Massima sottosequenza comune

Luca Foschini

Il problema

Supponiamo di volere confrontare 2 stringhe di caratteri per stabilire quanto differiscano l'una dall'altra. Innazitutto dovremmo definire il concetto di differenza tra stringhe. Potremmo dire che due stringhe differiscono tanto più l'una dall'altra quanto più contengono caratteri diversi. Ci accorgiamo subito che questa definizione di differenza è molto approssimativa, in quanto, secondo questo criterio, le stringhe IRTO e OTRI, che contengono gli stessi caratteri, dovrebbero non differe tra loro.

Notiamo subito che è importante considerare non solo i caratteri che compaiono nelle due stringhe, ma anche l'ordine con cui questi compaiono.

La *massima sottosequenza comune* rappresenta invece un valido criterio per stabilire quanto due o più sequenze di caratteri differiscano tra loro, un criterio che tiene conto sia di quali caratteri compaiono nelle due stringhe ma anche dell'ordine con cui questi compaiono.

Definizioni

Sottosequenza. Una stringa w è *sottosequenza* di un'altra stringa x se w è ottenibile da x cancellando zero o più caratteri di x. Più formalmente, la stringa $w_1 w_2 ... w_i$ è sottosequenza di un'altra $x_1 x_2 ... x_m$ se esiste una sequenza strettamente crescente di interi $(k_1, k_2, ..., k_i)$ con $0 < k_i \leq m$ tale che $w_i = x_{k_i}$.

Massima sottosequenza comune. Una stringa w si definisce *massima sottosequenza comune* tra N stringhe $S_1, S_2, ..., S_n$ se è sottosequenza di tutte le stringhe $S_1, S_2, ..., S_n$ ed è la massima (quella che contiene il maggior numero di caratteri) tra le tutte le stringhe che hanno questa proprietà.

La massima sottosequenza comune che indicheremo con $\mathrm{LCS}(S_1, S_2, ..., S_n)$può non essere unica, mentre sarà ovviamente unica la *lunghezza* della massima sottosequenza comune che denotiamo $\mathrm{lcs}(S_1, S_2, ..., S_n)$.

Ad sempio:

$$\mathrm{LCS}(\mathbf{a}\mathtt{c}\mathbf{i}\mathtt{d}\mathbf{o}, \mathtt{tart}\mathbf{a}\mathtt{r}\mathbf{i}\mathtt{c}\mathbf{o}) = \mathbf{aio}$$

ma anche:

$$\mathrm{LCS}(\mathbf{ac}\mathtt{id}\mathbf{o}, \mathtt{tart}\mathbf{a}\mathtt{ri}\mathbf{co}) = \mathbf{aco}.$$

Comunque sia:

$$\mathrm{lcs}(\mathtt{acido}, \mathtt{tartarico}) = 3$$

Algoritmi per il calcolo della massima sottosequenza comune.

Ci occupiamo, per prima cosa di risolvere un'istanza semplificata del problema calcolando la sola *lunghezza della massima sottosequenza comune tra 2 stringhe*; una volta trovata la soluzione a questa istanza ristretta prenderemo in considerazione il caso generale (ovvero trovare quale sia effettivamente questa sottosequenza e considerare il problema riferito a un numero arbitrario di stringhe).

Algoritmo ricorsivo. Il primo metodo che si presenta alla mente consiste senza dubbio in un approccio ricorsivo. Supponiamo che le due stringhe x e y lunghe rispettivamente n e m caratteri di cui si deve calcolare la lcs terminino con lo stesso carattere. In questo caso di sicuro $\mathrm{LCS}(x, y)$ conterrà questo carattere e $\mathrm{lcs}(x, y)$ sarà maggiore o

uguale 1 e il problema si ridurrebbe al calcolo di lcs(x', y') dove x' è x privato del suo ultimo carattere e lo stesso dicasi per y' rispetto a y.

Se invece gli ultimi due caratteri di x e y differiscono, allora LCS(x, y) non potrà contenerli entrambi (e forse potrebbe non contenerne nessuno dei due). lcs(x, y) sarà quindi uguale al maggiore tra lcs(x', y) e lcs(x, y') dove x' e y' hanno lo stesso significato attribuito loro precedentemente.

Siamo quindi facilmente giunti a una formulazione ricorsiva dell'algoritmo risolutivo per il problema:

```
function lcs_rec(var x:string; var y:string; i,j:integer) :integer;
begin
   if (i<=0) or (j<=0) then lcs_rec:=0
   else
      if x[i]=y[j] then lcs_rec:=1+lcs_rec(x,y,i-1,j-1)
      else lcs_rec := max(lcs_rec(x,y,i-1,j),lcs_rec(x,y,i,j-1))
end; { lcs_rec }
```

Notiamo che se le stringhe sono si lunghezze confrontabile($n \approx m$) e non hanno caratteri in comune allora l'algoritmo ricorsivo esegue approssimativamente 2^n passi.

Algoritmo di programmazione dinamica. È altresi facile notare che vi sono solo $n \cdot m$ possibili chiamate ricorsive diverse (in quanto gli indici i, j sono i soli veri dati di input della funzione lsc_rec(). Questo significa che le soluzioni a una grande moltitudine di sottoproblemi vengono ricalcolate più e più volte, inutilmente.

Per ovviare a questo problema sarebbe sufficiente memorizzare le soluzione dei sottoproblemi già risolti per poi richiamarle in un secondo tempo, senza ricalcolarle ogni volta. Ed è proprio in questa strategia che consiste l'algoritmo di programmazione dinamica che prendiamo in considerazione. L'algoritmo, per trovare lcs($x_1x_2...x_i$, $y_1y_2...y_j$), deve necessariamente conoscere:

- lcs($x_1x_2...x_{i-1}$, $y_1y_2...y_{j-1}$)
- lcs($x_1x_2...x_i$, $y_1y_2...y_{j-1}$)
- lcs($x_1x_2...x_{i-1}$, $y_1y_2...y_j$).

Ovviamente:

$$\text{lcs}(\varepsilon, y_j) = 0 \text{ per tutti i } j, \quad \text{lcs}(x_i, \varepsilon) = 0 \text{ per tutti gli } i \tag{1}$$

dove ε identifica la stringa vuota.

Inoltre come per l'algoritmo ricorsivo:

$$\text{lcs}(x_1...x_i, y_1...y_j) = \begin{cases} 1 + \text{lcs}(x_1...x_{i-1}, y_1...y_{j-1}) & \text{se } x_i = y_j \\ max(\text{lcs}(x_1...x_i, y_1...y_{j-1}), \\ \quad \text{lcs}(x_1...x_{i-1}, y_1...y_j)) & \text{se } x_i \neq y_j \end{cases} \tag{2}$$

Basta quindi immagazzinare l'informazione necessaria in una matrice $M = array[0..n][0..m]$ che in posizione i, j contenga lcs(i, j).

Dalla (1) e dalla (2) si ricava immediatamente l'algoritmo seguente:

```
function lcs_din(var x:string; var y:string) :integer;
var i,j : integer;
begin
   for i:=0 to length(x) do M[i][0]:=0;
   for i:=0 to length(y) do M[0][i]:=0;
```

```
    for i:=1 to length(x) do
      for j:=1 to length(y) do
          if x[i]=y[j] then M[i][j]:=1+M[i-1][j-1]
          else M[i][j]:= max(M[i-1][j],M[i][j-1]);
    lcs_din:=M[length(x)][length(y)];
end; { lcs_din }
```

Alla fine del ciclo $M[n][m]$ contiene lcs(x, y) È evidente che l'algoritmo di programmazione dinamica risolve il problema della massima sottosequenza comune tra 2 stringhe lunghe rispettivamente n e m caratteri utilizzando un tempo e una quantità di memoria entrambi proporzionali a mn.

Esempio. Vediamo ora come lavora l'algoritmo di programmazione dinamica per il calcolo di lcs(`dijkstra`, `knuth`).

Per prima cosa vengono poste uguali a zero tutte le celle della prima riga e della prima colonna (riga 0, colonna 0).

Si scandiscono poi tutte le celle della matrice, nel nostro caso seguendo l'ordine da sinistra verso destra e dall'alto verso il basso e, ad ogni cella, si applica la (2).

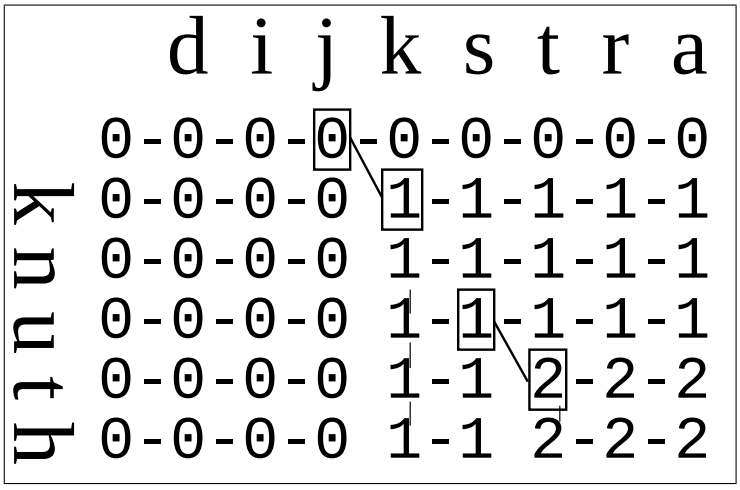

Figura 1: La matrice M.

I trattini in figura 1 indicano da dove proviene il valore di ogni cella, ovvero, se esiste un trattino che va dalla cella i alla cella j, questo significa che il valore della cella j è stato calcolato a partire da quello della cella i (dove, per calcolato, si intende o incrementato di 1 o lasciato invariato).

In particolare le caselle segnate con un rettangolo indicano dove si è avuto un aumento della lcs tra le stringhe parziali fino a quel momento analizzate. Alla fine, l'ultima casella in basso a destra contiene il valore di lcs(`dijkstra`, `knuth`)

Ricostruzione della soluzione. Se siamo interessati a conoscere, oltre alla lunghezza della massima sottosequenza comune, anche una di queste (ricordiamo che la LCS(x, y) può non essere unica) è necessario scandire la matrice $M = array[0..n][0.m]$ a ritroso, a partire dalla posizione (n, m) nel seguente modo:

```
function LCS(var x:string; var y:string; M:matrix):string
var tmp : string;
    i,j : integer;
begin
    tmp:='';
```

```
    i=length(x); j:=length(y);
    while(i>0) and (j>0) do
        if x[i]=y[j] then
        begin
            tmp:=x[i]+tmp;
            dec(i); dec(j);
        end
        else
            if M[i-1][j]>M[i][j-1] then dec(i)
            else dec(j);
    LCS=tmp;
end;
```

Generalizzazione.

Generalizziamo ora l'algoritmo di programmazione dinamica per il calcolo della massima sottosequenza comune tra 2 stringhe a un numero indefinito, diciamo N, stringhe S_1, S_2, ..., S_N

L'algoritmo è del tutto simile al precendente. Per conoscere, ad esempio, lcs($x_1...x_i$, $y_1...y_j$, $z_1...z_k$) sarà sufficiente conoscere

- lcs($x_1...x_{i-1}$, $y_1...y_j$, $z_1...z_k$)
- lcs($x_1...x_i$, $y_1...y_{j-1}$, $z_1...z_k$)
- lcs($x_1...x_i$, $y_1...y_j$, $z_1...z_{k-1}$)
- lcs($x_1...x_{i-1}$, $y_1...y_{j-1}$, $z_1...z_k$)
- lcs($x_1...x_i$, $y_1...y_{j-1}$, $z_1...z_{k-1}$)
- lcs($x_1...x_{i-1}$, $y_1...y_{j-1}$, $z_1...z_{k-1}$)

Servirà quindi una matrice M $= array[0..m][0..n][0..p]$ dove p è la lunghezza della stringa z per mantenere le informazioni necessarie, lo stesso dicasi se il numero delle stringhe delle quali calcolare la massima sottosequenza comune è maggiore di 3. L'algoritmo, nel caso di tre stringhe, è il seguente:

```
function lcs_din(var x : string; var y:string;var z:string) :integer;
var i,j,k       : integer;
begin
    for i:=0 to length(x) do M[i][0][0]:=0;
    for i:=0 to length(y) do M[0][i][0]:=0;
    for i:=0 to length(z) do M[0][0][i]:=0;

    for i:=1 to length(x) do
      for j:=1 to length(y) do
          for k:= 1 to length(z) do
          if (x[i]=y[j]) and (x[i]=z[k]) then M[i][j][k]:=1+M[i-1][j-1][k-1]
          else M[i][j][k]:= max(M[i-1][j][k],M[i][j-1][k],M[i][j][k-1],
                                M[i-1][j-1][k],M[i][j-1][k-1],M[i-1][j][k-1]);
    lcs_din:=M[length(x)][length(y)][length(z)];
end; { lcs_din }
```

Alla fine del ciclo $M[n][m][p]$ contiene lcs(x, y, z)

Ulteriori ottimizzazioni. Esistono metodi per diminuire ulteriormente la complessità spaziale dell'algoritmo di programmazione dinamica per il calcolo dell lcs. La trattazione di questo argomento esula dallo scopo di questa dispensa tuttavia, chi è interessato potrà trovare nella bibliografia ampi spunti di ricerca e approfondimento.

Riferimenti bibliografici

[1] A.V.Aho, D.S.Hirschberg, J.D.Ullman. Bounds on the complexity of the longest common subsequence problem. JACM V23 No1 p1-12 1976.

[2] D.S.Hirschberg. A linear space algorithm for computing maximal common subsequences. CACM V18 No6 p431-343 1975.

[3] D.S.Hirschberg. Algorithms for the longest common subsequence problem. JACM V24 No4 p664-675 1977.

[4] J.W.Hunt, T.G.Szymanski. A fast algorithm for computing longest common subsequences. CACM V20 No5 p350-353 1977.

[5] W.J.Masek, M.S.Paterson. How to compute string-edit distances quickly. in Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison. D.Sankoff, J.B.Kruskal eds. Addison-Wesley 1983.

[6] N.Nakatsu, Y.Kambayashi, S.Yajima. A longest common subsequence algorithm suitable for similar text strings. Acta Informatica V18 p171-179 1982.

[7] E.M.Reingold, J.Nievergelt, N.Deo. Combinatorial Algorithms: Theory and Practice. Prentice Hall 1977.

[8] D.Sankoff, J.B.Kruskal eds. Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley 1983.

[9] P.H.Sellers. On the theory and computation of evolutionary distances. SIAM Jrnl of Math' V26 No4 p787-793 1974.

[10] P.H.Sellers. The theory and computation of evolutionary distances: pattern recognition. Jrnl of Algorithms V1 No4 p359-373 1980.

[11] R.A.Wagner, M.J.Fischer. The string-to-string correction problem. JACM V21 No1 p168-173 1974.

[12] M.S.Waterman. General methods of sequence comparison. Bulletin of Math' Biology V46 No4 p473-500 1984.

[13] C.K.Wong, A.K.Chandra. Bounds for the string editing problem. JACM V23 No1 p13-16 1976.