CLELIA DE FELICE - ROCCO ZACCAGNINO - ROSALBA ZIZZA

STRUMENTI FORMALI PER LA BIOINFORMATICA
2024/2025

ANALISI DELL'ALGORITMO PER LA RICERCA DI INTERVALLI NEL SUFFIX ARRAY UTILIZZANDO BURROWS-WHEELER ALIGNMENT TOOL

EFFICIENZA E APPLICAZIONI PER IL PATTERN MATCHING DI SHORT READ

Università Degli Studi Di Salerno

Gaita Irene - 0522501839

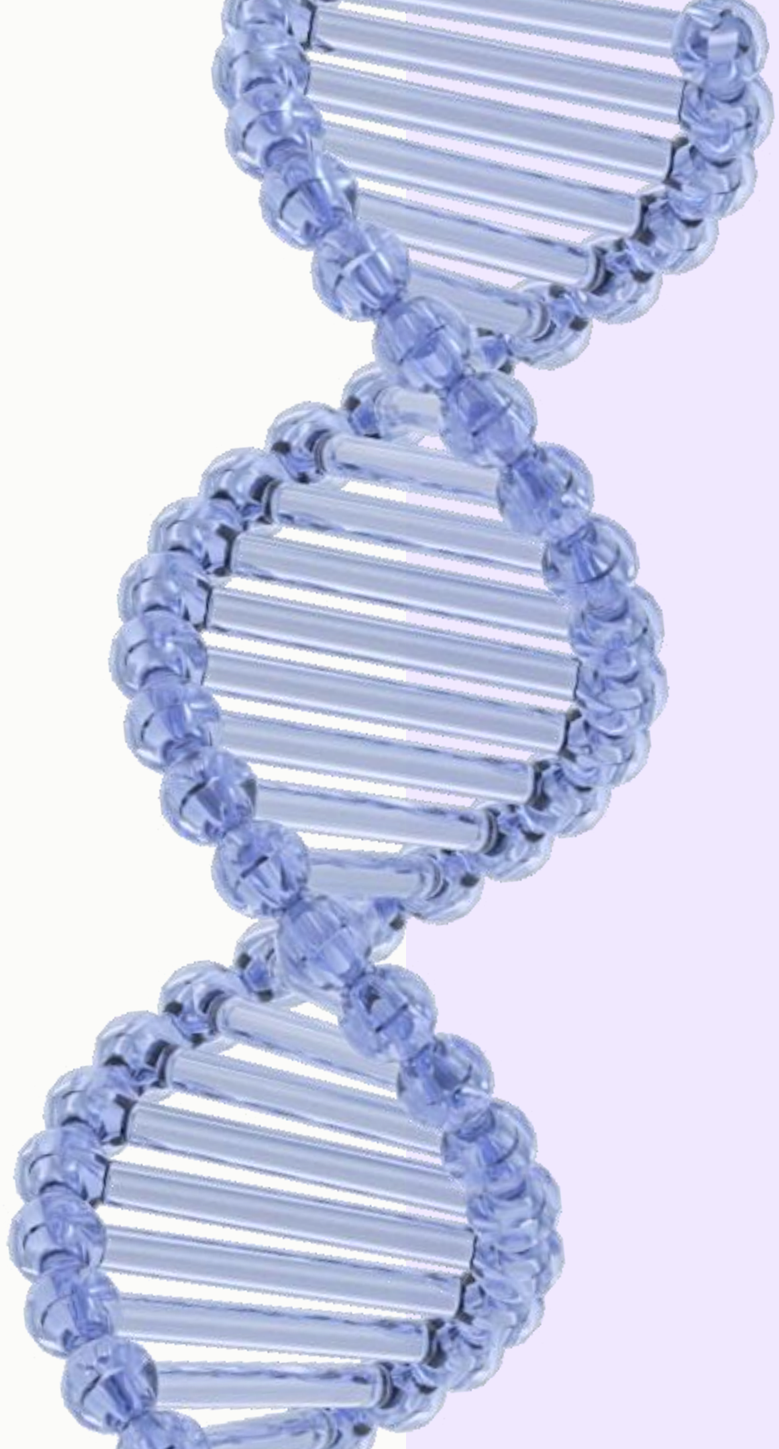
Perché è Importante il Sequenziamento del DNA?

- **Decodifica del codice genetico**
  Comprendere come le istruzioni genetiche regolano la crescita, lo sviluppo e le funzioni degli organismi.

- **Salute umana**
  Identificare varianti genetiche associate a malattie e tumori.

- **Evoluzione**
  Tracciare le relazioni tra specie e l'evoluzione del DNA.

- **Biotecnologia e medicina personalizzata**
  Sviluppare trattamenti su misura per individui.



Pattern Matching

Hashing del genoma

1

Questo approccio si basa sulla creazione di un indice dell'intero genoma.
Tuttavia, la costruzione e la memorizzazione di un indice richiedono **risorse di memoria significative**.
Inoltre, **strategie iterative possono subire un rallentamento** in presenza di errori di sequenziamento, che complicano il riconoscimento esatto dei pattern.

- SOAPv1, MOM, BFAST...

Hashing delle letture

2

Metodi che utilizzano l'hashing delle **letture stesse** anziché indicizzare l'intero genoma. Questi approcci **riducono il consumo di memoria** ma richiedono spesso la **scansione completa del genoma** di riferimento.
Di conseguenza, l'identificazione dei pattern può risultare più lenta.

- RMap,MAQ, ZOOM...

Ordinamento delle sequenze

3

Si basano sull'**ordinamento delle sottosequenze** del genoma di riferimento e delle letture.
Questo approccio **evita del tutto l'hashing** e offre vantaggi specifici in applicazioni dove l'ordinamento migliora la gestione delle sequenze. È particolarmente utile per il confronto di pattern su larga scala.

- Slider, SOAPv2,Bowtie,BWT,BWA...

Brute Force per il Pattern Matching

Confronto sequenziale di un pattern lungo tutte le posizioni del genoma.

Si esamina ogni posizione possibile della sequenza genomica e si verifica, carattere per carattere, se il pattern corrisponde esattamente alla sottosequenza in quella posizione.

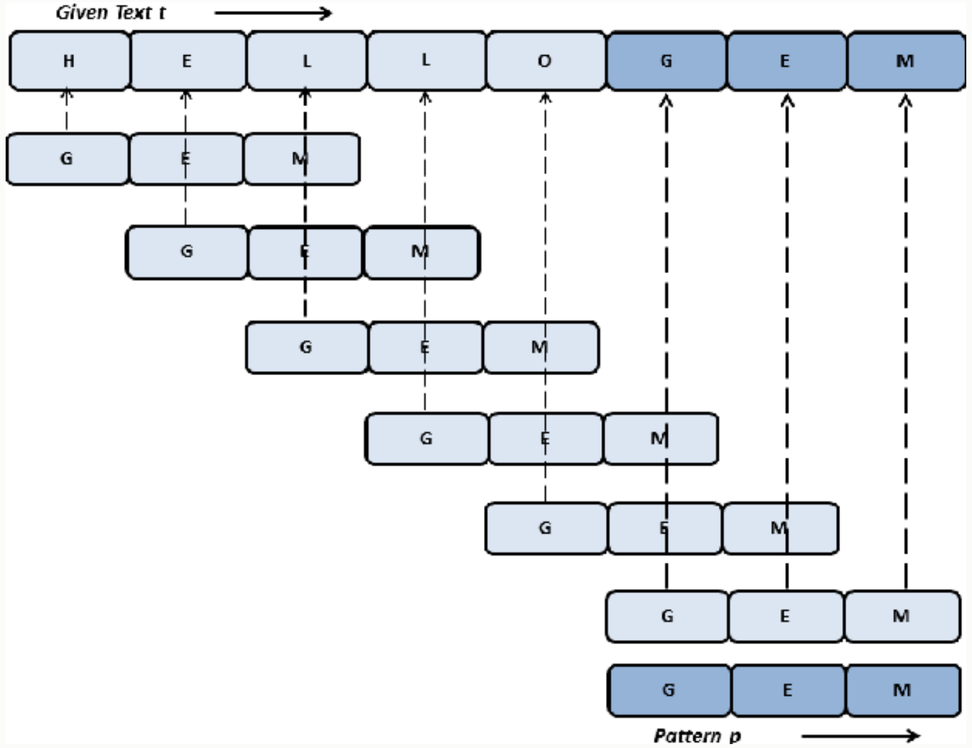
Svantaggi

- Inefficiente per genomi lunghi
- Richiede elevati tempi di calcolo e risorse computazionali.

Complessità computazionale

$O(m \cdot n)$

(m = lunghezza del genoma, n = lunghezza dei pattern)



Suffix tree per il Pattern Matching

Struttura ad albero che rappresenta tutti i suffissi di una sequenza genomica.

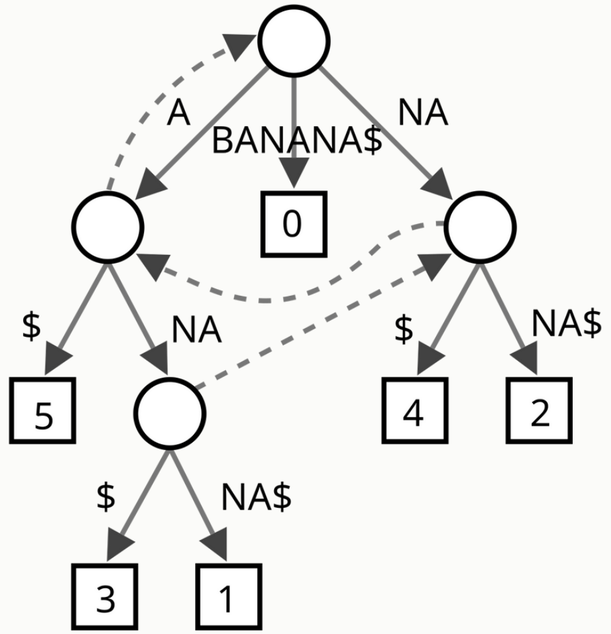
Permette di eseguire ricerche rapide evitando confronti ripetuti, richiede solo il percorso lungo i nodi della struttura, il che risulta essere pari a O(m) con m pari alla lunghezza del pattern.

Costruzione	Ricerca dei pattern
O (n)	O (m)

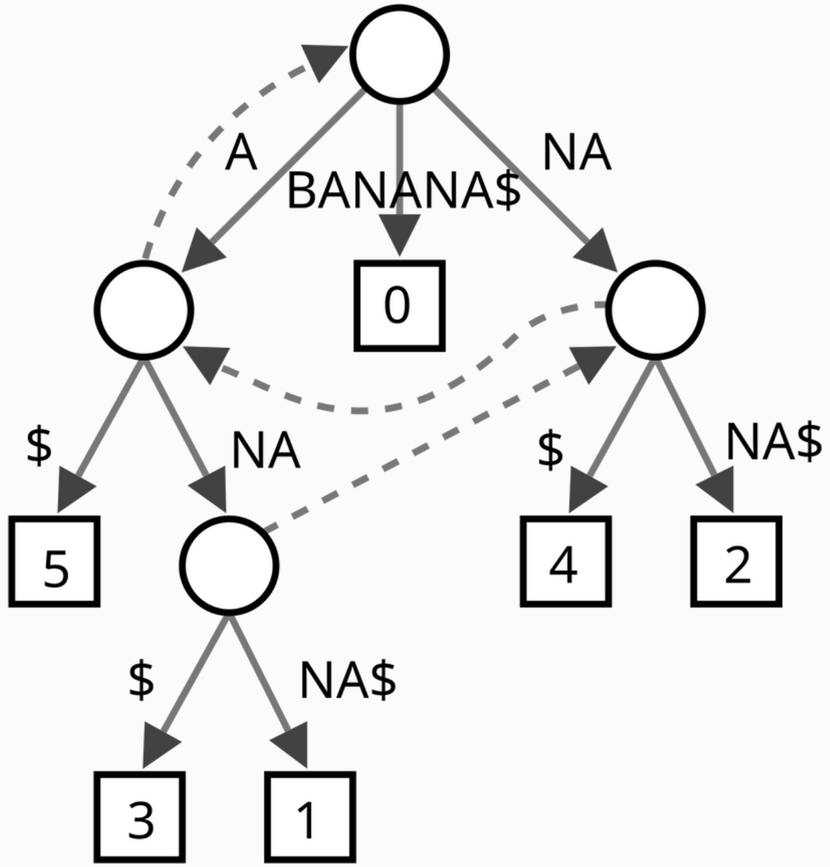
(n = lunghezza del genoma, m = lunghezza dei pattern)

Vantaggi

- Matching simultaneo per più pattern.
- Memorizzazione delle posizioni dei suffissi tramite simbolo speciale "$"



Suffix tree per il Pattern Matching



Costruzione	Ricerca dei pattern
O (n)	O (m)

(n = lunghezza del genoma, m = lunghezza dei pattern)

Svantaggi

- Elevato consumo di memoria (fino a 20 volte la lunghezza del genoma).
- Necessità di costruire completamente l'albero prima della compressione.

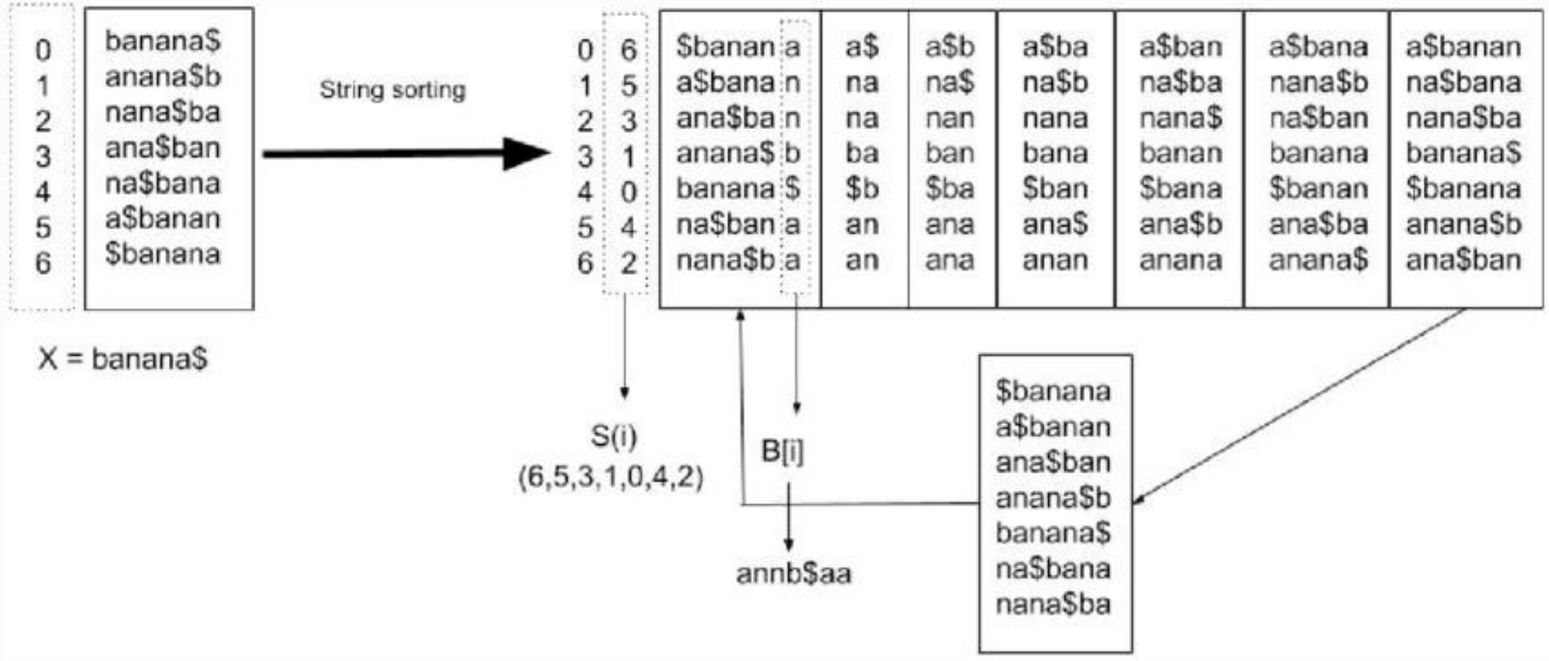
Burrows-Wheeler Transform (BWT)

Trasformazione che comprime una stringa S.

Riordina le **rotazioni cicliche** della stringa in ordine lessicografico e preleva i caratteri dell'ultima colonna della matrice ordinata.

Limiti

- Costruzione della matrice BWT: memoria e tempo quadratici.



Last-To-First mapping

Risalire alla stringa originale a partire dalla Burrows-Wheeler Transform (BWT).

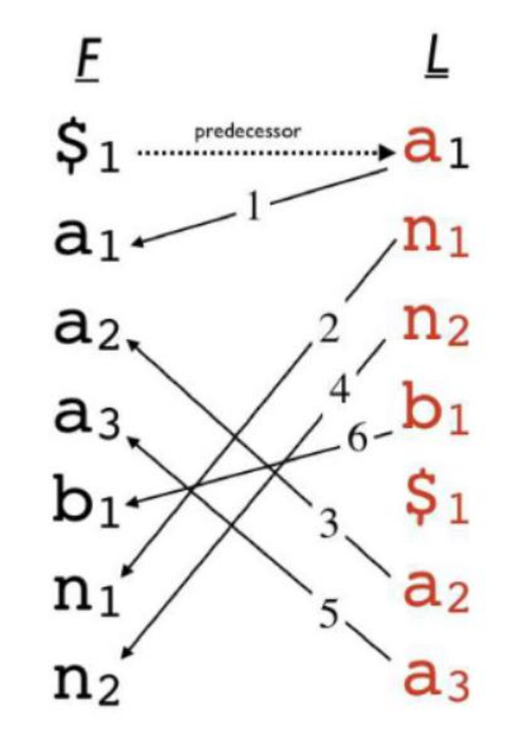
La prima occorrenza di un carattere nella prima colonna corrisponde alla sua prima occorrenza nella BWT. Questa relazione permette di ricostruire la stringa originale seguendo i caratteri passo per passo.

Vantaggi

- Compressione efficiente per sequenze con ripetizioni locali
- Riduzione dello spazio richiesto per memorizzare i dati.

Limiti

- Costruzione della matrice BWT: memoria e tempo quadratici.



FM-Index

Struttura dati progettata per l'allineamento efficiente di short reads a un genoma di riferimento.

Si basa sulla Trasformata di Burrows-Wheeler (BWT) riduce l’uso di memoria evitando la costruzione esplicita dell’intera matrice di rotazioni.

Vantaggi

- Occupa una quantità di memoria proporzionale alla compressione ottenuta dalla BWT

Index	0	1	2	3	4	5	6
X	b	a	n	a	n	a	$
SA	6	5	3	1	0	4	2
B(*X*)	a	n	n	b	$	a	a

C array:

$	a	b	n
0	1	4	5

Occ matrix:

	$	a	b	n
0	0	1	0	0
1	0	1	0	1
2	0	1	0	2
3	0	1	1	2
4	1	1	1	2
5	1	2	1	2
6	1	3	1	2

FM-Index

Index	0	1	2	3	4	5	6
X	b	a	n	a	n	a	$
SA	6	5	3	1	0	4	2
B(X)	a	n	n	b	$	a	a

C array:

$	a	b	n
0	1	4	5

Occ matrix:

	$	a	b	n
0	0	1	0	0
1	0	1	0	1
2	0	1	0	2
3	0	1	1	2
4	1	1	1	2
5	1	2	1	2
6	1	3	1	2

Suffix Array *(SA)*

Mappa la posizione originale dei suffissi ordinati nella matrice BWT, indicando dove ciascun suffisso inizia nel testo originale.

Occurrence Table *(OCC)*

Registra il conteggio cumulativo dei caratteri dell'alfabeto in un prefisso della BWT, essenziale per la ricerca backward.

Count Table *(C)*

Fornisce la posizione iniziale, nella matrice BWT, di suffissi che iniziano con ciascun simbolo dell'alfabeto, ordinati lessicograficamente.

Ricerca di una stringa di query W con FM-index

1

Identificazione dell'intervallo iniziale

Ricerca iniziale di un intervallo nella matrice BWT in cui possono trovarsi i suffissi che iniziano con l'ultimo carattere di W.

2

Raffinamento progressivo dell'intervallo

Ad ogni passo, si considerano i caratteri precedenti della query. Le tabelle C e OCC vengono utilizzate per restringere l'intervallo fino a rappresentare esclusivamente le posizioni dei suffissi corrispondenti alla query completa.

Identificazione dell'intervallo iniziale

- **Inizio dalla fine della query:**

La ricerca inizia considerando l'ultimo carattere della stringa di query, (W[|W|-1])

- **Utilizzo della tabella (C):**

La tabella (C) fornisce la posizione iniziale nella matrice BWT dei suffissi che iniziano con il carattere (W[|W|-1]).

L'intervallo iniziale ([k, l]) viene definito come:

$$k = C[W[|W|-1]] + 1$$
$$l = C[W[|W|-1] + 1]$$

dove $W\,[|W\,|-1] + 1$ rappresenta
il carattere successivo a $W\,[|W\,|-1]$ nell'ordine lessicografico.

Raffinamento progressivo dell'intervallo

- **Iterazione sui caratteri della query:**

  Il processo di raffinamento procede iterativamente, analizzando i caratteri di (W) dalla fine all'inizio, da (W[|W|-2]) fino a (W[0]).

- **Utilizzo della tabella (OCC):**

  Per ogni carattere (W[i]), si utilizza la tabella OCC per aggiornare l'intervallo ([k, l]) in modo che includa solo i suffissi che iniziano con la sottostringa (W[i, |W|-1]).

  L'aggiornamento avviene in questo modo:

$$k = C[W[i]] + OCC[k - 1][W[i]] + 1$$
$$l = C[W[i]] + OCC[l][W[i]]$$

- **Stretto intervallo finale:**

  Ad ogni iterazione, l'intervallo ([k, l]) si restringe, escludendo i suffissi che non corrispondono alla sottostringa corrente di (W).

Esempio

testo X = banana$

query Y = ana

matrice BWT di X = annb$aa

	F		*L*
1	$	BANAN	A
2	A	$BANA	N
3	A	NA$BA	N
4	A	NANA$	B
5	B	ANANA	$
6	N	A$BAN	A
7	N	ANA$B	A

Tabella *(C):*

c	*$*	*A*	*B*	*N*
C[c]	0	1	4	5

Tabella *(OCC)*:

i	*1*	*2*	*3*	*4*	*5*	*6*	*7*
OCC[’$’]	0	0	0	0	1	1	1
OCC[’a’]	1	1	1	1	1	2	3
OCC[’b’]	0	0	0	1	1	1	1
OCC[’n’]	0	1	2	2	2	2	2

Esempio

testo X = banana$ query Y = ana matrice BWT di X = annb$aa

Ricerca Iniziale:

W[|W|-1] = 'a'

k = C['a'] + 1 = 2

l = C['b'] = 4

Intervallo iniziale: [k, l] = [2, 4]

Raffinamento:

Iterazione 1 (W = 'n'):

k = C['n'] + OCC[k - 1]['n'] + 1 = 5 + 0 + 1 = 6

l = C['n'] + OCC[l]['n'] = 5 + 2 = 7

Nuovo intervallo: [k, l] = [6, 7]

Iterazione 2 (W = 'a'):

k = C['a'] + OCC[k - 1]['a'] + 1 = 1 + 1 + 1 = 3

l = C['a'] + OCC[l]['a'] = 1 + 3 = 4

Intervallo finale: [k, l] = [3, 4]

Questo intervallo rappresenta le posizioni nella BWT che terminano con "ana".

	F		*L*
1	$	BANAN	A
2	A	$BANA	N
3	A	NA$BA	N
4	A	NANA$	B
5	B	ANANA	$
6	N	A$BAN	A
7	N	ANA$B	A

Allineamento Esatto vs. Innesatto:

- L'allineamento **esatto** richiede una corrispondenza perfetta tra due sequenze.
- L'allineamento **inesatto** cerca allineamenti ottimali anche in presenza di differenze (mismatch o gap), cruciali in bioinformatica per sequenze di DNA o RNA soggette a variazioni.

La **Burrows-Wheeler Transform (BWT)** e la **backward search** permettono un'esplorazione efficiente dello spazio di ricerca per le possibili corrispondenze tra una sequenza di query e una di riferimento.

Algoritmo BWA

BWA (Burrows-Wheeler Alignment tool) utilizza la backward search per campionare sottostringhe distinte dal genoma, mirando a trovare l'allineamento ottimale con un numero limitato di differenze.

FM -Index & BWT

Heng Li, Richard Durbin.
“Fast and accurate short read alignment with Burrows-Wheeler transform”
Bioinformatics. Volume 25, Issue 14. 2009 July 15

Algoritmo per la ricerca inesatta

Componente chiave dell’algoritmo bwa è l’algoritmo per la ricerca inesatta degli intervalli di array di suffissi delle sottostringhe che corrispondono a una stringa di query

Array D(·)

Memorizza il limite inferiore del numero di differenze (mismatch o gap) presenti in un prefisso della sequenza di query.
Questo limite aiuta a ridurre lo spazio di ricerca durante la backward search, evitando percorsi non promettenti

Precalcolo

Calcola la stringa BWT ‘B’ per la stringa di riferimento X.
Calcola gli array C(·) e OCC(·, ·) da B.
Calcola la stringa BWT B' per il riferimento inverso.
Calcola l'array OCC’(·, ·) da B'.

Procedura

```
InexactSearch(W, z)
 CalculateD(W)
 return InexRecur(W, |W|-1, z, 1, |X|-1)

CalculateD(W)
 k <- 1
 l <- |X|-1
 z <- 0
 for i = 0 to |W|-1 do
  k <- C(W[i]) + OCC'(W[i], k-1) + 1
  l <- C(W[i]) + OCC'(W[i], l)
  if k > l then
   k <- 1
   l <- |X|-1
   z <- z + 1
  D(i) <- z
```

```
IInexRecur(W, i, z, k, l)
 if z < D(i) then
  return 0
 if i < 0 then
  return [k, l]
 l <- 0
 *l <- l  ∪ InexRecur(W, i-1, z-1, k, l)
 for each b ∈ {A, C, G, T} do
  k <- C(b) + OCC(b, k-1) + 1
  l <- C(b) + OCC(b, l)
  if k ≤ l then
   **l <- l ∪ InexRecur(W, i, z-1, k, l)
   if b = W[i] then
    l <- l ∪ InexRecur(W, i-1, z, k, l)
   else
    l <- l ∪ InexRecur(W, i-1, z-1, k, l)
 return l
```

Analisi dell’algoritmo 1/4

Precalcolo

Calcola la stringa BWT ‘B’ per la stringa di riferimento X.
Calcola gli array C(·) e OCC(·, ·) da B.
Calcola la stringa BWT B' per il riferimento inverso.
Calcola l'array OCC’(·, ·) da B'.

Prima di iniziare la ricerca, vengono precalcolate alcune strutture dati essenziali:

- la stringa BWT B e l'array di occorrenze OCC per la sequenza di riferimento X e l’array C.
- la stringa BWT B’ per la sequenza di riferimento inverso e l'array di occorrenze OCC’ per quest'ultima.

  Grazie alla stringa inversa è possibile verificare rapidamente se una sottostringa della query è presente anche nella sequenza di riferimento.

Analisi dell’algoritmo 2/4

Procedura

```
InexactSearch(W, z)
    CalculateD(W)
    return InexRecur(W, |W|-1, z, 1, |X|-1)
```

- **Questa procedura avvia la ricerca inesatta.** Prende in input la sequenza di query W e il numero massimo di differenze consentite z. Inizia calcolando l'array D con la procedura *CalculateD*, e successivamente richiama la procedura ricorsiva *InexRecur* per trovare gli intervalli nel Suffix Array delle corrispondenze.

Analisi dell’algoritmo 3/4

Procedura

```
CalculateD(W)
    k <- 1
    l <- |X|-1
    z <- 0
    for i = 0 to |W|-1 do
        k <- C(W[i]) + OCC'(W[i], k-1) + 1
        l <- C(W[i]) + OCC'(W[i], l)
        if k > l then
            k <- 1
            l <- |X|-1
            z <- z + 1
        D(i) <- z
```

- **Questa procedura calcola l'array D(·) che limita la ricerca.**

L'array D memorizza per ogni posizione i nella sequenza di query il limite inferiore del numero di differenze presenti nel prefisso W[0,i]. La procedura utilizza la stringa BWT B' e l'array OCC' del riferimento inverso per verificare se una sottostringa di W è presente anche nel riferimento. Se una sottostringa non è presente, il valore di z viene incrementato e il valore corrispondente in D viene aggiornato.

Analisi dell’algoritmo 4/4

Procedura

```
InexRecur(W, i, z, k, l)
    if z < D(i) then
            return 0
    if i < 0 then
            return [k, l]
    l <- 0
 *  l <- l  ∪ InexRecur(W, i-1, z-1, k, l)
    for each b ∈ {A, C, G, T} do
            k <- C(b) + OCC(b, k-1) + 1
            l <- C(b) + OCC(b, l)
            if k ≤ l then
**          l <- l ∪ InexRecur(W, i, z-1, k, l)
            if b = W[i] then
                l <- l ∪ InexRecur(W, i-1, z, k, l)
            else
                l <- l ∪ InexRecur(W, i-1, z-1, k, l)
     return l
```

- **Questa procedura ricorsiva esplora lo spazio di ricerca.**

I parametri i,z,k e l rappresentano rispettivamente la posizione corrente nella sequenza di query, il numero di differenze accumulate, e l'intervallo SA corrente.

- Se il numero di differenze accumulate z è minore del limite inferiore D(i), la **ricerca in quel ramo viene interrotta** (non può portare a un allineamento valido).
- Se i < 0, significa che **si è arrivati alla fine della sequenza** di query e l'intervallo nel Suffix Array corrente rappresenta una corrispondenza.
- Le righe marcate con * e ** gestiscono rispettivamente le inserzioni e le delezioni nella sequenza di riferimento.
- Per ogni possibile base b, la procedura calcola il nuovo intervallo nel Suffix Array e richiama ricorsivamente se stessa per esplorare il nuovo ramo.

Ottimizzazioni pratiche dell’algoritmo BWA

Penalità differenziale:

BWA assegna **penalità diverse** a mismatch e gap rendendo l'allineamento più biologicamente **realistico**.

Struttura dati a heap e ricerca breadth-first

BWA utilizza una struttura dati simile a un heap per **prioritizzare i migliori allinementi parziali**, elaborando contemporaneamente la sequenza inversa complementare con un approccio **breadth-first** (BFS) invece della depth-first (DFS) simulata dalla ricorsione.

Ottimizzazioni pratiche dell’algoritmo BWA

Strategia iterativa:

BWA accelera **interrompendo la ricerca per intervalli subottimali** se l'intervallo migliore è ripetitivo, o limitandola a (z+1) differenze se è unico (ossia si allinea solo in una posizione specifica). Tuttavia, la velocità dipende dal tasso di mismatch, poiché allineamenti con più differenze richiedono più tempo.

Seed sequence: *

BWA **limita le differenze** nelle seed sequence iniziali, accelerando l’allineamento fino a 2,5 volte con un impatto minimo sull’errore (dallo 0,08% allo 0,11%). Il seeding è meno efficace per letture brevi.

* Una **seed sequence** è un frammento della lettura che viene selezionato e utilizzato per identificare posizioni candidate sul genoma di riferimento.

Gestione della memoria in BWA

BWA adotta strategie ottimizzate che consentono di ridurre lo spazio necessario senza compromettere l'efficienza.
Queste strategie si applicano alle due principali strutture dati utilizzate: l'array delle occorrenze OCC e l'array dei suffissi SA.

Array OCC

Problema

- Memorizza il conteggio cumulativo dei caratteri per ogni posizione del genoma, e richiederebbe $4n\log_2 n$ bit per un genoma di lunghezza n.

Soluzione

- Memorizzare solo i valori O(·,k) per k multiplo di 128.
- I valori intermedi vengono calcolati al volo utilizzando la stringa BWT.

Array SA

Problema

- Memorizzare l'intero Suffix Array richiederebbe $n\log_2 n$ bit, impraticabile per genomi di grandi dimensioni.

Soluzione

- Salvare solo i valori SA(k) per k divisibili per 32.
- Gli altri valori vengono calcolati al momento usando il CSA inverso $\Psi-1$:

BWA supporta:

- L’allineamento con gap per **letture**. **single-end**
- La **mappatura paired-end**, con calcolo della qualità degli allineamenti.
- La generazione di **più possibili allinementi** se richiesto.
- L’output predefinito in **formato SAM** (Sequence Alignment/Map), compatibile con strumenti come SAMtools per analisi successive.

Confronto con altri strumenti

Gli autori hanno confrontato le prestazioni di BWA con altri tre programmi di allineamento: MAQ, SOAPv2 e Bowtie, utilizzando sia dati simulati sia dati reali.

Dati simulati dal genoma umano

Accuratezza

BWA e MAQ mostrano risultati **simili**, con BWA leggermente **più preciso** rispetto a Bowtie e SOAPv2.

Velocità

SOAPv2 si dimostra il **più veloce**, seguito da BWA, che è significativamente **più rapido** rispetto a MAQ.

Dati reali di sequenziamento Illumina

Prestazioni complessive

BWA e MAQ raggiungono un'**elevata percentuale di allineamenti corretti**, mantenendo coerenza con le informazioni di pairing.

Efficienza temporale

BWA è **più veloce** di MAQ, mentre SOAPv2, pur risultando il più veloce, mostra una leggera **diminuzione nella qualità delle mappature**.

Test su genoma ibrido uomo-pollo

Precisione

BWA si è dimostrato **altamente preciso**, mappando solo una minima frazione di reads al genoma errato.

Conclusioni

- **BWA è un ordine di grandezza più veloce di MAQ**
- **Supporta l'allineamento gapped per letture single-end,**

  Importante quando le letture diventano più lunghe e tendono a contenere indel
- **BWA emette l'allineamento nel formato SAM**

  Sfruttare i vantaggi delle analisi downstream implementate in SAMtools.
- **Prestazioni degradate su letture lunghe**

  BWA richiede sempre che la lettura completa sia allineata, dalla prima base all'ultima
  - Soluzione: Dividere la lettura in più frammenti corti, allineare i frammenti separatamente e quindi unire gli allineamenti parziali per ottenere l'allineamento completo della lettura.

CLELIA DE FELICE - ROCCO ZACCAGNINO - ROSALBA ZIZZA

STRUMENTI FORMALI PER LA BIOINFORMATICA
2024/2025

GRAZIE PER L'ATTENZIONE

Università Degli Studi Di Salerno

Gaita Irene - 0522501839