

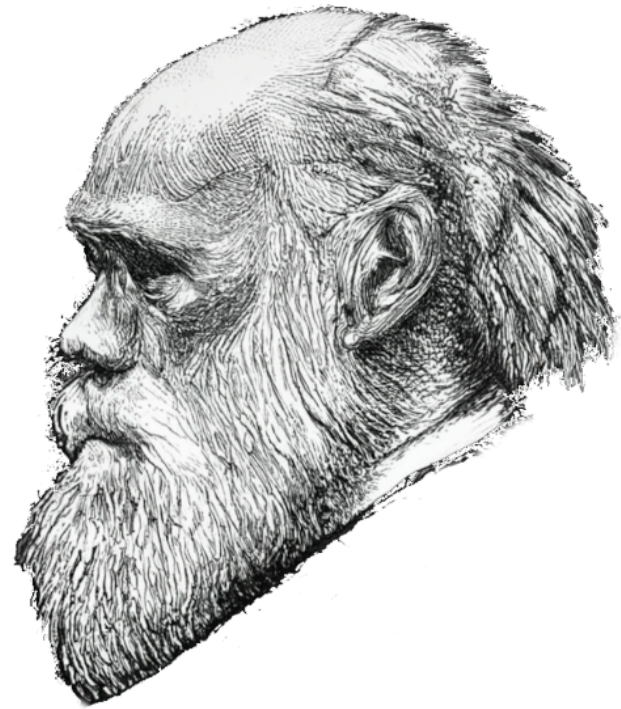
Indice

Elementi di algoritmica	5
Definizioni di base	5
Ordinamento	5
Prestazioni di algoritmi	7
Strutture dati elementari	9
Strutture dati complesse	10
 Algoritmi di pattern matching	 13
Algoritmo naive	13
Algoritmo Shift-And	13
Algoritmo Karp-Rabin	15
 Suffix tree e suffix array	 21
Trie e trie compatto	21
Suffix tree	22
Pattern matching mediante suffix tree	23
Suffix array	24
Longest common prefix	25
Pattern matching mediante suffix array	26
Generalized suffix tree e generalized suffix array	28
Range minimum query	29
Sottostringa comune più lunga	30
 Allineamento di sequenze	 33
Distanza fra due stringhe	33
Allineamento globale di due sequenze	36
 Ricostruzione della filogenesi	 37
Filogenesi ed evoluzione delle specie	37
Problema della filogenesi perfetta	37
Piccolo problema di parsimonia	39
Grande problema di parsimonia	43
Filogenesi mediante clustering	46
Modelli di evoluzione	48
 Sequenziamento del DNA	 51
Assemblaggio del DNA	51
Sequenziamento casuale	51
Sequenziamento mediante ibridazione	53

Genotipi e aplotipi	57
Ricostruzione degli aplotipi mediante pedigree	57
Ricostruzione degli aplotipi per singolo individuo	59



**U think we r
monkeys lol**



Yes

Capitolo 1

Elementi di algoritmica

1.1 Definizioni di base

Si definisce **algoritmo** una procedura di calcolo ben definita che, presi uno o piú valori detti **input**, restituisce uno o piú valori detti **output**. Un algoritmo può anche essere visto come uno strumento per risolvere un **problema computazionale**; la definizione del problema specifica in termini generali la relazione input/output desiderata, e l'algoritmo descrive una specifica procedura computazionale per ottenere tale relazione input/output.

Si definisce **stringa** S una sequenza di caratteri scritti ordinatamente da sinistra a destra. Si denota con $|S|$ la **lunghezza** di una stringa S , ovvero il numero totale di caratteri di cui é composta. Si denota con $S[i]$, o equivalentemente con $S(i)$, il carattere della stringa S che si trova in posizione i . Si assume che gli indici di posizione delle stringhe partano sempre da 1, non da 0¹.

Per ogni stringa S , é detta **sottostringa** la stringa $S[i : j]$ che inizia alla posizione i e termina alla posizione j di S . Ogni sottostringa di S nella forma $S[1 : i]$ é detta **prefisso** di S , e può venire indicata in forma piú compatta con la notazione $S[: i]$. Ogni sottostringa di S nella forma $S[i : |S|]$ é detta **suffisso** di S , e può venire indicata in forma piú compatta con la notazione $S[i :]$.

Date due stringhe S e T , si dice **concatenazione** di S e di T la stringa ottenuta disponendo ordinatamente prima tutti i caratteri di S e poi tutti i caratteri di T . Viene indicata con la notazione $S \cdot T$, o semplicemente con ST .

Una sottostringa $S[i : j]$ é la **stringa vuota**, ovvero la stringa avente lunghezza zero, se i é maggiore di j . Un prefisso, suffisso o sottostringa di una stringa S sono detti rispettivamente **prefisso proprio**, **suffisso proprio** e **sottostringa propria** se hanno una lunghezza inferiore a quella di S e non sono la stringa vuota.

1.2 Ordinamento

Il **problema dell'ordinamento** é un problema molto ricorrente e molto semplice da esprimere: dato un insieme di numeri (per comoditá, si assume interi), si richiede di disporli in ordine crescente.

1.2.1 Insertion sort

Insertion sort é un semplice esempio di algoritmo di ordinamento iterativo. Sia A un array di numeri di lunghezza n , e si supponga che i primi k elementi di A siano già ordinati (k può anche valere 0). Insertion sort prevede di inserire l'elemento di posizione $k + 1$ nella posizione ordinata nel sottoinsieme $0, \dots, k$ ripetendo il procedimento fino a quando $k = n$ (ovvero, tutto l'array é ordinato).

```
values = [...]  
  
for i in range(0, len(values)):  
    key = values[i]                # Inspect the first unsorted element  
    j = i - 1  
    while (j >= 0 and values[j] > key): # As long as a higher value is found,  
        values[j + 1] = values[j]      # keep shifting values right  
        j = j - 1  
    values[j + 1] = key             # Put the value in its proper place  
  
print(values)
```

1.2.2 Selection sort

1. Questa convenzione é in linea con una notazione piú di tipo matematico che di tipo informatico.

```

values = [...]

for i in range (0, len(values)):
    minimum = i
    for j in range (i, len(values)):
        if (values[j] < values[minimum]):
            minimum = j
            temp = values[minimum]
            values[minimum] = values[i]
            values[i] = temp

print(values)

```

1.2.3 Bubble sort

```

values = [...]
isSorted = False

while (not isSorted):
    isSorted = True
    for i in range (0, len(values) - 1):
        if (values[i] > values[i + 1]):
            temp = values[i + 1]
            values[i + 1] = values[i]
            values[i] = temp
            isSorted = False

print(values)

```

1.2.4 Merge sort

Merge sort é un algoritmo ricorsivo costruito mediante una tecnica chiamata **divide et impera**. Questa tecnica prevede, in termini molto generali, nel suddividere i dati in ingresso in sottoinsiemi (*divide*), risolvere ricorsivamente il problema su questi sottoinsiemi e poi ricombinare la soluzione dei sottoinsiemi (*impera*) per ottenere la soluzione globale del problema originario. Nel caso specifico di merge sort, i due passi della tecnica possono essere esplicitati come segue:

<i>Divide</i>	Si divida l'insieme da ordinare in due sottoinsiemi di cardinalità bilanciata e li si ordini ricorsivamente.
<i>Impera</i>	Si fondano due sottoinsiemi ordinati in un unico insieme ordinato.

```

values = [...]

def mergesort(array, left, right):
    if (left < right):
        center = int((left + right) / 2)
        mergesort(array, left, center)
        mergesort(array, center + 1, right)
        merge(array, left, center, right)

def merge(array, left, center, right):
    subLeft = center - left + 1
    subRight = right - center

    A = [0] * subLeft
    B = [0] * subRight

    for a in range (0, subLeft):
        A[a] = array[left + a]

    for b in range (0, subRight):
        B[b] = array[center + b + 1]

    i = j = 0
    current = left

    while (i < subLeft and j < subRight):
        if (A[i] <= B[j]):
            array[current] = A[i]
            i = i + 1
        else:
            array[current] = B[j]
            j = j + 1
        current = current + 1

    while (i < subLeft):
        array[current] = A[i]
        i = i + 1
        current = current + 1

    while (j < subRight):
        array[current] = B[j]
        j = j + 1
        current = current + 1

mergesort(values, 0, len(values) - 1)

```

1.2.5 Counting sort

Counting sort é un altro esempio di algoritmo iterativo: sia A un array di numeri di lunghezza n , e siano \min e \max i valori rispettivamente piú piccolo e piú grande che compaiono in A . Sia poi k l'ampiezza dell'intervallo dei numeri che compaiono in A , ottenuta dalla differenza tra \max e \min . Si costruisca un array B di lunghezza k , dove andranno riportate le occorrenze di ciascun valore che compare in A . La mappatura tra A e B viene realizzata in questo modo: la prima cella di B contiene il numero di volte che \min compare in A , la seconda cella di B contiene il numero di volte che il successore di \min compare in A , la terza cella di B contiene il numero di volte che il successore del successore di \min compare in A , e cosí via fino ad esaurire tutti i valori che compaiono in A . É possibile ottenere l'array ordinato semplicemente riportando ordinatamente da sinistra a destra quante volte compare (anche zero volte) ciascun i -esimo valore di A desumendoli da B .

```

values = [...]

lowest = min(values)
highest = max(values)

result = [0] * len(values)
counters = [0] * (highest - lowest + 1)

for i in range(0, len(values)):          # Count the occurrences
    counters[values[i] - lowest] += 1    # of each member

for i in range(1, len(counters)):        # Compute the cumulative
    counters[i] += counters[i - 1]      # frequencies

for t in range(len(values) - 1, 0, -1): # Print each value, properly
    key = values[t] - lowest            # shifted, as many times as
    result[counters[key] - 1] = values[t] # its occurrency
    counters[key] -= 1

print(result)

```

1.3 Prestazioni di algoritmi

Esistendo diversi algoritmi per risolvere uno stesso problema, é utile poter comparare fra loro algoritmi diversi per valutare quale tra questi sia il "migliore". Nella maggior parte dei casi non esiste un criterio univoco per determinare quando un algoritmo sia migliore di un altro, sia perché alcuni aspetti prestazionali di un algoritmo possono essere piú o meno utili a seconda di come lo si voglia implementare, sia perché uno stesso problema é meglio risolto da un algoritmo piuttosto che da un altro in base a quali condizioni lo delimitano.

Un parametro di valutazione molto importante per l'analisi algoritmica é il **tempo di esecuzione**, ovvero il numero di operazioni primitive che l'algoritmo esegue per un input generico. Per semplificarne il calcolo, si preferisce assumere che l'input di un algoritmo avente dimensione n sia sempre proporzionale a $\ln(n)$; questo non é, in genere, uno scenario realistico, ma permette di considerare ogni operazione primitiva di un algoritmo come non dipendente dalla dimensione dell'input. In termini pratici, una singola riga di codice può essere pensata come avente costo dove c_i é una costante indipendente dall'input. Naturalmente, se una primitiva dell'algoritmo avente costo c_i deve venire eseguita k volte, allora il tempo di esecuzione relativo a quella istruzione é $k \cdot c_i$. Il tempo di esecuzione complessivo é dato dalla somma dei tempi di esecuzione parziali delle singole istruzioni.

Si consideri l'algoritmo di ordinamento Insertion sort. L'istruzione 1 viene necessariamente eseguita tante volte quanto vale la dimensione dell'input, ovvero n , perché per ciascun valore dell'array in input é necessario incrementare il contatore i di 1 e valutare se questo sia inferiore alla lunghezza dell'array. Le istruzioni 2, 3 e 7 vengono eseguite tante volte quanto viene eseguita la prima istruzione meno uno, ovvero $n-1$, perché deve venire eseguita per ogni iterazione del ciclo `for` meno l'ultima, in cui viene fatto il confronto e risulta violato.

Studiare l'istruzione 4 é piú complesso, perché il numero di volte in cui viene eseguita non é determinabile a priori, perché dipende dall'input: se l'array é parzialmente ordinato verrà eseguita poche volte, mentre se é poco ordinato verrà eseguita molte volte. Sia allora m_i il numero di volte (quale che sia) che l'istruzione numero 4 viene eseguita per il i -esimo valore; il numero totale di volte in cui viene eseguita l'istruzione 4 sarà allora la somma di tutti i valori di m_i . Il numero di volte in cui vengono eseguite le istruzioni 5 e 6, similmente a quanto é stato detto per le istruzioni 2, 3 e 7, equivale al numero di volte in cui é stata eseguita 4 meno uno.

1) <code>for i in range(0, len(values)):</code>	1) $c_1 \cdot n$	5) $c_5 \cdot \sum_{i=0}^{n-1} (m_i - 1)$
2) <code>temp = values[i]</code>	2) $c_2 \cdot (n-1)$	6) $c_6 \cdot \sum_{i=0}^{n-1} (m_i - 1)$
3) <code>j = i - 1</code>	3) $c_3 \cdot (n-1)$	7) $c_7 \cdot (n-1)$
4) <code>while (j >= 0 and values[j] > temp):</code>	4) $c_4 \cdot \sum_{i=0}^{n-1} m_i$	
5) <code>values[j + 1] = values[j]</code>		
6) <code>j = j - 1</code>		
7) <code>values[j + 1] = temp</code>		

7

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{i=0}^{n-1} m_i + c_5 \cdot \sum_{i=0}^{n-1} (m_i - 1) + c_6 \cdot \sum_{i=0}^{n-1} (m_i - 1) + c_7 \cdot (n-1)$$

Naturalmente, l'input passato ad un algoritmo influisce su quante volte deve eseguire i suoi passi. Ad esempio, se ad un algoritmo di ordinamento viene passato come input un array già parzialmente ordinato sarà necessario eseguire meno istruzioni rispetto al passarvi un array con i valori disposti in maniera completamente casuale. Per questo motivo si preferisce non valutare il tempo di esecuzione su un input generico, ma piuttosto in due casi: il **caso migliore** (*best case*) ed il caso peggiore (*worst case*). Il primo é il caso che si verifica quando all'algoritmo viene passato un input per il quale deve eseguire le sue istruzioni il minimo numero di volte, mentre il secondo é il caso che si verifica quando all'algoritmo viene passato un input per il quale deve eseguire le sue istruzioni il massimo numero di volte. Individuare i due casi non é una operazione scontata, ed algoritmi diversi hanno casi migliori/peggiori diversi.

Per quanto riguarda Insertion sort, il caso migliore si ha quando i valori passati in input sono già di per loro ordinati, perché l'istruzione 4 viene eseguita una volta sola per ciascun ciclo e le istruzioni 5 e 6 (che dipendono dalla 4) non vengono mai eseguite. Pertanto, nel caso migliore si ha che m_i vale sempre 1 per ogni valore di i .

$$\begin{aligned} T_{best}(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{i=0}^{n-1} 1 + c_5 \cdot \sum_{i=0}^{n-1} (1-1) + c_6 \cdot \sum_{i=0}^{n-1} (1-1) + c_7 \cdot (n-1) = \\ &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) = \\ &= c_1 \cdot n + c_2 \cdot n - c_2 + c_3 \cdot n - c_3 + c_4 \cdot n - c_4 + c_7 \cdot n - c_7 = \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

Il caso peggiore si ha invece quando i valori passati in input sono ordinati in ordine decrescente, perché l'istruzione 4 deve venire eseguita tante volte quanto vale i , in quanto tutti i valori devono venire fatti scalare in prima posizione. É intuibile come, di conseguenza, le istruzioni 5 e 6 debbano venire eseguite $i-1$ volte.

$$\begin{aligned} \sum_{i=0}^{n-1} i &= \left(\frac{(n-1)(n-1+1)}{2} \right) = \frac{n(n-1)}{2} \sum_{i=0}^{n-1} (i-1) = \left(\frac{(n-1)(n-1+1)}{2} - (n-1) \right) = \frac{(n-2)(n-1)}{2} \\ T_{worst}(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{i=0}^{n-1} i + c_5 \cdot \sum_{i=0}^{n-1} (i-1) + c_6 \cdot \sum_{i=0}^{n-1} (i-1) + c_7 \cdot (n-1) = \\ &= c_1 \cdot n + c_2 \cdot n - c_2 + c_3 \cdot n - c_3 + \frac{c_4}{2} \cdot n(n-1) + \frac{c_5}{2} \cdot (n-1)(n-2) + \frac{c_6}{2} \cdot (n-1)(n-2) + c_7 \cdot n - c_7 = \\ &= \left(\frac{1}{2}c_4 + \frac{1}{2}c_5 + \frac{1}{2}c_6 \right) n^2 + \left(c_1 + c_2 + c_3 - \frac{1}{2}c_4 - \frac{3}{2}c_5 - \frac{3}{2}c_6 + c_7 \right) n - (c_2 + c_3 - c_5 - c_6 + c_7) \end{aligned}$$

Si noti come le istruzioni 1, 2, 3 e 7 dipendano esclusivamente dalla dimensione dell'input, pertanto vengono eseguite comunque a prescindere che si stia trattando il caso migliore o peggiore.

Analizzare il tempo di esecuzione di algoritmi ricorsivi é molto piú complesso. Algoritmi ricorsivi possono essere analizzati mediante una **relazione di ricorrenza**: il tempo richiesto da una routine é uguale al tempo speso all'interno della routine piú il tempo speso per le chiamate ricorsive. Questa relazione di ricorrenza é una equazione dove la variabile tempo compare sia a sinistra che a destra.

Un possibile approccio é il cosiddetto **metodo dell'iterazione**, che prevede di "sciogliere" la ricorsione per esprimerla sotto forma di una sommatoria dipendente solo da n . Applicare questo metodo non é sempre facile, perché richiede di utilizzare manipolazioni algebriche non scontate.

$$T\left(\frac{n}{2}\right) = \begin{cases} c + T\left(\frac{n}{4}\right) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases} \quad T\left(\frac{n}{2}\right) = c + T\left(\frac{n}{4}\right) = 2c + T\left(\frac{n}{8}\right) = 3c + T\left(\frac{n}{16}\right) = \dots = i \cdot c + T\left(\frac{n}{2^i}\right)$$

Il procedimento va iterato finché non si raggiunge il passo base. Questo accade quando $n / 2^i = 1$, ovvero quando $i = \log_2(n)$. Ponendo $i = \log_2(n)$ é possibile concludere $T(n) = c \log_2(n) + T(1) \in O(\log(n))$.

Se la relazione di ricorrenza é forma presentata di seguito, é possibile ricavare il tempo di esecuzione mediante un teorema chiamato **teorema fondamentale delle ricorrenze**:

$$T\left(\frac{n}{b}\right) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases} \quad \begin{aligned} 1. \quad & T(n) = \Theta(n^{\log_b(a)}) \text{ se } f(n) = O(n^{\log_b(a)-\epsilon}) \text{ per } \epsilon > 0; \\ 2. \quad & T(n) = \Theta(n^{\log_b(a)} \log(n)) \text{ se } f(n) = \Theta(n^{\log_b(a)}); \\ 3. \quad & T(n) = \Theta(f(n)), \text{ se } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ per } \epsilon > 0 \text{ e } a \cdot f(n/b) \leq c \cdot f(n) \text{ per } c > 1 \text{ e } n \text{ sufficientemente grande.} \end{aligned}$$

L'espressione del tempo di esecuzione di un algoritmo contiene spesso molte costanti che rendono la funzione difficile da leggere ed inutilmente complicata. Per questo motivo spesso non si è particolarmente interessati a considerare il tempo di esecuzione di per sé, ma bensì una sua delimitazione che ne dia una stima ragionevole. A questo scopo è possibile introdurre la **notazione asintotica**, che permette di approssimare tempi di esecuzione aventi una espressione complessa mediante espressioni più semplici. Data una funzione generica f_n :

•

$$O(f(n)) = \{g(n) : \exists c > 0 \wedge n_0 \geq 0 \text{ t.c. } g(n) \leq c f(n) \forall n \geq n_0\}$$

In altri termini, se $g(n) \in O(f(n))$ allora $g(n)$ cresce più lentamente di $f(n)$

•

$$\Omega(f(n)) = \{g(n) : \exists c > 0 \wedge n_0 \geq 0 \text{ t.c. } g(n) \geq c f(n) \forall n \geq n_0\}$$

In altri termini, se $g(n) \in \Omega(f(n))$ allora $g(n)$ cresce più velocemente di $f(n)$

•

$$\Theta(f(n)) = \{g(n) : \exists c_1, c_2 > 0 \wedge n_0 \geq 0 \text{ t.c. } c_1 f(n) \leq g(n) \leq c_2 f(n) \forall n \geq n_0\}$$

In altri termini, se $g(n) \in \Theta(f(n))$ allora $g(n)$ cresce alla stessa velocità di $f(n)$

È possibile verificare che, per Insertion Sort, $T_{best} \in O(n)$ e $T_{worst} \in O(n^2)$. Ovvero, T_{best} e T_{worst} crescono più rapidamente di, rispettivamente, n e n^2

1.4 Strutture dati elementari

Una **struttura dati** è un modo per organizzare e gestire una collezione di dati. L'implementazione di una struttura dati richiede in genere di costruire degli algoritmi che la gestiscono e la manipolano: cercare (se esiste) uno specifico dato, inserire (se possibile) un dato, eliminare un dato, eccetera. Viceversa, diversi algoritmi si appoggiano a delle strutture dati per le loro computazioni.

1.4.1 Array

Un **array** è una **struttura indicizzata** costituita da una collezione di celle numerate a partire da 0 che possono contenere elementi di un tipo prestabilito. La **dimensione** di un array, ovvero il numero massimo di elementi che può contenere, viene fissata quando l'array viene creata, e non è possibile modificarla.

Il fatto che sia una struttura indicizzata, ovvero dove ogni elemento ha assegnato un numero ordinato, implica che sia possibile raggiungere una certa cella con tempo costante $O(1)$. Allo stesso modo, aggiungere o eliminare un elemento da un array è una operazione immediata (in tempo $O(1)$). Non potendo però modificare direttamente la dimensione di un array, l'unico modo per farlo è mediante **riallocazione**, ovvero creare un secondo array (più grande o più piccolo) e copiarvi tutti i dati del precedente; evidentemente questo procedimento richiede tempo lineare $O(n)$.

1.4.2 Liste concatenate

Una **lista concatenata** è costituita da un insieme di **record**, simili alle celle degli array ma che, anziché essere ordinate sequenzialmente, sono fra loro collegati da dei puntatori. Se i record di una lista concatenata hanno un puntatore al record successivo ma non al precedente, la lista concatenata è detta **semplice**, ed è percorribile in una sola direzione; se invece i record di una lista concatenata hanno un puntatore al record successivo ed un puntatore al record precedente, la lista concatenata è detta **doppia**, ed è percorribile in entrambe le direzioni. Le uniche eccezioni sono l'ultimo record, che non punta a nulla e il primo record, a cui nessun record punta.

Il vantaggio delle liste concatenate è che non hanno una dimensione fissata, pertanto è possibile aggiungere un nuovo record semplicemente collegando il puntatore in uscita dall'ultimo record al puntatore in entrata al nuovo record, in tempo $O(1)$. Si ha però che, non essendo i record indicizzati, non è possibile accedere direttamente ad uno specifico record ma è necessario attraversare tutta la lista a partire dal primo record, e questo richiede un tempo $O(n)$.

1.4.3 Pila

Una **pila** è una struttura dati basata sulla disciplina di accesso **LIFO** (*Last In First Out*), ovvero dove l'elemento di prima posizione è l'ultimo elemento che è stato inserito. Gli inserimenti in una pila (**push**) aggiungono elementi alla fine della sequenza, mentre le cancellazioni (**pop**) ne rimuovono sempre l'ultimo elemento. In una pila, gli accessi avvengono sempre ad una sola estremità, e nessun elemento interno può essere estratto prima che tutti gli altri elementi che lo seguono sono stati estratti. Una pila può essere vista come una lista concatenata semplice della quale è possibile manipolare soltanto una delle due estremità.

1.4.4 Code

Una **coda** é una struttura dati basata sulla disciplina di accesso **FIFO** (*First In First Out*), ovvero dove l'elemento di prima posizione é il primo elemento che é stato inserito. Gli inserimenti in una coda (*enqueue*) aggiungono elementi alla fine della sequenza, mentre le cancellazioni (*dequeue*) ne rimuovono sempre il primo elemento. In una coda, gli accessi possono avvenire ad entrambe le estremitá, e nessun elemento interno può essere estratto prima che tutti gli altri elementi che lo precedono sono stati estratti. Una coda può essere vista come una lista concatenata doppia della quale é possibile manipolare soltanto le due estremitá.

1.4.5 Alberi

Un **albero** é una coppia $T = (N, A)$ costituita da un insieme N di **nodi** e da un insieme $A \subseteq N \times N$ di coppie di nodi, detti **archi**. In un albero, ogni nodo v ha uno ed un solo *genitore* u tale per cui $(u, v) \in A$; l'unica eccezione é il nodo **radice**, il nodo da cui tutti gli altri nodi derivano. Un nodo u può avere zero o piú *figli* v tali che $(u, v) \in A$, ed il loro numero viene chiamato **grado** del nodo. Un nodo che non ha figli é chiamato **foglia**, mentre un nodo che ha sia dei figli che dei genitori é detto **nodo interno**. La **profonditá** di un nodo é il numero di archi che occorre attraversare, a partire dalla radice, per raggiungerlo; l'**altezza** di un albero é la massima profonditá a cui é possibile trovare una foglia. Viene detto **albero n-ario** un albero costituito esclusivamente da nodi (interni) di grado n ; un albero n -ario in cui tutte le foglie sono sullo stesso livello é detto **completo**.

Un albero può essere implementato sia mediante array, sia mediante liste concatenate, anche se la seconda opzione é preferibile. Questo può essere fatto rappresentando ogni nodo dell'albero con un record che contiene l'informazione associata al nodo, piú altri puntatori che consentono di raggiungere i nodi a cui questo é collegato. In particolare, se ci si limita al caso in cui l'albero é composto esclusivamente da nodi aventi grado non superiore a d , con d valore fissato, é possibile costruire l'albero semplicemente mantenendo in ogni nodo un puntatore a ciascuno dei possibili d figli; se un nodo ha meno di d figli, alcuni dei puntatori non punteranno a nulla.

Restituire il contenuto dei nodi di un albero non é una operazione scontata: a differenza delle strutture dati precedenti, in un albero ad ogni elemento può essere associato piú di un elemento ordinatamente successivo, pertanto é possibile che si abbiano delle ramificazioni. L'approccio piú usato viene detto **visita in profonditá**, ed é basato sulla ricorsione: se il nodo non ha figli, se ne restituisce il contenuto, mentre se ha figli la funzione viene ricorsivamente applicata a questi.

```
class Node:

    def __init__(self, value):
        self.sons = []
        self.value = value
        # Create an object for a node,
        # where pointers to its sons
        # are saved in a list

    def printTree(node):
        if (len(self.sons) == 0):
            print(value)
            # If the node has no
            # sons, print its value
        else:
            for i in range (0, len(self.sons)):
                printTree(self.sons[i])
            # Otherwise, apply recursively
            # onto them
```

1.5 Strutture dati complesse

1.5.1 Heap

Uno **heap** é un albero binario che rispetta le seguenti proprietá: é completo almeno fino al penultimo livello, ad ogni nodo é associato un valore univoco, che non può trovarsi in nessun altro nodo, ed il valore contenuto in un nodo é sempre maggiore o uguale a quello contenuto nei suoi figli (se ne ha). Uno heap avente n nodi ha altezza $\Theta(\log(n))$. Dalla definizione di heap si evince che il nodo a cui é associato il valore maggiore é la radice, mentre il nodo a cui é associato il valore minore é uno dei nodi foglia.

Eliminare un nodo foglia é semplice, mentre eliminare un nodo interno potrebbe "rompere" la struttura dello heap. Ciò che é possibile fare con facilitá é invece scambiare i valori contenuti in un nodo foglia qualsiasi con quello contenuto nel nodo da eliminare, e dopodiché eliminare il neonato nodo foglia; questo permette di preservare la struttura dello heap, ma potrebbe invalidare la terza proprietá (che i valori associati ai nodi siano maggiori o uguali dei valori associati ai figli). Perché questo non accada occorre, dopo aver effettuato lo scambio, effettuare ripetuti confronti tra i valori dei nodi, scambiando i valori contenuti nei nodi in modo da "spingere" i valori piú piccoli in basso. Questa procedura viene genericamente chiamata *heapify*. Similmente, per aggiungere un nodo ad uno heap preservandone le proprietá é sufficiente aggiungere il nodo come foglia, scambiarne il valore con la radice e ricostruire la struttura.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

void heapify(unsigned int* theHeap, int curr)
{
    int biggest = curr;
    int l = curr * 2;
    int r = (curr * 2) + 1;
```

```

    if (theHeap[biggest] < theHeap[l] && l < SIZE)
        biggest = l;

    if (theHeap[biggest] < theHeap[r] && r < SIZE)
        biggest = r;

    if (biggest != curr) {
        swap(&theHeap[curr], &theHeap[biggest]);
        heapify(theHeap, biggest);
    }
}

```

1.5.2 Tabella hash

Una **tabella hash** é una struttura dati che costruisce corrispondenze fra delle chiavi e dei valori. Una tabella hash puó essere implementato mediante un array dove ogni indice é associato ad un valore, e a sua volta ogni chiave é associata ad un indice. L'associazione fra indici e chiavi é data da una **funzione di hashing**, una funzione $h : U \rightarrow \{0, \dots, n-1\}$ con n la dimensione dell'array, che manipola una chiave per ricavarne un indice univoco. Una funzione di hashing si dice **perfetta** se é iniettiva, ovvero se non soltanto ad ogni indice é associata una sola chiave ma al contempo ad ogni chiave é associato un solo indice. In altri termini, una funzione di hashing é perfetta se la risultante tabella hash ha abbastanza celle da poter contenere valori associati ad ogni chiave possibile.

La definizione di tabella hash non include una definizione specifica di funzione di hashing, ma una funzione che viene molto spesso utilizzata é nella forma $h(k) = k \bmod n$, dove k é una chiave ed n é la dimensione dell'array che costituisce la tabella. A prescindere da quale funzione di hashing si utilizzi, nella pratica il numero di chiavi possibili é spesso enorme, e di conseguenza avere una funzione di hashing perfetta richiederebbe di avere un array a sua volta enorme. In genere, si preferisce avere una funzione di hashing non perfetta; questo comporta però che questa non sia iniettiva, e quindi vi siano dei valori che la funzione di hashing associa alla stessa chiave. Un approccio molto semplice al problema prevede di associare a ciascuna cella dell'array che costituisce la tabella non un singolo valore, ma bensí una lista concatenata; in questo modo se piú valori competono per uno stesso indice é sufficiente concatenarli.

Capitolo 2

Algoritmi di pattern matching

2.1 Algoritmo naive

Data una stringa P , chiamata *pattern*, ed una stringa T di lunghezza maggiore di P , chiamata *testo*, il **problema dell'appaiamento esatto (exact matching problem)** consiste nel trovare, se ne esistono, tutte le occorrenze di P all'interno di T .

Con $P = aba$ e $T = bbabaxababay$, vi sono tre occorrenze di P all'interno di T , in particolare alle posizioni 3, 7 e 9 di T . Si noti che é ammesso che diverse occorrenze si sovrappongano, come accade alle occorrenze in posizione 7 e 9 di T .

Il metodo di risoluzione piú semplice e intuitivo per risolvere il problema dell'appaiamento esatto é quello che viene chiamato **algoritmo naive (algoritmo ingenuo)**. L'algoritmo naive consiste nell'allineare l'estremo sinistro di P con l'estremo sinistro di T , dopodiché confrontare ordinatamente da sinistra a destra i caratteri di P con quelli di T finché non viene trovata una coppia di caratteri $P[i]$ e $T[i]$ diversi o finché si raggiunge la fine di P . Nel secondo caso, si segnala la presenza di una occorrenza. Il procedimento viene poi ripetuto spostando P di una posizione a destra; l'algoritmo termina quando la posizione dell'estremo destro di P supera quella dell'estremo destro di T .

```
occurrences = 0                                # Number of occurrences
P = "... "                                     # Pattern
T = "... "                                     # Text

for i in range(0, len(T) - len(P) + 1):         # Iterate over Text
    mismatch = False

    for j in range(0, len(P)):                 # Iterate over Pattern, update the
        if (P[j] != T[j + i]):                # variable if a mismatch is found
            mismatch = True
            break

    if (mismatch == False):                    # If an occurrence is found,
        occurrences = occurrences + 1;         # increment the variable

print(occurrences)
```

Siano n la lunghezza della stringa P e m la lunghezza della stringa T . Il caso peggiore si ha quando esattamente ogni carattere di P si trova in T : per un numero n di volte viene valutato (con risposta affermativa) se $P[i]$ e $T[i]$ sono uguali per un numero $m - n + 1$ di volte. Si ha quindi che il tempo nel caso peggiore é $n \times (m - n + 1) \approx \Theta(n \times m)$.

Si noti come questo risultato in termini di tempo di esecuzione sia lungi dall'essere il risultato ottimale. Si evidenzia infatti facilmente che il lower bound per il tempo di esecuzione di un qualsiasi algoritmo che risolve il problema del pattern matching é $O(n + m)$. Questo perché é sempre necessario, al fine di risolvere il problema, leggere almeno una volta tutti i caratteri sia di P che di T .

2.2 Algoritmo Shift-And

L'algoritmo naive é basato sulla comparazione di caratteri. Esistono approcci al problema dell'appaiamento esatto basati su operazioni aritmetiche: fra gli algoritmi di questa famiglia figura l'**algoritmo shift-and**¹.

1. Anche chiamato **algoritmo di Baeza-Yates** o **algoritmo di Gonnet**

Siano P un pattern di lunghezza n e T un testo di lunghezza m . Sia M un array di dimensione n per $m + 1$ di valori binari, con indice i che va da 1 a n e indice j che va da 1 a m . Il valore in $M(i, j)$ é 1 se i primi i caratteri di P coincidono con gli i caratteri di T che terminano al carattere j , altrimenti é 0.

$T = \text{abracadabra}$

$P = \text{abr}$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Questo significa che se si ha $M(n, j) = 1$ allora l'intera stringa P é contenuta in T terminando alla posizione j di T . Pertanto, per conoscere il numero di occorrenze di P in T é sufficiente contare quanti 1 sono presenti sull'ultima riga della matrice M . Essendo però la matrice M composta esclusivamente da valori binari, contare il numero di 1 presenti in una sua riga equivale a sommare fra loro tutti i valori della riga.

Innanzitutto, si costruiscano tanti vettori U_x binari unidimensionali colonna di lunghezza n quante sono le lettere dell'alfabeto. La i -esima cella di U_x contiene il valore 1 se x coincide con l' i -esimo carattere di P , e 0 altrimenti.

$P = \text{abx}$

$$U_a = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$U_b = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$U_c = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

...

Sia $\text{Bit-Shift}(x)$ una operazione sui vettori colonna, che restituisce il vettore x facendo scalare tutti gli elementi di una posizione verso il basso ed inserendo un 1 nella posizione lasciata vuota (la nuova prima posizione). Vedendo x come un vettore riga trasposto, $\text{Bit-Shift}(x)$ si ottiene eseguendo uno shift a destra su x e accodando un 1 a sinistra.

$$\text{Bit-Shift}\left(\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}\right) = \text{Bit-Shift}([0 \ 0 \ 1 \ 0 \ 1])^T = [1 \ 0 \ 0 \ 1 \ 0]^T = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

La matrice M viene costruita colonna per colonna. Il primo passo é inizializzare i valori della prima colonna di M : se $T[1] \neq P[1]$, ovvero se il primo carattere di P ed il primo carattere di T differiscono, la prima colonna di M ha solo elementi 0; se invece $T[1] = P[1]$, allora il primo elemento della prima colonna di M vale 1 e tutti gli altri 0.

Ciascuna colonna restante $j > 1$ é data dall'AND bit-a-bit fra $\text{Bit-Shift}(j-1)$, il vettore che si ottiene dall'applicare Bit-Shift al vettore colonna precedente a j , e $U_{T[j]}$, il vettore U prima costruito relativo al carattere di T alla posizione corrente j .

$$M[:, j] = \text{Bit-Shift}(j-1) \text{ AND } U_{T[j]}$$

$$M[i, j] = \text{Bit-Shift}(j-1)[i] \text{ AND } U_{T[j]}[i]$$

```
T = "..."  
P = "..."  
occurrences = 0  
  
U = [[0] * (len(P)) for _ in range(26)]  
M = [[0] * (len(P)) for _ in range(len(T))]  
  
for i in range(0, len(P)):  
    Temp = ord(P[i]) - 97  
    U[Temp][i] = 1  
  
if (T[0] == P[0]):  
    M[0][0] = 1  
  
for i in range(1, len(T)):  
    for j in range(1, len(P)):  
        M[i][j] = M[i-1][j-1]  
    M[i][0] = 1  
    Temp = ord(T[i]) - 97  
    for j in range(0, len(P)):  
        M[i][j] = U[Temp][j] & M[i][j]  
  
for i in range(len(T)):  
    occurrences = occurrences + M[i][len(P) - 1]  
  
print(occurrences)
```

Per convincersi della correttezza dell'algoritmo é sufficiente osservare che, per ogni $i > 1$, il valore di $M[i, j]$ é 1 se e soltanto se sono contemporaneamente soddisfatte due condizioni: la prima é che i primi $i-1$ caratteri di P devono essere uguali agli $i-1$ caratteri di T che terminano in $j-1$, mentre la seconda é che $P[i]$ deve essere uguale a $T[j]$. La prima condizione é verificata se il valore contenuto in $M[i-1, j-1]$ é 1, mentre la seconda condizione é verificata se il valore in $U_{T[j]}[i]$ é 1. Tuttavia, il valore $M[i-1, j-1]$ é uguale al valore $(\text{Bit-Shift}(j-1)[i])$, pertanto per valutare se entrambe le condizioni sono verificate é sufficiente operare un AND fra $(\text{Bit-Shift}(j-1))$ e $U_{T[j]}$ e osservare quale valore si trova nella cella i . Nonostante il tempo di esecuzione dell'algoritmo Shift-And nel caso peggiore rimanga $\Theta(n \times m)$, l'algoritmo Shift-And é nettamente piú veloce dell'algoritmo naive se n é piccolo (grande quanto una parola). Questo perché ogni colonna di M ed ogni vettore U possono essere istanziati in una unica word, e le operazioni di Bit-Shift e di AND possono essere fatte in una sola istruzione macchina.

2.3 Algoritmo Karp-Rabin

Il metodo Shift-And si basa sull'operare uno shift logico su array di bit. Dato che shiftare un numero di n posizioni equivale a moltiplicarlo n volte per la sua base, é possibile ripensare l'algoritmo Shift-And in termini di operazioni aritmetiche anziché operazioni logiche.

Siano dati un pattern P ed un testo T . Si assuma per il momento che P e T siano stringhe binarie, ovvero esclusivamente composte da 0 e 1. Questo permette di considerare le stringhe non come concatenazioni di caratteri ma come numeri binari. Si indichi con T_r^n la sottostringa di T che inizia alla posizione r ed ha lunghezza n , ovvero $T[r : r + n]$. Siano infine $H(P)$ e $H(T_r^n)$ le rappresentazioni in forma decimale dei numeri binari P e T_r^n , ovvero:

$$H(P) = \sum_{i=1}^n 2^{n-i} \cdot P[i]$$

$$H(T_r^n) = \sum_{i=1}^n 2^{n-i} \cdot T[r + i - 1]$$

Se $P = 0101$, allora $H(P) = 2^3 \times 0 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = 5$.

Se $T = 101101010$, $n = 4$ e $r = 2$ allora $H(T_r^n) = 2^3 \times 0 + 2^2 \times 1 + 2^1 \times 1 + 2^0 \times 0 = 6$.

Ogni numero binario ha una ed una sola rappresentazione in forma decimale. Pertanto, se $H(P) = H(T_r^n)$, allora il numero binario/stringa P ed il numero binario/stringa $H(T_r^n)$ sono uguali, ovvero esiste una occorrenza di P alla posizione r di T . Si ha però anche che ogni numero decimale ha una ed una sola rappresentazione in forma binaria, pertanto se $P = T_r^n$, ovvero esiste una occorrenza di P alla posizione r di T , allora $H(P) = H(T_r^n)$.

Si potrebbe allora pensare di affrontare il problema del pattern matching utilizzando $H(P)$ e $H(T_r^n)$. Il problema di un approccio di questo tipo é che il costo per il calcolo di $H(P)$ e $H(T_r^n)$ cresce troppo velocemente con l'aumentare della lunghezza delle stringhe. Inoltre, se le stringhe non sono binarie ma n -arie, il processo diventa ancora piú oneroso.

L'**algoritmo Karp-Rabin** é un algoritmo che risolve il problema del pattern matching utilizzando $H(P)$ e $H(T_r^n)$. Tuttavia, non sfrutta i due valori per intero, bensí applicandovi l'operazione di modulo per un intero piccolo p . In questo modo, i valori sui quali l'algoritmo opera saranno sempre di dimensione contenuta, ma l'algoritmo diviene *probabilistico*, ovvero il risultato fornito é corretto a meno di un certo errore.

Per un intero positivo p , sia $H_p(P)$ il resto ottenuto dalla divisione intera di $H(P)$ per p , ovvero $H(P) \bmod p$. Analogamente, si definisce $H_p(T_r^n)$ come $H(T_r^n) \bmod p$. I valori $H_p(P)$ e $H_p(T_r^n)$ sono detti **fingerprint** di, rispettivamente, P e T_r^n .

Avendoli definiti come resto di una divisione, si ha che i valori $H_p(P)$ e $H_p(T_r^n)$ sono necessariamente compresi tra 0 e $p-1$. Tuttavia, se per ricavare $H_p(P)$ e $H_p(T_r^n)$ é necessario prima calcolare $H(P)$ e $H(T_r^n)$, allora il problema della dimensione si ripropone. Fortunatamente, é possibile calcolare $H_p(P)$ e $H_p(T_r^n)$ direttamente applicando le proprietà dell'aritmetica modulare:

$$H_p(X) = \{ \{ \dots (\{ \{ X[1] \times 2 \bmod p + X[2] \} \times 2 \bmod p + X[3] \} \times 2 \bmod p + X[4] \} \dots \} \bmod p + X[n] \} \bmod p$$

Siano $P = 101111$ e $p = 7$. Si ha allora che $H_p(P) = 101111 \bmod 7 = 47 \bmod 7 = 5$.

Il valore può anche essere calcolato direttamente a partire da P , come mostrato a fianco.

$$1 \times 2 \bmod 7 + 0 = 2$$

$$2 \times 2 \bmod 7 + 1 = 5$$

$$5 \times 2 \bmod 7 + 1 = 4$$

$$4 \times 2 \bmod 7 + 1 = 2$$

$$2 \times 2 \bmod 7 + 1 = 5$$

$$5 \bmod 7 = 5$$

Calcolare $H_p(P)$ e $H_p(T_r^n)$ in questo modo permette inoltre di operare con numeri di dimensione contenuta anche durante tutti gli step intermedi di computazione. Inoltre, é possibile mostrare che é possibile calcolare $H_p(T_r^n)$ in maniera efficiente a partire da $H_p(T_{r-1}^n)$ con un numero costante di operazioni. Infatti:

$$H(T_r^n) = 2 \times H(T_{r-1}^n) - 2^n T[r-1] + T[r+n-1]$$

E ricordando che $H_p(T_r^n) = H(T_r^n) \bmod p$, si ha:

$$H_p(T_r^n) = [(2 \times H(T_{r-1}^n) \bmod p) - (2^n \bmod p) T[r-1] + T[r+n-1]] \bmod p$$

Se il pattern P si trova in T a partire dalla posizione r , allora $H_p(P) = H_p(T_r^n)$. Questo perché se P si trova in T alla posizione r , ovvero se vale $H(P) = H(T_r^n)$, allora vale anche $H(P) \bmod p = H(T_r^n) \bmod p$. Tuttavia, diversamente da quanto detto per $H(P)$ e $H(T_r^n)$, si perde l'implicazione inversa, ovvero se vale $H_p(P) = H_p(T_r^n)$ non è detto che il pattern P si trovi in T a partire da r .

Quando si verifica una situazione in cui $H_p(P) = H_p(T_r^n)$ ma il pattern P non si trova effettivamente in T alla posizione r , si dice che si ha a che fare con un **falso positivo**. Si noti però che, siccome se il pattern P si trova in T a partire dalla posizione r è garantito che valga $H_p(P) = H_p(T_r^n)$, non potranno mai verificarsi dei **falsi negativi**, ovvero situazioni in cui P si trova in T a partire dalla posizione r ma l'algoritmo non lo registra. Questo significa che l'errore, se esiste, sarà sempre in eccesso, mai in difetto.

A questo punto, si hanno a disposizione tutti gli elementi necessari per implementare l'algoritmo Karp-Rabin. Dati un testo T ed un pattern P , entrambi sull'alfabeto binario:

1. Si scelga un numero p ;
2. Si calcolino $H_p(P)$ e $H_p(T_0^n)$, il valore iniziale di $H_p(T_r^n)$;
3. Per ogni posizione r di T , si calcoli $H_p(T_r^n)$ e si valuti se questo è uguale a $H_p(P)$. Se lo è, significa che è presente una occorrenza, potenzialmente falsa, di P in T alla posizione r .

```
T = "..."  
P = "..."  
pvalue = ...  
occurrences = 0  
  
def initiate(S):  
    H = 0  
    for i in range(0, len(P)):  
        H = H + (int(S[i]) * 2**(len(P) - i - 1))  
    H = H % pvalue  
    return H  
  
Hp = initiate(P)  
Ht = initiate(T)  
  
for i in range(1, len(T) - len(P) + 1):  
    Ht = (((2 * Ht) % pvalue) -  
          (2**(len(P)) % pvalue) *  
          int(T[i - 1]) +  
          int(T[len(P) + i - 1])) % pvalue  
    if (Ht == Hp):  
        occurrences = occurrences + 1  
  
print(occurrences)
```

2.3.1 Ottimizzazione: ricerca di un p efficiente

Una prima ottimizzazione per l'algoritmo Karp-Rabin è individuare un metodo che permetta di trovare dei p in maniera oculata. Nello specifico, occorre individuare dei valori p che siano abbastanza piccoli da rendere $H_p(P) = H_p(T_r^n)$ a loro volta piccoli, ma al contempo abbastanza grandi da minimizzare la probabilità che si verifichino falsi positivi.

È possibile mostrare che dei buoni valori p possono essere ottenuti sfruttando le proprietà aritmetiche dei numeri primi. Dato un numero intero positivo x , si indica con $\pi(x)$ la **funzione enumerativa dei numeri primi**; questa funzione restituisce quanti numeri primi esistono nell'intervallo che ha per estremi (inclusi) 1 e x .

Teorema di Hadamard-Poussin (anche chiamato semplicemente **Teorema dei numeri primi**). Per x sufficientemente grandi, si ha $\pi(x) \approx \frac{x}{\ln(x)}$.

Disuguaglianza di Rosser-Schoenfeld. Sia $x \geq 29$ un numero intero positivo. Il prodotto di tutti i numeri primi minori di x è strettamente maggiore di 2^x .

Siano u e x due numeri interi positivi tali per cui $u \geq 29$ e $x \leq 2^u$. Allora x è scomponibile in meno di $\pi(u)$ fattori primi distinti.

Dimostrazione. Si supponga per assurdo che x sia scomponibile in $k > \pi(u)$ fattori primi distinti: q_1, q_2, \dots, q_k . Essendo questi tutti diversi e tutti contati con molteplicità uno, si ha che $x \geq q_1 \cdot q_2 \cdot \dots \cdot q_k$, perché x è dato dal prodotto dei suoi fattori primi tenendo conto della molteplicità. Ricordando che per ipotesi si ha $2^u \geq x$, per proprietà transitiva $2^u \geq q_1 \cdot q_2 \cdot \dots \cdot q_k$.

Non è noto quali numeri primi q_1, q_2, \dots, q_k siano, ma è certo che il loro prodotto sia almeno pari al prodotto dei primi k numeri primi $2 \cdot 3 \cdot 5 \cdot \dots$. A sua volta, il prodotto dei primi k numeri primi deve essere maggiore del prodotto dei primi πu numeri primi, in quanto si è supposto per assurdo che $k > \pi(u)$.

Si ha quindi per proprietà transitiva che 2^u è maggiore del prodotto dei primi $\pi(u)$ numeri primi, ovvero che 2^u è maggiore del prodotto di tutti i numeri primi minori o uguali a u . Questo è però in contraddizione con la Disuguaglianza di Rosser-Schoenfeld, che stabilisce che il prodotto di tutti i numeri primi minori o uguali a u è strettamente maggiore di 2^u . Pertanto, non è possibile assumere che esista un $k > \pi(u)$, ed il teorema è provato.

I teoremi relativi ai numeri primi appena riportati permettono di definire delle chiare limitazioni alla probabilità che avvenga un falso positivo.

Teorema centrale Karp-Rabin. Siano dati un pattern P ed un testo T , entrambe stringhe binarie, rispettivamente di lunghezza n e m , tali per cui $n \cdot m \geq 29$. Siano poi dati un numero intero positivo I ed un numero primo p scelto in modo da essere inferiore o uguale ad I . Allora la probabilità che si verifichi (almeno) un falso positivo del pattern P nel testo T è minore o uguale a $\frac{\pi(nm)}{\pi(I)}$.

Dimostrazione. Sia R l'insieme che contiene tutte le posizioni di T in cui P non inizia, ovvero $s \in R$ se e solo se non c'è una occorrenza di P in T a partire dalla posizione s . In altri termini, si ha $H(P) \neq H(T_s^n)$ per qualsiasi $s \in R$.

Si consideri la produttoria $\prod_{s \in R} (H(P) - H(T_s^n))$. Essendo P e T due stringhe binarie, i massimi valori che $H(P)$ e $H(T_s^n)$ possono assumere sono, rispettivamente, 2^{n-1} e 2^{m-1} . Pertanto, deve certamente aversi che $H(P) - H(T_s^n) \leq 2^n$ per qualsiasi s . Avendo la produttoria non più di m termini (nel caso limite in cui P non compaia mai in T), si ha che $\prod_{s \in R} (H(P) - H(T_s^n)) \leq 2^{nm}$. Applicando il teorema precedente alla produttoria, è possibile stabilire che questa è scomponibile in meno di $\pi(u)$ fattori primi distinti.

Si supponga che ad una certa posizione r di T si verifichi un falso positivo. Per definizione questo significa che $H_p(P) = H_p(T_r^n)$, ovvero che p divide senza resto $H(P) - H(T_r^n)$. Questo significa che p divide senza resto anche la produttoria $\prod_{s \in R} (H(P) - H(T_s^n))$, e quindi p è uno dei suoi fattori primi.

Se p permette il verificarsi di una falsa corrispondenza fra P e T , allora p deve essere membro di un insieme di almeno $\pi(nm)$ elementi. Tuttavia, p è stato scelto casualmente da un insieme di $\pi(I)$ elementi, pertanto la probabilità che p sia un numero primo che comporta il verificarsi di un falso positivo di P in T è non superiore a $\frac{\pi(nm)}{\pi(I)}$.

Per avere una bassa probabilità di ottenere un falso positivo è quindi sufficiente scegliere un numero primo qualsiasi minore o uguale ad I . Il problema si sposta quindi dalla scelta di un buon valore di p alla scelta di un buon valore di I : più I è grande, più la probabilità che si verifichi un falso positivo si riduce, ma al contempo aumenta il numero di possibili valori che p può assumere, rendendo più oneroso il calcolo di $H_p(P)$ e $H_p(T_r^n)$. Una regola empirica stabilisce che un buon candidato per il valore di I è nm^2 .

Siano dati un pattern P di lunghezza $n = 250$ ed un testo T di lunghezza $m = 4000$. Ponendo $I = nm^2$ si ha, in virtù del Teorema di Hadamard-Poussin:

$$\frac{\pi(nm)}{\pi(I)} = \frac{\pi(nm)}{\pi(nm^2)} \approx \frac{\frac{nm}{\ln(nm)}}{\frac{nm^2}{\ln(nm^2)}} = \frac{nm \cdot \ln(nm^2)}{\ln(nm) \cdot nm^2} = \frac{1}{m} \left[1 + \frac{\ln(m)}{\ln(n) + \ln(m)} \right] = \frac{1}{4000} \left[1 + \frac{8.3}{5.52 + 8.3} \right] = 0.0004$$

Ovvero, la probabilità che si verifichi almeno un falso positivo nella ricerca di P in T è circa del 0.04%.

2.3.2 Ottimizzazione: utilizzare più fingerprint

Anziché utilizzare una sola fingerprint, ci si chiede cosa accada se si ripete l'algoritmo k volte utilizzando una fingerprint diversa ad ogni iterazione. Come già detto in precedenza, l'algoritmo Karp-Rabin risolve il problema del pattern matching a meno di un errore sempre in eccesso; questo significa che, qualsiasi sia il valore di p scelto, una iterazione dell'algoritmo riporterà sempre tutte le "vere" occorrenze del pattern del testo più eventualmente delle occorrenze in più dovute all'errore. Pertanto, se si eseguono k iterazioni dell'algoritmo, ognuna di queste riporterà tutte le "vere" occorrenze più un errore di volta in volta diverso. Questo significa che intersecando gli insiemi delle soluzioni di ciascuna delle k iterazioni tutte le "vere" occorrenze di P in T rimarranno, mentre se un falso positivo è assente nella soluzione di anche una sola delle k iterazioni, verrà scartato. Operare k iterazioni dell'algoritmo permette quindi di "raffinare" il risultato finale, ottenendo un margine di errore via via più ristretto.

Quando vengono scelti k numeri primi fra 1 e I in maniera casuale ed indipendente, e vengono utilizzate k iterazioni dell'algoritmo Karp-Rabin, la probabilità che si verifichi lo stesso falso positivo in ciascuna di queste è $\left[\frac{\pi(nm)}{\pi(I)} \right]^k$.

Dimostrazione. Il teorema precedente ha mostrato che la probabilità che si verifichi un falso positivo è $\frac{\pi(nm)}{\pi(I)}$. Se i numeri primi vengono scelti casualmente ed indipendentemente l'uno dall'altro, ciascuna scelta può essere vista come un evento indipendente. Pertanto, la probabilità che si verifichi lo stesso falso positivo su k iterazioni è il prodotto delle probabilità che si verifichi su ciascuna iterazione. Essendo queste probabilità tutte uguali, la probabilità che si verifichi lo stesso falso positivo in tutte le iterazioni è semplicemente $\left[\frac{\pi(nm)}{\pi(I)} \right]^k$.

Si noti come il ripetere l'algoritmo Karp-Rabin più volte diminuisca la probabilità di errore più di quanto faccia perdere in termini di tempo di esecuzione. Questo perché eseguire l'algoritmo k volte fa aumentare il tempo di esecuzione linearmente (dato che viene semplicemente moltiplicato per k), ma la probabilità di ottenere un errore decresce esponenzialmente.

2.3.3 Ottimizzazione: azzerare gli errori

L'algoritmo Karp-Rabin é un algoritmo probabilistico; in particolare, é un esempio di **algoritmo Monte Carlo**. Algoritmi di questo tipo sono veloci, ma corretti a meno di un certo margine di errore; questi si contraddistinguono dagli **algoritmi Las Vegas**, che sono non sempre veloci ma sempre corretti. In alcuni casi é possibile convertire un algoritmo di tipo Monte Carlo in un algoritmo di tipo Las Vegas equivalente (e viceversa), ovvero perdere in termini di tempo di esecuzione per guadagnare in termini di correttezza (o viceversa), e l'algoritmo Karp-Rabin figura fra questi.

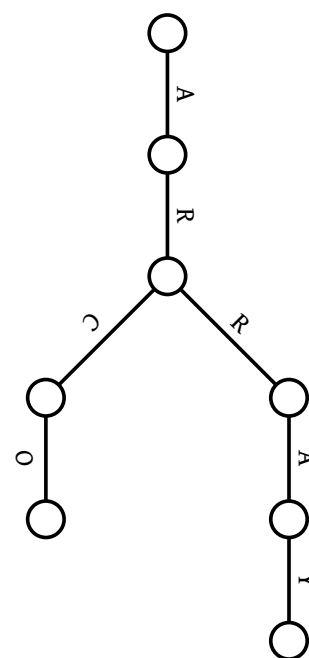
Capitolo 3

Suffix tree e suffix array

3.1 Trie e trie compatto

La struttura ad albero chiamata **trie** é la piú semplice struttura dati possibile che permette di determinare se una stringa appartiene o non appartiene ad un determinato insieme chiamato *dizionario*. Un trie é un DAG che ha un singolo carattere riportato sulle etichette di ciascun arco. Dato che il verso di percorrenza é sempre dall'alto verso il basso, si tende ad omettere esplicitamente la direzione degli archi. Concatenando tutte le etichette di un percorso che va dalla radice ad un nodo foglia si ottiene una delle stringhe che fanno parte del dizionario associato al trie. In particolare, ogni percorso identifica univocamente una stringa del dizionario, e viceversa.

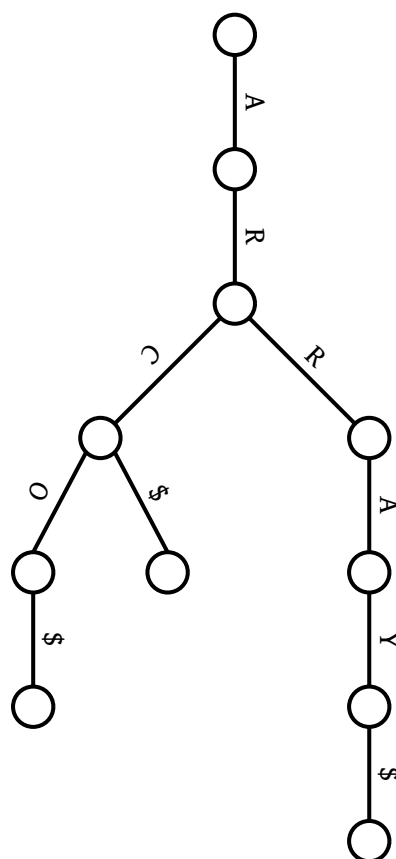
Seguendo tutti i possibili percorsi che esistono nel trie riportato a destra, si evidenzia come il dizionario a questo associato contenga le sole due parole *array* e *arco*.



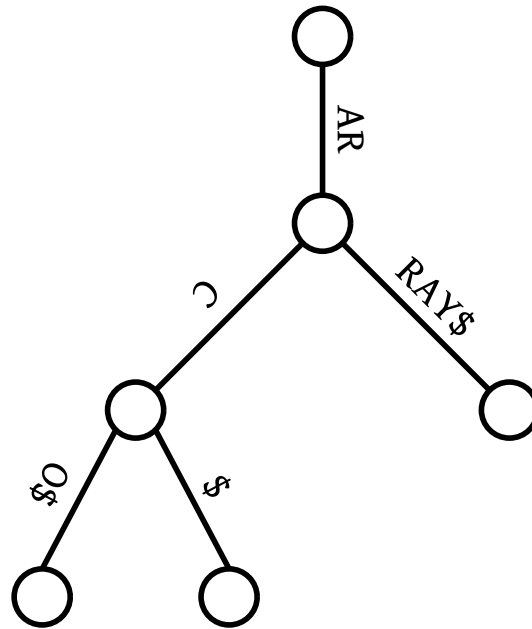
Per determinare se una stringa fa o non fa parte del dizionario associato ad un trie é sufficiente percorrerlo spostandosi dal i -esimo nodo al $i + 1$ -esimo nodo lungo gli archi che hanno per etichetta l' i -esimo carattere della stringa fino ad esaurirli tutti. Se si percorrono solo archi ordinatamente etichettati dai caratteri della stringa e l'ultimo arco conduce ad un nodo foglia, allora la parola é effettivamente parte del dizionario, altrimenti non lo é.

Questo criterio di appartenenza presenta però un problema: se nel dizionario sono presenti due stringhe S_1 ed S_2 tali per cui S_1 é prefisso di S_2 , non é possibile usare questo criterio per determinare se S_1 appartenga al dizionario. Questo perché passando S_1 come input all'algoritmo per l'appartenenza questo percorre tutti i nodi etichettati da caratteri della parola, ma termina in un nodo non foglia. Il problema può essere aggirato semplicemente modificando le stringhe in input e nel dizionario aggiungendovi un carattere di terminazione speciale (per convenzione si utilizza \$) che non fa parte dell'effettivo alfabeto delle stringhe.

Aggiungendo \$ alla fine di ogni parola é possibile rimuovere ogni ambiguitá legata ai prefissi. Infatti, la parola *arc* appartiene al dizionario relativo al trie riportato a lato perché percorrendo i nodi a partire dalla radice si ha un percorso etichettato *arc\$* che termina in un nodo foglia. D'altro canto, ad esempio, la parola *arra* non appartiene al dizionario.

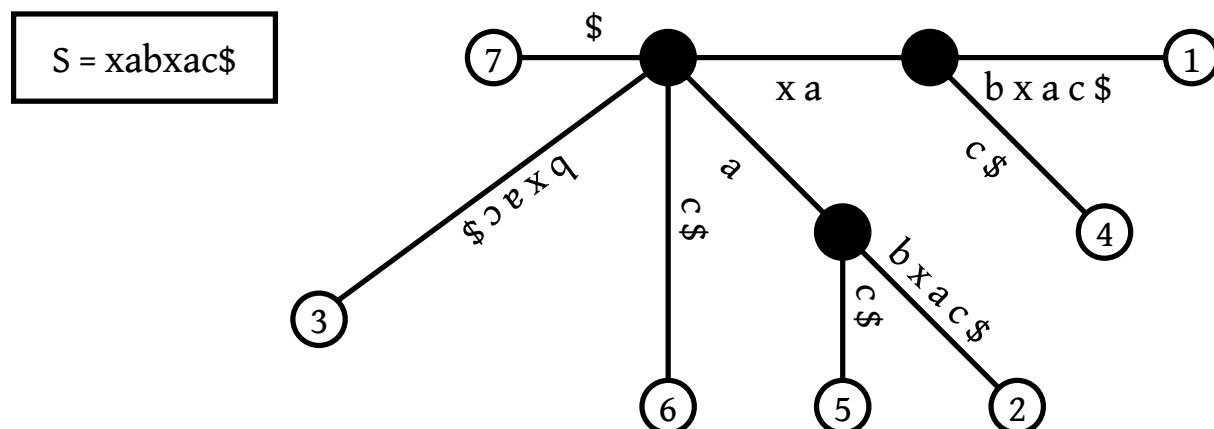


Un **compact trie** é un tipo particolare di trie dove ciascun cammino che non ha diramazioni viene accorpato in un unico arco fra due nodi, etichettato con la stringa che si ottiene concatenando ordinatamente tutti i caratteri sulle etichette che formavano il cammino. É possibile determinare se una parola appartiene oppure no al dizionario associato ad un compact trie nello stesso modo in cui si lo si fa rispetto ai trie, l'unica differenza é che non si procede di carattere in carattere ma per gruppi di caratteri.



3.2 Suffix tree

Data una stringa S di lunghezza m , viene chiamato **suffix tree** (o **albero dei suffissi**) per S il compact trie che ha per dizionario S e tutti i suoi suffissi. Un suffix tree per S ha esattamente m nodi foglia, numerati da 1 a m . Ogni nodo interno che non sia la radice ha almeno due nodi figli, ed ogni arco é etichettato da una sottostringa di S . Non possono esistere due archi in uscita da uno stesso nodo che abbiano una etichetta che inizi con lo stesso carattere. Esiste una corrispondenza biunivoca fra i percorsi che vanno dalla radice ai nodi foglia del suffix tree e i suffissi della stringa.



Dato un nodo u di un albero dei suffissi, Viene chiamata **path-label** di u la stringa costruita concatenando ordinatamente tutte le etichette degli archi del percorso che va dalla radice a u . La lunghezza della path-label di u viene chiamata **string-depth** di u . É possibile aggiungere a ciascun nodo di un albero dei suffissi la sua path-label e la sua string-depth semplicemente con una visita in profondità.

Data una stringa S di lunghezza m , é possibile costruire un albero dei suffissi relativo ad S mediante un algoritmo banale ¹ in tempo $O(n^2)$, basato

1. Esistono algoritmi efficienti che permettono di costruire alberi dei suffissi in tempo lineare, come l'**algoritmo di Ukkonen**, ma sono estremamente complessi.

sul costruire iterativamente m alberi intermedi per ciascun suffisso di S . Indicando con N_i l'albero intermedio che codifica tutti i suffissi di S da 1 a i , l'algoritmo può essere descritto come segue:

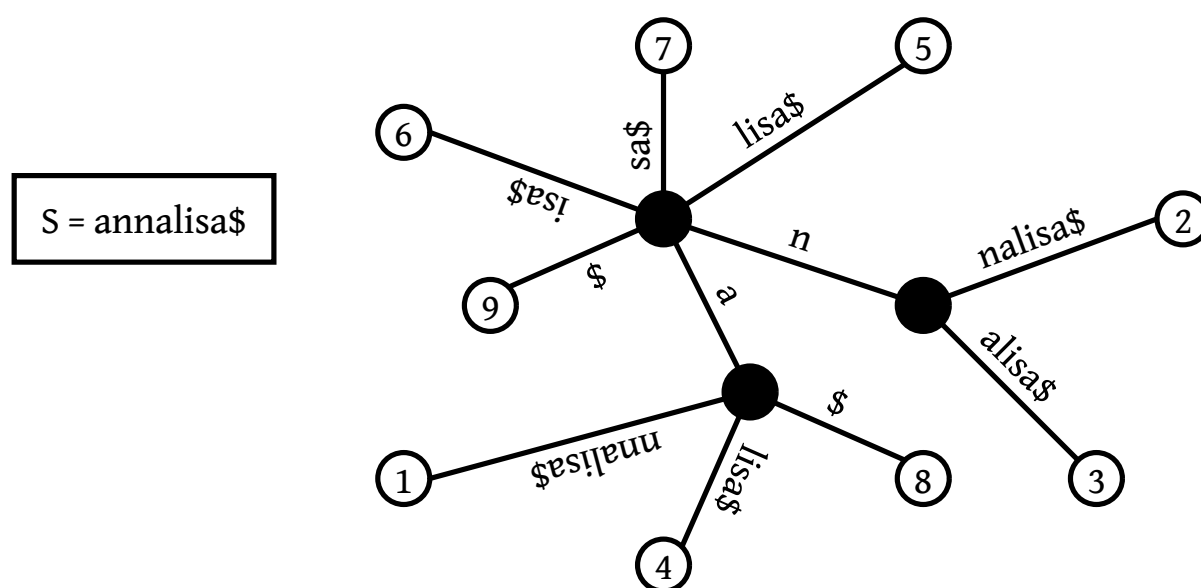
1. Innanzitutto viene costruito l'albero N_1 mediante un arco etichettato $S[1 : m]$, che equivale alla stringa S riportata per intero, che unisce la radice dell'albero con una foglia etichettata 1;
2. Il generico albero intermedio N_{i+1} è costruito a partire da N_i . Partendo dalla radice di N_i , si individui, se esiste, il percorso più lungo esistente a partire dalla radice la cui etichetta contiene un prefisso di $S[i + 1 : m]$;
3.
 - Se un percorso di questo tipo non esiste, viene aggiunto un nodo collegato direttamente alla radice avente $S[i + 1 : m]$ come etichetta;
 - Se invece questo percorso esiste, ci si aspetta che l'algoritmo si trovi nel mezzo di un arco (u, v) che va da un nodo u ad un nodo v , etichettato da una certa stringa c . Viene inserito nell'albero un nuovo nodo w , e l'arco (u, v) viene sostituito con due archi distinti, (u, w) e (w, v) . Il primo è etichettato con i caratteri di $S[i + 1 : m]$ che fanno da prefisso a c , mentre il secondo è etichettato con la stringa c a cui sono stati rimossi i primi $S[i + 1 : m]$ caratteri. Infine, vengono inseriti nell'albero un nodo foglia etichettato $i + 1$ ed un arco $(w, i + 1)$ etichettato con il suffisso di $S[i + 1 : m]$ che non ha trovato corrispondenza.
4. Se il numero di nodi foglia dell'albero finora costruito è inferiore ad m , si ripete l'algoritmo a partire dal punto 2. Altrimenti, l'algoritmo termina.

3.3 Pattern matching mediante suffix tree

Gli alberi dei suffissi si prestano bene ad essere utilizzati per risolvere il problema del pattern matching. Sia P un pattern di lunghezza n e sia T un testo di lunghezza m . Si costruisca in tempo lineare un albero dei suffissi per T (o si assuma che ve ne sia già uno a disposizione). Partendo dalla radice, si segua di nodo in nodo il percorso dell'albero dei suffissi etichettato con i caratteri di P . In ciascun passo dell'algoritmo possono verificarsi tre possibilità, mutualmente esclusive:

- Esiste un arco interamente etichettato da un prefisso $P[0 : i]$ di P , con $P[0 : i] \neq P$. In questo caso, ci si sposta lungo questo arco e si ripete l'algoritmo sul suffisso $P[i : m]$ sul sottoalbero dei suffissi che ha per radice il nodo corrente;
- Esiste un arco etichettato con l'intero P , oppure esiste un arco che ha l'intero P come prefisso della sua etichetta. In questo caso, ci si sposta lungo quest'arco e l'algoritmo termina segnalando che il numero di occorrenze di P in T è dato dal numero di nodi foglia che ha il sottoalbero dei suffissi che ha per radice il nodo corrente. Inoltre, il numero che etichetta questi nodi foglia corrisponde alle posizioni di T in cui si trova P ;
- Non esiste alcun arco etichettato con un prefisso di P , oppure non esiste alcun arco che abbia l'intero P come prefisso della sua etichetta. In questo caso, l'algoritmo termina segnalando che non esiste alcuna occorrenza di P in T .

L'algoritmo funziona perché, per definizione, una sottostringa di una stringa è un prefisso di un suo suffisso. Pertanto, P si trova in T in posizione i se e soltanto se P è un prefisso di $T[i : m]$. Questo accade se esiste un percorso che dalla radice dell'albero porta al (univoco) nodo etichettato i .



- Si consideri il pattern a . Partendo dalla radice, si percorre l'arco etichettato a , esaurendo completamente l'input, giungendo in un nodo interno avente tre nodi figli, etichettati rispettivamente con 1, 4, e 8. È possibile concludere che il pattern a compare 3 volte nella parola *annalisa*, precisamente in prima, quarta e ottava posizione;
- Si consideri il pattern na . Partendo dalla radice, si percorre l'arco etichettato n , dopodiché vi sono solamente due archi, etichettati rispettivamente *nalisa* e *alisa*. L'etichetta del secondo arco ha a come prefisso, e percorrendola si giunge al nodo foglia etichettato 3. È possibile concludere che il pattern nan compare 1 volta nella parola *annalisa*, precisamente in terza posizione;
- Si consideri il pattern nan . Partendo dalla radice, si percorre l'arco etichettato n , dopodiché vi sono solamente due archi, etichettati rispettivamente *nalisa* e *alisa*. Dato che l'input non è stato interamente consumato e che nessuna delle due etichette ha an come prefisso, è possibile concludere che il pattern nan compare 0 volte nella parola *annalisa*.

Per raggiungere il nodo radice del sottoalbero di interesse dove l'algoritmo termina occorre compiere al massimo m passaggi. A questo punto servono ulteriori k passaggi per attraversare il sottoalbero e individuarne tutti i nodi foglia. Pertanto, il tempo di esecuzione complessivo per eseguire l'algoritmo sui suffix tree per il pattern matching é $O(m + k)$. Naturalmente occorre anche tenere conto del tempo impiegato a costruire l'albero dei suffissi, che si presume essere $O(n)$.

3.4 Suffix array

I suffix tree sono strumenti molto potenti per risolvere il problema del pattern matching, ma sono anche molto esosi in termini di spazio. Nel caso in cui l'occupazione di memoria sia un fattore critico per la risoluzione di un problema, gli alberi dei suffissi potrebbero non rivelarsi lo strumento migliore. Esiste però una struttura dati **array dei suffissi** che é piú conservativa dell'albero dei suffissi in termini di spazio occupato e permette di risolvere il problema del pattern matching in un tempo comparabile.

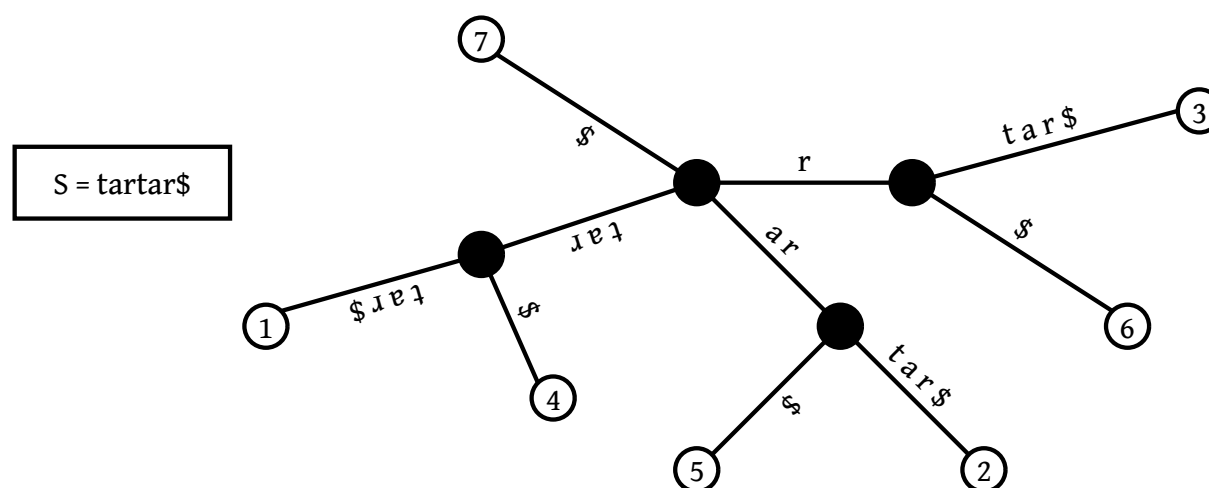
Data una stringa T di lunghezza m , l'array dei suffissi Pos é un array di interi nell'intervallo $[1, m]$ che specifica l'ordine lessicografico degli m suffissi della stringa T . Ovvero, il suffisso che inizia alla posizione $Pos[1]$ di T é il suffisso che lessicalmente si trova in prima posizione, mentre il suffisso che inizia alla posizione generica $Pos[i]$ di T viene prima, secondo l'ordine lessicografico, del suffisso che inizia alla posizione $Pos[i + 1]$.

Data la stringa $T = \text{mississippi\$}$, si ha $Pos = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$. Infatti:

12	11	8	5	2	1	10	9	7	4	6	3
\$	i\$	ippi\$	issippi\$	ississippi\$	mississippi\$	pi\$	ppi\$	sippi\$	sissippi\$	ssippi\$	ssissippi\$

Costruire un array dei suffissi a partire da una stringa é possibile, ma estremamente complesso. É però possibile convertire un albero dei suffissi in un array dei suffissi equivalente in tempo lineare; per farlo, occorre innanzitutto estendere la nozione di ordinamento lessicografico alle etichette degli archi degli array dei suffissi.

Dato un albero dei suffissi, siano u, v, w tre suoi nodi generici. Si dice che l'arco (v, u) dell'albero (se esiste) occupa una posizione inferiore in ordine lessicografico dell'arco (v, w) (se esiste) se e soltanto se il primo carattere dell'etichetta di (v, u) occupa una posizione inferiore in ordine lessicografico del primo carattere dell'etichetta del arco (v, w) . Per convenzione, si assume che il carattere speciale $\$$ venga prima di tutti gli altri. Dato che, per definizione, in un albero dei suffissi ogni nodo interno non può avere due archi uscenti con una etichetta che comincia con lo stesso carattere, é sempre possibile definire un ordine per queste etichette. Questo significa che per ottenere l'array dei suffissi é sufficiente visitare l'albero in profondità, partendo dalla radice e seguendo di volta in volta l'arco la cui etichetta viene prima secondo l'ordine lessicografico fino a raggiungere un nodo foglia, riportare l'etichetta del nodo e fare backtracking fino ad esaurire tutti nodi dell'albero. Il tempo per ricavare l'array dei suffissi é lineare perché, per il modo in cui l'albero dei suffissi é costruito, ogni nodo foglia viene visitato esattamente una sola volta.



1. Partendo dalla radice, si attraversa l'arco con l'etichetta che si trova prima in ordine lessicografico, ovvero \$, che termina subito nel nodo foglia etichettato 7. Occorre quindi fare subito backtracking e tornare alla radice;
2. Si procede a percorrere l'arco con l'etichetta che si trova prima in ordine lessicografico escluso \$, ovvero ar;
3. Qui si ha una biforcazione con due possibili cammini, etichettati con \$ e tar\$. Dato che \$ viene prima di tar\$ in ordine lessicografico, va attraversato prima il primo e poi il secondo;
4. Entrambi sono nodi foglia, rispettivamente 5 e 2, pertanto occorre fare backtracking fino alla radice;
5. Il arco con l'etichetta che si trova prima in ordine lessicografico escluso ar (che é già stato testato) é r;
6. Qui si ha una biforcazione con due possibili cammini, etichettati con \$ e tar\$. Dato che \$ viene prima di tar\$ in ordine lessicografico, va attraversato prima il primo e poi il secondo;
7. Entrambi sono nodi foglia, rispettivamente 6 e 3, pertanto occorre fare backtracking fino alla radice;
8. Il arco con l'etichetta che si trova prima in ordine lessicografico esclusi ar e r é tarr;
9. Qui si ha una biforcazione con due possibili cammini, etichettati con \$ e tar\$. Dato che \$ viene prima di tar\$ in ordine lessicografico, va attraversato prima il primo e poi il secondo;
10. Entrambi sono nodi foglia, rispettivamente 4 e 1, pertanto occorre fare backtracking fino alla radice;
11. Rileggendo l'ordine in cui i nodi foglia sono stati raggiunti, si ha che l'array dei suffissi associato all'albero in alto é $Pos = [7, 5, 2, 6, 3, 4, 1]$.

3.5 Longest common prefix

Data una stringa S di lunghezza m , sia Pos l'array dei suffissi associato a S , e siano i e j due posizioni su quest'ultimo. Viene chiamata **longest common prefix** di i e j , indicata con $Lcp(i, j)$, la lunghezza del prefisso più lungo che hanno in comune $S[Pos[i] : m]$ e $S[Pos[j] : m]$, ovvero rispettivamente il suffisso di S che inizia in posizione $Pos[i]$ ed il suffisso di S che inizia in posizione $Pos[j]$.

Si consideri l'array dei suffissi $Pos = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$ associato a $S = mississipp\$. Le due posizioni 5 e 3 sull'array dei suffissi corrispondono rispettivamente alle stringhe ississippi e ippi. Dato che il prefisso più lungo che queste hanno in comune é i, si ha $Lcp(5, 3) = 1$.$

Calcolare tutti i possibili valori LCP per una stringa é piuttosto complesso. Se però si considerano solamente i valori LCP di posizioni contigue, questi possono essere calcolati direttamente durante la costruzione dell'array dei suffissi a partire dall'albero dei suffissi, senza perdite in termini di tempo di esecuzione. Durante l'attraversamento in profondità dell'albero dei suffissi, si considerino i nodi interni che sono stati visitati fra la visita del nodo foglia etichettato $Pos[i]$ e del nodo foglia etichettato $Pos[i + 1]$ (ovvero, fra l' i -esimo nodo foglia ed il successivo nodo foglia). Fra questi nodi interni, si consideri quello più lontano dalla radice, eventualmente la radice stessa, chiamato **least common ancestor**, abbreviato come lca .

Il percorso che va dalla radice a $Pos[i]$ ed il percorso che va dalla radice a $Pos[i + 1]$ devono necessariamente passare entrambi per lca . Infatti, la string-depth di lca corrisponde alla lunghezza della parte di percorso in comune fra il percorso dalla radice a $Pos[i]$ ed il percorso dalla radice a $Pos[i + 1]$, che é proprio la definizione di $Lcp(i, i + 1)$. Individuare lca é semplice perché corrisponde al nodo che, durante la visita depth-first, ne fa invertire la direzione.

Si noti come, dati due nodi u e v dell'albero dei suffissi, se vale $Lcp(u, v) = 0$ allora u e v hanno come lca la radice. Questo perché se il valore LCP associato ai due nodi é nullo, allora la string-depth dei due nodi non ha alcun prefisso in comune, e l'unico nodo che non ha alcun prefisso associato é la radice. Viceversa, se due nodi u e v hanno come lca la radice, allora hanno nullo il valore LCP, perché significa che il prefisso più lungo in comune alle due string-depth é la stringa vuota.

Nello specifico, é possibile trovare le occorrenze di P in T mediante ricerca binaria sull'array dei suffissi. Si considera l'elemento di Pos che occupa la posizione centrale, $Pos[\frac{m}{2}]$, e si confronta $T[Pos[\frac{m}{2}] : m]$, il suffisso di T associato a quella posizione con P . Possono verificarsi tre situazioni:

- $T[Pos[\frac{m}{2}] : m]$ viene prima di P in ordine lessicografico. In questo caso, le eventuali occorrenze di P in T vanno cercate in $Pos[1 : (\frac{m}{2}-1)]$. Si prende allora l'elemento centrale di questo sotto-array come nuovo elemento di confronto e si ripete il procedimento;
- $T[Pos[\frac{m}{2}] : m]$ viene dopo di P in ordine lessicografico. In questo caso, le eventuali occorrenze di P in T vanno cercate in $Pos[(\frac{m}{2}+1) : m]$. Si prende allora l'elemento centrale di questo sotto-array come nuovo elemento di confronto e si ripete il procedimento;
- $T[Pos[\frac{m}{2}] : m]$ contiene interamente P come prefisso. In questo caso, occorre controllare le posizioni contigue a $Pos[\frac{m}{2}] : m$ fino ad esaurire posizioni da ambo i lati che contengano interamente P come prefisso (o fino a raggiungere gli estremi di Pos), tenendo traccia del loro numero. A questo punto, il procedimento termina.

Siano $T = \text{mississippi\$}$ e $P = \text{issi}$. Il relativo array dei suffissi é $Pos = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$, di lunghezza m .

- L'elemento centrale di Pos é 1, e $T[1 : m] = \text{mississippi\$}$ viene dopo P secondo l'ordine lessicografico. Pertanto, le occorrenze di P in T (se esistono) sono da cercarsi in $P_1 = [12, 11, 8, 5, 2]$;
- Si ripete il procedimento con l'elemento centrale di P_1 , ovvero 8. $T[8 : m] = \text{ippi\$}$ viene prima di P secondo l'ordine lessicografico, pertanto le occorrenze di P in T (se esistono) sono da cercarsi in $P_2 = [5, 2]$;
- Si ripete il procedimento con l'elemento centrale di P_2 , ovvero 5. $T[5 : m] = \text{issippi\$}$ contiene interamente P come prefisso, pertanto le altre occorrenze di P in T (se esistono) sono da cercare nelle posizioni contigue a 5;
- L'unica posizione contigua a 5 in P_2 é 2, ed effettivamente $T[2 : m] = \text{issippi\$}$ contiene interamente P come prefisso. Non avendosi altre posizioni da controllare, é possibile affermare che le occorrenze di P in T sono due, precisamente in posizione 2 e 5.

Un algoritmo di questo tipo, essendo di fatto una ricerca binaria, ha tempo di esecuzione logaritmico. Nello specifico, il tempo di esecuzione per la risoluzione del problema del pattern matching mediante ricerca binaria sull'array dei suffissi é $O(m \log(n))$. Ricordando che il problema del pattern matching viene risolto sfruttando gli alberi dei suffissi in tempo lineare, occorre operare delle ottimizzazioni per rendere i due tempi comparabili.

3.6.1 Ottimizzazione: ridurre il numero di confronti

In ogni iterazione dell'algoritmo di ricerca binaria, si indichi con L e R rispettivamente l'estremo inferiore e superiore dell'intervallo nell'array dei suffissi che contiene le posizioni delle possibili occorrenze di P , e sia M l'elemento mediano. Se sono note le quantità $Lcp(M, R)$, $Lcp(L, M)$ e $Lcp(L, R)$, é possibile utilizzarle per ridurre il numero di confronti necessari per determinare dove si trova P . Infatti, se $T[Pos[R] : m]$ e $T[Pos[L] : m]$ hanno un prefisso in comune, allora non é necessario confrontare tutti i loro caratteri, ma soltanto quelli a partire dalla posizione $Lcp[Pos[L], Pos[R]] + 1$.

Questa ottimizzazione non abbassa il tempo di esecuzione teorico (che rimane $O(m \log(n))$), ma permette comunque di rendere l'algoritmo leggermente piú efficiente sopprimendo alcuni confronti inutili. Il problema di questo approccio é che i valori LCP sono noti solamente per suffissi adiacenti in Pos , mentre in generale i suffissi che si trovano in L e R non lo sono. Pertanto, vanno calcolati ed aggiornati manualmente ad ogni iterazione.

3.6.2 Ottimizzazione: analisi caso per caso

3.6.3 Ottimizzazione: Calcolo dei valori LCP in tempo lineare

Osservando piú attentamente la struttura dell'algoritmo cosí definito, si può notare come gli estremi L e R dell'intervallo nell'array dei suffissi non sono mai del tutto casuali. Infatti:

$$(L_1, R_1) = (1, n) \quad (L_2, R_2) = \left(1, \frac{n}{2}\right) \cup \left(\frac{n}{2}, n\right) \quad \dots \quad (L_k, R_k) = \bigcup_{1 \leq i \leq k-1} \left(i \frac{n}{2^{k-1}}, (i+1) \frac{n}{2^{k-1}}\right)$$

L'array dei suffissi si trova allora ad essere ripartito in una struttura ad albero, dove le foglie sono gli intervalli di ampiezza 2 e la radice é l'intero array. Il numero totale di intervalli é certamente inferiore ad n , e in ciascuna iterazione dell'algoritmo viene calcolato un solo intervallo di ciascuna ampiezza. I valori LCP delle foglie, essendo consecutivi, corrispondono ai valori $Lcp(i, i+1)$ che sono stati ottenuti direttamente durante la costruzione dell'array dei suffissi, e pertanto sono già noti.

Occorre allora concentrarsi sul passo ricorsivo, ovvero come combinare fra loro i valori LCP di diversi intervalli. Si considerino due intervalli contigui aventi per estremi rispettivamente (L_1, R_1) e (L_2, R_2) , e si determini $Lcp(L_1, R_2)$. Si presume che il carattere in posizione L_1 ed il carattere in posizione R_2 siano diversi, perché se fossero uguali l'LCP verrebbe esteso di uno. Ci si interroga allora dove si trova il primo carattere diverso da quello in posizione L_1 . Vi sono tre possibilità:

- É contenuto nella prima metà. Allora, in questo caso, $Lcp(L_1, R_2) = Lcp(L_1, R_1)$;
- É contenuto nella seconda metà. Allora, in questo caso, $Lcp(L_1, R_2) = Lcp(L_2, R_2)$;
- Si trova esattamente nel mezzo. Allora, in questo caso, $Lcp(L_1, R_2) = Lcp(R_1, L_2)$.

Indipendentemente dal caso in cui ci si trovi, $Lcp(L_1, R_1), Lcp(L_2, R_2), Lcp(R_1, R_2) \geq Lcp(L_1, R_2)$, perché la porzione in comune fra due estremi di un sottointervallo non può essere inferiore alla porzione in comune fra i due estremi dell'intervallo intero. In particolare:

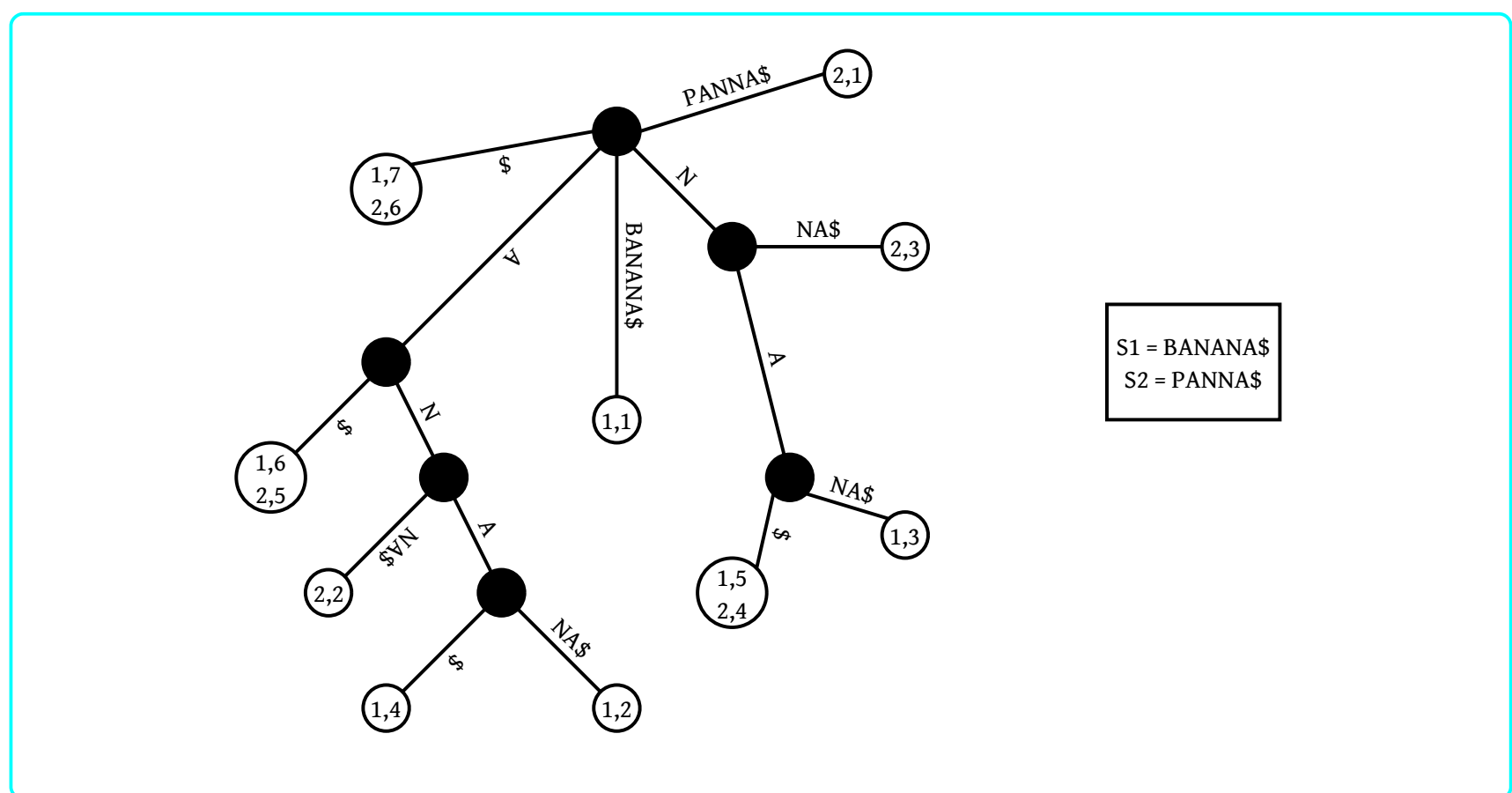
$$Lcp(L_1, R_2) = \min(Lcp(L_1, R_1), Lcp(L_2, R_2), Lcp(R_1, R_2))$$

Pertanto, il calcolo dei valori LCP per qualsiasi intervallo può essere calcolato in tempo lineare.

3.7 Generalized suffix tree e generalized suffix array

Un **albero dei suffissi generalizzato (generalized suffix tree)** è un albero dei suffissi che non è relativo ad una singola stringa e ai suoi suffissi, ma bensì a k stringhe e ai suffissi di ciascuna. I nodi foglia di un generalized suffix tree sono etichettati con al più k coppie di numeri: il primo numero di ciascuna coppia indica a quale stringa si sta facendo riferimento, mentre il secondo indica la posizione a partire dalla quale quel suffisso si trova in quella stringa.

Costruire un albero dei suffissi generalizzato per k stringhe è molto semplice. Occorre costruire l'albero dei suffissi per la prima stringa, dopodiché aggiungerci i suffissi della seconda stringa e così via fino ad esaurire tutte le k stringhe. Ciascun nodo foglia va etichettato riportando sia la i -esima stringa che si sta considerando sia la posizione del suffisso rispetto alla i -esima stringa. Nel caso in cui un nodo foglia venga raggiunto più volte, alla sua etichetta va aggiunta la i -esima coppia indice-posizione.



Un **array dei suffissi generalizzato (generalized suffix array)** è un array dei suffissi che non contiene i suffissi disposti in ordine lessicografico di una sola stringa, ma bensì contiene quelli di k stringhe. Per evitare che due diverse stringhe abbiano lo stesso suffisso, ai suffissi di ciascuna viene accodato uno stesso carattere di terminazione speciale univoco per ogni stringa.

Sebbene sia possibile costruire un array dei suffissi generalizzato "da zero", è molto più conveniente costruire un albero dei suffissi generalizzato per poi convertirlo in un array dei suffissi generalizzato equivalente. Innanzitutto, per definizione, ogni nodo foglia del suffix tree corrisponde ad un elemento del suffix array. Più in generale, preso un nodo interno del suffix tree, i nodi foglia del sottoalbero che ha tale nodo per radice corrispondono ad un intervallo di posizioni contigue nel suffix array, perché per definizione le string-depth di tali nodi sono stringhe che si trovano una dopo l'altra secondo l'ordine lessicografico.

Si noti però come la corrispondenza fra nodi foglia dell'albero dei suffissi generalizzato e elementi del suffix array generalizzato sia biunivoca, mentre la corrispondenza fra nodi interni ed intervalli non lo sia. Infatti, il numero totale di nodi di un generico albero è proporzionale al numero dei suoi nodi foglia, mentre il numero totale di intervalli costruibili in un array è proporzionale al quadrato della sua lunghezza. Avendo un generalized suffix array tanti elementi quanti sono i nodi foglia del generalized suffix tree equivalente, il numero totale di intervalli dell'array sarà per forza maggiore del numero di nodi dell'albero.

Siano x e y due nodi interni di un suffix tree. Se x è un progenitore di y , allora l'intervallo di posizioni contigue definito dai nodi foglia del sottoalbero avente x come radice contiene interamente l'analogo intervallo relativo ad y . Questo perché tutte le string-depth dei nodi foglia discendenti di y sono suffissi di prefissi delle string-depth dei nodi foglia discendenti di x . Se x e y sono invece due nodi che non sono in una relazione di discendenza diretta, allora l'intervallo di posizioni contigue dei nodi foglia discendenti di x e l'intervallo di posizioni contigue dei nodi foglia discendenti di y sono disgiunti.

3.8 Range minimum query

Viene chiamato **range minimum query (RMQ)** la soluzione del problema computazionale che, dato un vettore A di lunghezza n , richiede di calcolare il valore minimo di $A[i : j]$, il sottovettore di A che ha per estremi le posizioni i e j , con $1 \leq i \leq j \leq n$. Il problema può essere risolto in maniera banale osservando uno per uno gli elementi di $A[i : j]$ fino a trovarne il minimo, ed il tempo di esecuzione è $O(i-j) \approx O(n)$.

Una versione più raffinata del problema prevede di calcolare il valore minimo non soltanto per il singolo sottovettore $A[i : j]$ ma bensì, fissato j , calcolarlo per tutti i sottovettori nella forma $A[i : j]$, $A[i-1 : j]$, $A[i-2 : j]$ fino a $A[1 : j]$. Dato che il numero di possibili sottovettori compresi fra 1 e j con j fissato è non superiore ad n , se si volesse calcolare il valore minimo di ciascun sottovettore così definito usando lo stesso approccio il tempo di esecuzione sarebbe $O(n^2)$.

Un diverso approccio prevede di impiegare del tempo per preprocessare il vettore A al fine di ottenere una struttura dati ausiliaria che permetta però di conoscere il minimo di qualsiasi sottovettore di A in tempo costante. Sia a tal proposito M una matrice costruita a partire da A , dove ciascuna cella (x, y) contiene il valore minimo del sottovettore $A[x : y]$. Mediante una matrice di questo tipo è possibile conoscere il valore minimo di qualsiasi sottovettore di A semplicemente accedendo alla cella di M che ha per coordinate gli estremi dell'intervallo.

La matrice può essere costruita in maniera banale semplicemente calcolando il minimo per tutti i possibili sottovettori costruibili a partire da A , ma sarebbe una procedura estremamente inefficiente. Infatti, essendo il numero di tutti i possibili sottovettori di A pari a $O(n^2)$, e richiedendo ciascun valore minimo un tempo $O(n)$ per essere calcolato, un algoritmo di questo tipo impiegherebbe tempo di esecuzione $O(n^3)$.

3.8.1 Ottimizzazione: programmazione dinamica

È possibile velocizzare l'algoritmo mediante programmazione dinamica. Innanzitutto, è evidente come il valore minimo di un sottovettore di A formato da un solo valore è il valore stesso. Inoltre, presi due estremi x e y qualsiasi tali per cui $x < y$, si ha:

$$M[x, y] = \min(M[x, y-1], A[y])$$

In altre parole, il valore in posizione $M[x, y]$ deve essere il minimo fra $A[y]$, ovvero l'ultimo elemento del sottovettore $A[x : y]$, e $M[x, y-1]$, ovvero il minimo valore del sottovettore $A[x : y-1]$, cioè il sottovettore $A[x : y]$ a cui viene tolto l'ultimo elemento. In questo modo, venendo calcolato il minimo associato ad ogni elemento della cella una sola volta, il tempo di esecuzione scende a $O(n^2)$.

3.8.2 Ottimizzazione: lettura di due valori alla volta

Si noti come non sia strettamente necessario avere una struttura dati che contenga i valori minimi per tutti i sottovettori di un vettore. Se sono invece noti i valori minimi di solamente alcuni sottovettori (ma non tutti), è comunque possibile calcolare "sul momento" il valore minimo di un sottovettore qualsiasi. Naturalmente, più minimi vengono salvati sulla struttura dati e più tempo occorre per farlo, ma meno tempo occorrerà per usarli al fine di calcolare il valore minimo di un sottovettore qualsiasi. Viceversa, meno minimi vengono salvati e meno tempo occorre per calcolarli ma più tempo occorre per calcolare il minimo di un sottovettore qualsiasi.

Si consideri un sottovettore di A avente per estremi (a, d) , il cui valore minimo non è noto. Si assuma che siano invece noti i valori minimi di due sottovettori di A , parzialmente sovrapposti o contigui, aventi estremi (a, b) e (c, d) , con $a < c \leq b < d$. Il valore minimo di $A[a : d]$ deve essere il minimo fra il minimo di $A[a : b]$ e $A[c : d]$, perché per definizione $A[a : b] \cup A[c : d] = A[a : d]$.

Pertanto, è sufficiente che la matrice sia costruita in modo che, preso un qualsiasi sottovettore di A , questa contenga il minimo di (almeno) due sottovettori di A che lo contengono interamente. In particolare: preso un sottovettore di A avente estremi (x, y) , sia h il più piccolo esponente della potenza di 2 che ha meno elementi di quanti ne contenga tale intervallo, ovvero:

$$h = \lfloor \log_2(y-x+1) \rfloor \qquad 2^h \leq (y-x+1) \leq 2^{h+1}$$

A partire dal sottovettore $A[x : y]$, si considerino i due sottovettori $L = A[x : x+2^h-1]$ e $R = A[y-2^h+1 : y]$, di ampiezza h . Per il modo in cui h è stato definito, i due sottovettori così definiti saranno certamente contenuti all'interno di $A[x : y]$, pertanto il minimo di $A[x : y]$ può essere ottenuto ricavando il minimo di L ed il minimo di R , e scegliendo il minimo fra i due valori.

Si costruisca allora la matrice $D(x, h) = \min_{x \leq i \leq x+2^h} (A[i])$. Dato che x varia nell'intervallo $(1, n)$, mentre h varia nell'intervallo $(1, \log_2(n))$ la matrice D avrà dimensione $n \times \log_2(n)$. Come nella precedente ottimizzazione, la matrice D può essere costruita per via ricorsiva. Innanzitutto, anche in questo caso è evidente come $D[i, 0] = A[i]$, perché il minimo di un sottovettore con un solo elemento è l'elemento stesso. Per quanto riguarda la relazione di ricorrenza:

$$D[x, h] = \min(D[x, h-1], D[x+2^{h-1}, h-1])$$

La matrice D richiede un tempo $O(n \log(n))$ per essere costruita, ma ha il vantaggio di permettere di conoscere il valore minimo di un qualsiasi sottovettore in tempo costante. Infatti, è sufficiente scegliere due sottovettori distinti interamente contenuti nel sottovettore di interesse i quali valori minimi siano contenuti in D , e calcolare il minimo fra questi due.

	0	1	2	3
1	0	0	0	0
2	5	2	2	1
3	2	2	2	
4	5	4	1	
5	4	3	1	
6	3	1	1	
7	1	1		
8	6	3		
9	3			

Si consideri il vettore $A = [0, 5, 2, 5, 4, 3, 1, 6, 3]$, la cui matrice D é riportata a lato. Si voglia trovare il valore minimo del sottovettore $A[3 : 8]$:

$$h = \lfloor \log_2(8-3+1) \rfloor = 2$$

$$\begin{cases} A[3 : 3 + 2^2 - 1] = A[3 : 6] \\ A[8 - 2^2 + 1 : 8] = A[5 : 8] \end{cases}$$

Leggendo i valori della tabella, il minimo valore di $A[3 : 8]$ é dato da $\min(D[3, 2], D[5, 2]) = \min(2, 1) = 1$.

3.9 Sottostringa comune piú lunga

Viene chiamato **longest common substring** il problema che richiede di trovare la sottostringa piú lunga in comune fra due stringhe S_1 e S_2 (per comoditá, si assuma che entrambe siano di lunghezza n). Il problema puó essere risolto in maniera banale prendendo tutte le possibili sottostringhe di S_1 dalla piú lunga alla piú corta, e verificare se ogni i -esima sottostringa cosí costruita ha (almeno) una occorrenza all'interno di S_2 ; dato che si stanno considerando le sottostringhe di S_1 in ordine decrescente di lunghezza, appena ne viene individuata una che ha (almeno) una occorrenza in S_2 questa é certamente la piú lunga sottostringa in comune fra S_1 e S_2 .

Date $S_1 = \text{superiorcalifornialives}$ e $S_2 = \text{sealiver}$, la sottostringa comune piú lunga é *alive*.

L'algoritmo banale sopra descritto ha un tempo di esecuzione non inferiore a $O(n^3)$. Infatti, per individuare tutte le possibili sottostringhe di S_1 occorre impiegare un tempo quadratico (le sottostringhe di lunghezza $n-1$ sono 2, le sottostringhe di lunghezza $n-2$ sono 3, le sottostringhe di lunghezza $n-3$ sono 4, ecc...) e su ciascuna di questa occorre operare un algoritmo di pattern matching, che nella migliore delle ipotesi ha tempo $O(n)$.

Questo problema puó però essere risolto in tempo lineare facendo uso degli alberi dei suffissi generalizzati o degli array dei suffissi generalizzati. Si consideri il problema della sottostringa comune piú lunga limitato a due sole stringhe S_1 e S_2 , e si costruisca l'albero dei suffissi generalizzato relativo a queste. Un nodo si dice **favorevole** se il sottoalbero avente tale nodo per radice ha almeno un nodo foglia che corrisponde ad un suffisso di S_1 ed almeno un nodo foglia che corrisponde ad un suffisso di S_2 . Naturalmente, i nodi che hanno per discendenti dei nodi nella cui etichetta sono riportate informazioni relative ad entrambe le stringhe sono certamente favorevoli.

Se un nodo é favorevole, allora la sua path-label é necessariamente una sottostringa comune ad entrambe le stringhe, perché per definizione di albero dei suffissi tale path-label é un prefisso di un suffisso sia della prima stringa che della seconda stringa (non necessariamente la stessa). Non é però sempre vero l'opposto, ovvero se una sottostringa é comune ad entrambe le stringhe, allora non é detto che esista un nodo favorevole dell'albero dei suffissi generalizzato che ha per path-label tale sottostringa: questo puó accadere se la sottostringa é interamente contenuta all'interno dell'etichetta di un arco. Tuttavia, dato che si sta considerando il problema della sottostringa comune piú lunga, una situazione in cui la sottostringa comune piú lunga sia solo parte dell'etichetta di un arco non puó accadere, perché l'etichetta intera é sottostringa di entrambe ed é necessariamente piú lunga di una sua parte.

Dato che tutti i nodi favorevoli di un albero dei suffissi generalizzato hanno una path-label che é sottostringa comune ad entrambe le stringhe, per trovare la sottostringa comune piú lunga é sufficiente individuare il nodo favorevole avente la maggior string-depth fra tutti i nodi favorevoli. I nodi favorevoli di un albero dei suffissi generalizzato possono essere ricavati a partire da una visita bottom-up dell'albero: si contrassegni ciascun nodo dell'albero con un 1 se esiste almeno un nodo foglia nel sottoalbero avente per radice tale nodo che é relativo ad un suffisso di S_1 e/o con un 2 se esiste almeno un nodo foglia nel sottoalbero avente per radice tale nodo che é relativo ad un suffisso di S_2 .

Si consideri l'albero dei suffissi generalizzato associato alle stringhe $S_1 = \text{banana\$}$ e $S_2 = \text{panna\$}$. Le string-depth dei nodi favorevoli sono: ϵ , A , N , NA , AN . Le sottostringhe piú lunghe comuni alle due stringhe sono allora NA e AN , di lunghezza due.

Si consideri invece una situazione in cui si ha a disposizione un array dei suffissi generalizzato anziché un albero. Questo contiene tutti i possibili suffissi delle due stringhe e per ciascun suffisso è specificato a quale delle due stringhe si riferisce. La sottostringa comune piú lunga può essere ottenuta a partire da questa struttura dati operando una scansione lineare.

Per tutte le posizioni i sull'array dei suffissi generalizzato, si considerino gli intervalli (i, i) , $(i-1, i)$, $(i-2, i)$ e così via fino a $(1, i)$. Per ciascun intervallo, si calcoli il minimo valore LCP dei suffissi che contiene; se il successivo intervallo ha un LCP minimo piú grande del precedente, allora questo diventa l'intervallo che, per il momento, si presume contenga la sottostringa comune piú lunga. Quando tutti gli intervalli sono stati analizzati, l'algoritmo termina. Dato che il numero di intervalli è lineare ed il tempo per analizzare un qualsiasi intervallo è esso stesso lineare, l'algoritmo in questa forma ha un tempo di esecuzione $O(n^2)$.

3.9.1 Ottimizzazione: saltare gli intervalli che non contengono entrambe le stringhe

Gli intervalli sul generalized suffix array che non contengono almeno un suffisso di entrambe le stringhe non possono contenere una sottostringa comune alle due. Pertanto, una volta analizzato l'intervallo di estremi (i, j) , se l'elemento $i-1$ è un suffisso della stessa stringa di cui i è suffisso, allora l'intervallo $(i-1, j)$ è garantito che non possa contenere una sottostringa comune ad entrambe, e pertanto non è necessario analizzarlo. Questo può essere fatto in pratica, ad esempio, mediante un flag booleano che tenga traccia di quale delle due stringhe il suffisso in cima al precedente intervallo facesse parte. Si noti però che il tempo di esecuzione teorico rimanga comunque $O(n^2)$.

3.9.2 Ottimizzazione: valori LCP in tempo lineare

Il calcolo del valore LCP minimo di un intervallo può essere ottenuto in tempo costante perché il problema è di fatto identico al problema range minimum query. Essendo il numero di intervalli lineare e richiedendo ciascuno un tempo costante per estrarne il minimo dalla matrice RMQ, l'algoritmo termina in tempo $O(n)$. Ricordando però che la matrice che risolve questo problema richiede un tempo di esecuzione $O(n \log(n))$, ed essendo questo tempo maggiore di $O(n)$, il tempo complessivo perché l'algoritmo termini tenendo anche conto del preprocessing è $O(n \log(n))$.

Capitolo 4

Allineamento di sequenze

4.1 Distanza fra due stringhe

Il grado di somiglianza fra due stringhe viene genericamente chiamato **distanza**. Sebbene vi siano diverse formulazioni della distanza fra due stringhe, tutte possiedono le stesse proprietà di base. Matematicamente, la distanza è una metrica definita come $d : S \times S \mapsto R^+$, che assegna un valore reale ad ogni coppia di stringhe. La funzione distanza possiede le seguenti tre proprietà:

1. $d(x, z) \leq d(x, y) + d(y, z)$, la disuguaglianza triangolare;
2. $d(x, y) = 0$ se e soltanto se $x = y$;
3. $d(x, y) = d(y, x)$, la simmetricità.

La formulazione della distanza fra due stringhe più semplice e più utilizzata è la cosiddetta **distanza di modifica (edit distance)**, o **distanza di Levenshtein**. Date due stringhe S_1 e S_2 , la distanza di modifica fra le due è data dal numero di operazioni atomiche sui singoli caratteri necessario a trasformare S_1 in S_2 . Le operazioni ammesse sono quattro:

1. *Inserire* un carattere della seconda stringa nella prima;
2. *Rimuovere* un carattere dalla prima stringa;
3. *Sostituire* un carattere della prima stringa con uno della seconda;
4. non fare nulla (o equivalentemente, sostituire un carattere della prima stringa con sé stesso).

Siano le quattro operazioni sopra citate abbreviate rispettivamente con I , D e R e M . Una concatenazione di queste operazioni può essere espressa come una stringa sull'alfabeto $\{I, D, M, R\}$. Ovvero, una stringa del tipo $op_1op_2op_3...op_n$, costruita usando i quattro caratteri dell'alfabeto sopra descritto, è una stringa dove ciascun carattere op_i indica l'operazione applicata all' i -esimo carattere di S_1 per poterla trasformare in S_2 . Una stringa che esprime la sequenza di operazioni necessarie a trasformare una stringa in un'altra viene chiamata **trascrizione** delle due stringhe. La distanza di modifica fra due stringhe può essere allora vista anche come la lunghezza della trascrizione più corta possibile per tale coppia di stringhe escludendo le operazioni M , ovvero il minimo numero di operazioni "rilevanti" (tutte tranne quelle nulle) necessarie a trasformare una stringa in un'altra. Una trascrizione che permette di trasformare una stringa in un'altra nella maniera più efficiente possibile, ovvero quella che contiene il minimo numero di operazioni rilevanti, viene detta *ottimale*. Si noti come la trascrizione ottimale non sia necessariamente univoca, ma possano esservene diverse aventi tutte la stessa lunghezza.

R	I	M	D	M	D	M	M	I
v		i	n	t	n	e	r	
w	r	i		t		e	r	s

La stringa $S_1 = \text{vintner}$ può essere trasformata nella stringa $S_2 = \text{writers}$ mediante 5 operazioni: sostituire v con w , inserire r , eliminare n , eliminare n (di nuovo), inserire s . Le altre 4 operazioni sono no-op, perché quei caratteri in quelle posizioni sono già corretti. Pertanto, si ha che $d(S_1, S_2) = 5$.

È possibile descrivere meccanicamente il procedimento di trasformazione di una stringa in un'altra come segue. Siano S_1 ed S_2 due stringhe, delle quali è nota la trascrizione. Siano poi $next_1$ un puntatore a S_1 e sia $next_2$ un puntatore a S_2 , entrambi inizializzati ad 1. La trascrizione viene letta ed applicata da sinistra verso destra:

- Se viene letto il simbolo I , il carattere di S_2 in posizione $next_2$ viene inserito in S_1 alla posizione $next_1 - 1$, e $next_2$ viene incrementato di 1;
- Se viene letto il simbolo D , il carattere di S_1 in posizione $next_1$ viene eliminato da S_1 , e $next_1$ viene incrementato di 1;
- Se viene letto il simbolo R , il carattere di S_1 in posizione $next_1$ viene sostituito con il carattere di S_2 alla posizione $next_2$, ed entrambi i puntatori vengono incrementati di 1;
- Se viene letto il simbolo M , entrambi i puntatori vengono incrementati di 1 senza compiere alcuna operazione.

La lunghezza delle trascrizioni ottimali per una coppia di stringhe, e di conseguenza la loro distanza di modifica, può essere calcolata mediante programmazione dinamica. Siano S_1 e S_2 due stringhe di lunghezza rispettivamente n e m . Ciascuna cella (i, j) della matrice D contiene la lunghezza delle trascrizioni ottimali fra il prefisso $S_1[1 : i]$ di S_1 ed il prefisso $S_2[1 : j]$ di S_2 , ovvero $D(i, j) = d(S_1[1 : i], S_2[1 : j])$. La lunghezza delle trascrizioni ottimali fra S_1 e S_2 è quindi contenuta nella cella $D(n, m)$, ovvero $D(n, m) = d(S_1, S_2)$. Individuare il caso base della relazione di ricorrenza è semplice. Innanzitutto, è evidente come la lunghezza dell'unica trascrizione ottimale fra la stringa vuota e sé stessa sia zero, perché per ricavare ϵ a partire da ϵ nella maniera più efficiente possibile è sufficiente non compiere alcuna operazione. Inoltre, la lunghezza dell'unica trascrizione ottimale fra una qualsiasi stringa di lunghezza l e la stringa vuota è proprio l , perché per

ottenere la stringa vuota a partire da qualsiasi stringa nella maniera più efficiente possibile occorre effettuare tante operazioni D quanti sono i caratteri di tale stringa. Si ha allora:

$$D(0, 0) = d(\epsilon, \epsilon) = 0$$

$$D(i, 0) = d(S_1[1 : i], \epsilon) = i$$

$$D(0, j) = d(\epsilon, S_2[1 : j]) = j$$

Individuare il passo ricorsivo la relazione di ricorrenza è più complesso. Per prima cosa occorre, presa una cella generica (i, j) della matrice D , capire quali sono i possibili valori che questa può assumere.

Una cella generica (i, j) della matrice D può assumere esclusivamente uno fra questi quattro valori: $D(i-1, j) + 1$, $D(i, j-1) + 1$, $D(i-1, j-1) + 1$ o $D(i-1, j-1)$.

Dimostrazione. Si consideri una trascrizione ottimale qualsiasi associata alla trasformazione di una stringa S_1 in una stringa S_2 . Ci si concentri sull'ultimo carattere della trascrizione, che dovrà necessariamente essere I , D , R o M :

- Se è I , questo significa che l'ultima operazione che il processo di trasformazione ha compiuto è stata l'inserimento del j -esimo carattere della seconda stringa alla fine della prima stringa trasformata. Questo significa che i simboli della trascrizione precedenti costituiscono una trascrizione ottimale che trasforma $S_1[1 : i]$ in $S_1[1 : j-1]$, e questa trascrizione avrà per definizione lunghezza $D(i, j-1)$. Pertanto, se l'ultimo carattere della trascrizione è I , allora deve aversi $D(i, j) = D(i, j-1) + 1$;
- Se è D , questo significa che l'ultima operazione che il processo di trasformazione ha compiuto è stata l'eliminazione del i -esimo carattere della prima stringa. Questo significa che i simboli della trascrizione precedenti costituiscono una trascrizione ottimale che trasforma $S_1[1 : i-1]$ in $S_1[1 : j]$, e questa trascrizione avrà per definizione lunghezza $D(i-1, j)$. Pertanto, se l'ultimo carattere della trascrizione è D , allora deve aversi $D(i, j) = D(i-1, j) + 1$;
- Se è R , questo significa che l'ultima operazione che il processo di trasformazione ha compiuto è stata sostituire l' i -esimo carattere della prima stringa con il j -esimo carattere della seconda stringa. Questo significa che i simboli della trascrizione precedenti costituiscono una trascrizione ottimale che trasforma $S_1[1 : i-1]$ in $S_1[1 : j-1]$, e questa trascrizione avrà per definizione lunghezza $D(i-1, j-1)$. Pertanto, se l'ultimo carattere della trascrizione è R , allora deve aversi $D(i, j) = D(i-1, j-1) + 1$;
- Se è M , questo significa che l'ultima operazione che il processo di trasformazione ha compiuto è stata una no-op, e che l' i -esimo carattere della prima stringa era uguale al j -esimo carattere della seconda. Quindi $D(i, j) = D(i-1, j-1)$, perché il valore della distanza nello step precedente è rimasto lo stesso.

Una volta noti quali possono essere i valori che le celle della matrice D può assumere, occorre determinare quale fra questi è il valore che, per ciascuna iterazione dell'algoritmo, rende la trascrizione (parziale) ottimale.

Il valore ottimale da porre nella cella generica (i, j) della matrice D è il minimo dei quattro possibili.

Dimostrazione. Sia S_1 una qualsiasi stringa che si vuole trasformare, in maniera ottimale, in una stringa S_2 . Si consideri il passaggio intermedio (obbligato) dove il prefisso $S_1[1 : i]$ viene trasformato nel prefisso $S_2[1 : j]$:

1. È possibile trasformare $S_1[1 : i]$ in $S_2[1 : j]$ in maniera ottimale impiegando $D(i, j-1) + 1$ operazioni. Per fare questo occorre trasformare $S_1[1 : i]$ in $S_2[1 : j-1]$ in maniera ottimale, per poi compiere un'ultima operazione inserendo il j -esimo carattere di S_2 alla fine di S_1 . Dato che trasformare $S_1[1 : i]$ in $S_2[1 : j-1]$ in maniera ottimale richiede $D(i, j-1)$ operazioni, trasformare $S_1[1 : i]$ in $S_2[1 : j]$ richiederà $D(i, j-1) + 1$ operazioni;
2. È possibile trasformare $S_1[1 : i]$ in $S_2[1 : j]$ in maniera ottimale impiegando $D(i-1, j) + 1$ operazioni. Per fare questo occorre trasformare $S_1[1 : i-1]$ in $S_2[1 : j]$ in maniera ottimale, per poi compiere un'ultima operazione eliminando l' i -esimo carattere di S_1 . Dato che trasformare $S_1[1 : i-1]$ in $S_2[1 : j]$ in maniera ottimale richiede $D(i-1, j)$ operazioni, trasformare $S_1[1 : i]$ in $S_2[1 : j]$ richiederà $D(i-1, j) + 1$ operazioni;
3. È possibile trasformare $S_1[1 : i]$ in $S_2[1 : j]$ in maniera ottimale impiegando $D(i-1, j-1) + 1$ operazioni. Per fare questo occorre trasformare $S_1[1 : i-1]$ in $S_2[1 : j-1]$ in maniera ottimale, per poi compiere un'ultima operazione sostituendo l' i -esimo carattere di S_1 con il j -esimo carattere di S_2 . Dato che trasformare $S_1[1 : i-1]$ in $S_2[1 : j-1]$ in maniera ottimale richiede $D(i-1, j-1)$ operazioni, trasformare $S_1[1 : i]$ in $S_2[1 : j]$ richiederà $D(i-1, j-1) + 1$ operazioni;
4. È possibile trasformare $S_1[1 : i]$ in $S_2[1 : j]$ in maniera ottimale impiegando $D(i-1, j-1)$ operazioni, ma solo nel caso in cui $S_1[i] = S_2[j]$. Per fare questo occorre trasformare $S_1[1 : i-1]$ in $S_2[1 : j-1]$ in maniera ottimale, dopodiché essendo $S_1[i] = S_2[j]$ non è necessario compiere alcuna operazione perché $S_1[i]$ è già prefisso ottimale. Dato che trasformare $S_1[1 : i-1]$ in $S_2[1 : j-1]$ in maniera ottimale richiede $D(i-1, j-1)$ operazioni, trasformare $S_1[1 : i]$ in $S_2[1 : j]$ quando $S_1[i] = S_2[j]$ richiederà $D(i-1, j-1)$ operazioni;

Questo significa che $D(i, j)$ è certamente non superiore a $D(i-1, j) + 1$, a $D(i, j-1) + 1$, a $D(i-1, j-1) + 1$ e a $D(i-1, j-1)$. Essendo questi quattro valori anche gli unici che $D(i, j)$ può assumere, deve allora aversi che il valore ottimale per $D(i, j)$ è il minimo fra questi.

Si ha quindi che il passo ricorsivo della relazione di ricorrenza é dato da:

$$D(i, j) = \min(D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + 1, D(i-1, j-1))$$

Avendo a disposizione sia il caso base che il passo ricorsivo della relazione di ricorrenza, diventa possibile costruire la matrice risolutiva. I caratteri della prima stringa occupano la prima riga, mentre quelli della seconda stringa la prima colonna. La seconda riga e la seconda colonna vengono riempite con le informazioni disponibili dal caso base della relazione di ricorrenza, mentre le successive righe/colonne vengono calcolate una alla volta sulla base della precedente. Dato che per calcolare il valore di ogni cella occorre effettuare lo stesso confronto, e dato che il numero di celle complessivo é $(n + 1) \times (m + 1)$, si ha che l'algorithmo che calcola la distanza di modifica fra due stringhe ha un tempo di esecuzione di circa $O(mn)$.

		s	u	n	d	a	y
	0	1	2	3	4	5	6
s	1						
a	2						
t	3						
u	4						
r	5						
d	6						
a	7						
y	8						

		s	u	n	d	a	y
	0	1	2	3	4	5	6
s	1	0	1	2	3	4	5
a	2	1	1	2	3	3	4
t	3	2	2	2	3	4	4
u	4	3	2	3	3	4	5
r	5	4	3	3	4	4	5
d	6	5	4	4	3	4	5
a	7	6	5	5	4	3	4
y	8	7	6	6	5	4	3

```
S1 = "..."  
S2 = "..."  
  
m = len(S1) + 1  
n = len(S2) + 1  
  
Lev = [[0] * (n) for _ in range(m)]  
  
for i in range(1, m):  
    Lev[i][0] = i  
for j in range(1, n):  
    Lev[0][j] = j  
  
for i in range(1, m):  
    for j in range(1, n):  
        A = Lev[i - 1][j] + 1  
        B = Lev[i][j - 1] + 1  
        C = Lev[i - 1][j - 1]  
  
        if (S1[i - 1] != S2[j - 1]):  
            C = C + 1  
  
        Lev[i][j] = min(A, B, C)  
  
print(Lev[m - 1][n - 1])
```

Oltre a calcolare la lunghezza delle trascrizioni ottimali, é possibile estendere l'algorithmo per ricavare tali trascrizioni. Mentre la tabella viene costruita, é possibile aggiungere dei puntatori alle celle in modo da poter ricostruire a ritroso la trascrizione che trasforma una stringa nell'altra. In particolare, una volta che é stato calcolato il valore della cella (i, j) della tabella, viene messo un puntatore che va da quest'ultima alla cella di posizione $(i, j-1)$ se $D(i, j) = D(i, j-1) + 1$, alla cella di posizione $(i-1, j)$ se $D(i, j) = D(i-1, j) + 1$, alla cella di posizione $(i-1, j-1)$ se $D(i, j) = D(i-1, j-1) + 1$ oppure alla cella di posizione $(i-1, j-1)$ se $D(i, j) = D(i-1, j-1)$. Si noti come sia possibile che da una stessa cella si diramino piú puntatori, perché piú di una delle quattro condizioni possono verificarsi contemporaneamente.

Per ricavare una trascrizione ottimale a partire dalla tabella, é sufficiente seguire un qualsiasi percorso definito dai puntatori dalla cella (n, m) alla cella $(0, 0)$. Se ci si sposta *orizzontalmente*, ovvero da una cella (i, j) ad una cella $(i, j-1)$, allora significa che l'operazione piú efficiente per quel passaggio é un inserimento. Se ci si sposta *verticalmente*, ovvero da una cella (i, j) ad una cella $(i-1, j)$ allora significa che l'operazione piú efficiente per quel passaggio é una rimozione. Se ci si sposta *diagonalmente*, ovvero da una cella (i, j) ad una cella $(i-1, j-1)$, allora significa che l'operazione piú efficiente per quel passaggio é una sostituzione se $S_1[i] \neq S_2[j]$ o una no-op altrimenti.

Una trascrizione viene costruita concatenando alla stringa, inizialmente vuota, il carattere I ogni volta che si incontra \leftarrow , il carattere D ogni volta che si incontra \uparrow , il carattere R ogni volta che si incontra \nwarrow e $S_1[i] \neq S_2[j]$ oppure il carattere M ogni volta che si incontra \nwarrow e $S_1[i] = S_2[j]$. Dato che le operazioni stanno venendo lette dall'ultima alla prima, la trascrizione ottimale si ottiene leggendo tale stringa in ordine inverso.

		w	r	i	t	e	r	s
	0	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow 5$	$\leftarrow 6$	$\leftarrow 7$
v	$\uparrow 1$	$\nwarrow 1$	$\nwarrow \leftarrow 2$	$\nwarrow \leftarrow 3$	$\nwarrow \leftarrow 4$	$\nwarrow \leftarrow 5$	$\nwarrow \leftarrow 6$	$\nwarrow \leftarrow 7$
i	$\uparrow 2$	$\nwarrow \uparrow 2$	$\nwarrow 2$	$\nwarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow 5$	$\leftarrow 6$
n	$\uparrow 3$	$\nwarrow \uparrow 3$	$\nwarrow \uparrow 3$	$\nwarrow \uparrow 3$	$\uparrow 3$	$\uparrow \leftarrow 4$	$\uparrow \leftarrow 5$	$\uparrow \leftarrow 6$
t	$\uparrow 4$	$\nwarrow \uparrow 4$	$\nwarrow \uparrow 4$	$\nwarrow \uparrow 4$	$\nwarrow 3$	$\nwarrow \leftarrow 4$	$\nwarrow \leftarrow 5$	$\nwarrow \leftarrow 6$
n	$\uparrow 5$	$\nwarrow \uparrow 5$	$\nwarrow \uparrow 5$	$\nwarrow \uparrow 5$	$\uparrow 4$	$\nwarrow 4$	$\nwarrow \leftarrow 5$	$\nwarrow \leftarrow 6$
e	$\uparrow 6$	$\nwarrow \uparrow 6$	$\nwarrow \uparrow 6$	$\nwarrow \uparrow 6$	$\uparrow 5$	$\nwarrow 4$	$\nwarrow \leftarrow 5$	$\nwarrow \leftarrow 6$
r	$\uparrow 7$	$\nwarrow \uparrow 7$	$\nwarrow 6$	$\nwarrow \leftarrow \uparrow 7$	$\uparrow 6$	$\uparrow 5$	$\nwarrow 4$	$\leftarrow 5$

Seguendo tutti i possibili percorsi definiti dai puntatori, si ha che vi sono tre possibili trascrizioni ottimali: *RIMDMDMI*, *IRMDMDMI* e *RRRMDMMI*. Come ci si aspettava, tutte e tre hanno la stessa lunghezza.

Una volta che la tabella é stata costruita, ripercorrerla per ricavare le trascrizioni ottimali richiede un tempo di esecuzione $O(n + m)$, pertanto il tempo complessivo dell'algoritmo non ne viene inficiato.

4.2 Allineamento globale di due sequenze

La distanza di modifica si rivela inadatta ad esprimere il grado di somiglianza fra due sequenze. Una misura migliore é fornita dal loro **allineamento globale**, ovvero dall'inserimento all'interno delle stesse di un certo numero di caratteri speciali detti **indel** (*Insertion DEletion*) tale da renderle della stessa lunghezza.

Come il nome suggerisce, un indel in una sequenza deve essere interpretato come la rimozione di un carattere in tale sequenza o come l'inserimento di un carattere nell'altra sequenza. Dato che l'inserimento degli indel rende le due sequenze della stessa lunghezza, queste possono poi essere "incolonnate" carattere per carattere per mostrare il grado di somiglianza fra le due. L'unico vincolo al posizionamento degli indel in un allineamento globale é che non può essere presente un indel nella stessa posizione in entrambe le sequenze: tutti gli allineamenti globali che rispettano tale vincolo sono detti ammissibili.

Si considerino le sequenze *ABRACADABRA* e *BANANA*. Alcuni allineamenti globali ammissibili sono i seguenti:

A B R A C A D A B R A

- B - A N A - - - N A

A B R - A C - A D A B R A

- - - B - A N A - - - N A

A B R A C A D A B R A

- B A N A - - - - N A

Capitolo 5

Ricostruzione della filogenesi

5.1 Filogenesi ed evoluzione delle specie

La visione dominante dell'evoluzione della vita é che tutte le specie esistenti derivino da un antenato comune, ed una nuova specie possa emergere essenzialmente in due situazioni: quando una popolazione viene divisa in piú popolazioni che rimangono isolate abbastanza a lungo oppure quando il codice genetico di una popolazione subisce abbastanza mutazioni vantaggiose che vengono poi propagate nella prole. Due specie possono dirsi distinte quando un individuo della prima non é piú in grado di avere prole fertile con individui della seconda, e viceversa. Quando da una specie se ne genera una distinta si dice che é avvenuto un **evento di speciazione**.

La stragrande maggioranza delle mutazioni sono o neutre, ovvero non influenzano la capacità riproduttiva della specie, oppure negative, ovvero la riducono. Soltanto una piccola parte delle mutazioni si rivela essere vantaggiosa, e nella maggior parte dei casi sono relative a singoli nucleotidi. In casi molto rari possono però verificarsi mutazioni che coinvolgono intere sezioni del DNA, come le **inserzioni** (viene inserito in un punto del DNA un blocco di nucleotidi) o le **inversioni** (i nucleotidi di un blocco vengono riposizionati in ordine inverso).

Diviene allora naturale rappresentare l'evoluzione delle specie viventi come una struttura ad albero. Viene chiamato **albero filogenetico** un albero i cui nodi rappresentano le specie e i cui archi rappresentano la relazione di parentela che sussiste fra queste. In particolare, i nodi foglia dell'albero sono le specie attualmente esistenti, mentre i nodi interni rappresentano gli eventi di speciazione. La direzione degli archi rappresenta il verso in cui l'evoluzione é avvenuta (chi é antenato e chi é discendente), mentre la sua lunghezza o la sua etichetta rappresentano "quanto" una specie differisce dal suo antenato. In questo modo, un qualsiasi cammino sull'albero che va dalla radice ad una foglia corrisponde alla storia evolutiva della specie che il nodo foglia rappresenta.

La misura che quantifica la distanza fra una specie ed un suo antenato non é univoca, e influenza notevolmente la costruzione (e la struttura) dell'albero filogenetico. Fra le metriche in genere utilizzate ve ne sono due: la distanza temporale fra gli eventi di speciazione ed il numero di mutazioni che ha condotto alla speciazione. Si noti però che le informazioni relative a queste metriche, che sia la distanza temporale o il numero di mutazioni avvenute, sono spesso note solo parzialmente, e soltanto per alcuni step della storia evolutiva delle specie. Pertanto, gran parte della filogenesi é una ricostruzione fatta a posteriori, e non vi é modo di provarne la veridicità. Un **modello di filogenesi** é un determinato approccio alla ricostruzione della filogenesi, enfatizzando certi aspetti a discapito di altri e facendo determinate assunzioni.

L'evoluzione delle specie non può mostrare un effetto apprezzabile se ci si limita ad osservare un solo individuo della popolazione, perché la speciazione é un fenomeno che si verifica con l'accumulo di molteplici cambiamenti di generazione in generazione. Inoltre, come già detto, diverse mutazioni che un individuo subisce non possono trasmettersi ai discendenti, perché non hanno nulla a che vedere con mutazioni del suo codice genetico, e nascono e muoiono insieme all'individuo stesso.

Tuttavia, lo studio delle cellule di un singolo individuo e le loro mutazioni può essere assimilato allo studio dell'evoluzione di una popolazione, e fra le due non vi é quasi differenza. Pertanto, nonostante abbiano finalità del tutto diverse, l'evoluzione delle specie e l'evoluzione delle cellule di un singolo individuo sono fenomeni studiabili applicando le stesse metodologie, ed in particolare i modelli utilizzati per descrivere la filogenesi della prima possono essere applicati (quasi) indifferentemente anche per la seconda.

5.2 Problema della filogenesi perfetta







Viene chiamato **modello di filogenesi perfetta** un modello di filogenesi che identifica gli eventi di speciazione nella presenza o assenza di **caratteri** da parte delle specie. Con carattere si può intendere indifferentemente sia una proprietà fenotipica, quindi una proprietà osservabile che le specie possiedono o non possiedono (avere/non avere le branchie, avere quattro/cinque/sei/... dita dei piedi, ecc...) oppure una proprietà genotipica, ovvero una particolare sequenza di DNA avente una utilità nota che le cellule della specie hanno o non hanno (avere/non avere la sequenza che codifica per l'enzima galattasi, ad esempio). Il modello di filogenesi perfetta é un modello che enfatizza la quantità di "differenze" che sussistono fra le specie per determinarne la filogenesi. Inoltre, ha interesse esclusivamente a dedurre l'ordine in cui si sono verificati gli eventi di speciazione, mentre non fornisce alcuna indicazione sull'effettivo istante temporale in cui questi sono avvenuti. Questo modello, seppur molto semplice, é piuttosto ristretto, perché si basa su due presupposti molto forti:

1. Si considerano solo caratteri binari, ovvero caratteri che una specie può esclusivamente possedere o non possedere;
2. Non possono avvenire **backmutations**, ovvero una volta che una specie ha ereditato un carattere da un antenato non può più perderlo ¹.

1. Nonostante sia raro, fenomeni di questo tipo possono effettivamente accadere. Esistono altri modelli di filogenesi, come ad esempio il **modello Dollo** o il **modello 012**, che contemplano la possibilità che si verifichino backmutations, ma hanno un potere predittivo nettamente inferiore al modello di filogenesi perfetta.

Il problema della filogenesi perfetta consiste quindi, sulla base di un insieme di caratteri binari e su un insieme di specie delle quali é noto quali caratteri ciascuna specie possiede, nel costruire un albero filogenetico per le suddette specie. Innanzitutto, si costruisca una matrice binaria M , dove ciascuna riga rappresenta una certa specie mentre ciascuna colonna rappresenta un carattere che una specie può avere o non avere. Se una cella (i, j) contiene il valore 1, significa che la specie i possiede il carattere j , mentre se contiene il valore 0 significa che la specie i non possiede il carattere j .

Si consideri il seguente gruppo di animali: anguilla, leopardo, salamandra, scorpione, tartaruga, tonno. Si consideri poi il seguente insieme di caratteri morfologici: sacca amniotica, fauci, gambe, pelo, colonna vertebrale. Sulla base delle informazioni note a destra, é possibile costruire la matrice M a sinistra.







	sacca amniotica	fauci	gambe	pelo	colonna vertebrale
	0	0	0	0	1
	1	1	1	1	1
	0	1	1	0	1
	0	0	0	0	0
	1	1	1	0	1
	0	1	0	0	1

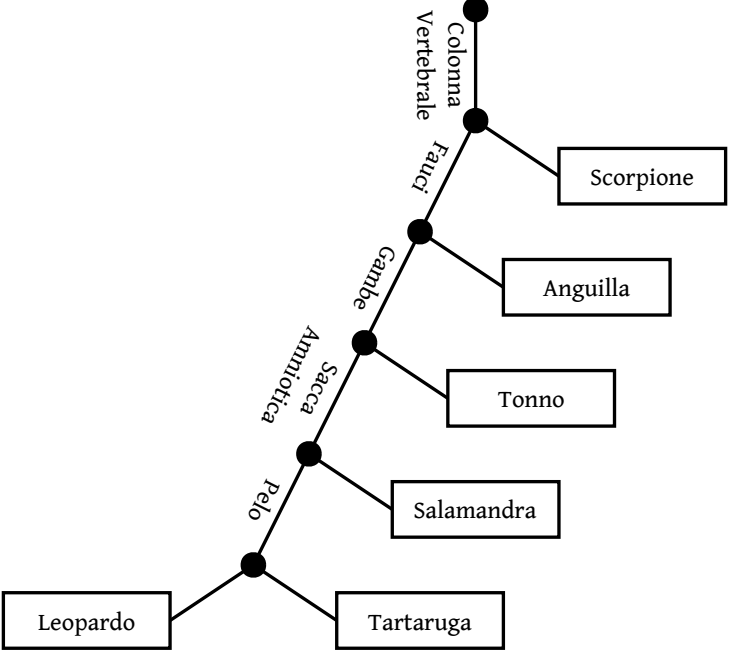
- La anguilla possiede la colonna vertebrale;
- Il leopardo possiede la sacca amniotica, le fauci, le gambe, il pelo e la colonna vertebrale;
- La salamandra possiede le fauci, le gambe e la colonna vertebrale;
- Lo scorpione non possiede nessuna di queste caratteristiche;
- La tartaruga possiede la sacca amniotica, le fauci, le gambe e la colonna vertebrale;
- Il tonno possiede le fauci e la colonna vertebrale.

La soluzione del problema della filogenesi perfetta viene allora ad essere un albero che **spiega** la matrice M , ovvero che rappresenta le stesse informazioni della matrice. Se esiste, questo albero filogenetico avrà le seguenti caratteristiche:

1. Ciascuna delle n specie etichetta una ed una sola foglia dell'albero;
2. Ciascuno degli m caratteri etichetta uno ed un solo arco dell'albero;
3. Per ciascuna specie p , i caratteri che etichettano gli archi lungo il (solo) percorso che va dalla radice dell'albero alla foglia p specificano tutti i caratteri di p che nella tabella hanno valore 1.

Un algoritmo molto semplice che risolve il problema della filogenesi perfetta, ovvero che permette di costruire un albero filogenetico a partire da una matrice binaria M , é l'**algoritmo di Gnsfeld**. L'algoritmo si compone di due parti: prima si ordina la matrice M in tempo lineare mediante radix sort, poi si costruisce l'albero aggiungendo una specie alla volta dalla più a sinistra alla più a destra nella matrice. Essendo radix sort un algoritmo con tempo di esecuzione lineare ed essendo il numero di passaggi necessari ad inserire tutte le specie esso stesso lineare, il tempo di esecuzione complessivo dell'algoritmo di Gnsfeld é lineare.

	CV	F	G	SA	P
	1	0	0	0	0
	1	1	1	1	1
	1	1	1	0	0
	0	0	0	0	0
	1	1	1	1	0
	1	1	0	0	0



I motivi per cui l'algoritmo funziona sono essenzialmente due. Il primo é da cercarsi in una proprietà della matrice M : per quanto sia possibile costruire una matrice binaria per qualsiasi insieme di caratteri e di specie, non é necessariamente detto che esista un albero filogenetico in grado di spiegarla.

Data una matrice binaria ordinata M , sia $S(i)$ l'insieme che contiene tutti i caratteri posseduti dalla specie i . Esiste un albero filogenetico che spiega la matrice M se, per qualsiasi coppia di due specie (i, j) , i relativi insiemi $S(i)$ e $S(j)$ sono o disgiunti (nessun elemento dell'uno si trova nell'altro) o uno dei due é interamente contenuto nell'altro.

Dimostrazione. Sia T l'albero filogenetico che spiega la matrice M , e si consideri la coppia di caratteri distinti (i, j) di T (o di M). Siano poi e_i e e_j gli archi di T etichettati, rispettivamente, i e j , ovvero gli archi a partire dai quali i due caratteri passano dall'avere valore 0 all'avere valore 1. Tutte le specie che possiedono il carattere i devono necessariamente trovarsi nei nodi foglia dopo e_i , cosí come tutte le specie che possiedono il carattere j devono necessariamente trovarsi nei nodi foglia dopo e_j . Si hanno tre possibilità, mutualmente esclusive:

1. e_i si trova sul percorso che dalla radice di T arriva a e_j . Questo significa che tutti gli oggetti che possiedono il carattere j possiedono anche il carattere i , pertanto $S(j)$ é un sottoinsieme di $S(i)$;
2. e_j si trova sul percorso che dalla radice di T arriva a e_i . Questo significa che tutti gli oggetti che possiedono il carattere i possiedono anche il carattere j , pertanto $S(i)$ é un sottoinsieme di $S(j)$;
3. Non c'è un percorso che dalla radice di T porti sia a e_i e che a e_j . Questo significa che un oggetto non può possedere contemporaneamente il carattere i ed il carattere j , pertanto gli insiemi $S(i)$ e $S(j)$ sono disgiunti.

Dato che in ciascuna di queste quattro possibilità i due insiemi sono o disgiunti o l'uno é interamente contenuto nell'altro, il teorema é dimostrato.

	1	2	3	4	5
A	1	1	0	0	0
B	0	0	1	1	0
C	1	1	1	0	0
D	0	0	0	1	1
E	1	0	1	0	0

Si considerino gli insiemi $S(1)$ e $S(3)$ della matrice riportata a lato (giá ordinata). Questi non sono né disgiunti (il terzo elemento é comune ad entrambi) né l'uno é sottoinsieme dell'altro (il secondo elemento differisce nei due insiemi). Pertanto, non può esistere un albero filogenetico che spieghi questa matrice.

Il secondo motivo é legato all'algoritmo di ordinamento impiegato. Si osservi infatti come radix sort, a differenza della maggior parte degli algoritmi di ordinamento, ordina i numeri in ordine decrescente. Questo significa che le colonne della matrice M vengono ordinate da quella che, intesa come una stringa binaria, ha valore maggiore a quella che ha valore minore. Avendo però mostrato che ogni colonna a sinistra di un'altra la comprende, l'albero filogenetico ottenuto avrà gli archi etichettati con caratteri posseduti da molte specie più in alto, mentre avrà gli archi etichettati con caratteri posseduti da poche specie più in basso.

5.3 Piccolo problema di parsimonia

Il problema della filogenesi perfetta richiedeva di dedurre la storia evolutiva delle specie in base all'ordine in cui hanno acquisito dei caratteri, ovvero in base all'ordine in cui il genoma di ciascuna specie ha subito la mutazione necessaria affinché quel carattere si manifestasse. É però possibile procedere anche in senso contrario, ovvero dedurre l'ordine in cui si sono verificate le mutazioni sul DNA di una specie sulla base di quali caratteri ha acquisito lungo la sua storia evolutiva.

In altre parole, se sono noti quali caratteri possiedono delle specie per le quali la filogenesi é nota, l'interesse é quello di determinare se e quali caratteri possedevano i loro progenitori. Si noti come queste informazioni non possono essere ottenute con certezza, dato che sarebbe necessario avere dati biologici su tutti gli antenati delle specie, e non sempre questo é possibile. Ciò che é possibile fare é dedurre se e quali fossero i caratteri che più "ragionevolmente" gli antenati avrebbero dovuto possedere, e assumere che siano quelli effettivamente posseduti da questi ultimi.

Un approccio spesso usato in contesti come questo é il cosiddetto **principio di massima parsimonia**, ovvero assumere che il processo che é

avvenuto sia sempre quello che aveva la maggior probabilità di avvenire ², perché è quello che richiede meno "sforzo" in termini evolutivi. Il problema basato su questo principio che richiede di determinare se e quali caratteri erano posseduti dai progenitori delle specie, conoscendo se e quali caratteri sono posseduti dalle specie attuali e conoscendo la loro filogenesi, è chiamato **piccolo problema di parsimonia**.

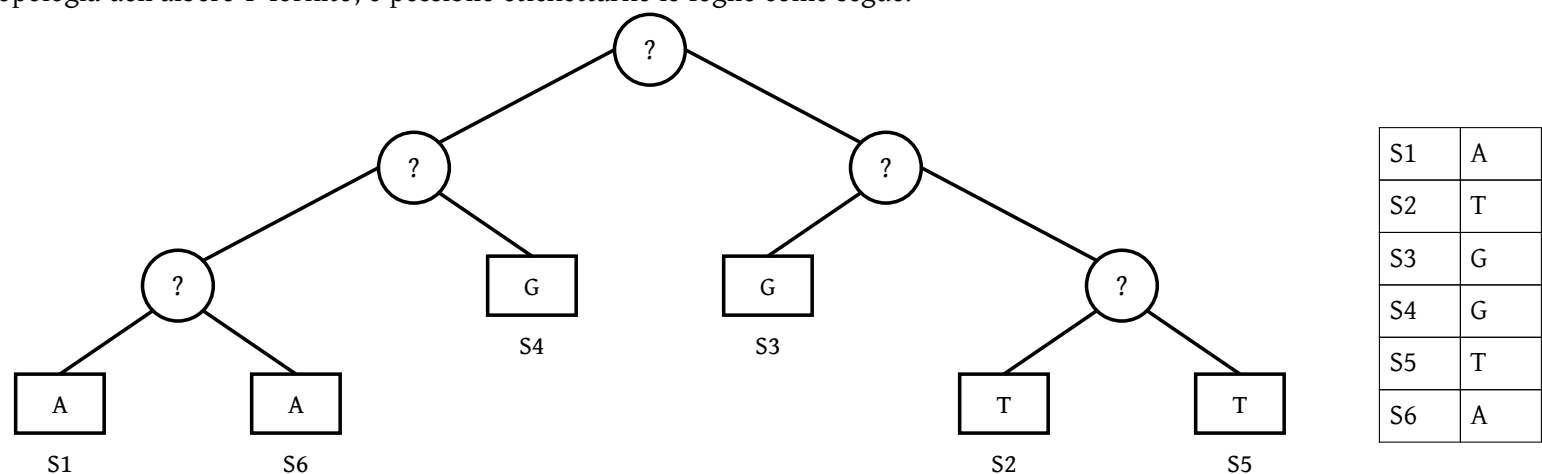
5.3.1 Peso uniforme: algoritmo di Fitch

Sia dato un insieme S di m specie S_1, S_2, \dots, S_m ed un certo carattere c . Questo carattere non è necessariamente binario, ma può esistere in k stati distinti. Dato che lo stato del carattere dipende (in buona approssimazione) da quale nucleotide il DNA della specie possiede nel locus relativo a tale espressione fenotipica, con un leggero abuso di notazione si tende ad indicare lo stato del carattere c con una delle quattro lettere A, C, G, T ³. Si assuma di conoscere lo stato del carattere c per ciascuna delle m specie di S .

Sia poi dato un albero filogenetico T per S del quale è nota la **topologia**, ovvero si conosce quanti nodi l'albero ha, come sono disposti, quanti archi esistono e quali coppie di nodi ciascuno di questi collega. Come da definizione, le foglie dell'albero rappresentano le specie viventi, mentre i nodi interni rappresentano gli antenati comuni.

Si etichetti ciascun nodo foglia con lo stato del carattere c per la specie che tale nodo rappresenta (è stato infatti assunto che fosse noto lo stato di c per ciascuna specie). Il piccolo problema di parsimonia, formulato a questo contesto, richiederebbe di determinare qual'è lo stato di c degli antenati delle specie di S , che di fatto equivale a determinare qual'è l'etichetta dei nodi interni dell'albero T . Si noti come questa non è necessariamente univoca, dato che potrebbero esserci diverse etichette per i nodi interni che con uguale probabilità rappresentano lo stato del carattere per quel nodo.

Sulla base della matrice a destra, che riporta le informazioni relative allo stato del carattere c per ciascuna specie di S , e sulla base della topologia dell'albero T fornito, è possibile etichettarne le foglie come segue:



Il carattere è presente in tutte le specie in un certo stato (non necessariamente distinto), pertanto si presume che questo possa o cambiare stato al verificarsi di una mutazione, e venire ereditato così dai figli, oppure restare nello stesso stato in cui si trova. Il principio di massima parsimonia prevede che le spiegazioni più probabili siano quelle più plausibili, ed essendo il verificarsi di una mutazione un evento molto raro, si assume che il cambio di stato del carattere di specie in specie avvenga con una probabilità molto più bassa del mantenimento dello stesso stato nei discendenti.

Se si assume che la probabilità del cambio di stato di un carattere sia la stessa a prescindere di quale sia lo stato di arrivo (e che sia inferiore al non verificarsi del cambio di stato), e se si assume che l'albero filogenetico T del quale è nota la topologia sia binario (ovvero che ogni specie abbia esattamente due discendenti), il problema di piccola parsimonia così formulato è risolvibile dall'**algoritmo di Fitch**.

Dato un nodo $x \in T$, sia $\lambda(x)$ l'etichetta che è stata scelta da assegnare ad x (o, nel caso dei nodi foglia, quella nota). Sia poi $S(x)$ l'insieme formato da tutti gli stati di c ottimali per x , ovvero quelli che hanno la maggior probabilità di essere gli effettivi stati del carattere per la specie relativa a x . Dato che la parte di etichettatura dell'albero già nota è quella delle foglie, l'algoritmo dovrà necessariamente partire dalle foglie e risalire verso la radice con una visita bottom-up, e terminare una volta che è stata trovata l'etichetta da assegnare alla radice.

Se x è un nodo foglia, dato che le etichette dei nodi foglia sono note, si ha che $S(x)$ è un singoletto formato dal solo elemento $\lambda(x)$. Se x è invece un nodo generico, dato che si sta considerando un albero binario questo dovrà avere esattamente due figli, f_l e f_r , e sono noti (da una iterazione precedente dell'algoritmo) i valori $S(f_l)$ e $S(f_r)$. Se i due insiemi hanno degli elementi (degli stati) in comune, allora gli stati di x che con più probabilità rispecchiano il vero stato della specie associata ad x sono quelli che si trovano sia in $S(f_l)$ che in $S(f_r)$. Se invece i due insiemi non hanno alcun elemento (alcuno stato) in comune, allora qualsiasi loro elemento ha la stessa probabilità di essere lo stato che etichetta x , perché non vi sono abbastanza informazioni per poter compiere una scelta. In altre parole:

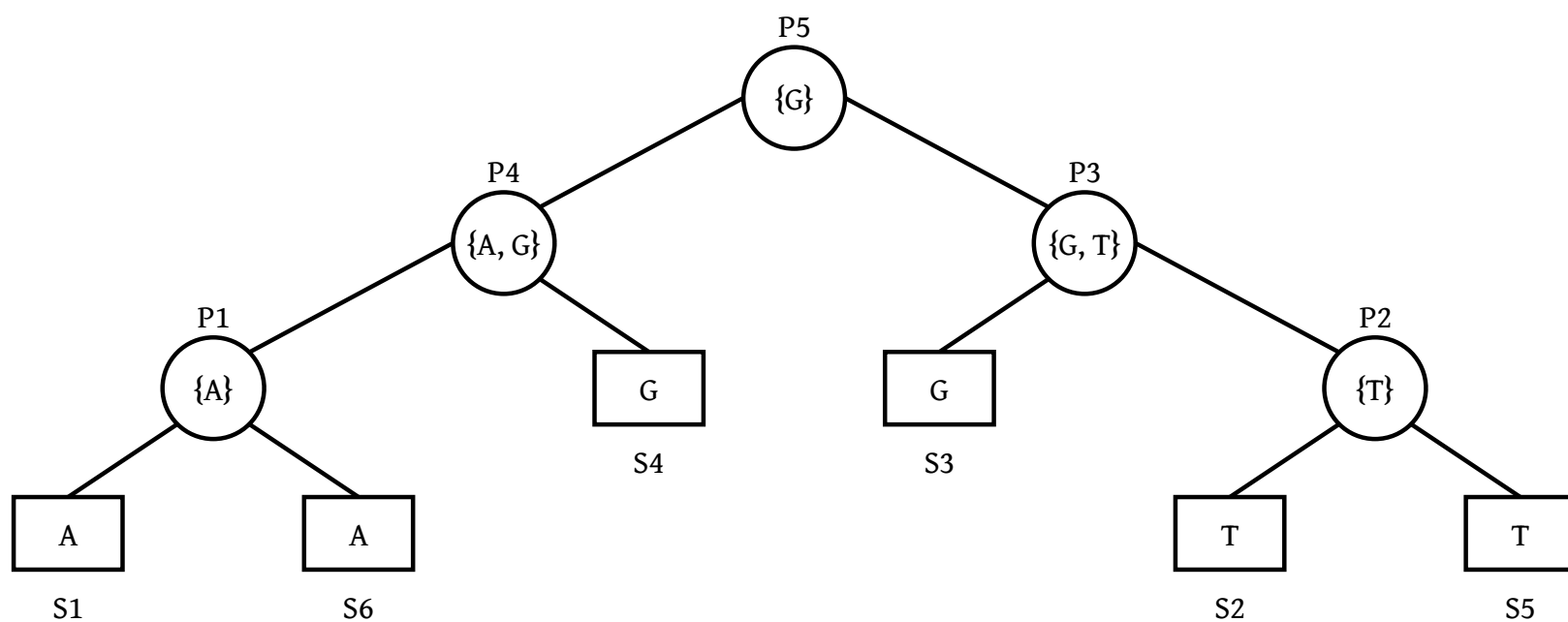
$$S(x) = \begin{cases} S(f_l) \cap S(f_r) & \text{se } S(f_l) \cap S(f_r) \neq \emptyset \\ S(f_l) \cup S(f_r) & \text{se } S(f_l) \cap S(f_r) = \emptyset \end{cases}$$

2. Il principio di massima parsimonia adotta lo stesso principio del Rasoio di Occam: la soluzione più semplice fra quelle possibili è quella più probabile. Naturalmente non è sempre così, ma non è possibile provarlo.

3. In realtà, nella maggior parte dei casi ciascun componente del fenotipo di una specie dipende da diverse sezioni di DNA. È infatti molto raro che un carattere sia determinato da uno ed un solo nucleotide.

Nel caso in cui vi siano nodi interni etichettati con più di uno stato e si volesse sceglierne arbitrariamente uno solo, una buona regola pratica consiste nel propagare (quanto possibile) lo stato che è stato scelto per etichettare la radice.

L'algoritmo di Fitch può essere esteso in maniera molto semplice anche al caso in cui l'albero T a disposizione non sia binario, ovvero ciascuna specie può avere k discendenti anziché due. Per farlo, è sufficiente modificare la scelta del valore per $S(x)$ con x generico, imponendo che $S(x)$ sia composto dai caratteri che compaiono con maggiore frequenza nei nodi figli. Si noti come questa sia semplicemente una generalizzazione del procedimento finora adottato: sia che si scelga l'intersezione fra i due insiemi nel caso in cui l'intersezione non sia nulla, sia che si scelga l'unione fra i due insiemi nel caso in cui l'intersezione sia nulla, in entrambi i casi questo corrisponde a scegliere i caratteri che compaiono con maggiore frequenza (in particolare, nel secondo caso tutti hanno la stessa frequenza).



- I nodi $S1$ e $S6$ sono entrambi etichettati A , pertanto il nodo $P1$ sarà etichettato $\{A\}$;
- Il nodo $P1$ è etichettato $\{A\}$, mentre il nodo $S4$ è etichettato G . Dato che le due etichette non hanno elementi in comune, il nodo $P4$ sarà etichettato $\{A, G\}$;
- I nodi $S2$ e $S5$ sono entrambi etichettati T , pertanto il nodo $P2$ sarà etichettato $\{T\}$;
- Il nodo $P2$ è etichettato $\{T\}$, mentre il nodo $S3$ è etichettato G . Dato che le due etichette non hanno elementi in comune, il nodo $P3$ sarà etichettato $\{G, T\}$;
- Il nodo $P3$ è etichettato $\{G, T\}$, mentre il nodo $P4$ è etichettato $\{A, G\}$. Dato che l'intersezione fra le due etichette non è vuota, l'etichetta di $P6$ (la radice) sarà $\{G, T\} \cap \{A, G\} = \{G\}$.

La regola pratica impone di propagare quanto possibile l'etichetta della radice nei figli, pertanto se si dovesse scegliere una etichetta per $P3$ o per $P4$, in entrambi i casi $\{G\}$ sarebbe la scelta da preferirsi.

5.3.2 Peso non uniforme: algoritmo di Sankoff

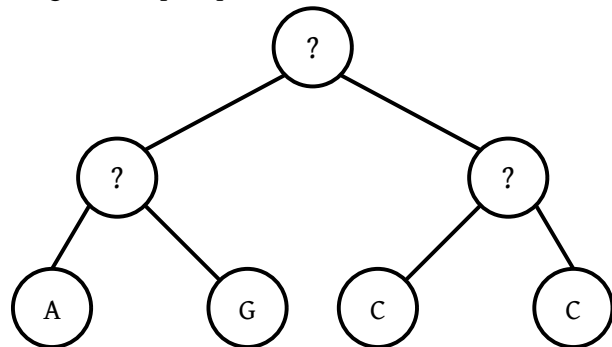
Si consideri una situazione analoga alla precedente, dove è dato un insieme S di m specie S_1, S_2, \dots, S_m ed un carattere c avente k stati distinti per ciascuna. Si introduca però un vincolo aggiuntivo, ovvero che la probabilità che avvenga una transizione da uno stato all'altro non sia uniforme, ma che sia (potenzialmente) diversa per qualsiasi coppia di stati.

Se le probabilità in questione sono note, possono essere espresse con un numero intero (similmente a quanto è stato fatto per l'allineamento delle sequenze), che ha un valore tanto alto quanto è bassa la probabilità che avvenga tale transizione. Per convenzione, si assume che la probabilità che un carattere non cambi valore, che equivale a dire che non va in contro ad una transizione di stato, sia zero. Questi valori possono essere riportati in una matrice di score di dimensione $k \times k$, dove ciascuna cella (i, j) contiene il numero intero associato alla probabilità che il carattere c cambi stato da i a j , chiamato **score**; ciascuno di questi valori viene indicato con $w(i, j)$.

Per ogni nodo $v \in T$, sia $\lambda(v)$ l'etichetta che è stata scelta (o fornita) per il nodo v . Se i nodi v_1 e v_2 sono uniti da un arco, è possibile riportare $w(\lambda(v_1), \lambda(v_2))$ sull'etichetta dell'arco v_1, v_2 , ad indicare che la transizione è avvenuta con tale probabilità. Ricordando che i valori $w(i, j)$ sono tanto bassi quanto è alta la probabilità che avvenga la transizione dallo stato i allo stato j , le migliori scelte per le etichette dei nodi interni dell'albero T sono quelle che rendono le etichette dei suoi archi le più piccole possibili:

$$\min \left(\sum_{(v_1, v_2) \in T} w(\lambda(v_1), \lambda(v_2)) \right)$$

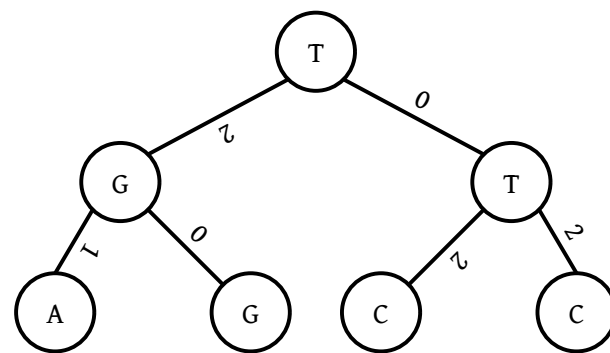
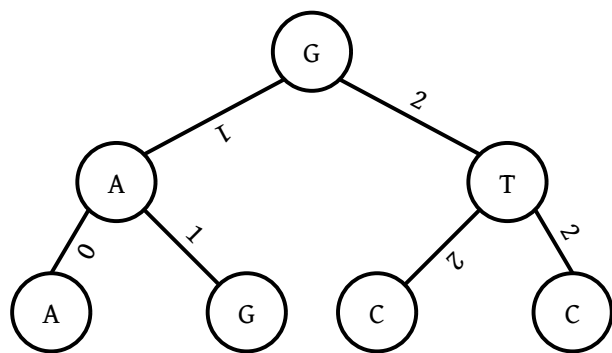
Si supponga di avere a disposizione 4 specie ed un carattere X, del quale é nota la matrice di score. Sia poi data la topologia dell'albero filogenetico per queste:



	X
S1	A
S2	G
S3	T
S4	T

	A	G	C	T
A	0	1	3	2
G	1	0	1	2
C	3	1	0	2
T	2	2	2	0

Di seguito sono presentati due possibili alberi, entrambi potenzialmente validi, etichettati dai valori del carattere X che gli antenati delle specie avevano.



La somma di tutti gli score riportati sugli archi dell'albero a sinistra é 7, mentre quella dell'albero a destra é 6. Questo significa che, assumendo come valido il principio di massima parsimonia, l'albero a destra é una migliore ricostruzione della filogenesi di tali specie rispetto a quello di sinistra, ed é pertanto quello fra i due da preferire. Si noti come questo non significhi né che l'albero a destra sia il migliore possibile e né tantomeno che sia quello che descriva come la filogenesi sia effettivamente avvenuta.

Il problema può essere risolto mediante programmazione dinamica. L'ultima componente della soluzione é determinare quale stato vada assegnato all'ultimo nodo dell'albero filogenetico in modo che il peso complessivo dell'albero sia minimizzato (assumendo che gli altri nodi siano già stati etichettati di modo che il peso complessivo sia ottimale). Dato che le etichette dei nodi foglia sono note, l'algoritmo che risolve il problema dovrà necessariamente iniziare dalle foglie, e l'ultimo nodo sarà quindi la radice.

L'interesse é quindi quello di trovare qual'é lo stato del carattere che etichetta la radice di modo che l'albero che ne risulti sia ottimale. Se vi sono v_1, v_2, \dots, v_k nodi con i quali la radice ha un arco, e ciascun sottoalbero $T_{v1}, T_{v2}, \dots, T_{vk}$ che ha tali nodi per radice é già stato etichettato in maniera ottimale, il miglior valore da assegnare alla radice é quello che minimizza la somma fra il peso complessivo di ciascun sottoalbero e il peso che si ha dal passare dalla radice ai nodi v_1, v_2, \dots, v_k .

L'algoritmo basato su programmazione dinamica che risolve il problema di piccola parsimonia così formulato é l'**algoritmo di Sankoff**. Lo scopo dell'algoritmo é quello di costruire una matrice P , dove ciascuna cella $P[x, z]$ contiene la soluzione ottimale del sottoalbero T_x se x viene etichettato con il valore z .

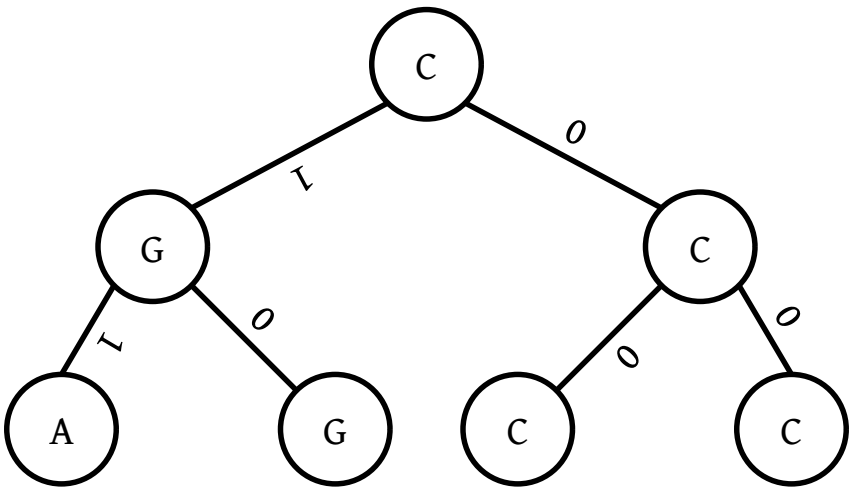
Se $v \in T$ é un nodo foglia, allora il peso complessivo di T_v é sempre zero, a prescindere da quale sia l'etichettatura, perché T_v non contiene alcun arco. In questo problema le etichette dei nodi foglia sono note, pertanto se x é un nodo foglia l'unico valore di $P[x, z]$ che é ragionevole scegliere é quello per il quale z é l'effettivo stato con cui x é etichettato, perché tutte le altre possibilità non possono verificarsi. Si ha allora che $P[x, z] = 0$ se z é l'effettivo stato che etichetta x , mentre si ha (simbolicamente) $P[x, z] = +\infty$ se z non é l'etichetta di x .

$$P[x, z] = \begin{cases} 0 & \text{se } z = \lambda(x) \\ +\infty & \text{se } z \neq \lambda(x) \end{cases}$$

Il valore di $P[x, z]$ dove x non é un nodo qualsiasi dipende sia da quale etichetta si sceglie per x , sia dalle etichette che sono state assegnate (nelle precedenti iterazioni) ai nodi di T_x . Se il nodo x ha L nodi figli v_1, v_2, \dots, v_L , il valore ottimale con cui etichettare x é quello che minimizza la somma fra $P[v_i, s]$, il peso ottimale del sottoalbero T_{v_i} se v_i viene etichettato con s , e $w(z, s)$, il costo che la mutazione dallo stato z allo stato s comporta. Ovvero:

$$P[x, z] = \sum_{i=1}^L \min_s (w(z, s) + P[v_i, s])$$

	A	C	G	T
P1	0	+ ∞	+ ∞	+ ∞
P2	+ ∞	+ ∞	0	+ ∞
P3	+ ∞	0	+ ∞	+ ∞
P4	+ ∞	0	+ ∞	+ ∞
P5	1	4	1	4
P6	6	0	2	4
P7	3	2	2	5

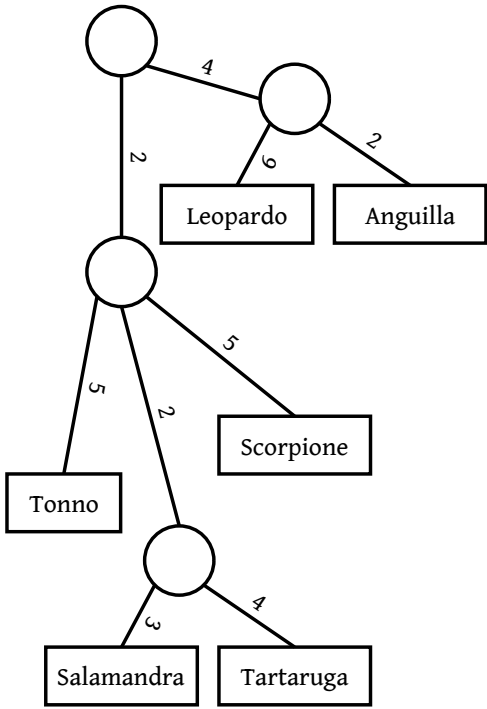


Indicando con k il numero di stati del carattere e con n il numero di nodi dell'albero, la matrice avrà $O(nk)$ caselle, per ciascuna delle quali occorre fare k somme. Il tempo di esecuzione complessivo dell'algoritmo é allora $O(nk^2)$.
L'algoritmo può essere esteso ad una situazione in cui si stanno considerando m caratteri distinti, ciascuno con la propria matrice di score, anziché uno solo, ammesso che la topologia dell'albero sia la stessa a prescindere dal carattere. Infatti, é sufficiente eseguire l'algoritmo m volte, con m alberi distinti, che possono poi venire unificati in uno solo. L'unificazione viene effettuata riportando, per ciascun nodo v_f dell'albero risultante, come etichetta la concatenazione delle etichette del nodo di ciascuno degli m alberi parziali che si trova in quella posizione. Questo é possibile perché, avendo gli m alberi la stessa topologia, hanno gli stessi nodi e gli stessi archi nelle stesse esatte posizioni, e quindi sono perfettamente "sovrapponibili".

5.4 Grande problema di parsimonia

Il **grande problema di parsimonia** é una generalizzazione del piccolo problema di parsimonia; é uguale a quest'ultimo in tutto e per tutto ma introduce l'ulteriore difficoltà di non avere a disposizione la topologia dell'albero T . Pertanto, il grande problema di parsimonia richiede sia di ricostruire la filogenesi di un insieme di specie, sia di dedurre lo stato dei caratteri dei loro antenati. Questo problema, se approcciato mediante il metodo di massima parsimonia, diviene un problema NP, quindi impossibile da modellare in un tempo ragionevole. Per questo motivo, si preferisce attaccare il problema a partire dalla distanza fra stringhe.
Sia dato un insieme di specie $S = \{S_1, S_2, ..., S_n\}$ delle quali é noto il sequenziamento del genoma. É possibile sfruttare la sequenza genomica delle specie in S per dedurre la filogenesi: infatti, se si assume che specie simili abbiano genomi simili, allora due specie filogeneticamente lontane avranno le rispettive stringhe genomiche distanti, e viceversa. Questo tipo di approccio al problema della ricostruzione della filogenesi prende il nome di **filogenesi basata su distanze**.
La distanza fra le specie (ovvero, la distanza fra la stringa di DNA dell'una e la stringa di DNA dell'altra) può essere espressa sia mediante una matrice, sia mediante un albero. Una matrice di distanze é una matrice che riporta in ciascuna cella (i, j) la distanza fra la stringa genomica di S_i e la stringa genomica di S_j . Un albero delle distanze é un albero dove le etichette dell'arco che va dal nodo v_i al nodo v_j corrisponde alla distanza fra la stringa genomica della specie (non necessariamente vivente) che etichetta v_i e la stringa genomica della specie (non necessariamente vivente) che etichetta v_j .

	0	11	20	20	21	20
	11	0	13	13	14	13
	20	13	0	10	7	10
	20	13	10	0	11	10
	21	14	7	11	0	11
	20	13	10	10	11	0



Noto un albero delle distanze, é possibile ricavare la distanza fra una specie S_i ed una specie S_j sommando le etichette del cammino che va da S_i a S_j . Ripetendo il procedimento per tutte le specie, É possibile costruire una matrice di distanze equivalente. Si noti però come il contrario non sia possibile, dato che esistono infiniti alberi delle distanze, tutti potenzialmente corretti, equivalenti ad una matrice delle distanze. Non tutte le distanze riportate su un albero possono considerarsi valide. Verrá studiato un caso particolare in cui vale un assunto chiamato **teoria dell'orologio molecolare**.

La **teoria dell'orologio molecolare** é una assunzione sulla natura delle mutazioni delle biomolecole, secondo la quale il numero di mutazioni positive che avvengono in una biomolecola in un certo intervallo di tempo é (circa) proporzionale alla durata dell'intervallo stesso. Pertanto, se sono note quante mutazioni sono presenti nella biomolecola di una specie A che non sono presenti nella biomolecola di una specie B (e viceversa), ed é nota la frequenza con cui avvengono le mutazioni nella biomolecola di A e di B a paritá di tempo, é possibile dare una stima di quanto tempo é passato da quando A e B hanno subito una divergenza evolutiva. La distanza fra la stringa della biomolecola delle due specie diventa allora una buona stima del numero di mutazioni avvenute nell'una che non sono presenti nell'altra ⁴.

5.4.1 Ultrametricitá

Se la teoria dell'orologio molecolare é valida, allora l'albero delle distanze ha una proprietá particolare: preso un qualsiasi nodo interno dell'albero, la somma dei valori riportati sulle etichette del cammino che inizia in tale nodo e termina in una qualsiasi foglia sua discendente ha sempre lo stesso valore. Un albero con questa proprietá prende il nome di **albero ultrametrico**, la distanza cosí definita é detta **distanza ultrametrica** e la matrice associata all'albero é detta **matrice ultrametrica**.

L'albero precedente, e di conseguenza la matrice precedente, non erano ultrametriche. Lo sono invece, ad esempio, le rispettive versioni modificate riportate di seguito:

	0	6	14	12	14	14
	6	0	14	14	14	14
	14	14	0	10	6	10
	12	14	10	0	10	10
	14	14	6	10	0	10
	14	14	10	10	10	0

La distanza ultrametrica ha, oltre a tutte le proprietá della distanza, una proprietá aggiuntiva. Presi tre elementi a, b e c , il massimo fra la distanza da a a b , la distanza da b a c e la distanza da c ad a deve sempre coincidere con almeno due di queste tre:

$$\max(d(a, b), d(b, c), d(c, a)) = \begin{cases} \{d(a, b), d(b, c)\} \text{ oppure} \\ \{d(b, c), d(c, a)\} \text{ oppure} \\ \{d(a, b), d(c, a)\} \text{ oppure} \\ \{d(a, b), d(b, c), d(c, a)\} \end{cases} \quad \forall a, b, c$$

Le matrici ultrametriche hanno la proprietá di avere associate uno ed un solo albero ultrametrico, costruibile a partire da questa per via ricorsiva. Sia D la matrice ultrametrica a disposizione e sia T l'albero ultrametrico desiderato. L'algoritmo banale, che é possibile dimostrare avere tempo di esecuzione $O(n^2)$, é presentato di seguito:

1. Si cerchi il valore piú piccolo presente nella matrice ultrametrica; si assuma che questo si trovi alle coordinate i, j . Allora l'albero ultrametrico deve avere due nodi foglia etichettati con le specie i e j , e la distanza complessiva fra i due é $D[i, j]$. Vi sono quattro possibilitá:
4. Prima della diffusione delle genoteche, questo tipo di stima veniva compiuta mediante metodi chimico-fisici. Si preleva il DNA di A e di B, lo si denatura in modo che i due filamenti di ciascuno si separino, uno dei filamenti di A viene ibridato con uno dei filamenti di B e dopodiché si determina quanta energia é necessaria affinché il DNA ibrido cosí ottenuto si denaturi. L'idea é che piú il DNA di A e di B é dissimile, ovvero piú mutazioni non sono presenti in ambo le specie, piú sará forte la loro ibridazione e tanto piú alta sará la temperatura necessaria a denaturare il DNA ibrido.

- Sia la specie i che la specie j non sono presenti nell'albero. Occorre allora aggiungere due nodi foglia etichettati rispettivamente con i e j ed un nodo interno v avente un arco con entrambi. Sia l'arco che va da v ad i che quello che va da v a j devono avere per etichetta $D[i, j] / 2$. Questo perché, dovendo l'albero essere ultrametrico, la somma totale delle distanze sui cammini che vanno da un nodo interno a tutte le sue foglie deve essere la stessa, ed avendo v soltanto due foglie la distanza é equamente divisa fra le due;
 - La specie i é già presente nell'albero, ma non j . Allora occorre aggiungere un nodo foglia etichettato j ed un nodo interno v che ha un arco con j etichettato con $D[i, j]$. Deve inoltre esistere, avendo i e j un antenato comune, un cammino composto da almeno due archi che va da v a i , che passerá per un certo numero di antenati di i . Affinché l'albero sia ultrametrico, la somma delle etichette degli archi di tale cammino deve essere $D[i, j]$. Sia u l'antenato di i piú lontano: il cammino che va da v a i sará allora composto da un arco che va da v ad u e dal cammino che va da u a i . Affinché l'albero sia ultrametrico, l'etichetta dell'arco che va da v ad u deve essere data dalla differenza fra $D[i, j]$ e la somma di tutte le etichette degli archi del cammino da u a i ;
 - La specie j é già presente nell'albero, ma non i . La casistica é analoga alla precedente, ma a parti invertite;
 - Sia la specie i che la specie j sono già presenti nell'albero. Occorre allora aggiungere un nodo interno v che ha un cammino composto da almeno due archi sia con i che con j etichettato con $D[i, j]$. Siano u e w rispettivamente gli antenati di i e di j piú lontani. Il cammino che va da v a i sará composto da un arco che va da v ad u e dal cammino che va da u a i ; allo stesso modo, il cammino che va da v a j sará composto da un arco che va da v ad w e dal cammino che va da w a j . Affinché l'albero sia ultrametrico, l'etichetta dell'arco che va da v ad u deve essere data dalla differenza fra $D[i, j]$ e la somma di tutte le etichette degli archi del cammino da u a i ; allo stesso modo, l'etichetta dell'arco che va da v ad w deve essere data dalla differenza fra $D[i, j]$ e la somma di tutte le etichette degli archi del cammino da w a j .
2. Si ripeta l'algoritmo escludendo la i -esima riga o la j -esima colonna di D (la matrice é simmetrica, pertanto é indifferente);
 3. L'algoritmo termina quando sono state esaurite tutte le righe/colonne. Si noti come possano essere presenti dei nodi in T aventi collegamenti etichettati con 0. Se questo accade, é possibile semplicemente unificarli in un solo nodo.

5.4.2 Additività

In un contesto reale, é sostanzialmente impossibile che i dati raccolti sulle mutazioni a cui le specie vanno incontro possano essere disposti in una matrice perfettamente ultrametrica, perché l'ipotesi di ultrametricità é estremamente stringente. É possibile rilassare tale ipotesi e rifarsi ad una sua forma piú debole, l'ipotesi di **additività**.

Una matrice delle distanze D si dice **matrice additiva** se soddisfa la **condizione dei quattro punti**. Presi 4 elementi qualsiasi x, y, v e w sulla matrice D , si considerino tutte le possibili celle di D definibili da tali punti, ovvero $D[x, y], D[x, v], D[x, w], D[y, w], D[y, v], D[v, w]$. Si considerino tutte le somme costruibili usando due di queste sei celle tali per cui tutti e quattro i punti compaiono esattamente una volta, ovvero queste tre:

$$D[x, y] + D[v, w]$$

$$D[x, v] + D[y, w]$$

$$D[x, w] + D[y, v]$$

La condizione dei quattro punti é valida se due di questi tre valori sono uguali e sono anche il massimo fra questi:

$$\begin{aligned} & \{D[x, y] + D[v, w]\} \text{ oppure} \\ \max(D[x, y] + D[v, w], D[x, v] + D[y, w], D[x, v] + D[y, w]) &= \{D[x, v] + D[y, w]\} \text{ oppure } \forall x, y, v, w \\ & \{D[x, v] + D[y, w]\} \end{aligned}$$

Se una matrice D é additiva, allora l'albero delle distanze T a questa associato é detto **albero additivo**, e la distanza definita da D (e da T) é detta **distanza additiva**. Essendo l'additività una proprietà piú debole dell'ultrametricità, si ha che qualsiasi matrice ultrametrica é anche additiva, qualsiasi distanza ultrametrica é anche additiva e qualsiasi albero ultrametrico é anche additivo.

La condizione dei quattro punti può essere definita in maniera equivalente anche rispetto agli alberi additivi. Siano x, y, v, w quattro nodi foglia dell'albero additivo T , associato alla matrice additiva D . Per la relazione che sussiste fra D e T , la distanza totale fra due di questi nodi i e j , ovvero la somma di tutte le etichette del cammino che va da i a j , corrisponde a $D[i, j]$. Si considerino allora tutte le possibili distanze totali fra due dei quattro possibili nodi: quella fra x e y (ovvero $D[x, y]$), quella fra v e w (ovvero $D[v, w]$), ecc... Si considerino tutte le somme costruibili usando due di queste sei distanze totali tali per cui tutti e quattro i nodi compaiono esattamente una volta, ovvero queste tre:

$$d(x, y) + d(v, w)$$

$$d(x, v) + d(y, w)$$

$$d(x, w) + d(y, v)$$

La condizione dei quattro punti é valida se due di questi tre valori sono uguali e sono anche il massimo fra questi. Si noti però come l'albero additivo non ha una radice, a differenza di quello ultrametrico, pertanto non ha una direzione di percorrenza privilegiata.

Prima di introdurre l'algoritmo per la costruzione di un albero additivo, occorre fare delle considerazioni. Innanzitutto, si considerino tre nodi generici x, y e z di un albero delle distanze (non per forza additivo o ultrametrico). Dato che la disuguaglianza triangolare é sempre valida, deve aversi che $d(x, y) \leq d(y, z) + d(x, z)$, ovvero che $d(y, z) + d(x, z) - d(x, y) \geq 0$. La quantità $d(y, z) + d(x, z) - d(x, y)$ prende il nome di **sbilancio** della tripla (x, y, z) .

Una tripla (x, y, z) di nodi su un albero delle distanze si dice **degenere** se il loro sbilancio é nullo, ovvero se $d(y, z) + d(x, z) = d(x, y)$. É facile verificare come questa situazione possa presentarsi, ad esempio, nel caso in cui z é un nodo interno al cammino che va da x a y . Se z' é un nodo raggiungibile a partire da z mediante un cammino che ha peso totale 0, ovvero se vale $d(z, z') = 0$, e se (x, y, z) é una tripla degenere, allora anche x, y, z' é degenere. Infatti:

$$\begin{aligned} d(y, z') + d(x, z') - d(x, y) &= (d(y, z) + d(z, z')) + (d(x, z) + d(z, z')) - d(x, y) = \\ &= (d(y, z) + 0) + (d(x, z) + 0) - d(x, y) = d(y, z) + d(x, z) - d(x, y) = 0 \end{aligned}$$

Si considerino gli archi di un albero delle distanze che uniscono un nodo foglia ad un nodo interno, ciascuno etichettato con una certa distanza. Sia δ un numero intero positivo; se si diminuisce di δ il valore riportato su ciascuna etichetta, si ha che ciascun cammino sull'albero che inizia da un nodo foglia e finisce in un altro nodo foglia sarà ridotto di 2δ , perché ciascuno di questi cammini deve attraversare esattamente due archi il cui valore dell'etichetta è stato diminuito di δ . Ma dato che il cammino fra due nodi foglia x e y corrisponde alla cella (x, y) sulla matrice delle distanze, dovrà aversi che tale matrice, se si diminuisce ciascun arco dell'albero delle distanze che unisce un nodo foglia ad un nodo interno di una quantità δ , avrà il valore in ciascuna cella diminuito di 2δ .

5.5 Filogenesi mediante clustering

Prende il nome di **algoritmo di clustering** un algoritmo che raggruppa gli elementi di un certo insieme in un certo numero di **clusters**. Un cluster è un sottoinsieme di un insieme più grande che rispetta due proprietà: *omogeneità* e *separazione*. La prima prevede che gli elementi di un cluster siano fra loro molto simili, mentre la seconda prevede che gli elementi di cluster siano molto diversi dagli elementi di tutti gli altri cluster. Un particolare modo di intendere il clustering è il **clustering gerarchico**, che ha una forte connessione con le strutture ad albero. L'idea del clustering gerarchico è quella di, dato un certo insieme di elementi, costruire un certo insieme di cluster, costruire dei cluster che inglobano questi cluster, e così via. Un clustering gerarchico può essere convertito in un albero semplicemente notando come i cluster più esterni, che contengono i più interni, corrispondono ai nodi dell'albero più in alto, da cui discendono i nodi in basso. In particolare: il cluster più grande è la radice, i suoi sottocluster sono i nodi di primo livello, i rispettivi sottocluster dei sottocluster sono i nodi di secondo livello, e via dicendo.

Il clustering, ed in particolare quello gerarchico, può essere applicato anche alla ricostruzione della filogenesi: due specie fanno parte di uno stesso ipotetico cluster se la distanza delle rispettive stringhe genomiche è piccola, mentre faranno parte di cluster diversi se la distanza delle rispettive stringhe genomiche è grande. Ad ogni cluster così costruito può venire assegnato un nuovo valore di distanza sulla base della distanza dei suoi elementi (in genere, una qualche forma di media pesata), che funge da "centro di massa" del cluster, e sulla base di questi i cluster possono essere inglobati in supercluster.

Se è possibile costruire un clustering gerarchico a partire da un insieme di specie, è poi possibile risolvere il problema della filogenesi semplicemente convertendo tale clustering in un albero, che per il modo in cui è stato definito sarà un albero filogenetico. Gli algoritmi che costruiscono un clustering gerarchico a partire da un insieme di specie sono molto simili, e le uniche due notevoli differenze si hanno nel metodo utilizzato per determinare quale sia la coppia di cluster candidata a venire fusa e come calcolare il "centro di massa" di ciascun cluster. Due algoritmi molto noti sono **UPGMA** e **Neighbor-Joining**.

5.5.1 UPGMA

L'algoritmo **UPGMA (unweighted pair group with arithmetic mean)** è un algoritmo di clustering molto semplice, che a partire dal clustering costruisce un albero ultrametrico. L'algoritmo si basa sul definire una distanza per ciascuna coppia di cluster (C_1, C_2) mediante la seguente formula:

$$D(C_1, C_2) = \frac{1}{|C_1| + |C_2|} \sum_{i \in C_1} \sum_{j \in C_2} D(i, j) = \frac{D(i_1, j_1) + \dots + D(i_1, j_n) + \dots + D(i_m, j_1) + \dots + D(i_m, j_n)}{|C_1| + |C_2|}$$

Dato un certo insieme di specie S di cui è nota la sequenza genomica e data una matrice delle distanze ultrametriche D , sia T l'albero ultrametrico che si vuole ottenere. L'algoritmo può essere descritto come segue:

1. Ciascuna specie S_1, S_2, \dots, S_n forma un cluster che contiene solo sé stessa, dove ciascuno corrisponde ad un nodo foglia dell'albero ultrametrico. Si aggiungano allora a T (inizialmente vuoto) i nodi foglia etichettati da tali cluster;
2. Ad ogni nodo $v \in T$ così aggiunto viene assegnato un valore $h(v) = 0$, che rappresenta l'altezza di tale nodo;
3. Si trovino i due cluster C_1 e C_2 che hanno minima distanza fra loro, e si fondano in un unico cluster C avente $|C_1| + |C_2|$ elementi;
4. Si calcoli la distanza fra ciascun cluster ed il nuovo cluster C così costruito;
5. Si aggiunga C ai nodi dell'albero, e lo si colleghi a C_1 e a C_2 mediante degli archi. C rappresenta l'antenato comune da cui le specie C_1 e C_2 si sono separate;
6. Si assegni a $h(C)$ il valore $D(C_1, C_2) / 2$;
7. Si etichetti l'arco che unisce il nodo C_1 al nodo C con $h(C) - h(C_1)$. Allo stesso modo, si etichetti l'arco che unisce il nodo C_2 al nodo C con $h(C) - h(C_2)$;
8. Si scartino le righe e le colonne di D relative a C_1 e a C_2 , e se ne aggiunga una per il nuovo cluster C ;
9. Se vi sono almeno due cluster rimasti, l'algoritmo ricomincia a partire dal punto 3, altrimenti termina.

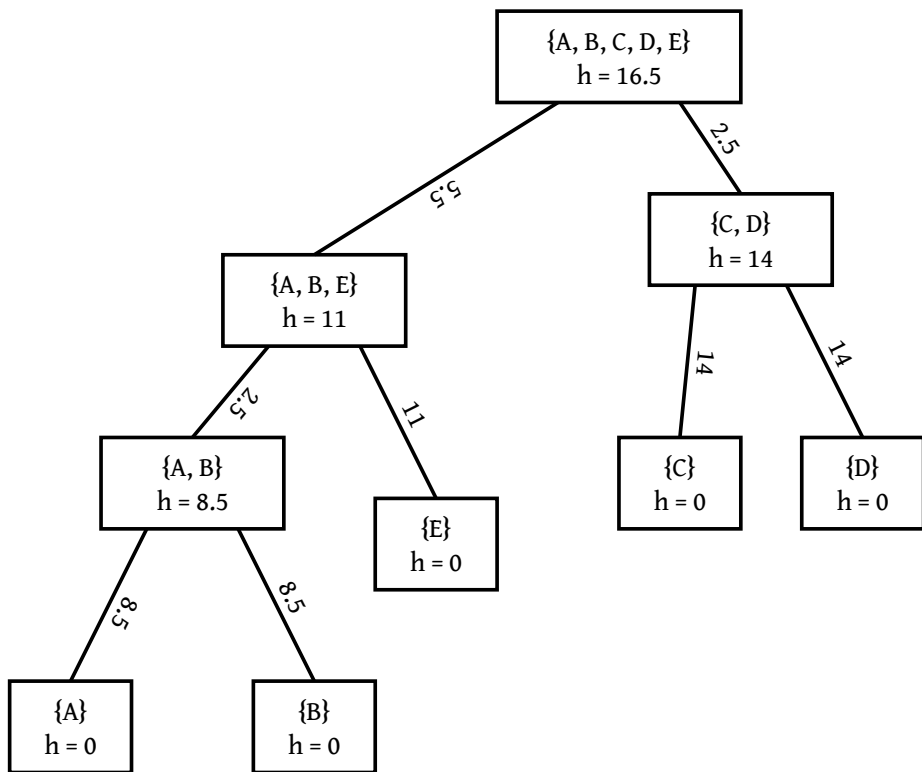
	A	B	C	D	E
A	0	17	21	31	23
B	17	0	30	34	21
C	21	30	0	28	39
D	31	34	28	0	43
E	23	21	39	43	0

$$D(AB, C) = \frac{D(A, C) + D(B, C)}{|\{A, B\}| + |\{C\}|} = \frac{21 + 30}{2 + 1} = 25.5$$

$$D(AB, D) = \frac{D(A, D) + D(B, D)}{|\{A, B\}| + |\{D\}|} = \frac{31 + 34}{2 + 1} = 32.5$$

$$D(AB, E) = \frac{D(A, E) + D(B, E)}{|\{A, B\}| + |\{E\}|} = \frac{23 + 21}{2 + 1} = 22$$

$$h(AB) = \frac{D(A, B)}{2} = \frac{17}{2} = 8.5$$



	AB	C	D	E
AB	0	25.5	32.5	22
C	25.5	0	28	39
D	32.5	28	0	43
E	22	39	43	0

5.5.2 Neighbor Joining

Neighbor Joining é una estensione di UPGMA che produce un albero additivo anziché ultrametrico. Neighbor Joining introduce un vincolo ulteriore sul come scegliere due cluster da fondere: anziché semplicemente fondere due cluster vicini, l'algoritmo fonde due cluster fra loro vicini ma al contempo lontani da tutti gli altri. Sia k il numero di cluster; il parametro che esprime la "lontananza" di un cluster C da tutti gli altri é indicato con $u(C)$:

$$u(C) = \frac{1}{k-2} \sum_{C_i} D(C, C_i)$$

Nella scelta di quale cluster fondere, Una prima ipotesi sarebbe quella di cercare la coppia di cluster (C_1, C_2) che contemporaneamente minimizzano $D(C_1, C_2)$ e massimizzano $u(C_1) + u(C_2)$, ma é molto difficile che vi siano effettivamente una coppia di cluster con tali caratteristiche. Un migliore approccio consiste nel minimizzare la quantità $D(C_1, C_2) - u(C_1) - u(C_2)$.

Dato un certo insieme di specie S di cui é nota la sequenza genomica e data una matrice delle distanze ultrametriche D , sia T l'albero additivo che si vuole ottenere. L'algoritmo può essere descritto come segue:

1. Ciascuna specie S_1, S_2, \dots, S_n forma un cluster che contiene solo sé stessa, dove ciascuno corrisponde ad un nodo foglia dell'albero ultrametrico. Si aggiungano allora a T (inizialmente vuoto) i nodi foglia etichettati da tali cluster;
2. Si calcoli il valore $u(C_i)$ per ciascun cluster C_i ;
3. Si costruisca una matrice Q che contiene i valori $D(C_1, C_2) - u(C_1) - u(C_2)$ per tutte le coppie di cluster (C_1, C_2) ;
4. Si trovino in Q i due cluster C_1 e C_2 che minimizzano la quantità $D(C_1, C_2) - u(C_1) - u(C_2)$ e si fondano in un unico cluster C avente $|C_1| + |C_2|$ elementi;
5. Per qualsiasi cluster $C^* \neq C$, si ha $D(C, C^*) = (D(C_1, C^*) + D(C_2, C^*) - D(C_1, C_2)) / 2$;
6. Si aggiunga C ai nodi dell'albero, e lo si colleghi a C_1 e a C_2 mediante degli archi. C rappresenta l'antenato comune da cui le specie C_1 e C_2 si sono separate;
7. Si etichetti l'arco che unisce il nodo C_1 al nodo C con $(D(C_1, C_2) + u(C_1) - u(C_2)) / 2$. Allo stesso modo, si etichetti l'arco che unisce il nodo C_2 al nodo C con $(D(C_1, C_2) - u(C_1) + u(C_2)) / 2$;
8. Si scartino le righe e le colonne di D relative a C_1 e a C_2 , e se ne aggiunga una per il nuovo cluster C ;

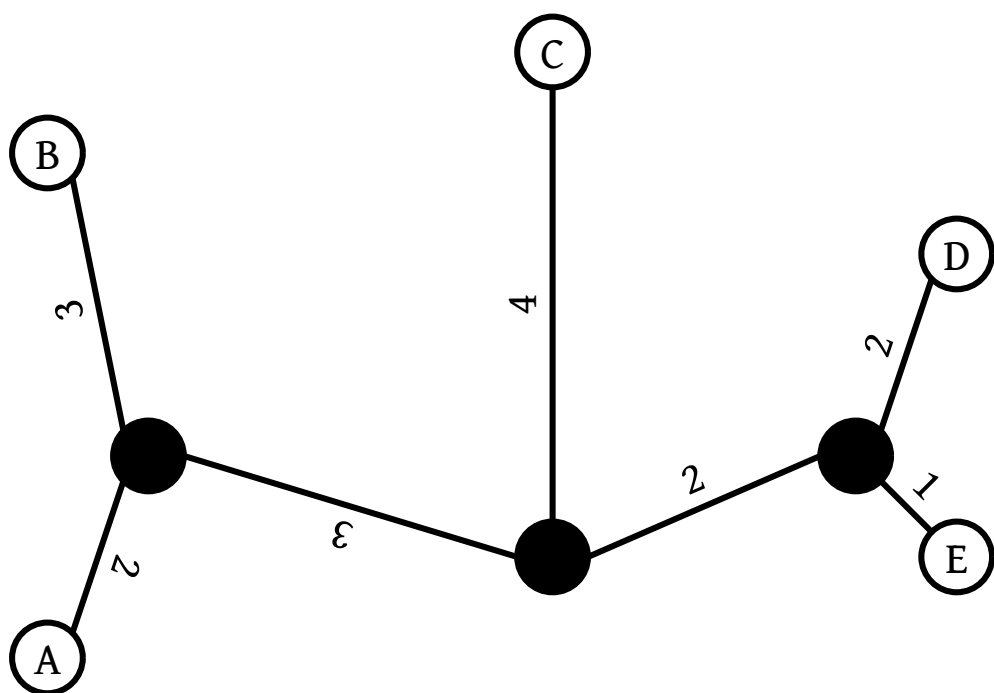
9. Se vi sono almeno due cluster rimasti, l'algoritmo ricomincia a partire dal punto 2, altrimenti termina.

	A	B	C	D	E
A	0	5	9	9	8
B	5	0	10	10	9
C	9	10	0	8	7
D	9	10	8	0	3
E	8	9	7	3	0

A	10.3
B	11.3
C	11.3
D	10
E	9

	A	B	C	D	E
A	/	-16.6	-12.6	-11.3	-11.3
B	-16.6	/	-12.6	-11.3	-11.3
C	-12.6	-12.6	/	-13.3	-13.3
D	-11.3	-11.3	-13.3	/	-16
E	-11.3	-11.3	-13.3	-16	/

$$\begin{aligned} u(A) &= \frac{\sum_{C_i} D(A, C_i)}{5-2} = \frac{D(A, B) + D(A, C) + D(A, D) + D(A, E)}{3} = \frac{5 + 9 + 9 + 8}{3} = \frac{31}{3} \approx 10.3 \\ u(B) &= \frac{\sum_{C_i} D(B, C_i)}{5-2} = \frac{D(B, A) + D(B, C) + D(B, D) + D(B, E)}{3} = \frac{5 + 10 + 10 + 9}{3} = \frac{34}{3} \approx 11.3 \\ u(C) &= \frac{\sum_{C_i} D(C, C_i)}{5-2} = \frac{D(C, A) + D(C, B) + D(C, D) + D(C, E)}{3} = \frac{9 + 10 + 8 + 7}{3} = \frac{34}{3} \approx 11.3 \\ u(D) &= \frac{\sum_{C_i} D(D, C_i)}{5-2} = \frac{D(D, A) + D(D, B) + D(D, C) + D(D, E)}{3} = \frac{9 + 10 + 8 + 3}{3} = \frac{30}{3} = 10 \\ u(E) &= \frac{\sum_{C_i} D(E, C_i)}{5-2} = \frac{D(E, A) + D(E, B) + D(E, C) + D(E, D)}{3} = \frac{8 + 9 + 7 + 3}{3} = \frac{27}{3} = 9 \end{aligned}$$



	AB	C	D	E
AB	0	7	7	6
C	7	0	8	7
D	7	8	0	3
E	6	7	3	0

Dato un insieme di n specie, l'algoritmo Neighbor Joining applicato a tale insieme richiede $n-3$ iterazioni. In ciascuna iterazione occorre costruire la matrice Q , la cui dimensione é $n \times n$ per la prima iterazione, $(n-1) \times (n-1)$ per la seconda, ecc..., e sulla quale occorre cercare il valore minimo. Si ha allora che Neighbor Joining, se implementato in maniera banale, ha un tempo di esecuzione $O(n^3)$.

5.6 Modelli di evoluzione

Nonostante siano noti i meccanismi che stanno dietro le mutazioni del DNA (deriva genetica, selezione naturale, ecc...), studiare come ed in che ordine queste si verificano é tutt'altro problema. Un modello statistico che descriva il rateo di mutazioni che avvengono nel DNA in funzione del tempo viene chiamato **modello di mutazione**.

5.6.1 Modello di Jukes-Cantor

Il modello di mutazione piú semplice possibile, chiamato **modello di Jukes-Cantor**, assume che una base del DNA possa subire una mutazione e venire sostituita con un'altra base (diversa da sé stessa) con la stessa probabilità μ . La probabilità che una base venga sostituita con una

base specifica é $\mu / 3$, essendo tutte e tre le possibilità equiprobabili, mentre la probabilità che questa non subisca una mutazione é $1-\mu$, perché é l'evento contrario all'evento "subire una mutazione qualsiasi":

	A	C	G	T
A	$1-\mu$	$\mu / 3$	$\mu / 3$	$\mu / 3$
C	$\mu / 3$	$1-\mu$	$\mu / 3$	$\mu / 3$
G	$\mu / 3$	$\mu / 3$	$1-\mu$	$\mu / 3$
T	$\mu / 3$	$\mu / 3$	$\mu / 3$	$1-\mu$

5.6.2 Modello K2P

Il modello di Jukes-Cantor é troppo semplicistico per descrivere la relazione fra mutazioni del DNA e tempo trascorso. Ad esempio, al verificarsi di una **transizione** (la sostituzione di una purina con un'altra purina, o di una pirimidina con un'altra pirimidina) o di una **trasversione** (la sostituzione di una purina con una pirimidina, o di una pirimidina con una purina) viene assegnata la stessa probabilità, mentre é piú ragionevole assumere che le due siano diverse (a causa della loro conformazione, é molto piú probabile che una purina venga sostituita da un'altra purina piuttosto che da una pirimidina, e viceversa).

Il **modello K2P** é una estensione del modello di Jukes-Cantor, dove la probabilità che una base muti é sempre la stessa, ma vengono assegnate due diverse probabilità al verificarsi di una transizione o di una trasversione. Sia μ la probabilità che si verifichi una mutazione qualsiasi, e sia R il rapporto fra la probabilità che si verifichi una transizione e la probabilità che si verifichi una trasversione. La probabilità che una base non subisca una mutazione rimane $1-\mu$, ma la probabilità che avvenga una transizione piuttosto che una trasversione differisce:

	A	C	G	T
A	$1-\mu$	$R / R + 1$	$1 / 2(R + 1)$	$1 / 2(R + 1)$
C	$R / R + 1$	$1-\mu$	$1 / 2(R + 1)$	$1 / 2(R + 1)$
G	$1 / 2(R + 1)$	$1 / 2(R + 1)$	$1-\mu$	$R / R + 1$
T	$1 / 2(R + 1)$	$1 / 2(R + 1)$	$R / R + 1$	$1-\mu$

5.6.3 Modello General Time-Reversible

Il modello **General Time-Reversible** é ancora piú raffinato dei precedenti, perché prevede di assegnare una probabilità univoca a ciascun tipo di mutazione, mantenendo però la simmetricità delle probabilità (la probabilità che i muti in j é la stessa che j muti in i , per qualsiasi coppia di basi (i, j)).

	A	C	G	T
A	α_1	α_2	α_3	α_4
C	α_2	χ_1	χ_2	χ_3
G	α_3	χ_2	γ_1	γ_2
T	α_4	χ_3	γ_2	τ_1

Il problema di questo modello é che se utilizzato per costruire alberi filogenetici, questi sono senza radice esplicita, mentre ci si aspetta che un albero filogenetico la possenga (dovendo descrivere un processo che ha un chiaro "prima" ed un chiaro "dopo").

Una soluzione molto semplice per trovare quale nodo dell'albero é il miglior candidato per essere la radice consiste nell'introdurre nell'insieme delle specie da analizzare una specie "spuria", chiamata **outgroup**, che non ha alcuna relazione con le altre specie dell'insieme. Dato che questa specie verrà certamente a trovarsi nell'albero da tutt'altra parte rispetto alle altre, può ben fungere da radice. Inoltre, ha anche una funzione di controllo incrociato, dato che se l'outgroup viene a trovarsi troppo vicino alle altre specie nell'albero filogenetico si ha un segnale evidente che é stato commesso un errore di classificazione.

Capitolo 6

Sequenziamento del DNA

6.1 Assemblaggio del DNA

Il **sequenziamento** di una biomolecola consiste nel determinare l'ordine degli elementi di base di cui é composta. Ad esempio, nel caso delle proteine questo significa determinare di quali amminoacidi é composta e qual'é la loro disposizione nello spazio, mentre nel caso del DNA significa ricavare l'ordine in cui sono disposte le basi azotate.

Ci si limiti, per semplicitá, al solo sequenziamento del DNA. Si noti come, a differenza ad esempio delle proteine, ogni specie ha un DNA sempre diverso da tutte le altre. Inoltre, nella maggior parte dei casi, ogni individuo di una stessa specie ha un DNA leggermente diverso dai suoi simili (e questo si riflette nelle loro differenze fenotipiche). Pertanto, non solo per ogni specie vivente occorre operare un proprio sequenziamento, ma questo deve anche tenere conto del fatto che esiste un certo "margine" di differenza ammesso fra individui della stessa specie.

La tecnologia attualmente esistente non é in grado di sequenziare l'intero genoma di una specie in un solo passaggio. Ciò che é in grado di fare in un tempo accettabile e con bassa probabilità di commettere un errore di lettura é sequenziare per intero piccoli frammenti di DNA, chiamati **read**, lunghi dalle 50 alle 10000 coppie di basi (l'intero genoma umano é composto da circa 3×10^8 basi). Non solo queste read sono molto piú piccole dell'intero DNA, ma non é nemmeno possibile sapere da quale regione del DNA sono state estratte. Infine, gli organismi diploidi (come l'essere umano) possiedono tutti i loro cromosomi in doppia copia, ciascuna ereditata da uno dei genitori, pertanto ciascuna read di uno stesso cromosoma potrebbe essere relativa indifferentemente ad un cromosoma piuttosto che all'altro.

Sebbene gli algoritmi per il sequenziamento del DNA siano fra loro molto variegati, tutti quanti si basano sugli stessi tre passaggi: **overlap**, **layout**, **consensus**.

- **Overlap.**

Avendo a disposizione un insieme di read, occorre determinare, per ciascuna coppia ordinata di read, quanto un suffisso della prima "combaci" con un prefisso della seconda, tenendo presente che le read contengono inevitabilmente degli errori di sequenziamento. Per questo motivo, si preferisce non osservare l'appaiamento diretto suffisso-prefisso quanto piú la loro somiglianza (questo può essere fatto, ad esempio, mediante programmazione dinamica).

- **Layout.**

Noto l'ordine in cui le read sono disposte, occorre unirle in un'unica stringa. In questo passaggio é critica la gestione delle stringhe duplicate, dato che per definizione é impossibile determinarne la posizione all'interno del DNA; inoltre, occorre anche determinare se due stringhe presentano un overlap significativo perché sono effettivamente consecutive all'interno del DNA o per una coincidenza.

- **Consensus.**

Il layout ha implicitamente assegnato ad ogni posizione del DNA una o piú basi che possono trovarvisi. Se tutte le read hanno lo stesso carattere in una certa posizione del loro overlap, allora si é (approssimativamente) certi che il DNA contenga effettivamente tale carattere in tale posizione. Se le diverse read non concordano sul carattere presente in una certa posizione, occorre prendere una decisione. Un semplice approccio potrebbe essere quello di riportare la frequenza di ciascun carattere per ciascuna posizione, e lasciare all'utente come interpretare questa informazione, oppure scegliere il carattere che si presenta piú spesso in una data posizione e assumere che sia quello corretto.

6.2 Sequenziamento casuale

Un primo approccio al problema del sequenziamento del DNA é offerto dallo **shotgun sequencing**, o **sequenziamento casuale**. Questo si compone di quattro passaggi:

1. Avendo a disposizione il DNA da sequenziare, ne vengono create diverse copie (ad esempio tramite PCR);
2. Si rompe ciascuna copia in piccoli frammenti, di modo che ciascuna stringa di DNA subisca tagli in punti quanto piú casuali possibile;
3. Ciascun frammento viene sequenziato separatamente;
4. Se vengono sequenziati abbastanza frammenti che si sovrappongono fra di loro, e queste sovrapposizioni sono abbastanza lunghe da essere certi che siano univoche, allora é possibile sequenziare il DNA unificando i sotto-sequenziamenti cosí ottenuti.

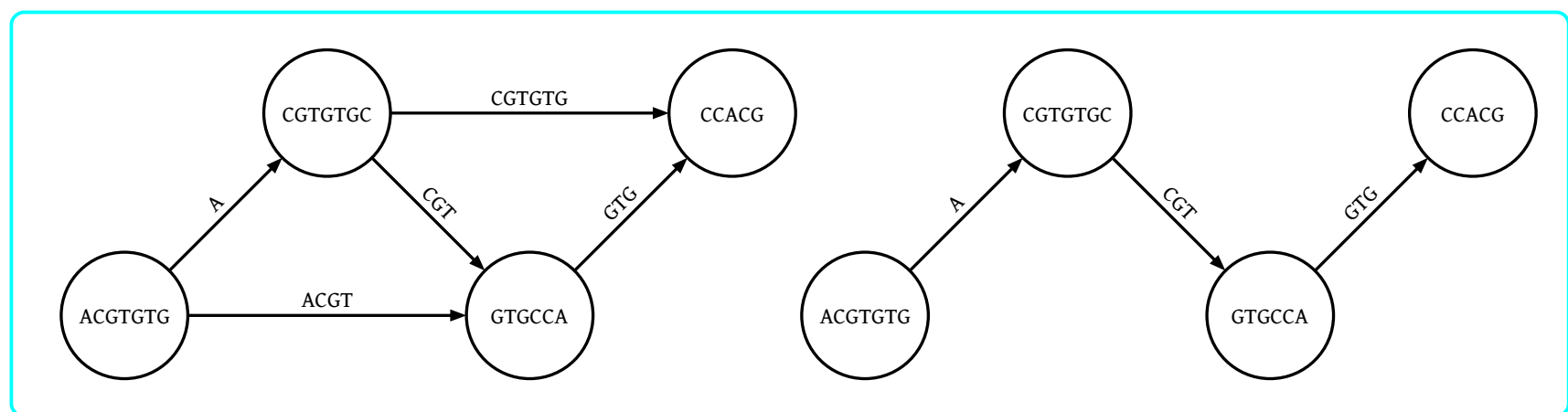
Se le read di DNA vengono intese come delle stringhe sull'alfabeto A, C, G, T , e l'insieme di tutte le read contiene le informazioni sull'intero DNA, allora questo non é altro che una stringa formata dalla concatenazione di sottostringhe di varia lunghezza di tutte le read. Piú formalmente, si ha che l'intero DNA é una **superstringa** delle relative read.

Dato un insieme di k stringhe $P = \{S_1, S_2, \dots, S_k\}$, una **superstringa** dell'insieme P é una stringa che contiene ogni stringa di P come sua sottostringa. Si noti come la superstringa di un insieme P non deve necessariamente essere membro di P . É sempre possibile ottenere una superstringa triviale per un generico insieme P concatenando in un qualsiasi ordine tutte le stringhe che fanno parte di P . Pertanto, almeno una superstringa esiste sempre per qualsiasi P .

Sia $P = \{abcc, efab, bccla\}$. Una superstringa molto semplice per P è $efababccbcla$. Un'altra superstringa per P , più difficile da individuare, è $efabccla$.

In relazione al problema del sequenziamento del DNA, la stringa che codifica l'intero DNA è, con molta probabilità, quella che ha la lunghezza minore possibile. Questo perché, in linea con l'approccio della massima parsimonia, è la stringa che richiede il minimo "sforzo" per essere costruita. Dato un insieme di stringhe P , sia $S^*(P)$ la superstringa di P più piccola possibile.

È possibile approcciare il problema di determinare la superstringa più corta di un insieme di stringhe mediante grafi. Dato un insieme di n read, un **grafo di overlap** è un grafo che ha i nodi etichettati con le read e ha archi fra tutte le coppie di read che hanno un overlap "sufficientemente lungo", etichettati con l'overlap stesso. Di norma si impone un valore soglia per la lunghezza degli overlap al di sopra del quale si assume che l'overlap non possa essere dovuto ad una coincidenza ma al fatto che le due read sono state estratte da porzioni contigue di DNA. Un grafo di overlap a cui vengono rimossi gli archi transitivi prende il nome di **grafo di stringhe**.



Noto il grafo di overlap associato ad un insieme di read, il sequenziamento del DNA può essere ricavato percorrendo un cammino sul grafo che visiti ciascun nodo esattamente una volta e che massimizzi la somma delle lunghezze degli archi attraversati. Alla stringa così ottenuta è sufficiente concatenare in testa il prefisso della prima read non presente nell'overlap e concatenare in coda il suffisso dell'ultima read non presente nell'overlap.

Il problema del sequenziamento del DNA ha notevoli somiglianze con il problema del determinare un **cammino hamiltoniano** su un grafo, in particolare nella sua variante dove le etichette degli archi ne determinano un peso, il **problema del commesso viaggiatore (travelling salesman problem)**, o **TSP**. Questo problema ha però una complessità NP-hard, peranto non può esistere un algoritmo in grado di risolverlo in tempo polinomiale. Nonostante questo, è comunque possibile risolvere piccole istanze di TSP in tempo accettabile.

Dato un grafo orientato e completo $G = (V, A)$, con archi pesati $w : A \mapsto \mathbb{Q}^+$, ogni soluzione del problema TSP è un percorso che visita ogni nodo esattamente una volta e torna al punto di partenza, dove il costo è dato dalla somma dei pesi di tutti gli archi attraversati. La soluzione ottimale è la permutazione $\Pi = \langle \pi_1, \dots, \pi_n \rangle$ di V che minimizzi la quantità:

$$w(\pi_n, \pi_1) + \sum_{i=1}^n w(\pi_i, \pi_{i+1})$$

Se è possibile convertire una istanza del problema del sequenziamento del DNA in una istanza del problema TSP equivalente, allora i due problemi possono essere risolti con lo stesso approccio. Fra i due problemi vi sono due notevoli differenze: nel TSP si cerca una stringa di lunghezza minima, mentre nell'assemblaggio si cerca una stringa di lunghezza massima; inoltre, nel primo occorre trovare un percorso che attraversi tutti i nodi esattamente una volta e abbia nodo iniziale e finale coincidenti, mentre nel secondo è sufficiente che il percorso attraversi tutti i nodi esattamente una volta.

La prima differenza può essere rimossa etichettando il grafo di overlap non con l'overlap fra due read ma con la lunghezza della parte di read che non fa parte dell'overlap. In questo modo, il problema del sequenziamento del DNA ha la soluzione migliore nel cammino che ha minimo il peso delle etichette.

La seconda differenza può essere risolta aggiungendo al grafo di overlap due nodi speciali, uno di partenza ed uno di arrivo, con queste caratteristiche: il nodo di partenza ha solo archi uscenti di peso nullo diretti a tutti i nodi del grafo tranne a quello di arrivo, mentre il nodo di arrivo ha solo archi entranti di peso nullo provenienti da tutti i nodi del grafo tranne da quello di partenza. Si aggiunga poi un arco uscente dal nodo di partenza ed entrante nel nodo di arrivo, avente sempre peso nullo. In questo modo, si ha la certezza che qualsiasi cammino sul grafo inizi sul nodo di partenza, passi per tutti i nodi del grafo fino a raggiungere quello di arrivo e poi tornare al nodo di partenza, ed il costo totale rimane invariato (tutti gli archi introdotti sono nulli).

6.3 Sequenziamento mediante ibridazione

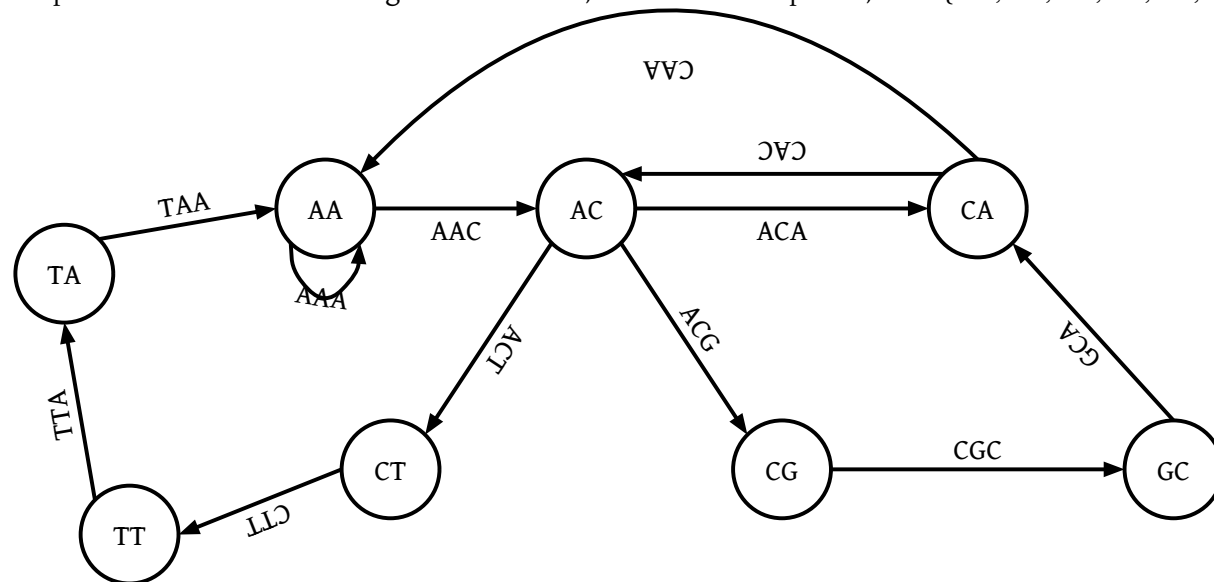
Lo shotgun sequencing risolve il problema del sequenziamento del DNA riducendolo ad una variante del problema TSP, che è NP-hard. Pertanto, sebbene risolvibile in pratica per istanze di dimensioni contenute, non è possibile risolverlo per istanze di dimensioni considerevoli. Un approccio alternativo è dato dal **sequenziamento mediante ibridazione (sequencing by hybridization)**, o **SBH**.

SBH comincia con la costruzione di una matrice di 4^k celle, con k valore fissato, chiamata **chip**¹. Ciascuna cella contiene una copia distinta di tutte le 4^k possibili permutazioni della stessa stringa di DNA di lunghezza k , ciascuna chiamata **k -mero**. Il chip viene messo in contatto con il DNA da sequenziare, al quale è stato applicato un marcatore radioattivo. Se il DNA da sequenziare presenta una sezione complementare ad uno dei k -meri, allora fra i due avverrà l'ibridazione, che viene resa evidente dalla presenza del marcatore radioattivo. A questo punto è sufficiente complementare i k -meri che hanno ibridato con il DNA per ottenere delle sezioni di DNA sequenziate.

Il valore k deve essere sia abbastanza piccolo da avere chip che riportano un numero contenuto di k -meri, ma al contempo abbastanza grande da evitare che una stessa sottostringa compaia più volte nella stringa bersaglio. Si noti come, a differenza dello shotgun sequencing prima trattato, nell'SBH due read non hanno un overlap di lunghezza arbitraria, bensì hanno un overlap di esattamente $k-1$ caratteri di lunghezza.

Questo alto livello di sovrapposizione può essere sfruttato per ottenere una soluzione semplice ed elegante al sequenziamento mediante ibridazione. Data una lista L contenente tutti i k -meri ottenibili a partire dalla stringa bersaglio S , si costruisca un grafo diretto $G(L)$ come segue: si creino 4^{k-1} nodi, ciascuno etichettato con uno dei possibili $(k-1)$ -meri, rimuovendo i duplicati se ve ne sono. Per ciascuna stringa χ in L , si tracci un arco etichettato χ dal nodo che ha per etichetta i $k-1$ caratteri più a sinistra di χ verso il nodo che ha per etichetta i $k-1$ caratteri più a destra di χ ; alcuni nodi in $G(L)$ potrebbero non essere raggiunti da alcun arco, pertanto possono essere soppressi. Un grafo così costruito viene chiamato **Grafo di de Bruijn**.

Sia data la lista $L = \{AAA, AAC, ACA, CAC, CAA, ACG, CGC, GCA, ACT, CTT, TTA, TAA\}$. Avendosi $k = 3$, tutte le stringhe di lunghezza $k-1 = 2$ che possono essere costruite dagli elementi di L , escludendo i duplicati, sono $\{AA, AC, CA, CG, GC, CT, TT, TA\}$.



Similmente al grafo di overlap, è possibile ricavare la stringa del DNA completo a partire dalle etichette degli archi/nodi del grafo di de Bruijn. Si noti però come i vertici di un grafo di de Bruijn non sono etichettati con delle read, ma bensì con sottostringhe di k -meri. Pertanto, non è possibile modellare il problema del sequenziamento del DNA in questa forma come un problema TSP. Dato che i k -meri si trovano sugli archi, l'interesse è allora non nell'attraversare ogni vertice del grafo esattamente una volta, bensì nell'attraversare ogni arco esattamente una volta.

Dato un grafo diretto G , si dice **percorso euleriano** un percorso diretto che attraversa ciascun arco di G esattamente una sola volta. Nello specifico, viene chiamato **circuito euleriano** un percorso euleriano dove l'arco iniziale e l'arco finale coincidono.

Sia G un grafo orientato. G è detto **semi-euleriano** se esistono esattamente due vertici s e t tali che s ha un arco uscente in più di quanti ne ha entranti, t ha un arco entrante in più di quanti ne abbia uscenti e tutti i vertici che non sono né s né t hanno tanti archi entranti quanti archi uscenti. Viene invece detto **euleriano** se ogni suo vertice ha tanti archi entranti quanti archi uscenti.

1. Il nome deriva dal fatto che la tecnologia per costruirli è molto simile a quella usata per costruire i chip in silicio dei calcolatori.

Sia G un grafo semi-euleriano, che possiede due vertici s e t tali che s ha un arco uscente in più di quanti ne abbia entranti e t ha un arco entrante in più di quanti ne abbia uscenti. Sia G_1 il grafo ottenuto a partire da G rimuovendo tutti gli archi di un qualsiasi percorso da s a t . Allora G_1 è euleriano.

Dimostrazione. Sia P un percorso da s a t . Essendo G un grafo semi-euleriano, qualsiasi vertice che non sia s o t ha tanti archi entranti quanti ne ha uscenti. Se viene costruito G_1 rimuovendo P , questo ha l'effetto collaterale di rimuovere un arco uscente da s ed un arco entrante da t . Questo significa che ora anche s e t hanno tanti archi entranti quanti ne hanno uscenti. Quindi, avendo tutti i vertici di G_1 lo stesso numero di archi entranti e uscenti, G_1 è euleriano.

Sia G un grafo euleriano e sia C un ciclo su G . Sia G_1 il grafo ottenuto a partire da G rimuovendo tutti gli archi che compongono C . Allora anche G_1 è euleriano.

Dimostrazione. Essendo G un grafo semi-euleriano, qualsiasi vertice ha tanti archi entranti quanti ne ha uscenti. Sia n il numero di archi entranti o uscenti di un qualsiasi nodo di G . Se viene costruito G_1 eliminando un intero ciclo, ciascun vertice avrà un arco entrante in meno ed un arco uscente in meno. Allora G_1 è ancora euleriano, perché tutti i nodi hanno comunque lo stesso numero di archi entranti e di archi uscenti, ovvero $n-1$.

Un grafo G é semi-euleriano se e solo se ha un percorso euleriano.

Dimostrazione. Sia G un grafo, e sia P un percorso euleriano dal vertice s al vertice t , con $s \neq t$. Per definizione, P percorre tutti gli archi di G esattamente una volta.

Sia v un vertice di G diverso sia da s che da t ; anche questo vertice dovrà per forza venire attraversato da P . Inoltre, non essendo v né il nodo iniziale né quello finale di P , il numero di archi entranti in v é pari al numero di archi uscenti da v .

s é il vertice di partenza per P , pertanto viene attraversato un suo arco uscente nell'immediato inizio di P e, per ogni volta (anche nessuna) che si percorre un arco entrante in s , é necessario anche percorrere un arco uscente da s , perché s é stato scelto distinto dal nodo di arrivo t . Dato che P attraversa tutti i vertici di G esattamente una volta, il vertice s deve necessariamente avere un arco uscente in piú di quanti ne abbia entranti.

t é il vertice di arrivo per P , pertanto viene attraversato un suo arco entrante al termine di P e, per ogni volta (anche nessuna) che si percorre un arco uscente in t , é necessario anche percorrere un arco entrante da t , perché t é stato scelto distinto dal nodo di partenza s . Dato che P attraversa tutti i vertici di G esattamente una volta, il vertice t deve necessariamente avere un arco entrante in piú di quanti ne abbia uscenti.

Si ha quindi che G ha un vertice s che ha piú archi uscenti che archi entranti, un vertice t che ha piú archi entranti che archi uscenti e qualsiasi altro vertice v che non sia né s né t ha tanti archi entranti quanti ne ha uscenti. Ma questa é precisamente la definizione di grafo semi-euleriano, pertanto G é un grafo semi-euleriano.

Sia G un grafo semi-euleriano. Sia s l'arco di G che ha un arco uscente in piú di quanti ne abbia entranti, e sia t l'arco di G che ha un arco entrante in piú di quanti ne abbia uscenti. Partendo da s , si costruisca un percorso P attraversando uno qualsiasi dei suoi archi uscenti raggiungendo un nodo x . Se $x = t$, allora P é un percorso euleriano, ed il teorema é provato. Se invece $x \neq t$, si attraversi un arco uscente da x , distinto da s , raggiungendo un nodo y e si reiteri questa procedura fino a raggiungere t .

Essendo il grafo semi-euleriano, qualsiasi nodo che non sia t ha un nodo di archi uscenti non ancora visitati da P pari al numero di archi entranti non ancora visitati da P , ed ogni volta che P entra in un nodo ha sempre a disposizione un arco non ancora visitato per uscire. Il teorema é provato se é possibile dimostrare che il percorso P cosí costruito é un percorso euleriano.

Si supponga che P non sia un percorso euleriano. Devono allora esistere degli archi di G che P non ha ancora attraversato. Allora, per il teorema precedente, se vengono rimossi da G gli archi non attraversati da P si ottiene un grafo G_1 euleriano. Se G_1 é un grafo euleriano, allora deve avere un ciclo euleriano: sia allora C un ciclo di G formato da soli archi non attraversati da P . Si costruisca un percorso P' composto sia da tutti gli archi attraversati da P , sia da tutti quelli attraversati da C : un percorso di questo tipo puó essere costruito semplicemente iniziando da uno dei nodi in P , percorrendone gli archi fino a raggiungere un nodo che si trova sia in C che in P , percorrendo tutto C fino a tornare al nodo in questione e proseguendo sugli archi rimanenti in P . Dato che P' é un percorso che va da s a t , ci si chiede se sia euleriano: se non lo é, si ripete il procedimento costruendo un nuovo percorso P'' , e cosí via fino ad esaurire tutti gli archi di G . Essendo G un grafo semi-euleriano, é garantito che questo processo possa fermarsi con la costruzione di un percorso euleriano.

Un grafo G é euleriano se e solo se ha un ciclo euleriano.

A partire da un cammino euleriano su un grafo $G(L)$ di De Bruijn, é possibile costruire la stringa corrispondente al genoma. La stringa ha la proprietà che l'insieme delle sue sottostringhe di lunghezza k é esattamente l'insieme L che definisce $G(L)$.

Una stringa S si dice **compatibile** con L se e soltanto se contiene ogni sottostringa in L e (assumendo che L contenga sottostringhe di lunghezza k) non contiene altre sottostringhe di lunghezza k . É evidente come una stringa S sia compatibile con L se e soltanto se é specificata da un cammino euleriano su $G(L)$: questo perché, se cosí non fosse, allora S potrebbe contenere sottostringhe che non si trovano in L .

Questo ha delle conseguenze interessanti: si assuma infatti che ciascuna sottostringa in L si presenti nel DNA bersaglio una ed una sola volta. Allora il dataset L determina senza ambiguitá la stringa DNA bersaglio se e soltanto se $G(L)$ ha un solo cammino euleriano. Inoltre, esiste una corrispondenza uno-ad-uno fra i cammini euleriani per $G(L)$ e le stringhe che sono compatibili con L . Ovvero, ciascuna stringa compatibile corrisponde ad un cammino euleriano distinto, e ciascun cammino euleriano corrisponde ad una stringa distinta.

Se i dati sono stati raccolti in maniera corretta e nessuna sottostringa di lunghezza k appare nella stringa bersaglio piú di una volta, allora $G(L)$ deve necessariamente avere almeno un cammino euleriano che specifica la stringa di DNA in questione. Il problema si presenta quando c'è piú di un cammino euleriano per $G(L)$, perché vi sono piú stringhe elegibili per essere la stringa di DNA voluta. Tuttavia, anche se si presenta questa situazione, é comunque possibile ottenere informazioni utili in merito alla stringa bersaglio. Ad esempio, se una sottostringa compare in tutte o molte delle stringhe elegibili per essere la stringa ricercata, allora si puó presumere che quella sottostringa sia effettivamente presente nella stringa bersaglio.

Capitolo 7

Genotipi e aplotipi

7.1 Ricostruzione degli aplotipi mediante pedigree

Gli esseri viventi piú semplici hanno una sola copia di ciascun loro cromosoma, e si dicono **aploidi**. Inoltre, si riproducono per **meiosi**, ovvero un organismo crea una copia identica di sé stessa. Per questo motivo, é molto semplice determinare la genealogia di esseri viventi di questo tipo, perché ogni organismo figlio é una copia identica del genitore.

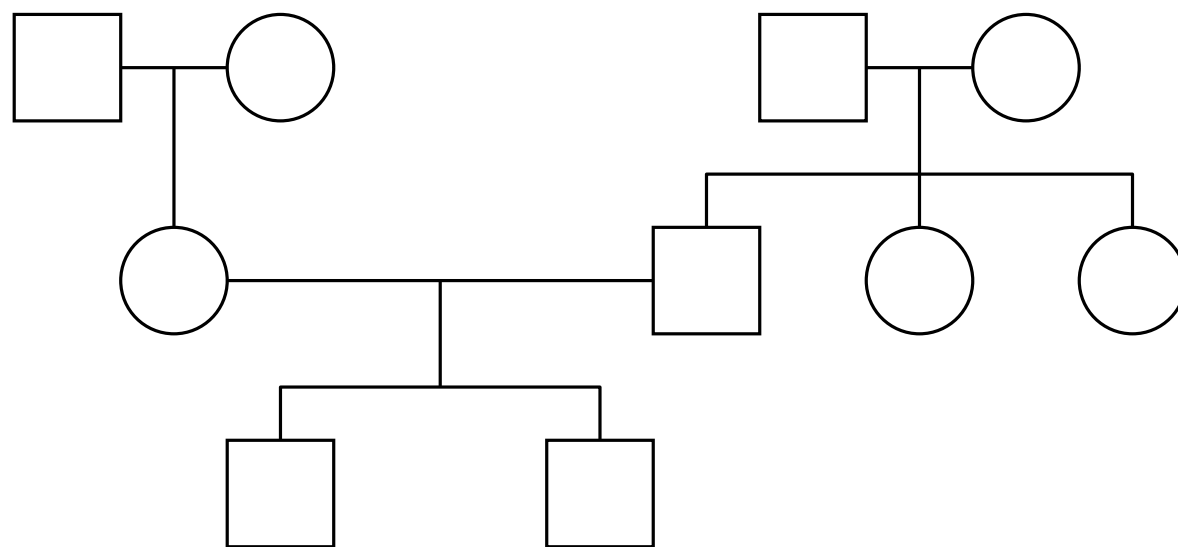
Gli esseri viventi piú complessi sono invece **diploidi**: ogni cromosoma é presente nel DNA in due copie, dove ciascuna copia é ereditata da uno dei due genitori secondo le **leggi di ereditarietà di Mendel**. La quasi totalità delle due copie é funzionalmente identica, nel senso che entrambe contengono le informazioni necessarie all'esprimere la stessa cosa, ma non hanno necessariamente la stessa espressione fenotipica, perché potrebbero presentare **alleli** distinti. Il caso piú comune di differenza fra i due nucleotidi prende il nome di **SNP**¹ (**single nucleotide polymorfism**), dove un singolo nucleotide é differente nei due cromosomi per lo stesso locus.

Idealmente, ci si aspetterebbe che il padre e la madre trasmettano alla prole uno dei loro due aplotipi (a loro volta ereditati dai rispettivi genitori) nella loro integrità. In realtà, durante la meiosi avviene un fenomeno chiamato **ricombinazione**, dove l'aplotipo risultante é un ibrido fra i due aplotipi dei genitori. Se in due loci é presente lo stesso nucleotide, si dice che quel locus é **omozigote**; se invece é presente un nucleotide diverso, si dice che quel locus é **eterozigote**.

L'informazione relativa a quale nucleotide si trovi su ciascun cromosoma per uno stesso locus viene chiamata **aplotipo**. La coppia di basi per uno stesso locus, senza specificare da quale cromosoma ciascuna base provenga, viene chiamata **genotipo**. La tecnologia attuale non é in grado di determinare immediatamente gli aplotipi di un cromosoma, ma é sufficientemente avanzata da determinare i genotipi (ovvero, é possibile dedurre quali basi sono presenti su uno stesso locus ma non é possibile dedurre da quale due cromosomi ciascuna provenga).

Si consideri uno stesso locus per entrambi i cromosomi di una coppia. Se sul primo é presente la base A e sul secondo la base A, sia il genotipo che l'aplotipo é AA. Se invece é presente sull'uno C e sull'altro T, il genotipo é sempre lo stesso (indifferentemente CT o TC) mentre l'aplotipo potrebbe essere CT o TC.

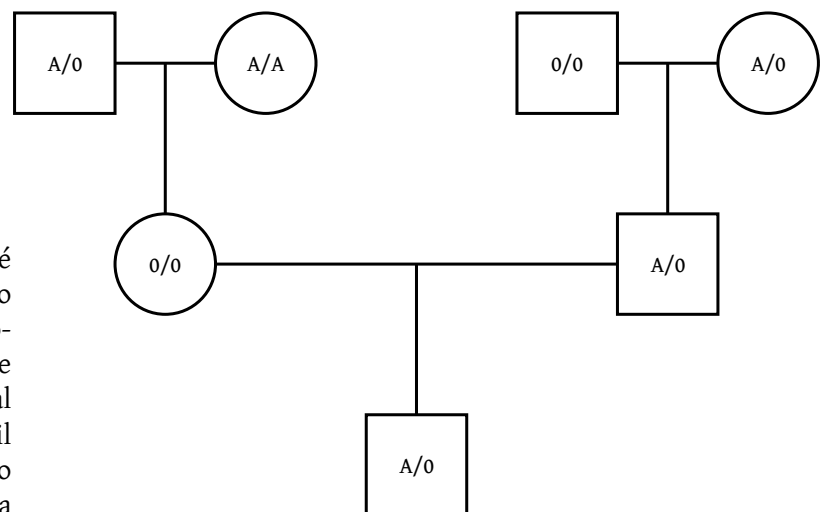
Un primo metodo per dedurre l'aplotipo di un individuo é studiare il genotipo dei suoi antenati. Un diagramma utile a questo scopo é il **pedigree**, una struttura che riporta, per ciascuno degli antenati di un individuo, le loro relazioni (padre, madre, figlio, ecc...), il loro sesso (indicando con un rettangolo il sesso maschile e con un cerchio il sesso femminile) ed il genotipo sul quale si vogliono fare inferenze.



Questa struttura può essere rassimilata ad un albero quando non sono presenti cicli, ovvero quando non si é verificato **inbreeding** fra gli antenati².

1. Da leggersi "snip"
2. Fintanto che si parla di *Homo Sapiens*, i pedigree sono quasi certamente alberi, perché l'inbreeding é estremamente raro. La stessa cosa non si può dire per gli animali domestici, specialmente quelli di razza.

Si consideri il pedigree qui presentato, dove per ogni individuo é riportato il genotipo relativo al suo gruppo sanguigno. L'individuo della terza generazione possiede il genotipo A/0, la madre ha genotipo A/0 ed il padre ha genotipo 0/0. Da questo é possibile dedurre che l'individuo ha necessariamente ereditato A dalla madre e 0 dal padre. In particolare, la madre ha un solo A, quindi é certo quale il figlio abbia ereditato, mentre il padre ha due 0, quindi non vi sono informazioni sufficienti a determinare quale dei due 0 il figlio abbia ereditato.



Si consideri una situazione semplificata in cui, per ogni individuo di una popolazione, é noto il suo genotipo per ogni locus, é noto il pedigree della popolazione e non vi é mai ricombinazione, ovvero il cromosoma dei genitori viene ereditato dai figli "integralmente". Per comoditá, si considerino esclusivamente loci binari, ovvero che possono avere soltanto due possibili alleli, indicati con 0 e 1. Per convenzione si indica con 0 l'allele **maggiore**, ovvero quello piú presente nella popolazione, e con 1 il **minore**, il meno presente. Si considerino inoltre esclusivamente pedigree senza cicli.

Sulla base di queste informazioni, é possibile ricostruire gli aplotipi della popolazione mediante programmazione lineare. Il problema puó essere affrontato risolvendo una equazione lineare nella forma $Ax = B$ mediante eliminazione gaussiana, dove A é una matrice di valori reali, x é un vettore colonna di incognite e B é un vettore colonna di valori reali. Dato che si sta considerando una situazione semplificata dove gli alleli sono sempre e solo due per ciascun locus, il campo dei numeri che possono presentarsi come termini noti viene ristretto dall'intero \mathbb{R} ai soli 0 e 1. Piú formalmente, viene ristretto al solo \mathbb{Z}_2 , ovvero il resto modulo 2 di numeri interi. Nello specifico, siano:

- $g_i[j]$ una matrice nota contenente il genotipo dell'individuo i nel locus j . Ogni cella della matrice puó contenere esclusivamente tre valori: 0 se l' i -esimo individuo ha genotipo omozigote maggiore, 1 se l' i -esimo individuo ha genotipo omozigote minore, 2 se l' i -esimo individuo ha genotipo eterozigote;
- $w_i[j]$ una matrice nota con celle a valori binari, dove una cella contiene valore 0 se il locus j dell' i -esimo individuo é omozigote e 1 se eterozigote;
- $h_{x,i}$ una matrice incognita, definita solamente se x é genitore di i . i ha necessariamente un aplotipo ereditato da x , ma non é noto quale dei due sia. Vale 0 se i ha ricevuto da x l'aplotipo da parte paterna e 1 se da parte materna;
- $p_i[j]$ una matrice che contiene l'aplotipo paterno dell' i -esimo individuo al locus j .

Non é necessario riportare esplicitamente l'aplotipo materno degli individui, perché questo puó essere interamente dedotto a partire da $w_i[j]$ e da $p_i[j]$. Infatti, se $w_i[j] = 0$, allora l'individuo i é omozigote per il locus j , e quindi l'aplotipo materno e quello paterno (ovvero $p_i[j]$) sono uguali, mentre se $w_i[j] = 1$ allora l'individuo i é eterozigote per il locus j , e quindi l'aplotipo materno é dato da $1 - p_i[j]$, ovvero il complementare di quello paterno. Ricordando che si stanno considerando esclusivamente valori in \mathbb{Z}_2 , l'aplotipo materno puó essere ottenuto semplicemente sommando $w_i[j]$ e $p_i[j]$.

Nel caso in cui un individuo abbia un aplotipo omozigote, questo é dato semplicemente da $(g_i[j], g_i[j])$. Se invece é eterozigote, potrebbe essere equivalentemente (0, 1) oppure (1, 0). Si indichi con p il padre dell'individuo, con m la madre e con x un genitore generico. In base a come $h_{x,i}$ é stata definita, si hanno quattro possibili combinazioni per come ciascun individuo abbia ereditato l'aplotipo dai genitori:

- Aplotipo del padre da parte paterna;
- Aplotipo del padre da parte materna;
- Aplotipo della madre da parte paterna;
- Aplotipo della madre da parte materna;

$$\begin{cases} h_{p,f} = 0 \Rightarrow p_f[j] = p_p[j] \\ h_{p,f} = 1 \Rightarrow p_f[j] = p_p[j] + w_p[j] \\ h_{m,f} = 0 \Rightarrow p_f[j] + w_f[j] = p_m[j] \\ h_{m,f} = 1 \Rightarrow p_f[j] + w_f[j] = p_m[j] + w_m[j] \end{cases}$$

Si noti però come, potendo $h_{p,f}$ e $h_{m,f}$ assumere esclusivamente valore 0 oppure 1, le quattro equazioni possono essere ridotte a due:

$$p_f[j] = p_p[j] + w_p[j]h_{p,f} \qquad p_f[j] + w_f[j] = p_m[j] + w_m[j]h_{m,f}$$

Le due equazioni possono poi essere ulteriormente unificate in una sola introducendo la variabile $d_{x,f}$, che vale 0 nel caso in cui $x = p$ (si sta considerando il padre) e vale w_f nel caso in cui $x = f$ (si sta considerando la madre):

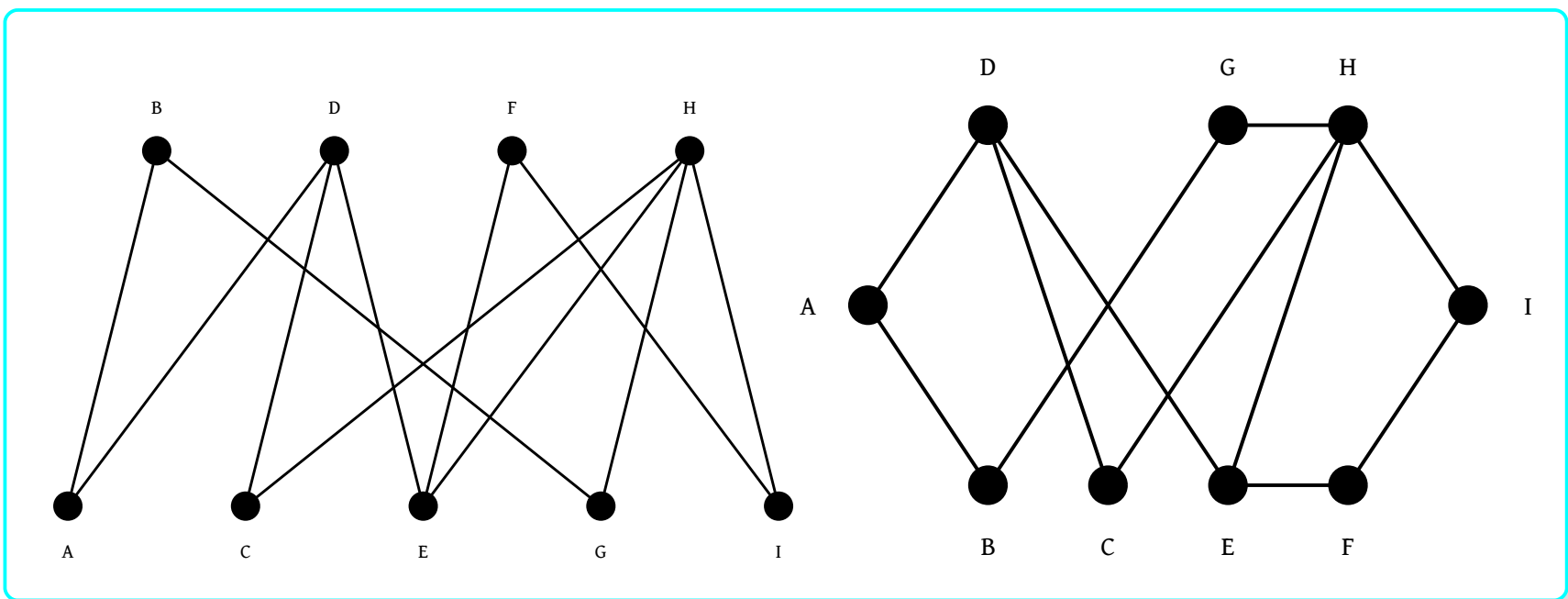
$$p_f[j] + d_{x,f} = p_x[j] + w_x[j]h_{x,f}$$

A questo punto, le uniche due incognite sono $h_{x,f}$ e $d_{x,f}$. L'equazione lineare cosí scritta sembrerebbe non essere risolvibile mediante eliminazione gaussiana, perché le quantità $w_x[j]$ e $h_{x,f}$ sono legate da un prodotto; tuttavia, nonostante il pedice x , $w_x[j]$ non é una incognita, pertanto il prodotto fra $w_x[j]$ e $h_{x,f}$ non il prodotto di due variabili, bensí il prodotto di una costante nota per una variabile, pertanto l'eliminazione gaussiana é ancora valida.

7.2 Ricostruzione degli aplotipi per singolo individuo

Sebbene sia piú complesso, é possibile fare inferenze sulla provenienza degli aplotipi di un individuo senza rifarsi al suo pedigree. Il problema consiste, dato un insieme di read del DNA di un individuo, nel separare l'insieme in due sottoinsiemi, uno per ciascun aplotipo di provenienza. Una possibile formulazione viene data dal **grafo dei conflitti**, un grafo i cui vertici sono le read ed ogni arco é una coppia di read non compatibili, ovvero che si é certi non possano provenire da uno stesso aplotipo. Il problema puó essere quindi riformulato come il partizionare il grafo in due sottografi, tali per cui ciascun sottografo contenga solo read provenienti da uno dei due aplotipi.

Un grafo di questo tipo viene detto **grafo bipartito**, dove tutti i nodi di una partizione ha archi solo e soltanto con nodi dell'altra partizione. Sebbene a volte si possa dedurre se un grafo sia o non sia bipartito semplicemente osservandone la topologia, é possibile dimostrare un risultato piú solido: un grafo é bipartito se e solo se non contiene cicli di lunghezza dispari. Preso infatti un grafo G non avente alcun ciclo di lunghezza dispari, si consideri un qualsiasi nodo v : é possibile partizionare il grafo in due parti, da una parte il solo v e dall'altra parte tutti i nodi adiacenti a v , dopodiché si ripete il ragionamento per ciascuno di questi nodi fino ad esaurirli tutti. Di fatto, questa é una ricerca in profondità (breadth-first search). Questa procedura divide il grafo in livelli.



Si noti però che molto raramente le read di aplotipi sono assimilabili a grafi aventi questa proprietà, pertanto non é possibile riportarli immediatamente come grafi bipartiti. Ci si chiede allora come si possa modificare un grafo nella maniera meno "invasiva" possibile tale per cui il nuovo grafo é bipartito, ovvero il minimo numero dgli archi da eliminare tali per cui un grafo non bipartito diviene bipartito.

Un approccio alternativo é il **MEC (minimal error correction)**, ovvero non eliminare vertici del grafo risultante fino a renderlo bipartito ma modificando quelli esistenti (che nel contesto della lettura del genoma significa modificare il minimo numero di caratteri). Questo approccio é certamente migliore, ma richiede necessariamente un genoma di riferimento su cui effettuare la comparazione. Infatti, altrimenti sarebbe impossibile sapere se un carattere osservato sia parte di un polimorfismo effettivamente esistente (e quindi non modificarlo) oppure non lo é (e quindi modificarlo).