Programmazione Dinamica

A.A. 2016-2017

Distanza di Edit

Distanza di edit

Quando uno spell checker incontra una parola “sbagliata” guarda nel suo dizionario per cercare parole “vicine”.
Come possiamo misurare la distanza tra due stringhe?

un modo in cui possono essere allineate o matchate

Ad esempio:

- S N O W - Y	S N O W Y	S - N O W Y
S U N - - N Y	S U N N Y	S U N N - Y

Ad ogni allineamento possiamo associare un costo: numero di operazioni (inserzione, cancellazione, sostituzione di un carattere) necessarie per rendere uguali le due stringhe.

Distanza di edit

Per gli allineamenti precedenti si hanno rispettivamente i costi:

- S N O W - Y	S N O W Y	S - N O W Y
S U N - - N Y	S U N N Y	S U N N - Y
costo = 5	costo = 3	costo = 3

La "distanza di edit" è il costo del miglior allineamento.
Ci sono molti possibili allineamenti: inefficiente provarli tutti
Cerchiamo una soluzione di programmazione dinamica: quali sono i sottoproblemi?

Consideriamo due stringhe $A = a_1, a_2, \ldots, a_n$ e $B = b_1, b_2, \ldots, b_m$
ed esaminiamo due caratteri: a_i e b_j.

Distanza di edit

Nell'allineamento migliore la situazione finale puo` essere una delle tre:

a_n	-	a_n
-	b_m	b_m

Nel primo caso il costo sarà δ + il costo di edit delle stringhe $a_1, a_2, \ldots, a_{n-1}$ e $b_1, b_2, \ldots, b_m$, nel secondo δ + il costo di edit delle stringhe $a_1, a_2, \ldots, a_n$ e $b_1, b_2, \ldots, b_{m-1}$. Nel terzo caso il costo sarà quello di edit delle stringhe $a_1, a_2, \ldots, a_{n-1}$ e $b_1, b_2, \ldots, b_{m-1}$, + $\alpha_{n,m}$ se i due simboli sono diversi, + 0 se a_n e b_m sono uguali

Per ogni valore di i tra 1 e n e ogni valore di j tra 1 e m, trovare un allineamento ottimo per a[1..i] e b[1..j] implica cercare allineamenti ottimi per i tre allineamenti a[1..i-1] e b[1..j], a[1..i] e b[1..j-1] e a[1..i-1] e b[1..j-1].

Denotiamo con E(i, j) la distanza di edit tra $a_1, a_2, \ldots, a_i$ e $b_1, b_2, \ldots, b_j$. Allora il problema E(i, j) è ridotto ai sottoproblemi:

E(i-1, j), E(i, j-1) e E(i-1, j-1).

Distanza di edit

$$E(i, j) = \min \{\delta + E(i-1, j),\ \delta + E(i, j-1),\ \alpha_{i,j} + E(i-1, j-1)\}$$

$$\text{dove } \alpha_{i,j} = \begin{cases} 0 & \text{se } a_i = b_j \\ \neq 0 & \text{se } a_i \neq b_j \end{cases}$$

Esempio: ESPO e POL, assumendo $\delta = 1$ e $\alpha_{i,j} = 1$

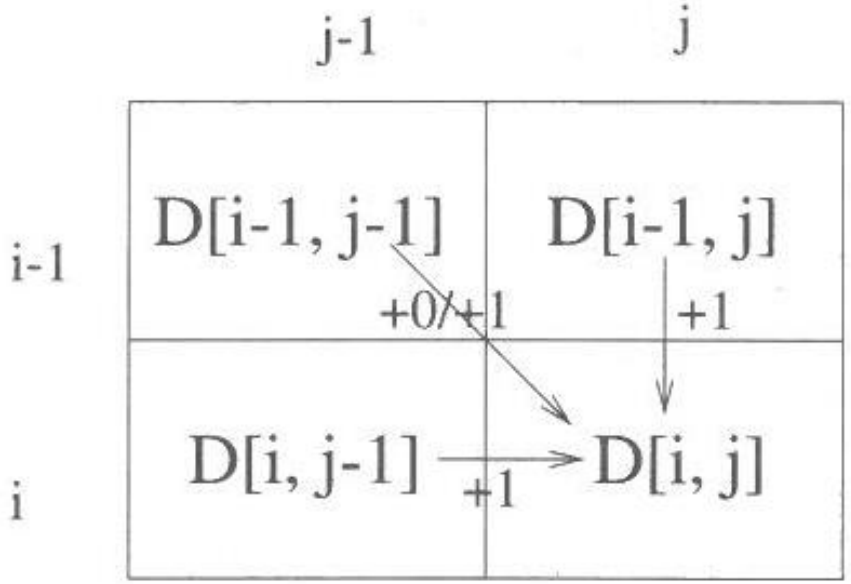
a_4	O	-	O
b_3	-	L	L

$E(4, 3) = \min \{1 + E(3, 3), 1 + E(4, 2), 1 + E(3, 2)\}$

Definiamo il caso base per la stringa vuota (lunghezza 0):

$E(0, 0) = 0 \quad E(0, j) = j \quad E(i, 0) = i$

Casella di testo



Distanza di edit

Sia diff(i,j) una matrice con le distanze $\alpha_{i,j}$ e d la costante δ considerate globali, come l'array E.

```
EDIT-DIST (n,m)
  for i ← 0 to m do
     E(i,0) ← i
  for j ← 0 to n do
     E(0,j) ← j
  for i ← 1 to m do
      for j ← 1 to n do
            E(i,j)  ← min{E(i-1, j-1) + diff(i,j), E(i-1,j)+d, E(i, j-1)+d}

  return E(m,n)
```

Distanza di edit

Esempio: si assuma diff(i,j) = 1 se i $\neq$ j e 0 se i=j e d=1

E		0	1	2	3	4	5	
		-	S	N	O	W	Y	
0	-	0	1	2	3	4	5	
1	S	1	0	1	2	3	4	
2	U	2	1	1	2	3	4	
3	N	3	2	1	2	3	4	
4	N	4	3	2	2	3	4	
5	Y	5	4	3	3	3	3	

Complessità in tempo dell'algoritmo iterativo:
$\Theta(n \cdot m)$

Distanza di edit

Si può ricostruire l'allineamento ottimale percorrendo all'indietra la tabella

		0	1	2	3	4	5
		-	S	N	O	W	Y
0	-	0	1	2	3	4	5
1	S	1	0	1	2	3	4
2	U	2	1	1	2	3	4
3	N	3	2	1	2	3	4
4	N	4	3	2	2	3	4
5	Y	5	4	3	3	3	3

Un allineamento ottimo sarà dato da;
(5,5)
(4,-)
(3,4)
(2,3)
(-,2)
(1,1)
Cioè:
S U N N - Y
S – N O W Y

Un altro esempio:

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

E X P O N E N _ T I A L

_ _ P O L Y N O M I A L

String Matching approssimato

Il problema dello string matching si può presentare in questo modo:

- trovare una occorrenza del pattern (stringa) P di *m* caratteri in un testo T di *n* caratteri con $m \leq n$, ammettendo che ci possano essere errori di questo tipo

1. I corrispondenti caratteri in P e T sono differenti
2. un carattere in P non compare in T
3. un carattere in T non compare in P

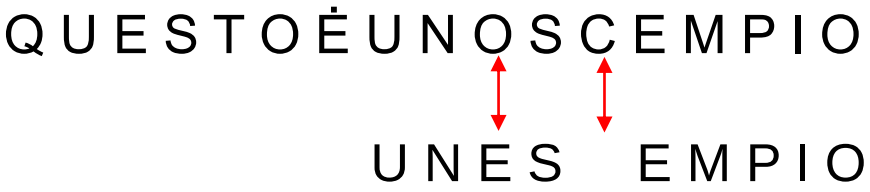
in modo che il matching del pattern sia il *migliore* possibile.

Per esempio se T = QUESTOĖUNOSCEMPIO con n = 18 e P = UNESEMPIO con m = 9, un occorrenza 2-approssimata è quella sottolineata.

String Matching approssimato

Per esempio se T = QUESTOĖUNOSCEMPIO con n = 17 e P = UNESEMPIO con m = 9, un occorrenza 2-approssimata è quella sottolineata.

Infatti:



String Matching approssimato

Si suppone anche ora che T e P inizino con un carattere ''vuoto'' in posizione 0. Sia ora

D[i,j] numero minimo di errori tra $p_0,\ldots,p_i$ e un segmento di T che termina in t_j.

- D[0,j] = 0 per $0 \le j \le n$ (p_0 = occorre in ogni posizione)
- D[i,0] = i per $0 \le i \le m$ ($t_0,..,t_i$ ha i caratteri più di $t_0 = \lambda$

Altrimenti D[i,j] sarà dato dal minimo tra:

- D[i-1, j-1] = 0 se $p_i = t_j$ oppure =1 se $t_i \neq t_j$.
- D[i-1, j] + 1 un carattere in P non compare in T (errore di tipo 2.)
- D[i, j-1] + 1 un carattere in T non compare in P (errore di tipo 3.)

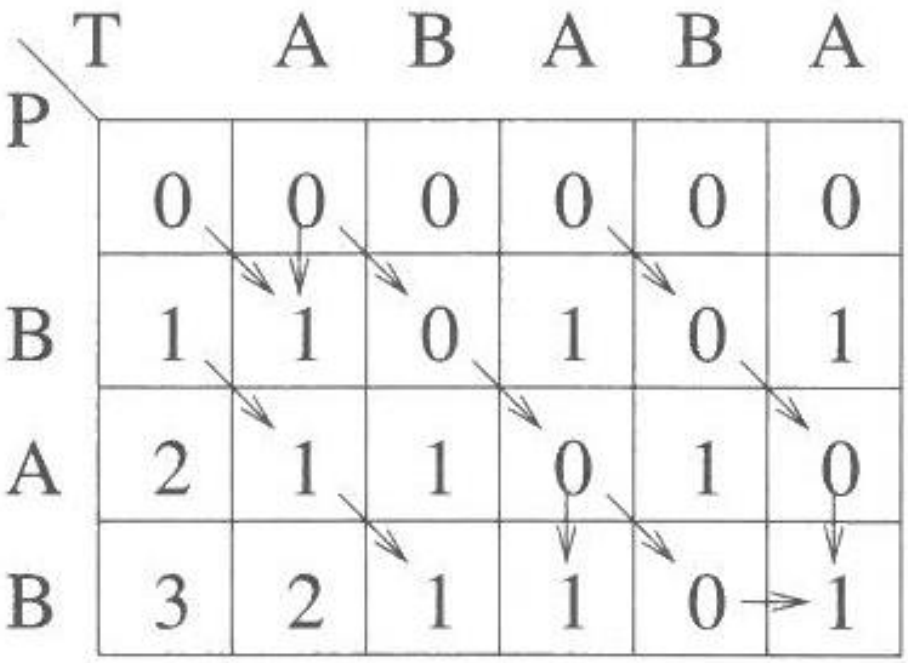
Nota: per semplicità assumiamo che i costi degli errori siamo tutti unitari. Un'impostazione più fine come nel caso precedente è ovviamente possibile.

Lo pseudo-codice:

```
function STRINGMATCHING(var P, T: vettore; m, n: integer): integer;
    var i, j, min: integer; D: matrice;
    begin
        for j := 0 to n do D[0, j] := 0;
        for i := 1 to m do D[i, 0] := i;
        for i := 1 to m do
            for j := 1 to n do begin
                if P[i] = T[j] then min := D[i − 1, j − 1]
                    else min := D[i − 1, j − 1] + 1;
                if D[i − 1, j] + 1 < min then min := D[i − 1, j] + 1;
                if D[i, j − 1] + 1 < min then min := D[i, j − 1] + 1;
                D[i, j] := min
            end;
        min := D[m, 0];
        for j := 1 to n do
            if D[m, j] < min then begin min := D[m, j]; i := j end;
        STRINGMATCHING := i
    end;
```

Complessità in tempo:
$\Theta(n \cdot m)$

Altro esempio: Occorrenza 0-approssimata di BAB in ABABA



P \ T		A	B	A	B	A
	0	0	0	0	0	0
B	1	1	0	1	0	1
A	2	1	1	0	1	0
B	3	2	1	1	0	1