Elementi di Bioinformatica

Gianluca Della Vedova

Univ. Milano–Bicocca
http://gianluca.dellavedova.org

22 ottobre 2017, revisione 030ec07

Gianluca Della Vedova

- Elementi di Bioinformatica
- Ufficio U14-2041
- https://gianluca.dellavedova.org
- https://elearning.unimib.it/course/view.php?id=15423
- gianluca.dellavedova@unimib.it
- https://github.com/bioinformatica-corso/programmi-elementi-bioinformatica
- https://github.com/bioinformatica-corso/lezioni

Notazione

- **simbolo**: $T[i]$
- **stringa**: $T[1]T[2]\cdots T[l]$
- **sottostringa**: $T[i : j]$
- **prefisso**: $T[: j] = T[1 : j]$
- **suffisso**: $T[i :] = T[i : |T|]$
- **concatenazione**: $T_1 \cdot T_2 = T_1T_2$

Pattern Matching

Problema

Input: testo $T = T[1]\cdots T[n]$, pattern $P = P[1]\cdots P[m]$, alfabeto Σ
Goal: trovare *tutte* le occorrenze di P in T
Goal: trovare tutti gli i tale che $T[i]\cdots T[i + m - 1] = P$

Algoritmo banale

Tempo: $O(nm)$

Lower bound

Tempo: $O(n + m)$

Bit-parallel

Algoritmi seminumerici

- 25
- 25 = 00011001
- 25 = 00011001 =FFFTTFFT

Operazioni bit-level

Or: $x \vee y$, **And**: $x \wedge y$, **Xor**: $x \oplus y$
Left Shift: $x << k$, **Right Shift**: $x >> k$,

- Tutte bitwise
- Tutte in hardware

Dömölki / Baeza-Yates, Gonnet

Matrice M

$M(i, j) = 1$ sse $P[: i] = T[j - i + 1 : j]$
$0 \le i \le m$, $0 \le j \le n$

Occorrenza di P in T

$M(m, \cdot) = 1$

- $M(0, \cdot) = 1$, $M(\cdot, 0) = 0$
- $M(i, j) = 1$ sse $M(i - 1, j - 1) = 1$ AND $P[i] = T[j]$

Esempio

Esempio

T=abracadabra
P=abr

10010101001
01000000100
00100000010 ← **occorrenze**

Matrice M

1 colonna = 1 numero

Colonne

$U[\sigma]$ = array di bit dove $U[\sigma, i] = 1$ sse $P[i] = \sigma$

$C[j]$ da $C[j - 1]$

- Right shift di $C[j - 1]$
- 1 in prima posizione
- AND con $U[T[j]]$
- ω: word size
- $C[j] = ((C[j - 1] >> 1) \mid (1 << (\omega - 1)))$ & $U[T[j]]$;

Note

- Tempo $O(n)$ se $m \leq \omega$
- Tempo $O(nm)$
- No condizioni
- $\omega < m \leq 2\omega$?

Karp-Rabin

Alfabeto binario

- $H(S) = \sum_{i=1}^{|S|} 2^{|S|-i} H(S[i])$
- sliding window di ampiezza m su T
- $H(T[i+1 : i+m]) =$
  $= (H(T[i : i+m-1]) - T[i]) /2 + 2^{m-1} T[i+m]$
- operazioni su bit
- $T[i : i+m-1] = P \Leftrightarrow H(T[i : i+m-1]) = H(P)$

Karp-Rabin: problema

Numeri troppo grandi

- Modello RAM: numeri $O(n+m)$
- mod p
- $H(T[i+1 : i+m]) =$
  $\left((H(T[i : i+m-1]) - T[i]) /2 + 2^{m-1} T[i+m]\right) \mod p$
- **NO**
- $2^{m-1} T[i+m] \mod p$ calcolato iterativamente, $\mod p$ ad ogni passo

Karp-Rabin: falsi positivi

Possibili errori

- Falso positivo (FP): occorrenza non vera
- Falso negativo (FN): occorrenza non trovata
- $H(T[i : i+m-1]) = H(P) \Leftrightarrow T[i : i+m-1] = P$
- $H(T[i : i+m-1]) \mod p = H(P) \mod p$
  $\Leftarrow T[i : i+m-1] = P$

Karp-Rabin: falsi positivi

Probabilità di errore

$P[\#FP \geq 1] \leq O(nm/I)$ se il numero primo p è scelto fra tutti i primi $\leq I$

Valori di I

- $I = n^2 m \Rightarrow P[\#FP \geq 1] \leq 2.54/n$
- $I = nm^2 \Rightarrow P[\#FP \geq 1] \in O(1/m)$

Abbassare probabilità di errore

Scegliere k primi casuali (indipendenti senza ripetizioni), cambiare primo dopo ogni FP

Las Vegas vs. Monte Carlo

Classificazione algoritmi probabilistici

- Las Vegas:
  - Sempre corretto
  - Forse non veloce
  - Quicksort con pivot random
- Monte Carlo:
  - Sempre veloce
  - Forse non corretto
  - Karp-Rabin

Controllo falsi positivi

L: posizioni iniziali in T delle occorrenze

Run

sequenza $\langle l_1, \ldots, l_k \rangle$ di posizioni in L distanti al massimo $m/2$

- $d = l_2 - l_1$
- P semiperiodico con periodo d
- $P = \alpha\beta^{k-1}$, α suffisso di β
- ogni run occupa $\geq n$ caratteri di T
- ogni carattere di T è in max 2 run

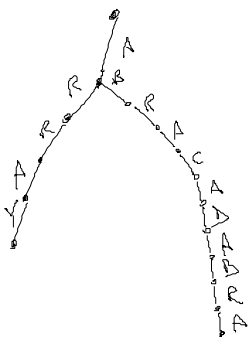
Trie

Trie

- Albero
- Query: parola $\in$ dizionario
- archi etichettati
- Percorso radice-foglia = parola

Dizionario

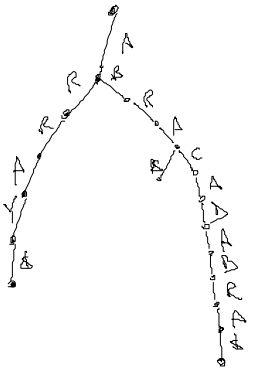
ABRACADABRA
ARRAY
ABRA



Trie

Terminatore
$ non appartiene all'alfabeto

Dizionario
ABRACADABRA$
ARRAY$
ABRA$



Suffix tree

Definizione
- Trie compatto di tutti i suffissi di T\$
- Le etichette degli archi uscenti da x iniziano con simboli diversi
- suffissi ⇔ percorso radice-foglia

BANANA$

Suffix tree 2: Definizione

- foglie etichettata con posizione inizio suffisso
- path-label(x): concatenazione etichette
- string-depth(x): lunghezza path-label(x)
- Pattern matching = visita

Problemi
- Spazio $O(n^2)$
- Puntatori al testo (posizioni)
- Spazio $20n$ bytes

Suffix array

Definizione
- Array dei suffissi in ordine lessicografico
- Posizioni iniziali del suffisso nell'array
- Spazio $4n$ bytes
- $Lcp[i]$: lunghezza prefisso comune $SA[i]$, $SA[i+1]$

BANANA$

i	0	1	2	3	4	5	6
SA	7	6	4	2	1	5	3
Lcp	0	1	3	0	0	2	-

Da Suffix tree a Suffix array

- Visita depth-first di ST
- archi uscenti di ogni nodo in ordine lessicografico
- $Lcp[i]$ = string-depth di $lca(i, i+1)$

Da Suffix array a Suffix tree

- $Lcp = 0$: partizione SA radice
- ricorsione prendendo i numeri minimi

Sottostringa comune più lunga di due stringhe

Due stringhe s_1 e s_2
- Suffix tree generalizzato = insieme di stringhe
- $ST(s_1\$_1 s_2\$_2)$
- Nodo x con foglie di s_1 e s_2
- Sottostringa di s_1 e s_2
- $ST(s_1\$s_2\$)$
- Max string-depth

Suffix tree generalizzato

s_1: BANANA$, s_2: PANNA$

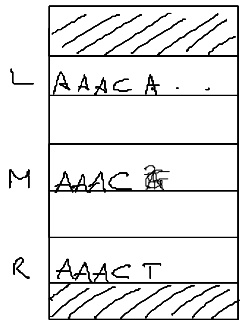
Pattern matching su suffix array

Occorrenza P in T
Suffissi di T che iniziano con P

Ricerca in SA
- Ricerca dicotomica
- Tempo $O(m \log n)$ – caso pessimo
- Controllare tutto P ad ogni iterazione
- $log_2 n$ iterazioni

Acceleranti 1

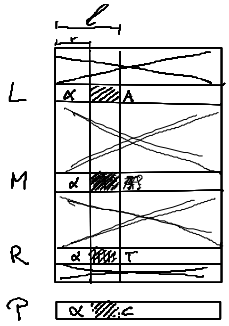
- Intervallo $SA(L, R)$ di SA
- Elemento mediano M
- Tutti i suffissi in $SA(L, R)$ iniziano con uno stesso prefisso lungo $Lcp(SA[L], SA[R])$
- Non confrontare con i primi $Lcp(SA[L], SA[R])$ caratteri



Acceleranti 2

l: $lcp(L, P)$; r: $Lcp(R, P)$

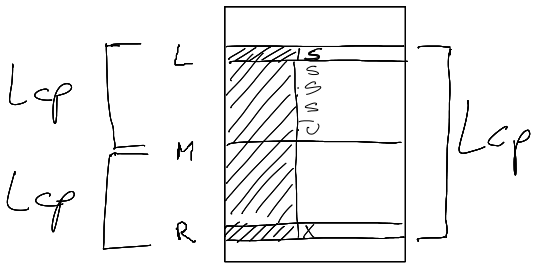
1. Caso 1: $l > r$
   $Lcp(L, M) > l \Rightarrow L \leftarrow M$
   $Lcp(L, M) < l \Rightarrow$
   $R \leftarrow M, r \leftarrow Lcp(M, L)$
   $Lcp(L, M) = l \Rightarrow$ confronto
   $P[l + 1 :], M[l + 1 :]$
2. Caso 2: $l = r$
   $Lcp(L, M) > l$
   $Lcp(M, R) > l$
   $Lcp(L, M) = Lcp(M, R) = l$



Acceleranti 2: calcolo Lcp in tempo $O(n)$

- Iterazione 1: $(L, R) = (1, m)$
- Iterazione 2: $(L, R) = (1, m/2)$ oppure $(m/2, m)$
- Iterazione k: $L = h\frac{m}{2^{k-1}}$, $R = (h + 1)\frac{m}{2^{k-1}}$
- Iterazione $\lceil \log_2 m \rceil$: $R = L + 1$, $Lcp(h, h + 1)$
- Iterazione $\lceil \log_2 m \rceil - 1$: aggrego i risultati dell'iterazione $\lceil \log_2 m \rceil$
- Iterazione k: $Lcp(h\frac{m}{2^{k-1}}, (h + 1)\frac{m}{2^{k-1}})$
- $= \min\{Lcp(2h\frac{m}{2^k}, (2h + 1)\frac{m}{2^k}), Lcp((2h + 1)\frac{m}{2^k} + 1, (2h + 2)\frac{m}{2^k}), Lcp((2h + 1)\frac{m}{2^k}), Lcp((2h + 1)\frac{m}{2^k} + 1)\}$

Acceleranti 2: calcolo Lcp in tempo $O(n)$



Passaggio da y a z deve esistere

Acceleranti 2: Osservazione

- Tempo per trovare un'occorrenza
- Tempo per trovare tutte le occorrenze?
- $O(n + m + k)$, per k occorrenze

Costruzione suffix array: nuovo alfabeto

- Alfabeto Σ con σ simboli, testo T lungo n
- Aggrego triple di caratteri
- Alfabeto Σ^3 con σ^3 simboli, testo lungo $n/3$
- $T_1 = (T[1], T[2], T[3]) \cdots (T[3i + 1], T[3i + 2], T[3i + 3]) \cdots$
  $T_2 = (T[2], T[3], T[4]) \cdots (T[3i + 2], T[3i + 3], T[3i + 4]) \cdots$
  $T_0 = (T[3], T[4], T[5]) \cdots (T[3i], T[3i + 1], T[3i + 2]) \cdots$
- suffissi$(T) \Leftrightarrow \bigcup_{i=0,1,2}$ suffissi(T_i)

Costruzione suffix array: ricorsione

1. Ricorsione su T_0T_1
2. suffissi$(T_0T_1) \Leftrightarrow$ suffissi(T_0), suffissi(T_1)
3. suffissi$(T_0T_1) \Leftrightarrow$ suffissi(T_2)
4. $T_2[i :] \approx T[3i + 2 :]$
5. $T[3i + 2 :] = T[3i + 2]T[3i + 3 :] = T[3i + 2]T_0[i + 1 :]$
6. suffissi(T_0) ordinati
7. Radix sort
8. Fusione suffissi(T_0T_1), suffissi(T_2)

Costruzione suffix array: fusione

Confronto suffisso di T_0 e T_2

1. $T_0[i:] <=> T_2[j:]$
2. $T[3i:] <=> T[3j+2:]$
3. $T[3i]T[3i+1:] <=> T[3j+2]T[3j+3:]$
4. $T[3i]T_1[i:] <=> T[3j+2]T_0[j+1:]$

Costruzione suffix array: fusione

Confronto suffisso di T_1 e T_2

1. $T_1[i:] <=> T_2[j:]$
2. $T[3i+1:] <=> T[3j+2:]$
3. $T[3i+1]T[3i+2:] <=> T[3j+2]T[3j+3:]$
4. $T[3i+1]T[3i+2]T[3i+3:] <=> T[3j+2]T[3j+3]T[3j+4:]$
5. $T[3i+1]T[3i+2]T_0[i+1:] <=> T[3j+2]T[3j+3]T_1[j+1:]$

KS

1. Juha Kärkkäinen, Peter Sanders and Stefan Burkhardt. Linear work suffix array construction. J. ACM, 53 (6), 2006, pp. 918-936.
2. Difference cover (DC) 3
3. Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking In Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03), LNCS 2676, Springer, 2003, pp. 55-69. `http://www.stefan-burkhardt.net/CODE/cpm_03.tar.gz`
4. Yuta Mori. SAIS `https://sites.google.com/site/yuta256/`

Sottostringa comune più lunga

k stringhe $\{s_1, \ldots, s_k\}$

1. Suffix tree generalizzato
2. Vettore $C_x[1:k]$ per ogni nodo x
3. $C_x[i]$: sottoalbero con radice x ha una foglia di s_i
4. $C_x = \bigvee C$ sui figli di C
5. Nodo z, C_z = tutti 1
6. Tempo $O(kn)$
7. n: summa lunghezze $|s_1| + \cdots + |s_k|$

Lowest common ancestor (lca)

Dati albero T e 2 foglie x, y

- z è antenato comune di x, y se z è antenato di entrambi x e y
- z è lca di x, y se:
  1. z è antenato comune di x e y
  2. nessun discendente di z è antenato comune di x e y

Proprietà

- Preprocessing di T in tempo $O(n)$
- Calcolo lca(x, y) in tempo $O(1)$
- Algoritmo complesso, ma pratico

Sottostringa comune più lunga di k stringhe

Arricchimento ST

1. $N_x[i]$: numero foglie di s_i discendenti di x
2. $N_x[i] = 0$ o 1 per ogni foglia
3. $N_x[i]$ = somma dei figli
4. $D_x[i]$: numero di consecutive di foglie di s_i, ordinate secondo visita depth-first, discendenti di x
5. $N_x[i] = 0 \Rightarrow D_x[i] = 0$
6. $N_x[i] = 1 \Rightarrow D_x[i] = 0$
7. $N_x[i] \geq 1 \Rightarrow D_x[i] = N_x[i] - 1$
8. $N_x[i] - D_x[i] = C_x[i]$

Sottostringa comune più lunga di k stringhe

Gestione ST

- Visita depth-first di ST
- L_i: lista ordinata delle foglie di s_i
- Per ogni coppia x, y consecutiva in L_i
  1. $z \leftarrow lca(x, y)$
  2. $D_z[i] =$
  3. Aggiorna C_z

Allineamento di 2 sequenze

Allineamento

- Input: 2 sequenze s_1 e s_2
- Aggiunta di **indel** in s_1 e s_2
- sequenze estese = stessa lunghezza
- NO colonne di indel

Allineamento: esempio

Input

```
ABRACADABRA
BANANA
```

sequenze allineate 1

```
ABRACADABRA
-B-ANA---NA
```

sequenze allineate 2

```
ABR-AC-ADABRA
---B-ANA---NA
```

sequenze allineate 3

```
ABRACADABRA
-BANA----NA
```


Allineamento: costo o valore?

Problema di ottimizzazione

- Istanza: insieme infinito di casi
- Soluzioni ammissibili: ammissibilità verificabile in tempo polinomiale
- Funzione obiettivo: Istanza $\mapsto \mathbb{R}^+$
- Massimizzazione o minimizzazione

costo o valore?

- Costo da minimizzare
- Valore da massimizzare

Valore

Valore di un allineamento

- Somma dei valori delle singole colonne
- Valore di una colonna =
- valore in ingresso

Istanza

- due sequenze s_1 e s_2
- matrice di score $d : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \mapsto \mathbb{R}^+$
- problema di massimizzazione = massima omologia

Needleman-Wunsch: Equazione di ricorrenza

Definizione

$M[i, j]$ = ottimo su $s_1[: i]$, $s_2[: j]$

$$M[i,j] = \max \begin{cases} M[i-1, j-1] + d(s_1[i], s_2[j]) \\ M[i, j-1] + d(-, s_2[j]) \\ M[i-1, j] + d(s_1[i], -) \end{cases}$$

Condizione al contorno

- $M[0, 0] = 0$
- $M[i, 0] = M[i-1, 0] + d(s_1[i], -)$
- $M[0, j] = M[0, j-1 + d(-, s_2[j])$

Allineamento locale

Allineamento globale

Si allineano le sequenze intere

Allineamento locale

1. Input: s_1, s_2, matrice di score d
2. Individuare sottostringhe t_1 di s_1 e t_2 di s_2 tale che
3. $All[t_1, t_2] \geq All[u_1, u_2]$ per ogni coppia di sottostringhe u_1, u_2 di s_1, s_2.
4. Algoritmo banale: calcolo tutte le sottostringhe di s_1, s_2 e ne calcolo allineamento globale
5. Tempo $O(n^3 m^3)$

Smith-Waterman

Osservazione 1

1. Matrice $M[i, j]$ memorizza allineamento di tutte le coppie di **prefissi** di s_1, s_2
2. Allineamento massimo fra coppie di prefissi = valore massimo in M

Osservazione 2

1. $M[0, 0] = 0$
2. quindi non si prendono sottostringhe con allineamento negativo

Equazione di ricorrenza

Definizione

$M[i, j]$ = ottimo fra tutte le stringhe $s_1[k : i]$, $s_2[h : j]$

$$M[i,j] = \max \begin{cases} M[i-1, j-1] + d(s_1[i], s_2[j]) \\ M[i, j-1] + d(-, s_2[j]) \\ M[i-1, j] + d(s_1[i], -) \\ 0 \end{cases}$$

Condizione al contorno

- $M[0, 0] = M[i, 0] = M[0, j] = 0$
- punto finale = valore massimo
- si risale nell'allineamento fino a uno 0.
- Tempo (nm)

Gap

Definizione

1. Sequenza contigua di indel in un **allineamento**

Esempio

`ABR-AC-ADABRA`: 2 gap
`---B-ANA---NA`: 3 gap

Osservazione

1. Un gap sposta il frame di lettura
2. 1 indel $\approx$ 2 indel

Allineamento con gap generici

- costo gap lungo l: $P(l)$
- Come descrivo l'allineamento ottimo?
- Come è fatta l'ultima colonna?
- Come è fatto l'ultimo gap?

Gap generico

Definizione

$M[i, j]$ = ottimo su $s_1[: i]$, $s_2[: j]$

$$M[i, j] = \max \begin{cases} M[i-1, j-1] + d(s_1[i], s_2[j]) \text{ no gap} \\ \max_{l>0} M[i, j-l] + P(l) \text{ gap in } s_1 \\ \max_{l>0} M[i-l, j] + P(l) \text{ gap in } s_2 \end{cases}$$

Condizione al contorno

- $M[0, 0] = 0$
- $M[i, 0] = P(i)$, $M[0, j] = P(j)$
- Tempo $O(nm(n + m))$

Allineamento con gap affine

- costo gap lungo l: $P_o + lP_e$
- P_o: costo apertura gap
- P_e: costo estensione gap
- $P_e, P_o > 0$
- Come descrivo l'allineamento ottimo?
- Come è fatta l'ultima colonna?
- Come è fatto l'ultimo gap?

Gap affine

Definizione

- $M[i, j]$ = ottimo su $s_1[: i]$, $s_2[: j]$
- $E_1[i, j]$ = ottimo su $s_1[: i]$, $s_2[: j]$, con estensione di gap finale in s_1
- $E_2[i, j]$ = ottimo su $s_1[: i]$, $s_2[: j]$, con estensione di gap finale in s_2
- $N_1[i, j]$ = ottimo su $s_1[: i]$, $s_2[: j]$, con apertura di gap alla fine di s_1
- $N_2[i, j]$ = ottimo su $s_1[: i]$, $s_2[: j]$, con apertura di gap alla fine di s_1

Gap affine

$$M[i, j] = \max \begin{cases} M[i-1, j-1] + d(s_1[i], s_2[j]) \\ E_1[i, j], E_2[i, j] \\ N_1[i, j], N_2[i, j] \end{cases}$$

$$E_1[i, j] = \max \begin{cases} E_1[i, j-1] + P_e \\ N_1[i, j-1] + P_e \end{cases}$$

$$E_2[i, j] = \max \begin{cases} E_2[i-1, j] + P_e \\ N_2[i-1, j] + P_e \end{cases}$$

$$N_1[i, j] = M[i, j-1] + P_o + P_e, \quad N_2[i, j] = M[i-1, j] + P_o + P_e$$

Allineamento multiplo

k sequenze

- Input: insieme di sequenze $\{s_1, \ldots, s_k\}$
- Aggiunta di **indel** nelle sequenze
- sequenze estese = tutte stessa lunghezza
- NO colonne di indel

Valore di una colonna

SP: sum of pairs

- $\{s_1, \ldots, s_k\} \mapsto \{s_1^*, \ldots, s_k^*\}$ allineate
- Valore $\{s_1^*[h], \ldots, s_k^*[h]\}$
- $\sum_{i<j} d(s_1^*[i], s_k^*[j])$

Complessità

- se k è arbitrario ⇒ NP-completo
- se k è fissato ⇒ tempo $O(n^k)$

Matrici di sostituzione

1. Utilizzate per valutare un allineamento
2. Implicitamente probabilità di transizione
3. Mutazioni ricorrenti
4. Allineamenti di proteine

PAM: unità di misura

1. PAM: point/percent accepted mutation
2. due sequenze s_1 e s_2: quanto sono distanti?
3. distanza 1PAM $\Rightarrow$ numero mutazioni = $\frac{1}{100}|s_1|$
4. semplice in assenza di indel
5. Mutazioni ricorrenti $\Rightarrow$ misura affidabile solo per piccoli valori
6. s_1 e s_2 distanti 100 PAM $\Rightarrow$ una singola base ha 36% di probabilità di non essere mutata

Matrici PAM

1. dipende dalla distanza attesa
2. PAM250, PAM200, PAM1

Calcolo PAMk

1. Costruzione PAMk
2. Si prendono varie sequenze distanti kPAM
3. si allineano le sequenze
4. si calcolano le frequenze $f(i)$, $f(i, j)$ di tutti i singoli caratteri e le coppie di caratteri
5. PAM$k(i, j) = \log \frac{f(i,j)}{f(i)f(j)}$

Log odds ratio

Odds ratio

1. $\frac{p}{1-p}$, p è la probabilità dell'evento interessante (target)
2. $\frac{f(i,j)}{f(i)f(j)}$
3. $f(i, j)$: frequenza della mutazione misurata
4. $f(i)f(j)$: ipotesi nulla (caratteri indipendenti)

Matrici PAM

Calcolo PAMk nella realtà

- Problema: come allineare se non si conosce la matrice
- Allineate sequenze molto simili
- no indel
- $M_k(i, j) = \log \frac{f(i)M_1^k(i,j)}{f(i)f(j)} = \log \frac{M_1^k(i,j)}{f(j)}$
- valori moltiplicati per 10
- arrotondati all'intero più vicino
- si somma un intero a tutti i valori

Matrici BLOSUM

Confronto con PAM

- PAM allinea sequenze vicine
- ma viene usata per allineare sequenze lontane
- regioni conservate e non conservate hanno stessa importanza

BLOCKS

- blocchi di regioni conservate
- scelte "a mano"
- $B(i, j) = \log \frac{f(i,j)}{f(i)f(j)}$

Matrici BLOSUM

BLOSUMx

- le sequenze che sono simili più di x% vengono clusterizzate
- cluster = rimuovere tutte tranne una
- scopo: evitare di sovrapesare parti sovrarappresentate nel campione
- BLOSUM62: più usata per gli allineamenti

Statistiche Karlin-Altschul

Ricerca in un database

- Punteggio positivo possibile
- Punteggio medio negativo
- Simboli indipendenti e equiprobabili
- Sequenze infinitamente lunghe
- Allineamenti senza gap

Equazione Karlin-Altschul

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

- E: numero allineamenti
- k: costante
- n: numero caratteri in database
- m: lunghezza stringa query
- λS: punteggio normalizzato

BLAST

Basic Local Alignment Search Tool

- Ricerca seed
- seed = pattern matching con sottostringa di lunghezza 3
- Costruzione high-scoring segment pair (HSP) = estensione seed
- Filtro seed tenuti solo HSP con alta significatività
- Fusione HSP vicine
- Smith-Waterman sulle regioni

Filogenesi perfetta.

Filogenesi. Neighbor-Joining.

Sequenziamento e grafi di de Brujin

Licenza d'uso