Università degli Studi di Salerno

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Corso di Strumenti Formali per la Bioinformatica

Analisi di BFCounter per il conteggio efficiente dei k-mer nel DNA tramite Bloom Filter

Docenti
Prof.ssa De Felice Clelia
Prof. Zaccagnino Rocco
Prof.ssa Zizza Rosalba

Candidata
Fortino Rosalia
Matricola: 0522502023

Anno Accademico 2024/2025

Indice

Abstract

Le tecnologie di sequenziamento di nuova generazione hanno rivoluzionato la genomica, generando dataset di grandi dimensioni che richiedono strumenti efficienti per la loro analisi, rendendo essenziale lo sviluppo di strumenti computazionali efficienti per la loro analisi.

Il conteggio dei k-mer, sottosequenze di lunghezza fissa, rappresenta un passaggio fondamentale in applicazioni come l'assemblaggio genomico, la metagenomica e l'identificazione di varianti genetiche. Tuttavia, il trattamento di dataset genomici su larga scala richiede soluzioni efficienti in termini di memoria e velocità di elaborazione.

Questo lavoro si concentra sull'uso dei Bloom Filter, strutture dati probabilistiche, come strumento per rappresentare in modo compatto i k-mer e ridurre significativamente il consumo di risorse computazionali. In particolare, viene analizzato BFCounter, un tool che integra i Bloom Filter per eliminare i k-mer unici e migliorare l'efficienza del conteggio, e viene effettuato un confronto con Jellyfish, una soluzione tradizionale basata su tabelle hash.

L'analisi comparativa, supportata da test sperimentali, evidenzia che BFCounter utilizza significativamente meno memoria, nonostante la tolleranza a una bassa percentuale di falsi positivi intrinseca ai Bloom Filter. Jellyfish, al contrario, si distingue per una maggiore velocità di calcolo e risultati deterministici, richiedendo però un dispendio più elevato di risorse computazionali.

Questo studio dimostra come i Bloom Filter possano rappresentare una soluzione efficace per migliorare l'efficienza delle analisi genomiche, offrendo spunti per future ottimizzazioni.

Capitolo 1

Introduzione

1.1 Contesto

Il progresso delle tecnologie di sequenziamento di nuova generazione (Next-Generation Sequencing) ha rivoluzionato il campo della genomica, consentendo l'acquisizione di dataset genetici di vaste dimensioni. Questi dataset hanno evidenziato sfide computazionali complesse nella gestione e nell'analisi delle informazioni genetiche.

In questo contesto, l'analisi genomica richiede tecniche avanzate in grado di gestire grandi volumi di dati con precisione ed efficienza. Un metodo fondamentale è il conteggio dei k-mer, ossia sottosequenze di lunghezza fissa k, che svolgono un ruolo cruciale in diverse applicazioni bioinformatiche.

L'elaborazione dei k-mer su larga scala presenta importanti sfide: nei dataset genomici moderni, i k-mer univoci possono raggiungere miliardi di unità, rendendo le strutture dati tradizionali inefficaci. I Bloom Filter emergono come soluzione innovativa, offrendo un metodo probabilistico compatto per rappresentare insiemi genomici, nonostante alcune limitazioni.

1.2 Obiettivi

Questo lavoro mira ad approfondire il ruolo e l'efficacia dei Bloom Filter nell'analisi genomica, con un focus specifico sul loro impiego per il conteggio dei k-mer. Il contesto della bioinformatica moderna richiede soluzioni che riescano ad unire precisione ed efficienza computazionale, soprattutto in presenza di dataset sempre più ampi e complessi. In quest'ottica, la tesina ha l'obiettivo di valutare come queste strutture dati probabilistiche possano rappresentare una risposta concreta alle sfide legate alla gestione di quantità enormi di dati genomici, e di analizzare i loro limiti intrinseci per delineare possibili miglioramenti futuri.

1.2.1 Principi e limiti dei Bloom Filter

Uno degli obiettivi principali di questo lavoro è esplorare i principi alla base dei Bloom Filter, strutture dati che permettono di rappresentare insiemi in modo estremamente compatto. Questi strumenti offrono soluzioni efficienti in termini di memoria, risultando particolarmente utili per analisi che richiedono di gestire dataset molto grandi, come per il conteggio dei

k-mer. Tuttavia, i Bloom Filter non sono privi di limiti: la loro natura probabilistica comporta la possibilità di falsi positivi e l'impossibilità di modificare dinamicamente il contenuto. Comprendere questi limiti è importante per valutare la loro applicabilità in contesti pratici e trovare strategie per migliorarne l'efficacia.

1.2.2 Confronto tra BFCounter e Jellyfish

L'aspetto centrale di questo lavoro è l'analisi comparativa di due strumenti utilizzati per il conteggio dei k-mer: BFCounter e Jellyfish. Questi due tool implementano approcci distinti per affrontare il problema, ma entrambi usano soluzioni innovative per garantire un compromesso ottimale tra velocità di calcolo e consumo di risorse. Attraverso l'analisi del loro funzionamento, e la replica di test descritti in letteratura su dataset reali, sarà possibile verificare le loro prestazioni. I dataset utilizzati rappresentano esempi significativi per comprendere come la scelta dei parametri, come la lunghezza dei k-mer, possa influenzare la precisione e la velocità delle analisi. L'obiettivo finale è fornire una visione completa e critica delle differenze tra i due strumenti, considerando aspetti come il consumo di memoria, i tempi di calcolo e la scalabilità a dataset sempre più grandi.

1.3 Analisi dei k-mer

L'analisi dei k-mer rappresenta un approccio fondamentale nelle ricerche genomiche moderne, offrendo un metodo estremamente versatile per l'esplorazione delle sequenze genetiche. Questa tecnica consente di decomporre le sequenze DNA in sottostringhe di lunghezza definita, permettendo di estrarre informazioni statistiche e strutturali per la comprensione dei genomi.

La sua importanza risiede nella capacità di fornire una rappresentazione sintetica ma informativa delle sequenze genetiche, superando i limiti delle analisi tradizionali basate sull'allineamento completo. Attraverso il conteggio e l'analisi delle frequenze dei k-mer, è possibile identificare pattern ricorrenti, caratterizzare variazioni genetiche e studiare complesse strutture genomiche con un approccio computazionalmente efficiente.

Le applicazioni includono l'assemblaggio genomico, la metagenomica, l'identificazione di varianti genetiche e studi evolutivi, dimostrando la versatilità e il potenziale di questa metodologia nelle scienze genomiche contemporanee.

1.4 Introduzione ai Bloom Filter

I Bloom Filter, introdotti da Burton Bloom nel 1970[1], sono strutture dati probabilistiche progettate per verificare l'appartenenza a insiemi in modo rapido ed efficiente. Basati su un array di bit e su funzioni hash, consentono di rappresentare insiemi di dati in modo estremamente compatto, riducendo significativamente il consumo di memoria rispetto alle strutture tradizionali. Questa efficienza si unisce ad una velocità di calcolo che non dipende dalla dimensione del dataset, rendendoli particolarmente adatti a gestire grandi quantità di dati.

Nonostante i vantaggi, i Bloom Filter presentano alcune limitazioni intrinseche alla loro natura probabilistica. Tra queste, la possibilità di falsi positivi, ovvero risultati che indicano erroneamente la presenza di un elemento, e l'impossibilità di rimuovere elementi una volta inseriti. Questi compromessi, tuttavia, sono accettati in molti contesti applicativi, dove la rapidità e il risparmio di memoria sono requisiti prioritari.

La loro versatilità ha favorito un uso diffuso in molti ambiti, tra cui le reti informatiche, il machine learning, la sicurezza informatica e l'analisi genomica. In questi settori, i Bloom Filter offrono soluzioni efficienti per problemi complessi, dimostrandosi una delle strutture dati più innovative nel trattamento di grandi insiemi di informazioni.

1.5 Struttura della tesina

- Introduzione: Questo capitolo introduce il tema della tesina, definendo gli obiettivi del lavoro e contestualizzando il problema del conteggio efficiente dei k-mer nelle sequenze genomiche. Viene fornita una panoramica sull'importanza dei Bloom Filter in bioinformatica e una breve descrizione della struttura della tesina;

- Background: Il secondo capitolo esamina la letteratura esistente sul conteggio dei k-mer, con particolare attenzione all'utilizzo dei Bloom Filter. Viene presentata una rassegna degli approcci e delle tecnologie disponibili, analizzando il funzionamento e le applicazioni di strumenti rilevanti;

- Analisi dei tool: Questo capitolo si focalizza sul confronto di due diversi strumenti che utilizzano i Bloom Filter per indicizzare i k-mer. Vengono discussi criteri di confronto come accuratezza, efficienza e utilizzo delle risorse, includendo una descrizione dettagliata delle loro caratteristiche;

- Replica dei test: In questo capitolo si illustrano i test sperimentali condotti per replicare i risultati riportati nella letteratura. Viene descritta la metodologia adottata, insieme ai dati utilizzati, e vengono analizzati i risultati ottenuti per confrontarli con quelli originali;

- Conclusione: L'ultimo capitolo sintetizza i risultati ottenuti e discute il contributo della ricerca.

Capitolo 2

Background

2.1 Definizione di k-mer

Nel campo della bioinformatica e nell'analisi delle sequenze genomiche, i k-mer rappresentano un concetto fondamentale che ha rivoluzionato l'approccio all'analisi delle informazioni genetiche. I k-mer sono definiti come tutte le possibili sottosequenze di lunghezza k che possono essere estratte da una sequenza più lunga di nucleotidi o amminoacidi.

Più formalmente, consideriamo un alfabeto Σ che, nel caso specifico del DNA, è costituito da quattro nucleotidi: $\Sigma = \{A, C, G, T\}$. Data una sequenza S di lunghezza n, definiamo un k-mer come una sottostringa continua di S di lunghezza k, dove $k \leq n$. Questa definizione matematica ci permette di quantificare precisamente il numero di k-mer che possono essere estratti da una sequenza data.

La relazione matematica che determina il numero di k-mer sovrapposti che possono essere estratti da una sequenza di lunghezza n è espressa dalla formula:

$$N_{k\text{-}mer} = n - k + 1 \tag{2.1}$$

Questa equazione fondamentale si basa sull'osservazione che possiamo iniziare l'estrazione di un k-mer da qualsiasi posizione della sequenza, partendo dalla posizione 1 fino alla posizione $n - k + 1$, garantendo sempre che ci sia spazio sufficiente nella sequenza per estrarre una sottosequenza completa di lunghezza k.

La complessità dello spazio dei k-mer rappresenta un aspetto particolarmente interessante della loro natura matematica. Il numero totale di possibili k-mer distinti che possono essere generati per un dato valore di k segue una crescita esponenziale descritta dalla formula:

$$N_{possibili} = |\Sigma|^k \tag{2.2}$$

dove $|\Sigma|$ indica la cardinalità dell'alfabeto. Nel caso specifico del DNA, dove $|\Sigma| = 4$, questo significa che esistono 4^k possibili k-mer distinti. Questa crescita esponenziale ha importanti conseguenze sia teoriche che pratiche nell'analisi genomica.[2]

Tabella 2.1: k-mer per GTAGAGCTGT

k	k-mer
1	G, T, A, C
2	GT, TA, AG, GA, AG, GC, CT, TG
3	GTA, TAG, AGA, GAG, AGC, GCT, CTG, TGT
4	GTAG, TAGA, AGAG, GAGC, AGCT, GCTG, CTGT
5	GTAGA, TAGAG, AGAGG, GAGCT, AGCTG, GCTGT
6	GTAGAG, TAGAGC, AGAGCT, GAGCTG, AGCTGT
7	GTAGAGC, TAGAGCT, AGAGCTG, GAGCTGT
8	GTAGAGCT, TAGAGCTG, AGAGCTGT
9	GTAGAGCTG, TAGAGCTGT
10	GTAGAGCTGT

2.2 Conteggio dei k-mer nelle analisi genomiche

Il conteggio dei k-mer è una delle operazioni fondamentali nell'analisi genomica moderna, e costituisce la base per numerose applicazioni bioinformatiche di grande rilevanza. Questa tecnica, che può sembrare semplice nella sua concezione ma risulta complessa nelle sue implementazioni, si è dimostrata essenziale per comprendere la struttura e la funzione dei genomi, oltre a contribuire allo sviluppo di strumenti analitici sempre più avanzati. L'importanza del conteggio dei k-mer deriva dalla sua capacità di fornire informazioni statistiche significative sulla composizione e l'organizzazione delle sequenze genomiche, consentendo di identificare pattern ricorrenti, varianti genetiche e caratteristiche strutturali specifiche.

2.2.1 Assemblaggio genomico

Nell'assemblaggio de novo dei genomi, il conteggio dei k-mer costituisce un passaggio preliminare fondamentale. La frequenza con cui specifici k-mer appaiono in un dataset di sequenziamento può fornire informazioni preziose sulla copertura del genoma e sulla presenza di regioni ripetute.[3] Questo tipo di analisi permette di identificare e correggere errori di sequenziamento, poiché k-mer che compaiono con frequenza molto bassa sono spesso indicativi di errori sperimentali.[4]

I k-mer sono alla base della costruzione dei grafi di De Bruijn, uno strumento computazionale fondamentale per rappresentare le sovrapposizioni tra frammenti genomici. In questi grafi, i nodi rappresentano le sequenze di lunghezza k-1, mentre gli archi mostrano le sovrapposizioni tra k-mer adiacenti. La costruzione e la semplificazione del grafo di De Bruijn sono fasi critiche per ottenere sequenze genomiche contigue complete e accurate.[5]

La scelta della lunghezza ottimale dei k-mer è un aspetto critico nell'assemblaggio genomico: k-mer troppo corti possono generare ambiguità nell'assemblaggio a causa della loro frequente ripetizione nel genoma, mentre k-mer troppo lunghi possono frammentare eccessivamente l'assemblaggio a causa della minore probabilità di trovare sovrapposizioni perfette

tra le sequenze.[6] L'analisi della distribuzione dei k-mer aiuta anche a identificare regioni genomiche con caratteristiche particolari, come elementi ripetitivi o regioni con composizione nucleotidica anomala, che potrebbero richiedere strategie di assemblaggio specifiche.[7]

2.2.2 Metagenomica

Le applicazioni del conteggio dei k-mer nella metagenomica hanno trasformato lo studio delle comunità microbiche. Questa tecnica permette di caratterizzare la composizione di ecosistemi microbici complessi senza la necessità di coltivare i singoli organismi in laboratorio, superando così uno dei principali limiti della microbiologia tradizionale. Attraverso l'analisi della frequenza e della distribuzione dei k-mer, è possibile identificare la presenza di specifiche specie batteriche all'interno di un campione ambientale, anche in assenza di un genoma di riferimento completo.[8]

Inoltre, l'analisi dei profili di k-mer consente la profilazione tassonomica e funzionale. La composizione della comunità microbica viene stimata confrontando i profili di k-mer del campione con database di riferimento, utilizzando metriche come la distanza di Bray-Curtis per quantificare le similarità.[9] Questo approccio consente di caratterizzare la diversità microbica, identificare specie sconosciute e monitorare variazioni ambientali.

Infine, i k-mer offrono la possibilità di prevedere la presenza di geni funzionali e pathway metabolici, fornendo informazioni preziose sul potenziale biochimico della comunità. Questo tipo di analisi è particolarmente rilevante in biotecnologia e medicina, dove può portare alla scoperta di nuovi enzimi e composti bioattivi.[10]

2.2.3 Identificazione di varianti genetiche

Il conteggio dei k-mer ha rivoluzionato il modo di identificare le varianti genetiche, offrendo una metodologia complementare alle tecniche tradizionali di allineamento. Questa tecnica si è dimostrata particolarmente utile per individuare varianti strutturali complesse, come inserzioni, delezioni e riarrangiamenti, che spesso non vengono catturate dai metodi basati sull'allineamento.[11]

L'approccio basato sui k-mer ha diversi vantaggi: non necessita di un genoma di riferimento completo, è meno sensibile agli errori di sequenziamento e può identificare varianti in regioni ripetitive del genoma.[12]

2.2.4 Studi evolutivi e filogenetici

L'analisi dei k-mer ha aperto nuove strade nello studio dell'evoluzione molecolare e della filogenesi. La composizione in k-mer di un genoma funge da impronta digitale molecolare, permettendo di ricostruire le relazioni evolutive tra specie anche molto divergenti, per le quali l'allineamento delle sequenze può risultare problematico.[13]

L'utilizzo dei k-mer consente di costruire alberi filogenetici robusti e di identificare dinamiche evolutive, come il trasferimento orizzontale di geni e l'evoluzione di famiglie geniche.[14] Questo approccio si è rivelato particolarmente efficace nello studio dei virus e dei batteri patogeni, fornendo informazioni chiave sulle loro origini evolutive.[15]

2.3 Tecniche di conteggio

Per far fronte alla complessità e alla scala dei dati generati dalle tecnologie di sequenziamento di nuova generazione, sono state sviluppate diverse tecniche per il conteggio dei k-mer, ognuna con specifici vantaggi e limitazioni in termini di tempo di esecuzione, memoria utilizzata e impiego dello spazio su disco. Gli strumenti per il conteggio k-mer possono essere categorizzati in base all'approccio e alle strutture dati che utilizzano, come mostrato nella Tabella 2.2.

Tabella 2.2: Ontologia degli approcci di conteggio k-mer

Approccio	Disco	Memoria
Hash table	Gerbil, MSPKmerCounter, DSK	Squeakr, Jellyfish, BFCounter
Sorting	KMC3, GenomeTexter, KMC2, KAnalyze, KMC1	Turtle
Burst tries	-	KCMBT
Enhanced suffix array	-	Tallymor

2.3.1 Approcci basati su tabelle hash, ordinamento e strutture avanzate

Gli approcci basati su tabelle hash si concentrano sull'efficienza nell'accesso e nell'aggiornamento dei dati. Strumenti come Jellyfish[16] utilizzano tabelle hash senza lock, consentendo aggiornamenti paralleli delle frequenze dei k-mer tramite istruzioni di compare-and-swap (CAS).[17] Questa tecnica riduce il rischio di colli di bottiglia nell'accesso concorrente ai dati, ma può risultare inefficace quando si trattano dataset di dimensioni molto elevate, in quanto il consumo di memoria aumenta proporzionalmente al numero di k-mer distinti da gestire. Inoltre, il rischio di collisioni nelle tabelle hash può introdurre ulteriori complessità nel processo.

Gli approcci basati sull'ordinamento, come quelli implementati da strumenti come KMC2[18] e GenomeTester4[19], sfruttano algoritmi di sorting per organizzare i k-mer in una sequenza ordinata. Dopo l'ordinamento, i k-mer ripetuti possono essere contati in modo lineare, poiché si trovano in posizioni contigue nella lista ordinata. Sebbene questa metodologia sia estremamente precisa e adatta per dataset di grandi dimensioni, il processo di ordinamento richiede operazioni significative di input/output (I/O), aumentando i tempi di elaborazione complessivi.

Le strutture dati avanzate, come i suffix array utilizzati in Tallymer[20] o i trie utilizzati da KCMBT[21], rappresentano alternative promettenti. Tallymer utilizza alberi di intervalli basati sul prefisso comune più lungo (lcp-interval tree), che consentono di contare i k-mer in modo implicito, riducendo la necessità di spazio aggiuntivo per la memorizzazione di conteggi intermedi.[22] Tuttavia, la costruzione di suffix array e l'implementazione di queste tecniche risultano computazionalmente costose, limitandone l'efficienza in presenza di dataset estremamente vasti.

2.3.2 Approcci basati su Bloom Filter

In questo contesto, i Bloom Filter emergono come soluzione particolarmente innovativa ed efficiente. Questi strumenti si basano su strutture dati probabilistiche che permettono di rappresentare insiemi di elementi con una quantità di memoria notevolmente ridotta. Nei Bloom Filter, i k-mer vengono trasformati in hash e memorizzati come bit in un vettore.[1] Una delle applicazioni più comuni di questa tecnica è la rimozione dei k-mer che appaiono una sola volta, che spesso rappresentano errori di sequenziamento e costituiscono la maggior parte dei dataset genomici. Ad esempio, strumenti come BFCounter utilizzano un Bloom Filter per eliminare i k-mer singoli prima di passare al conteggio delle frequenze tramite tabelle hash.

Nonostante il rischio di falsi positivi intrinseco nei Bloom Filter, l'accuratezza del conteggio può essere mantenuta a livelli accettabili ottimizzando la scelta delle funzioni hash e la dimensione del vettore di bit.[23] Rispetto ad altre tecniche, i Bloom Filter si distinguono per il loro equilibrio tra consumo di memoria e capacità di scalare su dataset di grandi dimensioni.

Gli approcci basati sui Bloom Filter sono particolarmente vantaggiosi per il pre-filtraggio dei dati. Ad esempio, cTurtle[24] utilizza un Bloom Filter per individuare e scartare i k-mer con frequenza pari a uno, riducendo significativamente la quantità di dati da processare nelle fasi successive. Questa ottimizzazione non solo riduce il tempo di calcolo, ma permette anche di risparmiare risorse hardware. Inoltre, grazie alla loro natura probabilistica, i Bloom Filter offrono una flessibilità intrinseca che li rende ideali per gestire dataset compressi, riducendo ulteriormente i costi I/O.[2]

2.4 Bloom Filter

I Bloom Filter rappresentano una struttura dati probabilistica ampiamente utilizzata in bioinformatica per risolvere problemi di appartenenza con un consumo di memoria ottimizzato. Sebbene offrano un'efficienza notevole, i Bloom Filter comportano un compromesso in termini di precisione, consentendo falsi positivi ma garantendo l'assenza di falsi negativi.

2.4.1 Struttura di base

Un Bloom Filter è implementato come un array di bit B di dimensione m, inizialmente popolato con valori pari a 0. Oltre all'array, vengono utilizzate k funzioni hash indipendenti, $\{h_1, h_2, \ldots, h_k\}$, ciascuna delle quali associa uniformemente un elemento di un insieme S a un valore compreso tra 0 e $m-1$.

Formalmente, dato un insieme $S = \{x_1, x_2, \ldots, x_n\}$ con $|S| = n$, la struttura di base del Bloom Filter può essere descritta come:

- Un array B di dimensione m, con $B[i] \in \{0,1\}$ per $i \in [0, m-1]$;
- Un insieme di k funzioni hash indipendenti $\{h_1, h_2, \ldots, h_k\}$, dove ogni $h_i : U \to [0, m-1]$, con U universo degli elementi.

Nell'esempio mostrato in Figura 2.1, il Bloom Filter utilizza tre funzioni hash rappresentate dalle frecce. I k-mer a e b sono stati inseriti nel Bloom Filter, quindi i bit corrispondenti

alle loro funzioni hash sono stati impostati a 1. Al contrario, i k-mer c e d non sono stati inseriti.

Per verificare se un elemento è presente nel Bloom Filter, basta controllare se tutti i bit corrispondenti alle sue funzioni hash sono impostati a 1. Se anche solo uno di questi bit è 0, allora significa che l'elemento non è presente nel Bloom Filter, come nel caso di c.

Tuttavia, può succedere che un elemento non inserito, come d, abbia comunque tutti i suoi bit impostati a 1 a causa delle collisioni delle funzioni hash di altri elementi inseriti. In questo caso, il Bloom Filter indicherà erroneamente che l'elemento è presente, generando un falso positivo.[25]

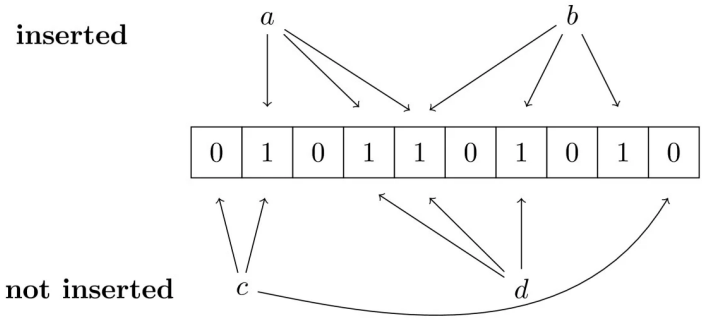

Figura 2.1: Struttura di un Bloom Filter

2.4.2 Algoritmi di inserimento e verifica

Gli algoritmi di base per i Bloom Filter sono l'inserimento e la verifica (o query).

Inserimento

L'inserimento di un elemento $x \in S$ in un Bloom Filter avviene calcolando i valori hash di x con le k funzioni hash e impostando a 1 i bit corrispondenti nell'array B.

L'algoritmo può essere descritto come segue:

Algoritmo 1 Inserimento in un Bloom Filter

```
1: procedure INSERT(x, B, {h_1, h_2, ..., h_k})
2:     for i ← 1 to k do
3:         index ← h_i(x)
4:         B[index] ← 1
```

Esempio: Supponiamo di voler inserire l'elemento $x =$ "$ACGT$" in un Bloom Filter con $m = 10$ e $k = 3$. Le funzioni hash restituiscono $h_1(x) = 2$, $h_2(x) = 6$ e $h_3(x) = 9$. Dopo l'inserimento, l'array B sarà aggiornato in questo modo: $B = [0, 0, 1, 0, 0, 0, 1, 0, 0, 1]$.

Verifica

La verifica di un elemento x consiste nel controllare se tutti i bit associati dai valori hash di x sono impostati a 1. Se almeno uno dei bit è 0, l'elemento non appartiene all'insieme. In caso contrario, l'elemento è considerato come appartenente, con una possibilità di falso positivo.

L'algoritmo di verifica è:

Algoritmo 2 Verifica in un Bloom Filter

```
1: function QUERY(x, B, {h_1, h_2, ..., h_k})
2:     for i ← 1 to k do
3:         index ← h_i(x)
4:         if B[index] ≠ 1 then return false
       return true
```

Esempio: Se $x =$ "$GCTA$" e i valori hash $h_1(x) = 3$, $h_2(x) = 5$, $h_3(x) = 8$ restituiscono tutti 1 nell'array B, la verifica restituisce `true`. Se uno qualsiasi dei valori restituisce 0, la verifica restituisce `false`.

2.4.3 Riduzione della memoria

Uno dei principali vantaggi dei Bloom Filter è la loro capacità di rappresentare insiemi di grandi dimensioni utilizzando una quantità ridotta di memoria. Questa efficienza è particolarmente importante in applicazioni come la bioinformatica, dove i dataset genomici possono contenere miliardi di k-mer. Rispetto a strutture dati tradizionali, come le tabelle hash o gli alberi bilanciati, i Bloom Filter offrono un compromesso vantaggioso tra precisione e spazio utilizzato.

La quantità di memoria necessaria per un Bloom Filter è determinata principalmente dalla dimensione m dell'array di bit e dal numero di funzioni hash k. La dimensione totale dello spazio è m bit, indipendentemente dal numero di elementi n inseriti. Con una scelta ottimale di k, la quantità di memoria richiesta può essere calcolata come:

$$m \approx -\frac{n \ln(P(fp))}{(\ln 2)^2}$$

2.4.4 Falsi positivi

Un altro aspetto fondamentale dei Bloom Filter è la gestione dei falsi positivi, un compromesso inevitabile che deriva dall'ottimizzazione della memoria. La probabilità $P(fp)$ di un falso positivo è calcolata come:

$$P(fp) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \tag{2.3}$$

dove:

- m è la dimensione dell'array;

- k è il numero di funzioni hash;
- n è il numero di elementi inseriti.

Per un dato m, la probabilità di falso positivo può essere minimizzata ottimizzando k. Il valore ottimale di k è:

$$k_{opt} = \frac{m}{n} \ln(2) \tag{2.4}$$

Per ridurre i falsi positivi senza aumentare significativamente l'uso della memoria, vengono utilizzate varianti dei Bloom Filter, che permettono la rimozione di elementi o che riducono la dimensione dell'array comprimendo i dati.

2.4.5 Impossibilità di rimuovere elementi

Un'altra limitazione dei Bloom Filter è l'impossibilità di rimuovere elementi già inseriti. Poiché un elemento è rappresentato da più bit impostati tramite funzioni hash, la rimozione di uno di essi potrebbe influire su altri elementi che condividono gli stessi valori hash. Questa caratteristica rende i Bloom Filter inadatti a insiemi dinamici dove è richiesto l'inserimento e la rimozione frequente di elementi.

Per superare questa limitazione, sono state sviluppate varianti come i Counting Bloom Filter, che sostituiscono l'array di bit con un array di contatori. Ogni contatore rappresenta il numero di elementi associati a una posizione hash, consentendo la rimozione decrementando il valore del contatore. Tuttavia, questo approccio aumenta il consumo di memoria e introduce complessità aggiuntive.

2.5 Applicazioni multidisciplinari dei Bloom Filter

2.5.1 Applicazioni in reti e sicurezza informatica

Nelle architetture di rete moderne, i Bloom Filter rivestono un ruolo fondamentale, soprattutto nei Software-Defined Network (SDN), dove facilitano l'instradamento efficiente dei pacchetti e riducono la complessità delle tabelle di routing. Sistemi distribuiti come Apache Cassandra e Google BigQuery utilizzano i Bloom Filter per ottimizzare query e implementare meccanismi di deduplicazione dei dati.[26]

Nel campo della sicurezza informatica, i Bloom Filter sono essenziali per sistemi di rilevamento intrusioni (IDS), grazie alla loro capacità di identificare rapidamente pacchetti sospetti con un overhead computazionale minimo. Allo stesso tempo, trovano applicazione nelle blockchain, dove supportano la verifica probabilistica delle transazioni e la sincronizzazione veloce dei nodi, migliorando l'efficienza complessiva delle reti distribuite.[27]

2.5.2 Machine learning e big data

L'ecosistema del machine learning sfrutta i Bloom Filter per tecniche avanzate di feature hashing, permettendo una rappresentazione compatta degli spazi vettoriali. I Bloom Filter

trovano ampio utilizzo anche nelle piattaforme di analisi dei Big Data, dove consentono la gestione efficiente di enormi quantità di informazioni, accelerando i tempi di elaborazione e riducendo il consumo di risorse.[28]

2.6 Utilizzo dei Bloom Filter in bioinformatica

I Bloom Filter sono strumenti fondamentali nel campo della bioinformatica, dove l'analisi di dati genetici su larga scala richiede approcci ottimizzati per velocità, efficienza e consumo di memoria. Grazie alla loro capacità di rappresentare insiemi in modo compatto e di effettuare verifiche di appartenenza con un consumo minimo di risorse, trovano numerose applicazioni in diverse fasi dell'analisi genomica.

2.6.1 Pre-processing dei dati di sequenziamento

Nel pre-processing dei dati, i Bloom Filter sono utilizzati per affrontare le sfide legate alla qualità e alla quantità dei dati generati dai sequenziatori moderni. Le loro principali applicazioni includono:

- Filtraggio di sequenze indesiderate: I Bloom Filter sono spesso impiegati per eliminare sequenze di bassa qualità, contaminanti o ridondanti, riducendo così la complessità dei dati da analizzare;[29]
- Rimozione di duplicati: In analisi come l'RNA-seq, i Bloom Filter possono identificare e rimuovere sequenze duplicate in modo rapido ed efficiente, migliorando l'accuratezza e l'affidabilità dei risultati successivi.[30]

Questi vantaggi derivano dalla capacità dei Bloom Filter di operare in tempo costante e di gestire dataset di dimensioni astronomiche senza compromettere le prestazioni.

2.6.2 Assemblaggio genomico e costruzione di grafi di De Bruijn

Un'applicazione chiave dei Bloom Filter è nell'assemblaggio genomico, in particolare nella costruzione dei grafi di De Bruijn. Questi grafi rappresentano i k-mer e le loro relazioni, elementi fondamentali per ricostruire il genoma a partire da sequenze di lettura frammentate.[3]

I Bloom Filter offrono due vantaggi fondamentali nell'assemblaggio genomico. In primo luogo, consentono di rappresentare i k-mer in modo estremamente compatto, portando a una drastica riduzione del consumo di memoria rispetto a strutture dati tradizionali come le tabelle hash.

In secondo luogo, svolgono un ruolo importante nella correzione degli errori di sequenziamento, poiché sono in grado di individuare e segnalare k-mer rari, che spesso sono associati a errori sperimentali, contribuendo così a migliorare la pulizia e l'affidabilità del grafo di De Bruijn.[4]

Questa combinazione di efficienza e semplicità ha reso i Bloom Filter uno strumento standard per affrontare le complessità dell'assemblaggio genomico.

2.6.3 Gestione efficiente dei k-mer nei dataset genomici

Nei dataset genomici di grandi dimensioni, i k-mer univoci possono raggiungere numeri astronomici, rendendo difficile l'uso di approcci tradizionali come le tabelle hash. I Bloom Filter offrono una soluzione scalabile, garantendo:

- Compressione dei dati: Rappresentando i k-mer in una struttura dati compatta, i Bloom Filter riducono significativamente il consumo di memoria, permettendo di analizzare dataset genomici di grandi dimensioni anche su hardware con risorse limitate;[18]
- Elaborazione rapida: La capacità di eseguire verifiche di appartenenza in tempo costante garantisce tempi di elaborazione rapidi anche su dataset complessi.[2]

2.6.4 Evoluzione e integrazione con altre strutture dati

L'efficacia dei Bloom Filter è ulteriormente potenziata attraverso l'integrazione di varianti più avanzate o altre strutture dati. Tra queste:

- Counting Bloom Filter: Questi permettono di gestire scenari dinamici, dove è necessario aggiungere o rimuovere elementi dall'insieme rappresentato, offrendo maggiore versatilità rispetto ai Bloom Filter tradizionali;[26]
- Applicazioni combinate: L'uso dei Bloom Filter in combinazione con algoritmi di hashing avanzati o con strutture dati più sofisticate permette di affrontare problemi specifici come il rilevamento di mutazioni o l'analisi dei metagenomi.[8]

Capitolo 3

Analisi dei tool

3.1 Introduzione ai tool per il conteggio dei k-mer

Gli strumenti per il conteggio dei k-mer si differenziano principalmente per tre parametri chiave:

- Efficienza computazionale: Capacità di gestire grandi dataset genomici con risorse computazionali limitate;
- Precisione di conteggio: Accuratezza nella identificazione e quantificazione delle sotto-sequenze;
- Versatilità: Flessibilità nell'adattarsi a diversi contesti di ricerca e tipologie di analisi.

Le principali strategie algoritmiche implementate in questi tool includono tecniche di hashing, suffix tree, e metodi di compressione basati su grafi. Tali approcci mirano a ottimizzare tanto la velocità di elaborazione quanto l'utilizzo della memoria, elementi cruciali nell'analisi di grandi dataset genomici.

Tra gli strumenti più significativi nel panorama bioinformatico contemporaneo si annoverano Jellyfish[16], KMC[18] e DSK[31], ciascuno con specifiche peculiarità nell'implementazione degli algoritmi di conteggio. La scelta dello strumento dipende strettamente dalle caratteristiche specifiche del dataset in analisi e dagli obiettivi della ricerca.

3.2 BFCounter

BFCounter rappresenta un approccio innovativo per il conteggio dei k-mer in sequenze di DNA, progettato per gestire l'esigenza di elaborare dataset di grandi dimensioni in modo efficiente in termini di memoria.

Questo algoritmo utilizza un Bloom Filter, una struttura dati probabilistica, per registrare in modo implicito i k-mer osservati, evitando di memorizzare direttamente quelli unici e riducendo così il consumo di memoria.

3.2.1 Architettura

L'algoritmo BFCounter è suddiviso in due fasi principali, ciascuna delle quali è fondamentale per l'identificazione e il conteggio dei k-mer non unici.

Fase 1: Identificazione preliminare

In questa fase, il Bloom Filter viene utilizzato per determinare se un k-mer è già stato osservato. Per ogni k-mer generato dalle sequenze di input, il procedimento è il seguente:

- Se il k-mer non è presente nel Bloom Filter, i bit corrispondenti vengono impostati a 1, indicando che il k-mer è stato osservato;
- Se il k-mer è presente nel Bloom Filter, si procede a verificare la sua presenza nella tabella hash. I k-mer non unici vengono aggiunti alla tabella hash.

Il Bloom Filter agisce come una struttura di staging, memorizzando in modo implicito tutti i k-mer, mentre la tabella hash è utilizzata esclusivamente per quelli con copertura maggiore o uguale a 2.

Fase 2: Conteggio esatto

Dopo la fase preliminare, l'algoritmo effettua una seconda scansione dei dati per determinare i conteggi precisi dei k-mer presenti nella tabella hash. In questa fase, tutti i k-mer unici vengono rimossi dalla tabella, riducendo ulteriormente il consumo di memoria.

Inoltre, poiché il Bloom Filter può generare falsi positivi, questa seconda fase consente di identificare e rimuovere i k-mer che sono stati erroneamente aggiunti alla tabella hash nella fase precedente. Questo processo rende i conteggi finali più accurati, minimizzando l'impatto dei falsi positivi.

3.2.2 Pseudocodice dell'algoritmo

Il funzionamento del conteggio dei k-mer di BFCounter può essere descritto mediante il seguente pseudocodice, che evidenzia i passi principali dell'algoritmo. In particolare, si distinguono le due fasi principali: la costruzione del Bloom Filter e l'elaborazione della tabella hash per il conteggio esatto dei k-mer.

Nella prima fase, per ogni k-mer derivato dalle letture di input, si calcola la sua rappresentazione canonica x_{rep} e si verifica se è presente nel Bloom Filter B. Se x_{rep} è già presente nel Bloom Filter ma non nella tabella hash T, viene aggiunto con un conteggio iniziale pari a zero; in caso contrario, x_{rep} viene aggiunto nel Bloom Filter.

Nella seconda fase, si esegue un'altra scansione per aggiornare i conteggi dei k-mer nella tabella hash. Dopo questa operazione, tutti i k-mer con un conteggio pari a 1, considerati unici, vengono rimossi dalla tabella T. Questo approccio a doppia scansione assicura un'efficiente gestione della memoria e un calcolo accurato dei k-mer non unici, combinando le proprietà del Bloom Filter con una tabella hash per il conteggio esatto.

Algoritmo 3 Algoritmo per il conteggio dei k-mer con Bloom Filter

```
1: B ← empty Bloom Filter of size m
2: T ← hash table
3: for all reads s do
4:     for all k-mers x in s do
5:         x_rep ← min(x, revcomp(x)) //x_rep is the canonical k-mer for x
6:         if x_rep ∈ B then
7:             if x_rep ∉ T then
8:                 T[x_rep] ← 0
9:         else
10:            add x_rep to B
11:    for all reads s do
12:        for all k-mers x in s do
13:            x_rep ← min(x, revcomp(x))
14:            if x_rep ∈ T then
15:                T[x_rep] ← T[x_rep] + 1
16:    for all x ∈ T do
17:        if T[x] = 1 then
18:            remove x from T
```

3.2.3 Ottimizzazioni ed estensioni

Gestione dei falsi positivi

BFCounter riduce l'impatto dei falsi positivi generati dal Bloom Filter utilizzando una strategia a due fasi: nella prima fase, i k-mer rilevati come già osservati vengono inseriti in una tabella hash per ulteriori verifiche; successivamente, nella seconda fase, la tabella hash viene utilizzata per calcolare con precisione il conteggio effettivo dei k-mer, consentendo di correggere eventuali segnalazioni errate dovute ai falsi positivi.

La probabilità di falsi positivi è calcolata come $(1 - e^{-d\frac{n}{m}})^d$, e nel caso di BFCounter è mantenuta inferiore al 2%, grazie all'uso di un Bloom Filter con una dimensione ottimizzata a 4 bit per k-mer e tre funzioni hash. Questo compromesso tra utilizzo della memoria e precisione consente di ridurre significativamente il consumo di risorse mantenendo un'alta affidabilità del processo.

Rimozione dei k-mer unici e gestione degli errori

Un altro aspetto distintivo di BFCounter è la sua capacità di ridurre significativamente la memoria necessaria eliminando i k-mer unici. Questi ultimi, spesso generati da errori di sequenziamento, costituiscono una frazione sostanziale dei dati, superando il 50% in molti dataset.

Identificando e rimuovendo tali elementi nella fase iniziale, BFCounter riduce il numero di k-mer da gestire nelle fasi successive. Questa ottimizzazione non solo diminuisce il consumo di memoria, ma migliora anche la qualità del grafo stesso, riducendo il rumore derivante dagli errori di sequenziamento.

Cutoff di copertura

BFCounter permette anche di configurare un valore minimo di copertura per i k-mer da considerare. Di default, l'algoritmo esclude tutti i k-mer con una copertura inferiore a 2, eliminando così gli elementi rari che spesso derivano da errori di sequenziamento.

Per applicazioni che richiedono un cutoff più elevato, è possibile utilizzare un Counting Bloom Filter, che permette di contare in modo efficiente le occorrenze di ciascun k-mer durante l'inserimento. Questo approccio consente di escludere direttamente i k-mer con copertura inferiore alla soglia configurata, evitando di occupare spazio nella tabella hash e migliorando ulteriormente l'efficienza del sistema.

Parallelizzazione

Sebbene l'implementazione base non sia multi-thread, BFCounter può essere parallelizzato utilizzando strutture dati senza blocco. Ad esempio, la gestione della concorrenza può essere ottenuta attraverso operazioni atomiche sui bit, evitando collisioni accidentali.

3.3 Codice di BFCounter

3.3.1 Implementazione del Bloom Filter

Nel software BFCounter, il Bloom Filter rappresenta una struttura dati centrale per elaborare in modo efficiente grandi dataset genomici. Grazie alla libreria di Partow C++ Bloom Filter Library, BFCounter ottimizza la gestione dei k-mer, sfruttando l'efficienza delle verifiche probabilistiche offerte da questa struttura dati.

Struttura della libreria Bloom Filter

La libreria implementa il Bloom Filter come una classe configurabile, consentendo di definire la dimensione e il numero di funzioni hash utilizzate. Questi parametri influenzano direttamente la probabilità di falsi positivi e il consumo di memoria. La struttura della classe include una bitmap e una serie di funzioni hash personalizzabili. Di seguito è riportata una rappresentazione semplificata della classe:

```
1  class bloom_filter
2  {
3  protected:
4      typedef uint64_t bloom_type;
5      typedef unsigned char cell_type;
6
7  public:
8      // Costruttore
9      bloom_filter(const std::size_t& predicted_element_count,
10                  const double& false_positive_probability,
11                  const std::size_t& random_seed)
12     : bit_table_(0),
13         predicted_element_count_(predicted_element_count),
```

```
14          nserted_element_count_(0),
15           random_seed_((random_seed) ? random_seed : 0xA5A5A5A5),
16          desired_false_positive_probability_(false_positive_probability)
17      {
18          find_optimal_parameters();
19           bit_table_ = new cell_type[table_size_ / bits_per_char];
20           generate_unique_salt();
21           std::fill_n(bit_table_,(table_size_ / bits_per_char),0x00);
22      }
23
24      // Inserimento di un elemento
25      inline void insert(const unsigned char* key_begin, const std::size_t
            length)
26      {
27          std::size_t bit_index = 0;
28          std::size_t bit = 0;
29          for(std::vector<bloom_type>::iterator it = salt_.begin(); it != salt_
                .end(); ++it)
30          {
31              compute_indices(hash_ap(key_begin,length,(*it)),bit_index,bit);
32              bit_table_[bit_index / bits_per_char] |= bit_mask[bit];
33          }
34          ++inserted_element_count_;
35      }
36
37      // Verifica della presenza di un elemento
38      inline virtual bool contains(const unsigned char* key_begin, const std::
            size_t length) const
39      {
40          std::size_t bit_index = 0;
41          std::size_t bit = 0;
42          for(std::vector<bloom_type>::const_iterator it = salt_.begin(); it !=
                 salt_.end(); ++it)
43          {
44              compute_indices(hash_ap(key_begin,length,(*it)),bit_index,bit);
45              if ((bit_table_[bit_index / bits_per_char] & bit_mask[bit]) !=
                    bit_mask[bit])
46              {
47              return false;
48              }
49          }
50          return true;
51      }
```

Il frammento di codice definisce una classe `bloom_filter`, che implementa una versione base di un Bloom Filter.

Nel costruttore, vengono inizializzati i parametri del Bloom Filter, tra cui il numero di elementi previsti, la probabilità di falso positivo, e un seme per la generazione di numeri casuali. Viene anche calcolata la dimensione della tabella dei bit `bit_table_` e allocata dinamicamente una memoria per essa.

La funzione `insert` inserisce un elemento nel Bloom Filter, utilizzando un insieme di valori di hash generati con la funzione `hash_ap`, e aggiorna i bit corrispondenti nella tabella dei bit.

La funzione `contains` verifica se un elemento è presente nel Bloom Filter: esegue gli stessi calcoli degli hash e controlla se tutti i bit corrispondenti sono settati. Se uno dei bit non è settato, l'elemento non è presente nel Bloom Filter. In caso contrario, l'elemento è presumibilmente presente, tenendo conto della probabilità di falso positivo.

3.3.2 Libreria SparseHash

La libreria `SparseHash`, sviluppata da Google, rappresenta un elemento chiave nell'implementazione di BFCounter per il conteggio dei k-mer. Questa libreria offre una hash table a densità sparsa, progettata per ottimizzare il consumo di memoria, mantenendo al contempo elevate prestazioni. In BFCounter, `SparseHash` viene utilizzata per memorizzare il conteggio dei k-mer, riducendo il carico computazionale e l'overhead.

Caratteristiche della libreria SparseHash

SparseHash implementa due principali strutture dati:

- `sparse_hash_map`: una tabella hash ottimizzata per memorizzare una quantità ridotta di elementi rispetto allo spazio totale disponibile;
- `dense_hash_map`: una tabella hash più compatta ma meno efficiente in termini di memoria, utilizzata quando il carico della tabella è elevato.

Per il contesto di BFCounter, `sparse_hash_map` è particolarmente utile poiché il numero di k-mer osservati è generalmente molto inferiore rispetto al numero totale di possibili k-mer, rendendo il modello a densità sparsa ideale per ridurre il consumo di memoria.

Gestione della memoria

Un aspetto fondamentale dell'uso di `sparse_hash_map` in BFCounter è la gestione della memoria. SparseHash utilizza un'allocazione efficiente per minimizzare lo spreco di spazio, ma richiede un'attenzione particolare per evitare conflitti o overhead.

Rispetto alle librerie standard, SparseHash offre un consumo di memoria significativamente inferiore, specialmente nei casi in cui molti slot della tabella hash rimangono vuoti.

Nuova versione

Con la nuova versione della libreria sono state osservate alcune modifiche che comportano un leggero rallentamento delle prestazioni di BFCounter. Questo incremento dell'overhead computazionale, seppur minimo, potrebbe essere attribuito a nuovi meccanismi interni di gestione della memoria e a ottimizzazioni nella struttura della hash table.

3.3.3 Funzione `hash_ap`

In particolare la funzione `hash_ap` è utilizzata all'interno dei Bloom Filter per generare un valore hash a partire da una sequenza di byte, ed è uno degli elementi chiave per garantire una bassa probabilità di collisione durante l'indicizzazione degli elementi.

```
1     bloom_type hash_ap(const unsigned char* begin, std::size_t
          remaining_length, bloom_type hash) const
2     {
3          const unsigned char* it = begin;
4          while(remaining_length >= 2)
5          {
6              hash ^=    (hash <<  7) ^  (*it++) * (hash >> 3);
7              hash ^= (~((hash << 11) + ((it++) ^ (hash >> 5))));
8              remaining_length -= 2;
9          }
10         if (remaining_length)
11         {
12             hash ^= (hash <<  7) ^ (it) * (hash >> 3);
13         }
14         return hash;
15    }
```

Questa funzione prende in input tre parametri principali:

- `begin`: un puntatore alla sequenza di byte da elaborare;
- `remaining_length`: la lunghezza della sequenza in input;
- `hash`: un valore hash iniziale che verrà modificato durante l'elaborazione.

La funzione elabora i byte di input in coppie, quando possibile, e utilizza operazioni bitwise per mescolare i byte di input con il valore hash. Questo processo è suddiviso in più passaggi:

- La funzione itera attraverso i byte della sequenza in input in coppie;
- Per ogni coppia, applica due operazioni per distribuire uniformemente i bit:
  1. `hash ^= (hash « 7) ^(*it++) * (hash » 3)`
  2. `hash ^= (~((hash « 11) + ((*it++) ^(hash » 5))))`
- Se rimane un solo byte alla fine della sequenza (sequenza di lunghezza dispari), questo viene processato con un'operazione simile per garantire che tutti i byte contribuiscano al risultato finale.

La funzione restituisce il valore hash modificato, che rappresenta una distribuzione ad alta qualità. Questo è molto utile nei Bloom Filter per minimizzare le collisioni e garantire che cambiamenti minimi negli input producano risultati hash significativamente diversi.

3.3.4 Operazioni bitwise nella funzione `hash_ap`

La funzione `hash_ap` utilizza due linee principali di operazioni bitwise per mescolare i byte dell'input e modificare il valore hash, in particolare:

Prima operazione: `hash ^= (hash « 7) ^(*it++) * (hash » 3);`

Questa operazione combina più trasformazioni del valore hash e dei byte di input. Il significato di ciascun componente è:

- `hash « 7`: Effettua uno shift a sinistra del valore hash di 7 bit.;
- `hash » 3`: Effettua uno shift a destra del valore hash di 3 bit;
- `(*it++)`: Dereferenzia il puntatore `it` per ottenere il valore del byte corrente e lo incrementa al byte successivo;
- `(*it++) * (hash » 3)`: Moltiplica il valore del byte corrente con il valore dello shift a destra del valore hash;
- `hash ^= ...`: Combina il risultato con l'hash corrente usando l'operazione XOR.

Seconda operazione: `hash ^= (~((hash « 11) + ((*it++) ^(hash » 5))))`

Questa linea aggiunge un ulteriore livello di complessità alla funzione di hash. Ecco il dettaglio di ogni componente:

- `hash « 11`: Effettua uno shift a sinistra di 11 bit sul valore hash;
- `hash » 5`: Effettua uno shift a destra di 5 bit sul valore hash;
- `(*it++) ^(hash » 5)`: Combina il valore del byte corrente con lo shift a destra del valore hash usando XOR;
- `~(...)`: L'operazione NOT bitwise inverte tutti i bit del risultato;
- `hash ^= ...`: Come nella prima linea, l'operazione XOR integra i risultati con l'hash corrente, garantendo una trasformazione pseudo-casuale.

3.3.5 Metodo `insert`

```
1     inline void insert(const unsigned char* key_begin, const std::size_t
          length)
2     {
3       std::size_t bit_index = 0;
4       std::size_t bit = 0;
5       for(std::vector<bloomtype>::iterator it = salt.begin(); it != salt_.end
            (); ++it)
6       {
7          compute_indices(hash_ap(key_begin,length,(*it)),bit_index,bit);
```

```
8           bittable[bit_index / bits_per_char] |= bit_mask[bit];
9       }
10      ++inserted_elementcount;
11  }
```

Il metodo `insert` utilizza la funzione `hash_ap` per inserire un nuovo elemento nel Bloom Filter. I parametri principali della funzione sono:

- `key_begin`: un puntatore alla sequenza di byte che rappresenta la chiave da inserire;
- `length`: la lunghezza della chiave in input.

Algoritmo

Il metodo segue i seguenti passaggi principali:

- Inizializza le variabili `bit_index` e `bit` per tenere traccia di quale bit deve essere impostato nella tabella del Bloom Filter;
- Itera attraverso i valori di `salt`, un insieme di semi hash, utilizzando ciascuno per generare un hash unico della chiave;
- Per ogni seme:
  - Calcola un valore hash usando `hash_ap`, passando la chiave, la sua lunghezza e il seme attuale;
  - Converte l'hash calcolato in un indice di bit specifico tramite la funzione `compute_indices`;
  - Imposta il bit corrispondente nella tabella del Bloom Filter utilizzando l'operazione OR bitwise.
- Incrementa un contatore che tiene traccia del numero totale di elementi inseriti.

Il metodo `insert` utilizza hash multipli, ciascuno derivato da un seme differente, per ridurre il rischio di falsi positivi; ogni hash punta a un bit nella tabella, che viene impostato a 1. Quando si esegue una query, il Bloom Filter verifica se tutti questi bit sono impostati, permettendo di determinare se l'elemento è probabilmente presente o certamente assente. Questo approccio offre una rappresentazione compatta ed efficiente per verificare l'appartenenza a un insieme con un rischio controllato di falsi positivi.

3.3.6 Algoritmo di conteggio dei k-mer

L'algoritmo di conteggio implementato in BFCounter rappresenta il cuore del software, consentendo l'identificazione e la quantificazione dei k-mer da sequenze genomiche di grandi dimensioni. Questo algoritmo sfrutta un design a pipeline che integra diverse strutture dati, per garantire efficienza sia in termini di tempo che di memoria.

Flusso operativo

L'algoritmo segue una sequenza strutturata di operazioni:

1. Lettura dei dati: Le sequenze genomiche vengono caricate da file FASTA o FASTQ;
2. Generazione dei k-mer: Ogni sequenza viene suddivisa in sottostringhe di lunghezza k, utilizzando uno sliding window;
3. Filtraggio con il Bloom Filter: Ogni k-mer viene cercato nel Bloom Filter per determinare se è già stato processato;
4. Aggiornamento della hash table: I k-mer nuovi vengono aggiunti alla hash table e il loro conteggio incrementato.

Conteggio

```
1   void CountBF_Normal(const CountBF_ProgramOptions &opt) {
2   // Inizializzazione
3   BloomFilter BF(opt.nkmers, (size_t) opt.bf, seed);
4   hmap_t kmap; // Tabella hash per i k-mer
5
6   // Primo ciclo: aggiunta dei k-mer al Bloom Filter
7   while (!done) {
8       size_t reads_now = 0;
9       while (reads_now < read_chunksize) {
10          if (FQ.read_next(name, &name_len, s, &len, NULL, NULL) >= 0) {
11              readv[reads_now].assign(s);
12              ++n_read;
13              ++reads_now;
14          } else {
15              done = true;
16              break;
17          }
18      }
19
20      #pragma omp parallel
21      {
22          // Itera su ogni read
23          for (size_t index = 0; index < reads_now; ++index) {
24              const char *cstr = readv[index].c_str();
25              KmerIterator iter(cstr);
26              for (; iter != iterend; ++iter) {
27                  Kmer rep = iter->first.rep(); // Calcola il k-mer
                        canonico
28                  if (BF.search(rep) == 0) { // Se non compare nel Bloom
                        Filter
29                      BF.insert(rep); // Aggiungi al Bloom Filter
30                      kmap.insert(KmerIntPair(rep, 0)); // Aggiungi alla
                            tabella hash
31                  }
```

```
32                  }
33              }
34          }
35      }
36
37      // Secondo ciclo: conteggio dei k-mer
38      FQ.reopen();
39      while (!done) {
40          // ...
41          for (size_t index = 0; index < reads_now; ++index) {
42              const char *cstr = readv[index].c_str();
43              KmerIterator iter(cstr);
44              for (; iter != iterend; ++iter) {
45                  Kmer rep = iter->first.rep();
46                  auto it = kmap.find(rep);
47                  if (it != kmap.end()) {
48                      it->second++; // Incrementa il conteggio
49                  }
50              }
51          }
52      }
53
54      // Rimozione dei k-mer con conteggio 1
55      for (auto it = kmap.begin(); it != kmap.end(); ) {
56          if (it->GetVal() <= 1) {
57              it = kmap.erase(it); // Rimuovi il k-mer
58          } else {
59              ++it;
60          }
61      }
62  }
```

Il codice implementa l'algoritmo per il conteggio dei k-mer. Nel primo ciclo, per ogni sequenza di lettura, i k-mer vengono estratti e aggiunti al Bloom Filter, qualora non siano già presenti. Contestualmente, i k-mer vengono inseriti in una tabella hash associandovi un conteggio iniziale pari a zero.

Nel secondo ciclo, il codice rielabora le sequenze di lettura e aggiorna i conteggi dei k-mer già presenti nella tabella hash, incrementando il valore associato ad ogni occorrenza.

Infine, nel terzo passaggio, i k-mer con un conteggio pari a uno vengono rimossi dalla tabella hash, in quanto considerati non significativi per l'analisi. Questo approccio sfrutta il Bloom Filter per una rapida identificazione dei k-mer già incontrati, mentre la tabella hash consente di mantenere un conteggio preciso delle occorrenze.

3.4 Jellyfish

3.4.1 Gestione dinamica

Jellyfish si distingue per l'implementazione di una struttura dati hash dinamica estremamente efficiente, progettata per garantire un accesso rapido ai dati e ridurre il consumo di memoria. Utilizza una rappresentazione compatta dei k-mer, che consente di comprimere le informazioni genetiche senza perdere precisione. Questa tecnica è particolarmente utile nel trattamento di grandi dataset genomici, dove ogni bit risparmiato può portare a significativi risparmi in termini di risorse computazionali.

L'approccio basato su hash dinamici permette a Jellyfish di gestire grandi collezioni di k-mer in modo estremamente veloce, adattandosi dinamicamente alla quantità di dati processati. Questo approccio non solo riduce il consumo di memoria, ma evita anche allocazioni superflue, ottimizzando il processo di elaborazione. In aggiunta, l'utilizzo di tecniche avanzate di gestione della cache riduce il tempo di accesso ai dati, migliorando notevolmente le prestazioni durante l'elaborazione.

3.4.2 Parallelismo e scalabilità

Uno degli aspetti più innovativi di Jellyfish è il suo utilizzo intensivo del parallelismo. L'intero spazio di ricerca dei k-mer viene suddiviso in porzioni più piccole che possono essere gestite simultaneamente da più thread. Questa suddivisione riduce i conflitti tra i thread, migliorando il throughput complessivo. Per garantire un bilanciamento ottimale del carico, l'algoritmo di Jellyfish assegna i compiti ai thread in modo da evitare sovraccarico in alcune aree di memoria o processori.

Grazie a questo design, Jellyfish si adatta perfettamente a sistemi multicore e multiprocessore, sfruttandone appieno le capacità. Questo è particolarmente vantaggioso quando si analizzano grandi quantità di dati provenienti dal sequenziamento di nuova generazione. La scalabilità non è limitata al numero di core; il software può essere eseguito su cluster di macchine, rendendolo adatto per applicazioni su larga scala, come il sequenziamento di interi genomi o metagenomi complessi.

3.4.3 Filtraggio dei dati

Jellyfish integra funzionalità avanzate di filtraggio dei k-mer che permettono agli utenti di migliorare la qualità dei dati analizzati. Ad esempio, è possibile impostare soglie per filtrare i k-mer con frequenze troppo basse, eliminando così errori di sequenziamento o rumore derivante da letture poco affidabili. Questa operazione contribuisce a ottimizzare il carico computazionale, poiché rimuove informazioni ridondanti o inutili prima che vengano elaborate ulteriormente.

Il software consente inoltre di pre-processare i dati per identificare specifiche caratteristiche genetiche o ridurre la complessità del dataset. Questa capacità rende Jellyfish uno strumento particolarmente utile in pipeline automatizzate, dove i dati grezzi vengono convertiti in informazioni utili attraverso una serie di passaggi predefiniti.

3.5 Confronto tra Jellyfish e BFCounter

Le implementazioni di Jellyfish e BFCounter affrontano il conteggio dei k-mer con approcci distinti, ottimizzati per esigenze diverse nel trattamento di dati genomici su larga scala.

Jellyfish utilizza tabelle hash dinamiche per garantire un accesso deterministico e una gestione precisa dei conteggi. Il sistema di hash distribuito bilancia efficacemente il carico computazionale tra i thread, sfruttando le architetture multicore e ottimizzando l'accesso alla cache e il consumo di memoria, garantendo elevate prestazioni anche su dataset di grandi dimensioni.

BFCounter, al contrario, si basa su un approccio probabilistico usando i Bloom Filter, che consentono una compressione significativa rispetto alle tabelle hash tradizionali. Questo implica una tolleranza a una piccola percentuale di falsi positivi, minimizzata da una fase di verifica finale che raffina i conteggi. La dimensione del Bloom Filter e il numero di funzioni hash sono configurabili, permettendo di bilanciare memoria e precisione. BFCounter è quindi ideale per scenari con risorse hardware limitate, dove il risparmio di memoria è prioritario.

Jellyfish suddivide l'intero spazio degli hash in partizioni indipendenti, ottimizzando l'efficienza dei thread e riducendo i colli di bottiglia. BFCounter segue un processo in due fasi: nella prima, i k-mer vengono inseriti nel Bloom Filter, in una fase parallela e leggera, mentre nella seconda i conteggi sono affinati su una struttura dati secondaria. Questo approccio rende BFCounter meno dipendente dal numero di thread disponibili rispetto a Jellyfish, ma mantiene comunque buone prestazioni su hardware meno potente.

Un'altra differenza riguarda la gestione della memoria. Jellyfish richiede memoria proporzionale ai k-mer distinti analizzati, mentre BFCounter è più compatto, permettendo di gestire dataset di grandi dimensioni con risorse limitate. Questo rende BFCounter particolarmente adatto per l'analisi di dati di sequenziamento su larga scala in ambienti con memoria limitata.

Infine, Jellyfish produce risultati esatti e deterministici, ideali per applicazioni che richiedono alta precisione, mentre BFCounter, con la sua tolleranza ai falsi positivi, è più adatto per analisi preliminari o stime relative di abbondanza, dove il risparmio di risorse è fondamentale.

Capitolo 4

Replica dei test

4.1 Obiettivi

La replica dei test costituisce un elemento cardine per garantire l'affidabilità e la riproducibilità dei risultati nel campo della bioinformatica, un settore in cui la validazione degli strumenti riveste un ruolo centrale per assicurare l'applicabilità delle analisi a contesti reali.

In questo capitolo verranno esaminati i risultati della replica dei test condotti su due strumenti per il conteggio dei k-mer, BFCounter e Jellyfish, entrambi ampiamente utilizzati per l'analisi di dati genomici su larga scala.

L'analisi si propone di valutare le prestazioni di ciascun tool in termini di efficienza e consumo di memoria, utilizzando un dataset e parametri definiti, al fine di evidenziare i punti di forza e le limitazioni di ciascun approccio.

4.2 Metodologia

4.2.1 Dataset

Il dataset impiegato per i test è stato scaricato dalla piattaforma NCBI Sequence Read Archive (SRA). Si tratta del file identificato con l'accession number SRR11528307, un dataset genomico rappresentativo in formato FASTA.

Questo dataset è stato selezionato per la sua adeguata complessità e per la sua rappresentatività rispetto a contesti di analisi reali. Il formato FASTA è uno standard ampiamente utilizzato nella comunità bioinformatica, contenente sequenze nucleotidiche prive di ambiguità e annotazioni dettagliate sulle sequenze stesse.

Le caratteristiche principali del dataset includono:

- Dimensione: una grandezza sufficiente a simulare scenari realistici di analisi di grandi volumi di dati genomici;
- Formato: compatibile con entrambi gli strumenti testati, garantendo uniformità nell'elaborazione dei dati;

- Origine: dati provenienti da un archivio pubblico e ben documentato, assicurando trasparenza e possibilità di replicazione da parte di altri ricercatori.

4.2.2 Configurazione

La configurazione sperimentale è stata progettata per garantire condizioni il più possibile omogenee tra le esecuzioni di BFCounter e Jellyfish. Le analisi sono state condotte su una macchina dotata delle seguenti specifiche hardware:

- Processore: CPU multi-core ad alte prestazioni;
- Memoria RAM: almeno 16 GB, per gestire carichi di lavoro intensivi;
- Sistema operativo: ambiente basato su Linux, configurato per garantire compatibilità con entrambi gli strumenti.

Ogni esecuzione è stata monitorata utilizzando strumenti di misurazione delle risorse, che hanno permesso di registrare metriche quali:

- Tempo di esecuzione: misurato in secondi, per confrontare l'efficienza computazionale;
- Consumo di memoria: per valutare l'efficacia dell'utilizzo delle risorse da parte dei tool.

4.2.3 Parametri

Per garantire una valutazione uniforme e comparabile, entrambi gli strumenti sono stati configurati utilizzando parametri equivalenti. Nello specifico, è stato utilizzato un valore di k = 31, che rappresenta la lunghezza dei k-mer.

Questo valore è comunemente utilizzato nelle analisi genomiche, poiché bilancia la sensibilità del conteggio con la capacità di rappresentare sequenze genomiche uniche.

Parametri per BFCounter

Per l'esecuzione di BFCounter, i parametri principali impostati sono stati:

- `-k 31`: lunghezza dei k-mer;
- `-n 1000`: numero di funzioni hash utilizzate;
- `-o output.bf`: file di output generato contenente i risultati del conteggio.

Parametri per Jellyfish

Per l'esecuzione di Jellyfish, i parametri principali sono stati:

- `-m 31`: lunghezza dei k-mer;
- `-s 100M`: dimensione iniziale della hash table;
- `-t 10`: numero di thread utilizzati per il calcolo parallelo;
- `-o output.jf`: file di output generato contenente i risultati del conteggio.

Strumenti di monitoraggio

Nel caso di BFCounter, è stato necessario modificare il codice sorgente per aggiungere il calcolo e la stampa del tempo di esecuzione e dell'uso della memoria, poiché tali funzionalità non erano incluse nativamente. Questo ha permesso di ottenere una valutazione comparabile con quella di Jellyfish, che invece dispone già di strumenti integrati per il monitoraggio di queste metriche, oltre a fornire ulteriori statistiche sulle risorse utilizzate.

Per ottenere una misurazione precisa delle risorse utilizzate, è stato impiegato il comando `/usr/bin/time` con opzione `-v`, che consente di registrare il tempo totale di esecuzione, la memoria massima utilizzata durante il processo e altri dettagli utili per valutarne il comportamento.

4.3 Risultati

4.3.1 Risultati sperimentali

L'analisi sperimentale condotta su BFCounter e Jellyfish ha prodotto i seguenti risultati.

Tempo di esecuzione:

- BFCounter: 28.386 secondi
- Jellyfish: 9.89 secondi

Utilizzo della memoria:

- BFCounter: 41172 KB
- Jellyfish: 844508 KB

Dai risultati è emerso che, come previsto, BFCounter si distingue per un utilizzo della memoria significativamente inferiore rispetto a Jellyfish, grazie all'impiego dei Bloom Filter, struttura dati particolarmente efficiente per la gestione della memoria. Tuttavia, questo vantaggio è compensato da un tempo di esecuzione maggiore, attribuibile anche alla necessità di effettuare due passaggi sui dati per ottenere i conteggi esatti.

D'altra parte, Jellyfish, grazie alla sua architettura multi-thread e all'uso di tabelle hash dinamiche, offre tempi di esecuzione notevolmente ridotti, rendendolo più adatto per applicazioni in cui la velocità è un requisito prioritario. Il costo di questa ottimizzazione si riflette in un consumo di memoria maggiore.

4.3.2 Confronto con i risultati in letteratura

Il confronto con il lavoro originale di Melsted e Pritchard[25] su BFCounter conferma le loro osservazioni principali. Gli esperimenti hanno sostanzialmente validato i risultati originali, evidenziando che:

- Memoria: BFCounter mantiene il suo vantaggio di richiedere meno memoria rispetto a Jellyfish, con variazioni dipendenti dalle specificità del dataset;

- Tempo di esecuzione: Analogamente, BFCounter ha mostrato tempi di esecuzione più elevati, coerenti con il metodo a due passaggi descritto.

Tuttavia, è importante notare che le specifiche hardware utilizzate e la natura del dataset possono influenzare significativamente questi risultati. Ad esempio, Melsted e Pritchard utilizzavano un dataset di copertura del genoma umano a 40x con una lunghezza delle read di 36 bp, mentre nel nostro caso il dataset presentava caratteristiche diverse. Queste differenze sottolineano la necessità di contestualizzare i risultati in base al caso d'uso specifico.

Capitolo 5

Conclusione

5.1 Analisi comparativa

In questo capitolo vengono presentati i risultati dell'analisi comparativa tra BFCounter e Jellyfish, con particolare attenzione alle prestazioni in termini di velocità di esecuzione e utilizzo della memoria.

La velocità di esecuzione rappresenta un aspetto importante per l'analisi di grandi volumi di dati. Jellyfish ha dimostrato un vantaggio significativo in termini di tempo di esecuzione grazie alla sua architettura multi-thread ottimizzata, che consente di sfruttare appieno i sistemi multicore.

BFCounter, pur avendo una struttura algoritmica efficiente basata sui Bloom Filter, risulta generalmente più lento, soprattutto su dataset di grandi dimensioni.

Per quanto riguarda il consumo di memoria, BFCounter ha mostrato un'efficienza superiore, grazie all'uso dei Bloom Filter, che comprimono efficacemente le informazioni genomiche riducendo la memoria richiesta per il conteggio.

Jellyfish, d'altro canto, utilizza una hash table dinamica che, pur garantendo elevate prestazioni computazionali, comporta un consumo di memoria più elevato, specialmente durante l'elaborazione di dataset complessi.

L'analisi comparativa ha evidenziato che entrambi gli strumenti sono accurati, ma differiscono in termini di efficienza computazionale e consumo di risorse. Jellyfish si distingue per la sua velocità, risultando particolarmente indicato per applicazioni che richiedono analisi rapide su sistemi con risorse hardware adeguate. BFCounter, invece, rappresenta una scelta ideale quando la memoria è una risorsa limitata, grazie alla sua ottimizzazione nell'utilizzo delle risorse. La scelta tra i due strumenti dipenderà quindi dalle specifiche esigenze applicative e dal contesto operativo.

Bibliografia

[1] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Commun. ACM* 13.7 (lug. 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[2] Swati C Manekar e Shailesh R Sathe. "A benchmark study of k-mer counting methods for high-throughput sequencing". In: *GigaScience* 7.12 (ott. 2018), giy125. ISSN: 2047-217X. DOI: 10.1093/gigascience/giy125. eprint: https://academic.oup.com/gigascience/article-pdf/7/12/giy125/27011542/giy125_reviewer_3_report_(original_submission).pdf. URL: https://doi.org/10.1093/gigascience/giy125.

[3] Zhenyu Li et al. "Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph". In: *Briefings in Functional Genomics* 11.1 (dic. 2011), pp. 25–37. ISSN: 2041-2649. DOI: 10.1093/bfgp/elr035. eprint: https://academic.oup.com/bfg/article-pdf/11/1/25/473950/elr035.pdf. URL: https://doi.org/10.1093/bfgp/elr035.

[4] Li Song, Liliana Florea e Ben Langmead. "Lighter: fast and memory-efficient sequencing error correction without counting". In: *Genome Biology* 15.11 (2014), p. 509. ISSN: 1474-760X. DOI: 10.1186/s13059-014-0509-9. URL: https://doi.org/10.1186/s13059-014-0509-9.

[5] Phillip E. Compeau, Pavel A. Pevzner e Glenn Tesler. "How to apply de Bruijn graphs to genome assembly". In: *Nature Biotechnology* 29.11 (2011), pp. 987–991. ISSN: 1087-0156. DOI: 10.1038/nbt.2023. URL: https://doi.org/10.1038/nbt.2023.

[6] Rayan Chikhi e Paul Medvedev. "Informed and automated k-mer size selection for genome assembly". In: *Bioinformatics* 30.1 (giu. 2013), pp. 31–37. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btt310. eprint: https://academic.oup.com/bioinformatics/article-pdf/30/1/31/48912483/bioinformatics_30_1_31.pdf. URL: https://doi.org/10.1093/bioinformatics/btt310.

[7] Todd J. Treangen e Steven L. Salzberg. "Repetitive DNA and next-generation sequencing: computational challenges and solutions". In: *Nature Reviews Genetics* 13.1 (2012), pp. 36–46. ISSN: 1471-0064. DOI: 10.1038/nrg3117. URL: https://doi.org/10.1038/nrg3117.

[8] Derrick E. Wood e Steven L. Salzberg. "Kraken: ultrafast metagenomic sequence classification using exact alignments". In: *Genome Biology* 15.3 (2014), R46. ISSN: 1474-760X. DOI: 10.1186/gb-2014-15-3-r46. URL: https://doi.org/10.1186/gb-2014-15-3-r46.

[9] Edoardo Pasolli et al. "Machine Learning Meta-analysis of Large Metagenomic Datasets: Tools and Biological Insights". In: *PLoS Computational Biology* 12.7 (2016), e1004977. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1004977. URL: https://doi.org/10.1371/journal.pcbi.1004977.

[10] Sahar Abubucker et al. "Metabolic Reconstruction for Metagenomic Data and Its Application to the Human Microbiome". In: *PLoS Computational Biology* 8.6 (2012), e1002358. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1002358. URL: https://doi.org/10.1371/journal.pcbi.1002358.

[11] Fritz J. Sedlazeck et al. "Accurate detection of complex structural variations using single-molecule sequencing". In: *Nature Methods* 15.6 (2018), pp. 461–468. ISSN: 1548-7105. DOI: 10.1038/s41592-018-0001-7. URL: https://doi.org/10.1038/s41592-018-0001-7.

[12] Zamin Iqbal et al. "De novo assembly and genotyping of variants using colored de Bruijn graphs". In: *Nature Genetics* 44.2 (2012), pp. 226–232. ISSN: 1546-1718. DOI: 10.1038/ng.1028. URL: https://doi.org/10.1038/ng.1028.

[13] Huan Fan et al. "An assembly and alignment-free method of phylogeny reconstruction from next-generation sequencing data". In: *BMC Genomics* 16.1 (2015), p. 522. ISSN: 1471-2164. DOI: 10.1186/s12864-015-1647-5. URL: https://doi.org/10.1186/s12864-015-1647-5.

[14] Guillaume Bernard et al. "Alignment-free inference of hierarchical and reticulate phylogenomic relationships". In: *Briefings in Bioinformatics* 20.2 (giu. 2017), pp. 426–435. ISSN: 1477-4054. DOI: 10.1093/bib/bbx067. eprint: https://academic.oup.com/bib/article-pdf/20/2/426/28834015/bbx067.pdf. URL: https://doi.org/10.1093/bib/bbx067.

[15] Kamil S. Jaron, Jiří C. Moravec e Natália Martínková. "SigHunt: horizontal gene transfer finder optimized for eukaryotic genomes". In: *Bioinformatics* 30.8 (dic. 2013), pp. 1081–1086. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btt727. eprint: https://academic.oup.com/bioinformatics/article-pdf/30/8/1081/48922529/bioinformatics_30_8_1081.pdf. URL: https://doi.org/10.1093/bioinformatics/btt727.

[16] Guillaume Marçais e Carl Kingsford. "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers". In: *Bioinformatics* 27.6 (gen. 2011), pp. 764–770. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btr011. eprint: https://academic.oup.com/bioinformatics/article-pdf/27/6/764/48866141/bioinformatics_27_6_764.pdf. URL: https://doi.org/10.1093/bioinformatics/btr011.

[17] Ori Shalev e Nir Shavit. "Split-ordered lists: Lock-free extensible hash tables." In: *J. ACM* 53 (gen. 2006), pp. 379–405. DOI: 10.1145/872035.872049.

[18] Sebastian Deorowicz et al. “KMC 2: fast and resource-frugal k-mer counting”. In: *Bioinformatics* 31.10 (gen. 2015), pp. 1569–1576. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btv022. eprint: https://academic.oup.com/bioinformatics/article-pdf/31/10/1569/49012901/bioinformatics_31_10_1569.pdf. URL: https://doi.org/10.1093/bioinformatics/btv022.

[19] Lauris Kaplinski, Maarja Lepamets e Maido Remm. “GenomeTester4: a toolkit for performing basic set operations - union, intersection and complement on k-mer lists”. In: *Gigascience* 4 (2015), p. 58. DOI: 10.1186/s13742-015-0097-y. URL: https://doi.org/10.1186/s13742-015-0097-y.

[20] Stefan Kurtz et al. “A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes”. In: *BMC Genomics* 9 (2008), p. 517. DOI: 10.1186/1471-2164-9-517. URL: https://doi.org/10.1186/1471-2164-9-517.

[21] Abu AS Mamun, Sushmita Pal e Sanguthevar Rajasekaran. “KCMBT: a k-mer Counter based on Multiple Burst Trees”. In: *Bioinformatics* 32.18 (2016), pp. 2783–2790. DOI: 10.1093/bioinformatics/btw345. URL: https://doi.org/10.1093/bioinformatics/btw345.

[22] Mohamed Ibrahim Abouelhoda, Stefan Kurtz e Enno Ohlebusch. “Replacing suffix trees with enhanced suffix arrays”. In: *Journal of Discrete Algorithms* 2.1 (2004). The 9th International Symposium on String Processing and Information Retrieval, pp. 53–86. ISSN: 1570-8667. DOI: https://doi.org/10.1016/S1570-8667(03)00065-0. URL: https://www.sciencedirect.com/science/article/pii/S1570866703000650.

[23] Dana Randall, Andrew Guerrieri e Wei Jin. *Bloom Filters and Hashing*. CS 6550 Design and Analysis of Algorithms. 2006. URL: https://randall.math.gatech.edu/AlgsF09/bloomfilters.pdf.

[24] Rahul S. Roy, Debarati Bhattacharya e Alexander Schliep. “Turtle: identifying frequent k-mers with cache-efficient algorithms”. In: *Bioinformatics* 30.14 (2014). Epub 2014 Mar 10, pp. 1950–1957. DOI: 10.1093/bioinformatics/btu132.

[25] Páll Melsted e Jonathan K. Pritchard. “Efficient counting of k-mers in DNA sequences using a bloom filter”. In: *BMC Bioinformatics* 12.1 (2011), p. 333. ISSN: 1471-2105. DOI: 10.1186/1471-2105-12-333. URL: https://doi.org/10.1186/1471-2105-12-333.

[26] Andrei Broder e Michael Mitzenmacher. “Survey: Network Applications of Bloom Filters: A Survey.” In: *Internet Mathematics* 1 (nov. 2003). DOI: 10.1080/15427951.2004.10129096.

[27] Shahabeddin Geravand e Mahmood Ahmadi. “Bloom filter applications in network security: A state-of-the-art survey”. In: *Computer Networks* 57.18 (2013), pp. 4047–4064. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2013.09.003. URL: https://www.sciencedirect.com/science/article/pii/S1389128613003083.

[28] Kilian Weinberger et al. “Feature hashing for large scale multitask learning.” In: gen. 2009, p. 140.

[29] Qingpeng Zhang et al. "These Are Not the K-mers You Are Looking For: Efficient Online K-mer Counting Using a Probabilistic Data Structure". In: *PloS one* 9 (set. 2013). DOI: 10.1371/journal.pone.0101271.

[30] Páll Melsted e Bjarni V Halldórsson. "KmerStream: streaming algorithms for k-mer abundance estimation". In: *Bioinformatics (Oxford, England)* 30.24 (2014), pp. 3541–3547. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btu713. URL: https://academic.oup.com/bioinformatics/article-pdf/30/24/3541/48931408/bioinformatics_30_24_3541.pdf.

[31] Guillaume Rizk, Dominique Lavenier e Rayan Chikhi. "DSK: k-mer counting with very low memory usage". In: *Bioinformatics* 29.5 (gen. 2013), pp. 652–653. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btt020. eprint: https://academic.oup.com/bioinformatics/article-pdf/29/5/652/50335664/bioinformatics_29_5_652.pdf. URL: https://doi.org/10.1093/bioinformatics/btt020.