BIOINFORMATICA

ONE ALGO – ONE PAGE

Autore:
GianQuzzo

Gennaio 2020

Indice

Bit-Parallel: Dömölki,Baeza-Yates/Gonnet

Comunemente chiamato SHIFT-AND, appartiene agli algoritmi semi-numerici, ovvero quelli algoritmi che manipolano i numeri ragionando sulla loro rappresentazione binaria, utilizzando operazioni Bit-Wise per il confronto (operazioni eseguite a livello HW). In questo modo è possibile eseguire pattern matching in modo efficiente abbattendo la costante moltiplicativa.

Algoritmo:

Sia T la stringa (di lunghezza n) e P il pattern (di lunghezza m). Si ipotizza di avere una matrice M nxm tale che:

$$M[i,j] = 1 \ \ se \ P[:i] = T[j-i+1:j]$$

Ovvero $M[i,j]$ = 1 sse il prefisso i-esimo del pattern è uguale alla sottostringa di lunghezza i in posizione $j-i+1$. L'ultima riga della matrice mostra se il pattern è presente o meno nel testo: 1 in corrispondenza di ogni occorrenza, precisamente nella posizione del carattere finale.

L'algoritmo utilizza dei vettori $U[\sigma]$ lunghi m per ogni $\sigma \in \Sigma$, avranno 1 in posizione i se $P[i] = \sigma$, 0 altrimenti.

Per costruire la matrice gestisco quest'ultima come un array di colonne, in particolare la prima colonna della matrice viene sempre inizializzata a 0 ad eccezione del primo valore che sarà 1 solo se $T[1] = P[1]$, le altre colonne dipendono dalla precedente in questo modo:

- La colonna i ($con\ i\ \geq 2$) corrisponde alla colonna $i-1$ shiftata di una posizione verso il basso
- Il primo valore della colonna i è settato a 1 di default
- La colonna così ottenuta si mette in AND con U[T[i]]

Equazione di ricorrenza:

$$C[j] = \Big((C[j-1] \gg 1) \mathrm{V} \big(1 \ll (\omega - 1)\big)\Big) \wedge \mathrm{U}\big[\mathrm{T}[\mathrm{j}]\big]$$

$$\omega \rightarrow word\ size$$

Tempi:

Tempo $O(n)$ se $m \leq w$

Tempo $O(nm)$ altrimenti

KARP-RABIN

L'algoritmo appartiene alla categoria degli algoritmi probabilistici, i quali non danno la garanzia che il risultato sia corretto e con lo stesso input potrebbero esserci output diversi.
L'idea centrale dell'algoritmo è codificare una stringa con un numero. In particolare, si dispone di una funzione di Hash, o fingerprint, che prende in input una stringa di 0 e 1 e ritorna un numero associato. La funzione è definita come:

$$H(S) = \sum_{i=1}^{|S|} 2^{(i-1)} H(S[i])$$

Sia T il testo (di lunghezza n) e P il pattern (di lunghezza m), calcolo la fingerprint del pattern, calcolo la fingerprint della sottostringa dei primi m caratteri del testo, eseguo il confronto, mi sposto in avanti di una posizione e ripeto.
In particolare, disponiamo di una sliding window di m caratteri su T. A ogni spostamento la fingerprint lunga m sul testo viene ricalcolata. Per rendere più veloce il calcolo della fingerprint a partire da quella precedente si utilizza la seguente formula:

$$H(T[i+1:i+m]) = \frac{(H(T[i:i+m-1]) - T[i])}{2 + 2^{m-1}T[i+m]}$$

Così com'è definita la fingerprint è esponenziale nella dimensione del pattern, in caso di numeri grandi si ha un tempo $O(n)$, si torna dunque ad un problema di complessità $O(nm)$.

Bisogna ridurre la dimensione della fingerprint perlomeno a qualcosa che cresca in modo polinomiale in n o m. Il costo di tale riduzione è la perdita di una corrispondenza 1:1 tra stringhe e fingerprint (si avranno più stringhe con la stessa fingerprint), ciò porta inevitabilmente a possibili falsi positivi.

Scelgo dunque un numero p primo (più grande è meno è probabile di avere falsi positivi), e riscrivo la formula della fingerprint come

$$H(T[i+1:i+m]) = \frac{(H(T[i:i+m-1]) - T[i])}{2 + 2^{m-1}T[i+m]} \; mod \; p$$

Per diminuire la possibilità di incontrare falsi positivi, è possibile scrivere k primi casuali indipendenti e cambiare numero primo ad ogni calcolo di fingerprint

Tempi:
Caso medio $O(n+m)$
Caso pessimo $O(nm)$

Da Suffix tree a Suffix array

Un Suffix array per un Suffix tree consiste in un array di interi relativi ai suffissi ottenuti dal Suffix tree ordinati lessicograficamente.

L'algoritmo che costruisce il Suffix array a partire dal Suffix tree è un algoritmo piuttosto semplice che permette di fare ciò in tempo lineare. Basta infatti eseguire una vista dell'albero in profondità, con l'accortezza che per ogni nodo i figli vengano visitati in ordine lessicografico, poiché per ogni nodo vi è un solo arco uscente per ogni lettera inziale vi è un ordine non ambiguo.

Per costruire il relativo LCP (longest common prefix) si agisce in questo modo:
Per ogni coppia di suffissi consecutiva nel Suffix array (SA[i], SA[i+1]) si cerca nell'albero il last common ancestor, ovvero il primo antenato dei 2 suffissi (ciò è unico proprio come il percorso che unisce i 2 suffissi), e ne si calcola la string depth, ovvero la lunghezza del suo path label, ovvero la stringa ottenuta concatenando tutte le label del percorso dalla radice a tale nodo.

Tempi:
Tempo di $O(n)$

Da Suffix array a Suffix tree

La conversione da Suffix array a Suffix tree è più complessa e viene eseguita ricorsivamente. Tale metodo utilizza l'LCP (longest common prefix) per trarre informazioni sulla tipologia dell'albero.

Osserviamo che se LCP[i]=0 vuol dire che nessun nodo a parte la radice è condiviso per andare dall'elemento i a i+1. Di conseguenza troveremo nel LCP tanti zeri quante le coppie che non condividono nessun carattere, quindi i sottoalberi dovranno essere figli diversi dalla radice.

Possiamo quindi identificare la presenza di uno zero come la suddivisione dell'albero in porzioni;

ES. LCP --> 0 1 3 0 0 2

Per ogni sotto porzione si agisce ricorsivamente individuando nuove sotto porzioni identificate da ogni occorrenza del valore più basso. Le sotto porzioni vuote indicano le foglie.

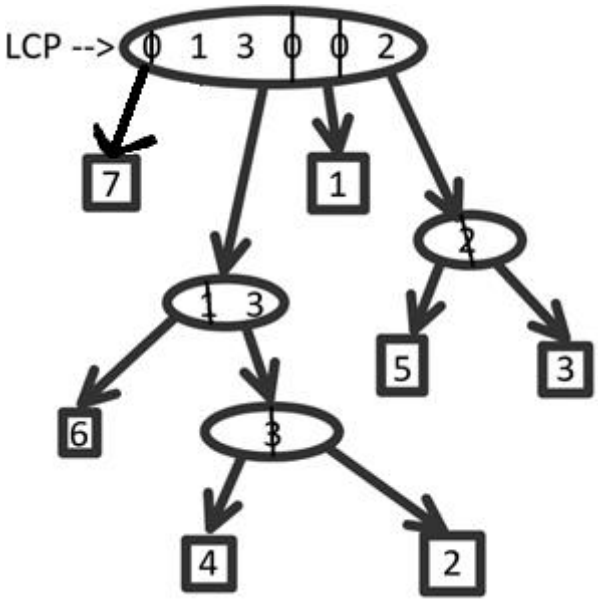
Più in generale l'algoritmo generico agisce in questo modo:

1. Analizzo una regione del LCP (a inizio algoritmo sarà LCP stesso) e cerco il numero più basso
2. Ho k occorrenze, suddivido in k+1 regioni
3. Reitero

CASO BASE: ho k+1 regioni vuote, mi fermo.

In conclusione, visito la struttura dell'albero appena costruita in Deph first e assegno agli archi gli elementi del Suffix array in ordine ogni volta che trovo una foglia.

Esempio: $SA \rightarrow 7642153$



Pattern matching con Suffix array e acceleranti

Avendo a disposizione una Suffix array sarebbe possibile sfruttare l'ordine lessicografico per eseguire una ricerca dicotomica per individuare un certo pattern P in un testo T, ovvero trovare quei suffissi che iniziano per P. Tale ricerca utilizza le posizioni L e R come elementi estremi tali che $SA[L] < P < SA[R]$. Ad ogni iterazione si confronta P con l'elemento mediano $SA[M]$: se questo è maggiore, allora la ricerca continuerà tra L ed M e viceversa. Tale algoritmo ha tempo approssimabile di $O(m \log n)$.

È possibile fare uso di 3 acceleranti per migliorare tale tempo:

Accelerante 1: Sfrutta la conoscenza del fatto che tutti i suffissi nell'intervallo $SA[L, R]$ iniziano con lo stesso prefisso lungo $LCP(SA[L], SA[R])$. Tale informazione può essere sfruttata per accelerare il confronto lessicografico tra M e P: poiché se tutti i suffissi tra L e R condividano i primi x caratteri, possiamo iniziare il confronto a partire da $x + 1$. Tuttavia, non miglioriamo il tempo dal punto di vista teorico, che rimane invariato.

Accelerante 2: Siano $l = LCP(L, P)$ ed $r = LCP(R, P)$ i rispettivi LCP tra il pattern e il primo e l'ultimo suffisso. Ad ogni iterazione le due quantità vengono confrontate e ricalcolate di conseguenza. Nello specifico si possono verificare i seguenti casi:

1. $l > r$, allora
    a) $LCP(L, M) > l \rightarrow$ Il pattern sta tra M e R
    b) $LCP(L. M) < l \rightarrow$ Il pattern sta tra L e M
    c) $LCP(L, M) = l \rightarrow$ è necessario confrontare il carattere successivo
2. $l = r$, allora
    a) $LCP(L, M) > l \rightarrow$ Il pattern sta tra M e R
    b) $LCP(L, M) < l \rightarrow$ il pattern sta tra L e M
    c) $LCP(L, M) = l = r \rightarrow$ è necessario confrontare il carattere successivo
3. $l < r$ è simmetrico al caso 1, allora
    a) $LCP(L, M) > l \rightarrow$ Il pattern sta tra L e M
    b) $LCP(L. M) < l \rightarrow$ Il pattern sta tra M e R
    c) $LCP(L, M) = l \rightarrow$ è necessario confrontare il carattere successivo

Accelerante 3: consiste in un pre-processamento inziale dove vengono calcolate tutte le coppie di suffissi che servono, ovvero gli estremi di ricerca dicotomica e relativi LCP. Il numero di coppie da calcolare risulta lineare (si dimezza ad ogni iterazione), il calcolo LCP può essere fatto anch'esso in tempo lineare (per intervalli di ampiezza dove già si possiedono i valori), il calcolo sugli n intervalli avviene in maniera ricorsiva e attraverso aggreganti. In particolare, siano L1 e R1 gli estremi della prima metà dell'intervallo e L2 e R2 quelli del secondo intervallo, è possibile ricavare $LCP(L1, R2)$ come $\min\{LCP(L1, R1), LCP(L2, R2), LCP(R1, L2)\}$. Notare che R1 e L2 poiché adiacenti, si è già in possesso di $LCP(R1, L2)$.

Grazie a quest'ultimo accelerante ho una complessità di tempo nel caso pessimo di $O(m + \log n)$

Tempi:

Tempo di $O(m \log n)$ senza acceleranti

Tempo di $O(m + \log n)$ nel caso pessimo con acceleranti

Pattern matching con Suffix tree

Dato un testo T (lungo m) e un pattern P (lungo n) si vogliono trovare tutte le occorrenze di P in T. Il Suffix tree permette di fare ciò in tempo lineare.

Tale procedura di ricerca si basa sul fatto che ogni sottostringa è un prefisso di qualche suffisso di T. Dobbiamo dunque analizzare i suffissi di T che iniziano per P, in un Suffix tree essi sono identificati dai percorsi Radice-Foglia.

Confronto dunque il pattern con gli archi uscenti dalla radice, e continuo sui sottoalberi fino a consumare l'intero pattern. In particolare:
Parto dal primo carattere del pattern e dalla radice dell'albero:

1. Per il carattere corrente del pattern, se c'è una corrispondenza con la stringa dell'arco si prosegue verso il basso;
2. Se il pattern non esiste nell'arco corrente, allora non esiste in assoluto;
3. Quando ho consumato tutto il pattern il numero di foglie che ho come discendenti è esattamente il numero di occorrenze del pattern nel testo (k occorrenze) e il loro valore indica l'indice di quest'ultimo.

Tempi:
Tale procedura ha una complessità di $O(m + k)$

Sottostringa comune a 2 stringhe

Date due stringhe s_1 e s_2 si vuole trovare la sottostringa comune più lunga. Per fare ciò è possibile utilizzare un Suffix tree generalizzato (insieme di stringhe), dove ogni stringa appartiene alla prima o alla seconda stringa o ad entrambi. Il testo ha la forma $ST(s_1\$_1 s_2\$_2)$, cioè i due testi vengono concatenati, viene usato un solo terminatore e viene costruito l'albero.

A ogni foglia è associata una coppia di valori: la stringa di appartenenza (riferimento) e la sua posizione di partenza in essa. Le stringhe che appartengono ad entrambe $s_1$ed s_2 avranno due coppie di valori.

A questo punto viene cercato un nodo x con discendenti di s_1e s_2, tale che la sua string depth sia massima.

Tempi:
La costruzione dell'albero e della ricerca hanno entrambi tempo lineare.

Sottostringa comune a k stringhe con Suffix tree

Come per la LCS tra due stringhe, l'obbiettivo risulta essere costruire un Suffix tree generalizzato a partire dalle stringhe in input ed analizzare i nodi interni.

In particolare, siano <$s_1, \dots, s_k$> un insieme di stringhe, vogliamo determinare una LCS. Abbiamo che, dato il Suffix tree generalizzato, vogliamo trovare un'nodo x tale che abbia come discendenti una foglia per ogni stringa dell'insieme in input, e la cui string depth sia massima. Ogni foglia identifica un suffisso di una delle stringhe in input ed è distinta dalla propria stringa di appartenenza (identificatore).

L'algoritmo tiene traccia per ogni nodo un vettore di booleani $C_x$definito come segue:

$C_x[i] = 1$ Se il nodo x ha una foglia discendente appartenente ad s_i

L'array di ogni nodo è definito dunque come l'or del vettore C dei figli. A questo punto calcolati gli array C per ogni nodo verrà considerato quello con string depth più lunga, il quale percorso nodo radice identifica proprio la LCS delle k stringhe.

Ricapitolando:

1. Viene costruito il Suffix tree generalizzato
2. Vengono calcolati gli array C per ogni nodo (post-order)
3. Viene rivisitato l'albero per capire quali prendere in considerazione (quelli con tutti 1 sull'array C)
4. Viene fatta un'ulteriore visita per trovare la string depth massima

Tempi:

Il costo computazionale risulta essere $O(nk)$, dove n è la somma di tutte le lunghezze delle stringhe in input.

Sottostringa comune a k stringhe con Suffix array

Siano <$s_1, \dots, s_k$> un insieme di stringhe, vogliamo determinare LCS utilizzando il Suffix array generalizzato.

L'algoritmo che utilizza il Suffix tree analizza i nodi di quest'ultimo al fine di trovare un nodo con discendenti foglie relative alle stringhe in input (almeno una per ogni stringa), di cui la string depth sia massima. Un nodo interno del Suffix tree corrisponde ad un intervallo del Suffix array. Se affianchiamo il Suffix array generalizzato al rispettivo LCP è possibile analizzare gli intervalli di quest'ultimi per risolvere il problema dell'LCS.

L'idea generale è quella di trovare un intervallo che abbia almeno un suffisso per ogni stringa in input e di questo intervallo trovare il minimo valore di LCP, tale che quest'ultimo sia il massimo tra tutti quelli analizzati.

Dunque, l'algoritmo analizza intervalli (i, j) tale che quest'ultimo contenga un suffisso per ogni stringa in input: per fare ciò utilizza un vettore Last definito come

$Last[i]$= posizione nel Suffix array dell'ultimo suffisso visto della stringa i

Per ogni valore di j si analizza dunque l'intervallo $(min(Last), j)$ ovvero si estrae il valore minimo dal vettore Last. Ad ogni iterazione di j viene incrementato e il vettore last ricalcolato di conseguenza.

Tempi:

Con l'utilizzo di RMQ per individuare il minimo LCP ad ogni intervallo l'algoritmo trova la LCS alle k stringhe in un tempo di $O(n \log n)$

Range Minimum Query

Tale algoritmo viene utilizzato nel calcolo dell'LCS per k stringhe per calcolare il minimo valore dell'LCP per intervalli (i, j).

L'input è dunque un array A di n interi, su cui si vuole essere in grado di calcolare per il generico intervallo (i, j) il minimo valore in A ovvero $\min_{i \le h \le j} \{A[h]\}$.

Si vuole rispondere a tale query con tempo costante $O(1)$ per fare ciò RMQ prevede di preprocessore l'array A.
L'idea generale dell'algoritmo per trovare il minimo in un intervallo (i, j) è precalcolare un intervallo da i a un certo punto (h) e un secondo intervallo da un altro punto (l) a j, tale che i due intervalli coprano l'intera ampiezza di (i, j).

$$[i, h] \qquad [l, j]$$

A questo punto il minimo nell'intervallo (i, j) sarà il minimo tra i valori precalcolati per questi due intervalli.
Nello specifico RMQ per un intervallo (i, j) utilizza i due intervalli di ampiezza 2^z con

$$z = \lfloor \log_2(j - i + 1) \rfloor$$

Da cui $2^z \le j - i + 1$

Ovvero gli intervalli $[i : i + 2^z - 1]$ $[j - 2^z + 1 : j]$, i quali rispettano le specifiche sopracitate.

Il preprocessamento consiste dunque nel costruire un array bidimensionale B, tale che l'elemento $B[x, y]$ contiene il minimo valore di A nell'intervallo $[x : x + 2^y - 1]$. Tale array viene costruito in tempo $O(n \log n)$.

Ricorrenza:

Base: $B[x, 0] = A[x]$

Ricorsivo: $B[x, y] = \min\{B[x, y - 1], B[x + 2^y - 1, y - 1]\}$

Una volta completato il preprocessamento ricavo il minimo di un intervallo (i, j) in A come:

$$\min_{i \le h \le j} A[h] = min\{B[i, w], B[j - 2^w + 1, w]\} \; dove \; w = \lfloor \log_2(j - i + 1) \rfloor$$

Allineamento Globale di Needleman-Wunsch

Date due stringhe di lunghezza arbitraria, per allineamento si intende una modalità per confrontare quest'ultime e capire quanto queste siano simili, e riuscire a calcolare il valore di tale confronto. Per fare ciò è possibile inserire all'interno delle 2 stringhe caratteri di indel (-) in modo tale da cercare di incolonnare caratteri uguali delle due stringhe. Per ogni possibile combinazione carattere-carattere e carattere-indel viene attribuito un valore, espresso tramite una matrice di score. Il valore di un allineamento sarà dunque la somma dei valori di tutte le colonne di quest'ultimo.

A questo punto è possibile esprimere il problema dell'allineamento come un problema di massimizzazione:

- Instanze: numero infinito di casi, in questo caso sono tutti i possibili allineamenti
- Soluzioni ammissibili: un allineamento ammissibile è determinato da una coppia di stringhe t_1 e t_2 tale che
  - $|t_1| = |t_2|$
  - Togliendo gli indel da t_1 ottengo S_1 (la prima stringa in input) idem per $t_2 \rightarrow S_2$
  - $\nexists j \in \{1, \dots, |t_1|\}$ tale che $t_1[j] = t_2[j] =$ '-', non posso avere colonna di solo indel
- Funzione obbiettivo: Instanza $\rightarrow \mathbb{R}$, associa ad ogni coppia di stringhe un valore calcolabile
  Come $Val(t_1, t_2) = \sum_{i=1}^{|t_1|} d(t_1[i], t_2[i])$ dove d è la matrice di score

È possibile risolvere tale problema con l'ausilio della programmazione dinamica: costruiamo una matrice M tale che il valore della casella $M[i,j]$ rappresenti il valore dell'allineamento ottimo tra $s_1[:i]$ e $s_2[:j]$. Come per qualsiasi problema di programmazione dinamica si suppone di aver calcolato i sotto problemi più piccoli e si esprime il calcolo di generico sotto problema con un'equazione di ricorrenza nel nostro caso sarà

$$M[i,j] = max \begin{cases} B[i-1, j-1] + d(S_1[i], S_2[j]) \\ B[i, j-1] + d(-, S_2[j]) \\ B[i-1, j] + d(S_1[i], -) \end{cases}$$

Con opportune condizioni al contorno

$$M[0,0] = 0$$
$$M[i,0] = M[i-1,0] + d(S_1[i], -)$$
$$M[0,j] = M[0,j-1] + d(-, S_2[j])$$

Una volta calcolato il valore dell'allineamento globale ottimo è possibile ottenere tale allineamento escludendo il backtracking a partire dall'ultima cella alla prima.
Per costruire tale matrice bastano 2 cicli for innestati ciò si traduce in un costo computazionale di $O(nm)$.

Allineamento Locale di Smith Waterman

A differenza dell'allineamento globale, il quale mette a confronto 2 stringhe nella loro totalità l'allineamento locale ha come scopo quello di individuare 2 sottostringhe delle stringhe in input tali che il loro allineamento sia ottimo. In particolare, date in input s_1 e s_2, si vogliono trovare 2 sottostringhe t_1e t_2 tali che $All(t_1, t_2) \geq All(u_1, u_2)$ per ogni coppia di sottostringhe u_1e u_2.

L'algoritmo di Smith Waterman permette di fare ciò sostituendo i valori negativi con valori nulli, a differenza dell'equazione di ricorrenza proposta da Needleman-Wunsch verrà infatti aggiunto un quarto caso, che si verifica quando nessuno degli allineamenti ha valore positivo, e a sfavore di quest'ultimi viene scelto il valore zero. Ciò permette di separare zone ad alta similarità (valori strettamente positivi) da zone di bassa similarità (valori strettamente negativi). Questo significa che il valore di una casella della matrice non dipenderà più dal punteggio accumulato a partire dalla prima casella ma bensì dall'ultima casella con valore nullo.

Formalmente

Input: Due stringhe s_1e s_2 e una matrice di score d. L'algoritmo individua 2 sottostringhe (t_1 e t_2) tale che il loro allineamento sia ottimo.

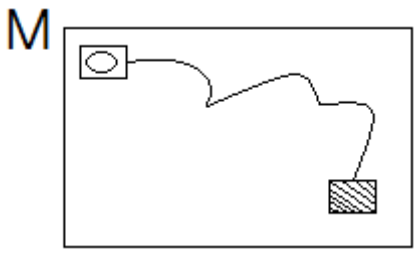
$$t_1 = s_1[n:i] \qquad t_2 = s_2[k:j]$$

Ricorrenza:

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

Condizioni al contorno: $M[i,0] = M[0,j] = M[0,0] = 0$

Una volta costruita la matrice si ottiene il valore dell'allineamento locale ottimo cercando in questa il valore massimo, per trovare l'effettivo allineamento e di conseguenza le sottostringhe ad esso associato, basterà eseguire il backtracking fino alla prima cella con valore nullo



Allineamento con gap generico

Definizione: Si definisce gap una sequenza consecutiva di indel in un allineamento

La variante dell'allineamento con gap generico introduce la definizione soprariportata al fine di diversificare, nell'ambito dell'allineamento, l'inserimento di un singolo indel dall'estendere una sequenza di indel. In particolare, la penalità per inserirne uno non sarà più costante ma sarà calcolata in funzione della sua lunghezza.

Viene introdotta dunque la funzione $P(l)$ che indica il costo di un generico GAP lungo l. Vogliamo dunque calcolare il minor allineamento tenendo conto della presenza di gap in quest'ultimo.

L'algoritmo procede con la costruzione di una matrice di programmazione dinamica M, la cui equazione di ricorrenza è così definita

$$M[i,j] = max \begin{cases} M[i-1, j-1] + d(s_1[i], s_2[j]) \\ \max_{l>0} M[i, j-l] + P(l) \;\; gap\ in\ s_1 \\ \max_{l>0} M[i-l, j] + P(l) gap\ in\ s_2 \end{cases}$$

In particolare, gli ultimi 2 casi si riferiscono alla presenza di un indel sull'ultima colonna (di s_1 o s_2): vengono dunque calcolati tutti i possibili valori di allineamento per ogni lunghezza di gap.

Condizioni al contorno:

$$M[0,0] = 0$$
$$M[i,0] = P(i)$$
$$M[0,j] = P(j)$$

Dunque, il calcolo di ogni casella non avviene più in tempo costante, in particolare per calcolare tutti i valori della matrice M, poiché per ogni casella avrà un numero variabile di casi, impiego un tempo pari a $T(n,m) = O(nm(n+m))$ ricavabile dalla risoluzione della seguente sommatoria

$$\sum_{i=1}^{n} \sum_{j=1}^{m} (i+j)$$

Allineamento multiplo

È possibile generalizzare l'algoritmo di allineamento su 2 stringhe a k stringhe in input; come input l'algoritmo riceve un insieme di k stringhe <$s_1, \dots, s_k$> e una matrice di score d, e calcola il valore dell'allineamento ottimo delle k stringhe.

È possibile esprimere tale problema come un problema di massimizzazione:

Instanza: Insieme dei possibili allineamenti

Soluzioni ammissibili: Un allineamento ammissibile è identificabile da un insieme di k Stringhe <$t_1, \dots, t_k$> tale che

- $|t_1| = |t_2| = \dots = |t_k|$
- Togliendo gli indel da t_i si ottiene la stringa s_i, questo $\forall i = 1 \dots k$
- $\nexists j \in \{1, \dots, |t_1|\}$ tale che $t_1[j] = \dots = t_k[j] = '-'$

Funzione obbiettivo: Associa ad ogni coppia di stringhe (t_i, t_h) il valore del loro allineamento sommando dunque il valore di tutte le coppie, ovvero

$$\sum_{i=1}^{k} \sum_{h=i+1}^{k} val(t_i, t_h) = \sum_{j=1}^{|t_1|} \sum_{i=1}^{k} \sum_{h=i+1}^{k} d(t_i[j], t_h[j])$$

Il problema è risolvibile con l'ausilio della programmazione dinamica in questo caso suddividendo il calcolo dell'allineamento considerando la possibile struttura dell'ultima colonna di quest'ultimo e quella della sottomatrice restante, sapendo che per ogni stringa ho 2 possibili casi (carattere o indel) ovvero 2^{k-1} possibili casi a rappresentazione dell'ultima colonna.

Nella parte che precede l'ultima colonna dell'allineamento avrò invece l'allineamento ottimo delle sotto-istanze così definito: per ogni sottostringa:

- Se ho un indel nell'ultima colonna per tale stringa a sinistra avrò l'allineamento ottimo tra quest'ultima e le altre sottostringhe meno il loro ultimo carattere
- Se ho un carattere nell'ultima colonna per tale sottostringa, a sinistra avrò l'allineamento ottimo di quest'ultima meno il suo ultimo carattere con le altre sottostringhe, ognuna di queste rianalizzate secondo i 2 casi appena citati.

La matrice costruita sarà una matrice multidimensionale, in particolare a k dimensioni

$$M[i_1, i_2, \dots, i_k]$$

Tempi:

Per calcolare tutte le sue caselle impiego un tempo di $O(2^k n^k)$

Grafi di de Bruijn

I grafi di de Bruijn sono una particolare rappresentazione che permette di risolvere il problema dell'assemblaggio riconducendolo ad un problema sui grafi.

In particolare, si ha che: ho un read lungo n caratteri, estraggo tutte le sottostringhe di lunghezza fissata (k- meri) per costruire il grafo di riferimento l'algoritmo agisce in questo modo:

- Ogni k-mero viene diviso in $(k-1)-\text{meri}$;
- Viene creato un nodo per ogni $(k-1)-\text{mero}$;
- Viene creato un arco per ogni $k-\text{mero}$;

A questo punto si vuole trovare un assemblaggio valido, ovvero un cammino che attraversi con ogni arco una e una sola volta, ovvero vogliamo trovare un cammino euleriano, introduciamo i seguenti teoremi:

- "Un grafo <V, E> orientato ha un cammino euleriano sse è semi-euleriano"
- "Un grafo <V, E> orientato è semi-euleriano se esistono due vertici s e t tali che il numero di archi entranti in s è uguale al numero dei suoi uscenti meno 1, il numero entranti in t è uguale al numero di uscenti più 1, mentre ogni altro vertice è bilanciato (entrambi = uscenti)"
- "Sia G un grafo semi-euleriano, e sia P un cammino da s a r allora $G_1 = <V, E-\{P\}>$ è euleriano"
- "Sia G un grafo semi-euleriano, e sia C un ciclo su G, allora $G_1 = <V, E-\{C\}>$ è euleriano"

A questo punto dato un grafo semi-euleriano è possibile trovare un cammino euleriano, ovvero un assemblaggio per il nostro grafo, nel seguente modo:

1. Traccio un percorso da s a t
2. Considero solo gli archi non appartenenti a tale cammino cosicché il grafo considerato è euleriano
3. Parto dal primo arco non ancora visitato e lo seguo: poiché il grafo ora è bilanciato, tornerò nel nodo di partenza evidenziando quindi un ciclo, non considero più tale ciclo
4. Continuo così fino a quando non avrò visitato tutti gli archi del grafo

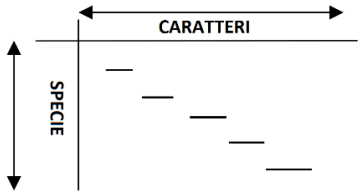
A questo punto unisco i cicli trovati al primo percorso "principale" da s a r, il risultato è il cammino euleriano che stavamo cercando. L'azione di accorpamento dei cicli di quest'ultimo, in questo modo possiamo risolvere il problema dell'assemblaggio in tempo lineare.

Filogenesi perfetta

Per mutazioni genetiche si intendono delle variazioni nel genoma di determinate specie di individui, studiate e visibili in intervalli di tempo relativamente lunghi, ovvero osservabili nell'arco di generazioni. L'obbiettivo è dunque riuscire a tenere traccia delle mutazioni nella storia evolutiva di un organismo.

Un semplice approccio prende il nome di filogenesi basata su caratteri: un carattere può essere una caratteristica osservabile o una caratteristica genetica. Si assume che l'evoluzione avvenga per accumulo di suddetti caratteri, i quali vengono acquisiti esattamente una volta e non vengono mai persi. Tale formulazione porta ad un problema computazionale noto come filogenesi perfetta.

Input: Matrice binaria M, utilizzata per indicare quali soggetti hanno un determinato carattere



Output: Un albero che spiega M, se questo esiste.

È possibile notare che non tutte le matrici hanno un corrispettivo albero di filogenesi perfetta, in particolare vale il seguente lemma:

Sia T una filogenesi perfetta: dati $c_1 e\ c_2$ 2 caratteri siano $S(c_1)$ e $S(c_2)$ gli insiemi di specie di T che possiedono i caratteri c_1 o c_2, allora deve verificarsi una delle seguenti condizioni

$S(c_1) \subseteq S(c_2)$
$S(c_1) \supseteq S(c_2)$
$S(c_1) \cap S(c_2) = \emptyset$

Per costruire l'albero evolutivo a partire dalla matrice di filogenesi è possibile utilizzare l'algoritmo di Gusfield: Quest'ultimo prevede in primis un ordinamento delle colonne con l'utilizzo di radix sort, fatto ciò è possibile leggere riga per riga la matrice così ottenuta per costruire l'albero: per ogni 1 che incontro continuo a scorrere l'albero se l'arco per tale carattere esiste, altrimenti lo creo, quando incontro lo zero inserisco la foglia corrispondete alla specie in analisi.

In conclusione, l'albero così costruito avrà le seguenti caratteristiche:

- Ogni specie corrisponde ad una foglia
- Ognuno degli m archi etichetterà esattamente un arco

Tempi:

La costruzione dell'albero risulta essere lineare nelle dimensioni della matrice: il tempo di esecuzione infatti è detto radix sort, il quale impiega il tempo pari $O(nm)$.

Sankoff – piccola parsimonia

Nell'ambito della filogenesi e alberi evolutivi solitamente non è possibile lavorare con modelli che impongono restrizioni molto forti, come per esempio il modello di filogenesi perfetta. Vengono quindi introdotti dei modelli (modelli evolutivi, es: Dollo, Comin Sokal) i quali impongono meno vincoli sulla definizione dell'albero, in questo caso si corre il rischio però di perdere l'unicità di quest'ultimo, ovvero possono esserci più alberi in grado di spiegare una matrice di caratteri/specie per seguire quale soluzione adottare è possibile utilizzare algoritmi basati su parsimonia: in particolare l'algoritmo di Sankoff è in grado di risolvere il problema di piccola parsimonia così definito:

Istanze:

- Matrice M
- Topologia T dell'albero
- Per ogni carattere c, costo w_c per ogni coppia di stati

Soluzioni ammissibili:

- Per ogni carattere $c \in C$, una etichettatura λ_c che assegna a ogni nodo uno degli stati possibili per C

Funzione obbiettivo:

- Per ogni coppia di nodi adiacenti vengono applicate le funzioni λ, i quali risultati vengono dati in input alla funzione di costo del rispettivo carattere. Il costo dell'intero albero sarà dato dalla somma di tutti i risultati. Si va poi a minimizzare tale funzione

$$\min \sum_{c \in C} \sum_{(x,y) \in E(T)} w_c(\lambda_c(x), \lambda_c(y))$$

Tale scrittura permette di analizzare e risolvere ogni singolo carattere in modo indipendente per poi ricombinare le soluzioni. L'algoritmo di Sankoff utilizza la programmazione dinamica partendo dalle foglie per poi risalire fino alla radice:

$M[x, z]$: soluzione ottimale del sottoalbero di T che ha radice x, sotto la condizione x abbia etichetta z.

$M[x, z] = 0$ se x è una foglia con etichetta z

$M[x, z] = +\infty$ se x è una foglia con etichetta diversa da z

$M[x, z] = \sum_{f \in F(x)} \min_s \{w(z, s) + M[f, s]\}$ dove $F(x)$ è l'insieme dei figli di x in T, se x è un nodo interno.

La soluzione ottimale $\min_s \{M[r, s]\}$ dove r è la radice di T

Fitch – piccola parsimonia

Nell'ambito della filogenesi e alberi evolutivi solitamente non è possibile lavorare con modelli che impongono restrizioni molto forti, come per esempio il modello di filogenesi perfetta. Vengono quindi introdotti dei modelli (modelli evolutivi, es: Dollo, Comin Sokal) i quali impongono meno vincoli sulla definizione dell'albero, in questo caso si corre il rischio però di perdere l'unicità di quest'ultimo, ovvero possono esserci più alberi in grado di spiegare una matrice di caratteri/specie per seguire quale soluzione adottare è possibile utilizzare algoritmi basati su parsimonia: in particolare l'algoritmo di Fitch è in grado di risolvere il problema di piccola parsimonia così definito:

Istanze:

- Matrice M
- Topologia T dell'albero
- Per ogni carattere c, costo w_c per ogni coppia di stati

Soluzioni ammissibili:

- Per ogni carattere $c \in C$, una etichettatura λ_c che assegna a ogni nodo uno degli stati possibili per C

Funzione obbiettivo:

- Per ogni coppia di nodi adiacenti vengono applicate le funzioni λ, i quali risultati vengono dati in input alla funzione di costo del rispettivo carattere. Il costo dell'intero albero sarà dato dalla somma di tutti i risultati. Si va poi a minimizzare tale funzione

$$\min \sum_{c \in C} \sum_{(x,y) \in E(T)} w_c(\lambda_c(x), \lambda_c(y))$$

L'algoritmo di Fitch è un algoritmo bottom up il quale permette di risolvere il problema di piccola parsimonia con l'utilizzo della programmazione dinamica ma solo nel caso in cui l'albero in input T sia binario. In particolare, per ogni nodo x va a costruire l'insieme dei suoi stati e agisce come segue: per ogni nodo interno, se i due figli hanno almeno uno stato in comune ne prendono l'intersezione, altrimenti ne prendo l'unione

Formalmente:

$S(x) = \lambda_c(x)$ se x è una foglia

$S(x) = S(f_l) \cap S(f_r)$ dove f_l e f_r sono i figli di x in T,
se $S(f_l) \cap S(f_r) \neq \emptyset$

$S(x) = S(f_l) \cup S(f_r)$ dove f_l e f_r sono i figli di x in T,
se $S(f_l) \cap S(f_r) = \emptyset$

Unificazione:

$B(x)$: insieme degli stati z tali che $M[x, z]$ è minimo $\quad B(x) = S(x)$

Neighborn joning

L'algoritmo di Neighborn joning permette la ricostruzione di un albero evolutivo nell'ambito degli approcci di filogenesi basati su distanze, si definisce distanza la funzione

d:$SxS \rightarrow \mathbb{R}^+$ tale che

- $d(a,b) = 0 <=> a = b, \forall a,b \in S$
- $d(a,b) = d(b,a), \forall a,b \in S$
- $D(a,b) \leq d(a,c) + d(c,b), \forall a,b,c \in S$

L'input è una matrice simmetrica (matrice delle distanze) in cui, per ogni coppia di individui è contenuta una stima della distanza dal punto di vista evolutivo.

Gli algoritmi che si basano sul clustering di generazioni permetto di ricostruire l'albero attraverso la fusione iterativa di cluster, ovvero gruppi di discendenti per i nodi di quest'ultimo. Questi approcci partono dall'insieme banale delle foglie, fondendo due o più insiemi per la ricostruzione dell'albero. Ad ogni passo vengono scelti i cluster da fondere basandosi sulla matrice delle distanze in input.

Neighborn joning controlla per ogni cluster quanto esso è lontano dagli altri: l'idea di base è che la fusione avvenga tra gruppi (classi) vicini tra di loro ma distanti da tutti gli altri. Viene introdotta una nuova quantità $u(c)$, che rappresenta la misura di quanto una cluster sia serrato da tutti gli altri.

$$D(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{i \in C_1} \sum_{j \in C_2} D(i,j)$$

$$u(c) = \frac{1}{num\text{cluster} - 2} \sum_{C_3} D(C, C_3)$$

All'inizio dell'algoritmo $h = 0$ per ogni specie: i due cluster che hanno minimo $D(C_1, C_2) - U(C_1) - U(C_2)$ vengono fusi ottenendo C. Le etichette sono ottenute con $\frac{1}{2}\big(D(C_1, C_2) + u(C_1) + u(C_2)\big)$.

Vale la proprietà della consistenza: al tendere di n a infinto (lunghezza della sequenza), la probabilità che l'algoritmo costruisca l'albero corretto tende a 1.

Tempi:

La procedura che realizza Neighborn joning viene eseguita in tempo computazionale cubico rispetto al numero di specie