



Algorísmica

Algorismes per text

Mireia Ribera & Jordi Vitrià

Cerca de *strings*...

Són algorismes crítics en moltes aplicacions importants de la informàtica:

- Editors de text (*search, spell, etc.*).
- Bioinformàtica.
- Cercadors d'Internet.
- Bases de dades.
- Compressió.
- Antivirus.
- Etc.

Cerca de *strings*...

Considerem el següent problema:

El patró

Tenim un *string* de m caràcters (el que volem trobar) i un *string* de n caràcters, $n > m$ dins el qual buscar.

El text

P: 001011

T: 10010101101001100101111010

P: happy

T: It is never too late to have a happy childhood.

P: GATTCAC

T: ATCGGATATCCGGAAACTGGTAGCGTGTAGGAGGTAGCCTGGAAG

Cerca de *strings*: la versió ingènua.

P: 001011

T: 10010101101001100101111010

En una primera instància, podríem
comparar tot el *string* amb cada
possible posició, però fàcilment
podem millorar-ho...

```
10010100010111101001001101
001011.....
.001011.....
..001011.....
...001011.....
....001011.....
.....001011.....
.....001011.....
.....001011.....
.....001011
```

Cerca de *strings*: la versió ingènua.

Algorisme de força bruta:

1. Alineem el patró al principi del text.
2. Ens movem d'esquerra a dreta, comparant cada caràcter del patró amb el caràcter corresponent del text fins que tots els caràcters fan correspondència o trobem una diferència.
3. Mentre hi hagi diferències i no haguem recorregut tot el text, re-alineem una posició més a la dreta i repetim el pas 2.

Cerca de *strings*: Algorisme de força bruta.

```
def BFStringMatching(t,p):  
    m=len(p)  
    n=len(t)  
    for i in range(0,n-m+1):  
        j=0  
        while j<m and p[j]==t[i+j]: j=j+1  
        if j == m: return i  
    return -1
```

10010100010111101001001101

001011

m-1

n-m

n-1

Cerca de *strings*: la versió ingènua.

La complexitat de l'algorisme es pot analitzar en tres situacions:

- En moltes ocasions, fem una comparació i movem. Aquest és el **millor cas**, i la complexitat si per tots els moviments féssim això seria $O(n)$.

Aquest seria el cas de tenir una patró que comença per una lletra que no apareix al text.

- En d'altres, fem totes les comparacions. Aquest és el **pitjor cas**, i la complexitat si per tots els moviments féssim això seria $O(nm)$.

- Quan parlem de *llenguatge natural*, la complexitat **mitja** s'acosta més a $O(n+m)=O(n)$.

S'ha de calcular empíricament

Altres problemes

La cerca no és l'únic problema interessant:

- Buscar el *substring* més gran en comú entre dos texts.
- Cerca aproximada.
- Etc.

Cerca aproximada de *strings*.

El **problema** és: donat un patró $P[1..m]$ i un text $T[1..n]$, trobar el *substring* de T amb la **distància d'edició mínima** respecte a P .

JIM.SAW.ME.IN.A.BARBERSHOP

BERBER → BERBER

BRBAR → B.RBAR

VARVAR → VARVAR

La distància d'edició és el nombre d'operacions primitives per convertir un *string* en un altre.

Un algorisme basat en la **força bruta** calcularia la distància d'edició de P a **tots els substrings** de T , i llavors escolliria el que té distància mínima.

Cerca aproximada de *strings*.

```
1  h
2  ho
3  hol
4  hola
5  o
6  ol
7  ola
8  l
9  la
10 a
```

```
a="hola"
cont=0
for j in range(len(a)):
    for i in range(j+1,len(a)+1):
        cont=cont+1
        print cont,(a[j:i])
```

El nombre de substrings és $\sum_{i=1}^n i = \frac{n(n+1)}{2} = O(n^2)$

Un algorisme basat en la força bruta per fer cerca aproximada de strings tindria una complexitat **$O(n^3m)$** , atès que (com veurem) el càlcul de la distància d'edició té **$O(nm)$** .

Cerca de *strings*: Versions avançades.

Una de les formes que tenim per reduir aquesta complexitat és **pre-processar** l'entrada de l'algorisme per optimitzar el seu funcionament.

Aquesta estratègia s'usa en molts àmbits de l'algorísmica.

Algorisme de Levenshtein

Abans de veure com cercar un patró (curt) en un text (llarg), anem a veure com calcular la “distància” d entre dos *strings* (curts).

$d(\text{"BARBER"}, \text{"BRBAR"}) = ?$

$d(\text{"BARBER"}, \text{"BABRER"}) = ?$

...



Això es fa amb l'algorisme de **Levenshtein**.

В.И. Левенштейн (1965). "Двоичные коды с исправлением выпадений, вставок и замещений символов". Доклады Академий Наук СССР **163** (4): 845–8. Appeared in English as: Levenshtein VI (1966). "Binary codes capable of correcting deletions, insertions, and reversals". *Soviet Physics Doklady* **10**: 707–10.

Algorisme de Levenshtein

Aquest algorisme (també anomenat “**distància d’edició**”) calcula el nombre mínim **d’operacions d’edició** que són necessàries per modificar un *string* P i obtenir-ne un altre T .

Usualment, les operacions d’edició són:

- **inserció** (p.e., canviar *cot* per *coat*),
- **eliminació** (p.e., canviar *coat* per *cot*), i
- **substitució** (p.e., canviar *coat* per *cost*).

També es podria considerar la **transposició**: canviar *cost* per *cots*.

Algorisme de Levenshtein

Per fer-ho, va omplint una matriu d de manera que la posició $[m,n]$ representa la distància d'edició entre el prefix de m caràcters d'un *patró* i el prefix de n caràcters d'un *text*.

patró	L	E	V	E	N	S	H	T	E	I	N
text	M	E	I	L	E	N	S	T	E	I	N

$d[1][1]=1$, $L \rightarrow M$, doncs només és una substitució.

$d[1][3]=3$, $L \rightarrow MEI$, és una substitució i 2 insercions.

Algorisme de Levenshtein

<i>d</i>	.	<i>t1</i>	<i>t2</i>	<i>t3</i>	<i>t4</i>
.	0	1	2	3	4
<i>p1</i>	1				
<i>p2</i>	2			$d[2,3]$	
<i>p3</i>	3				

Aquests valors són evidents

Com calculem
aquests valors ?

Algorisme de Levenshtein

Suposem que ja tenim una alineació òptima entre els **prefixos** $p[0,i-1]$ i $t[0,j-1]$. Què podem fer amb $p[i]$ i $t[j]$ i com calculem $d[i,j]$?

...	p_{i-1}	p_i	p_{i+1}	...
...	t_{j-1}	t_j	t_{j+1}	...

Només podem fer tres coses!

Algorisme de Levenshtein

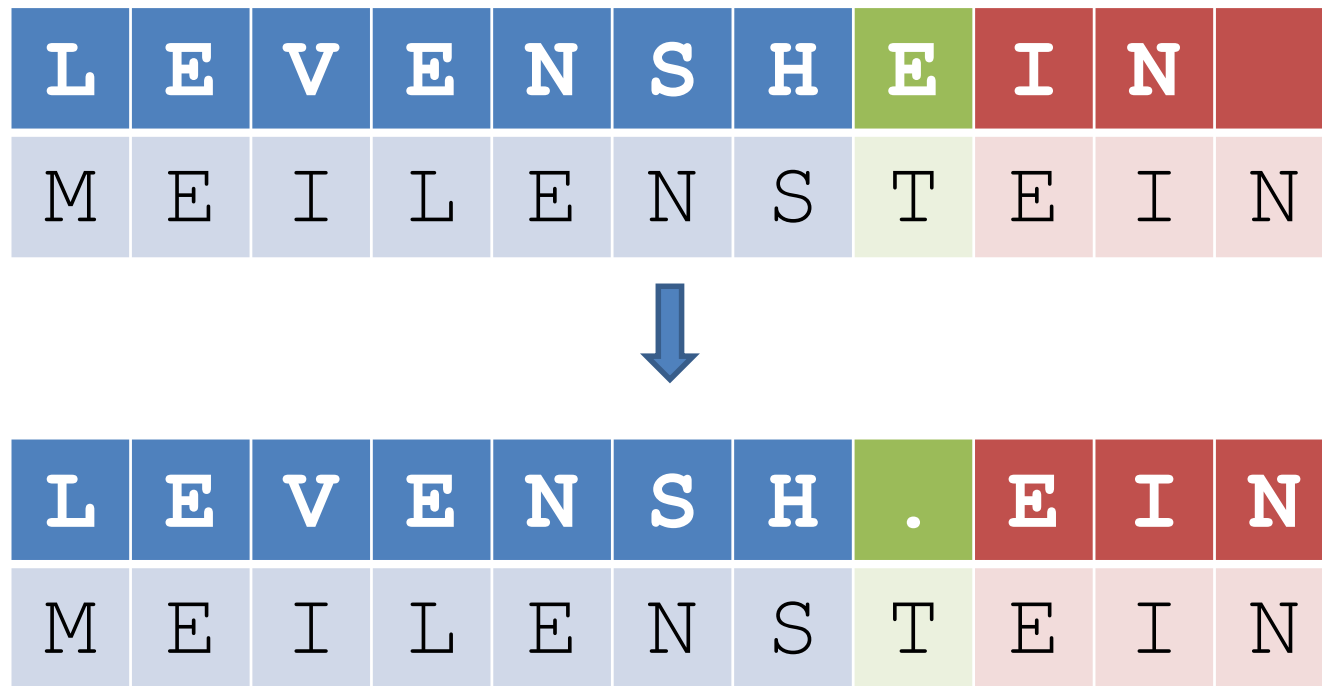
1. Fem que $p[i]$ i $t[j]$ facin correspondència. Si $p[i]=t[j]$ llavors $d[i,j]=d[i-1,j-1]$. Sinó, $d[i,j]=d[i-1,j-1]+1$

L	E	V	E	N	S	H	T	E	I	N
M	E	I	L	E	N	S	T	E	I	N

L	E	V	E	N	S	H	T	E	I	N
M	E	I	L	E	N	S	T	E	I	N

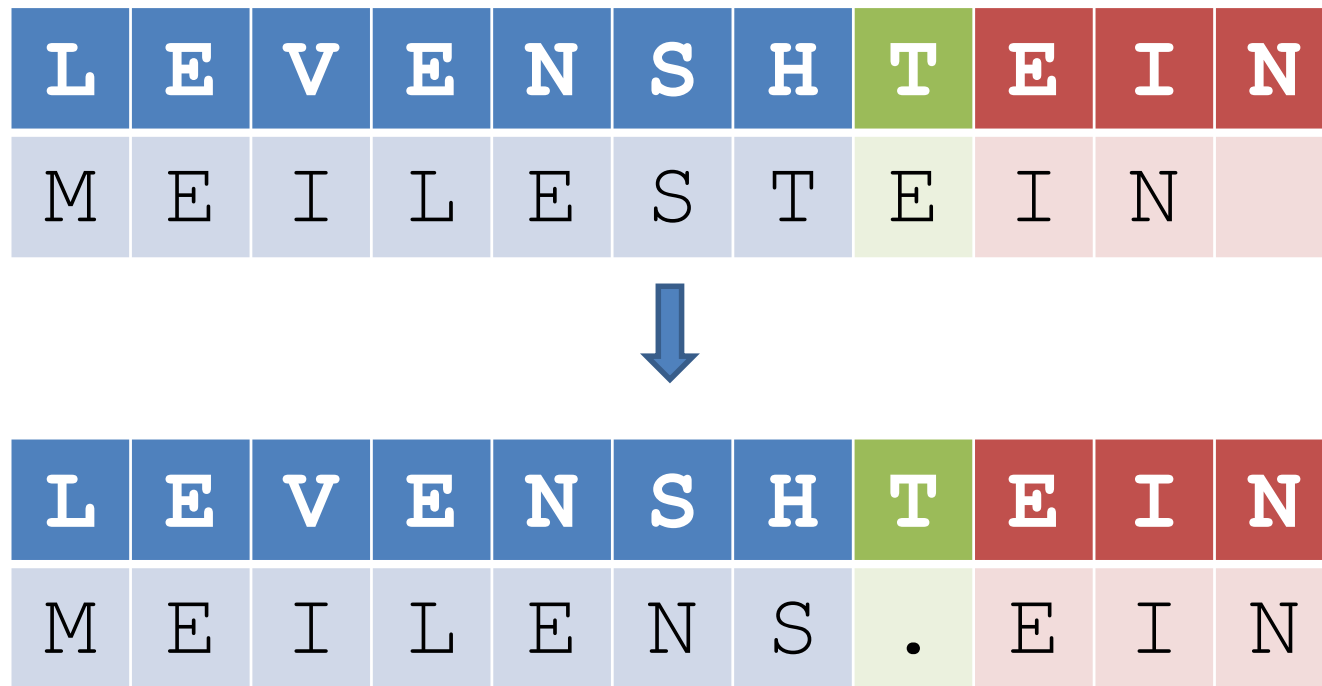
Algorisme de Levenshtein

2. Decidim que hi ha un forat al patró, i per tant $d[i,j]=d[i-1,j]+1$



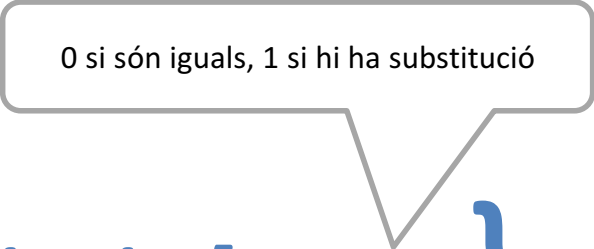
Algorisme de Levenshtein

3. Decidim que hi ha un forat al text, i per tant $d[i,j]=d[i,j-1]+1$



Algorisme de Levenshtein

Observació:

$$d[i,j]=\min\{d[i-1,j] + 1, d[i,j-1] + 1, d[i-1,j-1] + \text{cost}\}$$


0 si són iguals, 1 si hi ha substitució

Això es podria resoldre amb una crida recursiva, atès que nosaltres volem $d[m,n]$ i coneixem $d[0,:]$ i $d[:,0]$, **però la crida recursiva té massa cost computacional!**

Podem seguir la mateixa estratègia que vam fer servir per la seqüència de Fibonacci.

Algorisme de Levenshtein

$\text{mínim } (d[i-1,j] + 1, d[i,j-1] + 1, d[i-1,j-1] + \text{cost})$

		G	U	M	B	O
	0	1	2	3	4	5
G	1					
A	2					
M	3					
B	4					
O	5					
L	6					

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0				
A	2	1				
M	3	2				
B	4	3				
O	5	4				
L	6	5				

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1			
A	2	1	1			
M	3	2	2			
B	4	3	3			
O	5	4	4			
L	6	5	5			

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2		
A	2	1	1	2		
M	3	2	2	1		
B	4	3	3	2		
O	5	4	4	3		
L	6	5	5	4		

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	
A	2	1	1	2	3	
M	3	2	2	1	2	
B	4	3	3	2	1	
O	5	4	4	3	2	
L	6	5	5	4	3	

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	1	2	3	4
M	3	2	2	1	2	3
B	4	3	3	2	1	2
O	5	4	4	3	2	1
L	6	5	5	4	3	2

Algorisme de Levenshtein

La matriu es pot omplir seqüencialment:

Per cada caràcter de s (i des de 1 fins n)

Per cada caràcter de t (j des de 1 fins m)

Si $s[i] == t[j]$, $\text{cost} = 0$

Si $s[i] != t[j]$, $\text{cost} = 1$

$d[i, j] = \text{mínim} (d[i-1, j] + 1,$

$d[i, j-1] + 1,$

$d[i-1, j-1] + \text{cost})$

Inserció al text

Substitució

Eliminació del text

Això té una complexitat $O(mn)$ = calcular tots els elements de la matriu.

Algorisme de Levenshtein

El nombre que queda a la cantonada de baix a la dreta de la matriu és la **distància de Levenshtein, o d'edició, entre les dues paraules.**

Si volem saber les operacions d'edició efectuades, hem de buscar el camí mínim entre els extrems de la matriu o simplement guardar a cada pas la decisió presa respecte a l'edició.

Algorisme de Levenshtein

		m	e	i	l	e	n	s	t	e	i	n
l	0	1	2	3	4	5	6	7	8	9	10	11
e	1	1	2	3	3	4	5	6	7	8	9	10
v	2	2	1	2	3	3	4	5	6	7	8	9
e	3	3	2	2	3	4	4	5	6	7	8	9
n	4	4	3	3	3	3	4	5	6	6	7	8
s	5	5	4	4	4	4	3	4	5	6	7	7
h	6	6	5	5	5	5	4	3	4	5	6	7
t	7	7	6	6	6	6	5	4	4	5	6	7
e	8	8	7	7	7	7	6	5	4	5	6	7
i	9	9	8	8	8	7	7	6	5	4	5	6
n	10	10	9	8	9	8	8	7	6	5	4	5
	11	11	10	9	9	9	8	8	7	6	5	4

Algorisme de Levenshtein

Pot haver-hi diversos possibles passos de cost mínim:

patró	L	E		V	E	N	S	H	T	E	I	N
	S	=	-	S	=	=	=	+	=	=	=	=
text	M	E	I	L	E	N	S		T	E	I	N

patró	L	E	V		E	N	S	H	T	E	I	N
	S	=	S	-	=	=	=	+	=	=	=	=
text	M	E	I	L	E	N	S		T	E	I	N

Algorisme de Levenshtein

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Algorisme de Levenshtein

```
def levenshtein_distance(first, second):
    if len(first) > len(second):
        first, second = second, first
    if len(second) == 0:
        return len(first)
    first_length = len(first) + 1
    second_length = len(second) + 1
    distance_matrix = [[0] * second_length for x in
        range(first_length)]
    for i in range(first_length): distance_matrix[i][0] = i
    for j in range(second_length): distance_matrix[0][j] = j
    for i in xrange(1, first_length):
        for j in range(1, second_length):
            deletion = distance_matrix[i-1][j] + 1
            insertion = distance_matrix[i][j-1] + 1
            substitution = distance_matrix[i-1][j-1]
            if first[i-1] != second[j-1]:
                substitution += 1
            distance_matrix[i][j] = min(insertion,
                deletion, substitution)
    return distance_matrix[first_length-1][second_length-1]
```

Algorisme de Levenshtein (i)

```
def levenshtein_distance(first, second):  
    if len(first) > len(second):  
        first, second = second, first  
  
    if len(second) == 0:  
        return len(first)  
  
    first_length = len(first) + 1  
    second_length = len(second) + 1  
  
    distance_matrix = [[0] * second_length for x  
                        in range(first_length)]  
  
    ...
```

```
>>> a = [[0] * 3 for x in range (3)]  
>>> a  
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Algorisme de Levenshtein (ii)

```
....  
  
for i in range(first_length): distance_matrix[i][0] = i  
for j in range(second_length): distance_matrix[0][j] = j  
  
for i in range(1, first_length):  
    for j in range(1, second_length):  
  
        deletion = distance_matrix[i-1][j] + 1  
        insertion = distance_matrix[i][j-1] + 1  
        substitution = distance_matrix[i-1][j-1]  
  
        if first[i-1] != second[j-1]: substitution += 1  
  
        distance_matrix[i][j] = min(insertion,  
                                     deletion, substitution)  
  
return distance_matrix[first_length-1][second_length-1]
```

Cerca aproximada de *strings*.

Recordem que el nostre problema era:

Donat un patró $P[1..m]$ i un text $T[1..n]$, trobar el *substring* de T amb la **distància d'edició mínima** respecte a P .

JIM.SAW.ME.IN.A.BARBERSHOP
BERBER → BERBER
BRBAR → B.RBAR
VARVAR → VARVAR

Cerca aproximada de *strings*.

Aquest càlcul es pot fer amb l'algorisme de Levenshtein.

Només cal adonar-se que si omplim la primera fila amb zeros (=considerar que podem inserir tants espais en blanc al davant del patró com sigui necessari) podem trobar els substrings de distància mínima!

Si busquem totes les solucions (camins) amb $\text{dist} \leq 1$, només cal mirar l'última fila. v

	-1	0	1	2	3	4	5	6	7	8	9	10	11
		C	A	G	A	T	A	A	G	A	G	A	A
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0 G	1	1	1	0	1	1	1	1	0	1	0	1	1
1 A	2	2	1	1	0	1	1	1	1	0	1	0	1
2 T	3	3	2	2	1	0	1	2	2	1	1	1	1
3 A	4	4	3	3	2	1	0	1	2	2	2	1	1
4 A	5	5	4	4	3	2	1	0	1	2	3	2	1

Aquest valor indica que aquí acaba un string amb distància 0

Aquest valor indica que aquí acaba un string amb distància 1. Si volem trobar el seu principi hem de fer el camí invers a la matriu!

(G A T A A
C A G A T - A A G A G A A)

(G A T A A
C A G A T A - A G A G A A)

(G A T A A
C A G A T A A G A G A A)

(- G A T A A
C A G A T A A G A G A A)

(G A T A A
C A G - A T A A G A G A A)

(G A T A A -
C A G A T A A G A G A A)

(G A T A A
C A G A T A A G A G A A)

Cerca aproximada de *strings*.

El càlcul de la matriu té una complexitat de $O(mn)$, mentre que la cerca del camí marxa enrere té $O(n+m)$.

T: la cassa mes gran que mai ha existit

P: casa

Troblem tres respostes a distància 1:

cas

cass

cassa