

Elementi di Bioinformatica

UniShare

Davide Cozzi
@dlcgold

Indice

1	Introduzione	2
2	Introduzione alla Bioinformatica	3
2.1	Bit-Parallel	3
2.1.1	Algoritmo Dömölki/Baeza-Yates	4
2.2	Algoritmo Karp-Rabin	6
2.3	Trie	8

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: <https://github.com/dlccgold/Appunti>.

Grazie mille e buono studio!

Capitolo 2

Introduzione alla Bioinformatica

Un po' di notazione per le stringhe:

- **simbolo:** $T[i]$
- **stringa:** $T[1]T[2]\dots T[n]$
- **sottostringa:** $T[i : j]$
- **prefisso:** $T[: j] = T[1 : j]$ (inclusi gli estremi)
- **suffisso:** $T[i :] = T[i : |T|]$ (inclusi gli estremi)
- **concatenazione:** $T_1 \cdot T_2 = T_1T_2$

In bioinformatica si lavora soprattutto con le stringhe, implementando algoritmi, per esempio, di pattern matching. Nel pattern matching si ha un testo T come input e un pattern P (solitamente di cardinalità minore all'input) da ricercare. Si cerca tutte le occorrenze di P in T . L'algoritmo banale prevede due cicli innestati e ha complessità $O(nm)$ con n lunghezza di T e m lunghezza di P . Il minimo di complessità sarebbe $O(n + m)$ (è il **lower bound**). Si ragiona anche sulla costante implicita della notazione O-Grande cercando di capire quale sia effettivamente l'algoritmo migliore con la quantità di dati che si deve usare. Bisogna quindi bilanciare pratica e teoria.

2.1 Bit-Parallel

È un algoritmo veloce in pratica ma poco performante a livello teorico, ha complessità $O(nm)$.

```

for  $i = 1 \rightarrow n$  do
   $trovato \leftarrow true$ 
  for  $j = 1 \rightarrow m$  do
    if  $T[1 + j - 1] \neq P[j]$  then
       $trovato \leftarrow false$ 
    end if
  end for
  if  $trovato$  then
     $print(i)$ 
  end if
end for

```

Questo algoritmo è facilmente eseguibile dall'hardware del pc.

In generale si hanno **algoritmi numerici** che trattano i numeri e gli **algoritmi simbolici** che manipolano testi.

Si hanno poi gli **algoritmi semi-numerici** che trattano i numeri secondo la loro rappresentazione binaria, manipolando quest'ultima con *or* \vee , *and* \wedge , *wedge*, *xor* \oplus , *left-shift* \ll e *right-shift* \gg . Ricordiamo che il left shift sposta di k posizioni a sinistra i bit, scartandone k in testa e aggiungendo altrettanti zeri in coda (lo shift a destra sposta a destra, scarta in coda e aggiunge zeri in testa). Queste sono operazioni bitwise e sono mappate direttamente sull'hardware, rendendo tutto estremamente efficiente.

2.1.1 Algoritmo Dömölki/Baeza-Yates

Questo algoritmo viene anche chiamato **algoritmo shift-and** o anche **bit parallel string matching**.

Si definisce in input una stringa T di cardinalità n e un pattern P di cardinalità m .

Si costruisce una matrice M *ipotetica*, di dimensione $n \times m$, con un indice i per P e uno j per T dove:

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

Quindi $M(i, j) = 1$ sse i primi i caratteri del pattern sono uguali alla sottostringa lunga i in posizione $j - i + 1$ del testo.

Questa matrice è veloce da costruire e si ha:

$$M(m, \cdot) = 1, \quad M(0, \cdot) = 1, \quad M(\cdot, 0) = 0$$

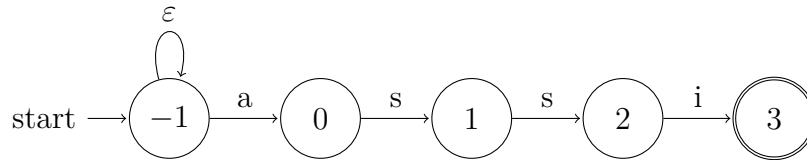
$$M(i, j) = 1 \text{ sse } M(i-1, j-1) \text{ AND } P[i] = T[j]$$

la prima riga saranno tutti 1 ($M(0, \cdot) = 1$) in quanto la stringa vuota c'è sempre mentre la prima colonna saranno tutti 0 ($M(\cdot, 0) = 0$) in quanto un testo vuoto non matcha mai con una stringa non vuota.

Quindi la matrice avrà 1 solo se i primi caratteri del pattern $P[i]$ sono uguali alla porzione di testo $T[j-i+1:j]$. Ma in posizione $M(i-1, j-1)$ mi accorgo che ho 1 se ho un match anche con un carattere in meno di P e T. Quindi se $M(i-1, j-1) = 0$ lo sarà anche $M(i, j)$. Se invece $M(i-1, j-1) = 1$ devo controllare solo il carattere $P[i]$ e $T[j]$ e vedere se $P[i] = T[j]$. Ovvero, avendo $P = \text{assi}$ e $T = \text{apassi}$ si avrebbe (omettendo la prima riga e la prima colonna in quanto banali):

		j	1	2	3	4	5	6
i		a	p	a	s	s	i	
1	a	1	0	1	0	0	0	
2	s	0	0	0	1	0	0	
3	s	0	0	0	0	1	0	
4	i	0	0	0	0	0	1	

Con un automa non deterministico che accetta una stringa terminante con P sarebbe:



La matrice la costruisco con due cicli e controllo solo l'ultima riga. Non si ha un guadagno a livello di complessità, dato che rimane $O(nm)$, ma grazie all'architettura a 64 bit della cpu. Infatti con una word della cpu posso memorizzare una colonna intera, in quanto vista come numero binario. Ora lavoro in parallelo su più bit, con un algoritmo **bit-parallel**, facendo ogni volta 64 confronti tra binari. In questo modo crolla la costante moltiplicativa nell'O-grande.

Ma come passo da una colonna $C[j]$ a una $C[j-1]$? Con questi step:

- la colonna $C[j]$ corrisponde al right shift della colonna $C[j-1]$
- aggiungo 1 in prima posizione per compensare lo shift
- faccio l'AND con $U[T[j]]$, che è un array binario lungo come il pattern dove ho un binario con 1 se è il carattere di riferimento:

P=abca
 U[a]=1001
 U[b]=0100
 U[c]=0010

- ragiono sul word size ω in caso di pattern più grandi di 64bit.

ottengo:

$$C[j] = ((C[j-1]) \gg 1) | (1 \ll (\omega-1) \& U[T[j]])$$

Conoscendo una colonna della matrice voglio calcolare la successiva. Quindi $M[i, j] = M[i-1, j-1] \text{ AND } P[i] = T[j]$ (per esempio, $M[1, j] = \text{TRUE AND } (p[i] = T[j])$), cioè conta solo il confronto dei caratteri.

Ogni 1 nell'ultima riga corrisponde ad un'occorrenza.

Questo algoritmo ha il vantaggio di non avere branch if/else, però si ha un limite nella lunghezza del pattern (64 bit) pattern e l'uso di più word comporta il riporto sulla colonna seguente, fattore che si complica all'aumentare della lunghezza del pattern, soprattutto se arbitraria.

2.2 Algoritmo Karp-Rabin

Vediamo un altro algoritmo di pattern matching che sfrutta una codifica binaria e che, pur non risultando sempre corretto, è estremamente più veloce, viene infatti eseguito in tempo lineare.

Uso un alfabeto binario e devo fare il match di due stringhe con ciascuna la sua codifica $H(S) = \sum_{i=1}^{|S|} 2^{i-1} H(S[i])$. Ad ogni carattere di una string si associa un numero nel range $[0, 2^{m-1}]$. Praticamente si usano due funzioni hash che trasformano una stringa in un decimale rappresentate in binario (ogni numero intero è facilmente rappresentabile come somma di potenze di 2 e quindi in binario). Viene quindi facile paragonare le due fingerprints. Mi muovo sul testo T mediante finestre di ampiezza m pari a quella del pattern e controllo il fingerprint di quella porzione con quella del pattern. Inoltre il fingerprint di una finestra è facilmente calcolabile da quello della precedente. Per farlo elimino il contributo del carattere della finestra precedente e includo l'unico aggiunto dalla finestra successiva, in quanto mi sposto di 1:

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

Essendo il primo carattere quello meno pesante viene rimosso ad ogni spostamento sfruttando la divisione per due per lo shift

La sottostringa è uguale al pattern solo se le fingerprint lo sono:

$$T[i : i + m - 1] = P \Leftrightarrow H(T[i : i + m - 1]) = H(P)$$

Per estendere la codifica binaria in k caratteri avrò la finestra che si sposta di k con la divisione per k anziché per 2.

Si ha il problema della lunghezza del pattern in quanto ho un 2^{m-1} che fa esplodere l'algoritmo perché usa un numero di bit grandissimo. Si ricorda che un'operazione "costa 1" solo se sono piccoli i numeri in gioco, nel nostro caso il costo diventa proporzionale al numero di bit coinvolti. La soluzione di Karp-Rabin è di continuare con la logica di sopra ma solo con numeri piccoli, cambiando la definizione di fingerprint prendendo il resto di quanto sopra con un numero primo p :

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

ma in questo modo la fingerprint non è più iniettiva, con la possibilità che più stringhe abbiano la stessa fingerprint e di conseguenza si avranno degli errori. Si ha che $2^{m-1}T[i+m]$ viene calcolato iterativamente facendo $\mod p$ ad ogni passo. Si può quindi avere una sottostringa di T con lo stesso fingerprint del pattern che però non è uguale al pattern, è un **falso positivo**. Non si possono tuttaaavia avere falsi negativi, quindi tutte le occorrenze sono trovate con la possibilità di trovare occorrenze false in più:

$$H(T[i : i + m - 1]) \mod p = H(P) \mod p \Leftarrow T[i : i + m - 1] = P$$

Se il numero primo p è scelto a caso minore di un certo I so che l'errore è minore di $O(\frac{nm}{I})$.

Vogliamo sfruttare però che si hanno solo falsi positivi e provare ad eseguire l'algoritmo con due p diverse, le vere occorrenze saranno trovate da entrambe mentre i falsi positivi probabilmente no. Itero quindi su k numeri primi e il risultato sarà l'intersezione di tutte le k iterazioni dell'algoritmo, riducendo moltissimo le probabilità di avere un risultato errato. Paghiamo quindi un incremento di un prodotto k delle operazioni (diventa $O(k(n + m))$) per ridurre esponenzialmente le chances di errore.

Proponiamo una versione semplificata dell'algoritmo (lunghezza del testo $= n$ e del pattern $= m$):

```

function RabinKarp(text, pattern)
  patternHash  $\leftarrow$  hash(pattern[1 : m])
  for  $i \leftarrow 1$  to  $n - m + 1$  do
    textHash  $\leftarrow$  hash(text[i : i + m - 1])
    if textHash = patternHash then
      if text[i : i + m - 1] = pattern[1 : m] then
        return(i)
      end if
    end if
  end for
  return(NotFound)
end function

```

È quindi un algoritmo probabilistico in quanto i p sono scelti a caso. Ci sono due categorie di algoritmi probabilistici:

1. **Monte Carlo**, come Karp-Rabin, veloci ma non sempre corretti
2. **Las Vegas**, sempre corretti ma non sempre veloci, come per esempio il quicksort con pivot random (dove il caso migliore è un pivot che è l'elemento mediano mentre il peggiore è che il pivot sia un estremo, portando l'algoritmo ad essere quadratico).

È possibile rendere Karp-Rabin un algoritmo della categoria Las Vegas controllando tutti i falsi positivi (anche se non è una procedura utilizzata).

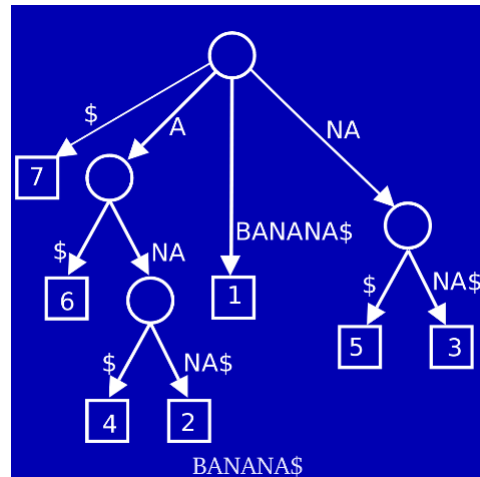
2.3 Trie

Si usi la filosofia che prevede il preprocessing del testo.

Il **trie** è una struttura ad albero, con archi etichettati, che, preso un insieme di parole, detto dizionario, controlla che quella sequenza sia nell'insieme di parole. Si tratta di un **problema di membership**. Voglio un tempo $O(n)$ con n lunghezza della query. Quindi si preprocessa una volta sola il dizionario in un albero e poi si procede con le query. Nella pratica si ha che un percorso radice-foglia deve essere esattamente la query richiesta. Si ha però un problema, essendo ogni ramo un elemento del dizionario, non si possono avere parole diverse nel dizionario che siano l'una prefissa dell'altra (per

esempio se ho *abraabra* non posso avere anche *abra* nel dizionario), perché non riuscirei ad andare da radice a foglia. Introduco quindi il carattere \$, il quale non appartiene all'alfabeto che viene aggiunto alla fine di ogni stringa, viene infatti detto **terminatore**, così *abra*\$ non è prefisso di *abraabra*\$ etc..., rimuovendo così ogni ambiguità.

Usiamo una struttura dati chiamata **suffix tree**, che è il trie di tutti i suffissi di $T\$$, che quindi è un insieme più specifico di stringhe (i suffissi). È una sorta di trie compatto, dove un suffisso diventa l'etichetta di un arco. Le etichette degli archi uscenti, i figli, da X iniziano con simboli diversi. I suffissi sono il percorso radice-foglia.



Una sottostringa è il *prefisso di un suffisso*. Dato un pattern voglio trovare tutti i suffissi che iniziano col pattern. Quindi, dato che ogni nodo ha un solo figlio con un certo prefisso, la procedura di pattern matching nel suffix-tree naviga nell'albero seguendo il pattern nell'unico arco possibile con quel pattern, posto che esista. Se non esiste il pattern cercato termina, così come termina nel momento in cui lo trova raggiungendo un \$. Il compattamento da trie a suffix tree serve a migliorare le performance di costruzione della struttura dati, non quelle del pattern matching in sé. Un suffix-tree ha un rapporto tra il numero di nodi e il numero di foglie (?).

Questo pattern matching ha $O(m)$ con m lunghezza del pattern da ricercare ma poi ho tante foglie k sotto il nodo a cui sono arrivato quante sono le occorrenze del pattern nel testo, che quindi visito in $O(k)$. Nel complesso ho la costruzione dell'albero in $O(n)$, n lunghezza testo, matching in $O(m)$ e visita finale delle foglie in $O(k)$, quindi nel complesso ho

$$O(n + m + k)$$

Se il pattern termina prima del passaggio ad un nodo successivo non mi interessa in quanto i suffissi corrispondono (se ho NA e mi fermo a N va bene lo stesso, in quanto i suffissi di NA sono dello stesso numero di quelli di N). Nelle foglie ho indicato l'indice dove inizia ogni occorrenza.

Definisco la **path-label(X)** di un nodo X è la concatenazione delle stringhe fino a quel nodo. Definisco invece **string-depth(X)** di un nodo X la lunghezza del path-label, che è calcolabile in $O(n)$. La string-depth di X sarà la lunghezza dell'etichetta sommata alla string-depth del padre.

Molti algoritmi sfruttano molte visite per arricchire le informazioni dell'albero ai fini di rendere semplice la risoluzione di un problema.

Non posso fare lo stesso ragionamento per la path-label perché, concatenando quella del padre alla propria raggiungerei quasi $O(n^2)$. Si usa quindi una tecnica basata su puntatori al testo e non su stringhe, questo fa sì che ogni arco sia etichettato da una coppia di numeri (posizione di inizio e lunghezza), quindi raggiungo tempo costante per etichettare ogni albero.

Il grande problema del suffix-tree è lo spazio occupato, circa $20n$ byte, quindi per il genoma umano servono 60gb di memoria. Per risolvere questo problema si usano i **suffix array (SA)**, che occupa meno spazio, ed è *l'array dei suffissi in ordine lessicografico*. Il suffix array non permette il pattern matching in tempo lineare quindi viene usato il suffix array per costruire il suffix tree. Tutto quello che si può fare sul suffix tree si può fare sul suffix array con tempi diversi ma simili. Non memorizzo esplicitamente i suffissi ma memorizzo le posizioni iniziali del suffisso. Al suffix array si aggiunge l'array ausiliario *LCP (longest common prefix)* con la lunghezza del prefisso comune $LPC[i]$ tra $SA[i]$ e $SA[i + 1]$.

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

Lo spazio diventa $4n$ bytes, quindi per il genoma 12gb.

Passiamo ora dall'albero all'array. Facciamo una visita depth-firsts (pre-order) del suffix array, assumendo che a sinistra ci siano etichette in ordine lessicografico minore (a sinistra ho suffissi con una lettera dell'alfabeto iniziale "precedente").

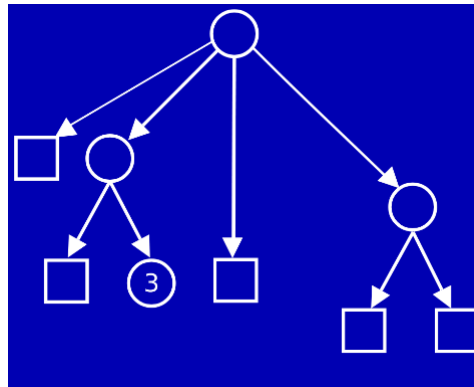
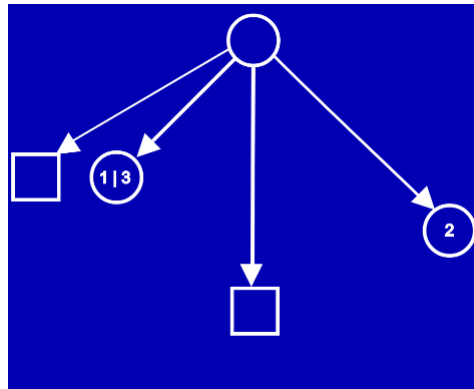
Ho quindi che $LPC[i]$ è la string-depth di $LCA(i, i + 1)$, ovvero *least common ancestor*, che quindi mi indica dove due percorsi divergono e vedo quanto vale lì la string-depth.

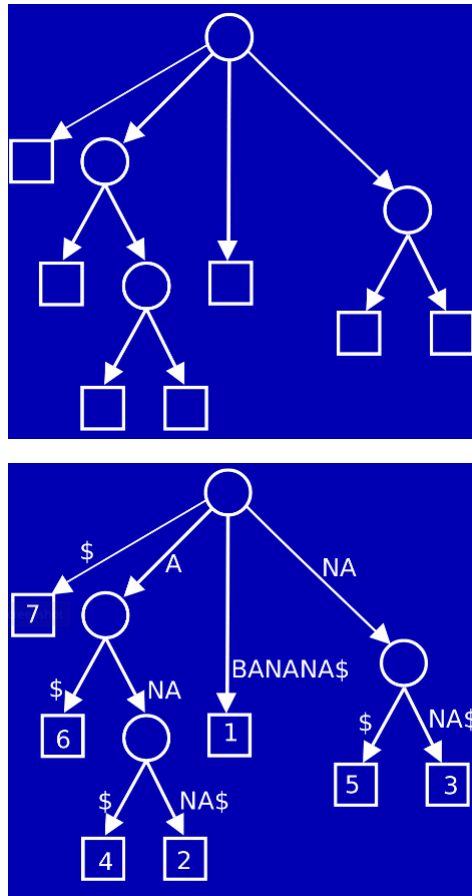
Calcolare LPC costa un tempo quadratico. Ma possiamo fare un'altra cosa. Ogni arco viene visitato almeno due volte inq aundo una volta arrivati ad una foglia si torna indietro. Mi salvo la string depth dell'LCA nell'array ogni

volta che ho un cambio di direzione nella visita dell'albero lavorando quindi in tempo lineare. Nel disegno guardiamo due foglie consecutive e contiamo i nodi che hanno in comune sopra esclusa la radice.

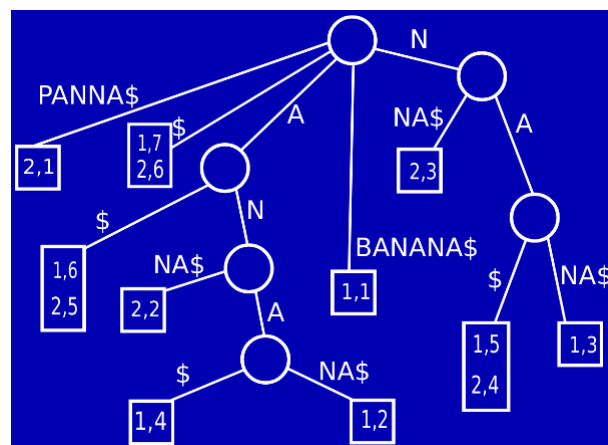
Dati SA e LCP passiamo ora a costruire l'albero. Gli LCP nulli partizionano il SA e corrispondono ai figli della radice del suffix array aggiungendo 1 (tre zeri corrispondono 4 figli). Di questi 4 figli prendo i valori non nulli e saranno un cammino che prosegue ripetutamente sul valore minimo.

Quindi se l'array è 013002 avrò 4 figli della radice, uno 0, uno 13, uno 00 (che conta come 0) e uno 2. Il 13 avrà una foglia a sinistra e poi scenderà di un nodo. Avendo solo un numero (3) avrà due foglie. Infine avrò due, che essendo un solo numero, avrà solo 2 foglie.





Un **suffix tree generalizzato** rappresenta un insieme di testi. Prendo x stringhe con terminatore, le concateno in un unico testo e ne genero il suffix tree ma nelle foglie avremo le coppie (numero stringa, posizione inizio suffisso) che potrebbero essere una per ogni stringa e lo costruisco in tempo lineare alla lunghezza del testo completo.



Cerco ora la sottostringa comunque più lunga tra due stringhe sfruttando il suffix tree generalizzato. Una sottostringa comune corrisponde ad un cammino tra una radice e un nodo X e tra i discendenti di X devo avere almeno un discendente per ogni stringa. La più lunga sarà quindi quella che arriva al nodo con string-depth massimo. Visito quindi due volte l'albero, alla prima vedo se ha le foglie giuste e la seconda per cercare il max delle string-depth. La prima visita la faccio dalle foglie verso la radice, creo un array di booleani per il nodo X lungo quanto il numero di stringhe i cui elementi valgono true se esiste un discendente di X tale che è suffisso della stringa S . Alla fine faccio l'and tra tutti i valori dell'array, se è true significa che ho un discendente per ogni stringa. Se X è nodo interno faccio l'or con l'array del nodo discendente per determinare se X va bene. Se è foglia determino il vettore leggendone l'etichetta. La seconda visita può essere in qualsiasi direzione e dove ho nodi con la proprietà di essere comune alle stringhe e trovo quello con la string-depth max, ottengo quindi un tempo che è $O(k \cdot n)$.