FABRIZIO LUCCIO

INFORMATICA PER LE BIOTECNOLOGIE

ALGORITMICA

Seconda Parte: Algoritmi orientati a problemi biologici

7 CONFRONTO TRA SEQUENZE

Il paradigma della programmazione dinamica introdotto nella dispensa 6 è alla base della risoluzione di molti problemi di **confronto fra sequenze** che si incontrano nella elaborazione di testi e nella biologia molecolare. Iniziamo con lo studio del problema della *edit distance* (distanza di edizione, in una versione in italiano scarsamente usata): lo schema algoritmico potrà poi essere facilmente modificato per risolvere altri problemi sulle sequenze.

EDIT DISTANCE. Date due sequenze di caratteri X,Y trovare un *allineamento ottimo* delle due che corrisponda alla *minima distanza* tra X e Y, ove la distanza è definita secondo le regole seguenti:

• si ammette che nelle due sequenze possano essere inseriti spazi indicati con - e che, per ogni posizione nelle due sequenze, il carattere o lo spazio che appare nella X sia posto in corrispondenza (allineato) con il carattere o lo spazio nella stessa posizione della Y;

• la distanza è la somma delle distanze tra coppie di caratteri o spazi, uno in X e l'altro in Y, in posizioni corrispondenti nell'allineamento: caratteri uguali hanno distanza zero, indicata con 0 (**match**); caratteri diversi hanno distanza 1 (**mismatch**); un carattere allineato a uno spazio ha distanza 1 (**space**);

• la presenza di uno spazio si interpreta come l'avvenuta **cancellazione** di un carattere dalla sequenza, o come l'**inserzione** di un carattere nell'altra in posizione corrispondente: per questo motivo non si considerano spazi in posizioni corrispondenti di X,Y;

• la **edit distance** tra X e Y è la distanza relativa all'allineamento (o agli allineamenti) che minimizza tale distanza.

Valga come esempio la distanza tra le sequenze alfabetiche X = ALBERO e Y = LABBRO. Quattro allineamenti con distanza uguale a 3 (che come vedremo è la minima possibile, cioè la edit distance) sono riportati qui sotto. Il segno • indica le coppie nell'allineamento che danno luogo a distanza locale 1; per tutte le altre coppie si verifica un match e la distanza locale è 0.

```
A L B E R O    A L - B E R O    - A L B E R O    - A L B E R O
L A B B R O    - L A B B R O    L A B B - R O    L A - B B R O
 • •   •        •   •   •        •   •   •        •   •   •
```

Indicate le sequenze come $X = x_1\, x_2 \ldots x_n$, $Y = y_1\, y_2 \ldots y_m$, il calcolo impiega una matrice M di dimensioni $(n+1)\times(m+1)$ ove **M[i,j] indica la edit distance tra il prefisso $x_1\, x_2 \ldots x_i$ di X e il prefisso $y_1\, y_2 \ldots y_j$ di Y**. Dunque i caratteri di X corrispondono alle righe di M, i caratteri di Y alle colonne, e l'ultimo valore M[n,m] indica la edit distance tra le due sequenze. La riga e la colonna 0 corrispondono a prefissi vuoti, cioè X e Y non sono ancora stati esaminati, quindi la M si inizializza sulla riga 0 e la colonna 0 ponendo:

$M[0,j]=j$ per $0\le j\le m$, $M[i,0]=i$ per $0\le i\le n$,

valori che corrispondono ad affermare che il prefisso vuoto di X allineato con il prefisso $y_1\, y_2 \ldots y_j$ di Y corrisponde a una distanza j, e che il prefisso vuoto di Y allineato con il prefisso $x_1\, x_2 \ldots x_i$ di X corrisponde a una distanza i. Per le sequenze viste sopra il valore M[0,3] = 3 corrisponde all'allineamento del prefisso LAB di LABBRO con tre spazi --- inseriti prima di ALBERO.

Indicata con p(i,j) la distanza locale tra x_i e y_j, cioè posto **$p(i,j)=0$ se $x_i=y_j$ (match), $p(i,j)=1$ se $x_i\neq y_j$ (mismatch)** secondo i valori indicati sopra, gli elementi M[i,j] con $1\le i\le n$ e $1\le j\le m$ si calcolano progressivamente, riga per riga, con la formula ricorsiva:

$$\mathbf{M[i,j] = \min\{M[i,j-1]+1,\ M[i-1,j]+1,\ M[i-1,j-1]+p(i,j)\}} \qquad (1)$$

che si spiega notando che l'allineamento ottimo tra i prefissi $x_1\, x_2 \ldots x_i$ e $y_1\, y_2 \ldots y_j$ si può costruire dagli allineamenti ottimi per i prefissi privati dell'ultimo carattere, confrontando $x_1\, x_2 \ldots x_i$ con $y_1\, y_2 \ldots y_{j-1}$-; $x_1\, x_2 \ldots x_{i-1}$- con $y_1\, y_2 \ldots y_j$; e $x_1\, x_2 \ldots x_i$ con $y_1\, y_2 \ldots y_j$; e scegliendo tra essi quello che genera distanza minima. Avremo per il nostro esempio la seguente matrice:

	Y	L	A	B	B	R	O
X	0	1	2	3	4	5	6
A	1	1	1	2	3	4	5
L	2	1	2	2	3	4	5
B	3	2	2	2	2	3	4
E	4	3	3	3	3	3	4
R	5	4	4	4	4	3	4
O	6	5	5	5	5	4	3

L'ultimo valore M[6,6] = 3 indica la edit distance tra le due sequenze, che come già affermato è uguale a tre. Si noti che la relazione (1) è ricorsiva ma l'algoritmo di costruzione di M è iterativo e segue uno schema di programmazione dinamica. Tale algoritmo ha **complessità quadratica** $\Theta(nm)$ perché richiede di costruire una matrice M di dimensioni (n+1)×(m+1), e il valore in ciascuna cella è calcolato in tempo costante, perché i tre valori che appaiono nella formula ricorsiva indicata sopra sono stati già calcolati in passi precedenti e memorizzati in M.

La formulazione dell'algoritmo in forma di un programma che chiameremo ED, fa uso di due vettori X, Y di dimensioni n+1 e m+1 che contengono le sequenze da confrontare a partire dalla cella 1 (la cella 0 non è utilizzata), e costruisce la matrice M secondo lo schema già descritto a parole:

```
ED(X,Y)
// Calcolo della edit distance tra due sequenze contenute nei
vettori X,Y, costruendo la matrice M[0:n,0:m]
   for (i=0,i≤ n,i++) M[i,0]=i;  //inizializza la colonna 0
   for (j=0,j≤ m,j++) M[0,j]=j;  //inizializza la riga 0
   for (i=1,i≤ n,i++)
       for (j=1,j≤ m,j++)
         { if (X[i]==Y[j]) p=0; else p=1;
           M[i,j]=min(M[i-1,j]+1,M[i-1,j-1]+p,M[i,j-1]+1); }
   return M[n,m];
```

Gli allineamenti che danno luogo alla edit distance M[n,m] (M[6,6] nell'esempio) si ricostruiscono risalendo all'indietro nella matrice, dalla casella [n,m] fino alla casella [0,0]. Da M[i,j] si risale a M[i-1,j-1] se questi due valori sono uguali e $x_i=y_j$ (per esempio si risale da M[6,6] a M[5,5] entrambi = 3, che indica un match tra caratteri); oppure si risale al valore di M uguale a M[i,j]-1 tra i tre adiacenti che lo precedono: in questo caso vi possono essere più alternative, corrispondenti ad allineamenti di uguale distanza. Nell'esempio si ricostruiscano i quattro allineamenti ottimi tra ALBERO e LABBRO già indicati in precedenza, partendo da M[6,6]: ci proponiamo di trovarne uno solo perché le soluzioni potrebbero essere moltissime.

Se si richiede di ricostruire un solo allineamento ottimo l'algoritmo richiede tempo $\Theta(n+m)$ per tracciare il percorso all'indietro dalla casella [n,m] alla [0,0]: infatti n+m è il numero massimo di passi che l'algoritmo può compiere sulla matrice (nel caso che questi avvengano sempre in orizzontale o in verticale) e ogni passo richiede un numero costante di confronti.

Un programma per eseguire questo compito, che chiameremo ALLINEA, fa uso come il precedente ED dei due vettori X,Y che

contengono le sequenze da confrontare e della matrice M prodotta da ED, e costruisce l'allineamento ottimo in due nuovi vettori ALX,ALY che conterranno le due sequenze inclusi gli spazi che devono apparire in tale allineamento. Per esempio, per il secondo allineamento ottimo visto sopra avremo:

```
ALX:    A L - B E R O
ALY:    - L A B B R O
```

I due vettori contengono lo stesso numero di elementi (sette, nell'esempio), ma questo numero non è noto a priori; quindi essi saranno definiti per I due vettori contengono lo stesso numero di elementi (sette, nell'esempio), ma questo numero non è noto a priori; quindi essi saranno definiti per k=n+m elementi che sono certamente sufficienti poiché la massima edit distance possibile si ha tra due sequenze che non hanno alcun carattere in comune, allineando i caratteri della più breve con altrettanti caratteri dell'altra e inserendo poi spazi nelle posizioni successive della prima fino a raggiungere la fine della seconda.

Si noti che i due vettori vengono riempiti a partire dall'ultima casella k (in corrispondenza all'esame di M[n,m]), per posizioni decrescenti e fino a una casella h, $1 \le h \le k$, che conterrà l'inizio delle sequenze allineate: gli elementi nelle caselle 0...h-1 non avranno alcun significato.

ALLINEA(X,Y,M)

// Costruzione dell'allineamento ottimo tra due sequenze contenute nei vettori X,Y utilizzando la matrice M costruita dal programma ED. Il valore di h, restituito dall'algoritmo attraverso la frase **return**, indica da che punto interpretare il contenuto dei vettori ALX, ALY //

```
k=max(n,m); h=k; i=n; j=m;
while ((i>0) or (j>0))
    {if (((i>0) and (j>0))
          and ((M[i,j]==M[i-1,j-1]) and (X[i]==Y[j])
              or ((M[i,j]==M[i-1,j-1]+1) and (X[i]!=Y[j])))
                 {ALX[h]=X[i]; ALY[h]=Y[j]; i=i-1; j=j-1;}
     else {if ((j>0) and (M[i,j]==M[i,j-1]+1))
              {ALX[h]= -; ALY[h]=Y[j]; j=j-1;}
           else if (i>0)  //deve essere M[i,j]==M[i-1,j]+1
                {ALX[h]=X[i]; ALY[h]= -; i=i-1;}
          }
      h=h-1;
     }
```

```
return h;
```

Esercizio 1. Si esamini attentamente il programma ALLINEA per comprendere come funziona e con quale criterio viene scelto l'allineamento tra i tanti possibili. Nell'esempio ALBERO-LABBRO quale allineamento sceglierà ALLINEA?

A titolo di ulteriore esempio sull'uso e sul significato della matrice M relativamente alla edit distance si consideri l'elemento M[3,6]=4 corrispondente al confronto tra ALB e LABBRO cui corrisponde l'allineamento ottimo:

```
- A L B - -
L A B B R O
•   •   •  •
```

Oppure l'elemento M[4,3]=3 corrispondente al confronto tra ALBE e LAB cui corrispondono gli allineamenti ottimi:

```
A L B E     A L B E     - A L B E
- L A B     L A B -     L A - B -
•   • •     • •   •     •   •   •
```

Il calcolo della edit distance può essere immediatamente adattato a risolvere il problema con diversi valori del costo delle distanze tra singoli caratteri. Si tratta unicamente di modificare la relazione di ricorrenza (1) in funzione dei nuovi costi e inserire questi costi negli algoritmi ED e ALLINEA.

Esercizio 2. Si calcoli nuovamente la distanza tra ALBERO e LABBRO e il loro allineamento ottimo ponendo $p(i,j)=2$ per $x_i \neq y_j$ e lasciando invariati gli altri costi.

Un'altra importante variante dell'algoritmo, che citiamo qui senza presentarla esplicitamente, consente di abbassarne la complessità evitando di calcolare e memorizzare valori di M troppo grandi per concorrere alla determinazione della edit distance. Per comprendere dove questi valori possono apparire nella matrice si consideri anzitutto che **il valore della minima distanza è limitato tra |n-m| e max(n,m).** Infatti ogni percorso tra M[0,0] e M[n,m] deve includere almeno |n-m| passi orizzontali o verticali, il che determina il limite inferiore; e un tale percorso con numero minimo di passi deve includere min(n,m) passi diagonali in aggiunta a quelli detti, includendone in tutto |n-m|+min(n,m) = max(n,m), il che determina il limite superiore immaginando che tutti questi passi abbiano distanza locale 1 (ovvero che si verifichino tutti mismatch nei passi diagonali).

Si consideri il caso più semplice con n = m, come nell'esempio visto sopra ove n = m = 6 e quindi la edit distance è compresa tra 0 e 6. Nella matrice M non possono concorrere al calcolo della edit

distance tutti gli elementi M[i,j] il cui valore, sommato alla distanza orizzontale o verticale dalla diagonale della matrice, supera n: infatti qualsiasi percorso da tali M[i,j] a M[n,m] determinerebbe una distanza che eccede il limite superiore n. Nell'esempio tra ALBRERO e LABBRO sono elementi con valore 4 e 5 nelle zone superiore destra e inferiore sinistra della matrice M. Più precisamente gli elementi M[i,j] a distanza $h > n/2$ possono essere ignorati. Tutti i valori utili sono quindi compresi in una banda di larghezza n+1 (per n pari) o n (per n dispari) attorno alla diagonale della matrice. Per $n \neq m$ la definizione della banda è più complicata.

Il programma ED può essere modificato per escludere il calcolo degli elementi fuori della banda, con qualche attenzione se $n \neq m$. Questa modifica è importante se si decide di accettare solo allineamenti di massimo valore prefissato k costante: in questo caso l'algoritmo esegue un numero di operazioni proporzionale al numero di elementi kn della banda e la sua complessità da quadratica diviene lineare.

Come già detto l'algoritmo di base per il calcolo di distanza può essere applicato alla risoluzione di altri problemi di confronto tra sequenze con semplicissime trasformazioni.
Uno di questi problemi, di grande importanza sia per gli editori di testo che per la biologia molecolare, è il seguente:

RICERCA APPROSSIMATA DI UN PATTERN IN UN TESTO. Date una sequenza Y di m caratteri detta **testo** e una sequenza X di n caratteri detta **pattern**, con $n << m$, trovare tutte le apparizioni di X in Y ammettendo che queste apparizioni possano contenere qualche differenza e quindi la distanza tra la X e la sottosequenza di Y con cui si confronta la X possa non essere zero.

A tale scopo si può impiegare l'algoritmo di edit distance, notando però che **M[i,j] dovrà ora indicare la edit distance tra il prefisso $x_1 x_2 \ldots x_i$ di X e le sottosequeze di Y che terminano in posizione j**, cioè le sottosequenze $y_k y_{k+1} \ldots y_j$ **con** $1 \leq k \leq j$, ove sia i prefissi di X che le sottosequenze di Y possono contenere degli spazi e il valore di k non è noto a priori. Per ottenere questo risultato occorre unicamente inizializzare la riga zero con tutti 0, cioè porre:

$M[0,j]=0, \quad$ per $0 \leq j \leq m$

che corrisponde ad assegnare edit distance zero all'allineamento tra il prefisso vuoto della X e qualunque prefisso della Y, perché il pattern può apparire in qualsiasi punto del testo.

Chiariamo questi punti con un esempio. La seguente matrice per la ricerca delle apparizioni di X = RAT in Y = SERRATURA.

	Y	S	E	R	R	A	T	U	R	A
X	0	0	0	0	0	0	0	0	0	0
R	1	1	1	0	0	1	1	1	0	1
A	2	2	2	1	1	0	1	2	1	0
T	3	3	3	2	2	1	0	1	2	1

I valori rilevanti son quelli nell'ultima riga che indicano la minima edit distance tra la X e le sottosequenze di Y che terminano in quella posizione. Tra questi si sceglieranno i più bassi, ovvero quelli con un valore inferiore a un limite prefissato. Nell'esempio il valore M[3,6]=0 corrisponde all'apparizione esatta di X=RAT nelle posizioni 4,5,6 di Y=SERRATURA. Il valore M[3,4]=2 corrisponde al migliore allineamento tra RAT e le sottosequenze di Y che terminano in posizione 4, cioè ai tre allineamenti:

```
R A T     R A T     R A T
R R -     R - R     R - -
  • •       • •       • •
```

Per determinare gli allineamenti tra la X e le sottosequenze di Y si procede come nel caso precedente, risalendo dalla posizione considerata nell'ultima riga fino alla riga 0, ma senza dover poi recedere fino alla posizione [0,0]. Per esempio l'ultimo elemento M[3,9]=1 corrisponde all'allineamento di RAT con RA- (il calcolo di M[3,9] è stato eseguito scendendo in verticale dalla casella M[2,9]). Si noti che tra questi allineamenti alcuni potrebbero contenere spazi nella X, anche se ciò non si verifica in questo esempio.

Un problema rilevante in biologia molecolare e legato molto strettamente a quello della edit distance è noto come:

GLOBAL COMPARISON. Date due sequenze X,Y trovare un allineamento di **massima similarità**, ove la similarità è definita secondo le regole seguenti:

• come in precedenza, nelle due sequenze possano essere inseriti degli spazi e nell'allineamento non si considerano spazi in posizioni corrispondenti di X,Y;

• il peso di un allineamento è la somma dei pesi tra coppie di caratteri, o tra caratteri e spazi, in posizioni corrispondenti nell'allineamento, ove caratteri uguali (**match**) hanno peso positivo +1, caratteri diversi (**mismatch**) hanno peso negativo -1; un carattere allineato a uno spazio (**space**) ha peso negativo -1;

questi pesi possono essere variati senza modificare la struttura dell'algoritmo allo scopo di assegnare rilevanza diversa ai tre tipi di differenze (mismatch, spazio in X, spazio in Y), o pesare diversamente il valore del match;

• la **similarità** tra X e Y è il peso relativo all'allineamento (o agli allineamenti) che massimizza tale peso.

Dunque in questo problema si attribuisce un credito ai match e un debito alle differenze e il valore di similarità è tanto più alto quanto più i match prevalgono sulle differenze. Mentre la edit distance tra due sequenze è funzione solo delle loro differenze, la similarità dipende in modo rilevante anche dal numero di match. Così coppie di sequenze con le stesse differenze puntuali avranno uguale edit distance ma similarità tanto più alta quanto più sono lunghe poiché coincideranno in tutti gli altri caratteri.

Come in precedenza si impiega l'algoritmo di base sulla matrice M ove M[i,j] indica ora la similarità tra il prefisso $x_1\, x_2 \ldots x_i$ di X e il prefisso $y_1\, y_2 \ldots y_j$ di Y contenenti eventualmente degli spazi, quindi l'ultimo valore M[n,m] indica la similarità tra le due sequenze. La riga e la colonna 0 della M corrispondenti a prefissi vuoti di X e Y si inizializzano con valori negativi relativi alle sequenze di spazi ponendo:

$M[0,j]=-j$ per $0 \le j \le m$, $M[i,0]=-i$ per $0 \le i \le n$.

Indicato con p(i,j) il peso locale tra x_i e y_j, cioè posto $p(i,j)=+1$ se $x_i=y_j$ e $p(i,j)=-1$ se $x_i \neq y_j$, gli elementi M[i,j] si calcolano con la formula ricorsiva:

$$\mathbf{M[i,j] = \max\{M[i,j-1]-1,\ M[i-1,j]-1,\ M[i-1,j-1]+p(i,j)\}} \qquad (2)$$

il cui significato dovrebbe essere ormai chiaro. Ovvia è la trasformazione della formula per pesi diversi da +1 e -1. Utilizzando l'esempio precedente con X = ALBERO e Y = LABBRO otterremo la matrice:

	Y	L	A	B	B	R	O
X	0	-1	-2	-3	-4	-5	-6
A	-1	-1	0	-1	-2	-3	-4
L	-2	0	-1	-1	-2	-3	-4
B	-3	-1	-1	0	0	-1	-2
E	-4	-2	-2	-1	-1	-1	-2
R	-5	-3	-3	-2	-2	0	-1

```
O  -6  -4  -4  -3  -3  -1  +1
```

L'ultimo valore M[6,6] = +1 indica la similarità tra le due sequenze è uguale a +1.

La formulazione dell'algoritmo in forma di un programma è un'ovvia variazione del programma ED. Gli allineamenti che danno luogo alla similarità tra X e Y si ricostruiscono risalendo dalla cella M[n,m] alla M[0,0] come per la edit distance. Nell'esempio tre allineamenti hanno similarità +1: li indichiamo con i costi puntuali associati:

```
 A  L  -  B  E  R  O     -  A  L  B  E  R  O     -  A  L  B  E  R  O
 -  L  A  B  B  R  O     L  A  B  B  -  R  O     L  A  -  B  B  R  O
-1 +1 -1 +1 -1 +1 +1    -1 +1 -1 +1 -1 +1 +1    -1 +1 -1 +1 -1 +1 +1
```

Si noti che l'allineamento

```
A L B E R O
L A B B R O
```

che aveva la sessa edit distance dei primi due ha ora similarità 0 e non si ricava dalla matrice perché nella cella M[6,6] dà conto di allineamenti migliori.

Esercizio 3. Riempire manualmente la matrice M per determinare la similarità tra due sequenze a scelta di lunghezza almeno dieci costruite su un alfabeto di quattro caratteri, con pesi +1 match, -1 mismatch, -2 carattere contro spazio.

La similarità definita nel problema di global comparison può essere impiegata anche nel problema, rilevante in biologia molecolare, della ricerca approssimata di un pattern in un testo in genere molto più lungo. Il problema diviene:

PATTERN COMPARISON. Ricerca con **massima similarità** di una sottosequenza X in una sequenza Y.

Come per la ricerca di un pattern in un testo dovremo inizializzare con 0 tutti gli elementi nella prima riga di M, cioè porre:

$M[0,j]=0$, per $0 \le j \le m$; $M[i,0]=-i$ per $0 \le i \le n$.

Il calcolo di M[i,j] si esegue con l'espressione (2) introdotta per la global comparison. Il valore di massima similarità si trova nell'ultima riga. Il percorso sulla matrice M per determinare l'allineamento termina nella riga zero.

Consideriamo l'esempio X = G A T, Y = C A G A G T A T. Otterremo la matrice:

	Y	C	A	G	A	G	T	A	T
X	0	0	0	0	0	0	0	0	0
G	-1	-1	-1	+1	0	+1	0	-1	-1
A	-2	-2	0	0	+2	+1	0	+1	0
T	-3	-3	-1	-1	+1	+1	+2	+1	+2

La similarità massima +2 appare nelle celle M[3,6] e M[3,8] e corrisponde agli allineamenti:

```
M[3,6]:  Y  . . G A G T . .
         X      G A - T     con percorso che termina in M[0,2]

M[3,8]:  Y  . . G T A T
         X      G - A T     con percorso che termina in M[0,4].
```

Un altro problema rilevante in biologia molecolare che impiega la definizione di similarità ed è in certo modo un'estensione del problema di global comparison, è noto come:

LOCAL COMPARISON. Date due sequenze X,Y trovare una sottosequenza di X e una sottosequenza di Y che hanno un allineamento di **massima similarità.**

Questo problema potrebbe sembrare molto più difficile del precedente perché non si conoscono a priori le sottosequenze di X e Y candidate alla soluzione, che diverranno note solo come risultato dell'algoritmo di confronto. Tuttavia il problema è risolubile con il consueto impiego della matrice M con qualche semplicissima variazione.

Dette S_X e S_y due sottosequenze di X e Y da confrontare per stabilirne la similarità, notiamo che dovranno essere confrontate a partire dal loro inizio tralasciando tutti i caratteri che le precedono in X e Y. Questo proprietà è stata già incontrata nella ricerca di un patttern in un testo in cui però solo i caratteri iniziali del testo andavano trascurati. Estendendo il criterio lì visto dovremo ora inizializzare con 0 tutti gli elementi nella prima riga e nella prima colonna di M, cioè porre:

$M[0,j]=0$, per $0 \leq j \leq m$; $M[i,0]=0$, per $0 \leq i \leq n$.

Nel caso presente il valore contenuto nella cella M[i,j] deve indicare la massima similarità tra due sottosequenze S_X,S_y che terminano in i,j, cioè $S_X = x_h \ldots x_i$ e $S_y = y_k \ldots y_j$ per opportuni valori $1 \leq h \leq i$, $1 \leq k \leq j$(ovvero S_X è un suffisso - o tratto finale - del prefisso $x_1\, x_2 \ldots x_i$ di X, e S_y è un suffisso del prefisso $y_1\, y_2 \ldots y_j$ di Y); e come sempre tali sottosequenze possono contenere degli spazi. L'osservazione chiave a questo punto è che se tutte le possibili sottosequenze S_X,S_y di almeno un carattere che terminano in i,j hanno similarità negativa, la similarità massima è zero perché si ha per S_X,S_y entrambe vuote. Il calcolo di M[i,j] si ottiene quindi con la formula (si noti lo zero nel calcolo del massimo):

$$\mathbf{M[i,j] = max\{M[i,j-1]-1,\ M[i-1,j]-1,\ M[i-1,j-1]+p(i,j),\ 0\}} \quad (3)$$

Il calcolo in M si sviluppa come nei casi precedenti impiegando ora la formula (3). **Il risultato potrà ora apparire in qualsiasi cella della matrice**: si sceglieranno le celle M[i,j] contenenti i valori più alti, che indicano la terminazione in quel punto di due sottosequenze con tale similarità. La ricostruzione dell'allineamento, e quindi l'individuazione delle due sottosequenze "simili", avverrà con il solito metodo di tracciamento all'indietro arrestandosi quando si incontra una cella con valore zero; tuttavia in questo caso si possono ottenere anche altre informazioni come ora vedremo.

Consideriamo l'esempio X = CAGTACT e Y = TAGAGTCG. Otterremo la matrice:

	Y	T	A	G	A	G	T	C	G
X	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	+1	0
A	0	0	+1	0	+1	0	0	0	0
G	0	0	0	+2	+1	+2	+1	0	+1
T	0	+1	0	+1	+1	+1	+**3**	+2	+1
A	0	0	+2	+1	+2	+1	+2	+2	+1
C	0	0	+1	+1	+1	+1	+1	**+3**	+2
T	0	+1	0	0	0	0	+2	+2	+2

I punti di massima similarità si incontrano nelle celle M[4,6]=+3 e M[6,7]=+3. Dalla prima si risale all'indietro lungo un solo percorso raggiungendo la cella M[1,3] che contiene zero, individuando così due sottosequenze identiche A G T in X e Y allineate senza spazi. Da M[5,7] si risale fino alla stessa cella M[1,3] in un solo percorso che corrisponde all'allineamento con similarità +3:

S_X = A G T A C
S_y = A G T - C

che comprende le due sottosequenze A G T già individuate in M[4,6]. Naturalmente il metodo individua anche coppie di sottosequenze con similarità minore della massima, come le due coppie di sottosequenze A G e T A relative alle celle M[3,3] e M[5,2], allineate con similarità +2.

Esercizio 4. Ricostruire gli allineamenti tra sottosequenze relativi ai valori positivi di similarità nella matrice qui sopra.

8 IL PROBLEMA DEL FRAGMENT ASSEMBLY

Motivazioni ed esempi elementari

In questo capitolo considereremo esplicitamente tratti di DNA costituiti come noto da sequenze di *basi* di quattro tipi: A (adenina), G (guanina), C (citosina), T (timina), dove A e G appartengono alla famiglia delle *purine*, C e T a quella delle *pirimidine.* In tali sequenze si riconosce un orientamento dovuto alla struttura della molecola di DNA, sicché possiamo rappresentarle e esaminarle con direzione da sinistra a destra. L'intera molecola di DNA è costituita da due **filamenti** (strands) accoppiati tra loro e che, con un'opportuna interpretazione, recano la stessa informazione: ci riferiamo dunque alla sequenza che rappresenta uno di questi filamenti.

Dal punto di vista algoritmico tutto quello che studieremo potrebbe essere riferito a sequenze costruite su un alfabeto arbitrario e afferenti ad ambienti diversi. Tuttavia la motivazione del capitolo è indicare alcune linee di studio che nascono nella risoluzione di un importantissimo problema di biologia molecolare, consistente nella ricostruzione di un'intera sequenza di DNA sulla base della conoscenza di alcune (in effetti moltissime) sue parti dette **frammenti** (fragnments). Il problema nel complesso è detto **sequenziamento**, e la tecnica è basata sulla composizione dei frammenti. Questa tecnica è motivata dal fatto che gli esperimenti che consentono di *leggere* le basi di filamenti di DNA non sono in grado di operare su un'intera sequenza se questa contiene un numero di basi superiore a qualche migliaio. E anche se lo fosse, è tecnicamente impossibile oggi rendere disponibile all'esame un intero filamento di DNA senza che questo si rompa in un grandissimo numero di frammenti. La tecnologia sta compiendo grandissimi passi avanti, ma anche per organismi la cui intera sequenza di DNA è relativamente breve, non è possibile leggere tale sequenza in una sola passata. Abbiamo dunque il problema:

FRAGMENT ASSEMBLY. Dato un insieme di frammenti di una sequenza, tra loro indipendenti (cioè derivanti da parti distinte della sequenza) o parzialmente sovrapponibili (cioè derivanti da parti parzialmente sovrapposte), ricostruire la sequenza originale con il minimo numero di errori.

Nel caso del DNA i frammenti devono essere moltissimi, ma è possibile ottenerli in breve tempo in quanto una molecola di DNA può essere replicata in laboratorio in un numero di copie che **cresce esponenzialmente nel tempo** con una tecnica detta **PCR** (polymerase chain reaction).

Una semplice riflessione preliminare può convincerci che è l'esistenza di frammenti parzialmente sovrapposti che permette la ricostruzione della sequenza complessiva: se infatti questa fosse suddivisa in frammenti tutti separati, come avverrebbe se si partisse da un'unica sua copia, non si avrebbe modo di stabilire in che ordine essi debbano essere assemblati.

Iniziamo a esaminare il problema su un esempio elementare, ovviamente costruito su sequenze brevissime per poterlo discutere. Immaginiamo di avere a disposizione i quattro frammenti:

T C C G A C G A C T A A T C A T C C G A

e di sapere che la sequenza complessiva detta **di consenso** deve contenere circa dieci basi, quindi i frammenti devono essere parzialmente sovrapposti perché complessivamente ne contengono venti. In questo caso esiste un allineamento in cui le parti contengono sotto-sequenze identiche che non danno luogo a errori. Tale allineamento, e la corrispondente sequenza di consenso di nove basi, sono i seguenti:

```
- - T C C G A - -
- - - - C G A C T
A A T C - - - - -
- A T C C G A - -
_________________

A A T C C G A C T
```

Questo è un caso ideale perché gli errori sono in genere inevitabili e scaturiscono sia dagli strumenti di lettura dei frammenti che dalla imprecisa replicazione della sequenza originale nella PCR. Vediamo nel nostro esempio come questi errori possano emergere e condizionare l'assemblaggio.

• Se il secondo frammento fosse C C A C T (C anziché G nella seconda posizione) l'allineamento e la sequenza di consenso sarebbero gli stessi di prima, con un errore di sostituzione sul secondo frammento, nella sesta posizione dell'allineamento dove la presenza di due G e una C induce a scegliere G a maggioranza.

• Se il quarto frammento fosse A T C G A (perdita del terzo o quarto carattere) potremmo costruire l'allineamento:

```
- - T C C G A - -
- - - - C G A C T
A A T C - - - - -
- A T C - G A - -
_________________

A A T C C G A C T
```

che dà luogo alla solita sequenza di consenso con un errore di cancellazione sul quarto frammento, nella quinta posizione dell'allineamento dove la presenza di due C e uno spazio induce a scegliere C a maggioranza.

• Se il secondo frammento fosse C T G A C T (inserzione di T come secondo carattere) potremmo costruire l'allineamento:

```
- - T C C - G A - -
- - - - C T G A C T
A A T C - - - - - -
- A T C C - G A - -
_____________________

A A T C C - G A C T
```

che dà luogo a una sequenza di consenso di dieci caratteri contenente uno spazio, con un errore di cancellazione sul quarto frammento, nella sesta posizione dell'allineamento dove la presenza di due spazi e una T induce a scegliere lo spazio a maggioranza.

Vi sono poi casi più complicati cui accenneremo brevemente senza dare qui ragguagli sui metodi impiegati per affrontarli.

• Nell'insieme dei frammenti generati è possibile che due di essi, provenienti da parti distanti della sequenza originale e parzialmente sovrapponibili, si connettano tra loro dando luogo a un nuovo frammento detto **chimera**. Chiaramente le chimere non sono parte della sequenza cercata e bisogna escluderle durante la costruzione di questa.

• La sequenza originale può avere tratti (approssimativamente) ripetuti due o più volte, detti **repeat**. Per esempio la sequenza può essere del tipo:

$$T_1\ ABC\ T_2\ ABC\ T_3\ ABC\ T_4$$

ove T_1, T_2, T_3, T_4 sono tratti diversi, e ABC è un repeat composto da tre sotto-sequenze A, B, C. Se tra i frammenti sono presenti:

$F_1 = C\ T_2\ A$ che lega il primo e il secondo repeat con T_2

$F_2 = C\ T_3\ A$ che lega il secondo e il terzo repeat con T_3

la sequenza di consenso può risultare indifferentemente $T_1\ R\ T_2\ R\ T_3\ R\ T_4$ oppure $T_1\ R\ T_3\ R\ T_2\ R\ T_4$ di cui solo la prima è corretta.

• Infine una **mancanza di copertura** si verifica quando mancano i frammenti relativi a un tratto della sequenza originale. Ovviamente in questo caso si ricostruiscono solo porzioni disgiunte della sequenza, come è evidente nel risultato. L'unico rimedio è ripetere

la frammentazione su un maggior numero di copie della sequenza, tenendo presente che i costi di maggiore duplicazione con la PCR, di sequenziamento in laboratorio, e di successivo calcolo dell'assemblaggio nel computer, crescono col numero di frammenti; ma che, d'altra parte, oltre il completamento della copertura, migliora l'affidabilità dell'intera sequenza di consenso.

L'algoritmo di base

Come già detto, l'assemblaggio di sequenze è oggi il problema più importante di algoritmica rivolta alla biologia molecolare. Un'impressionante quantità di risorse di calcolo è dedicata alla sua soluzione su sequenze genomiche di moltissimi organismi incluso l'uomo. Questi studi hanno consentito di ricavare un'enorme mole di dati genomici conservati in diversi Data Base di libero accesso. Una caratteristica molto positiva di questi è che i dati sono presentati in forme sostanzialmente compatibili tra loro rendendone agevole e rapido l'accesso e la fruizione.

La discussione del paragrafo precedente ha mostrato come ricostruire una sequenza da un grandissimo numero di frammenti non può essere un compito semplice data l'assenza di ogni informazione sulle zone della sequenza originale cui i frammenti si riferiscano, la parziale sovrapposizione tra loro e l'inevitabile presenza di errori. Si ricordi in proposito che il progetto per la ricostruzione del genoma umano richiese molti anni e uno sforzo considerevole da parte di qualificatissimi centri che disponevano di grandi risorse; e se da allora le apparecchiature sia di laboratorio che di calcolo hanno fatto enormi progressi, sequenziare un genoma non ancora noto è un compito considerevole.

In queste note dobbiamo limitarci a presentare un algoritmo di base, cuore dell'intero processo: si tratta di un'estensione degli algoritmi di GLOBAL COMPARISON e LOCAL COMPARISON riportati nella dispensa 7, anch'essi di grande rilevanza nella biologia, che impiegano il concetto di similarità. A scopo di presentazione adotteremo di nuovo i pesi +1, -1, e -1 rispettivamente per match, mismatch e carattere-spazio rimandando al prossimo paragrafo per qualche considerazione in merito. Poniamo:

SUFFIX-PREFIX COMPARISON. Date due sequenze X,Y trovare un suffisso S_X di X e un prefisso P_y di Y che presentino la **massima similarità.**

Dovrebbe essere immediatamente chiaro come questo problema intervenga nel Fragment Assembly, perché due frammenti si compongono in un unico frammento più grande sovrapponendo una parte finale (suffisso) di uno a una parte iniziale (prefisso) del secondo. Nel caso presente non si conosce a priori la lunghezza delle sotto-sequenze S_X e P_y cercate, che diverranno note solo come risultato dell'algoritmo. In particolare non si conosce il punto

dove inizierà S_X in X né il numero di caratteri che conterrà la zona di sovrapposizione perché vi potranno essere spazi inseriti in S_X e P_y. Ancora una volta impiegheremo la programmazione dinamica operando sulla matrice M con qualche variazione.

L'inizializzazione della riga e della colonna zero sarà:

$M[0,j]=-j$, per $0 \leq j \leq m$; $M[i,0]=0$, per $0 \leq i \leq n$

ove i valori $M[0,j]=-j$ nella prima riga hanno il consueto significato e i valori $M[i,0]=0$ nella prima colonna sono dovuti al fatto che S_X può iniziare da qualunque posizione di X. La formula ricorsiva che guida il calcolo di M[i,j] sarà la consueta:

$$\mathbf{M[i,j] = max\{M[i,j-1]-1,\ M[i-1,j]-1,\ M[i-1,j-1]+p(i,j)\}}$$

con p(i,j) uguale a +1 (match) o -1 (mismatch), mentre il valore contenuto nella cella M[i,j] indica la massima similarità tra due sotto-sequenze di X, Y, in particolare $x_h \ldots x_i$ di X per un opportuno valore $1 \leq h \leq i$, e $y_1 \ldots y_j$ di Y, che terminano in i,j e che nell'allineamento possono contenere degli spazi.

I risultati dovranno ora essere cercati nell'ultima riga della matrice, ove in una cella M[n,j] l'esame di un suffisso S_X è stato completato paragonandolo al prefisso P_y fino alla colonna j. L'interpretazione di questi risultati è nuova rispetto a quanto visto fin qui, come spiegheremo ora riferendoci al seguente esempio.

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

Come detto, ogni valore M[n,j] nell'ultima riga della matrice corrisponde a uno o più allineamenti di massima similarità tra il prefisso di Y che termina in y_j e un opportuno suffisso di X. Per conoscere l'elemento x_h in cui comincia questo suffisso si deve costruire l'allineamento con l'algoritmo noto sulla matrice M, risalendo in senso inverso nel percorso di calcolo fino a raggiungere la colonna zero nella riga ove il suffisso ha inizio (con eventuali spazi iniziali) e l'algoritmo termina. Nel nostro esempio il valore di similarità M[10,7] = +3 corrisponde ai due allineamenti:

```
Y:     T - A A C G - T G A A C
X: G A T C A A - G C T G

Y:     T - A A - C G T G A A C
X: G A T C A A G C - T G
```

che partono dalla riga 3 dando entrambe luogo a una sovrapposizione di lunghezza nove, tra sette caratteri e due spazi in Y, e otto caratteri e uno spazio in X. Il valore di similarità M[10,5] = +2 corrisponde all'allineamento:

```
Y:     T - A A - C - G T G A A C
X: G A T C A A G C T G
```

che parte dalla riga 3 dando luogo a una sovrapposizione di lunghezza otto, tra cinque caratteri e tre spazi in Y, e otto caratteri in X. Infine il valore di similarità M[10,8] = +2 corrisponde all'inserzione di uno spazio alla fine di S_x accoppiato al carattere A in P_y, scartata perché non vi è alcun vantaggio a protrarre con uno spazio la porzione finale di X allineata a Y.

Una sovrapposizione utilizzabile nel fragment assembly deve avere lunghezza k e similarità s relativamente alte. Il criterio di accettazione, o di scelta tra varie possibilità, è stabilito dal biologo. Nel nostro esempio sono ovviamente preferibili i due primi allineamenti ove le sovrapposizioni hanno maggiore lunghezza e similarità del terzo, ma può essere necessario un confronto tra due sovrapposizioni con valori (k_1,s_1) e (k_2,s_2) tali $k_1>k_2$ e $s_1<s_2$, caso che qui non appare. Deve comunque essere chiaro che negli esperimenti reali i frammenti possono contenere migliaia di caratteri e le sovrapposizioni accettabili tra frammenti devono avere lunghezza fino a centinaia di basi con numero massimo di differenze di qualche unità; e che, per esempio nelle sequenze geniche degli eucarioti, vi sono zone più *conservate* di altre su cui sono tollerabili meno errori.

9 ALTRI PROBLEMI ALGORITMICI

Il problema dei pesi e dei gap

Nel precedente capitolo 8-1 abbiamo adottato i pesi +1, -1, e -1 per match, mismatch e carattere-spazio. Tuttavia, in dipendenza dal problema trattato, questi pesi possono non essere i più significativi e deve essere il biologo a indicare i pesi da scegliere, notando che non sono importanti i loro valori assoluti ma la differenza tra di essi, perché è questa differenza a influenzare il risultato dell'algoritmo di confronto.

Due punti particolarmente rilevanti sono il peso di differenti situazioni di mismatch e il peso associato alla corrispondenza carattere-spazio rispetto al mismatch (che finora abbiamo posto entrambi uguali a -1). Per quanto riguarda i mismatch nelle sequenze genomiche si deve notare che una sostituzione tra purine (A e G) o tra pirimidine (T e C) è in genere più probabile che una sostituzione tra una purina e una pirimidina quindi il peso negativo del mismatch dovrebbe essere maggiore in valore assoluto nel secondo caso. Una regola impiegata frequentemente è assegnare peso +2 al match, peso -1 al mismatch tra purine o tra pirimidine e peso -2 al mismatch tra una purina e una pirimidina, come indicato nella seguente "matrice" P di ovvio significato1. Impiegheremo questi pesi nel seguito, ma ancora una volta sarà il biologo a stabilire di volta in volta i valori da inserire in P.

	A	G	T	C
A	+2	-1	-2	-2
G	-1	+2	-2	-2
T	-2	-2	+2	-1
C	-2	-2	-1	+2

Matrice P dei pesi di match e mismatch tra basi

Il valore di ogni elemento M[i,j] della matrice di programmazione dinamica si calcola come visto nel capitolo 7 a partire M[i-1,j-1], M[i,j-1], M[i-1,j] applicando i nuovi pesi di match e mismatch. Quanto agli spazi la cosa è più complicata per

1 Con una forzatura nel linguaggio tratteremo la P come una matrice scrivendo per esempio P[G,T]=-2, o P[X[i],Y[j]]nel caso sia X[i]=G e Y[j]=T. Nella realtà programmativa le basi sono codificate con numeri interi, per esempio A=0, G=1, T=2, C=3, le sequenza X e Y sono rappresentate in vettori di interi, e questi interi saranno impiegati come indici di riga e colonna nella matrice P.

l'eventuale trattamento di *gap*, ovvero di sequenze di spazi adiacenti in una delle sequenze. Per esempio scelto il peso -1 da dare a uno spazio, e impiegando i pesi di match-mismatch riportati nella matrice P, l'allineamento seguente presenta un gap di lunghezza quattro sulla X e ha similarità -1:

```
Y:   A  T  A  A  T  C  G  A  C
X:   G  T  -  -  -  -  G  T  C

    -1 +2 -1 -1 -1 -1 +2 -2 +2 = -1
```

Tuttavia in caso di gap si può assegnare una penalizzazione minore a ciascuno spazio che segue il primo se la perdita di k caratteri consecutivi è meno significativa di k perdite di un solo carattere in posizioni non contigue. Esistono varie funzioni matematiche per trattare i gap: la più comunemente usata, detta ***costo affine,*** è espressa come

$$c(k) = - a - (k-1)b$$

ove il costo $c(k)$ di un gap di $k \geq 1$ caratteri è dato dal costo ***a*** del primo carattere più il costo $(k-1)$***b*** dei caratteri successivi (se presenti), con $b < a$ costo di ciascuno di essi. Nell'esempio precedente, ponendo $a = -1$ e $b = -0.2$ avremmo costo del gap $-1 -0.6 = -1,6$ anziché -4 e conseguente similarità +1.4 tra X e Y.

L'introduzione del costo affine per i gap introduce una sostanziale variazione negli algoritmi di allineamento, perché è necessario stabilire se uno spazio assegnato in X o Y costituisce il primo carattere di un gap o uno dei successivi. Dunque per calcolare il costo di uno spostamento orizzontale sulla matrice (gap sulla sequenza X) si deve sapere se nella cella di provenienza M[i,j-1] è contenuto un valore determinato da un gap già iniziato, e in questo caso si sottrae *b*, altrimenti si sottrae *a;* una situazione perfettamente simmetrica si ha per uno spostamento verticale (gap sulla sequenza Y). Il metodo più semplice per trattare questo problema richiede una seconda matrice G di dimensioni pari a M ove ogni elemento G[i,j] si valuta dopo M[i,j] secondo le regole:

G[i,j] = 0, se il valore M[i,j] è stato ottenuto unicamente da M[i-1,j-1] (spostamento diagonale in M);

G[i,j] = 1, se il valore M[i,j] è stato ottenuto da M[i,j-1] (spostamento orizzontale in M) ed eventualmente anche da M[i-1,j-1];

G[i,j] = 2, se il valore M[i,j] è stato ottenuto da M[i-1,j] (spostamento verticale in M) ed eventualmente anche da M[i-1,j-1];

G[i,j] = 3, se il valore M[i,j] è stato ottenuto sia da M[i,j-1]

che da M[i-1,j], ed eventualmente anche da M[i-1,j-1].

Si noti che se M[i,j] è stato ottenuto per spostamento orizzontale o verticale, il valore G[i,j] = 1, 2 indica che ci troviamo su un gap con minore penalizzazione in eventuali spostamenti successivi sulla stessa riga o colonna. Nel caso G[i,j] = 3 ci troviamo su due gap orizzontale e verticale. In tutti questi casi possiamo aver raggiunto M[i,j] anche per spostamento diagonale, ma come vedremo nella ricostruzione dell'allineamento seguiremo i gap.

Vediamo a titolo d'esempio come varia l'algoritmo di *pattern comparison* presentato nel capitolo 7 per la ricerca di una sottosequenza X in una sequenza Y, che è un tipico problema biologico rilevante nel contesto presente: gli altri algoritmi visti si modificano in modo simile. Poniamo

X = A A T G T C Y = C A A G G G T T T C A

Come nel problema originale la riga 0 della matrice M è inizializzata a 0 poiché non ha costo un eventuale gap nella sequenza Y all'esterno della X; similmente la riga 0 della matrice G è inizializzata a 0 per indicare assenza di gap. Porremo dunque:

M[0,j]=0, per $0 \le j \le m$; G[0,j]=0, per $0 \le j \le m$.

La colonna 0 di M indica invece il costo di un gap iniziale nella sequenza X a partire dalla riga 1 (valori -1, -1.2, -1.4 ecc); e la colonna 0 della matrice G indica la presenza di un gap su Y. Porremo dunque:

M[i,0]=-0.8 -0.2·i, per $1 \le i \le n$; G[0,j]=2, per $1 \le j \le n$.

Si vedano le inizializzazioni nelle matrici M, G riportate sotto.

Noti i valori di M[i-1,j-1], M[i,j-1], M[i-1,j], G[i,j-1], G[i-1,j], ogni elemento M[i,j] per i>0 e j>0 si calcola come

$$M[i,j] = \mathbf{max}\{v_1, v_2, v_3\} \qquad (1)$$

ove i valori v_1, v_2, v_3 sono ottenuti per match-mismatch, spazio in X e spazio in Y, come segue (nell'esempio porremo a = -1, b = -0.2)

$$\mathbf{v_1} = M[i-1,j-1] + P[X[i],Y[j]]; \qquad (2)$$

```
if (G[i,j-1]==0 or G[i,j-1]==2) v2 = M[i,j-1]+a
                                // inizio gap in X
    else v2 = M[i,j-1]+b;       // estensione gap in X        (3)

if (G[i-1,j]==0 or G[i,j-1]==1) v3 = M[i-1,j]+a
                                // inizio gap in Y
    else v3 = M[i-1,j]+b.       // estensione gap in Y        (4)
```

Calcolato M[i,j], G[i,j] si calcola poi con la formula:

if(v_1 > v_2 **and** v_1 > v_3) G[i,j] = 0 **else**
{**if**(v_2 >= v_1 **and** v_2 > v_3) G[i,j] = 1 **else**
{**if**(v_3 >= v_1 **and** v_3 > v_2) G[i,j] = 2 **else** G[i,j] = 3}}. (5)

ove l'ultima clausola **else** nella relazione (5) corrisponde all'unico caso rimanente $v_2 = v_3 >= v_1$.

Si segua con attenzione la costruzione delle matrici M e G applicando le relazioni da (1) a (5), i pesi nella matrice P e i valori a = -1, b = -0.2 per i gap. Per esempio M[1,1] = -1 si calcola per spostamento verticale (inizio gap) da M[0,1], mentre il mismatch A-C da M[0,0] e lo spostamento orizzontale da M[1,0] generano valore -2 < -1. E si pone anche G[1,1]=2. M[2,4] = +3 si

M	**Y**	C	A	A	G	G	G	T	T	T	C	A
X	0	0	0	0	0	0	0	0	0	0	0	0
A	-1	-1	+2	+2	+1	+0.8	+0.6	+0.4	+0.2	0	-0.8	+2
A	-1.2	-1.2	+1	+4	+3	+2.8	+2.6	+2.4	+2.2	+2	+1.8	+1.6
T	-1.4	-1.4	+0.8	+3	+2	+1.8	+1.6	+4.6	+4.4	+4.2	+4	+3.8
G	-1.6	-1.6	+0.6	+2.8	+5	+4	+3.8	+3.6	+3.4	+3.2	+3	+2.8
T	-1.8	-1.8	+0.4	+2.6	+4	+3	+2.8	+5.8	+5.6	+5.4	+4.4	+4.2
C	-2	+0.2	+0.2	+2.4	+3.8	+2.8	+2.6	+4.8	+4.6	+4.4	+7.4	+6.4

G	**Y**	C	A	A	G	G	G	T	T	T	C	A
X	0	0	0	0	0	0	0	0	0	0	0	0
A	2	2	0	0	1	1	1	1	1	1	1	0
A	2	2	2	0	1	1	1	1	1	1	1	1
T	2	2	2	2	3	3	3	0	1	1	1	1
G	2	2	2	2	0	1	1	3	3	3	3	3
T	2	2	2	2	2	3	2	0	0	0	1	1
C	2	0	2	2	2	3	3	2	2	2	0	1

genera da M[2,3] per spostamento orizzontale, e si ha G[2,4] = 1. M[4,4] = +5 si genera da M[3,3] per match G-G e si ha G[4,4]=0. M[4,8] = +3.4 si genera da M[3,8] (inizio di gap verticale) e da M[4.7] (prosecuzione di gap orizzontale, infatti G[4,7]=3 mostra che era già in corso un gap orizzontale), e si ha G[4,8]=3. La massima similarità tre X e una sottosequenza di Y è +7.4 come si rileva in M[6,10].

Per stabilire un allineamento tra X e Y impiegheremo il consueto tracciamento all'indietro, da M[6,10] fino alla riga 0. Nel caso presente i valori di G[i,j] via via incontrati indicheranno il passo (o i passi) all'indietro da M[i,j] tenendo però conto di una novità. Se M[i,j] è stato generato sia in un gap che in diagonale il tracciamento dovrà procedere sul gap fino al suo inizio, per giustificare i pesi b < a inseriti nell'estensione del gap (se M[i,j] si trova in un gap orizzontale e in uno verticale si sceglierà uno dei due). Per esempio M[4,6] = +3.8 è generato per match e gap orizzontale, e incontrandolo si percorrerà all'indietro il gap fino al suo inizio. Nella seguente copia della matrice M indichiamo in grassetto un tracciamento all'indietro che dà luogo all'allineamento

Y: C A A - G G G T T T C A

X: A A T G - - - - T C

del quale non fanno parte i caratteri iniziale e finale di Y.

M	**Y**	C	A	A	G	G	G	T	T	T	C	A
X	0	**0**	0	0	0	0	0	0	0	0	0	0
A	-1	-1	**+2**	+2	+1	+0.8	+0.6	+0.4	+0.2	0	-0.8	+2
A	-1.2	-1.2	+1	**+4**	+3	+2.8	+2.6	+2.4	+2.2	+2	+1.8	+1.6
T	-1.4	-1.4	+0.8	**+3**	+2	+1.8	+1.6	+4.6	+4.4	+4.2	+4	+3.8
G	-1.6	-1.6	+0.6	+2.8	**+5**	**+4**	**+3.8**	**+3.6**	**+3.4**	+3.2	+3	+2.8
T	-1.8	-1.8	+0.4	+2.6	+4	+3	+2.8	+5.8	+5.6	**+5.4**	+4.4	+4.2
C	-2	+0.2	+0.2	+2.4	+3.8	+2.8	+2.6	+4.8	+4.6	+4.4	**+7.4**	+6.4

Individuazione di un gene in una sequenza di DNA

Un altro problema assai importante riguarda la ricerca di un gene, di cui si è determinata la sequenza S, all'interno di una sequenza già nota T del DNA di un intero organismo. Scopo dell'operazione è per esempio stabilire se la S compare (approssimativamente) come sotto-sequenza U di T, o controllare quali mutazioni presenta la S rispetto alla U. Questo è il problema di pattern comparison già trattato nella dispensa 7, cui ora si applicheranno pesi significativi per il problema biologico con eventuale trattamento dei gap.

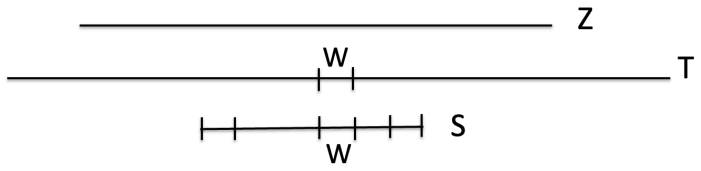
Però, se per esempio la S ha una lunghezza |S| di migliaia di basi perché rappresenta il gene di un eucariote che contiene esoni e introni, e la T ha lunghezza |T| di centinaia di milioni di basi perché è la sequenza di un cromosoma umano, l'algoritmo quadratico di programmazione dinamica richiede un numero di operazioni di ordine $|S| \cdot |T|$ che è elevatissimo anche se polinomiale. Il meccanismo di ricerca si può organizzare allora in un modo differente che spiegheremo senza presentare l'algoritmo relativo che è piuttosto complicato.

Come sappiamo il risultato che S è contenuta in T viene accettato se il numero di errori e nell'allineamento ottimo non supera un limite prefissato, poniamo $e = 49$ errori in totale. Se un tale allineamento esiste, dividendo S in $e + 1 = 50$ tratti consecutivi di $|S|/50$ basi ciascuno, almeno uno di questi tratti deve essere privo di errori: si noti che ne esiste uno solo nel caso in cui gli errori siano ugualmente distanziati, altrimenti possono esistere molti tratti senza errori.

Nel ricco mondo degli algoritmi classici su sequenze ne esiste uno detto **KMP** (dalle iniziali di Knuth, Morris e Pratt che lo hanno proposto) che permette di determinare la presenza **esatta** (cioè senza errori) di un pattern S in un testo T in tempo di ordine $|T|+|S|$, lineare nella dimensione dell'input. Per tornare al nostro esempio questo ordine è |T|, perché |T| è molto maggiore di |S|. Affrontiamo dunque il problema applicando 50 volte l'algoritmo KMP per ricercare senza errori in T ognuno dei tratti di S. Due risultati sono possibili:

1) nessuno di questi tratti appare in T, dunque $ non appare in T senza superare il numero massimo di errori richiesto;

2) uno di questi tratti, diciamo W, appare in T: dunque la zona Z di T dove dovrebbe trovarsi la S si sviluppa attorno a W e ha lunghezza di ordine |S| (può essere maggiore per la presenza di basi in T che non appartengono a S e danno luogo a spazi in S nell'allineamento). Si confrontano quindi S e Z con l'algoritmo di programmazione dinamica.



Il tempo richiesto dal caso 1) è di ordine $e|T|$ per la ricerca degli $e+1$ tratti in T. Il tempo richiesto dal caso 2) è anch'esso di ordine $e|T|$ per la stessa ricerca (anche se questa si ferma appena si trova W) più il tempo di ordine $|S|^2$ richiesto dall'algoritmo di pattern comparison applicato a due sequenze di lunghezza di ordine $|S|$; complessivamente il tempo rimane di ordine $e|T|$ se $|T|>|S|^2$ come nel nostro esempio. L'applicazione diretta dell'algoritmo di pattern comparison applicato a S e T avrebbe invece richiesto un tempo $|S|\bullet|T| >> e|T|$ (tipicamente 100 volte superiore nel nostro esempio).

Determinazione della sequenza di DNA di un organismo

La tecnica di confronto approssimato attraverso la ricerca esatta di un pattern in un testo è anche utilizzata, con metodo praticamente identico a quanto appena visto, quando si tratta di sequenziare l'intero DNA di un organismo di cui già si conosce il genoma di riferimento memorizzato in una base di dati: per esempio sequenziare il genoma **G** di un uomo particolare, che avrà circa 1 milione di basi sconcordanti con quelli del genoma umano **R** di riferimento sequenziato a suo tempo (questo è l'ordine di grandezza medio del numero di differenze: 1 Mega su un totale di oltre 3 Giga basi, cioè circa 0.3 millesimi).

Partendo da un insieme di frammenti di G ottenuti come per il suo sequenziamento, anziché assemblarli come visto sopra si cercano questi frammenti in R col metodo detto sopra per stabilire in che zona di G si dovranno collocare e quindi con quali altri frammenti collegarli. Il procedimento è naturalmente molto più complicato di così, ma l'idea permette di capire come sequenziare un genoma ex novo con questo metodo richieda un numero di operazioni estremamente inferiore a quelle richieste da un completo fragment assembly.

Qualche considerazione conclusiva.

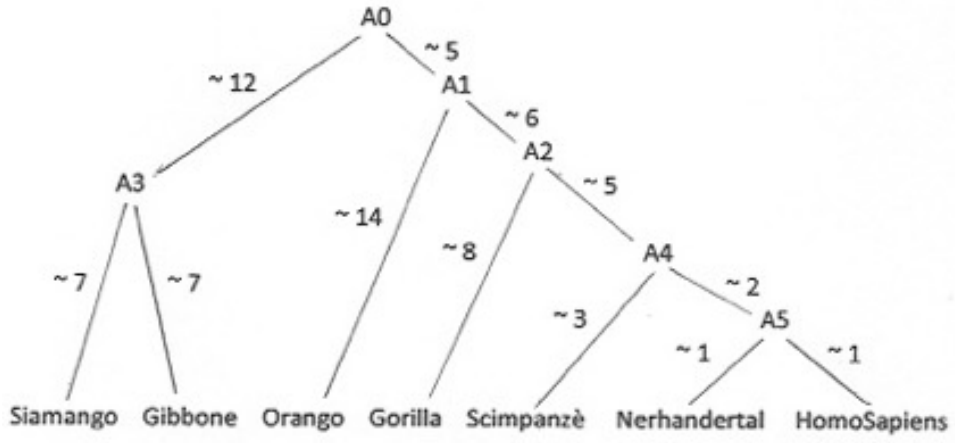
Quanto discusso sopra ci permette di fare una considerazione generale sulle proprietà degli algoritmi, a commento di quanto esposto in tutte queste dispense. L'apparizione esatta di un pattern in un testo può essere determinata sia con l'algoritmo KMP, sia con quello di programmazione dinamica per l'apparizione approssimata: infatti in questo caso le apparizioni esatte del

pattern corrispondono alle celle della matrice che riportano un errore zero. Il primo algoritmo è molto più complicato ma molto più efficiente del secondo, e mostra che il problema è (in gergo) *molto facile* perché ha complessità lineare, che è anche un limite inferiore in quanto è proporzionale al tempo necessario a esaminare tutti i dati (quindi KMP è un algoritmo *ottimo*). Nel gergo degli algoritmi la *facilità* di un problema è legata alla sua natura che gli permette di risolverlo in modo efficiente, non alla facilità con cui si scopre e si progetta un algoritmo di soluzione.

Esistono infine molti problemi su sequenze biologiche che appartengono alla classe dei problemi **NP-completi**, per la cui soluzione si deve inevitabilmente ricorrere a algoritmi approssimati. Questi studi non trovano spazio in queste note: per essi rimandiamo a qualsiasi buon teso di Biologia Computazionale.

10 GLI ALBERI DI FILOGENESI

La famiglia degli Ominoidi (Hominoidea) è una sottofamiglia di primati che deriva da un progenitore vissuto circa diciannove milioni di anni (~ 19 M-anni) fa, alcuni dei quali sono riportati nel seguente Albero di Filogenesi:



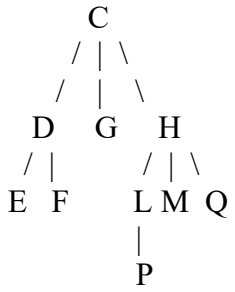
Le etichette da A0 a A5 indicano antenati sconosciuti degli organismi oggi esistenti, da cui discende una mutazione di specie; i numeri associati ai segmenti indicano i M-anni stimati intercorsi tra un ominoide e un suo successore. ~19 M-anni fa A0 dette origine agli ilobatidi A3 e agli ominidi A1; ~7 M-anni fa gli ilobati si divisero in siamanghi e gibboni, e così via.

Questa struttura astratta, in matematica si chiama albero. L'abbiamo già incontrata per la determinazione dei limiti inferiori di complessità e ora ne tratteremo in modo specifico. Un albero è composto di nodi (A0, A1, . . . fino a HomoSapiens) e archi che il congiungono a coppie stabilendo una relazione di padre-figlio (A0 è padre di A3 e A1, eccetera). Questo albero in particolare ha radice A0 e sette foglie (nodi senza figli) Siamango, Gibbone, . . . fino a HomoSapiens, ed è un albero binario perché i suoi nodi hanno al massimo due figli (in questo esempio, zero o due figli). Una porzione dell'albero composta da un nodo diverso dalla radice e tutti i suoi discendenti è detta sottoalbero: per esempio A4 e i suoi quattro discendenti Scimpanzè, A5, Nehandertal e HomoSapiens costituiscono un sottoalbero di A2. Un nodo e i suoi figli costituiscono una famiglia.

In biologia la struttura matematica di albero è legata strettamente agli alberi di filogenesi. Il significato di questi ultimi e le loro proprietà, criteri e metodi di costruzione e di analisi non rientrano in questo testo che si limita a esaminare come una struttura ad albero possa essere rappresentata in un calcolatore ed elaborata in un suo programma, limitandosi all'esame di semplici casi di base. Potrà forse stupire come un disegno apparentemente abbastanza astratto e possibilmente molto esteso possa essere ospitato in una semplice struttura di dati che contiene tutta e sola l'informazione necessaria a descriverla, anche se la comprensione del metodo potrà richiedere qualche attenzione.

Rappresentazione di alberi in memoria

Gli alberi, anche quelli impiegati come strutture matematiche in biologia, non sono necessariamente binari perché un nodo potrebbe diramarsi in un numero arbitrario di figli: ad esempio non è binario l'albero **S** indicato di seguito. Inoltre è possibile che in ogni famiglia abbia senso l'ordine in cui appaiono i figli, rappresentati in modo standard da sinistra a destra: ad esempio D, G, H sono il primo, secondo e terzo figlio di C. Un ordine arbitrario viene imposto comunque tra i figli anche se questo non ha relazione con il problema studiato, perché come vedremo l'ordine è fondamentale per rappresentare l'albero in un calcolatore.



Un albero **S**

Ogni albero di $n \geq 1$ nodi, binario o no, contiene $n - 1$ archi, come si dimostra facilmente notando che può essere costruito partendo dalla sola radice (un nodo e zero archi) e aggiungendovi via via un nodo da connettere alla parte già costruita attraverso un arco che lo connette: la costruzione ultimata conterrà quindi n nodi e n-1 archi. Per rappresentare un albero di n nodi avremo dunque bisogno di specificare almeno $2n-1$ informazioni per rappresentare esplicitamente o implicitamente nodi e archi: vediamo ora un modo in cui questa informazione possa essere organizzata.

Anzitutto possiamo renderci conto che l'intera struttura di un albero può essere riassunta indicando per ogni nodo **z** il suo primo figlio e il suo successivo fratello. Se **z** non ha figli, o non ha fratelli dopo di lui, ciò dovrà essere esplicitamente indicato. Per l'albero **S** assoceremo al nodo D il primo figlio E e il successivo fratello G; assoceremo al nodo M un indicatore di assenza (primo figlio inesistente) e il fratello Q ; assoceremo al nodo F due indicatori di assenza; e così via. Questa informazione esprime esplicitamente i nodi e implicitamente gli archi: per esempio possiamo stabilire che dal nodo H partono tre archi verso i figli L, M, Q perché L è specificato come suo primo figlio, M e Q sono individuati in catena come prossimi fratelli di L e di M. Impiegheremo a questo scopo tre vettori NODO[0:n-1], FIGL[0:n-1], FRAT[0:n-1]. NODO[i] conterrà il nome di un nodo; FIGL[i] e FRAT[i], detti puntatori, conterranno gli indirizzi (nel vettore NODO) del primo figlio e del successivo fratello di NODO[i]. Se uno o entrambi questi nodi non esistono, apparirà il valore -1 in FIGL[i] e FRAT[i]. Ponendo la radice in NODO[0], per l'albero **S** indicato sopra avremo:

indice i	0	1	2	3	4	5	6	7	8	9
NODO	C	D	E	F	G	H	L	P	M	Q
FIGL	1	2	-1	-1	-1	6	7	-1	-1	-1
FRAT	-1	4	3	-1	5	-1	8	-1	9	-1

In genere i nodi possono essere allocati in posizioni arbitrarie del vettore NODO e i puntatori seguono queste posizioni. Si accede all'albero mediante un **puntatore esterno R** che indica la cella in cui è allocata la radice, nel nostro esempio $R = 0$. È importante che il lettore esamini attentamente questo esempio per comprendere tutto quanto seguirà.

Si noti che non tutti i linguaggi di programmazione consentono di utilizzare vettori

contenenti caratteri alfabetici anziché numeri: per esempio in linguaggio C possiamo definire vettori di singoli caratteri ma non di stringhe alfabetiche, come ad esempio sono i nomi dei nodi negli alberi di filogenesi. In questo caso questi nomi (o qualsiasi informazione ad essi associata) si **codificano con numeri** inseriti nel vettore NODO, e una tabella a parte mostra la corrispondenza tra questi numeri e i nomi dei nodi. Vedremo nel seguito un esempio di tale metodo di memorizzazione.

Come si vede l'allocazione in memoria dell'albero è molto compatta e prende spazio $\Theta(n)$ (3n celle per l'esattezza), che coincide in ordine di grandezza con il limite inferiore $\Omega(n)$ perché almeno 2n - 1 informazioni devono essere memorizzate per nodi e archi. Dobbiamo però considerare altri problemi, il primo dei quali riguarda **la rappresentazione degli archi**. Nello schema visto questi sono rappresentati attraverso i puntatori in FIGL e FRAT che però hanno un'implicita direzione da padre a figlio: nel nostro esempio FIGL[0] = 1 indica dove reperire D primo figlio di C, e l'allocazione degli altri figli G e H di C si ricava attraverso una catena di puntatori registrata in FRAT; ma nei tre vettori non vi è alcuna indicazione esplicita di quale sia il padre di D, G, H. Mentre nel disegno di un albero generico gli archi non sono orientati, cioè sono percorribili nei due sensi indicando direttamente che D, G e H sono i figli di C e anche che C è padre di D, G, H, nella rappresentazione a vettori è più complicato recuperare la seconda informazione. Per risolvere efficientemente questo problema, fondamentale negli alberi di filogenesi, si deve aggiungere un ulteriore vettore PADR[0:n-1] di puntatori ai padri. Nel nostro esempio avremo:

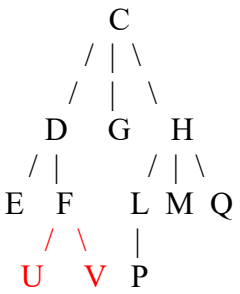
i	0	1	2	3	4	5	6	7	8	9
NODO	C	D	E	F	G	H	L	P	M	Q
PADR	-1	0	1	1	0	0	5	6	5	5

perché i nodi D, G, H hanno padre C che appare in posizione 0, i nodi E, F hanno padre D in posizione 1, e così via. La radice non ha padre quindi PADR[0] = -1. Anche con l'aggiunta di questo nuovo vettore la memorizzazione complessiva prende spazio $\Theta(n)$.

Poiché **anche** gli archi possono avere un'informazione associata, un ulteriore vettore ARCO[0:n-1] potrà codificarla con lo stesso metodo adottato per i padri: nell'albero **S**, per esempio, ARCO[3] = x indicherebbe che l'informazione x è associata all'arco dal nodo NODO[3] = F al padre NODO[1] = D. Questa situazione è propria degli alberi di filogenesi dove l'informazione su un arco è per esempio una misura del tempo in cui è avvenuta la mutazione tra le due specie rappresentate nei nodi, come nell'albero degli ominoidi riportato all'inizio del capitolo. Pur senza soffermarci sull'argomento che deve essere studiato in un testo di biologia, notiamo che attualmente la filogenesi si studia in genere attraverso le mutazioni di sequenze omologhe di DNA: **l'informazione associata a un arco quindi riflette la natura o l'entità del fenomeno che ha causato tale mutazione, o il tempo che ha impiegato ad apparire.**

Lo schema di memorizzazione indicato sopra si adatta in modo naturale a rappresentare trasformazioni della struttura di un albero come inserzione o cancellazione di nodi e archi. A tale scopo i vettori saranno inizialmente definiti con dimensioni [0 : m], con m > n sufficiente a contenere prevedibili incrementi delle dimensioni dell'albero, e si deve prevedere un metodo per gestire le celle dei vettori non utilizzate in ogni momento. Queste vengono legate tra loro mediante il vettore FRAT in una **lista libera** che inizia in un indirizzo indicato da un puntatore **LL** e termina nella cella m con FRAT[m] = -1. Nelle celle della lista libera saranno allocati nuovi nodi e archi, o in essa saranno trasferite le porzioni dei vettori cancellate dall'albero. Indichiamo queste operazioni su una trasformazione dell'albero **S** già visto, che sarà trasformato in **S'** aggiungendo due nodi **U**,

V al nodo F. Le variazioni rispetto a **S** sono indicate in rosso. La lista libera che iniziava in posizione L = 10 ora inizierà in L = 12.



L'albero **S'**

											LL			
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13 ...
NODO	C	D	E	F	G	H	L	P	M	Q	U	V	*	* ...
FIGL	1	2	-1	10	-1	6	7	-1	-1	-1	-1	-1	*	* ...
FRAT	-1	4	3	-1	5	-1	8	-1	9	-1	11	-1	13	14 ...

In caso di cancellazione di parti dell'albero le celle relative saranno incluse all'inizio della lista libera: cambierà il valore di LL e in genere sia le sue celle che quelle che rappresentano l'albero non saranno più consecutive nei vettori come visto finora, senza provocare alcun problema perché saranno i valori contenuti in FIGL e FRAT a stabilire l'ordine di connessione tra le celle. Notiamo anche che l'impiego della lista libera consente in modo immediato anche la **memorizzazione di più alberi** in celle diverse degli stessi vettori, e ciascun albero avrà un suo puntatore d'accesso.

Visita e ricerca

Un'operazione chiave sugli alberi è la visita, cioè la percorrenza di tutta la struttura a partire dalla radice per esaminare uno a uno i suoi nodi (e archi se richiesto). Tra gli scopi della visita vi è quello di cercare se un nodo esiste e quali sono i suoi vicini nel caso non si sappia a priori lungo che archi dirigersi. Si possono definire vari ordini di visita ma noi indicheremo il più utilizzato noto come **preorder**, in italiano ordine anticipato. Si tratta in sostanza dell'ordine dinastico, ovvero di come si percorre l'albero genealogico di una famiglia regnante per stabilire la successione al trono (le nostre nonne sapevano tutto su questo). Immaginiamo che i membri viventi siano rappresentati dall'albero **S'** ove all'inizio la radice C è il re. Immaginiamo che durante il processo di successione non nascano altri eredi e che i re nominati via via muoiano fino a estinzione totale della famiglia (attorno alla quale covano sempre degli assassini). L'odine di successione sarà C D E F U V G H L P M Q : morto il re C gli succede il primo figlio D, poi E. In assenza di figli di E il successore sarà F suo primo fratello. Seguiranno U e V, quindi si dovrà risalire al prozio G, poi nell'ordine a H, L, P, M, Q ove la regola si applica ogni volta alla morte del regnante.

Per progettare un algoritmo di visita in preorder è opportuno osservare che un albero ammette una definizione ricorsiva in cui è implicita l'esistenza degli archi, secondo la quale un albero non vuoto è un insieme Z di uno o più nodi tale che:

1. un nodo di Z è designato come radice;

2. i rimanenti nodi di Z, se esistono, sono ripartiti in successione ordinata in insiemi non vuoti e disgiunti $Z_1, \ldots, Z_m$ ciascuno dei quali a sua volta è un albero ($Z_1, \ldots, Z_m$ sono ovviamente i sottoalberi della radice).

L'algoritmo PREORDER si definisce in modo naturalmente ricorsivo sui vettori in cui l'albero è memorizzato, ammettendo che l'albero sia non vuoto, accedendovi dalla radice e visitando con chiamate ricorsive i sottoalberi nell'ordine opportuno:

PREORDER (i)
// visita in preorder del sottoalbero di radice NODO(i): nella chiamata iniziale si pone i = R puntatore alla radice. Il comando ESAMINA si riferisce a qualsiasi operazione si desideri eseguire sui nodi, per esempio la stampa o il confronto con un nome dato (oltre l'implicita uccisione del re).

```
ESAMINA(NODO(i));
if (FIGL(i) ≠ -1) PREORDER(FIGL(i));
if (FRAT(i) ≠ -1) PREORDER(FRAT(i));
```

Il lettore è invitato a simulare il funzionamento dell'algoritmo sull'albero **S'**. Si comprenderà immediatamente che l'algoritmo richiede tempo Θ(n) ed è quindi ottimo.

Mentre la visita si prefigge di esaminare tutti i nodi dell'albero, l'operazione di ricerca di un nodo di cui si conosca il nome potrà essere eseguita rapidamente disponendo di una tabella costruita una volta per tutte in cui i nomi dei nodi vengono ordinati in tempo Θ(n log n), e per ciascuno di essi è riportato anche l'indirizzo dove il nodo appare nel vettore NODO. La ricerca del nodo si effettuerà nella tabella in tempo Θ(log n) mediante la ricerca binaria, e se il nodo esiste nell'albero l'indirizzo corrispondente permetterà di accedervi in tempo costante e di muoversi poi tra i suoi vicini mediante i vettori FIGL, FRAT e PADR.

Un esercizio

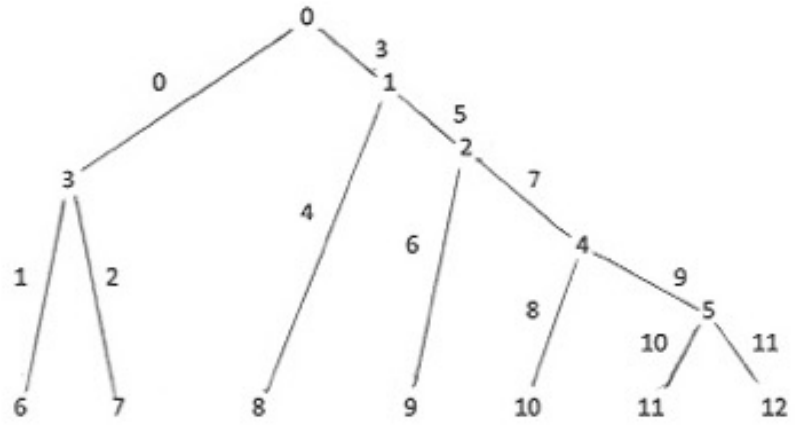
Rappresentiamo ora l'albero degli ominoidi nella struttura dati studiata. A causa della rappresentazione con codici numerici per i nomi dei nodi e i tempi sugli archi, indicati nelle tabelle seguenti, la memorizzazione finale è pesante da seguire. È tuttavia importante che il lettore comprenda il metodo almeno su questo esempio esaminando i numeri contenuti nella tabella finale. Ovviamente tale memorizzazione è ottenuta automaticamente da un programma.

Anzitutto stabiliamo le tabelle dei codici numerici per rappresentare i nomi dei nodi e le informazioni (M-anni) sugli archi.

codice	0	1	2	3	4	5	6	7	8	9	10	11	12
nodo	A0	A1	A2	A3	A4	A5	Siam.	Gib.	Oran.	Gori.	Scimp.	Neh.	Homo

codice	0	1	2	3	4	5	6	7	8	9	10	11
arco	~12	~7	~7	~5	~14	~6	~8	~5	~3	~2	~1	~1

Usando questi codici l'albero può essere ridisegnato come segue:



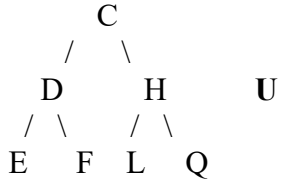
L'albero quindi è memorizzato come segue. Il puntatore è R = 0.

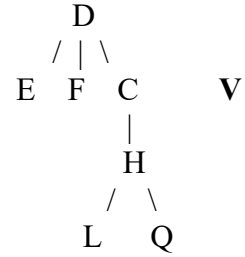
i	0	1	2	3	4	5	6	7	8	9	10	11	12
NODO	0	3	6	1	7	8	2	9	4	10	5	11	12
FIGL	1	2	-1	5	-1	-1	7	-1	9	-1	11	-1	-1
FRAT	-1	3	4	-1	-1	6	-1	8	-1	10	-1	12	-1
PADR	-1	0	1	0	1	3	3	6	6	7	8	10	10
ARCO	-1	0	1	3	2	4	5	6	7	8	9	10	11

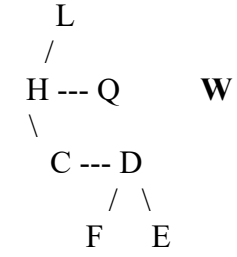
Per esempio nella colonna i = 7 troviamo il nodo NODO(7) = 9 (Gorilla); l'indicazione PADR(7) = 6 della colonna dove si trova suo padre NODO(6) = 2 (A2); l'informazione ARCO(7) = 6 (~ 8 M-anni) associata all'arco da 9 a 2 (cioè la distanza temporale tra Gorilla e suo padre A2); l'assenza FIGL(7) = -1 di figli di Gorilla; l'indicazione FRAT(7)=8 della colonna dove si trova il suo successivo fratello NODO(8) = 4 (A4).

Alberi liberi e filogenie perfette

Gli alberi studiati fin qui sono radicati perché un nodo è designato come radice, e ordinati perché è stato indicato un ordine tra i figli di uno stesso padre. Gli alberi di filogenesi sono idealmente radicati perché la radice indica l'antenato più antico di cui si studiano i discendenti, ma in molti casi la cosa non è così semplice perché non sempre sono chiari i tempi o modi in cui si sono determinate le divisioni. I seguenti alberi **U**, **V** possono rappresentare la stessa discendenza se la specie C è apparsa prima o dopo la D. L'ordine tra i figli invece non è in genere significativo come nelle case regnanti: il primo figlio del re è il suo successore, ma non ha senso chiedersi se Nehandertal abbia precedenza su HomoSapiens.

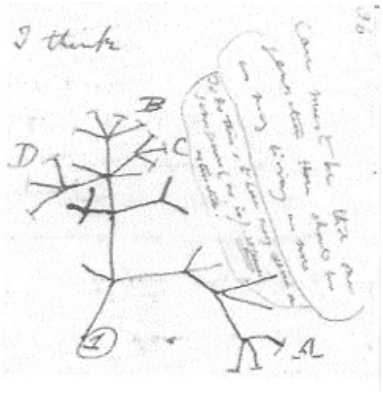






Nel caso più generale un albero non radicato né ordinato si chiama libero, ed è matematicamente definito come un insieme di nodi e archi che li congiungono che sia connesso e privo di cicli, cioè tale che ogni coppia di nodi è connessa da una successione di archi secondo un solo percorso. Permane la proprietà che un albero di n nodi ha n-1 archi e che vi sono foglie definite come nodi su cui incide un solo arco. In questo senso l'albero libero **W** indicato sopra è equivalente a **U** e **V**, e le foglie E, F, L, Q sono le stesse per i tre. La rappresentazione in memoria che abbiamo visto cambia poco tra un albero e l'altro ed è facile modificare quella di **U** per ottenere quella di **V** o viceversa, o per ottenere una rappresentazione di **W** dopo aver scelto per questo una radice e un ordinamento tra figli. In sostanza queste memorizzazioni, o qualunque altra in cui si scelga un nodo diverso come radice e un ordine diverso tra i nodi, possono essere impiegate per memorizzare un albero quando la radice stessa non è certa o l'ordine non è definito.

In molti casi la struttura di un albero di filogenesi può essere determinata solo con un certo grado di approssimazione perché i dati biologici sono incompleti e a volte contraddittori. Anzitutto mentre le foglie rappresentano organismi esistenti non è detto che sia possibile definire quale nodo interno sia il progenitore di tutti gli altri. Gli alberi di filogenesi sono in genere alberi liberi, ed è obbligatorio ricordare che la loro significatività è stata proposta per la prima volta da Charles Darwin che li chiamò "alberi della vita": lo schizzo seguente, e la spiegazione associata, sono contenuti in un suo blocco di appunti datato 1837.

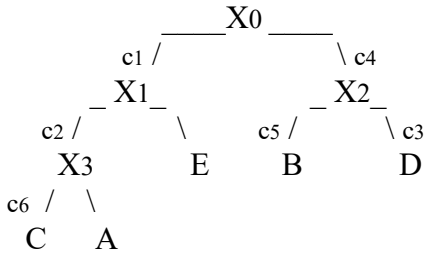


Le ramificazioni fino a qualche decennio fa erano stabilite studiando l'insorgenza o la mutazione di caratteristiche morfologiche mentre oggi si riferiscono principalmente a differenze nelle sequenze geniche o nelle catene di aminoacidi. In genere si parla di differenze tra caratteri, ove questo termine ha un significato generale e può indicare per esempio un tratto di sequenza di DNA o una proprietà corporea. Se queste differenze hanno natura binaria, come presenza-assenza di un carattere o mutazione tra due valori specifici di uno stesso carattere, si può sperare di costruire efficientemente una filogenia perfetta come quella degli ominoidi con il primo progenitore nella radice, gli organismi esistenti nelle foglie e gli archi associati alle mutazioni in modo univoco. Definiamo ora con precisione queste filogenie accennando a come costruirle: nel caso più generale il problema si complica immediatamente fino alla sua NP-completezza.

Per un insieme di n organismi esistenti e di m caratteri considerati, il metodo di base consiste nel costruire una matrice di stato **S** in cui gli organismi sono associati alle righe, i diversi caratteri alle colonne, e ogni cella **S**[i,j] contiene un valore di riferimento (stato) del carattere j nell'organismo i. Per esempio dati cinque organismi A, B, C, D, E e sei caratteri di natura binaria c_1, c_2, c_3, c_4, c_5, c_6 consideriamo la seguente matrice di stato:

	c_1	c_2	c_3	c_4	c_5	c_6
A	1	1	0	0	0	0
B	0	0	0	1	1	0
C	1	1	0	0	0	1
D	0	0	1	1	0	0
E	1	0	0	0	0	0

Immaginiamo che per ogni carattere lo stato 0 indichi il valore più antico e l'1 il valore mutato nel tempo. Il più antico antenato X_0 avrà quindi tutti i caratteri a 0. L'albero associato è il seguente: vicino agli archi sono indicati i caratteri che hanno generato la diramazione.



Per esempio l'organismo C deriva da X_0 attraverso le mutazioni di c_1, c_2 e c_6 come indica la sua riga nella matrice. L'organismo E ha invece subito la sola mutazione c_1 ed è quindi immutato rispetto al padre X_1 che è presente nell'albero perché ha poi subito altre mutazioni mentre E esiste tuttora.

Un fenomeno possibile è l'inversione (reversal) di un carattere, cioè una mutazione tra due stati che si verifica in entrambi i sensi: nella nostra rappresentazione si potrebbero verificare le transizioni $0 \rightarrow 1$ e $1 \rightarrow 0$ nello stesso albero e X_0 non avrebbe necessariamente tutti i caratteri a 0. Poiché le inversioni sono molto rare e rendono la costruzione dell'albero un problema NP-completo (intuitivamente si dovrebbero provare tutte le combinazioni di inversione per tutti i caratteri) assumiamo che non si verifichino. Aggiungiamo che vi possono essere caratteri non binari, che ammettono cioè transizioni tra diversi stati: per esempio una sequenza di DNA può mutare in diversi modi. Anche in questo caso il problema della costruzione dell'albero diventa un problema NP-completo. Proseguiamo quindi ammettendo che non vi siano inversioni e che i caratteri siano binari, e definiamo esattamente la filogenia perfetta e il modo di costruirla: vi sarà sempre un limitato rischio che la soluzione proposta non sia quella giusta.

Definizione. Una filogenia perfetta ammette un albero T tale che per ogni carattere c e per ogni suo stato s l'insieme di tutti i nodi di T per cui c è nello stato s forma un sottoalbero di T.

Nell'esempio visto sopra, per il carattere c_1 e per il suo stato 1 l'insieme dei nodi X_1, X_3, E, C, A per cui c_1 è nello stato 1 forma un sottoalbero, e la stessa proprietà vale per ogni altra coppia carattere-stato. Dunque la filogenia è perfetta e l'albero è l'unico che possa derivare dalla matrice associata. Vediamo ora come questa proprietà possa desumersi dalla matrice. Per ogni colonna c_j indichiamo con U_j l'insieme di righe che contengono 1; per esempio per c_4 abbiamo $U_4 = \{B,D\}$.

Lemma. Una matrice binaria corrisponde a una filogenia perfetta se e solo se per ogni coppia di colonne c_i, c_j gli insiemi U_i , U_j sono disgiunti o uno contiene l'altro.

Non dimostriamo qui il lemma anche se sarebbe semplice farlo. Possiamo però osservare che è soddisfatto per ogni coppia di colonne della matrice vista sopra: per esempio $U_2 = \{A,C\}$ è contenuto in $U_1 = \{A,C,E\}$ ed è disgiunto da $U_4 = \{B,D\}$.

Costruita una matrice di stato con i caratteri osservati in un esperimento il primo problema che si pone è stabilire se essa corrisponde a una filogenia perfetta. Il lemma stesso suggerisce un possibile algoritmo di soluzione consistente nel confrontare a coppie tutte le colonne. Per una matrice $n \times m$ il numero di coppie di colonne è $\Theta(m^2)$ e il confronto tra due colonne richiede tempo $\Theta(n)$, quindi la complessità di questo algoritmo è $\Theta(nm^2)$. Un algoritmo più sofisticato che non riportiamo qui richiede tempo $\Theta(nm)$ ed è quindi ottimo perché concorda col limite inferiore $\Omega(nm)$ del problema che richiede di esaminare tutti gli elementi della matrice.

Stabilito che la filogenia perfetta esiste si deve costruire l'albero. Anche per questo problema esiste un algoritmo di complessità $\Theta(nm)$ che non riportiamo qui.