IL PROBLEMA LCS E LE SUE APPLICAZIONI

1. Introduzione

Il *Longest Common Subsequence Problem* (*LCS problem*) è un problema di *string matching* che consiste nel trovare una sottosequenza massima comune a due sequenze date di caratteri.
Esso trova applicazione in vari ambiti: nei programmi di testo, ad esempio per ricercare parole del dizionario che si avvicinano ad una data parola inesistente, in biologia molecolare, per ricercare correlazioni tra due organismi confrontando i loro DNA, acidi nucleici rappresentati da sequenze di quattro basi (adenina, citosina, guanina, timina) collegate tra loro, che altro non sono che sequenze di simboli nell'alfabeto {A, C, G, T}, oppure nel confronto tra file di testo.
Formalmente, fissato un *alfabeto di input* Σ, contenente σ simboli, e indicato con Σ^* l'insieme di tutte le possibili stringhe ottenute concatenando zero o più simboli di Σ (anche la parola vuota), consideriamo due stringhe X, $Y \in \Sigma^*$,
$X = < x_1, x_2, \ldots, x_m >$, $Y = < y_1, y_2, \ldots, y_n >$ di lunghezza, rispettivamente, *m* e *n*.
Una *sottosequenza* di X è una stringa ottenuta a partire da X cancellando zero o più elementi, mantenendo però l'ordine degli elementi rimasti. Più precisamente una stringa $Z \in \Sigma^*$, $Z = < z_1, z_2, \ldots, z_k >$ è una sottosequenza di X se:

$$\exists\, i_1, i_2, \ldots, i_k : 1 \leq i_1 < i_2 < \ldots < i_k \leq m \wedge z_j = x(i_j),\ \forall\ j = 1, 2, \ldots, k$$

Ad esempio, fissato Σ = {A,B,C,D}, se X = < A, B, C, B, D, A, B >, allora
Z = < B, C, D, B > è una sottosequenza di X, corrispondente alla sequenza di indici < 2, 3, 5, 7 >, detta *sequenza di corrispondenza*.
Una stringa $Z \in \Sigma^*$ è una *sottosequenza comune* a X e Y se è una sottosequenza sia di X che di Y. Ad esempio, se X = <A, B, C, B, D, A, B> e Y = <B, D, C, A, B, A>, allora Z = <B, C, A> è una sottosequenza comune a X e Y.
Il problema della più lunga sottosequenza comune consiste nel determinare, date due sequenze X e Y, *una* sottosequenza comune a X e Y di lunghezza *massima*

(LCS). In generale, non esiste un'unica LCS di due date sequenze: nell'esempio precedente <B, C, B, A>, <B, C, A, B> e < B, D, A, B> sono tre LCS di X e Y.

2. Algoritmo di base

Un algoritmo efficiente, dovuto a Wagner e Fischer [1], per determinare la lunghezza di una LCS di due date sequenze X e Y, di lunghezza, rispettivamente, m e n, utilizza la tecnica della *programmazione dinamica*, partendo dal calcolo della lunghezza di una LCS per combinazioni di prefissi delle due stringhe. Un prefisso X_i di una stringa X, con $|X| = n$, è una stringa di lunghezza i costituita dai primi i caratteri di X, con $0 \leq i \leq n$. Esso consente poi di costruire anche una LCS.
La relazione di ricorrenza per tale algoritmo è la seguente:

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ oppure } j = 0 \\ c[i-1, j-1] + 1 & \text{se } X[i] = Y[j] \\ \max(c[i-1, j], c[i, j-1]) & \text{se } X[i] \neq Y[j] \end{cases}$$

dove $c[i,j]$ rappresenta la lunghezza di una LCS di X_i e Y_j, prefissi, rispettivamente di X e Y.

L’algoritmo per il calcolo della lunghezza di una LCS di X e Y basato su questa relazione di ricorrenza è il seguente:

```
LCS_LENGTH (X, Y, c, b)

BEGIN
m := length[X];
n := length[Y];
for i := 1 to m do
  c[i,0] := 0;
for j := 1 to n do
  c[0,j] := 0;
for i := 1 to n do
  for j := 1 to m do
       if X[i] = Y[j] then
            c[i,j] := c[i-1,j-1] + 1;
            b[i,j] := “↖”;
       else if c[i-1,j] ≥ c[i,j-1] then
            c[i,j] := c[i-1,j];
            b[i,j] := “↑”;
       else
            c[i,j] := c[i,j-1];
            b[i,j] := “←”;
return c[m,n];
END
```

Esso consiste di fatto nel riempimento di una tabella *c* di *m* righe e *n* colonne, quindi la complessità dell’algoritmo è O(*mn*). Il valore di *c*[*m*,*n*] nella tabella corrisponde alla lunghezza di una LCS tra X e Y.

In riferimento alle sequenze X e Y dell'esempio precedente, la tabella calcolata dall'algoritmo è la seguente:

X \ Y		0	1 B	2 D	3 C	4 A	5 B	6 A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	**1**	1	**1**
2	B	0	**1**	1	1	1	**2**	2
3	C	0	1	1	**2**	2	2	2
4	B	0	**1**	1	2	2	**3**	3
5	D	0	1	**2**	2	2	3	3
6	A	0	1	2	2	**3**	3	**4**
7	B	0	**1**	2	2	3	**4**	4

Nell'esempio $|LCS(X,Y)| = 4$. L'algoritmo utilizza anche un'altra tabella *b* per memorizzare informazioni, nella forma di puntatori, a partire dalle quali, tramite un meccanismo di backtracking da $c[m,n]$ è possibile ottenere i simboli corrispondenti di X e Y che hanno portato alla soluzione e quindi costruire effettivamente una LCS di X e Y.

3. Ottimizzazioni

La complessità dell'algoritmo di base è $O(mn)$ in tutti i casi e per qualunque coppia di sequenze X e Y di lunghezza, rispettivamente, *m* e *n*. E' possibile ridurre tale complessità nel caso medio utilizzando strategie di ottimizzazione per arrivare alla soluzione [2,3,4]. Supponiamo, nel seguito, che $m \leq n$. Alcune locazioni della tabella sono più importanti di altre, perché fanno riferimento alle cosiddette *corrispondenze dominanti*.

In generale, date due sequenze X e Y di lunghezza *m* e *n* rispettivamente, una coppia (i, j) è detta *corrispondenza* se $X[i] = Y[j]$.

L’insieme di tutte le corrispondenze è:

$M = \{ (i, j) \mid X[i] = Y[j], 1 \leq i \leq m, 1 \leq j \leq n \}$.

In riferimento alla tabella dell’esempio, tutte le locazioni (i, j) con valori in grassetto o in rosso sono corrispondenze.
Indicando con l la lunghezza di una LCS(X,Y), ogni corrispondenza appartiene ad una classe C_k

$C_k = \{ (i, j) \mid (i, j) \in M \wedge c[i,j] = k \}$

con $1 \leq k \leq l$. Una corrispondenza che appartiene a C_k è detta k-corrispondenza.
Formalmente, in modo ricorsivo, definita la pseudoclasse $C_0 = \{(0,0)\}$, una corrispondenza (i, j) è detta *k-corrispondenza* se esistono i_1 e j_1 tali che $i_1 < i$, $j_1 < j$ e (i_1, j_1) è una (k-1)-corrispondenza. Si dice che (i_1, j_1) *genera* (i, j).
Nell’esempio si ha:

$C_1 = \{ (1,4), (1,6), (2,1), (4,1), (7,1) \}$
$C_2 = \{ (2,5), (3,3), (5,2) \}$
$C_3 = \{ (4,5), (6,4) \}$
$C_4 = \{ (6,6), (7,5) \}$

Poiché ogni corrispondenza appartiene ad una e una sola classe, gli insiemi C_k formano una partizione nell’insieme M. Alcune k-corrispondenze, però, sono più importanti di altre.
Siano (i_1, j_1) e (i_2, j_2) due k-corrispondenze. Se $i_1 \geq i_2$ e $j_1 \geq j_2$, allora possiamo dire che (i_2, j_2) *esclude* (i_1, j_1), perché qualsiasi (k+1)-corrispondenza che può essere generata da (i_1, j_1) può essere generata anche da (i_2, j_2). Ciò segue dalla definizione formale di k-corrispondenza.
Questo significa che per il calcolo delle (k+1)-corrispondenze sono sufficienti solo alcune k-corrispondenze particolari, dette *k-corrispondenze dominanti*, che si possono ottenere applicando la precedente regola di esclusione agli elementi dell’insieme C_k.

Nella tabella dell'esempio tutte le locazioni (i, j) con valori in rosso sono corrispondenze dominanti. Come si può osservare, per ogni riga esiste al più una k-corrispondenza dominante.
Le corrispondenze dominanti godono della *proprietà di ordinamento*:

Se D_k è l'insieme delle k-corrispondenze dominanti, $D_k = \{(i_1, j_1),(i_2, j_2), \ldots,(i_k, j_k)\}$, allora le corrispondenze possono essere ordinate in modo tale che $i_1 < i_2 < \ldots < i_k$ e $j_1 > j_2 > \ldots > j_k$.

Nella tabella dell'esempio le regioni dove le locazioni hanno lo stesso valore sono limitate da linee spezzate, chiamate *contorni*, tanti quanti la lunghezza l di una LCS. Come si può osservare, tutte le k-corrispondenze stanno immediatamente al di sotto del k-esimo contorno e tutte le locazioni delle corrispondenze dominanti definiscono gli angoli di queste linee, quindi possono essere individuate facilmente nella tabella, una volta tracciati i contorni.
Per determinare la lunghezza di una LCS (e una soluzione al problema), quindi, basta individuare le corrispondenze dominanti. Il seguente lemma è utile nel localizzare le corrispondenze dominanti:

Lemma 1 [3]:
Sia $k \geq 1$. (i, j) è una k-corrispondenza dominante se j è il minimo valore tale che $X_i = Y_j$ e $low < j < high$, dove *high* è il minimo indice j_1 tra tutte le k-corrispondenze (i_1, j_1) per cui $i_1 < i$ (se esiste, altrimenti è n+1) e *low* è il minimo indice j_2 tra tutte le (k-1)-corrispondenze (i_2, j_2) per cui $i_2 < i$.

Dimostrazione:
Sia (i, j) una k-corrispondenza dominante e sia per assurdo $j \geq high$. Allora, per la definizione di *high*, esiste una k-corrispondenza (i_1, j_1) tale che $j_1 = high \leq j$ e $i_1 <$ i. Questa corrispondenza, dunque, esclude (i, j), che non sarebbe dominante. Quindi deve essere $j < high$.
Se per assurdo, inoltre, $j \leq low$, allora non ci sarebbe nessuna (k-1)-corrispondenza in grado di generare (i, j), che non sarebbe dominante. Quindi $j > low$. ■

Esistono tre approcci differenti per localizzare le corrispondenze dominanti e raggiungere una soluzione.

4. Primo approccio : processare la tabella riga per riga

Il primo approccio consiste nel processare la tabella riga per riga, seguendo lo stesso procedimento di riempimento della tabella dell'algoritmo di base, ma concentrando la propria attenzione solo sulle locazioni delle corrispondenze dominanti [2,5].
Per seguire questo approccio è necessario un array ausiliare, che chiameremo *MinYPrefix*, di dimensione *m*, pari alla lunghezza di X. *MinYPrefix*[*k*] rappresenta l'indice della colonna dove è localizzato il k-esimo contorno e, come suggerisce il nome, può essere visto come la lunghezza minima del prefisso di Y necessario per produrre una sottosequenza comune di lunghezza *k* col prefisso di X di lunghezza *i*.
L'approccio consiste nel calcolare, al variare di *i*, ovvero riga per riga, le posizioni delle k-corrispondenze dominanti, memorizzandole in *MinYPrefix*, utilizzando i risultati del *Lemma 1*, per delimitare l'intervallo in cui ricercarle. Una (k+1)-corrispondenza, quindi, andrà ricercata considerando *MinYPrefix*[*k*] come *low* e *MinYPrefix*[*k+1*] come *high*.
Inizialmente tutti gli elementi dell'array sono inizializzati ad un valore indefinito, cioè *n+1*. Al variare di *i*, il contenuto di *MinYPrefix*[*k*] può cambiare in base alla seguente *regola di aggiornamento*:

Supponiamo di processare la riga *i*. Per ogni intervallo di ricerca *MinYPrefix*[*k*] … *MinYPrefix*[*k+1*] aperto, si verifica se esiste almeno una corrispondenza (*i*, *j*) tale che l'indice *j* cada in tale intervallo. Se non esiste alcuna corrispondenza l'estremo destro dell' intervallo, ovvero *MinYPrefix*[*k+1*], rimane invariato, altrimenti viene aggiornato con il valore del più piccolo indice *j* per cui esiste una corrispondenza (*i*, *j*) che cade nell'intervallo.

La regola di aggiornamento assicura che le condizioni del *Lemma 1* siano soddisfatte nel momento in cui si trova una k-corrispondenza dominante.

Lo schema generale per trovare la lunghezza *l* di una LCS è il seguente:

```
BEGIN
for i := 1 to m do
   MinYPrefix[i] := n+1;
MinYPrefix[0] := 0;
r :=0;
for i := 1 to m do
   /*  Aggiorna l'array di valori per la riga i. */
   for j := 0 to r do
        if range (MinYPrefix[j] ... MinYPrefix[j+1])
        contains matches then
        /*  low < j < high. */
        MinYPrefix[j+1]:= min{k|(i,k) is a match in this
        range };
        if j = r then
             r := r + 1;
return r;
END
```

La variabile *r* contiene il numero di contorni via via scoperti dalla procedura e viene aggiornata ogni volta che viene trovato un nuovo contorno, ovvero ogni volta che una locazione *undefined* dell'array *MinYPrefix* (con valore *n+1*) viene aggiornata con un valore definito.
Utilizzando una lista concatenata è possibile inoltre memorizzare come nodi i caratteri di una LCS per ricostruire alla fine la soluzione in tempo al massimo O(*m*+*n*).

Le diverse implementazioni di questo approccio si distinguono in base al modo in cui viene effettuato il ciclo **for** più interno e la ricerca del minimo.
L'algoritmo di Hunt e Szymanski [2] processa la tabella riga per riga ed effettua una ricerca delle corrispondenze da destra verso sinistra e per ciascuna corrispondenza trovata aggiorna l'array *MinYPrefix* applicando la regola di aggiornamento.

Per rendere efficiente l'algoritmo, invece di scorrere per intero una riga alla ricerca delle corrispondenze, si preprocessa l'input e si costruisce un array di liste concatenate, MATCHLIST, di lunghezza *m* pari alla lunghezza di X.
MATCHLIST[*i*] contiene gli indici *j* delle corrispondenze (*i*,*j*) presenti nella riga *i*, ordinati in senso decrescente.
L'array MATCHLIST può essere ottenuto con una procedura che richiede tempo O(*m*lg*n*). Si costruisce una sequenza $\hat{Y}$ e si ordinano i caratteri di Y, utilizzando come prima chiave i simboli (in senso crescente) e come seconda chiave le relative posizioni (in senso decrescente).
In riferimento alla sequenza Y dell'esempio, se indichiamo tra parentesi le posizioni dei simboli nella sequenza si ha:

Y = < B (1), D (2), C (3), A (4), B (5), A (6) >

$\hat{Y}$ = < A (6), A (4), B (5), B (1), C (3), D (2) >

A questo punto si scandisce la sequenza X e, per ciascun carattere, si cercano le corrispondenze in $\hat{Y}$ effettuando una ricerca binaria. Se per un carattere la MATCHLIST è già stata calcolata, si può evitare di ripetere la ricerca binaria.
Per le sequenze X e Y dell'esempio si ottengono le seguenti liste:

MATCHLIST [1] = < 6, 4 >
MATCHLIST [2] = < 5, 1 >
MATCHLIST [3] = < 3 >
MATCHLIST [4] = MATCHLIST [2]
MATCHLIST [5] = < 2 >
MATCHLIST [6] = MATCHLIST [1]
MATCHLIST [7] = MATCHLIST [2]

L'algoritmo utilizza infine una lista concatenata, LINK, per memorizzare i caratteri di una LCS. Scorrendo questa lista dalla coda, è possibile restituire anche una LCS (X, Y).

```
HUNT_SZYMANSKI (X, Y)

BEGIN
integer array MinYPrefix [0:m];
list array MATCHLIST [1:m];
linked list LINK;
pointer PTR;
integer r;                  /* Lunghezza di una LCS. */
/* Passo 1: Costruzione array MATCHLIST (Pre-
processing). */
…

/* Passo 2 : Inizializzazione di r, MinYPrefix, LINK. */
r := 0;
MinYPrefix [0] := 0;
for i := 1 to m do
  MinYPrefix [i] := n+1;
LINK[0] := null;

/* Passo 3 : Calcolo dei valori di MinYPrefix. */
for i := 1 to n do
  for j on MATCHLIST[i] do
       trova k tale che MinYPrefix[k-1] < j <
MinYPrefix[k]
       if k exists then
            oldvalue := MinYPrefix[k];
            MinYPrefix[k] := j;
            if oldvalue = n+1 then
            /* MinYPrefix[k] era una locazione con
            valore undefined: ho scoperto un nuovo
            contorno e un nuovo carattere da concatenare
            ad una LCS(X,Y). */
                 r := r + 1;
                 LINK[k] := newnode (i, j, LINK[k-1]);
```

```
/* Passo 4: stampa di una soluzione. */
PTR := LINK[1];
while PTR ≠ null do
   print nodo (i,j) puntato da PTR;
   advance PTR;
return r;
END
```

Per quanto riguarda la complessità dell'algoritmo, si può osservare che il passo 1, per quanto detto in precedenza, richiede tempo O($m$lg$n$), mentre il passo 2 richiede tempo O(m).
La ricerca di k al passo 3 può essere effettuata in tempo O(lg$n$) effettuando una ricerca binaria sull'array MinYPrefix, perché all'inizio di ogni iterazione del ciclo for esterno gli elementi sono ordinati in senso crescente. Poiché la ricerca di k viene effettuata per ogni corrispondenza, indicando con p il numero totale di corrispondenze, il passo 3 richiede tempo O($p$lg$n$).
Il passo 4, infine, richiede tempo al più O(n), in quanto $|\text{LCS}(X,Y)| \leq \max(m,n)$.
La complessità dell'algoritmo è, dunque, O(($p + m$)lg n), ovvero O(($p + n$)lg n) supponendo che $m \leq n$. Ciò significa che nel caso peggiore, in cui $p \sim n^2$ e le due sequenze sono quasi identiche, l'algoritmo si comporta leggermente peggio rispetto a quello di base, ma nel caso medio, in cui $p \sim n$, l'algoritmo ha complessità O($n$lg$n$) ed è più efficiente. La complessità spaziale è lineare in m, migliore di quella dell'algoritmo di base, che è O(mn), cioè quadratica.

5. Secondo approccio: processare la tabella contorno per contorno

Lo svantaggio del primo approccio è dato dal fatto che occorre processare per ogni riga tutte le corrispondenze per l'intera stringa Y. Il secondo approccio consiste nel processare la tabella avanzando contorno per contorno [3,5].
Per ogni k-corrispondenza dominante si determinano tutte le possibili (k+1)-corrispondenze che stanno all'interno di una *regione di interesse*, i cui confini sono

determinati dalla k-corrispondenza dominante in esame e da quella successiva, e dall'ultima (k+1)-corrispondenza dominante trovata. La ricerca delle corrispondenze, quindi, non è più globale, ma locale, e viene effettuata per valori crescenti di *k*, a partire dalla 0-corrispondenza (0,0).

Supponiamo di aver trovato una k-corrispondenza dominante (*i*, *j*) e siano (*u*, *v*) la successiva k-corrispondenza dominante e (*s*, *t*) l'ultima (k+1)-corrispondenza dominante trovata. La regione di interesse in cui cercare la successiva (k+1)-corrispondenza dominante è definita dagli intervalli [*s*+1 … *u*] × [*j*+1 … *t*-1]. Se non esiste alcuna (k+1)-corrispondenza precedente, allora la massima colonna è *n* anziché *t*-1 e la minima riga è *i*+1 anziché *s*+1. Se non esiste una successiva k-corrispondenza, allora la massima riga è *m* anziché *u*.

Una volta determinata, la regione di interesse può essere processata utilizzando il primo approccio (riga per riga, da destra verso sinistra). Quando tutte le k-corrispondenze sono state processate, si passa al contorno successivo.

Il numero *k* di contorni esaminati rappresenta la lunghezza di una LCS.

Lo schema generale basato sul secondo approccio è il seguente:

```
BEGIN
D0 := {(0,0)};          /* 0-corrispondenza dominante. */
k := 0;
while  Dk ≠ ∅ do
   while esistono k-corrispondenze dominanti (i, j) in Dk
do
   /*Esamina le corrispondenze dominanti da destra a
   sinistra.*/
        Trova la regione di interesse per (i, j);
        for each (k+1)-corrispondenza dominante (s, t)
        nella regione di interesse do
             aggiungi (s, t) a Dk+1;
   k := k + 1;
return k - 1;
END
```

La complessità delle implementazioni di questo approccio è legata al criterio con cui viene determinata la regione di interesse.
L'algoritmo di Hirschberg [3] sfrutta la proprietà di ordinamento delle corrispondenze dominanti per determinare riga per riga dove inizia il k-esimo contorno e il *Lemma 1* per determinare in quale intervallo ricercare la successiva corrispondenza dominante.
L'algoritmo utilizza una matrice D per memorizzare riga per riga l'inizio dei contorni, ovvero la posizione delle corrispondenze dominanti. D[*k*, *i*] rappresenta l'indice *j* della k-corrispondenza dominante che si trova sulla i-esima riga della tabella, oppure D[*k*, *i*-1] se la corrispondenza non esiste.
Come per l'algoritmo di Hunt e Szymanski, è possibile costruire un'array di liste MATCHLIST, di dimensione *m*, per memorizzare la lista delle corrispondenze e rendere più efficiente la ricerca, preprocessando gli input X e Y.
Per ottimizzare ulteriormente la ricerca su un contorno *k* e non esaminare più volte le stesse corrispondenze, è possibile utilizzare un array di puntatori, PTR.
PTR[i] punterà al primo elemento di MATCHLIST[i] non ancora esaminato nella ricerca delle corrispondenze dominanti sul contorno *k*. Ciò assicura che, per ogni contorno *k*, ogni elemento di Y viene esaminato al più una volta, se fa parte di qualche k-corrispondenza. Come per l'array MATCHLIST, se per un carattere la lista delle corrispondenze è già stata calcolata, si possono riutilizzare le stesse MATCHLIST e gli stessi puntatori già creati, utilizzando, quindi, una sola lista e un solo puntatore per ogni carattere di Y. Per i puntatori questo procedimento è necessario se non si vogliono esaminare più volte le stesse corrispondenze nella ricerca su un contorno *k*.
La variabile *lowcheck* corrisponde alla riga in cui inizia il k-esimo contorno e consente di stabilire la riga a partire dalla quale cominciare la ricerca delle (k+1)-corrispondenze dominanti.
Le variabili *low* e *high* stabiliscono gli estremi inferiore e superiore di ricerca delle corrispondenze dominanti, in base alle condizioni del *Lemma 1*. L'estremo superiore *high* viene aggiornato ogni volta che viene trovata una corrispondenza, in base alla regola di aggiornamento.
La variabile booleana FLAG vale 1 se esiste almeno una k-corrispondenza, 0 altrimenti. In quest'ultimo caso vuol dire che non esiste un k-esimo contorno.

Questa variabile consente di stabilire quanti contorni sono presenti, e quindi qual è la lunghezza di una LCS.
A partire dalla matrice D, infine, è possibile risalire tramite backtracking anche ad una soluzione del problema (nell'algoritmo Z rappresenta una LCS).

```
HIRSCHBERG (X, Y)

BEGIN
integer matrix D;
list array MATCHLIST [1:m];
pointer array PTR [1:m];
integer lowcheck;
integer low;
integer high;
boolean flag;
/* Passo 1: Costruzione array MATCHLIST (Pre-
processing). */
…

/* Passo 2:  Inizializzazione di D, lowcheck, k e FLAG.
*/
for i := 0 to m do
  D [0, i] := 0;
lowcheck := 0;
k := 1;
FLAG := 1;

/* Passo 3: Calcolo delle k-corrispondenze dominanti
contorno per contorno a partire dal primo (k=1). */
1) while FLAG ≠ 0 do
  FLAG := 0;
  low := D [k-1, lowcheck];
  high := n + 1;
```

```
   /* Inizializzazione dei puntatori dell'array PTR al
   primo elemento di ogni lista. */

  2) for i := lowcheck + 1 to m do
        if MATCHLIST[i] ≠ ∅ then
             3) while j successivo a quello puntato da
             PTR[i]>low do
                  advance PTR[i];
             if high >  j puntato da PTR[i]  > low then
                  high := j;
                  D[k, i] := high;
                  if FLAG = 0 then
                       lowcheck := i;
                       FLAG := 1;
             else
                  D[k, i] := D[k, i-1];
             if D[k-1, i] ≠ D[k-1, i-1] then
                  low := D[k-1, i];
        else
             D[k, i] := D[k, i-1];
   k := k + 1;

/* Passo 4: calcolo della lunghezza di una LCS e stampa
di una soluzione
k := k -1;
for i := m to 1 do
   if D[k, i] ≠ D[k, i-1] then
        Z[k] := X[i]
        k := k - 1;
return k, Z;
END
```

La complessità dell'algoritmo di Hirschberg è O(r(n + m) + $n$log$n$), se r è la lunghezza di una LCS, ovvero O(rn + $n$lg$n$) assumendo che $m \leq n$.
Il passo 1, come per l'algoritmo di Hunt e Szymanski, richiede tempo O($n$lg$n$), mentre il passo 2 richiede tempo O(m).
Per quanto riguarda il passo 3, il ciclo 1) viene eseguito r + 1 volte, se r è la lunghezza di una LCS, perché viene esaminato anche il contorno r + 1, che non contiene corrispondenze e lascia in tal caso FLAG pari a 0. La procedura di inizializzazione dei puntatori all'interno del ciclo 1) ha complessità O(m), perché consiste nello scorrere l'array MATCHLIST per intero.
Non è possibile stabilire quante volte venga eseguito il ciclo 2), ma possiamo osservare che il ciclo 3) interno, viene eseguito complessivamente, cioè considerando tutte le iterazioni del ciclo 2), O(n) volte, perché, come osservato in precedenza, l'utilizzo dei puntatori assicura che, per ogni contorno k, ogni elemento di Y viene esaminato *al più* una volta, se fa parte di qualche k-corrispondenza. Il resto delle operazioni all'interno del ciclo 2) richiede tempo costante. Quindi il passo 3) può essere completato in tempo O(r(m+n)).
Il passo 4, infine, richiede tempo O(m). Quindi la complessità dell'algoritmo è O(rn + $n$lg$n$). Ciò significa che nel caso peggiore, in cui $r \sim m$ (essendo $m \leq n$) e le due sequenze sono quasi identiche, l'algoritmo ha complessità O(mn) pari a quella dell'algoritmo di base, ma nel caso medio, più frequente, l'algoritmo è più efficiente. La complessità spaziale è O(rn), per via dell'utilizzo della matrice D, cioè O(mn) nel caso peggiore.

6. Terzo approccio: processare la tabella lungo le diagonali

Il terzo approccio consiste nel processare la tabella muovendosi lungo le diagonali a partire dalla locazione (0,0), in modo da raggiungere il più velocemente possibile la locazione c[m,n], utilizzando un approccio *greedy* [4,5]. Gli algoritmi che implementano il terzo approccio si basano sul calcolo della *distanza di edit* tra due sequenze di input.

Date due stringhe X e Y, la distanza di edit tra X e Y, indicata con D, è il numero minimo di operazioni elementari (inserimenti e cancellazioni) necessarie per trasformare X in Y.
Il problema del calcolo della distanza di edit è strettamente legato al problema dell'LCS: se *r* è la lunghezza di una LCS di X e Y, per trasformare X in Y si possono prima cancellare $m - r$ simboli da X per ottenere una LCS di X e Y e poi inserire $n - r$ simboli in quest'ultima sequenza per ottenere X, quindi occorrono $m + n - 2 * r$ operazioni elementari. Segue che:

$$D(X,Y) = m + n - 2 * r(X,Y)$$

Ovvero:

$$r(X,Y) = (m + n - D(X,Y)) / 2$$

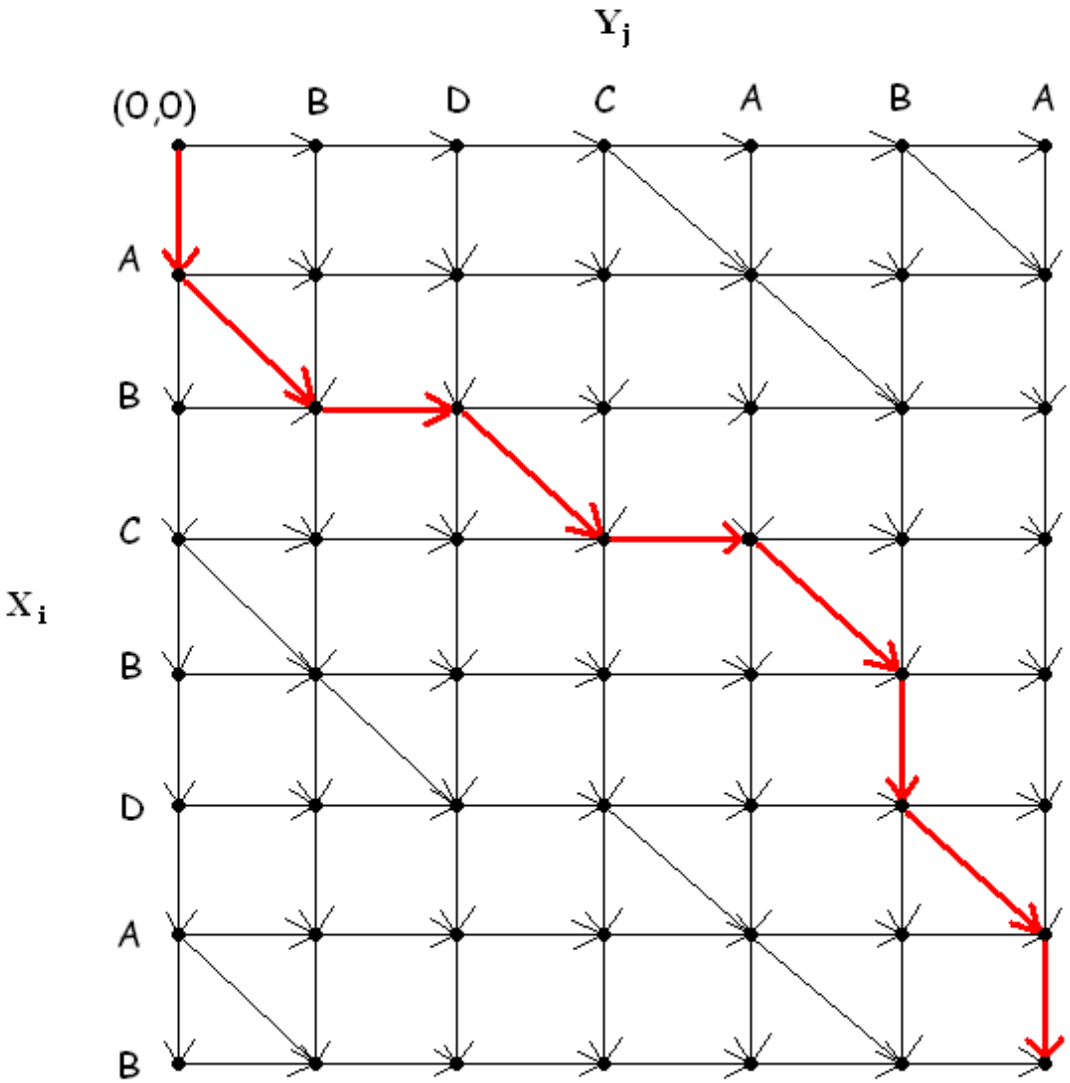
L'ultima relazione rappresenta un altro modo di calcolare la lunghezza di una LCS, conoscendo la distanza di edit tra X e Y. Come si può osservare, aumentando *r*, diminuisce la distanza D, e viceversa, aumentando la lunghezza di una LCS, *r* diminuisce. Inoltre, poiché $r \geq 0$, deve essere sempre $D \leq m + n$.

Un' implementazione di questo approccio è l'algoritmo di Miller e Myers [4], in cui la distanza di edit viene calcolata costruendo, a partire dalla tabella, un grafo, detto *grafo di edit*.
I vertici del grafo sono tutte le locazioni (i, j) della tabella e sono connessi da archi orizzontali, verticali ed eventualmente diagonali orientati. Gli *archi orizzontali* collegano ciascun vertice al suo vicino a destra (se esiste), gli *archi verticali* collegano ciascun vertice al suo vicino in basso (se esiste), un vertice $(i-1, j-1)$ è collegato al vertice (i, j) mediante un *arco diagonale* se (i, j) esiste e $X_i = Y_j$, cioè (i, j) è una corrispondenza.

La figura seguente mostra il grafo di edit relativo alle sequenze X e Y dell'esempio:

LCS = <B, C, B, A>

Traccia = (2,1) (3,3) (4,5) (6,6)



Una sequenza di lunghezza L, costituita da L corrispondenze $(x_1, y_1)\ (x_2, y_2)\ldots(x_L, y_L)$, è detta *traccia* se $x_i < x_{i+1}$ e $y_i < y_{i+1}$ con $1 \leq i \leq L\text{-}1$. Essa corrisponde alla sequenza delle corrispondenze attraversate con un cammino dal vertice (0,0) a (m,n). Dal momento che ad ogni corrispondenza è associato un arco diagonale, L rappresenta anche il numero di archi diagonali presenti su tale cammino. Non è detto che ad una traccia corrisponda un solo cammino: una traccia

determina una sequenza di archi diagonali, che possono essere collegati tra loro in vari modi con archi orizzontali e verticali. Ad ogni sottosequenza comune di X e Y di lunghezza *k*, $< X(i_1), X(i_2), \ldots, X(i_k) > = < Y(j_1), Y(j_2), \ldots, Y(j_k)>$ corrisponde una traccia $(i_1, j_1), \ldots, (i_k, j_k)$, e quindi un cammino da (0,0) a (*m*,*n*) contenente *k* archi diagonali.

Ciò significa che trovare la distanza di edit tra X e Y, ovvero la lunghezza di una LCS, equivale a trovare un cammino da (0,0) a (*m*,*n*) con il maggior numero di archi diagonali, ovvero con il minor numero di archi orizzontali o verticali. Tale numero corrisponde alla distanza di edit o alla lunghezza di una LCS. La figura evidenzia un cammino corrispondente alla LCS <B,C,B,A>.

Sia ora un *D-cammino* un cammino che inizia in (0,0) che contiene esattamente D archi non diagonali. Uno 0-cammino, quindi, deve essere formato solo da archi diagonali e, per induzione, un D-cammino deve essere formato da un (D-1)-cammino, seguito da un arco non diagonale e una sequenza, anche vuota, di archi diagonali, chiamata *serpente*.

Numeriamo le diagonali della tabella su cui è costruito il grafo in modo tale che gli elementi (*i*, *j*) stanno sulla diagonale *j* – *i*. Quindi le coppie (0,0), (1,1), …, (*m*,*m*) stanno sulla diagonale 0, le diagonali 1, 2, 3, …, *n* si trovano al di sopra di essa e le diagonali -1, -2, -3, …, -*m* si trovano al di sotto di essa. Osserviamo che, partendo da un vertice sulla diagonale *k*, percorrendo un arco orizzontale ci spostiamo sulla diagonale *k* + 1, percorrendo un arco verticale ci spostiamo sulla diagonale *k* – 1, percorrendo un arco diagonale, o in generale un serpente, rimaniamo ancora sulla diagonale *k*.

Una proprietà riguardante i D-cammini è espressa dal seguente:

Lemma 2 [4]:

Un D-cammino deve terminare su una diagonale *k*, con $k \in \{-D,-D+2, \ldots, D-2, D\}$.

Dimostrazione:

La proprietà si può dimostrare per induzione su $D \geq 0$. Uno 0-cammino ha solo archi diagonali, quindi inizia e termina sulla diagonale $k = D = 0$. Fissato D, supponiamo che la proprietà sia vera per un D-cammino. Un (D+1)-cammino, come osservato in precedenza, è costituito da un D-cammino che supponiamo termini sulla diagonale *k*, un arco non diagonale, che ci porta sulla diagonale *k* +1 o *k*-1, e

un serpente che termina anch'esso sulla diagonale *k*+1 o *k*-1. Quindi ogni (D+1)-cammino termina in una diagonale in { -D±1, (-D+2)±1, …, (D-2)±1, D±1 }, ovvero nell'insieme { -D-1, -D+1, …, D-1, D+1 } = { -(D+1), -(D+1)+2, …, (D+1)-2, D+1 }. ■

Il *Lemma 2* implica che esistono solo D+1 diagonali in cui un D-cammino può terminare e un D-cammino termina in una diagonale pari se D è pari, dispari se D è dispari.
Un D-cammino è detto *furthest reaching* sulla diagonale *k* se è un D-cammino che termina sulla diagonale *k* nel punto (*i, j*) con indice di riga *i* massimo, cioè tra tutti i D-cammini che terminano sulla diagonale *k* è quello che raggiunge il punto più "distante" rispetto all'origine (0,0) del cammino. Il seguente lemma caratterizza i D-cammini furthest reaching:

Lemma 3 [4]:
Uno 0-cammino furthest reaching termina in (*i*,*i*), dove $i=\min\{k-1 \mid X_k \neq Y_k$ oppure k>M o k>N$\}$. Un D-cammino furthest reaching sulla diagonale *k* può essere decomposto in un (D-1)-cammino furthest reaching sulla diagonale *k*-1, seguito da un arco orizzontale e dal più lungo possibile serpente, oppure in un (D-1)-cammino furthest reaching sulla diagonale *k*+1, seguito da un arco verticale e dal più lungo possibile serpente.

Dimostrazione:
La proprietà sullo 0-cammino furthest reaching segue dalla definizione di 0-cammino. Come osservato in precedenza, un D-cammino consiste in un (D-1)-cammino, seguito da arco non diagonale e da un serpente. Il serpente finale deve essere massimo: se non lo fosse, il serpente potrebbe essere esteso e il D-cammino non sarebbe furthest-reaching. Supponiamo che il (D-1)-cammino non sia furthest reaching nella sua diagonale (*k*-1 o *k*+1). Allora è possibile collegare un (D-1)-cammino further reaching rispetto al precedente al serpente finale mediante un appropriato arco non diagonale. Quindi un D-cammino può essere sempre decomposto in questo modo. ■

Il *Lemma 3* suggerisce un criterio greedy per ottenere D-cammini furthest reaching a partire da (D-1)-cammini furthest reaching.
Consideriamo due (D-1)-cammini furthest reaching sulle diagonali *k*-1 e *k*+1 terminanti, rispettivamente, nei vertici (x_1,y_1) e (x_2,y_2) del grafo di edit.
Consideriamo il further reaching tra questi due cammini, cioè quello che termina nel punto con indice di riga più basso, percorriamo un arco verticale o orizzontale in base al (D-1)-cammino scelto in modo da portarci sulla diagonale *k*, quindi seguiamo archi diagonali finché troviamo corrispondenze nel grafo o superiamo i limiti della tabella su cui il grafo è costruito (sia come riga che come colonna). In quest'ultimo caso siamo arrivati con un cammino furthest reaching alla locazione *c*[*m*,*n*].
Per applicare questo criterio occorre quindi procedere in maniera sistematica per valori crescenti di D. Inoltre, per il *Lemma 2*, esistono D+1 diagonali in cui può terminare un D-cammino, quindi la procedura per la ricerca dei D-cammini furthest reaching va effettuata per tutte le possibili D+1 diagonali in cui un (D-1)-cammino furthest reaching può terminare.
La procedura seguita dall'algoritmo di Miller e Myers, che è anche uno schema per il terzo approccio, è la seguente:

```
BEGIN
for D := 0 to M+N do
  for k := -D to D in steps of 2 do
       Trova il punto terminale di un D-cammino furthest
       reaching sulla diagonale k;
       if (N,M) è il punto terminale then
            /* D è la distanza di edit. */
            /* Stampa lunghezza LCS. */
            return (m + n – D)/2;
            stop
END
```

La procedura calcola il più piccolo D per cui esiste un D-cammino furthest reaching che termina in (*m*,*n*), ovvero il cammino da (0,0) a (*m*,*n*) con il minor numero di

archi non diagonali, cioè D. Questo numero, come osservato in precedenza, corrisponde alla distanza di edit tra X e Y. Poiché $D \leq m + n$, il cammino ottimale deve essere trovato prima che si concluda il ciclo più esterno.
Le diverse implementazioni di questo approccio greedy si differenziano nel modo in cui viene calcolato il punto terminale. L'algoritmo di Miller e Myers utilizza un array V di interi, di dimensione 2*(m + n), contenente gli indici di colonna dei punti terminali dei D-cammini furthest reaching nelle locazioni V[-D], V[-D+2], …, V[D-2], V[D]. Per il *Lemma 1* l'insieme di queste locazioni è disgiunto da quello delle locazioni utilizzate per il calcolo dei punti terminali dei (D+1)-cammini furthest reaching, cioè V[-D-1], V[-D+1], …, V[D-3], V[D-1]. Ciò consente di memorizzare contemporaneamente i punti terminali dei D-cammini furthest reaching e dei (D+1)-cammini furthest reaching calcolati a partire da essi. L'indice di riga relativo ad un punto terminale sulla diagonale k può essere calcolato a partire da j = V[k] come $j - k$.

MILLER_MYERS(X,Y)

```
BEGIN
Constant MAX := m + n;
integer array V [-MAX : MAX];
V[1] := 0;
for D := 0 to MAX do
   for k := -D to D in steps of 2 do
        if k = -D  OR  (k≠D  AND  V[k-1] < V[k+1]) then
             /*Il D-cammino further reaching è quello che
             termina sulla diagonale k+1, mi sposto verso
             il basso. */
             j := V[k+1];
        else
             /*Il D-cammino further reaching è quello che
             termina sulla diagonale k-1, mi sposto a
             destra. */
             j := V[k-1]+1;
        i := j – k;
```

```
        /* Percorro lo snake più lungo possible. */
        while i < m  AND  j < n  AND  X_{i+1}=Y_{j+1} do
             i := i+1;
             j := j+1;
        /* Memorizzo il punto terminale del D-cammino
        furthest reaching sulla diagonale k, */
        V[k] := j;
        if i ≥ m AND j ≥ n then
             /* D è la distanza di edit. */
             return (m + n – D)/2;
             stop
END
```

Con l'istruzione V[1] = 0, l'algoritmo definisce inizialmente un punto terminale fittizio (-1,0) come punto di partenza per il calcolo del punto terminale dello 0-cammino furthest reaching.

L'algoritmo di Miller e Myers ha complessità O((*m*+*n*)*D). Il doppio ciclo **for**, infatti, viene eseguito al più (*m*+*n*)*(D+1) volte. Tutte le operazioni definite all'interno richiedono tempo costante, al parte il ciclo **while**. Quest'ultimo può essere eseguito più volte in una iterazione del ciclo **for** interno, però complessivamente viene eseguito una volta sola per ogni diagonale attraversata nell'estensione di cammini furthest reaching: poiché tutti i D-cammini scorrono nella regione compresa tra le diagonali –D e D e ci sono al più (2D+1)*min(*m*,*n*) punti all'interno di questa, al più O((*m*+*n*)*D) diagonali vengono attraversate. Quindi il ciclo **while** viene eseguito complessivamente O((*m*+*n*)*D) volte. La complessità spaziale è lineare dal momento che viene utilizzato un array anziché una matrice, a differenza dei precedenti algoritmi.

L'algoritmo quindi, è molto efficiente nel caso in cui le due sequenze sono molto simili, essendo D molto basso (e *r* alto) e, al contrario, poco efficiente nel caso in cui le sequenze sono molto diverse, essendo D molto alto (e *r* basso). L'algoritmo di Miller e Myers, quindi, ha un comportamento opposto in funzione degli input assegnati rispetto a quelli di Hunt e Szymanski e di Hirschberg.

7. Risultati sperimentali

I tempi di esecuzione degli algoritmi per il problema LCS presentati nei capitoli precedenti dipendono da diversi fattori: la lunghezza delle sequenze di input (ovvio) e il loro grado di similarità, valutabile in funzione della lunghezza di una LCS, che influiscono soprattutto sulla complessità computazionale dell'algoritmo, e la dimensione dell'alfabeto di input, che influisce soprattutto sulla complessità spaziale.
In questo capitolo, quindi, vengono illustrati i tempi di esecuzione dei tre algoritmi al variare dei parametri discussi in precedenza in secondi.
I valori misurati dipendono ovviamente dalle caratteristiche hardware della macchina su cui girano i programmi, cioè frequenza del processore, capacità della RAM, ecc.., ma sono indicativi delle prestazioni degli algoritmi. Tutti gli esperimenti sono stati condotti su una macchina Intel Centrino Duo, frequenza di processore 1.60 Ghz, sistema operativo Windows XP, memoria RAM da 2 GB. Gli algoritmi sono stati implementati in linguaggio Java, utilizzando il compilatore javac incluso nel pacchetto Java 5.
I test sono stati effettuati con sequenze di input molto lunghe (di lunghezza n pari a 50000 caratteri). In ogni caso di prova, una delle due sequenze viene generata dall'altra alterando un certo numero di caratteri in maniera casuale: in questo modo si ottiene una LCS di lunghezza r desiderata (molto bassa, media o molto alta).
Sia Σ l'alfabeto di input e σ il numero di simboli contenuti in esso.
Abbiamo considerato i seguenti scenari, abbastanza frequenti:

1) Confronto tra sequenze di DNA per stabilire il grado di similarità ($\sigma = 4$)
2) Confronto tra stringhe alfanumeriche, cioè contenenti solo lettere e numeri ($\sigma = 62$)
3) Confronto tra file di testo, con caratteri dai sistemi ASCII e UNICODE ($\sigma = 220$)

Per ciascuno scenario, ad eccezione del primo, abbiamo considerato 3 casi: sequenze di input quasi identiche (r molto alto, caso frequente nel confronto tra

DNA), sequenze un po' diverse ($r \sim n/2$, caso frequente nel confronto tra file di testo) e sequenze molto diverse con r molto basso.
Per il primo scenario, dal momento che l'alfabeto σ è molto piccolo, si considerano solo i casi r alto e $r \sim n/2$.
Le seguenti tabelle riassumono i risultati dei test condotti. Per ogni scenario e ogni caso sono state effettuate 10 prove ed è stato calcolato un valore medio. In grassetto sono evidenziati caso per caso i risultati migliori.

	Hunt-Szymanski	Hirschberg	Miller-Myers
r ∼ n	550.44 s	1212.13 s	**0.44 s**
r ∼ n/2	244.75 s	353.75 s	**37.47 s**

Scenario 1: σ = 4 (confronto tra sequenze DNA)

	Hunt-Szymanski	Hirschberg	Miller-Myers
r ∼ n	59.38 s	149.5 s	**0.35 s**
r ∼ n/2	**38.09 s**	60.15 s	**38.01 s**
r = O(1)	**6.95 s**	11.87 s	113.85 s

Scenario 2: σ = 62 (confronto tra sequenze alfanumeriche)

	Hunt-Szymanski	Hirschberg	Miller-Myers
r ∼ n	24.76 s	69.28 s	**0.33 s**
r ∼ n/2	**15.42 s**	28.60 s	37.26 s
r = O(1)	**0.47 s**	2.19 s	114.21 s

Scenario 3: σ = 220 (confronto tra file di testo)

Dall'analisi dei risultati, si deduce che:

- Il tempo di esecuzione dell'algoritmo di Miller e Myers è indipendente dal numero σ di caratteri dell'alfabeto di input e dipende solo da *r*, mentre gli altri due algoritmi si comportano molto meglio all'aumentare di σ. Ciò dipende dal fatto che, aumentando σ, pur essendoci più liste di corrispondenze differenti (la procedura di costruzione dell'array matchlist richiede più tempo), ogni lista è più breve e si scorre più rapidamente, quindi diminuisce il tempo necessario per il calcolo dell'array MinYPrefix nell'algoritmo di Hunt e Szymanski e per il calcolo delle k-corrispondenze dominanti in quello di Hirschberg, che sono le procedure che determinano la complessità dei due algoritmi.

- Il tempo di esecuzione dell'algoritmo di Hunt e Szymanski è in tutti i casi più basso rispetto a quello di Hirschberg, nonostante i due algoritmi abbiano più o meno la stessa complessità computazionale nel caso migliore (O(nlgn) per entrambi) o Hirschberg sia migliore nel caso peggiore ($O(n^2 \lg n)$ per il primo e $O(n^2)$ per il secondo). Ciò è dovuto allo spazio di memoria superiore utilizzato dall'algoritmo di Hirschberg per mantenere, in particolare, la matrice D, dipendente dal parametro *r*.

- Nessuno dei tre algoritmi si rivela particolarmente efficiente nel caso in cui *r* ∼ n/2 e nessuno dei tre algoritmi prevale nettamente sugli altri due in questo caso.

Dal confronto dei tempi nei vari scenari, risulta che per $\sigma = 4$ (confronto di DNA) l'algoritmo di Miller e Myers risulta nettamente il migliore, perché arriva subito alla soluzione utilizzando poco spazio supplementare (solo l'array V). Negli altri due scenari c'è più equilibrio con l'algoritmo di Miller e Myers che si rivela migliore per valori di *r* bassi e l'algoritmo di Hunt Szymanski che si comporta meglio per valori medi e alti di *r*.

Riferimenti bibliografici

[1] **Wagner, R. A. & Fischer, M. J. :** The String-to-String Correction Problem, *Journal of the ACM, Vol. 21, No. 1, January 1975, pp. 168-173*.

[2] **Hunt, J. W. & Szymanski, T. G. :** A Fast Algorithm for Computing Longest Common Subsequences, *Comm. of the ACM, Vol. 20, No. 5, 1977, pp. 350-353*.

[3] **Hirschberg, D. S. :** Algorithms for the Longest Common Subsequence Problem, *Journal of the ACM, Vol. 24, No. 4, 1977, pp. 664-675*.

[4] **Miller, W. & Myers, E. W. :** A File Comparison Program, *Softw. Pract. Exp., Vol. 15, No. 1, pp. 18-31*.

[5] **Bergroth, L. & Hakonen, H. & Raita, T. :** A Survey of Longest Common Subsequence Algorithm, *SPIRE, A Coruña, Spain, 2000, pp. 39-48*.

INDICE