Appunti di
Elementi di Bioinformatica

A cura di:
Francesco Refolli
Matricola 865955

Anno Accademico 2022-2023

Chapter 1

Note sul Corso

todo: segnare delle note

Part I

Teoria

Chapter 2

Pattern Matching

2.1 Introduzione

Pattern Matching Per **Pattern Matching** si intende trovare tutte le occorrenze di un pattern P di lunghezza m all'interno del testo T di lunghezza n.

Notazione Data una stringa o sequenza S, si identifica con $S[i : j]$ la sottostringa che contiene gli elementi da i a j (compreso).

2.2 Algoritmo Banale

Ragionamento L'algoritmo piu' semplice a cui si puo' pensare consiste nello scorrere linearmente il pattern P e provare per ogni posizione $i \in [1, n]$ la corrispondenza con una porzione di testo di uguale lunghezza.

Algorithm Confronta Stringhe

```
procedure ConfrontaStringhe(X, Y)
    for i ← 1 to n do
        if X[i] ≠ Y[i] then
            return false
        end if
    end for
    return true
end procedure
```

Algorithm Pattern Matching banale

```
procedure PMB(T, P)
    for i ← 1 to n do
        if ConfrontaStringhe(T[i : i + m − 1], P) then
            print(i)
        end if
    end for
end procedure
```

Procedura

Complessita' Come si puo' notare, sia PMB che $ConfrontaStringhe$ sono procedure la cui complessita' e' legata principalmente al singolo ciclo che contengono.
PMB contiene un ciclo di n iterazioni fisse, quindi il suo tempo nel caso medio sara' $T_{PMB} = \Theta(n)$.
ConfrontaStringhe al contrario contiene si un ciclo di m iterazioni, ma il numero di volte in cui saranno ripetuto il confronto dipendera' dalla similitudine delle due stringhe.
Nel caso medio sara' circa meta' dei caratteri, quindi $T_{ConfrontaStringhe} = \Theta(m/2) = \Theta(m)$.
Visto che PMB incorpora una chiamata a ConfrontaStringhe, la complessita' totale sara': $T(n, m) = \Theta(n * m)$.
Per quanto riguarda lo spazio, e' facilmente intuibile che sia $S(n) = \Theta(n + m)$.

2.3 Baeza-Yates-Gonnet

Approccio Bit-Parallel Quando si devono effettuare piu' operazioni dello stesso tipo poco costose e ripetitive e' possibile ridurre il problema ad azioni elementari che la CPU puo' processare in parallelo per ridurre il numero di passi da fare per completare un algoritmo. Per esempio se si devono sommare vettori di bit e' possibile assemblare una word che li contenga in modo da poi effettuare operazioni bit-wise (ovvero bit-a-bit, ogni bit non interferisce con quello successivo) per ottimizzare il calcolo. Si puo' applicare in certi casi anche agli algoritmi di pattern matching.

Ragionamento Si supponga di disporre di una matrice M di dimensione $m \cdot n$. Ogni cella e' riempita come segue:

Con $j = 0$:

$$M[i][j] = \begin{cases} 1 & \text{sse } P[i] = T[0] \\ 0 & \text{altrimenti} \end{cases}$$

Per ogni $j > 0$:

$$M[i][j] = \begin{cases} 1 & \text{sse } P[:i] = T[j - i + 1 : j] \\ 0 & \text{altrimenti} \end{cases}$$

Sappiamo per certo che:
$P[:i] = T[j - 1 + 1 : j] \iff P[:i - 1] = T[j - i + 1 : j - 1] \wedge P[i] = T[j]$.
Quindi possiamo scrivere la cella generica $M[i][j]$ in modo ricorsivo rispetto alla matrice:

Con $j = 0$:

$$M[i][j] = \begin{cases} 1 & P[i] \wedge T[0] \\ 0 & \text{altrimenti} \end{cases}$$

Per ogni $j > 0$:

$$M[i][j] = \begin{cases} 1 & M[i-1][j-1] = 1 \wedge P[i] = T[j] \\ 0 & \text{altrimenti} \end{cases}$$

Visto che $\wedge$ e' un'operazione bitwise rispetto a $(0, 1)$ possiamo scrivere:
Con $j = 0$:

$$M[i][j] = \begin{cases} 1 & P[i] \wedge T[0] \\ 0 & \text{altrimenti} \end{cases}$$

Per ogni $j > 0$:

$$M[i][j] = \begin{cases} 1 & M[i-1][j-1] \wedge P[i] \wedge T[j] \\ 0 & \text{altrimenti} \end{cases}$$

Definiamo un vettore U con $|U| = |P|$ come:

$$U(j) = \begin{cases} 1 & P[i] \wedge T[j] \\ 0 & \text{altrimenti} \end{cases}$$

Visto che ogni cella $M[i][j]$ e' il risultato di un $\wedge$ tra la cella $M[i-1][j-1]$ e $U(j)[i]$, possiamo sfruttare la colonna $M[\cdot][j-1]$ per generare la colonna $M[\cdot][j]$.

Da qui in poi chiamero' ω la dimensione in bit della word della CPU.
Per poter utilizzare $M[i-1][j-1]$ per $M[i][j]$ possiamo sfruttare l'operazione bitwise rshift su $M[\cdot][j-1]$ e inserire un 1 in posizione 0:
$tmp_{M[\cdot][j-1]} = (M[\cdot][j-1] >> 1) \vee (1 << (\omega - 1))$.

Quindi:
$M[\cdot][j] = tmp_{M[\cdot][j-1]} \wedge U(j)$

Per trovare quindi le occorrenze di P occorrera' quindi controllare l'ultima riga di M. Se $M[|P|-1][j] = 1$ si dice che P occorre in T in posizione j.

Visto che queste operazioni sono bitwise possiamo usare il paradigma Bit-Parallel trattando le colonne $M[\cdot][j]$ come un numero intero processabile dalla CPU. Lo stesso vale per il vettore $U(j)$.

Algorithm Baeza Yates Gonnet

```
procedure BYG(T, P)
    V ← U(0)
    for j = 1 to n do
        tmp ← (V >> 1) ∨ (1 << (ω − 1))
        V ← (tmp ∧ U(j))
        if V mod 2 = 1 then
            print(j)
        end if
    end for
end procedure
```

Procedura

Complessita' Di primo acchito potrebbe sembrare che $T(n,m) = \Theta(n)$. Nell'algoritmo abbiamo un solo ciclo con $\Theta(n)$ iterazioni.
Tuttavia questo avviene sse $|P| \leq \omega$. In quel caso infatti $T_{U(j)} = \Theta(1)$, perche' la costruzione di U richiedera' sempre $O(\omega)$ iterazioni. L'algoritmo necessita solo di una piccola modifica, ma che porta la complessita' a cambiare radicalmente.
Nel caso $|P| > w$, la transizione $M[\cdot][j-1] \leftarrow M[\cdot][j]$ sara' $T_{U(j)}(n,m) = \Theta(\frac{m}{\omega})$, perche' il processo di transizione occupera' piu' operazioni della CPU. Quindi la complessita' dell'algoritmo nel caso generale e' $T(n,m) = O(n \cdot m)$.

2.4 Karp Rabin

Ragionamento L'idea e' quella di usare una funzione di hash $H(S)$ per confrontare il pattern P con il testo T.
Si itera con una finestra scorrevole W con $|W| = m$. Si calcola una sola volta $H(P)$, quindi si confronta $H(W)$ con $H(P)$ alla ricerca di occorrenze.

La funzione di hash usata in questo caso e':
$H(S) = \Sigma_{i=1}^{|S|} 2^{i-1} H(S[i])$

Il calcolo di $H(T[i+1 : i+m])$ puo' essere ottimizzato usando $H(T[i : i+m-1])$:
$H(T[i+1 : i+m]) = \frac{H(T[i:i+m-1]) - T[i]}{2} + 2^{m-1} T[i+m]$.

Algorithm Karp Rabin

```
procedure KB(T, P)
    H_P ← H(P)
    H_W ← H(T[: m])
    if H_P H_W then
        print(0)
    end if
    for i = 1 to n − m do
        H_W ← ((H_W − T[i])/2) + 2^{m−1} T[i + m]
        if H_P = H_W then
            print(i)
        end if
    end for
end procedure
```

Procedura

Complessita' Visto che l'aggiornamento di H_W costa $\Theta(1)$, la creazione del primo H_W costa $\Theta(m)$ ed e' presente solo un ciclo di $\Theta(n-m)$ iterazioni, la complessita' temporale e' $T(n,m) = \Theta(n+m)$.

Problemi Purtroppo i calcolatori hanno capacita' limitata, e, come nel caso dello Baeza-Yates-Gonnet, se $m > \omega$ allora la transizione di H_W viene a costare $\Theta(\frac{m}{\omega})$ e quindi l'algoritmo costera' $T(n,m) = O(n \cdot m)$.

Si puo' pensare di risolvere questo problema limitando la dimensione di H_W applicando un modulo p. Tuttavia questo introduce necessariamente la possibilita' di ottenere **Falsi Positivi** a causa delle collisioni che si possono creare con il modulo sbagliato.

Se descriviamo la probabilita' che un falso positivo si verifichi possiamo scrivere: $P(FP) = \frac{1}{2}$. Ad ogni modo e' possibile generalizzare questa variante provando in successione k moduli diversi per ottenere una probabilita' di errore di $P(FP) = \frac{1}{2^k}$. Dopo ogni falso positivo si sostituisce il modulo che ha generato la collisione. Per questo motivo questa variante del Karp Rabin rientra nella classe degli algoritmi probabilistici.

Algoritmi Probabilistici Esistono due class di algoritmi probabilistici.

Monte Carlo Puo' essere non corretto, ma e' comunque veloce ed efficiente. La vairiante del Karp Rabin rientra in questa categoria.

Las Vegas E' sicuramente corretto ma potenzialmente costoso. Il quicksort con pivot randomico rientra in questa categoria.

Chapter 3

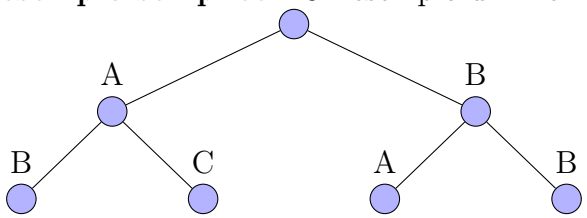
Suffix Tree e Suffix Array

3.1 Trie

Un **Trie** e' un albero ordinato in cui i nodi non mantengono una copia della loro chiave, ma essa e' rappresentata dal cammino radice-nodo. Infatti ogni arco tra nodi e' etichettato con una stringa.

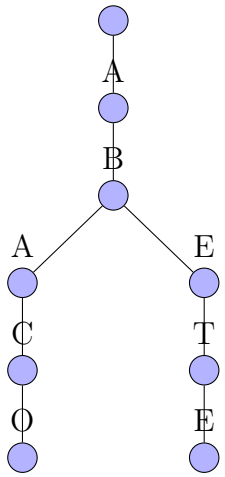
Esempi

esempio semplice Un esempio di Trie:



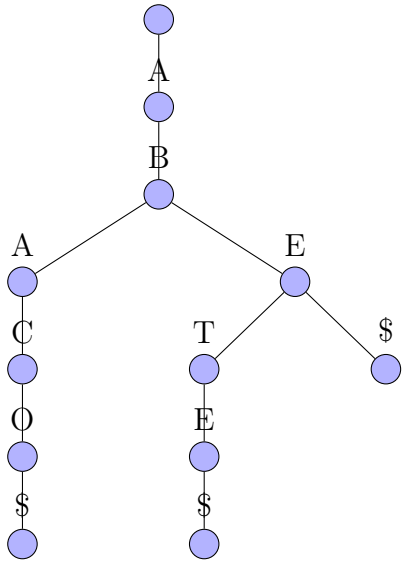
con dizionario Puo' essere generato anche a partire da un dizionario, per esempio:

$$\text{ABE} \tag{3.1}$$
$$\text{ABETE} \tag{3.2}$$
$$\text{ABACO} \tag{3.3}$$



con terminatori Facendo terminare ogni parola del dizionario con un terminatore $\$ \notin \Sigma$, e' possibile costruire un Trie basato su dizionario che indichi chiaramente quali siano le parole del dizionario. Infatti se il dizionario non gode della prefix-free property, ovvero se $\exists w, x \in \Sigma^*$ tali che $\exists s \in \Sigma^* \mid w \cdot s = x$, il Trie conterra' per x e w due foglie distinte che segnalano la fine delle due parole. Ovvero per ogni parola nel dizionario sara' presente la rispettiva e distinta foglia corrispondente al terminatore \$.

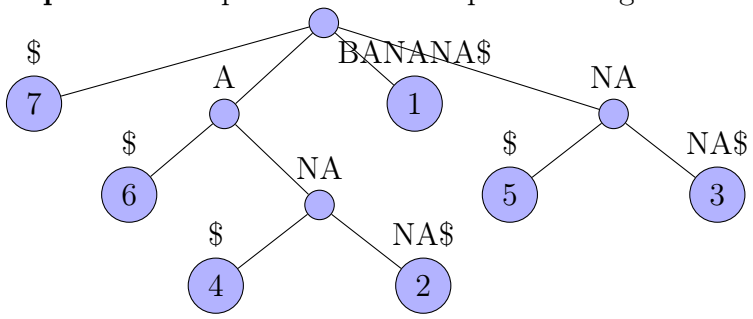
$$\text{ABE\$} \tag{3.4}$$
$$\text{ABETE\$} \tag{3.5}$$
$$\text{ABACO\$} \tag{3.6}$$



3.2 Suffix Tree

Un **Suffix Tree** e' un **Trie** che rappresenta un insieme di suffissi in cui tutti i nodi hanno archi che iniziano con simboli diversi e solo la radice puo' avere meno di due archi. Ogni nodo contiene come valore la posizione di inizio del suffisso rispetto al testo di partenza. Un Trie su una stringa S e' generato a partire dall'elenco di tutti i suffissi di S terminati in \$ (quindi un dizionario di lunghezza $|S|$). Questa struttura dati richiede spazio in memoria con una complessita' di $O(n^2)$.

Esempi Un esempio di Suffix Tree per la stringa BANANA\$:



L'esempio riprende quello dei lucidi del prof. Della Vedova.

3.3 Suffix Array

E' un array di tutti i suffissi di S in ordine lessicografico associati alle rispettive posizioni di inizio nel testo.

Esempi Riprendendo il Suffix Tree della sezione precedente e il rispettivo esempio:

i	1	2	3	4	5	6	7
SA	$	A$	ANA$	ANANA$	BANANA$	NA$	NANA$

Possiamo quindi calcolare la Lunghezza del Prefisso Comune ai suffissi utilizzando la formula di ricorrenza:
$Lcp[i] = |CommonPrefix(SA[i], SA[i+1])|$

i	1	2	3	4	5	6	7
SA	$	A$	ANA$	ANANA$	BANANA$	NA$	NANA$
LCP	0	1	3	0	0	2	–

3.4 Costruzione di Suffix Array

Un metodo naive potrebbe essere riassunto in tre fasi.

Creazione Si crea l'array raccogliendo tutti i suffissi di S, con una complessita' lineare: $T(n) = \Theta(n)$

Ordinamento Si puo' quindi riordinare i suffissi con un algoritmo a piacere, per esempio con un heap sort, il quale ha $T(n) = O(n \cdot logn)$. Siccome confrontare due stringhe ha $T(n) = O(n)$, il totale e': $T(n) = O(n^2 \cdot logn)$.

LCP Si tratta semplicemente di calcolare LCP per ogni due entry consecutive: $T(n) = \Theta(n)$

Totale La complessita' totale e':
$T(n) = \Theta(n) + O(n^2 \cdot logn) + \Theta(n)$
$T(n) = O(n^2 \cdot logn)$

3.5 Da Suffix Tree a Suffix Array

Per creare il Suffix Array e' necessaria una visita depth-first del Suffix Tree in cui i nodi fratelli sono visitati secondo l'ordine lessicografico delle stringhe dei relativi archi uscenti al nodo padre. Questo processo puo' essere usato per calcolare anche la Lunghezza del Prefisso Comune, modificando la formula usata nella sezione precedente:
$Lcp[i] = NodeDepth(LCA(i, i+1))$, dove LCA e' il Lowest Common Ancestor, trattato piu' avanti.

3.6 Da Suffix Array a Suffix Tree

Innanzitutto si deve disporre del Suffix Array con le informazioni sul LCP. L'algoritmo si divide in due fasi:

- Ricostruzione del Suffix Tree.
- Riempimento del Suffix Tree.

Si considerino le seguenti procedura:

Ricostruzione del Suffix Tree Sia dato un array LCP. Se $|LCP| = 0$ allora ritorno una foglia. Altrimenti creo un nodo e procedo.

Sia quindi $k \in LCP$ l'elemento piu' piccolo in LCP, tratto quell'elemento come un separatore e applico ricorsivamente la procedura sui sotto array.

Esempio:
Dato $LCP = [0, 1, 3, 0, 0, 2]$. Il valore piu' piccolo e' 0, quindi formo i sotto array $< [], [1, 3], [], [2] >$ e creo un sottoalbero che come figli i nodi creati dall'applicazione ricorsiva di questa procedura ai singoli sottoarray.

Riempimento del Suffix Tree Per riempire il ST e' sufficiente una visita depth-first in cui si assegnano i valori delle foglie con i valori di SA e i rispettivi archi.

3.7 Suffix Tree Generalizzato

Un **Suffix Tree Generalizzato** e' un **Suffix Tree** per un insieme di stringhe. Ovvero e' Suffix Tree di tutti i suffissi w che appartengono a una delle stringhe. I nodi saranno decoranti non piu' solo con la posizione nel testo ma con le posizioni nelle relative stringhe, ovvero un insieme di coppie (stringa, posizione).

Il modo piu' semplice per ottenerlo e' la concatenazione delle stringhe con terminali speciali distinti:
$S_{tot} = S_0 \cdot \$_0 \cdot S_1 \cdot \$_1 \cdot ...S_n \cdot \$_n$.

Come esempio, siano $S_0 = MONDIALE$ e $S_1 = CAMBIALE$. Quindi costruisco $S_{tot} = S_0 \cdot \$_0 \cdot S_1 \cdot \$_1$, ovvero $S_{tot} = MONDIALE \cdot \$_0 \cdot CAMBIALE \cdot \$_1$.

Quindi costruisco il Suffix Tree di S_{tot}. Posso usare il Suffix Tree Generalizzato per cercare la LCS di piu' stringhe al posto di usare un algoritmo banale che avrebbe complessita' temporale $T(S_0, S_1, ...S_n) = \Pi_{i=0}^{n}|S_i|$.

3.8 Pattern Matching con Suffix Array

Visto che il Suffix Array e' un array ordinato lessicograficamente di suffissi, il pattern matching si riduce ad una ricerca dicotomica dei suffissi che hanno come prefisso il pattern P. In particolare:

- Visto che tutti i suffissi del range considerato nella singola chiamata ricorsiva $SA[L, R]$ condividono lo stesso prefisso, si puo' evitare di confrontare $LCP(SA[L], SA[R])$ caratteri.

3.9 Longest Common Substring con Suffix Tree

Come sappiamo, le foglie di un Suffix Tree contengono le informazioni su a quali stringhe appartengono i suffissi descritti dalle foglie stesse.

Siano $Node_i$ l'i-esimo nodo in SuffixTree e $String_j$ la j-esima stringa da confrontare. Si puo' estendere la proprieta' sopra destritta $P(Node_i, String_j) \Leftrightarrow Leaf(Node_i) \wedge String_j \in Node_i$ a tutti i nodi, in questo modo: $P(Node_i, String_j) \Leftrightarrow \exists Node_k \in Node_i \mid P(Node_k, String_j)$.

Ebbene l'algoritmo consiste nel cercare il nodo interno $Node_i$ piu' profondo nel Suffix Tree tale che: $\forall String_j \mid P(Node_i, String_j)$.

3.10 Lowest Common Ancestor

Un algoritmo semplice potrebbe essere questo.

Detti A e B i due nodi di cui si deve calcolare LCA. Detti P_A e P_B i percorsi dalla radice R ai due nodi.

Si confrontano i percorsi fino a trovare dove divergono.

Trovare P_A e P_B impiega $T(n) = O(n)$. Confrontare P_A e P_B si usa $T(n) = O(n)$.

Quindi l'algoritmo ha una complessita' totale di $T(n) = O(n)$.

3.11 Range Minimum Query

Il problema Range Minimum Query consiste nel cercare il minimo tra due intervalli. Si ha in input un array X con $|X| = n$.

Preprocessing Si richiede di indicizzare X in tempo $O(n \cdot logn)$.

Query Quindi si puo' rispondere a RMQ in tempo O(1).

Il problema, dati $i, j \in [0, n)$, calcola il valore minimo compreso in $X[i : j]$ (estremi compresi).

Preprocessing Consiste nel calcolare gli intervalli e su essi calcolare il minimo. Si costruisce un array bidimensionale A indicizzato da x, y che conterra' il minimo della porzione del sotto array $X[x : x + 2^y - 1]$. L'ampiezza di ogni sotto array e' 2^y, quindi vengono calcolati solo gli intervalli che corrispondono a una portenza di 2. Questa matrice ha $n \cdot logn$ elementi.

$$A[x][y] = \begin{cases} X[x] & y = 0 \\ min(X[x][y-1], X[x + 2^(y-1)][y-1]) & y > 0 \end{cases}$$

Query

3.12 Longest Common Substring con Suffix Array

TODO

Chapter 4

Allineamento Sequenziale

4.1 Somiglianza di Sequenze

Si vuole sapere quanto due sequenze siano simili. In genomica questo ha particolarmente peso perche' il Dogma Centrale della Biologia Molecolare enuncia che se due sequenze sono simili in struttura, hanno funzione simile. Inoltre questo puo' essere utile per individuare somiglianza tra piu' sequenze per osservare la catena evolutiva.

La **Distanza di Hamming** e' il numero di caratteri differenti nella stessa posizione di due stringhe. Per distanza invatti si intende una funzione $f : S \times S \to \mathbb{R}^+$ con $S \times S$ insieme di coppie di stringhe. Il calcolo del valore di tale funzione e' molto semplice, ed e' possibile usarla per confrontare la somiglianza di due stringhe o sequenze.

Proprieta'

1. Riflessivita': $d(x, y) = 0 \Leftrightarrow x = y$
2. Simmetria: $d(x, y) = d(y, x)$
3. Disuguaglianza triangolare: $d(x, y) + d(y, z) \leq d(x, z)$

Questa stima di somiglianza ha pero' un importante difetto: e' definito solo tra due sequenze di uguale lunghezza.

4.2 Allineamento

Se due stringhe avessero la stessa lunghezza si potrebbe fare un confronto in colonna, ma non essendo cosi' bisogna ricondursi a questo caso. Per riempire i buchi vengono inseriti degli spazi denotati da un simbolo *indel* $\notin \Sigma$. Questo simbolo indica sia un carattere inserito che uno cancellato. Ovviamente si pongono vincoli per assicurare il rispetto del Rasoio di Occam:

- Non possono esistere colonne solo di *indel*
- Le stringhe estese devono avere la stessa lunghezza

Esistono infiniti Allineamenti, ma non tutti hanno la stessa "qualita'", quindi si cerca l'ottimo.

4.3 Needleman-Wunsch

L'algoritmo usa le soluzioni dei sottoproblemi per trovare la soluzione del problema corrente. Essendo una astrazione del concetto di LCS, non deve stupire la sua somiglianza con esso. Si ragiona per prefissi, $M[i][j] = NW(X_i, Y_j)$.

L'input e' lo stesso: due stringhe X, Y, una matrice dei costi D, ma l'algoritmo individua le sottostringhe P, Q tali che $Allineamento(P, Q) \geq Allineamento(p, q) \forall p, q \in X, Y$.

Sia $A_x = alfabeto(X)$ e $A_y = alfabeto(Y)$. Si usa una matrice dei costi (o "premi") D, dove ogni cella $D[i, j]$ indica quando sono simili $(A_x)_i$ e $(A_y)_j$.

$$M[i][j] = \begin{cases} 0 & i = 0 \wedge j = 0 \\ M[i-1][0] + d(X[i], indel) & i > 0 \wedge j = 0 \\ M[0][j-1] + d(indel, Y[j]) & i = 0 \wedge j > 0 \\ max \begin{cases} M[i-1][j-1] + d(X[i], Y[j]) \\ M[i][j-1] + d(indel, Y[j]) \\ M[i-1][j] + d(X[i], indel) \end{cases} & i > 0 \wedge j > 0 \end{cases}$$

4.4 Allineamento locale

Nel caso in cui due sequenze abbiano una parte limitata simile e tutto il resto dei simboli diversi, il valore complessivo di somiglianza e' relativamente basso. L'Allineamento Globale non riuscirebbe a evidenziare questa somiglianza.

Visto che lo scopo di questo confronto e' anche funzionale, avere due regioni fortemente simili e' interessante: si definisce **Allineamento Locale**.

L'input e' lo stesso: due stringhe X, Y, una matrice dei costi D, ma l'algoritmo individua le sottostringhe P, Q tali che $Allineamento(P, Q) \geq Allineamento(p, q) \forall p, q \in X, Y$.

Sia $A_x = alfabeto(X)$ e $A_y = alfabeto(Y)$. Si usa una matrice dei costi (o "premi") D, dove ogni cella $D[i, j]$ indica quando sono simili $(A_x)_i$ e $(A_y)_j$.

Smith-Waterman E' una variante di Needleman-Wunsch

Si ragiona per suffissi, $H[i][j] = SW(X[n-i:], Y[m-j:])$.

$$H[i][j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ max \begin{cases} 0 \\ H[i-1][j-1] + D[X[i], Y[j]] \\ H[i-1][j] + D[X[i], indel] \\ H[i][j-1] + D[indel, Y[j]] \end{cases} & i > 0 \wedge j > 0 \end{cases}$$

4.5 Distanza di Edit

Si definisce **Distanza di Edit** la quantita di modifiche (sostituzione, cancellazione, inserimento) minimo per rendere uguale due stringhe X, Y. Si ragiona per prefissi, $M[i][j] = NW(X_i, Y_j)$.

$$H[i][j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ M[i][j] & i > 0 \vee j > 0 \wedge X[i] = Y[j] \\ min \begin{cases} M[i-1][j] + 1 \\ M[i][j-1] + 1 \\ M[i-1][j-1] + 1 \end{cases} & i > 0 \wedge j > 0 \wedge X[i] \neq Y[j] \end{cases}$$

4.6 Allineamento con GAP

Si definisce *gap* una sequenza continua di *indel* a seguito dell'allineamento.

Voglio ottimizzare il Needleman-Wunsch per assegnare un costo anche direttamente ad un gap. Quindi sia $P(k)$ = costo di un gap lungo k indel.

$$M[i][j] = \begin{cases} 0 & i = 0 \wedge j = 0 \\ P(i) & i > 0 \wedge j = 0 \\ P(j) & i = 0 \wedge j > 0 \\ max \begin{cases} M[i-1][j-1] + d(X[i], Y[j]) \\ maxk > 0 \mid M[i][j-k] + P(k) \\ maxk > 0 \mid M[i-k][j] + P(k) \end{cases} & i > 0 \wedge j > 0 \end{cases}$$

4.7 Allineamento con Gap affine

Se volessi invece assegnare un costo P_0 all'apertura del gap e contemporaneamento un costo P_e all'estensione del gap, si potrebbe formulare in questo modo:

- $M[i,j] = ottimo(X_i, Y_j)$
- $E_x[i,j] = ottimo(X_i, Y_j)$ vincolato ad avere estensione di gap finale in X_i
- $E_y[i,j] = ottimo(X_i, Y_j)$ vincolato ad avere estensione di gap finale in Y_j
- $N_x[i,j] = ottimo(X_i, Y_j)$ vincolato ad avere apertura di gap alla fine di X_i
- $N_y[i,j] = ottimo(X_i, Y_j)$ vincolato ad avere apertura di gap alla fine di Y_j

$$M[i][j] = max \Big\{ M[i-1, j-1] + D[x_i, y_j] E_x[i,j] + E_y[i,j] N_x[i,j] + N_y[i,j]$$

$$E_x[i][j] = max \Big\{ E_x[i, j-1] + P_e N_x[i, j-1] + P_e$$

$$E_y[i][j] = max \Big\{ E_y[i-1, j] + P_e N_y[i-1, j] + P_e$$

$$N_x[i,j] = M[i, j-1] + P_0 + P_e$$
$$N_y[i,j] = M[i-1, j] + P_0 + P_e$$

4.8 Matrici di Sostituzione

La matrice dei costi D introdotta nei precedenti algoritmi viene comunemente chiamata Matrice di Sostituzione. Semanticamente esprime la similarita' di simboli e caratteri. Di solito vengono ottenute con metodi statistici, ma possono anche essere create arbitrariamente, purche' garantiscano un senso compiuto. Vengono utilizzate per esprimere le mutazioni e le loro frequenze.

PAM Le matrici PAM-k riflettono una distanza espressa in PAM (Percentage Accepted Mutation), ovvero la frequenza delle mutazioni.

Due sequenze si dicono distanti k-PAM se il numero di mutazioni accertate e' $\partial(X,Y) = (\frac{k}{1}00)|X|$, ovvero la percentuale dei caratteri mutati e' **circa** k.

Una PAM-k viene costruita selezionando varie sequenze distanti k-PAM, quindi queste vengono allineate e si calcolano le frequenze per i singoli caratteri e le coppie di caratteri in base agli allineamenti ottenuti.

$$\text{PAM-k}[i,j] = log(\frac{f(i,j)}{f(i)f(j)})$$

BLOSUM Mentre le PAM allineano sequenze vicine, le matrici BLOSUM confrontanto sequenze lontane e le regioni mutate hanno la stessa importanza di quelle non mutate.

Il valore BLOSUM-k si riferisce al valore di soglia percentuale k usato per determinare se due sequenze sono simili e quindi se devono essere aggregate prima di effettuare l'analisi di frequenza.

Quella piu' usata e' la BLOSUM62.

4.9 Karlin-Altschul

La statistica di Karlin-Altschul esprima la probabilita' che l'allineamento locale venga trovato tra due sequenze della stessa lunghezza:

$$E = kmne^{-\lambda S}$$

variabile	valore
E	numero di allineamenti
k	costante
n	numero di caratteri nel database
m	lunghezza stringa query
λS	punteggio normalizzato

4.10 Basic Local Alignment Search Tool

E' un algoritmo per confrontare le sequenze di amminoacidi o nucleotidi o comunque sequenze di DNA o RNA.

- Si rimuovono le regioni della sequenza a "bassa complessita'", ovvero composte da pochi elementi uguali, perche' potrebbero drogare il risultato con punteggi inverosimilmente alti.

- Si crea una lista di tutte le sequenze consecutive lunghe 3 presenti della sequenza in input.
- Si confrontano le parole in questa lista con tutte le parole presenti nel database utilizzano una matrice di sostituzione per attribuire un punteggio di similarita', come per esempio BLOSUM62.
- Vengono scartate le parole con punteggio basso,
- Per tutte le parole che sono rimaste con punteggio alto si inizia a estendere a destra e a sinistra di un carattere rispettivamente finche' il relativo punteggio non cala.
- Si listano tutte queste regioni estese e si fondono quelle vicine
- Si usa l'algoritmo Smith-Waterman

Chapter 5

Grafi

5.1 Assemblaggio di genomi

I genomi vengono sequenziati in pezzi, chiamati *read*, con dimensione 50-10000 base pairs. Spesso vengono sequenziati in coppie e con posizione originaria ignota.

L'obiettivo resta riscostruire il genoma, che ha circa 3 miliardi di base pairs.

Le tecnologie in nostro possesso hanno vari range di efficienza. Alcune richiedono piu' read per comporre un sequenziamento lungo, con pero' le singole read piu' veloci, altre l'opposto. Inoltre le varie tecnologie hanno anche tipologie di errori differenti. Alcune soffrono di errori sistematici, altri ricorrenti. Ma e' bene comunque ricordare che sono stati fatti passi da gigante in 20/30 anni.

5.2 Overlap

L'overlap e' la condizione di due sequenze in cui condividono una porzione di esse che e' rispettivamente suffisso e prefisso.

Esempio:

```
   TCTATATCTCGGCTCTAGG--
--|||||||  -||||||--
--TATCTCGACTCTAGGCCC
```

Probabilente l'overlap avviene perche' queste due sequenze sono read di porzioni consequenti, ma puo' anche capitare che sia per colpa del fatto che l'essere umano e' un organismo diploide. Infatti disponendo due filoni genitici (tendenzialmente uno materno e l'altro paterno) si possono avere errori di questo tipo.

L'obiettivo qui e' ricostruire il filone che ha prodotto queste due read.

5.3 Greedy Algorithm

Si puo immaginare di creare un algoritmo greedy come segue:

- trovare le due read nel batch con overlap massimo
- fondere queste due read e inserire nel batch la stringa ottenuta al posto delle due read

- continuare fino a che non rimane solo una stringa nel batch

Il problema e' che questo algoritmo non identifica correttamente i genomi con sequenze ripetute. Per esempio:

1. ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long **g_time ng_tim**
2. ng_time ng_lon **_long_** a_long long_l ong_ti ong_lo long_t **g_long**
3. ng_time g_long_ ng_lon a_long long_l **ong_ti** ong_lo **long_t**
4. ng_time long_ti g_long_ **ng_lon** a_long long_l **ong_lo**
5. **ng_time** ong_lon **long_ti** g_long_ a_long long_l
6. **ong_lon** long_time g_long_ a_long **long_l**
7. long_lon **long_time g_long_** a_long
8. **long_lon g_long_time** a_long
9. **long_long_time a_long**
10. a_long_long_time

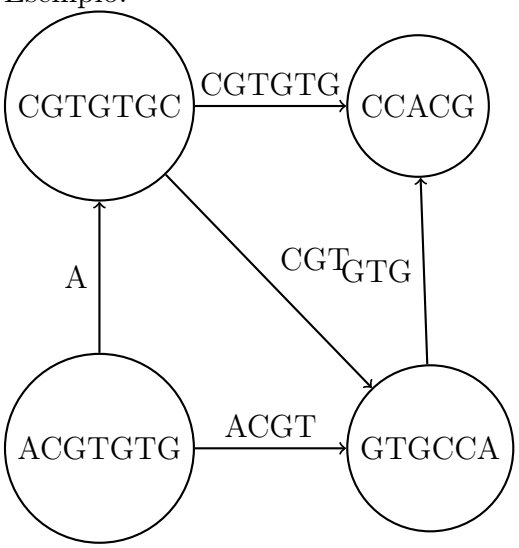
La sequenza originale era **a_long_long_long_time** tuttavia quella ottenuta e' **a_long_long_time**.

5.4 Grafi di overlap

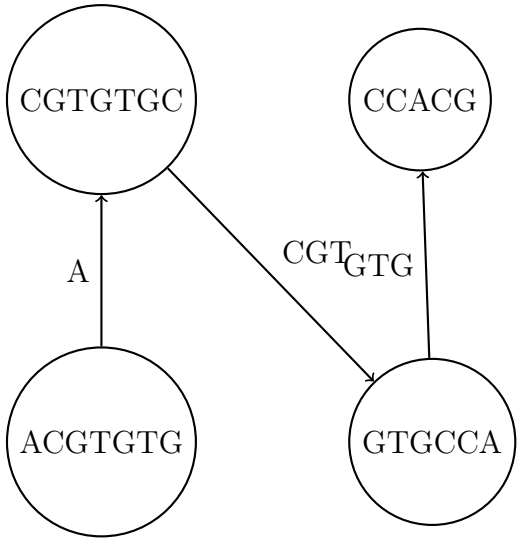
Sia R un insieme di read su un genoma G.

Si costruisce il grafo $G = (V, E)$, tale che $V \cong R$ e vi sia un arco fra tutte le coppie di vertici corrispondenti a read che abbiano overlap abbastanza lungo.

Esempio:



Per trovare una soluzione si cerca un percorso che rimuova gli archi transitivi del grafo di overlap, per esempio:



Questa e' la shortest superstring.

5.5 Problema del Commesso Viaggatore (TSP)

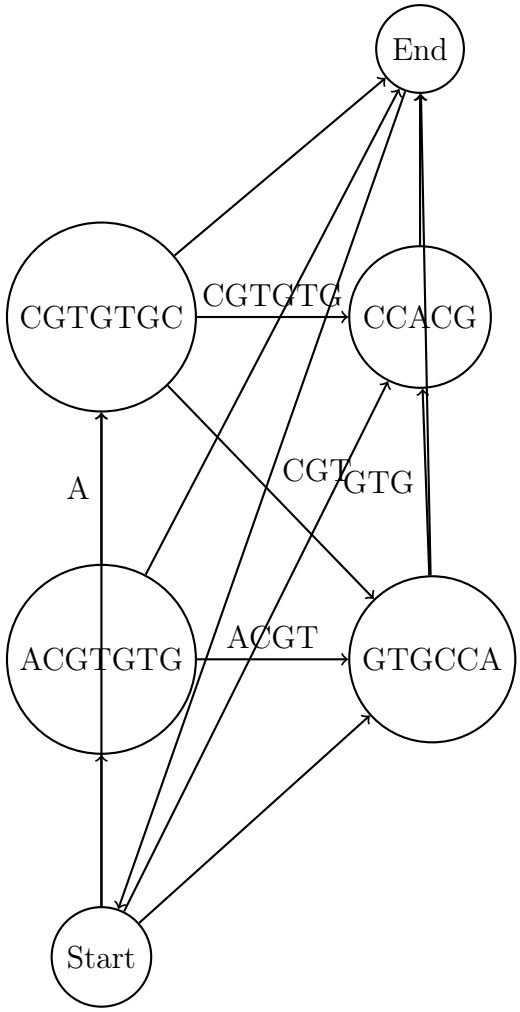
E' un problema che dato un grafo $G = (V, E)$ trova il cammino di minimo costo tale che tutti i nodi vengano attraversati una sola volta e che sia nella forma $w = s \cdot v_1 \cdot v_2 ... \cdot v_n \cdot s$ (ovvero formi un ciclo).

Per risolvere il grafo di overlap si necessita di trovare il cammino di minimo costo che attraversi tutti i nodi una sola volta, ma tale che sia aciclico.

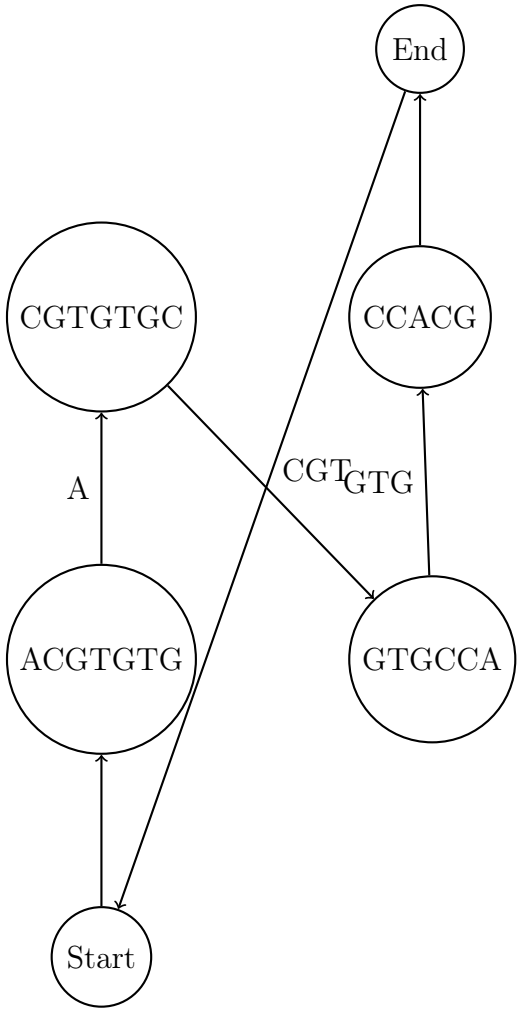
Per fare questo si puo' sfruttare TSP e modificare il grafo inserendo due vertici e degli archi che li colleghino al resto del grafo in modo da poter trasformare il problema in TSP.

Per ricostruire la soluzione bastera' escludere Start e End, i quali appariranno per forza consecutivamente nel ciclo TSP.

Esempio:



Che si risolve in:

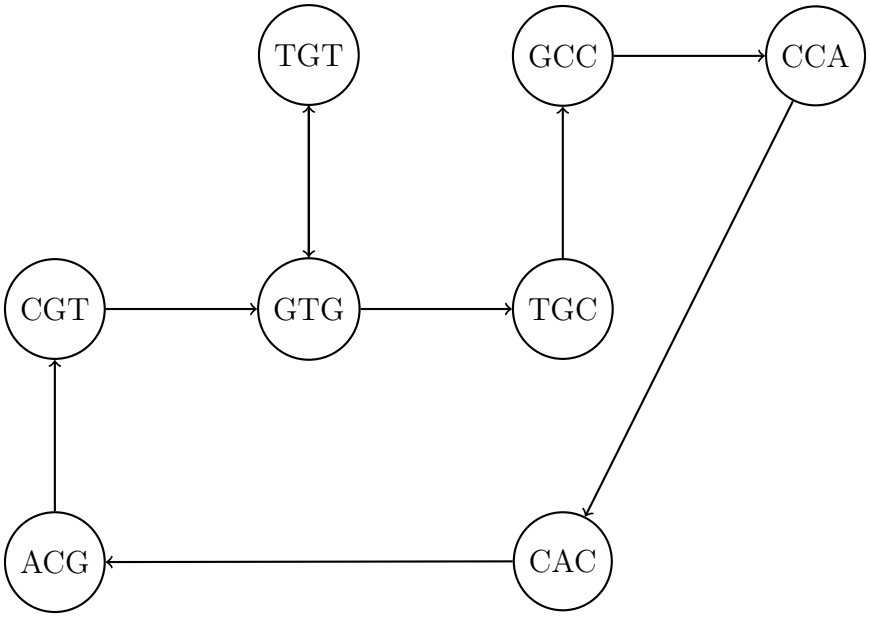


5.6 SBH

Una tecnologia vecchia se si usava erano i DNA array. Per ogni k-mero si conosce se appare nel genoma. Spesso si usa $k \cong 8$.

Ogni k-mero viene diviso in (k-1)-meri, ognuno dei quali sara' un vertice del grafo di overlap. Si avra' invece un arco per ogni k-mero.

Quindi si applica TSP. Questa procedura e' detta Grafo di de Bruijn. In questo caso non possiamo sapere quante volte un certo k-mero appare nel genoma ma solo se appare.



5.7 Ciclo Euleriano

Un assemblaggio valido di un Grafo di de Bruijn e' un cammino euleriano, ovvero un cammino che attraversa ogni arco esattamente una volta.

Sia $N_G^u(v)$ il numero di archi uscenti da v e $N_G^e(v)$ il numero di archi entranti in v. Sia G = (V, E) un grafo orientato. G e' semi-euleriano se esistono due vertici s, t tali che $N_G^u(s) = N_G^e(s) + 1$ e $N_G^u(t) = N_G^e(t) - 1$ mentre per ogni altro vertice v vale che $N_G^u(v) = N_G^e(v)$.

Sia G = (V, E) un frafo orientato. G e' euleriano se per ogni vertice v vale che $N_G^u(v) = N_G^e(v)$.

Sia G = (V, E) un grafo connesso. G ha un cammino euleriano sse G e' un grafo semi-euleriano. Inoltre G ha un ciclo euleriano sse G e' un grafo euleriano.

Siano G = (V, E) un grafo semi-euleriano e P un cammino da s a t. Sia G_1 il grafo ottenuto da G eliminando tutti gli archi di P. G_1 e' euleriano.

Siano G = (V, E) un grafo euleriano e C un ciclo di G. Sia G_1 il grafo ottenuto da G eliminando tutti gli archi di C. G_1 e' euleriano.

5.8 Riduzione del Grafo di Overlap

Teoria Sia G = (V, E) un grafo di overlap con $(a \to b_1)$ unico arco irriducibile uscente da a e con (a, b_1),...,(a,b_n) archi uscenti da a. Allora $(b_i \to b_{i+1})$ con $1 \leq i \leq n-1$ sono archi di G.

Sia G = (V, E) un grafo di overlap con $(a \to b_1)$ unico arco irriducibile uscente da a e con (a, b_1),...,(a,b_n) archi uscenti da a. Allora $(b_i \to b_i)$ con $2 \leq i \leq n-1$ sono archi di G.

Algoritmo

1. b_i vengono ordinati per peso dell'arco
2. si marcano "da-eliminare" i vertici b_j tali che $(b_i \to b_j)$ con $i < j$

3. si rimuovono gli archi che terminano in vertici "da-eliminare"

Chapter 6

Alberi Evolutivi e Filogenesi

6.1 Evoluzione

L'Evoluzione e' l'effetto di una sequenza di mutazioni che a lungo termine porta a dei cambiamenti sostanziali e diffusi nella composizione del genoma di una specie.

Le cellule subiscono continue mutazioni durante la loro vita. Alcune fanno parte di un processo intrinseco di mutazione, altreno invece sono provocate da agenti esterni o dall'ambiente in cui ci si ritrova.

La teoria moderna dell'evoluzione e' basata su caratteri che vengono acquisiti una sola volta nell'albero evolutivo.

6.2 Algoritmo di Gusfield

La filogenesi e' il processo di ramificazione delle linee di discendenza nell'evoluzione della vita.

Sia M la matrice $n \cdot n$ che ha come colonne i caratteri e come righe gli organismi La cella $M[i, j]$ indica con un numero $\{0, 1\}$ se l'organismo i-esimo ha il carattere j-esimo.

Sia per esempio la seguente matrice:

	A	J	H	L	V
Scorpione	0	0	0	0	0
Anguilla	0	0	0	0	1
Tonno	0	1	0	0	1
Salamandra	0	1	0	1	1
Tartaruga	1	1	0	1	1
Leopardo	1	1	1	1	1

Abbiamo bisogno di un algoritmo che prenda in input una matrice binaria siffatta e restituisca in output l'albero che spiega M, se esiste. Usiamo per esempio l'algoritmo di Gusfield.

Si usa prima il Radix Sort sulle colonne in ordine decrescente (anche del numero di 1), quindi si passa a costruire l'albero una specie alla volta. Quindi si fa:

- Considerando ogni colonna binaria come un numero binario si riordinano con il Radix Sort in ordine descrescente in modo da avere il numero piu' grande in colonna 1

- Per ogni riga p di M_o si costruisce la stringa dei caratteri, in ordine crescente, che p possiede
- Si costruisce l'albero dei suffissi T per le n stringhe cosi' costruite
- Si testa se T e' una filogenesi perfetta per M (ovvero se la spiega)

Caratteri e Stato Un carattere c e' acquisito quando lo stato di c passa da 0 a 1 in un arco. Un carattere c e' perso quando lo statoc di c passa da 1 a 0 in un arco. (Mutazione ricorrente)

Modelli di evoluzione Ogni carattere c e' acquisito esattamente una volta nell'albero.

- Filogenesi perfetta: nessuna mutazione ricorrente, nessuna perdita
- Dollo: mutazioni ricorrenti senza limiti, ma senza perdite
- Camin-Sokal: perdite senza limiti, ma senza mutazioni ricorrenti

6.3 Approcci basati su parsimonia

I problemi di parsimonia si dividono in due categorie: piccola parsimonia (topologia nota) e massima parsimonia (topologia ignota).

Per i problemi di piccola parsimonia si trattano due algoritmi: Fitch e Sankoff.

Istanza del problema Input:

- Matrice M con n specie in S e m caratteri in C
- Albero T le cui foglie corrispondono alle specie di M
- Per ogni carattere un costo w_c fra ogni coppia di stati

Output:

Per ogni carattere c una etichettatura λ_c che assegna ad ogni nodo uno degli stati possibili per c

Funzione obiettivo:

$min\Sigma_{c\in C}\Sigma_{(x,y)\in archi} w_c(\lambda_c(x), \lambda_c(y))$

6.4 Algoritmo di Sankoff

Ogni carattere e' gestito separatamente. Si usa la programmazione dinamica.

Sia P un matrice con $P[x, z]$ = soluzione ottimale del sottoalbero T che ha radice x sotto la condizione che abbia etichetta z.

Caso base Se x e' una foglia con etichetta z allora $P[x, z] = 0$.

Se x e' una foglia con etichetta diversa da z allora $P[x, z] = +\infty$.

Caso ricorsivo Se x e' un nodo interno, detto F(x) = insieme di nodi figli di x allora:
$P[x, z] = \Sigma_{f \in F(x)} min_s\{w(z, s) + P[f, s]\}$.
La soluzione ottimale e' $min_s\{P[r, s]\}$, dove r e' la radice di T.

6.5 Algoritmo di Fitch

Solo per il caso non pesato di un albero T binario.

Con $S(x)$ = insieme degli stati ottimali per il nodo x.

Se x e' una foglia allora $S(x) = \lambda_c(x)$.

Se x e' un nodo interno, siano a e b si suoi figli in T. Se $S(a) \cap S(b) = \emptyset$ allora $S(x) = S(a) \cup S(b)$. Altrimenti $S(x) = S(a) \cap S(b)$.

6.6 Approcci basati su distanza

Sia $d : SxS \to N.$ tale che:

$$\begin{aligned} d(a, b) = 0 \Leftrightarrow a = b & \qquad \forall a, b \in S \\ d(a, b) = d(b, a) & \qquad \forall a, b \in S \\ d(a, b) \leq d(a, c) + d(c, b) & \qquad \forall a, b, c \in S \end{aligned}$$

6.7 UPGMA

UPGMA sta per Unweighted Pair Group with Arithmetic Mean/

1. Per ogni specie si crea un cluster C_i.
2. Quindi si definisce la funzione di distanza $D : CxC \to N$ con $D(C_a, C_b) = \frac{\Sigma_{i \in C_a} \Sigma_{j \in C_b} d(i,j)}{|C_a| \cdot |C_b|}$.
3. All'inizio $h(C_i) = 0 \forall i$.
4. Quindi si fondono i due cluster C_a, C_b piu' vicini, ovvero tali che $min\{D(C_a, C_b) \forall a, b\}$.
5. La fusione di C_a, C_b la chiamo C_x.
6. Per ogni cluster $C_y \neq C_x$, calcolo la distanza $D(C_x, C_y)$.
7. Quindi $h(C_x) \leftarrow \frac{D(C_a, C_b)}{2}$
8. (C_x, C_a) e' etichettato con $h(C_x) - h(C_a)$
9. (C_x, C_b) e' etichettato con $h(C_x) - h(C_b)$

UPGMA produce una ultrametrica. Una ultrametrica e' uno spazio metrico (basato su distanza) in cui la diseguaglianza triangolare e' rinforzata. Ovvero in cui vale anche che $d(a, b) \leq max\{d(a, c), d(c, b)\} \forall a, b, c \in S$.

6.8 Modelli di Evoluzione

TODO

Chapter 7

Aplotipi

7.1 Ricostruzione aplotipi in pedigree

Istanza del problema Abbiamo un Pedigree P, per ogni individuo il suo genotipo (00, 11, 01), nessuna ricombinazione e nessun dato mancante. Siamo nel caso biallelico.

Variabili

- $g_i[j]$ = genotipo dell'individuo i-esimo nel locus j. Vale 0 o 1 se omozigote, 2 se eterozigote.
- $w_i[j] = 0$ se nel locus j-esimo dell'individuo i-esimo e' omozigote, 1 se eterozigote.
- detto x genitore di i, se x passa il suo aplotipo paterno a i allora $h_{xi} = 0$, altrimenti $= 1$.
- $p_i[j]$ = aplotipo paterno dell'individuo i-esimo nel locus j-esimo.

$(g_i[j], g_i[j]) = (0, 1)$ se omozigote, oppure $= (1, 0)$ se eterozigote. L'aplotipo materno e' $= p_i[j] + w_i[j] \in \mathbb{Z}_2$

7.2 Eretitarieta' di Mendel

Variabili

- p padre, m madre, f figlio/a, x genitore
- $h_{pf} = 0 \Rightarrow p_f[j] = p_p[j]$
- $h_{pf} = 1 \Rightarrow p_f[j] = p_p[j] + w_p[j]$
- $h_{mf} = 0 \Rightarrow p_f[j] + w_f[j] = p_m[j]$
- $h_{mf} = 1 \Rightarrow p_f[j] + w_f[j] = p_m[j] + w_m[j]$
- $p_f[j] = p_p[j] + w_p[j] \cdot h_{pf}$
- $p_f[j] + w_f[j] = p_m[j] + w_m[j] \cdot h_{mf}$
- $p_f[j] + d_{xf} = p_x[j] + w_x[j] \cdot h_{mf}, dove d_{pf} = 0 e d_{mf} = w_f$

Equazioni

- $p_f[j] + d_{xf} = p_x[j] + w_x[j] \cdot h_{xf}$ per ogni coppia (x, f) genitore-figlio
- $d_{pf} = 0$ per ogni coppia (p, f) padre-figlio
- $d_{mf} = w_f$ per ogni coppia (m, f) madre-figlio
- $p_i[j] = g_i[j]$ se $g_i[j] \neq 2$
- $w_i[i] = 1$ se $g_i[j]$
- $w_i[i] = 0$ altrimenti

Incognite

- $p_x[j]$
- h_{xf}

Part II

Esercitazione

Part III

Laboratorio