

Elementi di Bioinformatica



Lezione 1 - Bit parallel

Notazione

Simbolo $T[i]$

Stringa $T[1] \dots T[\ell]$

Sottostringa $T[i:j]$

Prefisso $T[:j]$

Suffisso $T[i:]$

Concatenazione $T_1 T_2$

Problema: Pattern Matching

Input: Ho un testo $T = T[1], \dots, T[n]$ ed un pattern $P = P[1], \dots, P[m]$

Goal: Trovare tutte le occorrenze P in T

Lower Bound: Il minor tempo di calcolo possibile è $\Theta(n+m)$ perché devo almeno leggere T e P

Con un algoritmo banale (uso dei for) otterrei $\Theta(n \cdot m)$. Introduciamo un algoritmo che seppur con tempo $\Theta(n \cdot m)$, è nell'effettivo più veloce in quanto vengono eseguiti pochi calcoli dalla cpu. Costante moltiplicativa minore.

Prima di introdurre l'algoritmo, introduciamo una forma di calcolo definita bit-parallel: eseguiamo 3 operazioni parallelamente su tutti i bit della macchina. Utilizzeremo come operazioni:

- and: $x \wedge y$
- or: $x \vee y$
- xor: $x \oplus y$
- right-shift: $x \gg k$
- left-shift: $k \ll x$

Lezione 1 - Bit parallel

Algoritmo di DomoK:

- Costruiamo una matrice con sulle righe i prefissi P e sulle colonne i caratteri del testo T . Inserisco \sqcup in posizione $M(i,j)$ sse $P[:i] = T[j-i+1:i]$ avendo se la lunghezza del testo fissato il pattern termina con lo stesso carattere:

	A	B	R	A	C	A	D	A	B	R	A
A	$\sqcup A$	O	O	$\sqcup A$	O	$\sqcup A$	O	$\sqcup A$	O	O	$\sqcup A$
AB	O	$\sqcup AB$	O	O	O	O	O	O	$\sqcup AB$	O	O
ABR	O	O	$\sqcup ABR$	O	O	O	O	O	O	$\sqcup ABR$	O

- Nota che per controllare se $M(i+1, j+1)$ matcha, devo solo considerare l'ultimo carattere, in quanto i caratteri precedenti sono stati già controllati da $M(i, j)$.

	A	B	R	A	C	A	D	A	B	R	A
A	$\sqcup A$	O	O	$\sqcup A$	O	$\sqcup A$	O	$\sqcup A$	O	O	$\sqcup A$
AB	O	$\sqcup B$	O	O	O	O	O	O	$\sqcup B$	O	O
ABR	O	O	$\sqcup B$	O	O	O	O	O	O	$\sqcup B$	O

- L'algoritmo che definisce ciò, effettua le seguenti operazioni:

3.1 Right-shift di $C[j-s]$

3.2 Inserisco \sqcup in cima

3.3 Eseguo AND tra $C[j]$ e il valore del right-shift

	A	B			A	B
A	\sqcup	$\sqcup \wedge B=A$			A	\sqcup
AB	O	$\sqcup \wedge B=B$	→	AB	O	\sqcup
ABR	O	$\sqcup \wedge B=R$		ABR	O	O

- Ottimizzo ulteriormente salvandomi le codifiche dell'alfabeto del testo: $U[\sigma]$

- Ottengo in forma compatta la seguente istruzione:

$$C[j] = ((C[j-s] >> s) | (s << C[j-s])) \wedge U(T(j))$$

Lezione 2 - Karp-Rabin

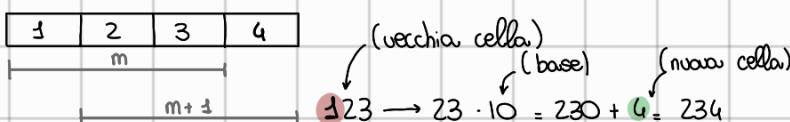
Dopo aver visto DomoKij, affrontiamo un nuovo algoritmo dove accetto come caso pessimo $O(n \cdot m)$ ma sarà raro finirci.

Introduzione all'idea Karp-Rabin: Data una stringa su alfabeto binario (per semplicità), considero la sua fingerprint, ossia il suo valore numerico, e uso uno sliding window di ampiezza m su T :

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

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

Ovviamente, lo sliding window successivo controlla solo l'ultima posizione, in quanto conosce il valore delle posizioni precedenti:



Effettuando solo operazioni algebriche su bit mi assicuro che è una implementazione valida, ma purtroppo non efficiente in quanto con m molto grande il costo delle operazioni non sarebbe unitario ma logaritmico, in quanto proporzionale al numero di bit.

Per ottenere tempi costanti faccio tutte le operazioni mod p primo casuale, ottenendo tutti i numeri $< p$:

· **problema**: ci sono più numeri che mod p sono uguali ($6 \equiv 1 \pmod{5}$ e $11 \equiv 1 \pmod{5}$) → genero falsi positivi.

· **soluzione**: utilizzo K numeri primi: così facendo ho il tempo moltiplicato per K , ma il tasso di errore diminuisce a q^K . Questo è possibile dato che siamo sicuri che non esistano falsi negativi.

Cambio p ogni volta che p genera FP. Se la sequenza genera FP per tutti i K p , allora è FP.

Osservazione: I migliori primi p da scegliere casualmente sono i più grandi vicini alla word size perché generano meno valori uguali.

Classificazione algoritmi probabilistici

· MonteCarlo: sempre veloce, non sempre corretto. Es. Karp-Rabin

· Las Vegas: non sempre veloce, sempre corretto Es. Quicksort con Pivot random.

Oss. Posso passare da MC a LV aggiungendo controlli, non vale viceversa.

Lezione 3 - Suffix-Tree-Array

Dopo aver visto due algoritmi con caso peggiore $O(n \cdot m)$, con il nuovo algoritmo riusciremo ad arrivare al lower bound $O(n+m)$. Cioè nonostante, si dimostrerà che nella pratica non è poi così veloce, e bisognerà "ribassare" la teoria per migliorarlo.

Introduciamo i tries: albero dove gli archi sono etichettati con le lettere, la query sull'albero consiste nel verificare se una parola appartiene all'albero, cioè se esiste il cammino radice-foglia.

problema: non riesco ad identificare le prefissi, in quanto non termino su una foglia.

Soluzione: aggiungo \$ alla fine delle parole, dove \$ è Vocabolario

Il trie è ottimo in quanto il tempo della query non cambia, dipende solo dal pattern, indipendente dal vocabolario

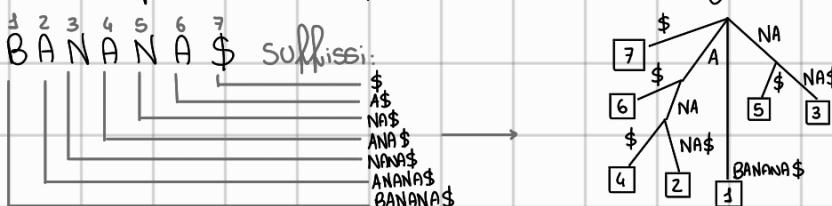
Il legame tra trie e pattern match è che ogni sottostringo è prefisso del suffisso. Se riesco a costruire il trie di tutti i suffissi risolvo il problema: tra i suffissi, i suffissi tali che il prefisso lungo m è uguale al pattern:

se lo costruisco su un generico dizionario, non ho legami

se lo costruisco sui suffissi, questi sono legati tra loro

Suffix tree: trie compatto di tutti i suffissi $T\$$: le etichette degli archi uscenti da x hanno iniziali diverse.

Esempio: B A N A N A \\$ suffissi:



Osservazione: affinché il pattern faccia match non è necessario consumare tutte le lettere dell'arco:

se il pattern è NAN match con NANA senza consumare la A.

Definizioni utile:

- path-label(x): concatenazione etichette
- string-depth(x): lunghezza path-label(x)
- pattern-matching(x): visita

Lezione 3: Suffix-Tree - Array

Problema: $O(n^2)$ di spazio richiesto

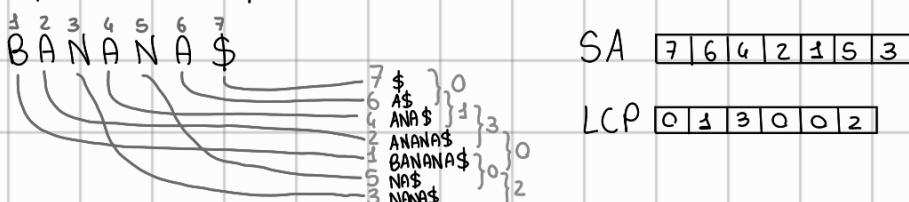
Soluzione: Solviamo gli archi con gli indici.

utilizza 3 puntatori per inizio e uno per length

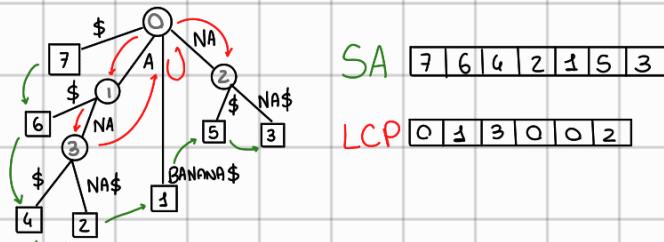
Problema: spazio per carattere 20 n bytes \rightarrow genoma umano in 128 gb e brutta località

Soluzione: suffix-array, bastano 4n bytes \rightarrow genoma umano in 36 gb e buona località

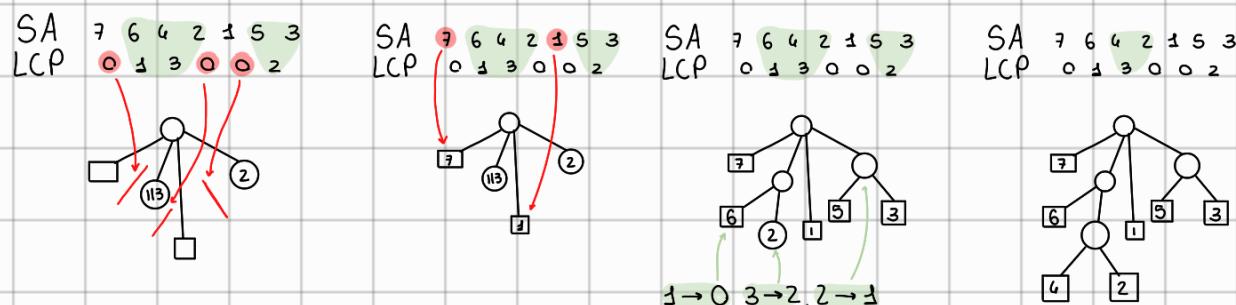
Suffix-array: Array dei suffissi in ordine lessicografico, posizioni iniziali del suffisso memorizzati. Inoltre salvo con $Lcp[i]$: lunghezza prefisso comune tra due suffissi consecutivi $SA[i], SA[i+1]$



Suffix tree \rightarrow Suffix Array: Per costruire il SA partendo da ST mi basta visitare in profondità e ordire l'albero. Per LCP ogni volta che salgo/scendo lungo un arco, tolgo/aggiungo la string-depth. LCP serve per poter attraversare in modo efficiente sia top-down che bottom-up

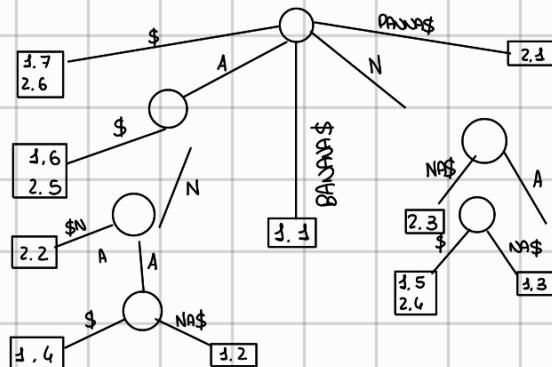


Suffix array (+ LCP) \rightarrow Suffix-tree: Prendiamo LCP: ogni volta che tra 0 so che ho una nuova partizione (prefissi diversi). Altrimenti gli elementi di LCP indicano il numero di nodi mancanti, a cui sottraggo 1. Da eseguire ricorsivamente



Lezione 3 - Suffix-Tree-Array

Suffix-tree generalizzato. Posso prendere ora più stringhe e generalizzare il problema LCS sul suffix-tree generalizzato. Esempio con BANANA\$ e PANNA\$



Longest Common Subsequence su Suffix-Tree generalizzato: (non pattern matching!)

Solo nelle foglie ho parole che considero (z, z, \dots, n) e le posizioni $(1, \dots, m)$. Come si fa LCS?

Le concateno e distinguo i dollari: BANANA\$, PANNA\$₂ e genero SF. Ora posso avere due tipi di sottostringhe: quelle che sono suffisso solo a w₂ oppure a entrambe: BANANA\$, PANNA\$₂

Se un nodo è suffisso di una stringa, lo è anche il nodo padre: etichetto ogni nodo con le stringhe della quale è suffisso in bottom-up (ottimizzo): metto T/F.

Per effettuare LCS leggo l'albero e cerco nodo con le etichette delle stringhe e string-depth maggiore (top-down)

PM SA vs PM ST: tree $O(m+k)$ vs array $O(m \log n + k)$ con $m = |\text{pattern}|$ e $n = |\text{text}|$.

- tree percorre l'albero (lineare)

- array ricerca dicotomica (si aggiunge log)

Riassunto idea PM su GST: solo in ogni nodo un array di K booleani dove K è il numero di parole. Se array[i]=1, allora il nodo ha almeno una foglia che contiene la i-esima parola

Problema: per ogni nodo abbiamo $K \cdot n$ operazioni e quindi tempo $O(K \cdot n^2)$?

Soluzione: ricordando che ogni nodo ha un solo padre e quindi ogni nodo viene letto 1 volta come padre e 1 come figlio. $O(n \cdot K^2)$ in visita bottom-up.

Lezione 4 - P1 - Suffix-Array

Ideas: Per fare P1 su SA posso fare ricerca dicotomica $O(\log_2 n)$ e dovrei controllare tutto $O(m)$ ottenendo tempo $O(m \log_2 n)$. Vogliamo velocizzare.

Acceleranti: Introduciamo 3 acceleranti per ridurre il tempo (i primi due nell'effettivo, l'ultimo a livello teorico)

Accelerante 1: Avendo un SA e un suo intervallo $SA(L, R)$ di elemento mediano M. Se $S[L]$ e $S[R]$ cominciano con n caratteri, allora i primi n caratteri del loro intervallo sono uguali e posso evitare di confrontarli.

Problema: non ho niente nel LCP che mi dice il numero di caratteri uguali tra L e R.

Accelerante 2: Denoto $l = lcp(L, P)$ e $r = (R, P)$ dove P è il pattern. Mi posso trovare in 3 casi:

Caso $l > r$: Calcolo $m = lcp(L, M)$

1. $l > m$: faccio match sopra m e quindi $r = m \wedge R = M$

2. $l < m$: faccio match + lungo sotto m e quindi $L = M$

3. $l = m$: confronto dal carattere $l+1$

Caso $l < r$: simmetrica a $l > r$

Caso $l = r$: calcolo $m = lcp(L, M)$ e $m' = lcp(M, R)$

$l < m$: faccio match + lungo sotto a M e quindi $L = M$

$l < m'$: caso opposto al precedente: $R = M$

$m = m'$: confronto il carattere successivo



Teoria: ricerca logaritmica e numero costante di operazioni, impiego $O(\log n)$?

Pratica: abbiamo aggiornare l e r 2m volte al massimo. Impiego di più di $O(\log n)$

Problema: come faccio ad ottenere i valori necessari per aggiornare l e r?

Soluzione: se riesco a pre processare il SA per ottenere $lcp(L, M)$ ci impiego $O(m + \log n)$

aggiorno ricerca
 \downarrow \downarrow

Lezione 4 - P1 - Suffix-Array

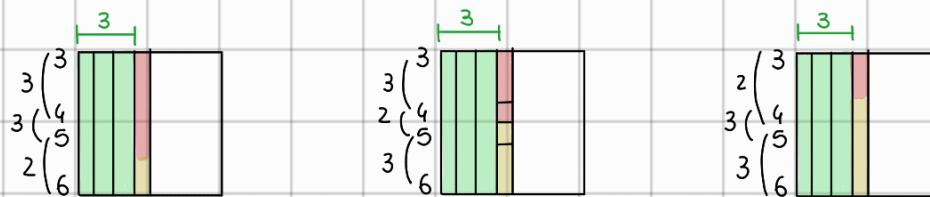
Accelerante 3: Devo considerare il mio array sempre /2.

L
1)
2)
3)
4)
5)
6)
7)
R
8)

Quando calcolo $LCP(L, R)$ ho due casi:

- se L, R sono adiacenti uso LCP

- altrimenti: considero di saper risolvere i problemi + piccoli (op). se so $LCP(1, 4)$ e $LCP(5, 8)$ voglio calcolare $LCP(1, 8)$ posso usare i casi adiacenti intermedi. nell'esempio $LCP(4, 5)$. Posso trovarmi in 3 casi:



Posso confermare che $LCP(5, 8) = \min\{LCP(3, 4), LCP(4, 5), LCP(5, 8)\}$

Conclusioni: Posso processare in tempo lineare $O(n)$ e effettuare la richiesta richiede $O(m \log n)$

Sottostringa comune più lunga. Abbiamo visto come trasformare il PM dal GST al SA per migliorare spazio e località. Ora poniamo la stessa trasformazione per il LCS.

Lezione 5 - rmq

Possaggio LCS su ST a LCS su SA. Per poter effettuare il Longest Common Substring sul suffix-array ci basiamo su 3 concetti fondamentali:

- Se nel ST usavamo un root, adesso prendiamo dei sottoorray
- Nel root solleviamo se esistevano nei discendenti entrambe le stringhe, ora basta guardare l'intervallo stesso
- la string depth del ST è simulabile tramite LCP nel ST.

Intervallo buono:

- deve contenere almeno un suffisso di ogni parola
- per ogni parola mi basta esattamente un suffisso.

Intervallo dominante: vogliamo quindi l'ampiezza d'intervallo minima che contenga tutte le parole. Per farlo ci solviamo in un array $\text{last}[i]$ dove memorizzo l'ultimo indice in cui ho visitato la parola i -esima.

Lunghezza LCS: Dato l'intervallo dominante i -esimo, LCS sarà il minimo LCP partendo da j aumentando di 1: $\text{LCS} = \min(\text{lcp}[j, i+1])$. Tempo lineare? No, devo calcolare il minimo lcp e quindi ho due scorrimenti, quindi $O(n^2)$.

Range min query: Data un array A , per trovare il minimo in un range $A[i:j]$ uso una matrice $B[i:h]$ che contiene il minimo di $A[i:i+2^h]$.

Idea: Preprocessiamo i calcoli per ottenere il minimo in modo tale che l'accesso a quest'ultimo sia $\Theta(1)$.

Lezione 5 - rmq

Esempio:

\downarrow	1	2	3	4	5	6	7	8
A.	5	2	4	6	6	1	3	4
—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—

\downarrow	1	2	3	4	5	6	7	8
B.	5	2	4	6	6	1	3	4
1 min	2	2	4	6	1	1	3	4
1 min	2	2	3	3	1	1	3	4
1	1	1	1	1	1	1	1	1

$$\log_2 8 = 3$$

soluzione ottimale per A[3:8]

Tempi: Conoscendo le due metà. $\min(m_1, m_2)$ è $\Theta(\downarrow)$ $\rightarrow A[i, h] = \min(B[i, h], B[i + \log_2 h, h])$

Implementazione: Dato che se conosco le due metà lavoro in tempo costante, divido l'array in 2 parti: con ciascuna ampiezza pari alla massima potenza di 2 inferiore a h, in modo che possa coprire tutto l'array con minimo spreco, e tra i due prendo il minore.

Formulari: $RMA(v, w) = \min\{B[v, q], B[w - 2^{\lfloor \log_2(w-v+1) \rfloor}, q]\}$ tc $q = \lfloor \log_2(w-v+1) \rfloor$

\downarrow 1 2 3 4 5 6 7 8

Esempio: Preso A.

5	2	4	6	6	1	3	4
---	---	---	---	---	---	---	---

 calcolo vari RMA

$$RMA(2, 5): q = \lfloor \log_2(5-2+1) \rfloor = 2, \text{ rmq}(2, 5) = \min\{(2, 2), (2, 2)\}$$

2	4	6	6
---	---	---	---

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

$$RMA(3, 7): q = \lfloor \log_2(7-3+1) \rfloor = 2, \text{ rmq}(3, 7) = \min\{(2, 2), (4, 2)\}$$

4	6	6	1	3
---	---	---	---	---

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

Lezione 5-mq

Osservazione: Il tempo per trovare tutte le occorrenze di LCS nel ST è $O(n+m+K)$ per K occorrenze

Costruzione suffix-array: Voglio tempo lineare. Ho alfabeto Σ con σ simboli, testo T lungo n .

Allora aggrego triple di caratteri ed ottengo alfabeto Σ^3 con σ^3 simboli, testo T lungo $n/3$:

$$T_3 = (T[3], T[2]T[3]) \dots (T[3i+1], T[3i+2]T[3i+3]) \dots$$

$$T_2 = (T[2], T[3]T[4]) \dots (T[3i+2], T[3i+3]T[3i+4]) \dots$$

$$T_1 = (T[1], T[2]T[3]) \dots (T[3i], T[3i+1]T[3i+2]) \dots$$

Ricorsione: Uso un algoritmo ricorsivo basato su radix-sort e merge-sort. La ricorsione è un po' più complicata in quanto, per ottenere linearità raggruppa il testo in triple ed effettua la ricorsione solo su $2/3 n$ del testo, scelto in modo tale che ricaviamo l' $'1/3$ mancante.

1. Ricorsione su $T_0 T_3$

2. suffissi $(T_0 T_3) = \text{suffissi}(T_2) \rightarrow T_2[i:] = T[3i+2:]$

3. suffissi T_0 ordinati:

$$T[3i+2]T[3i+3:]$$

$$T_0[i+1:]$$

4. singola passata radix-sort

5. fusione suffissi $(T_0 T_3)$ e T_2

Vantaggio: Il vantaggio rispetto al merge è che i testi sono "accavallati" e posso sfruttare i caratteri in comune.

Lcp: Quando fondiamo, se il carattere di due el. consecutivi sono uguali av Lcp, avremo quello calcolato al passo precedente + 1, altrimenti 0.

L'ezione 6 - Alignment Globale

Programmazione dinamica: Tecnica di programmazione che può essere sfruttata quando il problema è suddivisibile in più sottoproblemi e ha una sottostruttura ottima. Possiamo affermare che la soluzione ottimale ad un problema non cambia se esso diventa sottoproblema di un problema più esteso.

Allineamento: Due sequenze s_1 e s_2 vengono allineate attraverso operazioni di insert (-), ovvero insert e delete. A differenza della distanza di Hamming che misura solo il n° di caratteri diversi, qui non abbiamo il vincolo sull'uguaglianza di lunghezza di testi e possiamo assegnare importanza.

Limitazione: Prese due stringhe s_1 e s_2 e messe in colonna, non posso avere alla stessa colonna 2 (-)

Valore di allineamento: Il valore di allineamento è la somma dei valori delle singole colonne. Viene dato in input, insieme a s_1 e s_2 , la matrice di score, maggiore il coefficiente, maggiore sarà il valore della coppia, ovvero più raro.

Algoritmo di Needleman-Wunsch: L'ultimo componente della mia equazione di ricorrenza è l'ultima colonna. Prendendo ispirazione dalla Edit Distance possiamo trovarci in 3 casi:

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

Costo/Valore: Se voglio sapere il costo delle operazioni, avrò un problema di minimizzazione, se voglio sapere il valore sarà un problema di massimizzazione.

Condizioni di contorno:

$$M[0, 0] = 0$$

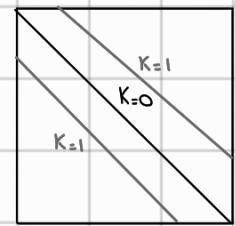
$$M[i, 0] = M[i-1, 0] + d(s_1[i], -)$$

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

Lezione 6 - Alignment Globale

Edit Distance: Voglio sapere se la distanza di edit di due sequenze è $= K$. Con l'algoritmo precedente impiego $\Theta(n \cdot m)$. Voglio impiegurci meno.

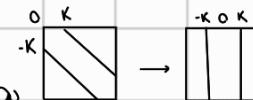
$K=0$ calcolo bande in $\Theta(n)$ in quanto rimango sulla diagonale



$K>1$ calcolo non bande in $\Theta(Kn)$: preso la matrice mi sposto dalla diagonale di max K posizioni, quindi non necessito di tutta la matrice.

Problema: Nell'esempio precedente ho dato per scontato che conosciamo K . Se così non fosse dobbiamo stimare K : partiamo da una banda piccola, così è veloce. Se il percorso è interno allora ho trovato un K buono, altrimenti raddoppio K finché la sol non tocca la banda

Spazio: Per ottenere anche spazio $\Theta(Kn)$ devo traslare la banda



L'ezione 7 - Allineamento Locale e Multiplo

Allineamento locale: date in input due stringhe s_1, s_2 e la matrice di score d , vogliamo individuare le sottostinghe t_1 di s_1 e t_2 di s_2 tali che l'allineamento globale sia massimo.

Algoritmo banale: Calcolo tutte le sottostinghe di s_1 e s_2 e calcolo allineamento globale

Tempo: avrei n^2 sottostinghe per s_1 , m^2 sottostinghe per s_2 e impiego $m \cdot n$ per allineamento globale. $\Theta(n^3 \cdot m^3)$. Vogliamo migliorare

Osservazione: usiamo un problema di massimizzazione del valore e non di minimizzazione del costo in quanto avremmo valore ottimale con caso base $t_1 = t_2 = \epsilon$

Miglioramento: Pensando al problema con i prefissi posso usare la matrice di Needleman-Wunsch dove $M[i, j] = \text{ottimo su } s_1[:i] \text{ e } s_2[:j]$

Tempi migliorato: Ricordando che una sottostringa è prefisso di un suffisso, prendo a coppie di suffissi ed eseguo l'algoritmo sui prefissi. $t_1 = s_1[a:i]$, $t_2 = s_2[b:j]$. Ho $n \cdot m$ suffissi e l'algoritmo è sempre $n \cdot m$, quindi: $\Theta(n^2 \cdot m^2)$

Passaggio globale → locale: $M[i, j] = \text{ottimo fra tutte le sottostinghe che finiscono in } i \text{ e } j$.

Equazioni di ricorrenza: Ora non lavoro più con stringhe come in NW ma con sottostinghe. Dobbiamo elencare tutti i casi per l'ultimo componente, ovvero ultima colonna. Avremo gli stessi casi di NW ma con una novità: l'ultima colonna può essere vuota! (è una sottostringa.)

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

Cessione 7 - Allineamento Locale e Multiplo

Calcolo soluzione: Sappiamo quindi calcolare la fine della sottostringa (max nella tabella) ma non dove inizia.

Posizione iniziale: La trovo ricostruendo la soluzione e fermandomi al primo 0 che incontro. So che qualsiasi elemento precedente può solo che abbassare il max in quanto ≤ 0 . Per essere sicuri di trovare almeno uno 0 aggiungo una condizione di controllo

Condizioni di controllo: $M[i,0] = M[j,0] = 0$

Tempo: $O(m \cdot n)$

Osservazione centrale: ogni casella intermedia della matrice non sono risultati temporanei ma valori che riusciamo a risfruttare e ci servono!

Allineamento multiplo: Dobbiamo estendere il problema affinché ragioni su s_1, \dots, s_K stringhe con len diverse e se $K=2$ sia la def. all. glob tra 2 stringhe.

Allineamento: inserimento di indel per ottenere le stringhe estese s_1^*, \dots, s_K^* tutte di uguale lunghezza. Proibisco le colonne con tutti indel ($\forall p \exists S_i^*[p] \neq -$)

Calcolo: Per il calcolo del valore dell'allineamento posso fare la somma delle distanze per ogni coppia $p = \sum_{i,j} d(s_i^*[p], s_j^*[p])$ con controllo che almeno uno dei due sia $\neq -$.

Equazione di ricorrenza: Nell'ultima colonna ho K elementi che possono assumere 2 valori: valore e indel, quindi $2^K - 1$ combinazioni (-1 per caso tutti indel)

Osservazione: Avrei matrice K dimensionale. Se avessi tutte le lunghezze uguali ho una matrice 2^K e tempo $O(2^K \cdot n^K)$. Il problema diventa intrattabile in fretta. Se la lunghezza non fosse fissata diventa NP-complesso. Inoltre il maggior limite è la memoria: con cella=8b, n=1000 e K=4 necessito 3TB di ram

Lezione 8 · Gap

Finora abbiamo considerato gli indel come costo fisso. Ora avranno un costo variabile e dobbiamo cercare di minimizzarlo.

Gap: Sequenza contigua di indel in un allineamento. Spostando il frame di lettura è penalizzante. Un nuovo gap penalizza di più di un prolungamento.

Costo gap: Funzione $P(l)$ che prende in input la lunghezza di un gap e restituisce il costo.

Gap generico - equazione di ricorrenza: Devo elencare tutti i casi possibili per l'ultima colonna. Novità: equazione di ricorrenza con numero di casi variabili.

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

$$M[0, 0] = 0$$

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



Tempo $(n \cdot m \cdot (n+m))$. Non riusciamo a migliorare, allora cambiamo la funzione costo.

Gap affine: $P(l)$ è una retta: $P(l) = P_0 + lP_e \rightarrow$ extension
 ↓
 opening

Equazione di ricorrenza: devo separare i casi in cui ho aperto un nuovo gap o sto estendendo.

$$M_1: \text{opt}(s_1[i-1], s_2[j-1]) + d(s_1[i], s_2[j])$$

$$N_1: \text{opt}(s_1[i-1], s_2[j]) + P_0 + P_e$$

$$E_1: \text{opt}(s_1[i-1], s_2[j]) + P_e$$

$$N_2: \text{opt}(s_1[i], s_2[j-1]) + P_0 + P_e$$

$$E_2: \text{opt}(s_1[i], s_2[j-1]) + P_e$$

Lezione 8 - Gap

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

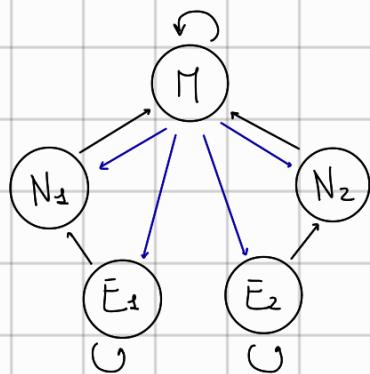
$$E_1[i, j] = \max \begin{cases} E_1[i, j-1] + P_e \\ N_1[i, j-1] + P_e \end{cases} \quad E_2[i, j] = \max \begin{cases} E_2[i-1, j] + P_e \\ N_2[i-1, j] + P_e \end{cases}$$

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

Spazio: Ω^{mn} e accesso costante al valore quindi:

Tempo: Ω^{mn})

Sincronizzare: Devo sincronizzare le loro costruzioni.



Scopo allineamento: Trovare nei DB similarità della mia entry e quanto questa sia casuale.

Karlin-Atschul: Quantificazione numerica dell'allineamento trovato.

Punteggio positivo possibile

Punteggio medio negativo

Simboli indipendenti ed equiprobabili

Sequenze infinitamente lunghe

Allineamenti senza gap

Lezione 8 - Gap

Equazione Karlin-Atschul: $E = Kmne^{-\lambda s}$

E : numero allineamenti;

K : costante

n : numero caratteri database

Cresce linearmente rispetto alle dimensioni.

m : lunghezza stringa query

λs : punteggio normalizzato

Decresce esponenzialmente rispetto al valore che mi interessa.

BLAST: Basic Local Alignment Search Tool

value = probabilità che ho trovato allineamenti per caso.

BLAST non può usare Smith-Waterman perché DB troppo grande, preferisce algoritmo euristico.

Ricerca seed: cerco porzioni identiche lunghezza = 3

Costruzione HSP: high-scoring segment pair, estensione seed.

Filtro seed: tenuti solo HSP con alta significatività;

Funzione HSP vicine

Smith-Waterman sulle regioni.

In questo modo siamo sicuri di evitare valori inutili.

