optimo(E, c):
    coste = [0] * (E+1)
    anterior = [None] * (E+1)
    coste[1] = 0
    coste[2] = c[1,2]
```

```

anterior[1] = None
anterior[2] = 1
for e in range(3, E+1):
    if coste[e-1] + c[e-1, e] <= coste[e-2] + c[e-2, e]:
        coste[e] = coste[e-1] + c[e-1, e]
        anterior[e] = e-1
    else:
        coste[e] = coste[e-2] + c[e-2, e]
        anterior[e] = e-2
camino = [E]
while anterior[ camino[-1] ] != None:
    camino.append( anterior[ camino[-1] ] )
camino.reverse()
return camino

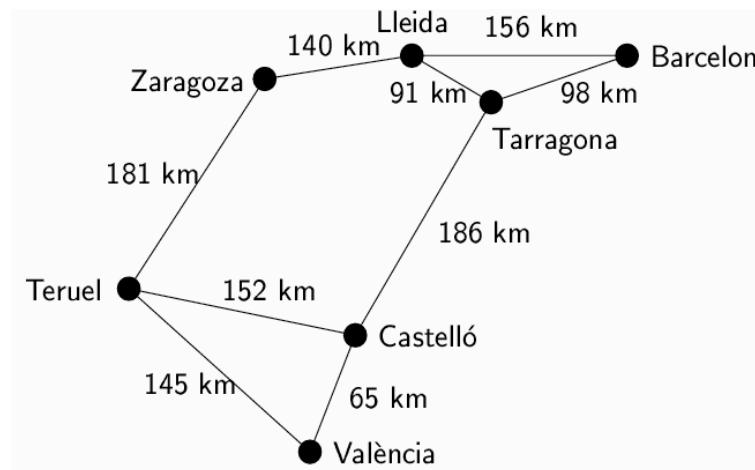
print 'Trayecto mas barato entre 1 y 6: ', camino_optimo(6,c)

```

Cuándo es aplicable con éxito la programación dinámica

- Cuando una resolución recursiva del problema conduce a la repetición de cálculos.
- Cuando la solución al problema se puede componer con “subestructuras óptimas” (principio de optimalidad):

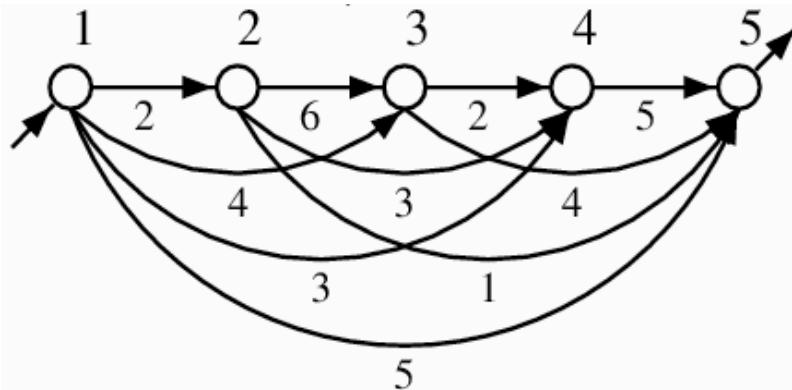
Por ejemplo, en este plano de ciudades se observa que si el camino óptimo de Valencia a Barcelona pasa por Castellón y Tarragona, el camino óptimo de Valencia a Tarragona pasa por Castellón.



La ruta óptima entre dos ciudades de compone de “sub-rutas” óptimas desde la ciudad origen a cada uno de los destinos intermedios.

Una variante del problema anterior

Supón que es posible llegar a cada embarcadero desde cualquier embarcadero anterior. ¿Cuál es el coste de la ruta óptima?



Variante en la que es posible llegar a cada embarcadero desde cualquier embarcadero anterior.

$$\text{coste}(e) = \begin{cases} 0, & \text{si } e = 1; \\ \min_{1 \leq e' < e} (\text{coste}(e') + c(e', e)), & \text{si } e > 2. \end{cases}$$

```

c = {(1,2): 2, (1,3): 4, (1,4): 3, (1,5): 5,
      (2,3): 6, (2,4): 3, (2,5): 1,
      (3,4): 2, (3,5): 4,
      (4,5): 5 }

def coste_iter(E, c):
    coste = [0] * (E+1)
    coste[1] = 0
    for e in range(2, E+1):
        coste[e] = min( [ coste[i] + c[i, e] for i in range(1, e) ] )
    return coste[E]

print 'Coste del trayecto mas barato entre 1 y 5: ', coste_iter(5,c)

```

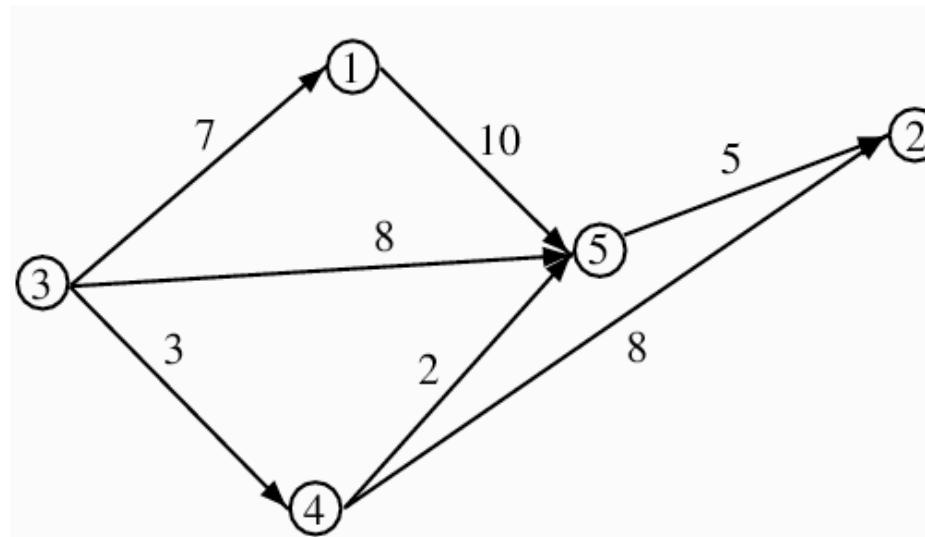
Tiempo: $O(E^2)$. Espacio: $O(E)$.

En este caso no es posible reducir el consumo espacial. La reducción del coste espacial depende de las relaciones entre nodos del grafo.

El problema del camino óptimo en un grafo acíclico

$G = (V, A, p)$ grafo dirigido, acíclico y ponderado

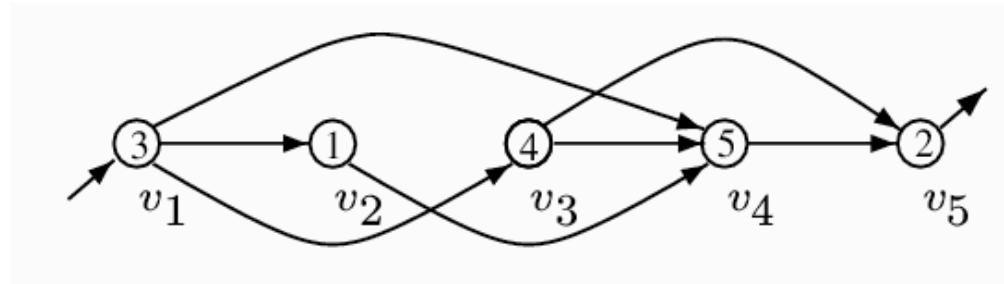
- V conjunto de vértices o nodos
- A conjunto de arcos o aristas $A \subseteq V \times V$
- $p : A \rightarrow \mathbb{R}^{\geq 0}$



Grafo acíclico.

Problema: Encontrar el camino más corto de un nodo origen $s \in V$ a un nodo destino $t \in V$.

Primer paso: encontrar un orden topológico.



Grafo con nodos ordenados topológicamente.

Si v es alcanzado desde u , $(u, v) \in A$, entonces $u < v$ (se puede obtener con coste $O(|V| + |A|)$).

Supondremos que $s = v_1$ y $t = v_n$.

Se desea calcular el **coste del camino mínimo**:

$$\min_{(c_1 c_2 \dots c_n) | c_1 = v_1, c_n = v_N, c_{i-1} \in \text{pred}(c_i), 2 \leq i \leq n} \sum_{i=2}^n p(c_{i-1}, c_i)$$

Planteado en estos términos, el problema es una nueva variante del problema del río.

Ecuación recursiva:

$$F(v) = \begin{cases} \min_{u \in \text{pred}(v)} (F(u) + p(u, v)), & \text{si } v > v_1; \\ 0, & \text{si } v = v_1. \end{cases}$$

```
import copy

class GrafoAciclico:
    def __init__(self, p):
        self.succ = {}
        self.pred = {}
        for i, j in p.keys():
            try: self.succ[i].append(j)
            except: self.succ[i] = [j]
            try: self.pred[j].append(i)
            except: self.pred[j] = [i]
        self.p = p
        self.topsort = self.topological_sort()
```

```

def topological_sort(self):
    numpreds = {}
    succ = copy.deepcopy(self.succ)
    for i, j in self.p.keys(): numpreds[i] = numpreds[j] = 0
    for i, j in self.p.keys(): numpreds[j] += 1
    starters = filter(lambda x, n=numpreds: n[x] == 0, numpreds.keys())
    for x in starters:
        del numpreds[x]
        for y in succ.get(x, []):
            numpreds[y] -= 1
            if numpreds[y] == 0: starters.append( y )
        if succ.has_key(x): del succ[x]
    return starters

```

```

def peso_optimo(self, f):
    F = {}
    for v in self.topsort:
        if not self.pred.has_key(v): F[v] = 0
        else: F[v] = min( [ F[u] + self.p[u,v] for u in self.pred[v] ] )

```

```

    if v == f: break
    return F.get(v, -1)

def camino_optimo(self, f):
    F = {}
    B = {}
    for v in self.topsort:
        if not self.pred.has_key(v): F[v], B[v] = 0, None
        else:
            F[v], B[v] = min([(F[u]+self.p[u,v], u) for u in self.pred[v]])
        if v == f: break
    camino = [f]
    while B[camino[-1]] != None:
        camino.append(B[camino[-1]])
    camino.reverse()
    return camino

G = GrafoAciclico( {(3,1):7, (3,4):3, (3,5):8,
                     (1,5):10,

```

```
(4,5):2, (4,2):8,  
(5,2):5} )
```

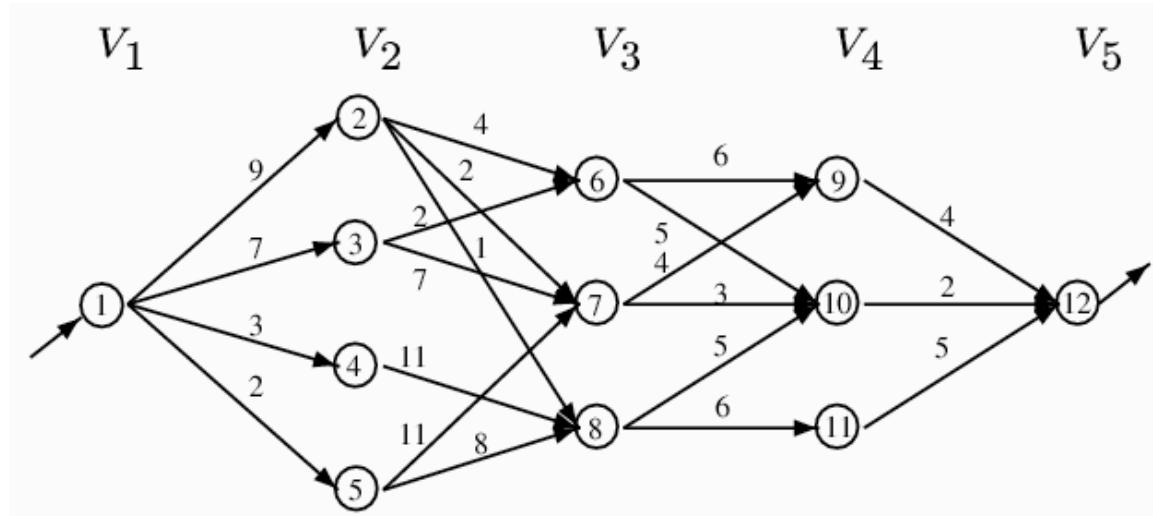
```
print G.peso_optimo(2)  
print G.camino_optimo(2)
```

El problema del camino óptimo en un grafo multietapa

$G = (V, A, p)$ un grafo dirigido, acíclico y ponderado

- $V = \bigcup_{i=1}^K V_i$ con $V_i \cap V_j = \emptyset, i \neq j$.
- $A \subseteq \{(u, v) \in V^2 \mid \exists i : u \in V_i, v \in V_{i+1}\}$.
- $p : A \rightarrow \mathbb{R}^{\geq 0}$.
- $\mathcal{I} \subseteq V_1$ vértices iniciales y $\mathcal{F} \subseteq V_K$ finales.

Cada conjunto V_i recibe el nombre de **etapa**.



Grafo multietapa.

Problema: Camino de coste mínimo entre un nodo cualquiera de \mathcal{I} y otro de \mathcal{F} .

Espacio de búsqueda:

$$\mathcal{S} = \{(u_1 u_2 \dots u_K) \in V^K \mid u_1 \in \mathcal{I}, u_K \in \mathcal{F}, u_i \in V_i, 1 \leq i \leq K, u_{j-1} \in \text{pred}(u_j), 2 \leq j \leq K\}$$

Buscamos:

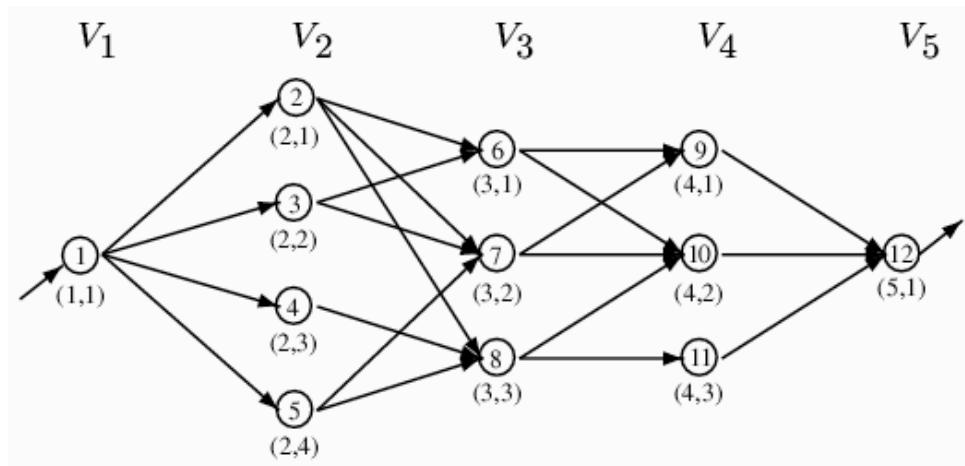
a) el peso del camino óptimo:

$$\min_{(u_1 \dots u_K) \in \mathcal{S}} \sum_{j=2}^K p(u_{j-1}, u_j)$$

b) y también dicho camino:

$$\arg \min_{(u_1 \dots u_K) \in \mathcal{S}} \sum_{j=2}^K p(u_{j-1}, u_j)$$

- Se trata de un **grafo acíclico**.
- Cualquier ordenamiento de nodos que siga el **orden impuesto por las etapas** es un orden topológico.
- Por conveniencia, renombraremos los nodos y nos referiremos a ellos con un par (i, j) , donde i es el número de etapa y j es un número de lo identifica dentro de la etapa correspondiente.



Renombramiento de nodos.

Ecuación recursiva:

$$F(k, v) = \begin{cases} 0, & \text{si } (1, v) \in \mathcal{I}; \\ +\infty, & \text{si } (1, v) \in V_1 - \mathcal{I}; \\ \min_{(k-1, u) \in \text{pred}(k, v)} (M(k-1, u) + p((k-1, u), (k, v))), & \text{si } k > 1 \end{cases}$$

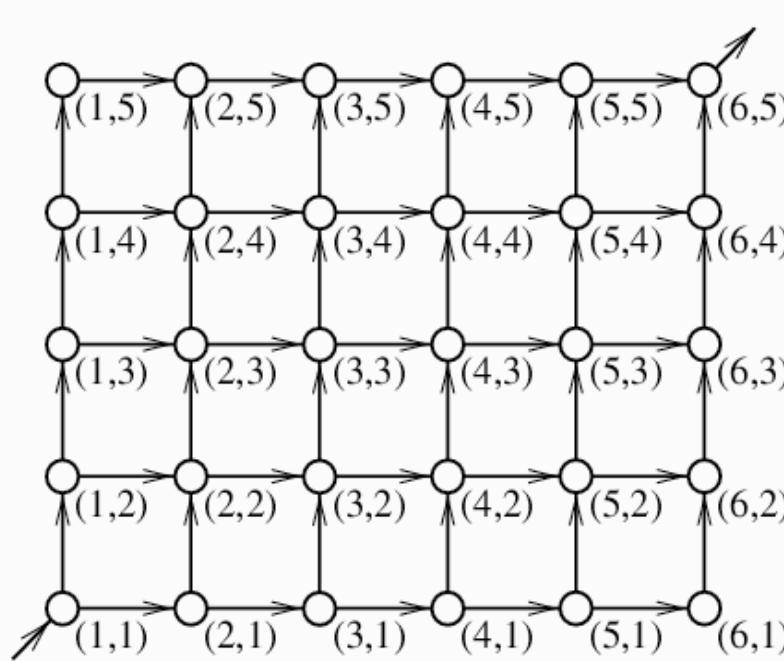
El peso óptimo es

$$\min_{(K, v) \in \mathcal{F}} F(K, v)$$

Tiempo: $O(K \max_{1 \leq k \leq K} |V_k|^2)$. Espacio = $O(K \max_{1 \leq k \leq K} |V_k|)$.

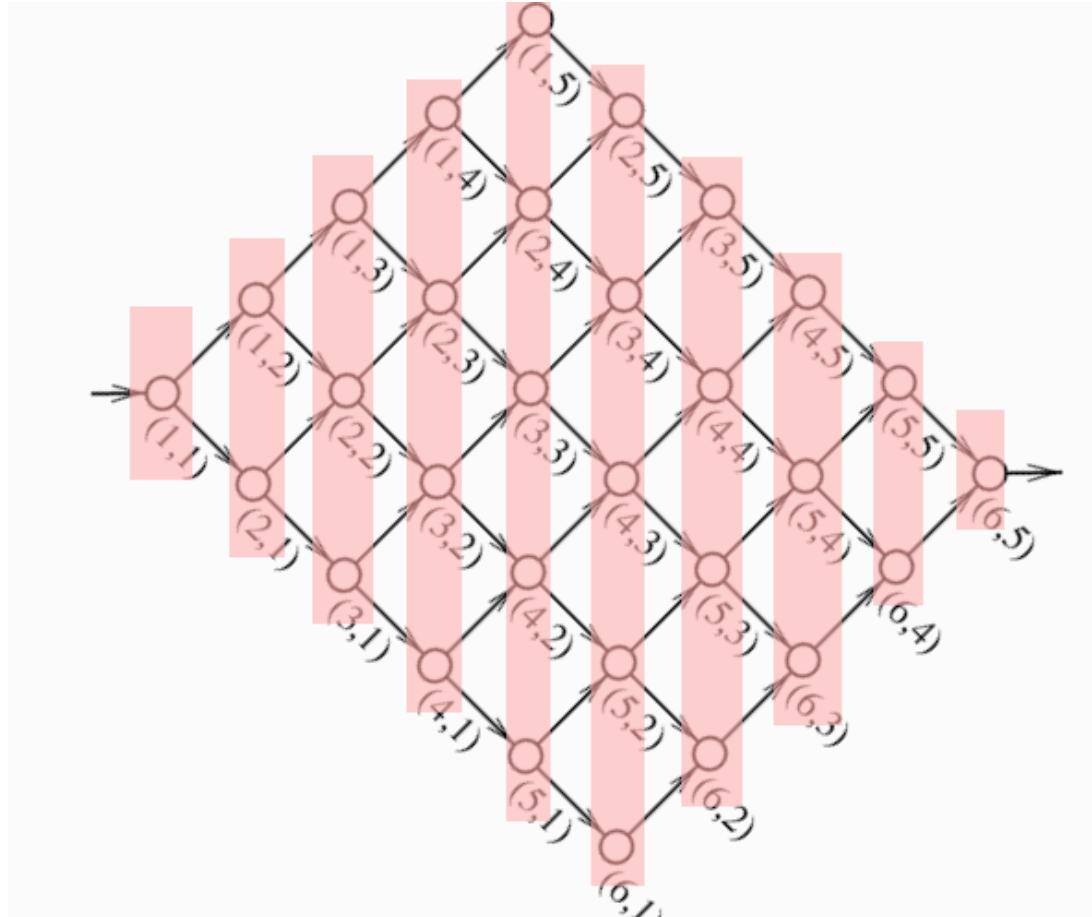
El camino más corto en una cuadrícula

Disponemos de un grafo cuyos nodos están etiquetados por elementos de $\{1, 2, \dots, I\} \times \{1, 2, \dots, J\}$. Los nodos de la forma (i, j) están conectados a los nodos $(i + 1, j)$ y $(i, j + 1)$ (a excepción de los nodos en los extremos superior y derecho de la cuadrícula). Cada arco $((i, j), (i', j'))$ presenta un peso $p((i, j), (i', j'))$. Deseamos conocer el menor peso con el que es posible ir del nodo $(1, 1)$ al (I, J) .



Cuadrícula.

Se trata de un caso particular de grafo multietapa, así que podemos resolverlo aplicando las técnicas del apartado anterior.



Cuadrícula como grafo multietapa.

Pero una formulación específica casi resulta más sencilla:

Soluciones factibles:

$$\begin{aligned}\mathcal{S}(I, J) = \{ & ((i_1, j_1), \dots, (i_n, j_n)) \mid (i_1, j_1) = (1, 1), \\ & (i_n, j_n) = (I, J), \\ & (i_k - i_{k-1}, j_k - j_{k-1}) = (1, 0) \\ & \vee (i_k - i_{k-1}, j_k - j_{k-1}) = (0, 1), \\ & 2 \leq k \leq n \} \end{aligned}$$

Buscamos:

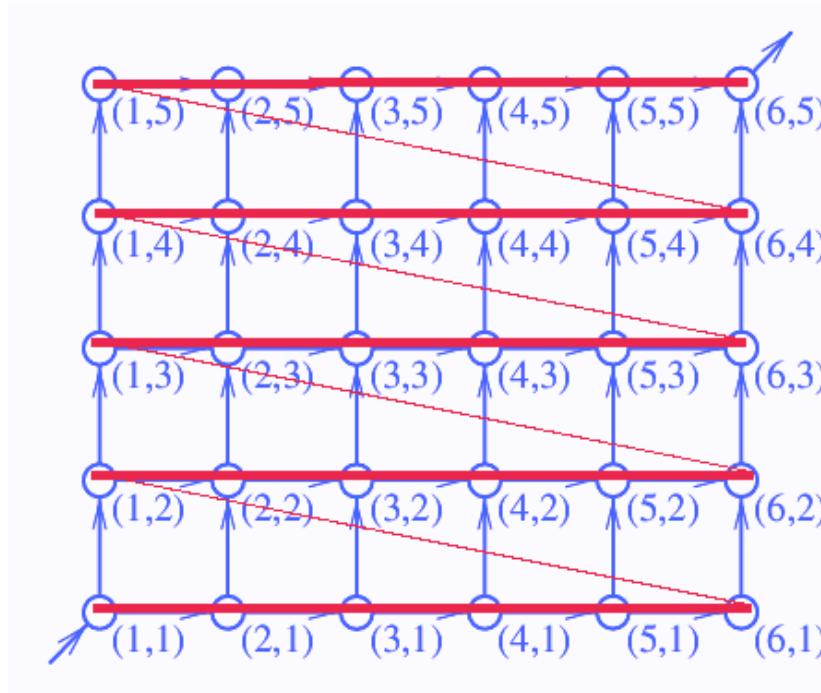
$$\min_{((i_1, j_1), \dots, (i_n, j_n)) \in \mathcal{S}(I, J)} \sum_{k=2}^n p((i_{k-1}, j_{k-1}), (i_k, j_k)) = P(i, j)$$

Ecuación recursiva:

$$P(i, j) = \min \begin{cases} 0, & \text{si } i = 1 \text{ y } j = 1; \\ P(1, j - 1) + p((1, j - 1), (1, j)) & \text{si } i = 1 \text{ y } j > 1; \\ P(i - 1, 1) + p((i - 1, 1), (i, 1)), & \text{si } i > 1 \text{ y } j = 1; \\ \min \left\{ \begin{array}{l} P(i - 1, j) + p((i - 1, j), (i, j)), \\ P(i, j - 1) + p((i, j - 1), (i, j)), \end{array} \right\}, & \text{si } i > 1 \text{ y } j > 1. \end{cases}$$

Solución en $P(I, J)$.

Solución iterativa: recorrido por filas o columnas.



Recorrido por filas de la cuadrícula.

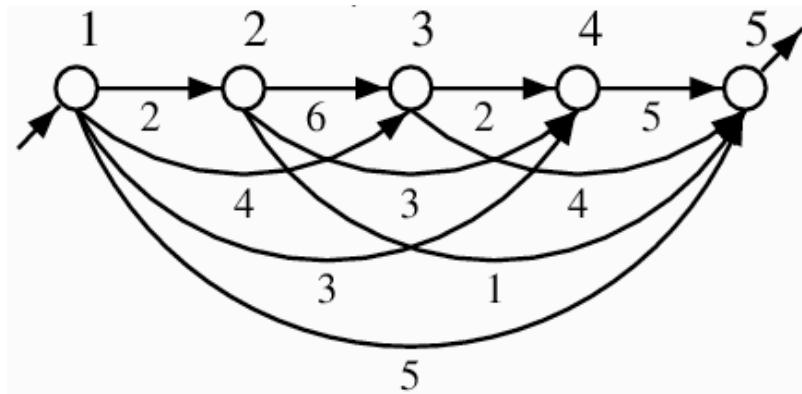
Tiempo: $O(IJ)$. Espacio $O(IJ)$.

Reducción de espacio: memorización de filas (o columnas) actual y previa $O(I)$ u $O(J)$.

Camino más corto con K tramos en un grafo

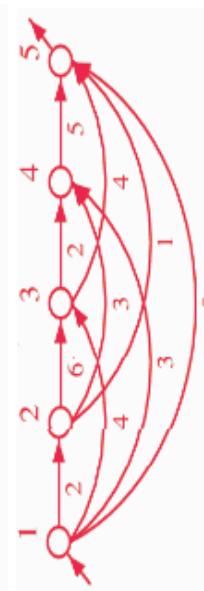
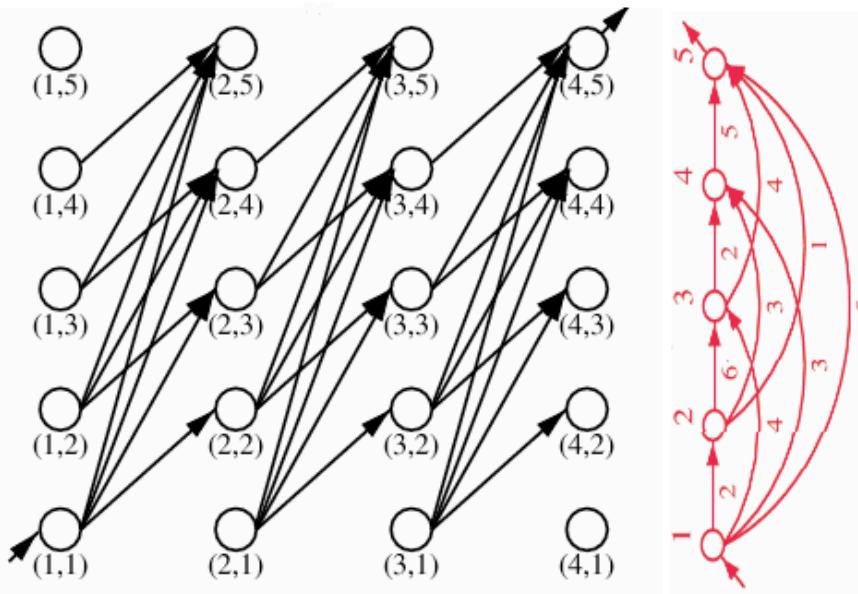
Se desea calcular el camino más corto que esté compuesto por K arcos (o sea, $K + 1$ nodos).

El grafo sobre el que deseamos efectuar la búsqueda:



Grafo.

- Se transforma en un grafo multietapa con K etapas.
- Cada etapa contiene tantos vértices como nodos tiene el grafo original.
- Cada arco en el grafo original se traduce en $K - 1$ arcos conectando nodos de etapas contiguas.



Grafo multietapa asociado el grafo anterior.

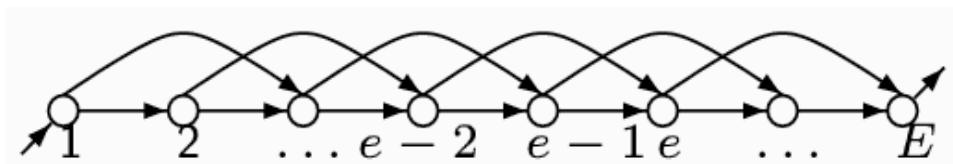
Tiempo: $O(KV^2)$. Espacio: $O(V)$.

Programación dinámica en problemas de no optimización

Aunque la programación dinámica se considera una técnica de optimización, los principios que la rigen se pueden aplicar a ciertos problemas de no optimización.

Ejemplo: cálculo de los números de Fibonacci.

$$F_n = \begin{cases} 1, & \text{si } n = 0 \text{ o } n = 1; \\ F_{n-2} + F_{n-1}, & \text{en otro caso.} \end{cases}$$



Grafo subyacente al cálculo de números de Fibonacci.

Versión recursiva:

```
def recursive_fibonacci (n):  
    if n == 0 or n == 1:  
        return 1  
    else:  
        return recursive_fibonacci(n-2) + recursive_fibonacci(n-1)  
  
print 'Fibonacci de 6: ', recursive_fibonacci(6)
```

Tiempo: $O(2^n)$. Espacio: $O(n)$.

Versión iterativa:

```
def iterative_fibonacci_1(n):
    f = [1, 1] + [0] * (n-1)
    for i in range(2, n+1):
        f[i] = f[i-2] + f[i-1]
    return f[n]

print 'Fibonacci de 6: ', iterative_fibonacci_1(6)
```

Tiempo: $O(n)$. Espacio: $O(n)$.

Versión iterativa con reducción de coste espacial:

```
def iterative_fibonacci_2(n):  
    a, b = 1, 1  
    for i in range(2, n+1):  
        a, b = b, a+b  
    return b  
  
print 'Fibonacci de 6: ', iterative_fibonacci_2(6)
```

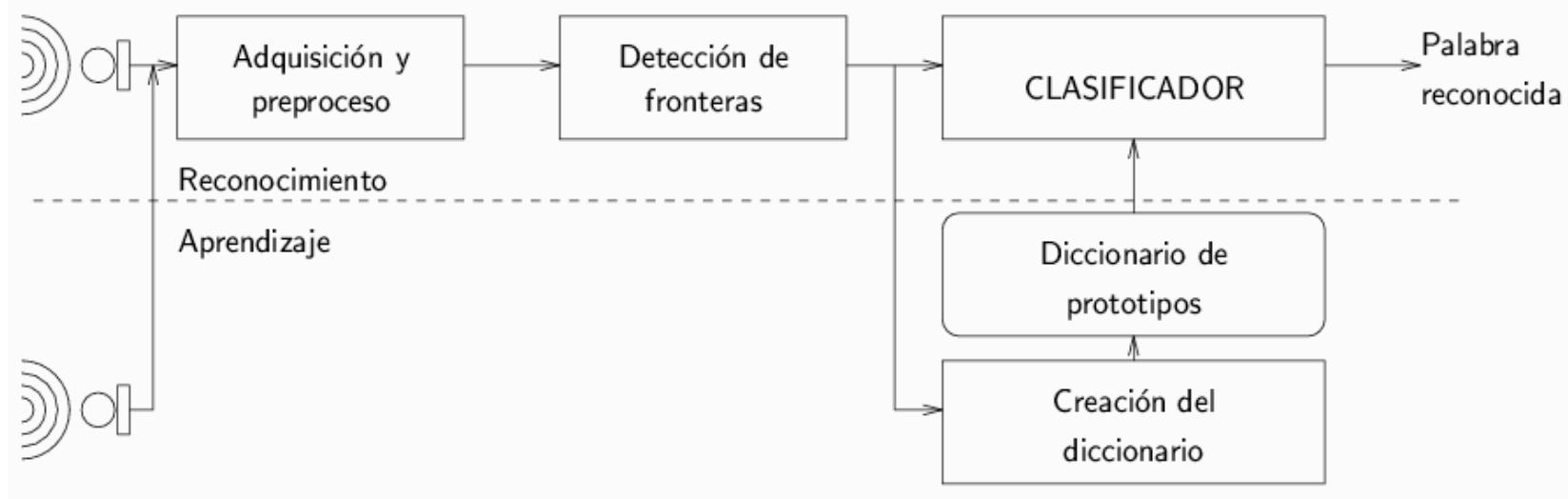
Tiempo: $O(n)$. Espacio: $O(1)$.

Reconocimiento de palabras aisladas

Arquitectura

Vamos a diseñar un sistema de reconocimiento para palabras pronunciadas individualmente y pertenecientes a un vocabulario conocido.

Cada palabra se representará con una o más pronunciaciones concretas de la misma (*plantillas*).



Etapas en un sistema de reconocimiento de palabras aisladas basado en plantillas.

Notación y definiciones

Cada pronunciación será una cadena de vectores de parámetros (o de etiquetas provenientes de un proceso de cuantificación vectorial).

$$X = x_1 x_2 \dots x_n \in \Sigma^+$$

$$Y = y_1 y_2 \dots y_m \in \Sigma^+$$

Los vectores pueden compararse entre sí mediante una función $d : \Sigma \times \Sigma \rightarrow \mathbb{R}$ que denominaremos *distancia local*. Suele ser la **distancia euclídea** entre los vectores comparados o, si se trata de etiquetas obtenidas por cuantificación vectorial, distancias precalculadas.

¿Cómo se define una distancia \mathcal{D} entre X e Y ?

Si $n = m$ podemos, en principio, definir

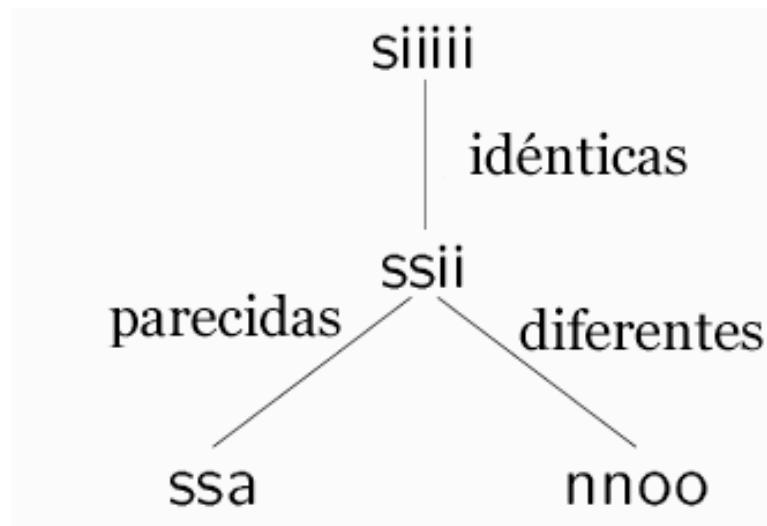
$$\mathcal{D}(X, Y) = \sum_{k=1}^n d(x_k, y_k)$$

Raramente son de la misma longitud. Posible solución: **normalización de las duraciones** interpolando $Y' = y'_1 y'_2 \dots y'_n$ a partir de Y .

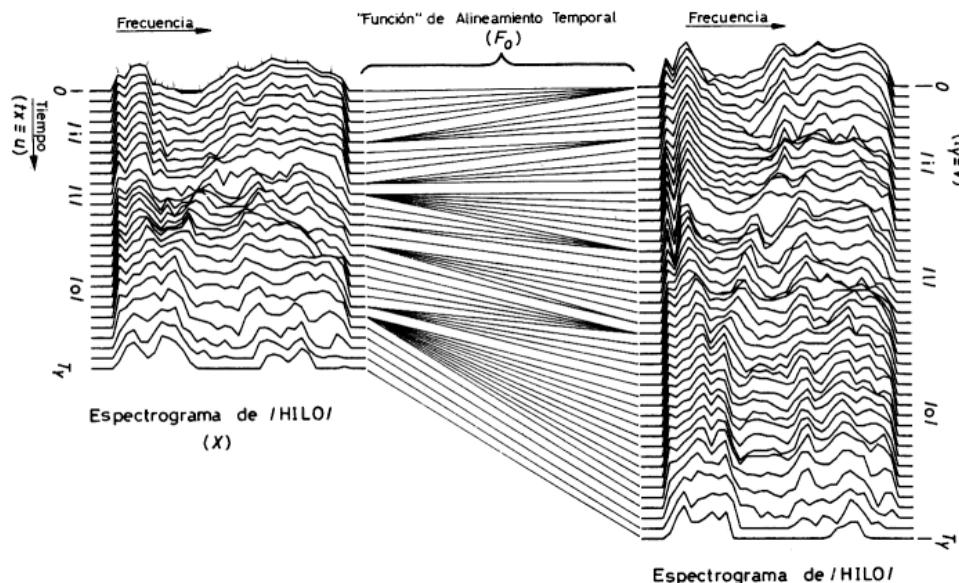
$$\mathcal{D}(X, Y) = \sum_{k=1}^n d(x_k, y'_k)$$

Deseamos definir un criterio de distancia con las siguientes características:

- Vale 0 si dos pronunciaciones son idénticas.
- Vale ∞ (idealmente) si dos pronunciaciones son “totalmente” diferentes.
- Que considere idénticas las pronunciaciones que sólo difieren en la duración de los fonemas, y parecidas las que varían en pocos fonemas.

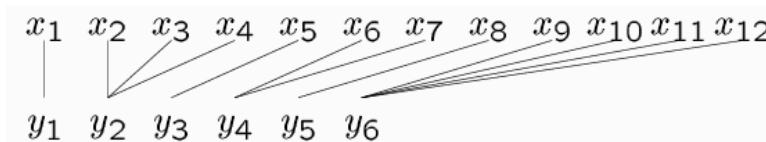


La normalización de duraciones no es una buena idea: la relación entre secuencias de tramas no es lineal.



Alineamiento no lineal entre dos pronunciacin de "hilo".

Si trasladamos el problema al alineamiento de secuencias de etiquetas se puede ver que el mejor alineamiento asocia grupos de etiquetas de forma no lineal:



Alineamiento óptimo.

¿Qué es un alineamiento?

Un alineamiento es una secuencia de pares de índices:

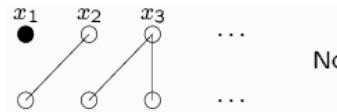
$$(1, 1)(2, 2)(3, 2)(4, 2)(5, 3)(6, 4)(7, 4)(8, 5)(9, 6)(10, 6)(11, 6)(12, 6)$$

El conjunto de alineamientos es:

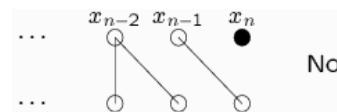
$$\mathcal{A}(X, Y) = \{(i_1, j_1) \dots (i_p, j_p) \mid 1 \leq i_k \leq n, 1 \leq j_k \leq m, 1 \leq k \leq p\}$$

Se pueden imponer ciertas restricciones “naturales” a los alineamientos:

- $(i_1, j_1) = (1, 1)$.



- $(i_p, j_p) = (n, m)$.

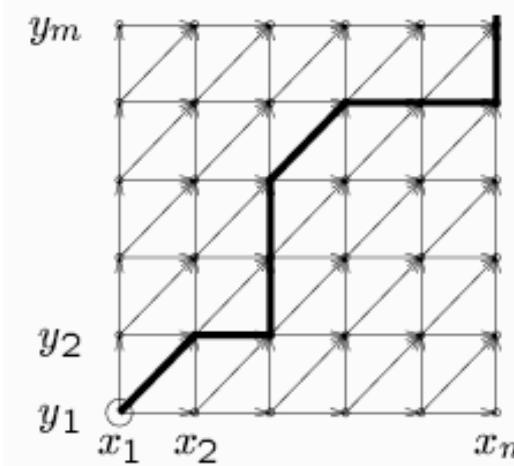


- $0 \leq i_k - i_{k-1} \leq 1$ y $0 \leq j_k - j_{k-1} \leq 1$, para $1 \leq k \leq p$.



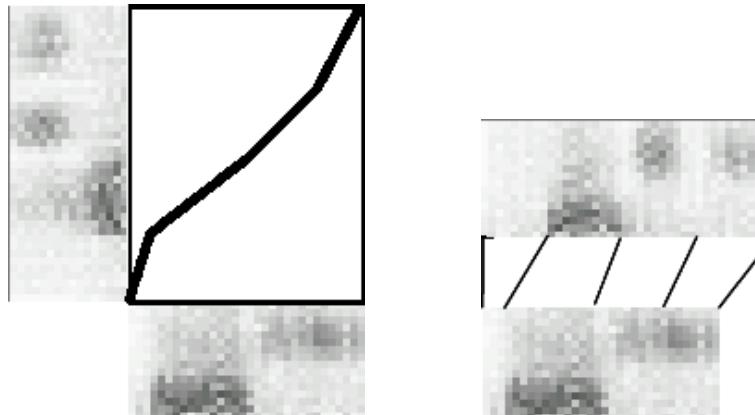
$$\begin{aligned} \mathcal{A}(X, Y) = \{ & (i_1, j_1) \dots (i_p, j_p) \mid 1 \leq i_k \leq n, 1 \leq j_k \leq m, 1 \leq k \leq p; \\ & (i_1, j_1) = (1, 1); (i_p, j_p) = (n, m); \\ & (i_k - i_{k-1}) \in \{(1, 0), (0, 1), (1, 1)\}, 1 \leq k \leq p \}. \end{aligned}$$

Los alineamientos son caminos entre el nodo $(1, 1)$ y el nodo (n, m) en un grafo reticulado:

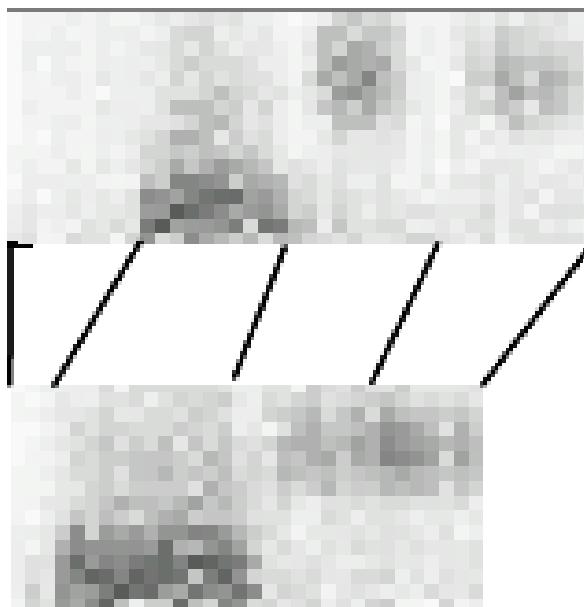


Retícula de alineamientos con alineamiento en trazo grueso.

¿Qué es un buen alineamiento? ¿Cuál es el alineamiento óptimo?



Camino de alineamiento óptimo entre dos pronunciaciones. En este caso, cada símbolo de X e Y es un vector de parámetros.



Un alineamiento es mejor que otro si presenta una suma de distancias locales menor:



Alineamientos.

- Primer alineamiento: $d(a, a) + d(a, b) + d(b, b) = 1$.
- Segundo alineamiento: $d(a, a) + d(a, a) + d(b, b) + d(b, b) = 0$.

El alineamiento óptimo es el que hace mínima la suma de distancias locales:

$$\arg \min_{(i_1, j_1), \dots, (i_p, j_p) \in \mathcal{A}(X, Y)} \sum_{k=1}^p d(x_{i_k}, y_{j_k})$$

Definimos la **distancia** de **alineamiento temporal no lineal** (ATNL)^a entre dos pronuncias como

$$\mathcal{D}(X, Y) = \min_{(i_1, j_1), \dots, (i_p, j_p) \in \mathcal{A}(X, Y)} \sum_{k=1}^p d(x_{i_k}, y_{j_k})$$

^aDTW, del inglés, “Dynamic Time Warping”.

Como un alineamiento es un camino en un grafo, calcular el alineamiento óptimo es calcular el camino óptimo en un grafo.

Cálculo (del peso) del alineamiento óptimo

El tamaño de $\mathcal{A}(X, Y)$ hace prohibitiva una aproximación por fuerza bruta. Esta definición recursiva permite plantear el problema en términos de programación dinámica:

$\mathcal{D}(X, Y) = D(n, m)$ donde

$$D(i, j) = \begin{cases} d(x_1, y_1), & \text{si } i = 1 \wedge j = 1; \\ D(i - 1, 1) + d(x_i, y_1), & \text{si } i > 1 \wedge j = 1; \\ D(1, j - 1) + d(x_1, y_j), & \text{si } i = 1 \wedge j > 1; \\ \min \left\{ \begin{array}{l} D(i, j - 1) + d(x_i, y_j), \\ D(i - 1, j) + d(x_i, y_j), \\ D(i - 1, j - 1) + d(x_i, y_j) \end{array} \right\} & \text{si } i > 1 \wedge j > 1. \end{cases}$$

```
x = "0123456789"
```

```
y = "02468"
```

```
def d(a, b): return a != b
```

```
def D(x, y, d):
```

```
    F = [ [0] * len(y) for i in range(len(x)) ]
```

```
    F[0][0] = d(x[0], y[0])
```

```
    for i in range(1, len(x)):
```

```
        F[i][0] = F[i-1][0] + d(x[i], y[0])
```

```
    for j in range(1, len(y)):
```

```
        F[0][j] = F[0][j-1] + d(x[0], y[j])
```

```
    for i in range(1, len(x)):
```

```
        for j in range(1, len(y)):
```

```
            F[i][j] = min( F[i][j-1] + d(x[i], y[j]),
```

```
                           F[i-1][j] + d(x[i], y[j]),
```

```
                           F[i-1][j-1] + d(x[i], y[j]) )
```

```
    return F[len(x)-1][len(y)-1]
```

```
print 'Distancia entre x e y: ', D(x,y,d)
```

Complejidad temporal: $O(nm)$.

Complejidad espacial: $O(nm)$.

Una versión con memoria reducida

```
x = "0123456789"  
y = "02468"  
  
def d(a, b): return a != b  
  
def D2(x, y, d):  
    previo, actual = [0] * len(y), [0] * len(y)  
    actual[0] = d(x[0], y[0])  
    for j in range(1, len(y)):  
        actual[j] = actual[j-1] + d(x[0], y[j])  
    for i in range(1, len(x)):  
        actual, previo = previo, actual  
        actual[0] = previo[0] + d(x[j], y[0])  
        for j in range(1, len(y)):  
            actual[j] = min( actual[j-1] + d(x[i], y[j]),  
                            previo[j] + d(x[i], y[j]),  
                            previo[j-1] + d(x[i], y[j]) )  
    return actual[len(y)-1]
```

```
print 'Distancia entre x e y: ', D2(x,y,d)
```

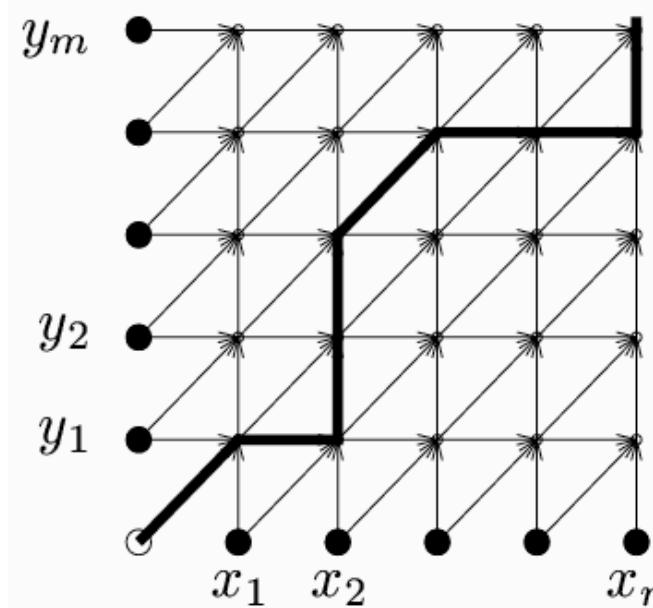
Complejidad temporal: $O(nm)$.

Complejidad espacial: $O(m)$.

Una recursión equivalente

$D(X, Y) = D(n, m)$ donde

$$D(i, j) = \begin{cases} 0, & \text{si } i = 0 \wedge j = 0; \\ \infty, & \text{si } (i = 0 \vee j = 0) \wedge i \neq j; \\ \min \left\{ \begin{array}{l} D(i, j - 1) + d(x_i, y_j), \\ D(i - 1, j) + d(x_i, y_j), \\ D(i - 1, j - 1) + d(x_i, y_j) \end{array} \right\} & \text{en otro caso.} \end{cases}$$



- inicialización a $+\infty$
- inicialización a 0

Grafo asociado a la nueva recursión.

```
x = "0123456789"  
y = "02468"  
  
def d(a, b): return a != b  
  
def D3(x, y, d):  
    infinito = 2**30  
    F = [ [0] * (len(y)+1) for i in range(len(x)+1) ]  
    F[0][0] = 0  
    for i in range(1, len(x)+1):  
        F[i][0] = infinito  
    for j in range(1, len(y)+1):  
        F[0][j] = infinito  
    for i in range(1, len(x)+1):  
        for j in range(1, len(y)+1):  
            F[i][j] = min( F[i][j-1] + d(x[i-1], y[j-1]),  
                            F[i-1][j] + d(x[i-1], y[j-1]),  
                            F[i-1][j-1] + d(x[i-1], y[j-1]) )
```

```
return F[len(x)][len(y)]
```

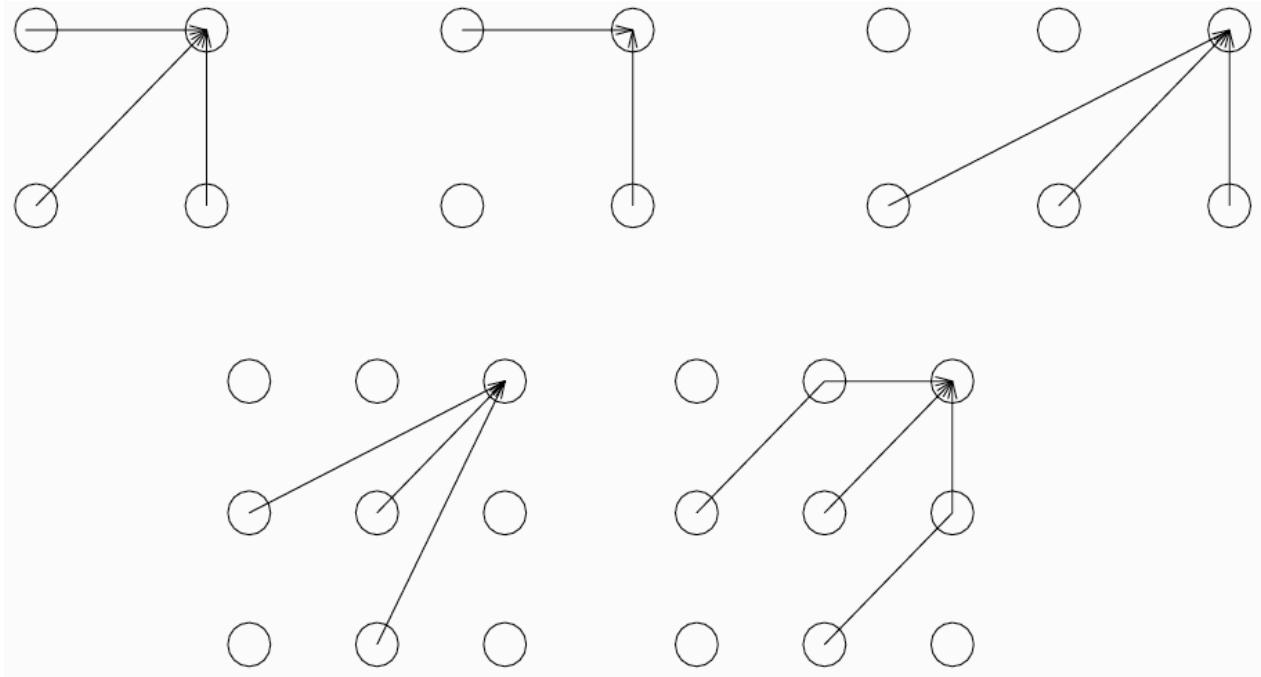
```
print 'Distancia entre x e y: ', D3(x,y,d)
```

Complejidad temporal: $O(nm)$.

Complejidad espacial: $O(nm)$, reducible a $O(m)$.

Juegos de producciones

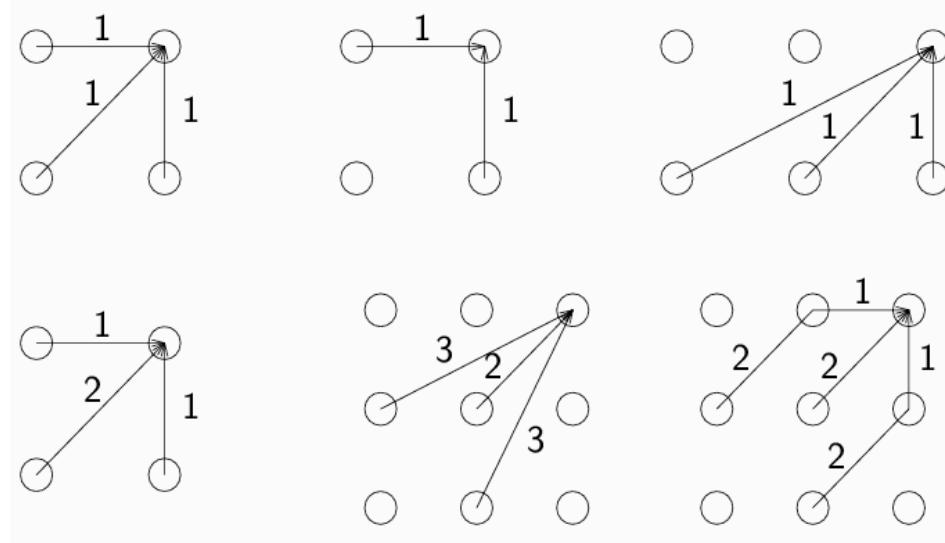
Otra manera de imponer restricciones al alineamiento es cambiar el **conjunto de producciones**:



Diferentes conjuntos de producciones.

Las producciones permiten imponer restricciones de continuidad, monotonía, simetría/asimetría, etc.

Es posible ponderar las producciones:



Diferentes conjuntos de producciones ponderadas.

$\mathcal{D}(X, Y) = D(n, m)$ donde

$$D(i, j) = \begin{cases} 0, & \text{si } i = 0 \wedge j = 0; \\ \infty, & \text{si } i = 1 \vee j = 1; \\ \min_{(a,b) \in P} (D(i - a, j - b) + w(a, b) \cdot d(x_i, y_j)), & \text{en otro caso.} \end{cases}$$

Normalización

Las distancias por ATNL que estamos considerando tienen un serio inconveniente: una propensión a dar valores altos para pronunciaciones largas, y bajos para pronunciaciones cortas.

Ejemplo: ¿Qué se parece más, “a” a “b” o “cosas” a “casos”.

Según el método de distancia por ATNL considerado tenemos:

$$D(a, b) = 1 \quad D(\text{cosas}, \text{casos}) = 2$$

Es necesario efectuar alguna forma de **normalización por la duración** de los objetos comparados.

Una posible definición de la distancia por ATNL normalizada sería:

$$\mathcal{D}(X, Y) = \min_{(i_1, j_1), \dots, (i_p, j_p) \in \mathcal{A}(X, Y)} \frac{\sum_{k=1}^p w(i_k - i_{k-1}, j_k - j_{k-1}) \cdot d(x_{i_k}, y_{j_k})}{\sum_{k=1}^p w(i_k - i_{k-1}, j_k - j_{k-1})}$$

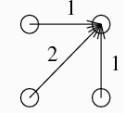
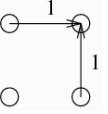
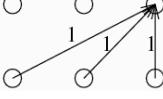
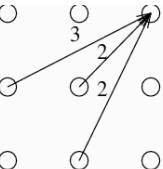
donde suponemos $(i_0, j_0) = (0, 0)$.

Con esta definición:

$$D(a, b) = \frac{1}{2} \quad D(\text{cosas}, \text{casos}) = \frac{2}{10}.$$

Para ser resoluble por programación dinámica *con el mismo coste computacional*, esta distancia debe presentar un denominador que no dependa del alineamiento.

Ciertos conjuntos de producciones permiten trabajar con denominadores constantes:

Producciones ponderadas	Denominador
	$n + m$
	$n + m$
	n
	$n + m$

Este algoritmo normaliza correctamente para producciones 1-2-1:

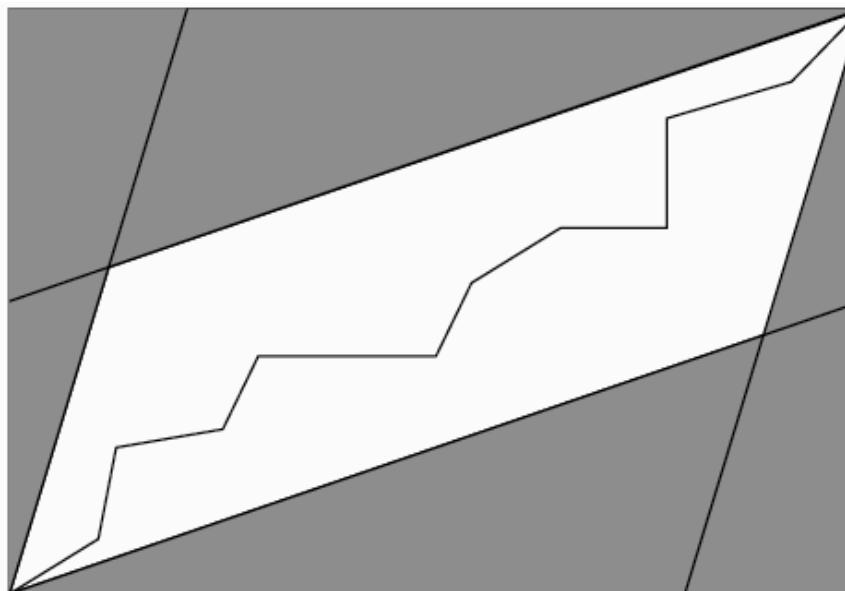
```
x = "0123456789"  
y = "02468"  
  
def d(a, b): return a != b  
  
def D4(x, y, d):  
    F = [ [0] * len(y) for i in range(len(x)) ]  
    F[0][0] = 2 * d(x[0], y[0])  
    for i in range(1, len(x)):  
        F[i][0] = F[i-1][0] + d(x[i], y[0])  
        for j in range(1, len(y)):  
            F[0][j] = F[0][j-1] + d(x[0], y[j])  
        for i in range(1, len(x)):  
            for j in range(1, len(y)):  
                F[i][j] = min( F[i][j-1] + d(x[i], y[j]),  
                               F[i-1][j] + d(x[i], y[j]),  
                               F[i-1][j-1] + 2 * d(x[i], y[j]) )  
    return F[len(x)-1][len(y)-1]/float(len(x)+len(y))
```

```
print 'Distancia entre x e y: ', D4(x,y,d)
```

Aceleración con sacrificio de optimidad

Imposición de restricciones de pendiente Los alineamientos que estamos considerando pueden describir cualquier camino válido en el grafo.

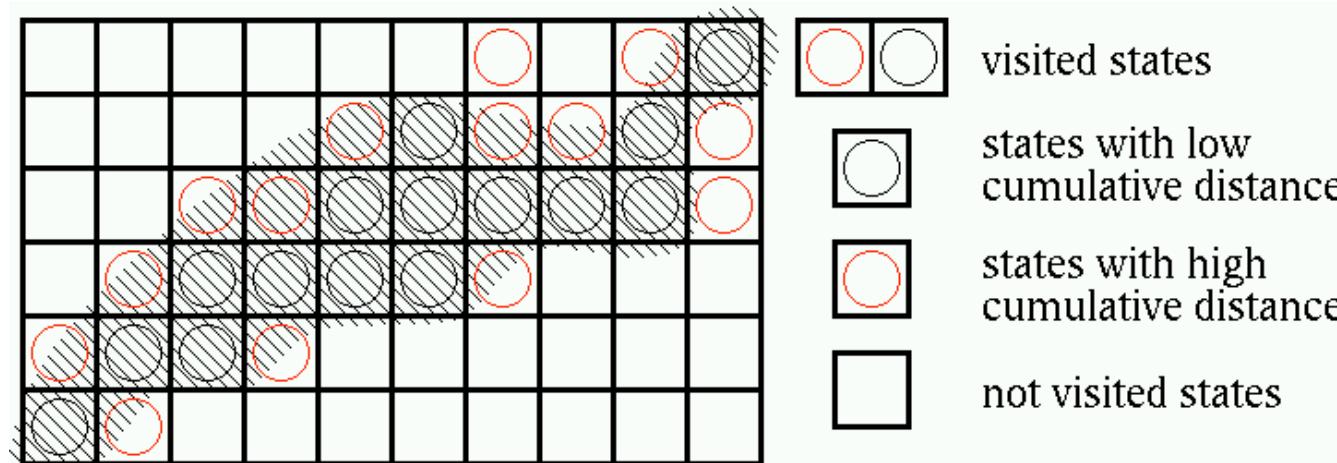
A veces interesa no considerar alineamientos demasiado deformes. Por ejemplo, recorriendo sólo la zona en blanco de la retícula, se puede acelerar el cálculo:



Aceleración del cálculo visitando únicamente la zona blanca. Se sacrifica la corrección del resultado..

Búsqueda en haz (beam search) Una idea alternativa consiste en no visitar todos los nodos del grafo, sino sólo aquellos que parecen prometedores.

1. Se empieza en el nodo inicial que ingresa en un “conjunto de estados prometedores”,
2. se visitan todos los estados sucesores de todos los estados prometedores y éstos últimos se extraen del conjunto,
3. aquellos sucesores cuya puntuación satisface cierto criterio, ingresan en el conjunto de estados prometedores,
4. repetir desde el paso 2 hasta que se alcanza el estado final.

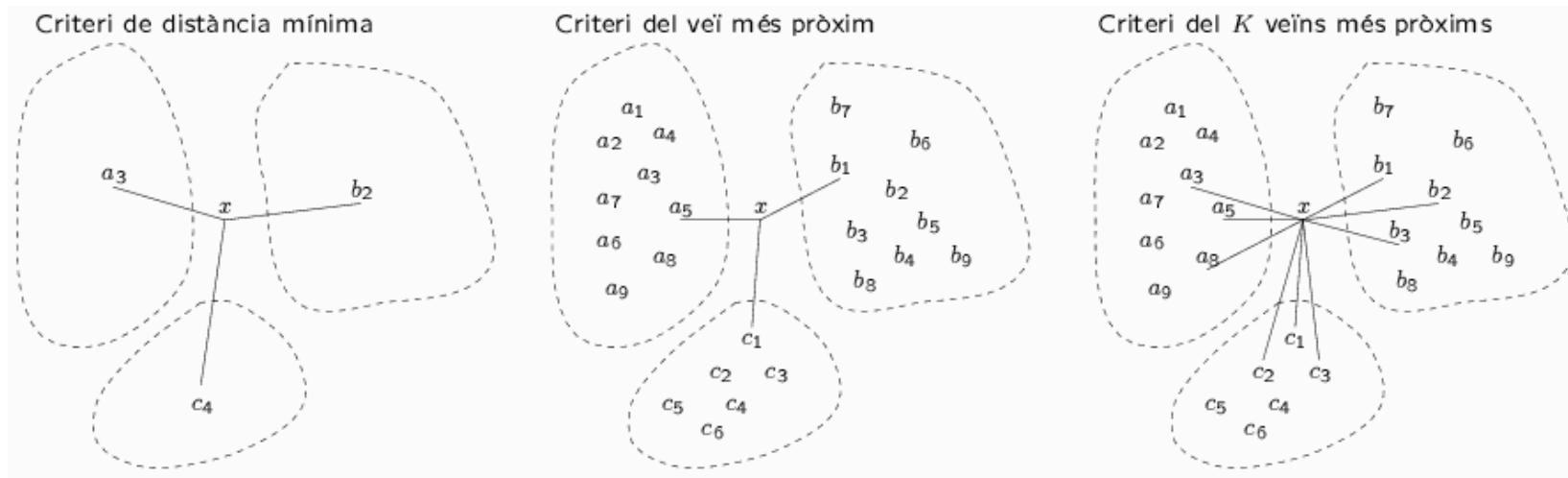


Búsqueda en haz.

Clasificación

Ahora disponemos de un sistema que cuantifica la similitud/disimilitud entre pronunciaciones.

Podemos construir un reconocedor a partir de un diccionario que contenga una o más pronunciasiones de cada palabra a reconocer (plantillas).



Clasificadores por vecindad.

Problema: para clasificar (aproximación de fuerza bruta) es necesario calcular la distancia de la pronunciación a **todos** los prototipos, y *cada comparación* es $O(nm)$.

El clasificador funciona tanto mejor cuantos más prototipos tiene almacenados por cada palabra del vocabulario.

Conclusión: El coste de la aproximación por fuerza bruta es prohibitivo para medios y grandes diccionarios.

- Se puede *reducir el número de prototipos* para cada palabra tratando de quedarnos con los más “interesantes” (los que contribuyen a definir las fronteras entre clases): *multiedit-condensing*.
- Es posible encontrar rápidamente el(los) vecino(s) más próximo(s) para una pronunciación con técnicas como los *kd*-árboles o algoritmos como AESA.

Reconocimiento de palabras conectadas

Notación y definiciones

Disponemos de un diccionario con N palabras

$$W = \{\Omega^1, \Omega^2, \dots, \Omega^N\}$$

donde cada palabra es una cadena de Σ^+ de talla n_i

$$\Omega^i = \Omega_1^i \Omega_2^i \dots \Omega_{n_i}^i \in \Sigma^+$$

Nos dan la pronunciación de una frase, que es una concatenación de $|a|$ eventos u observaciones acústicas:

$$a = a_1 a_2 \dots a_{|a|} \in \Sigma^+$$

Sabemos que a se compone de una sucesión de palabras del diccionario. ¿Cuál es esa sucesión?

$$\overbrace{a_1 \dots a_i}^{w_1} \overbrace{a_{i+1} \dots a_j}^{w_2} \dots \overbrace{a_k \dots a_{|a|}}^{w_n}$$

El problema de la descodificación óptima

Deseamos encontrar la descodificación óptima en cierto sentido (la que hace mínima la suma de distorsiones entre segmentos de a y palabras del diccionario) y su segmentación asociada.

$$\min_{s_0 s_1 \dots s_n} \min_{w_1 w_2 \dots w_n} \sum_{i=1}^n D(a_{s_{i-1}+1} \dots a_{s_i}, w_i).$$

Una primera idea:

- Detectar el inicio y el final de cada palabra (encontrar la mejor segmentación de a)...
- ...y clasificar cada segmento mediante técnicas propias del reconocimiento de palabras aisladas.

Pero si no hay pausas entre palabras...



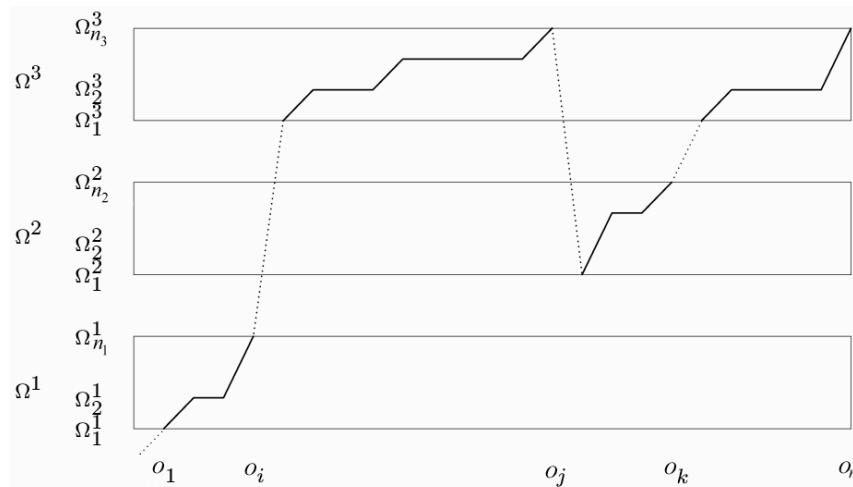
Las pausas no necesariamente tienen lugar entre palabras.

...¿cómo encontramos la segmentación óptima, la que parte la secuencia acústica a en la secuencia de palabras más similares a lo pronunciado?, ¿y cómo encontramos la secuencia de palabras que proporciona una segmentación óptima?

El algoritmo de Un Paso

El algoritmo En 1979, Sakoe propuso el denominado algoritmo de **Dos Pasos** para hallar simultáneamente la segmentación y secuencia de palabras óptimas.

En 1984 se reinventó una técnica de 1971 (Vintsyuk): el algoritmo de **Un Paso**, más rápido que el de Dos Pasos, aunque de aplicación algo más restringida.



Representación esquemática del grafo subyacente al algoritmo de *Un Paso* y solución como camino en él. En el eje horizontal, la pronunciación. En el eje vertical, la pronunciación de cada una de las palabras del diccionario. El camino recorre una serie de palabras ($\Omega^1\Omega^3\Omega^2\Omega^3$) y determina una segmentación de a en los saltos entre palabras ($0, i, j, k, |a|$).

En la figura se puede ver una segmentación ($0, i, j, k, |a|$) y una descodificación asociada

$\Omega^1 \Omega^3 \Omega^2 \Omega^3$.

Dado un diccionario $W = \{\Omega^1, \Omega^2, \dots, \Omega^N\}$ y la pronunciación de una frase $a = a_1 a_2 \dots a_{|a|}$ queremos calcular

$$\min_{w_1 \dots w_n \in W^+} D(a_1 a_2 \dots a_{|a|}, w_1 \dots w_n)$$

Esta expresión es equivalente a

$$\min_{1 \leq j \leq N} \min_{w_1 \dots w_n \in W^*} D(a_1 a_2 \dots a_{|a|}, w_1 \dots w_n \Omega^j)$$

ya que n es una variable libre.

Definimos

$$\mathcal{D}(i, j, k) = \min_{w_1 \dots w_n \in W^*} D(a_1 a_2 \dots a_i, w_1 \dots w_n \circ (\Omega_1^j \dots \Omega_k^j))$$

como la **distorsión óptima** resultante de alinear el prefijo $a_1 \dots a_i$ con una frase (descodificación) que finaliza con la secuencia de tramas $\Omega_1^j \dots \Omega_k^j$ (que es el principio de la palabra, Ω^j).

Nuestro objetivo puede plantearse como calcular

$$\min_{1 \leq j \leq N} \mathcal{D}(|a|, j, n_j)$$

ya que queremos calcular la distorsión resultante de alinear óptimamente una pronunciación entera con una descodificación que finaliza con una palabra entera (que aún no conocemos).

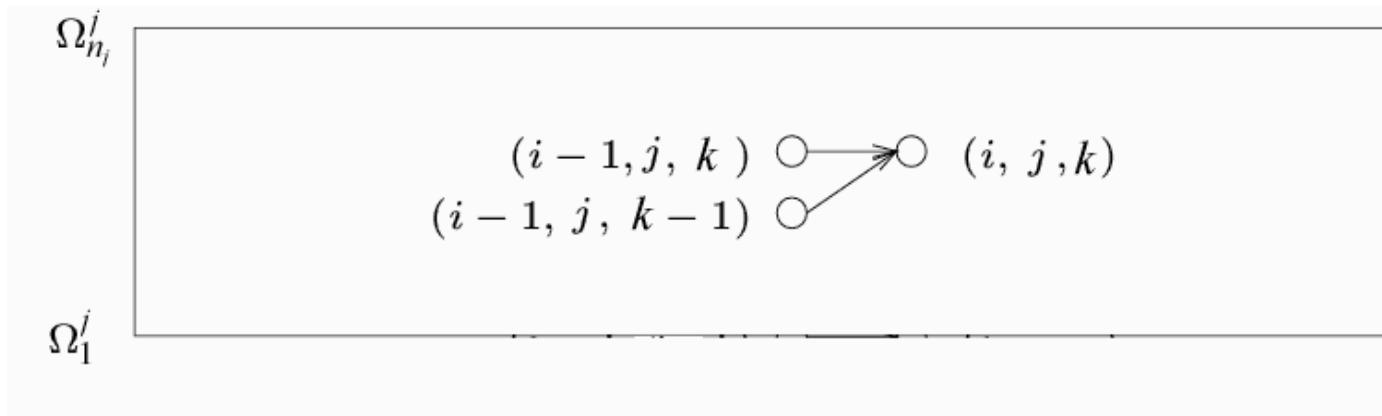
La secuencia de palabras y segmentación asociada se pueden obtener por *backtracing*.

Supongamos que estamos calculando la distancia entre dos pronunciaciones por ATNL mediante el conjunto de producciones (1, 0) y (1, 1) con peso 1.

Tenemos

$$D(a_1 a_2 \dots a_i, w_1 \dots w_{n-1} \circ (\Omega_1^j \dots \Omega_k^j))$$

$$\begin{aligned} &= \min \left\{ \begin{array}{l} D(a_1 a_2 \dots a_{i-1}, w_1 \dots w_{n-1} \circ (\Omega_1^j \dots \Omega_k^j)) + d(a_i, \Omega_k^j), \\ D(a_1 a_2 \dots a_{i-1}, w_1 \dots w_{n-1} \circ (\Omega_1^j \dots \Omega_{k-1}^j)) + d(a_i, \Omega_k^j) \end{array} \right\} \\ &= \min \left\{ \begin{array}{l} \mathcal{D}(i-1, j, k) + \delta(a_i, \Omega_k^j), \\ \mathcal{D}(i-1, j, k-1) + \delta(a_i, \Omega_k^j) \end{array} \right\} \end{aligned}$$

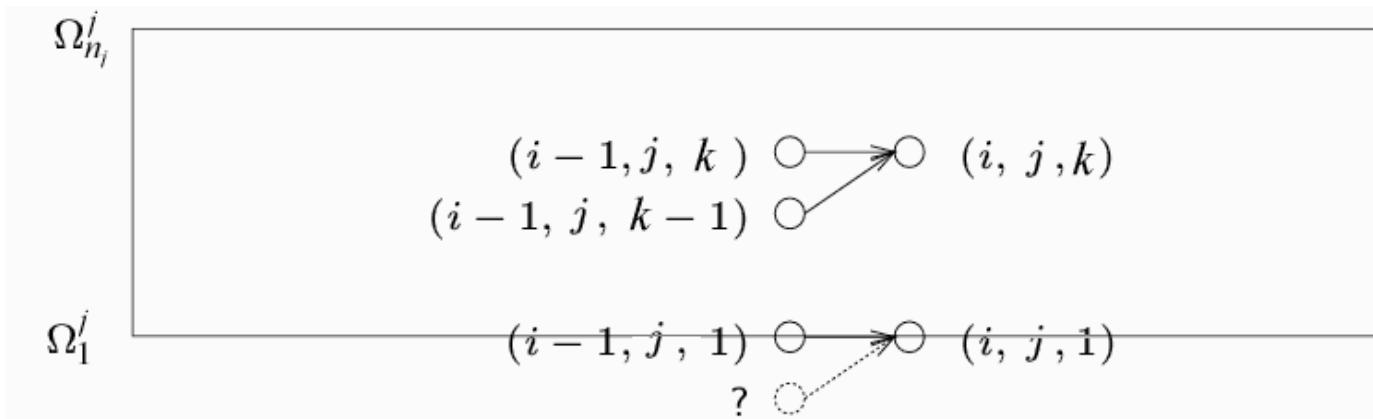


Recursión del algoritmo de Un Paso al alinear una palabra.

Podemos completar la expresión con el tratamiento de ciertas situaciones extremas:

$$\mathcal{D}(i, j, k) = \begin{cases} 0, & \text{si } i = 0 \text{ y } k = 1. \\ +\infty, & \text{si } i = 0 \text{ y } k > 1; \\ \min \left\{ \begin{array}{l} \mathcal{D}(i - 1, j, k) + \delta(a_i, \Omega_k^j), \\ \mathcal{D}(i - 1, j, k - 1) + \delta(a_i, \Omega_k^j), \end{array} \right\}, & \text{en otro caso.} \end{cases}$$

¡Pero aún falta considerar un caso extremo! ¿Qué pasa si $k - 1 \leq 0$?



¿Qué recursión efectuar en la frontera?

Si con una trama a_i nos encontramos al principio de Ω^j hay dos posibilidades para la trama a_i :

1. Que también se encuentre al *principio* de Ω^j .
2. Que corresponda al *final* de cualquier palabra.

En este segundo caso, ¿de qué palabra? De la que haga óptima la distorsión, es decir,

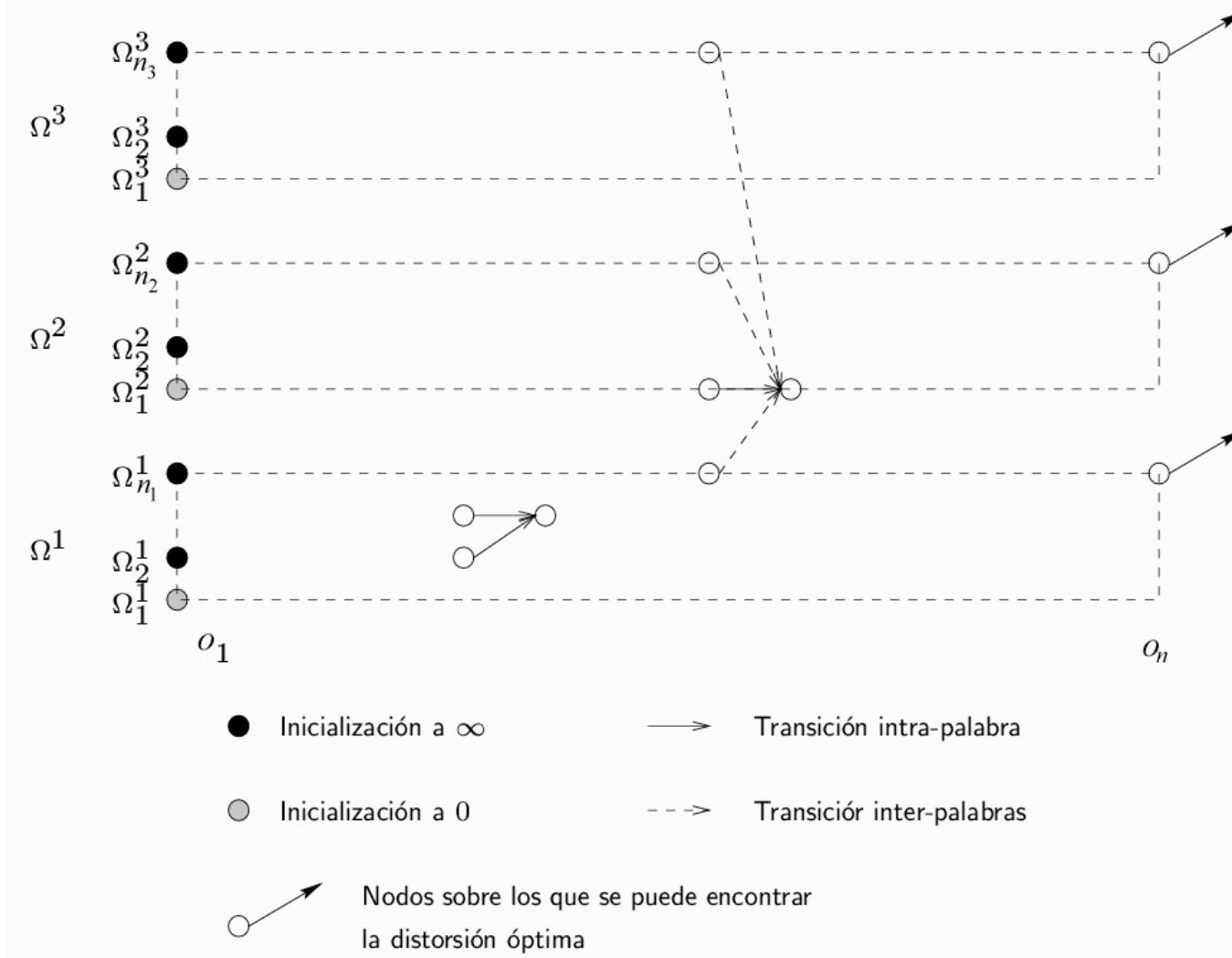
$$\min_{1 \leq l \leq N} D(a_1 \dots a_{i-1}, w_1 \dots w_{n-2} \circ (\Omega_1^l \dots \Omega_{n_l}^l)) = \min_{1 \leq l \leq N} \mathcal{D}(i-1, l, n_l)$$

Conclusión:

$$\mathcal{D}(i, j, k) = \begin{cases} 0, & \text{si } i = 0 \text{ y } k = 1. \\ +\infty, & \text{si } i = 0 \text{ y } k > 1; \\ \min \left\{ \begin{array}{l} \mathcal{D}(i - 1, j, k) + \delta(a_i, \Omega_k^j), \\ \mathcal{D}(i - 1, j, k - 1) + \delta(a_i, \Omega_k^j), \end{array} \right\}, & \text{si } i > 1 \text{ y } k > 1. \\ \min \left\{ \begin{array}{l} \mathcal{D}(i - 1, j, 1) + \delta(a_i, \Omega_1^j), \\ \min_{1 \leq l \leq N} D(i - 1, l, n_l) + \delta(a_i, \Omega_1^j) \end{array} \right\}, & \text{si } i > 0 \text{ y } k = 1. \end{cases}$$

y queremos calcular

$$\min_{1 \leq j \leq N} \mathcal{D}(|a|, j, n_j).$$



Producciones intrapalabra e interpalabras.

```

def unpaso (a, W, d):
    infinito = 2**20
    F = []
    for i in range(len(a)+1):
        F.append({})
        for j in W.keys():
            F[-1][j] = [0] * len(W[j])

    for j in W.keys():
        F[0][j][0] = 0
        for k in range(1,len(W[j])): F[0][j][k] = infinito

    for i in range(1, len(a)+1):
        for j in W.keys():
            F[i][j][0] = min(F[i-1][j][0] + d(a[i-1], W[j][0]),
                            min([F[i-1][l][len(W[l])-1] + d(a[i-1], W[j][0]) for l in
                                 W.keys()])
            for k in range(1, len(W[j])):
                F[i][j][k] = min(F[i-1][j][k] + d(a[i-1], W[j][k]),
                                F[i-1][j][k-1] + d(a[i-1], W[j][k]))

```

```
return min([F[len(a)][j][len(w[j])-1] for j in w.keys()])
```

Complejidad temporal: $O(|a| \cdot \sum_{i=1}^N n_i + |a| \cdot N^2)$.

Complejidad espacial: $O(|a| \cdot \sum_{i=1}^N n_i)$.

La complejidad espacial puede reducirse si almacenamos únicamente las columnas actual y previa:

```
def unpaso2(a, W, d):
    infinito = 2**20
    actual = {}
    previa = {}
    for j in W.keys():
        previa[j] = [0] * len(W[j])
        actual[j] = [0] * len(W[j])
        actual[j][0] = 0
        for k in range(1,len(W[j])): actual[j][k] = infinito

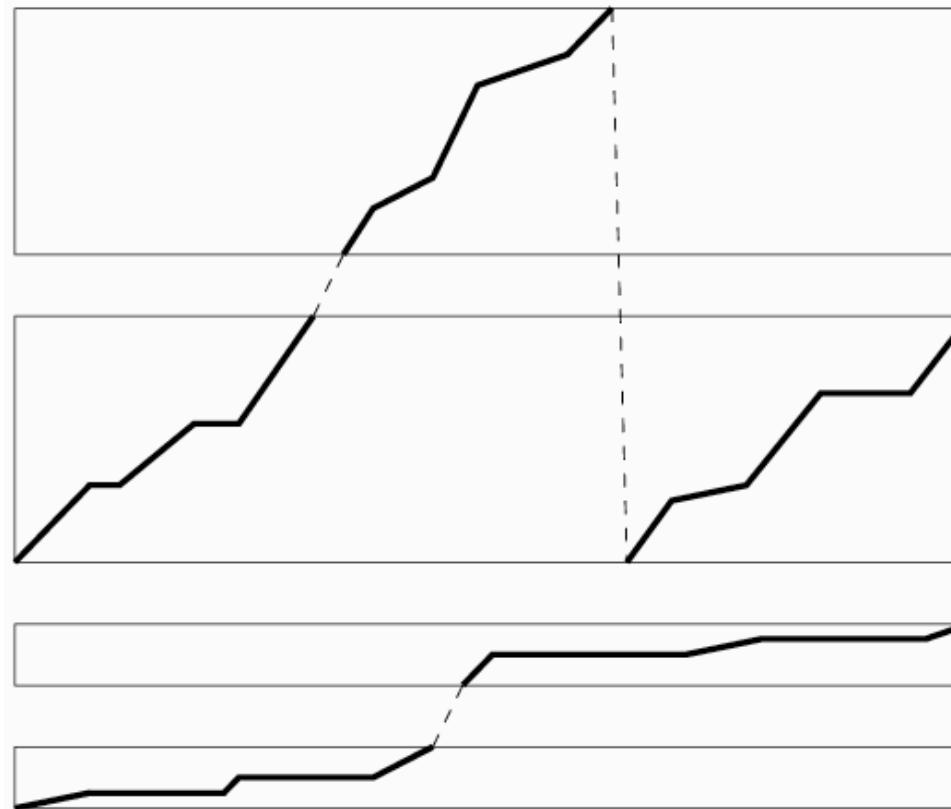
    for i in range(1, len(a)+1):
        actual, previa = previa, actual
        for j in W.keys():
            actual[j][0] = min(previa[j][0] + d(a[i-1], W[j][0]),
                                min([previa[l][len(W[l])-1] + d(a[i-1], W[j][0]) for l in W
                                     if l < i]))
            for k in range(1, len(W[j])):
                actual[j][k] = min(previa[j][k] + d(a[i-1], W[j][k]),
```

```
    previa[j][k-1] + d(a[i-1], w[j][k]))  
return min([actual[j][len(w[j])-1] for j in w.keys()])
```

Complejidad temporal: $O(|a| \cdot \sum_{i=1}^N n_i + |a| \cdot N^2)$.

Complejidad espacial: $O(\sum_{i=1}^N n_i)$.

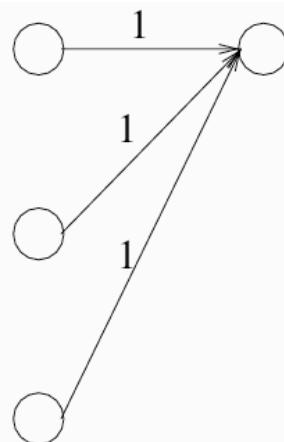
Normalización El algoritmo tiene tendencia a seleccionar caminos con el menor número posible de producciones: hay un problema de normalización.



Dos caminos en el grafo del algoritmo de Un Paso. El camino superior consta de 3 palabras de tamaño medio. El camino inferior, de 2 muy cortas. Es probable que el segundo camino esté menos penalizados, aunque su alineamiento sea, intuitivamente, peor.

Se puede penalizar los saltos entre palabras y/o normalizar adecuadamente por la longitud del camino.

Las producciones 1-2-1 no permiten efectuar una correcta normalización (la longitud depende del número de palabras y de su longitud). Sí lo permiten otras producciones *asimétricas*:



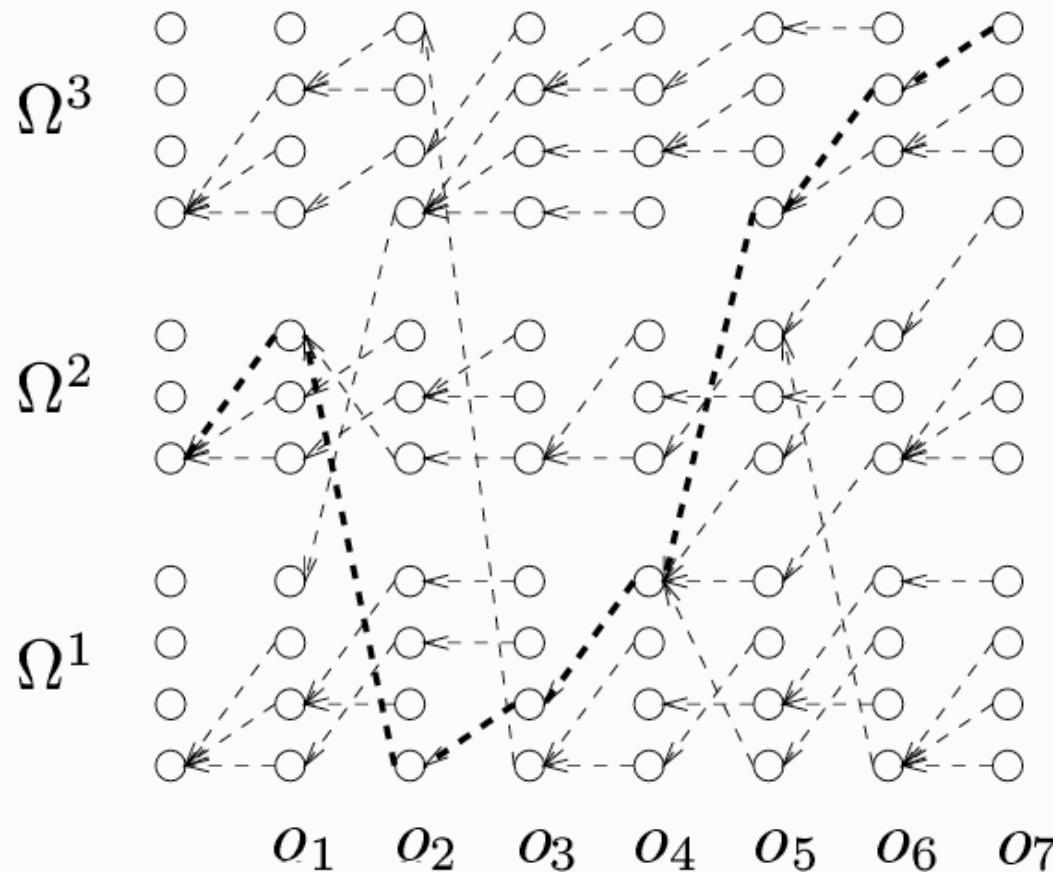
Producciones asimétricas. Todo camino usa |o| producciones (y todas tienen peso 1).

Para evitar esta tendencia, el algoritmo de Un Paso suele usar producciones asimétricas.

Recuperación del camino Con este algoritmo sabemos calcular la distorsión óptima, pero nos interesa recuperar la secuencia de modelos que hace óptima la distorsión.

Podemos recuperar la secuencia mediante la técnica de punteros hacia atrás (*backtracing*), guardando en cada nodo del grafo un puntero al nodo del que viene el camino óptimo.

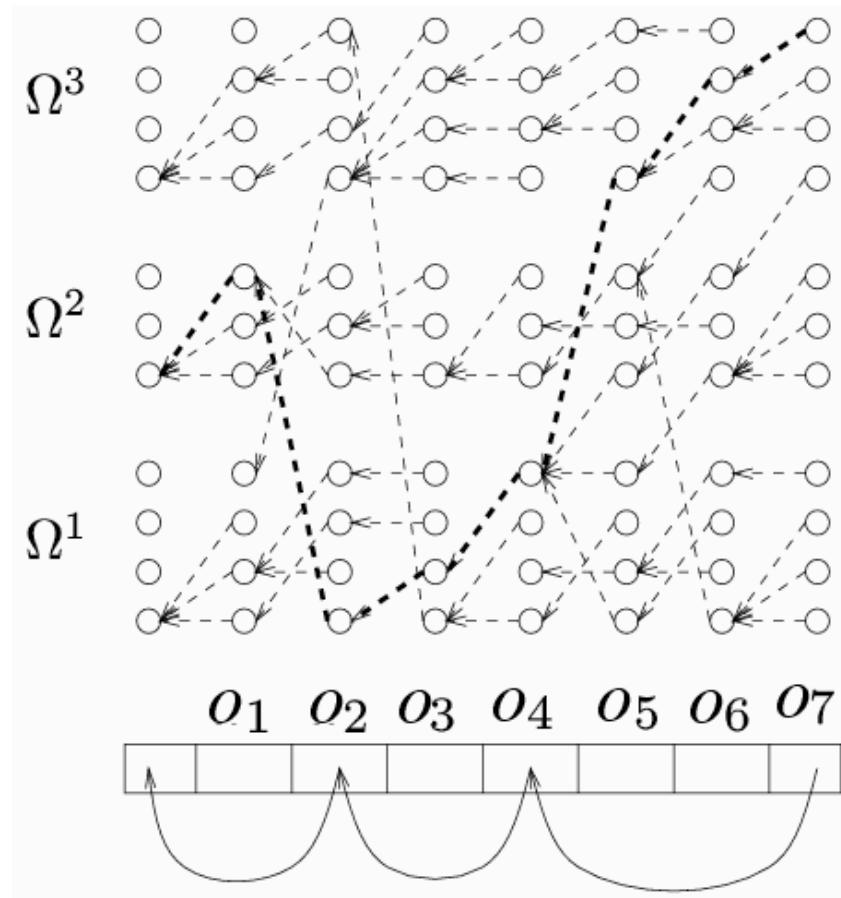
Pero en este caso, la complejidad espacial es otra vez $O(|a| \cdot \sum_{i=1}^N n_i)$ que, en la práctica, es demasiado grande.



Punteros hacia atrás.

No nos interesa todo el alineamiento: sólo la secuencia de *palabras* que visita el camino óptimo.

¡Puede almacenarse únicamente un puntero por columna! El puntero de la columna i corresponde a un final de palabra y señala el final de la anterior palabra.



Un puntero por columna.

¿Cómo calcularlo?

Sea

$B : \text{vector } [0..|a|] \text{ de } \mathbb{N}$

el vector de punteros hacia atrás y sea

$P : \text{vector } [0..|a|] \text{ de } W$

un vector donde recordamos la mejor palabra que finaliza en cada trama, y sea

$b_{\text{actual}}, b_{\text{previa}} : \text{vector } [W, \max_{1 \leq i \leq N} n_i] \text{ de } \mathbb{N}$

un vector sendos vectores auxiliares de punteros para dos columnas del grafo.

En realidad, sólo necesitamos dos columnas de b : la actual y la anterior.

```
def un paso3(a, W, d):  
    infinito = 2**20  
    actual = {}  
    previa = {}  
    b_actual= {}  
    b_previa= {}
```

```

B = [None] * (len(a)+1)
P = [None] * (len(a)+1)

for j in W.keys():
    previa[j] = [0] * len(W[j])
    actual[j] = [0] * len(W[j])
    b_previa[j] = [0] * len(W[j])
    b_actual[j] = [0] * len(W[j])
    actual[j][0] = 0
    for k in range(1, len(W[j])):
        actual[j][k] = infinito
        b_actual[j][k] = None

for i in range(1, len(a)+1):
    actual, previa = previa, actual
    b_actual, b_previa = b_previa, b_actual
    for j in W.keys():
        actual[j][0] = min(previa[j][0] + d(a[i-1], W[j][0]),
                            min([previa[l][len(W[l])-1] + d(a[i-1], W[j][0]) for l in
if actual[j][0] == previa[j][0] + d(a[i-1], W[j][0]):
```

```

        b_actual[j][0] = b_previa[j][0]

else:
    b_actual[j][0] = i-1
    for k in range(1, len(W[j])):
        actual[j][k] = min(previa[j][k] + d(a[i-1], W[j][k]),
                            previa[j][k-1] + d(a[i-1], W[j][k]))
    if actual[j][k] == previa[j][k] + d(a[i-1], W[j][k]):
        b_actual[j][k] = b_previa[j][k]
    else:
        b_actual[j][k] = b_previa[j][k-1]
aux = infinito
for j in W.keys():
    if actual[j][len(W[j])-1] < aux:
        aux = actual[j][len(W[j])-1]
        B[i] = b_actual[j][len(W[j])-1]
        P[i] = j
return min([actual[j][len(W[j])-1] for j in W.keys()]), B, P

```

Complejidad de todo el proceso:

- Temporal: $O(|a| \cdot \sum_{i=1}^N n_i + |a| \cdot N^2)$.
- Espacial: $O(|a| + \sum_{i=1}^N n_i)$.

Bibliografía

- Lawrence Rabiner, Biing-Hwang Juang: *Fundamentals of speech recognition*. Prentice Hall. 1993.
- Francisco Casacuberta, Enrique Vidal: *Reconocimiento automático del habla*. Marcombo. 1987.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to algorithms* (2nd ed.). The Massachusetts Institute of Technology. 2001.