UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Assemblaggio di genomi tramite De Bruijn Graph: studio del sequenziamento, Bloom Filter e analisi dell'assemblatore Minia

PROFESSORI

Rosalba Zizza

Rocco Zaccagnino

Clelia De Felice

STUDENTI

Costantino Paciello

Matricola: 0522502045

Vittorio Guida

Matricola: 0522502008

Anno Accademico 2025-2026

Abstract

Il sequenziamento genomico di nuova generazione (NGS) ha rivoluzionato la biologia molecolare, generando volumi di dati che richiedono algoritmi efficienti per la ricostruzione dei genomi. Questo lavoro affronta il problema dell'assemblaggio genomico *de novo* attraverso l'analisi teorica e sperimentale di **Minia** (Chikhi e Rizk, 2013), un assemblatore che utilizza strutture dati probabilistiche per ridurre drasticamente il consumo di memoria.

L'approccio basato su De Bruijn Graph offre complessità computazionale lineare $O(n \cdot L)$, superiore ai metodi Overlap-Layout-Consensus ($O(n^2)$), ma richiede memoria proporzionale al numero di k-mer unici: circa 260 MB per *E. coli*, oltre 300 GB per il genoma umano con rappresentazioni esplicite.

Minia utilizza tre componenti per rappresentare implicitamente il grafo: (1) **Bloom Filter** (11 bit/k-mer) per memorizzare i k-mer solidi con assenza di falsi negativi; (2) **struttura cFP** (2 bit/k-mer) per correggere selettivamente solo i falsi positivi che alterano la topologia del grafo ($< 3\%$ del totale); (3) **marking structure** (0.5-4.5 bit/k-mer) per tracciare i nodi complessi. Questa architettura riduce la memoria di 59× rispetto agli assemblatori tradizionali (da 336 GB di ABySS a 5.7 GB per il genoma umano).

La validazione sperimentale su *E. coli* B str. REL606 (dataset SRR2584866, 586 MB paired-end Illumina) ha prodotto 4.17 Mbp (N50: 7,274 bp) coprendo il 90% del genoma con solo 2 misassemblies, in 1 minuto con 4.5 GB RAM (di cui $\sim$8 MB per le strutture dati). I risultati confermano che l'integrazione tra teoria algoritmica e strutture dati probabilistiche consente di assemblare genomi su hardware consumer, democratizzando l'accesso all'analisi genomica.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione e obiettivi del progetto

1.1 Contesto generale

Negli ultimi decenni la biologia ha vissuto una profonda trasformazione grazie allo sviluppo delle tecnologie di sequenziamento del DNA ad alta produttività (Next Generation Sequencing, NGS) [3]. Tali tecnologie hanno reso possibile la produzione di enormi quantità di dati genomici in tempi ridotti e a costi sempre più contenuti, aprendo nuove prospettive nello studio dei genomi, del microbioma, dell'evoluzione e delle malattie genetiche.

Tuttavia, il sequenziamento non fornisce direttamente l'intera sequenza genomica di un organismo. Le macchine di sequenziamento producono infatti milioni di frammenti di DNA, chiamati reads, che rappresentano solo porzioni parziali del genoma originale. Questi frammenti devono essere opportunamente analizzati, filtrati e combinati per ricostruire la sequenza completa. È in questo contesto che nasce il problema dell'assemblaggio del genoma [4].

L'assemblaggio genomico rappresenta una delle principali sfide della bioinformatica moderna [5], poiché richiede l'integrazione di conoscenze biologiche, algoritmi efficienti e strutture dati avanzate in grado di gestire grandi volumi di dati.

1.2 Il problema dell'assemblaggio genomico

L'obiettivo dell'assemblaggio è ricostruire una sequenza genomica il più possibile fedele a quella originale a partire dall'insieme delle reads ottenute dal sequenziamento [4]. Questo problema può essere assimilato a un puzzle estremamente complesso, in cui:

- il numero di pezzi è molto elevato;
- i pezzi possono contenere errori;
- molte porzioni del genoma risultano simili o identiche tra loro (ripetizioni).

Per affrontare queste difficoltà sono stati sviluppati diversi approcci algoritmici. Tra i più rilevanti vi sono [5]:

- l'approccio Overlap–Layout–Consensus (OLC), basato sulla ricerca delle sovrapposizioni tra reads;
- l'approccio basato sui De Bruijn Graph, che utilizza la scomposizione delle sequenze in sottostringhe di lunghezza fissa (k-mer).

In questo progetto l'attenzione è rivolta in particolare ai De Bruijn Graph [6], poiché essi rappresentano la base teorica di numerosi assemblatori moderni e consentono di affrontare in modo efficiente il problema dell'assemblaggio, soprattutto nel caso di grandi quantità di reads corte.

1.3 Motivazioni e importanza delle strutture dati

Uno degli aspetti critici nell'assemblaggio tramite De Bruijn Graph è la gestione dell'elevato numero di k-mer generati a partire dalle reads [6]. La memorizzazione esplicita di tutti i nodi e degli archi del grafo può richiedere quantità di memoria molto elevate, rendendo l'assemblaggio impraticabile su hardware standard.

Per risolvere questo problema, negli ultimi anni sono state introdotte strutture dati probabilistiche, come i Bloom Filter [7], che permettono di rappresentare insiemi molto grandi di elementi in modo compatto, accettando un compromesso controllato

in termini di accuratezza. L'integrazione di tali strutture dati negli algoritmi di assemblaggio ha portato allo sviluppo di tool innovativi, capaci di ridurre drasticamente l'uso di memoria mantenendo buone prestazioni.

1.4 Obiettivo del progetto

L'obiettivo principale di questo progetto è lo studio dell'intero processo che conduce dall'acquisizione dei dati di sequenziamento all'assemblaggio del genoma, con particolare attenzione agli aspetti algoritmici e computazionali. In particolare, il lavoro si propone di:

- fornire un'introduzione alle tecnologie di sequenziamento e ai formati dei dati prodotti;
- analizzare la qualità dei dati di sequenziamento e le tecniche di preprocessing;
- studiare il problema dell'assemblaggio genomico e l'approccio basato sui De Bruijn Graph;
- approfondire il funzionamento dei Bloom Filter come struttura dati fondamentale per la rappresentazione compatta dei grafi;
- analizzare in dettaglio un tool di assemblaggio basato su De Bruijn Graph, con particolare riferimento a Minia [2];
- valutare, ove possibile, il comportamento del tool tramite esecuzioni sperimentali su dati genomici reali.

Attraverso questo percorso, il progetto mira a evidenziare il ruolo centrale degli algoritmi e delle strutture dati nella bioinformatica moderna, mostrando come soluzioni teoricamente solide possano tradursi in strumenti pratici ed efficienti per l'analisi dei dati biologici.

CAPITOLO 2

Sequenziamento del DNA

2.1 Introduzione al sequenziamento

Il sequenziamento del DNA è il processo mediante il quale si determina l'ordine dei nucleotidi (A, C, G, T) che compongono una molecola di DNA. Le moderne tecnologie di sequenziamento, note come Next Generation Sequencing (NGS), hanno rivoluzionato la biologia molecolare permettendo di ottenere grandi quantità di dati genomici in tempi ridotti e con costi relativamente bassi.

A differenza dei metodi di prima generazione, come il sequenziamento di Sanger, le tecnologie NGS non producono una singola sequenza continua, ma generano milioni di reads, ovvero brevi frammenti di DNA che rappresentano porzioni casuali del genoma originale. La lunghezza e la qualità di tali frammenti dipendono fortemente dalla tecnologia di sequenziamento utilizzata [3].

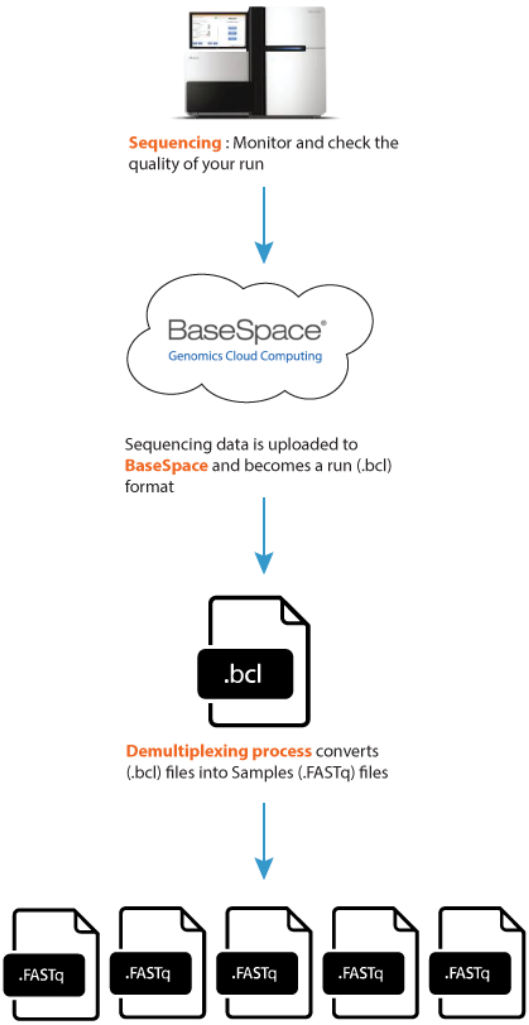


Figura 2.1: Schema generale del processo di NGS (dalla frammentazione alla generazione di reads).

2.2 Tecnologie di sequenziamento

2.2.1 Sequenziamento Illumina (short reads)

La tecnologia Illumina è attualmente una delle più diffuse nel sequenziamento genomico. Essa produce reads corte (tipicamente da 100 a 300 bp) caratterizzate da un'elevata accuratezza. Il principio di funzionamento si basa sul sequenziamento per sintesi, in cui nucleotidi fluorescenti vengono incorporati uno alla volta e rilevati tramite segnali ottici [3]. I principali vantaggi di Illumina sono:

- basso tasso di errore;
- elevata profondità di sequenziamento;
- ampia disponibilità di tool di analisi.

Tuttavia, la breve lunghezza delle reads rende più complessa la risoluzione di regioni ripetute durante l'assemblaggio del genoma [5].

2.2.2 Sequenziamento PacBio e Oxford Nanopore (long reads)

Le tecnologie PacBio e Oxford Nanopore appartengono alla categoria del long-read sequencing e producono reads molto più lunghe, che possono raggiungere decine o centinaia di migliaia di basi. Queste tecnologie consentono di attraversare regioni ripetute del genoma, semplificando alcune fasi dell'assemblaggio.

Il principale svantaggio è rappresentato da un tasso di errore più elevato rispetto alle tecnologie short-read, soprattutto nelle prime versioni di tali piattaforme. Nonostante ciò, i long reads risultano particolarmente utili in contesti di assemblaggio *de novo* [1] e vengono spesso combinati con dati Illumina in approcci ibridi [8].

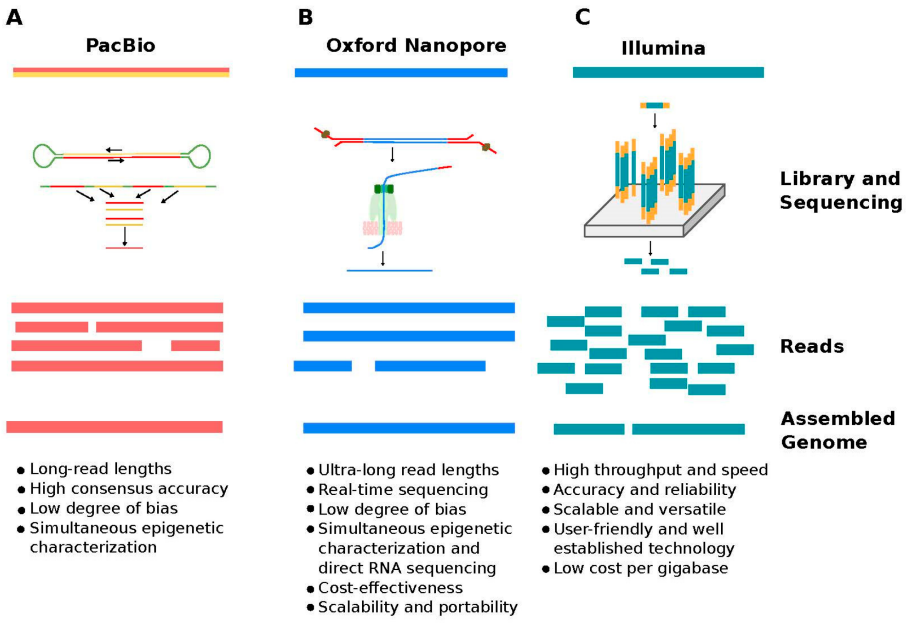


Figura 2.2: Confronto tra tecnologie di sequenziamento: lunghezza delle reads, accuratezza e applicazioni [1].

[1] Con il termine de novo genome assembly si indica la ricostruzione di un genoma a partire esclusivamente dalle reads di sequenziamento, in assenza di un genoma di riferimento.

2.3 Formati di output del sequenziamento

2.3.1 Formato FASTA

Il formato FASTA è uno dei formati più semplici e diffusi per la rappresentazione di sequenze biologiche. Ogni sequenza è preceduta da una riga di intestazione, identificata dal carattere >, seguita dalla sequenza stessa. Il formato FASTA non contiene informazioni sulla qualità delle basi e viene utilizzato principalmente per rappresentare sequenze già assemblate o di riferimento [6].

2.3.2 Formato FASTQ

Il formato FASTQ è lo standard per la rappresentazione delle reads prodotte dal sequenziamento NGS. Ogni read è descritta da quattro righe: intestazione, sequenza nucleotidica, separatore (+) e stringa degli score di qualità.

Gli score di qualità sono codificati tramite il sistema **Phred**, che associa a ciascuna base una probabilità di errore P secondo la formula:

$$Q = -10 \log_{10}(P) \tag{2.3.1}$$

dove Q è il quality score. Invertendo la formula, si ottiene:

$$P = 10^{-Q/10} \tag{2.3.2}$$

La Tabella 2.1 mostra la corrispondenza tra score e probabilità di errore.

Phred Score (Q)	Probabilità errore (P)	Accuratezza
10	0.1	90%
20	0.01	99%
30	0.001	99.9%
40	0.0001	99.99%

Tabella 2.1: Corrispondenza tra Phred score e accuratezza di base calling

Per una read Illumina tipica di 150 bp con Q medio di 30, ci aspettiamo statisticamente $\sim$0.15 errori per read. Su un dataset di 100 milioni di reads, questo si traduce in circa 15 milioni di basi potenzialmente errate, che genereranno *k-mer* spuri nel De Bruijn Graph [3].

2.4 Qualità del dato e preprocessing

La qualità delle reads ha un impatto diretto sull'accuratezza dell'assemblaggio genomico. Errori di sequenziamento, basi con basso score di qualità e contaminazioni possono introdurre artefatti nel grafo di assemblaggio, aumentando la complessità computazionale e degradando la qualità del risultato finale [5].

Per questo motivo, prima della fase di assemblaggio è necessario applicare procedure di preprocessing, che includono:

- rimozione di reads di bassa qualità;
- trimming delle estremità delle sequenze;
- filtraggio basato sugli score di qualità.

Tali operazioni permettono di ottenere un insieme di dati più affidabile, riducendo il rumore e migliorando le prestazioni degli algoritmi di assemblaggio basati su De Bruijn Graph [6].

2.4.1 Impatto degli errori sui k-mer

Un singolo errore di sequenziamento in una read di lunghezza L genera fino a k *k-mer* errati, dove k è la lunghezza del *k-mer*. Consideriamo l'esempio seguente con $k = 5$:

```
Read corretta (L=10):  ACGTACGTAC
Read con errore:       ACGTAgGTAC
                            ^
```

Questo singolo errore corrompe 5 diversi 5-mer:

- `TAcGT` → `TAgGT`
- `AcGTA` → `AgGTA`
- `cGTAC` → `gGTAC`
- `GTACg` → `GTAgg`
- `TACgT` → `TAgGT`

Dato un dataset di N reads di lunghezza media L con tasso di errore ε, il numero atteso di *k-mer* errati è approssimativamente:

$$\mathbb{E}[k\text{-mer errati}] \approx N \cdot L \cdot \varepsilon \cdot k \tag{2.4.1}$$

Per un dataset tipico con $N = 10^8$ reads, $L = 150$ bp, $\varepsilon = 0.001$, e $k = 31$:

$$\mathbb{E} \approx 10^8 \cdot 150 \cdot 0.001 \cdot 31 = 4.65 \times 10^8 \text{ } k\text{-mer spuri} \tag{2.4.2}$$

Questo calcolo evidenzia perché il preprocessing è critico: ridurre ε anche di un solo ordine di grandezza ha un impatto drammatico sulla complessità del grafo.

2.4.2 Strategie di quality filtering

Le principali strategie di preprocessing includono:

Trimming basato su soglia

Rimozione di basi con $Q < Q_{\text{min}}$ (tipicamente 20).

- **Vantaggi**: semplice e computazionalmente veloce
- **Svantaggi**: può ridurre eccessivamente la copertura in regioni naturalmente più rumorose

Trimming adattivo

Utilizzo di una *sliding window* che mantiene solo regioni con Q medio superiore a una soglia. Implementato in tool come Trimmomatic e fastp. Offre un miglior bilanciamento tra qualità e lunghezza delle reads.

Correzione degli errori

Algoritmi basati sulla frequenza dei *k-mer*:

- **Assunzione**: *k-mer* corretti appaiono più volte nel dataset; *k-mer* errati sono rari (appaiono una sola volta)
- **Strategia**: identificare *k-mer* a bassa frequenza e correggerli sostituendoli con *k-mer* simili ad alta frequenza
- Minia stesso incorpora strategie di error correction implicite durante la costruzione del grafo

La scelta della strategia dipende dalla copertura disponibile: con alta copertura ($> 50\times$) è preferibile essere stringenti nel filtering; con bassa copertura ($< 20\times$) serve un approccio più conservativo per non perdere informazione genomica.

2.5 Impatto della qualità sull'assemblaggio

La scelta della tecnologia di sequenziamento, del formato dei dati e delle tecniche di preprocessing influisce significativamente sul comportamento degli algoritmi di assemblaggio.

2.5.1 Confronto tecnologie

- **Illumina (short reads)**: reads corte (50–300 bp) e molto accurate ($Q > 30$) generano grafi complessi ma stabili. La complessità deriva dall'elevato numero di *k-mer* ma la loro affidabilità facilita la risoluzione di ambiguità.
- **PacBio / Oxford Nanopore (long reads)**: reads lunghe (10–100 kbp) con maggiore tasso di errore ($\varepsilon \sim 5$–15%) producono grafi più semplici (meno nodi) ma richiedono strategie di correzione degli errori più sofisticate [8].

2.5.2 Collegamento con gli assemblatori

Comprendere la natura dei dati di sequenziamento è fondamentale per interpretare le scelte progettuali degli assemblatori moderni:

- Gli assemblatori basati su De Bruijn Graph (come Minia) sono ottimizzati per short reads ad alta copertura
- La gestione dei *k-mer* errati richiede strutture dati efficienti sia in termini di memoria che di velocità di query
- L'uso di strutture probabilistiche (Bloom Filter) diventa essenziale quando il numero di *k-mer* unici supera i miliardi

Nei capitoli successivi vedremo come il De Bruijn Graph formalizza il problema dell'assemblaggio e come Minia utilizza i Bloom Filter per renderlo trattabile computazionalmente.

CAPITOLO 3

Il problema dell'assemblaggio genomico

3.1 Definizione del problema

L'assemblaggio genomico consiste nel ricostruire la sequenza originale del DNA di un organismo a partire da un insieme di frammenti, chiamati *reads*, ottenuti tramite tecnologie di sequenziamento. Poiché le tecnologie attuali non permettono di leggere l'intero genoma in un'unica operazione, il risultato del sequenziamento è un grande insieme di sequenze parziali, spesso sovrapposte e affette da errori.

Il problema può essere formalizzato come segue:

Definition 3.1.1 (Problema dell'assemblaggio)**.** *Dato un multiset $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ di stringhe (reads) sull'alfabeto $\Sigma = \{A, C, G, T\}$, trovare una o più stringhe S che contengano ogni r_i come sottostringa (o approssimativamente, considerando errori) e che rappresentino nel modo più accurato possibile il genoma originale.*

Dal punto di vista pratico, l'assemblaggio è una fase fondamentale in numerose applicazioni bioinformatiche:

- scoperta di nuovi genomi (*de novo* assembly)
- studio di organismi non modello

- analisi del microbioma
- rilevazione di varianti genomiche strutturali

3.2 Principali difficoltà dell'assemblaggio

L'assemblaggio genomico è un problema computazionalmente complesso a causa di diversi fattori.

3.2.1 Errori di sequenziamento

Le reads possono contenere errori dovuti alle limitazioni tecnologiche delle piattaforme di sequenziamento. Come discusso nel Capitolo 2, questi errori introducono *k-mer* errati che:

- frammentano il grafo di assemblaggio creando nodi e archi spuri
- complicano la ricostruzione corretta della sequenza
- aumentano il consumo di memoria

Un singolo errore può generare fino a k *k-mer* errati, moltiplicando l'effetto dell'errore originale.

3.2.2 Regioni ripetute

I genomi reali contengono numerose regioni ripetute che possono essere più lunghe delle reads stesse. Esempi comuni:

- **Tandem repeats**: ripetizioni consecutive (es. microsatelliti)
- **Trasposoni**: elementi mobili presenti in centinaia di copie
- **Geni duplicati**: famiglie geniche con alta similarità

Nel genoma umano, circa il 45% della sequenza è costituita da elementi ripetuti. Questo rende difficile stabilire l'ordine corretto dei frammenti, poiché diverse parti del genoma risultano indistinguibili a livello di sequenza.

Nel contesto dei De Bruijn Graph, le ripetizioni creano **cicli** e **percorsi ambigui**, rendendo non unica la soluzione del problema del cammino euleriano.

3.2.3 Copertura non uniforme

La distribuzione delle reads lungo il genoma non è uniforme a causa di:

- bias nel processo di frammentazione del DNA
- regioni difficili da amplificare (GC-rich o AT-rich)
- variazioni stocastiche nel campionamento

Alcune regioni possono essere sovra-rappresentate (alta copertura locale), mentre altre risultano scarsamente coperte o completamente assenti, creando **gap** nell'assemblaggio finale.

3.2.4 Complessità computazionale

Il numero di reads prodotte dai sequenziatori moderni può raggiungere centinaia di milioni. Per un genoma umano sequenziato a $30\times$ con reads da 150 bp:

$$N_{\text{reads}} = \frac{|G| \cdot C}{L} = \frac{3 \times 10^9 \cdot 30}{150} = 6 \times 10^8 \text{ reads} \tag{3.2.1}$$

dove $|G|$ è la lunghezza del genoma, C la copertura e L la lunghezza delle reads.

Questo volume di dati richiede algoritmi e strutture dati estremamente efficienti in termini di tempo e memoria.

3.3 Approcci algoritmici all'assemblaggio

Nel corso degli anni sono stati sviluppati due principali paradigmi per affrontare il problema dell'assemblaggio genomico.

3.3.1 Overlap–Layout–Consensus (OLC)

L'approccio OLC si articola in tre fasi:

1. **Overlap**: identificazione delle sovrapposizioni tra tutte le coppie di reads mediante algoritmi di allineamento locale
2. **Layout**: costruzione di un grafo delle sovrapposizioni in cui:
   - i nodi rappresentano le reads
   - gli archi rappresentano sovrapposizioni significative
3. **Consensus**: determinazione della sequenza finale attraverso un cammino hamiltoniano nel grafo

Complessità computazionale

Il passo di *overlap* richiede il confronto di tutte le coppie di reads:

$$\text{Confronti} = \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2) \tag{3.3.1}$$

Per $n = 10^8$ reads, questo corrisponde a $\sim 5 \times 10^{15}$ confronti, anche con euristiche per ridurre lo spazio di ricerca.

Vantaggi: adatto a reads lunghe (>10 kbp), mantiene l'informazione completa delle reads.

Svantaggi: complessità quadratica insostenibile per dataset di sequenziamento NGS con miliardi di reads.

3.3.2 De Bruijn Graph

L'approccio basato sui De Bruijn Graph rappresenta una soluzione alternativa più scalabile, particolarmente adatto per reads corte prodotte dalle tecnologie NGS.

Definizione formale

Dato un insieme di reads $\mathcal{R}$ e un parametro k (lunghezza del *k-mer*), il De Bruijn Graph $G = (V, E)$ è definito come:

- V: insieme di tutti i $(k-1)$-mer distinti presenti nelle reads
- E: esiste un arco orientato da u a v se esiste un *k-mer* nelle reads il cui prefisso di lunghezza $k-1$ è u e il suffisso di lunghezza $k-1$ è v

Ogni arco è quindi etichettato implicitamente da un *k-mer*.

Esempio

Consideriamo $k = 3$ e le reads:

```
R = {"ACGT", "CGTA"}
```

I *k-mer* estratti sono:

```
3-mer: {ACG, CGT, GTA}
```

I nodi ($(k-1)$-mer) sono:

```
V = {AC, CG, GT, TA}
```

Gli archi sono:

$$\text{AC} \to \text{CG} \quad \text{(da ACG)}$$
$$\text{CG} \to \text{GT} \quad \text{(da CGT)}$$
$$\text{GT} \to \text{TA} \quad \text{(da GTA)}$$

Un possibile cammino euleriano: AC $\to$ CG $\to$ GT $\to$ TA, che ricostruisce "ACGTA".

Complessità computazionale

La costruzione del De Bruijn Graph ha complessità:

$$O(n \cdot L) \tag{3.3.2}$$

dove n è il numero di reads e L la loro lunghezza. Questa complessità **lineare** nel numero di basi è drammaticamente migliore rispetto a OLC.

3.4 Dal grafo alla sequenza: il problema del cammino euleriano

3.4.1 Fondamenti teorici

L'assemblaggio in un De Bruijn Graph corrisponde a trovare un cammino che attraversi ogni **arco** esattamente una volta, noto come *cammino euleriano*.

Theorem 3.4.1 (Eulero, 1736)**.** *Un grafo orientato ammette un cammino euleriano se e solo se:*

1. *il grafo è debolmente connesso*
2. *ogni nodo ha in-degree = out-degree*[1]*, eccetto al più due nodi (sorgente e pozzo)*

Un algoritmo efficiente per trovare un cammino euleriano (algoritmo di Hierholzer) ha complessità $O(|E|)$, lineare nel numero di archi.

3.4.2 Il problema nei dati reali

In pratica, i genomi reali **non soddisfano** le condizioni del teorema a causa di:

- **Errori di sequenziamento**: creano nodi con in-degree $\neq$ out-degree
- **Regioni ripetute**: generano cicli e percorsi ambigui
- **Copertura non uniforme**: producono componenti disconnesse

Di conseguenza, gli assemblatori devono:

1. **Semplificare il grafo**: rimozione di artefatti
   - *tip clipping*: rimozione di rami corti terminali (errori)
   - *bubble removal*: eliminazione di percorsi alternativi dovuti a SNP[2] o errori
2. **Trovare cammini semi-euleriani**: percorsi che massimizzano la copertura degli archi
3. **Gestire ambiguità**: nelle ripetizioni, output di contigs multipli invece di una sequenza unica

[1] In ogni nodo deve esserci lo stesso numero di entrate e uscite.

[2] Single Nucleotide Polymorphism: variazione di una singola base azotata nel codice del DNA che avviene in una posizione specifica.

3.5 Vantaggi e limiti dei De Bruijn Graph

3.5.1 Vantaggi

- **Scalabilità**: complessità lineare $O(n \cdot L)$ nella costruzione
- **Efficienza**: riduzione drammatica rispetto a OLC
- **Adattabilità**: particolarmente efficace per short reads
- **Gestione errori**: possibilità di filtrare *k-mer* a bassa frequenza

3.5.2 Limiti

- **Dipendenza da** k:
  - k piccolo $\rightarrow$ grafo complesso, molte ambiguità
  - k grande $\rightarrow$ grafo frammentato, perde sovrapposizioni
- **Sensibilità agli errori**: ogni errore genera k *k-mer* errati
- **Consumo di memoria**: rappresentazione esplicita del grafo richiede memoria proporzionale al numero di *k-mer* unici

3.5.3 Il problema della memoria: quantificazione

Consideriamo un genoma di *E. coli* (4.6×10^6 bp) sequenziato a $50\times$ di copertura con reads da 150 bp e $k = 31$.

Numero di *k-mer*

Numero di reads:

$$N_{\text{reads}} = \frac{G \times C}{L} = \frac{4.6 \times 10^6 \times 50}{150} \approx 1.53 \times 10^6 \tag{3.5.1}$$

k-mer totali nelle reads (con ridondanza):

$$N_{\text{kmers}}^{\text{tot}} = N_{\text{reads}} \times (L - k + 1) = 1.53 \times 10^6 \times 120 \approx 1.84 \times 10^8 \tag{3.5.2}$$

k-mer unici stimati (considerando errori e ripetizioni): $\sim 5 \times 10^6$

Memoria richiesta

Rappresentazione esplicita dei *k-mer*:

- Sequenza del *k-mer*: 8 bytes (codifica a 2 bit/base più padding)
- Conteggio: 4 bytes
- Totale per *k-mer*: 12 bytes

$$M_{\text{kmers}} = 5 \times 10^6 \times 12 \text{ bytes} = 60 \text{ MB} \tag{3.5.3}$$

Rappresentazione del grafo (archi):

- Ogni arco: 2 puntatori (16 bytes) + metadata (conteggio, flag)
- Archi stimati: $\sim 10 \times 10^6$

$$M_{\text{grafo}} = 10 \times 10^6 \times 20 \text{ bytes} = 200 \text{ MB} \tag{3.5.4}$$

Memoria totale:

$$M_{\text{totale}} = M_{\text{kmers}} + M_{\text{grafo}} \approx 260 \text{ MB} \tag{3.5.5}$$

3.5.4 Implicazioni

Sebbene per un genoma batterico come *E. coli* la memoria richiesta ($\sim$260 MB) sia gestibile anche su hardware standard, questo esempio evidenzia come il consumo di memoria cresca linearmente con la dimensione del genoma e la copertura di sequenziamento.

Per genomi più complessi:

- Genoma umano (3×10^9 bp): $\sim$104 GB
- Genomi vegetali complessi (10^{10} bp): $\sim$350 GB
- Metagenomica (complessità 10^{11} bp): $\sim$3.5 TB

Questo dimostra come l'approccio esplicito diventi rapidamente impraticabile per dataset reali di grandi dimensioni, motivando fortemente lo sviluppo di strutture dati compatte.

La soluzione: strutture dati probabilistiche come i **Bloom Filter**, che permettono di ridurre il consumo di memoria da centinaia di MB/GB a pochi MB/GB (riduzione di $\sim$90–95%), rendendo l'assemblaggio scalabile anche per genomi complessi su hardware consumer. Questa è l'innovazione chiave di Minia, che verrà analizzata in dettaglio nel Capitolo 4.

CAPITOLO 4

Bloom Filter: una struttura dati fondamentale

4.1 Motivazione

Come discusso nel capitolo precedente, l'approccio basato sui **De Bruijn Graph** consente di affrontare il problema del *de novo genome assembly* in modo scalabile, in particolare quando si lavora con dati di sequenziamento di nuova generazione (NGS). Tuttavia, tale approccio presenta un limite significativo quando il grafo viene rappresentato in maniera esplicita: **l'elevato consumo di memoria**.

La costruzione di un De Bruijn Graph richiede infatti la gestione di un numero estremamente elevato di **k-mer distinti**, che crescono rapidamente all'aumentare della dimensione del genoma, della profondità di sequenziamento e della presenza di errori nelle reads. Nei dataset genomici reali, il numero di k-mer può raggiungere ordini di grandezza tali da rendere impraticabile l'utilizzo di strutture dati classiche, come tabelle hash, per la loro memorizzazione.

Questo problema di scalabilità ha motivato lo sviluppo di **strutture dati compatte**, in grado di rappresentare grandi insiemi di elementi riducendo drasticamente l'occupazione di memoria. Tra queste, i **Bloom Filter**, introdotti da Bloom nel 1970 [7], rappresentano una soluzione particolarmente efficace per la gestione dei k-mer nei contesti bioinformatici moderni.

4.2 Che cos'è Bloom Filter

Un Bloom Filter è una struttura dati probabilistica progettata per rispondere in modo efficiente a interrogazioni di appartenenza a un insieme (membership query), minimizzando il consumo di memoria[7].

A differenza delle strutture dati deterministiche, il Bloom Filter non memorizza esplicitamente gli elementi dell'insieme, ma ne mantiene una rappresentazione indiretta che può introdurre una probabilità controllata di errore.

In particolare, un Bloom Filter consente di stabilire se un elemento **non appartiene sicuramente** all'insieme oppure **appartiene probabilmente** all'insieme.

L'errore possibile è limitato ai **falsi positivi**, mentre i **falsi negativi sono assenti**, proprietà che risulta fondamentale in molte applicazioni bioinformatiche.

4.3 Funzionamento

Dal punto di vista implementativo, un Bloom Filter è costituito da [7]:

- un array di bit di dimensione fissa m, inizialmente impostato a zero;
- un insieme di k funzioni hash indipendenti. Ogni funzione hash mappa un elemento a un indice nell'array di bit.

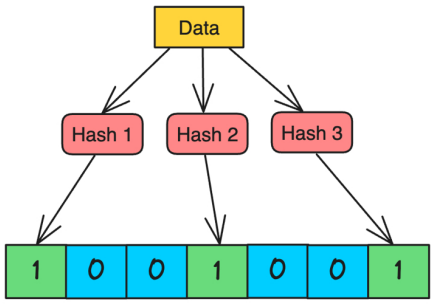


Figura 4.1: Struttura Bloom Filter.

4.3.1 Inserimento di un elemento

Quando un elemento viene inserito nel Bloom Filter, esso viene processato da ciascuna funzione hash, che restituisce k posizioni all'interno del bit array. I bit corrispondenti vengono impostati a 1.

4.3.2 Verifica di appartenenza

Per verificare se un elemento è presente nell'insieme è necessario passarlo nuovamente alle funzioni hash ottenendo nuovamente le k posizioni; se tutti i bit in queste posizioni sono impostati a 1, l'elemento è considerato presente con una certa probabilità. Se un bit in queste posizioni è 0, l'elemento non fa sicuramente parte dell'insieme.

4.3.3 Implementazione concettuale

Dal punto di vista astratto, il funzionamento di un Bloom Filter può essere descritto tramite semplici operazioni di inserimento e interrogazione.

Pseudocodice

```
Procedure ADD(item)
    For each hash_function in HashFunctions do
        index ← hash_function(item) mod m
        BitArray[index] ← 1
    End For
End Procedure
```

```
Procedure MIGHT_CONTAIN(item)
    For each hash_function in HashFunctions do
        index ← hash_function(item) mod m
        If BitArray[index] = 0 then
            Return FALSE
        End If
    End For
    Return TRUE
End Procedure
```

Il seguente pseudocodice descrive il funzionamento fondamentale di un Bloom Filter. La procedura `ADD` inserisce un elemento impostando a 1 i bit corrispondenti agli indici generati dalle funzioni hash, mentre la procedura `MIGHT_CONTAIN` verifica la possibile appartenenza di un elemento controllando che tutte le posizioni risultino impostate.

4.3.4 Complessità:

Dal punto di vista computazionale:

- le operazioni di inserimento e interrogazione hanno complessità O(k);
- lo spazio occupato è proporzionale alla dimensione del bit array ed è indipendente dal numero di elementi inseriti.

Rispetto alle strutture classiche, il Bloom Filter offre quindi un compromesso vantaggioso tra spazio occupato ed efficienza temporale.

4.4 Proprietà teoriche

4.4.1 Falsi positivi e assenza di falsi negativi

I Bloom Filter possono generare **falsi positivi**, ovvero possono indicare erroneamente che un elemento è presente nell'insieme dei dati quando non lo è.

Esempio: si consideri il caso in cui, dando in input alle funzioni hash una chiave non presente nell'insieme dei dati, queste restituiscono 1 in posizioni già impostate da altri elementi; in tal caso, il Bloom Filter segnala erroneamente che l'elemento è presente nell'insieme.

La probabilità di ottenere un falso positivo è una conseguenza diretta della natura probabilistica del Bloom Filter e dipende dalla dimensione del bit array m, dal numero di funzioni hash k e dal numero di elementi inseriti n. Assumendo funzioni hash ideali e uniformemente distribuite, la probabilità di falso positivo può essere stimata analizzando il comportamento dei bit nel Bloom Filter durante la fase di inserimento. Dopo l'inserimento di n elementi, ognuno dei quali viene mappato tramite k funzioni

hash su un array di m bit, la probabilità che un singolo bit rimanga impostato a zero è approssimativamente pari a:

$$P(\text{bit} = 0) \approx e^{-\frac{kn}{m}}.$$

Di conseguenza, la probabilità che un bit sia impostato a uno è:

$$P(\text{bit} = 1) = 1 - e^{-\frac{kn}{m}}.$$

Un falso positivo si verifica quando tutte le k posizioni associate a un elemento non presente nell'insieme risultano ugualmente impostate a uno. La probabilità complessiva di falso positivo è quindi [7]:

$$p = \left(1 - e^{-\frac{kn}{m}}\right)^k.$$

Tale probabilità può essere controllata scegliendo opportunamente i parametri m e k in funzione del numero atteso di elementi n.

Al contrario, un Bloom Filter non produce mai falsi negativi, garantendo che un elemento dichiarato assente non sia mai stato inserito nel filtro [7].

Questa asimmetria dell'errore rende il Bloom Filter particolarmente adatto all'uso come struttura di filtraggio preliminare nei sistemi di analisi genomica.

4.5 Limiti

4.5.1 Nessun supporto per le eliminazioni

I Bloom Filter standard, non supportano l'eliminazione di elementi. Una volta che un bit viene impostato a 1 durante l'inserimento di un elemento, non può più essere annullato, poiché altri elementi potrebbero far affidamento sul quel bit. L'utilizzo di contatori al posto di semplici bit permette di eliminare elementi a discapito della memoria; questo è il caso dei **Counting Bloom Filter**.

4.5.2 Limitati alle membership query

I Bloom Filter sono progettati per rispondere alle membership query. Non forniscono informazioni sugli elementi effettivi dell'insieme dei dati, né supportano query o operazioni complesse che vanno oltre i controlli di appartenenza.

4.5.3 Vulnerabile alle collisioni hash

Una collisione hash si verifica quando dati differenti input a una funzione hash essa produce lo stesso output. La probabilità di collisione aumenta con il numero di elementi presenti nel Bloom Filter, perché più elementi possono impostare o basarsi sugli stessi bit. Con l'aumento delle collisione l'efficacia del Bloom Filter diminuisce drasticamente. L'utilizzo di funzioni hash aggiuntive può aiutare a ridurre le collisioni; tuttavia, l'incremento del numero di funzioni hash aumenta anche la complessità computazionale e i requisiti di memoria.

4.6 Confronto con strutture dati classiche

In questa sezione viene presentato un confronto tra i Bloom Filter e alcune strutture dati classiche comunemente utilizzate per la gestione di insiemi di dati.

- **Tabelle hash**: consentono interrogazioni esatte e supportano operazioni di inserimento ed eliminazione, ma richiedono una quantità di memoria elevata, soprattutto quando il numero di elementi è molto grande. Nei contesti genomici, la memorizzazione esplicita di miliardi di k-mer risulta spesso impraticabile.

- **Array o liste**: sono semplici da implementare, ma richiedono ricerche lineari per le interrogazioni di appartenenza e non offrono alcun meccanismo di compressione, risultando inefficaci sia in termini di spazio sia di tempo per dataset genomici di grandi dimensioni.

- **Bloom Filter**: rispetto a strutture deterministiche tradizionali, i Bloom Filter consentono di ridurre drasticamente il consumo di memoria, accettando una probabilità di errore controllata, spesso trascurabile nel contesto delle analisi genomiche.

4.7 Bloom Filter in bioinformatica

L'uso dei Bloom Filter in bioinformatica è stato ampiamente studiato e consolidato, come evidenziato nel lavoro di Marchet et al. (2018) [9], che analizza diverse strutture dati basate su Bloom Filter per la gestione di dati genomici su larga scala.

In particolare, tali strutture vengono impiegate per:

- k-mer counting;
- membership query su grandi collezioni di sequenze;
- supporto alla costruzione di De Bruijn Graph compatti.

Di conseguenza, nei contesti bioinformatici moderni, in cui la scalabilità e l'efficienza in memoria rappresentano requisiti fondamentali, i Bloom Filter rappresentano una soluzione ideale per la gestione compatta dei k-mer nei moderni algoritmi di assemblaggio genomico. Nel capitolo seguente verrà analizzato un tool di assemblaggio che fa uso diretto di tali strutture dati: Minia.

CAPITOLO 5

Studio del tool di assemblaggio Minia

Come illustrato nel capitolo precedente, i Bloom Filter rappresentano una struttura dati probabilistica estremamente efficiente per la gestione di grandi insiemi di k-mer, grazie al ridotto consumo di memoria e all'assenza di falsi negativi. Tali proprietà risultano particolarmente rilevanti nel contesto dell'assemblaggio genomico, dove la costruzione esplicita dei De Bruijn Graph comporta requisiti di memoria spesso proibitivi.

In questo capitolo viene analizzato **Minia**, un assemblatore *de novo* [1] basato su De Bruijn Graph, che utilizza i Bloom Filter come struttura dati centrale per rappresentare l'insieme dei k-mer. Minia implementa l'approccio proposto da Chikhi e Rizk (2013) [2], per ottenere una rappresentazione del grafo **esatta** e **scalabile**, mantenendo al contempo un basso consumo di memoria.

In particolare, Minia affronta esplicitamente il problema dei falsi positivi introdotti dai Bloom Filter, proponendo una rappresentazione probabilistica del grafo accompagnata da strutture ausiliarie che garantiscono la correttezza dell'assemblaggio.

[1] Un assemblatore *de novo* ricostruisce un genoma utilizzando esclusivamente le reads di sequenziamento, senza fare affidamento su un genoma di riferimento.

L'obiettivo del capitolo è duplice: da un lato descrivere l'algoritmo di base utilizzato da Minia per la costruzione e l'attraversamento del De Bruijn Graph, dall'altro analizzare come i concetti teorici introdotti nel capitolo precedente trovino concreta applicazione nell'implementazione del tool.

5.1 Introduzione e contestualizzazione

Minia è un assemblatore *de novo* progettato per la ricostruzione di genomi a partire da dati di sequenziamento di nuova generazione (NGS). Il tool si basa sul modello dei De Bruijn Graph e si distingue per l'utilizzo di una rappresentazione compatta del grafo, ottenuta attraverso l'impiego dei Bloom Filter.

A partire da reads di sequenziamento in input, Minia ricostruisce porzioni del genoma sotto forma di **contig**, generalmente forniti in formato FASTA, ottenuti tramite l'attraversamento dei cammini non ambigui del De Bruijn Graph.

L'obiettivo principale di Minia è rendere l'assemblaggio genomico accessibile anche su macchine con risorse di memoria limitate, senza compromettere la correttezza dell'assemblaggio[2].

5.1.1 Contestualizzazione teorica

Per comprendere appieno l'innovazione rappresentata da Minia, è necessario prima stabilire quali siano i limiti teorici fondamentali per la rappresentazione di un De Bruijn Graph. La teoria dell'informazione fornisce un limite inferiore, noto come *self-information*, che indica la quantità minima di bit necessaria per codificare esattamente un insieme di k-mer.

Come vedremo nella sezione successiva, Minia riesce a ottenere una rappresentazione che richiede meno memoria del limite informazionale teorico. Questo risultato apparentemente paradossale è possibile grazie a un'importante osservazione: per l'assemblaggio genomico non è necessario supportare query di membership arbitrarie, ma solo operazioni di attraversamento del grafo.

Questa distinzione concettuale è alla base dell'intera architettura di Minia e verrà approfondita analizzando dapprima il lower bound teorico, per poi mostrare come

la struttura dati proposta da Chikhi e Rizk si collochi al di sotto di tale limite pur garantendo la correttezza dell'assemblaggio.

5.2 Lower bound informazionale per la rappresentazione del grafo

Prima di analizzare in dettaglio l'architettura di Minia, è fondamentale stabilire il limite teorico minimo di memoria necessario per rappresentare esattamente un De Bruijn Graph. Questo limite fornisce un punto di riferimento per valutare l'efficienza delle diverse implementazioni.

5.2.1 Self-information teorica

Conway e Bromage [10] hanno osservato che la *self-information* degli archi rappresenta un limite inferiore teorico per la codifica esatta di un De Bruijn Graph. Per un grafo con $|E|$ archi, ognuno definito da una sequenza di lunghezza $k+1$, il numero minimo di bit necessari è:

$$I_{\text{archi}} = \log_2 \binom{4^{k+1}}{|E|} \text{ bits}$$

Nell'articolo di Minia [2], gli autori adottano una definizione node-centric[2] del De Bruijn Graph, considerando i k-mer come nodi anziché come archi. In questo caso, il limite inferiore derivato dalla *self-information* dei nodi è:

$$I_{\text{nodi}} = \log_2 \binom{4^k}{|N|} \text{ bits}$$

dove $|N|$ è il numero di nodi (k-mer distinti) nel grafo.

Per valori grandi di k e $|N|$, questa quantità può essere approssimata utilizzando l'approssimazione di *Stirling*:

$$I_{\text{nodi}} \approx |N| \cdot \log_2 \left(\frac{4^k}{|N|} \right) \text{ bits}$$

Dividendo per il numero di k-mer, si ottiene il costo per nodo:

[2] si riferisce alla modalità con cui i nodi e gli archi vengono mappate all'interno del grafo.

$$\frac{I_{\text{nodi}}}{|N|} \approx \log_2 \left(\frac{4^k}{|N|} \right) \text{ bit per k-mer}$$

5.2.2 Calcolo per il genoma umano

Consideriamo il caso del genoma umano con i seguenti parametri [2]:

- Numero di k-mer distinti: $|N| \approx 2.4 \times 10^9$
- Lunghezza dei k-mer: $k = 27$

Applicando la formula precedente:

$$I_{\text{nodi}} \approx 2.4 \times 10^9 \times \log_2 \left(\frac{4^{27}}{2.4 \times 10^9} \right)$$

Calcolando:

$$4^{27} = 2^{54} \approx 1.8 \times 10^{16}$$
$$\frac{4^{27}}{2.4 \times 10^9} \approx 7.5 \times 10^6$$
$$\log_2(7.5 \times 10^6) \approx 22.8$$

Quindi:

$$I_{\text{nodi}} \approx 2.4 \times 10^9 \times 22.8 \text{ bit} \approx 5.5 \times 10^{10} \text{ bit} \approx 6.8 \text{ GB}$$

Questo corrisponde a circa **24 bit per k-mer** [2].

5.2.3 Perché Minia supera il lower bound

Come vedremo nelle sezioni successive, Minia utilizza una rappresentazione che richiede circa **13.2 bit per k-mer** per il genoma umano, un valore significativamente inferiore ai 24 bit teorici della *self-information*.

Questo risultato apparentemente sorprendente è possibile perché la struttura di Minia **non memorizza l'insieme esatto dei k-mer in memoria**. In particolare [2]:

- Il Bloom Filter non supporta efficientemente query di membership per k-mer arbitrari, ma solo per k-mer che sono estensioni di nodi già noti

- La struttura è ottimizzata per le operazioni di *traversal* del grafo (enumerazione dei vicini di un nodo), non per la verifica casuale di appartenenza
- L'insieme dei k-mer viene rappresentato implicitamente attraverso il Bloom Filter, accettando una certa probabilità di falsi positivi che vengono poi corretti dalla struttura cFP (*critical false positives*) [2].

Questo trade-off è perfettamente accettabile per l'assemblaggio genomico, dove l'operazione fondamentale è l'attraversamento sistematico del grafo per generare i contigs, non la verifica dell'appartenenza di k-mer arbitrari.

Il lower bound informazionale rimane quindi valido per strutture che devono supportare membership query generali, ma non si applica a rappresentazioni specializzate come quella di Minia, ottimizzate per un insieme ristretto di operazioni.

Nelle sezioni seguenti analizzeremo in dettaglio come Minia implementa questa rappresentazione efficiente attraverso la combinazione di **Bloom Filter**, **struttura dei falsi positivi critici (cFP)** e **struttura di marcatura**.

5.3 Il grafo di De Bruijn probabilistico

5.3.1 Definizione formale

Il concetto di **grafo di De Bruijn probabilistico** è stato introdotto da Pell et al. (2011) [11] come alternativa alle rappresentazioni esplicite del grafo. L'idea fondamentale consiste nel memorizzare l'insieme dei nodi del grafo (i k-mer) all'interno di un Bloom Filter, anziché in una struttura dati deterministica come una hash table [2].

Formalmente, dato un insieme S di k-mer estratti dalle reads di sequenziamento, il grafo probabilistico $G_P = (V_P, E_P)$ è definito come segue:

- **Nodi** (V_P): l'insieme dei k-mer per cui il Bloom Filter restituisce una risposta positiva alla query di membership
- **Archi** (E_P): dedotti implicitamente verificando l'appartenenza delle estensioni di ciascun nodo

Più precisamente, un k-mer v appartiene a V_P se e solo se:

$$\text{BloomFilter.contains}(v) = \texttt{true}$$

La presenza degli archi non viene memorizzata esplicitamente, ma viene determinata dinamicamente durante l'attraversamento del grafo [2].

5.3.2 Deduzione implicita degli archi

Per determinare i vicini di un nodo v, Minia utilizza il concetto di **estensione** di un k-mer. Data la natura del De Bruijn Graph, un nodo v può avere al massimo:

- 4 vicini in uscita (estensioni a destra)
- 4 vicini in entrata (estensioni a sinistra)

Estensioni a destra. Sia $v = b_1 b_2 \cdots b_k$ un k-mer, dove $b_i \in \{A, C, G, T\}$. Le estensioni a destra di v sono i 4 k-mer ottenuti concatenando il suffisso di lunghezza $k-1$ di v con ciascuno dei 4 nucleotidi possibili:

$$\text{ext}_{\text{right}}(v) = \{b_2 b_3 \cdots b_k \cdot x \mid x \in \{A, C, G, T\}\}$$

Un arco $v \to w$ esiste in G_P se e solo se:

1. w è un'estensione a destra di v
2. $\text{BloomFilter.contains}(w) = \texttt{true}$

Estensioni a sinistra. Analogamente, le estensioni a sinistra sono definite come:

$$\text{ext}_{\text{left}}(v) = \{x \cdot b_1 b_2 \cdots b_{k-1} \mid x \in \{A, C, G, T\}\}$$

Un arco $u \to v$ esiste se v è un'estensione a sinistra di u verificata positivamente dal Bloom Filter.

Esempio. Consideriamo il k-mer $v = \texttt{ACGT}$ (con $k = 4$). Le sue estensioni a destra sono:

$$\begin{aligned}\text{ext}_{\text{right}}(\texttt{ACGT}) = \{&\texttt{CGTA},\\ &\texttt{CGTC},\\ &\texttt{CGTG},\\ &\texttt{CGTT}\}\end{aligned}$$

Minia interroga il Bloom Filter per ciascuna di queste 4 estensioni. Se, ad esempio, solo `CGTA` e `CGTC` risultano presenti, allora il nodo `ACGT` avrà due archi uscenti nel grafo probabilistico.

5.3.3 Over-approssimazione del grafo

Poiché i Bloom Filter possono produrre falsi positivi ma mai falsi negativi (come discusso nel Capitolo 4), il grafo probabilistico G_P è una **sovra-approssimazione** del grafo esatto $G = (V, E)$, dove $V = S$ è l'insieme dei k-mer reali:

$$V \subseteq V_P \quad \text{e} \quad E \subseteq E_P$$

Questa proprietà garantisce che [2]:

1. **Completezza**: tutti i k-mer reali sono rappresentati nel grafo probabilistico
$$\forall v \in S : v \in V_P$$
2. **Archi reali preservati**: tutti gli archi del grafo esatto sono presenti nel grafo probabilistico
$$\forall (u, v) \in E : (u, v) \in E_P$$
3. **Possibili nodi/archi spuri**: possono esistere k-mer e archi falsi dovuti ai falsi positivi del Bloom Filter
$$\exists w \in V_P \setminus V : \text{BloomFilter.contains}(w) = \texttt{true} \text{ ma } w \notin S$$

La Figura 5.1, riportata nelle sezioni successive illustra questa situazione: i nodi circolari pieni rappresentano i k-mer reali (V), mentre i nodi tratteggiati rappresentano falsi positivi introdotti dal Bloom Filter.

5.3.4 Impatto quantitativo dei falsi positivi

La presenza di falsi positivi nel grafo probabilistico ha conseguenze dirette sulla sua topologia e sulla complessità computazionale dell'assemblaggio.

Numero atteso di falsi positivi. Sia F il tasso di falsi positivi del Bloom Filter e $n = |S|$ il numero di k-mer reali. Ogni k-mer ha esattamente 8 possibili estensioni (4 a destra e 4 a sinistra). Il numero atteso di estensioni spurie è:

$$\mathbb{E}[\text{estensioni false}] = 8 \cdot n \cdot F$$

Esempio: genoma umano. Per un genoma umano con:

- $n = 2.4 \times 10^9$ k-mer solidi
- $F = 0.01$ (1% di falsi positivi)

Il numero atteso di estensioni false è:

$$\mathbb{E}[\text{est. false}] = 8 \times 2.4 \times 10^9 \times 0.01 = 1.92 \times 10^8 \approx 192 \text{ milioni}$$

Questo corrisponde a circa l'8% del numero totale di k-mer reali, un valore significativo che altererebbe drasticamente la topologia del grafo.

Conseguenze topologiche. I falsi positivi introducono:

- **Nodi complessi artificiali**: k-mer reali che nel grafo esatto hanno in-degree e out-degree pari a 1 (fanno parte di un cammino lineare) possono acquisire vicini spuri, diventando nodi di branching
- **Percorsi alternativi errati**: le ramificazioni spurie creano ambiguità durante l'attraversamento, portando a una frammentazione prematura dei contigs
- **Inflazione della struttura di marcatura**: poiché ogni nodo complesso deve essere memorizzato nella struttura di marcatura (come vedremo nella sezioni successive), i falsi positivi che creano branching aumentano significativamente il consumo di memoria

5.3.5 Necessità della struttura cFP.

Senza meccanismi di correzione, un grafo probabilistico con tasso di falsi positivi anche solo dell'1% produrrebbe assembly fortemente degradati. Pell et al. [11] hanno osservato che per tassi di falsi positivi superiori al 18% (corrispondenti a Bloom Filter con meno di 4 bit/k-mer), la struttura globale del grafo viene compromessa, con nodi distanti che risultano erroneamente connessi.

Per questo motivo, Minia introduce la struttura dei **falsi positivi critici** (cFP), che permette di identificare ed eliminare esattamente quei falsi positivi che alterano la topologia del grafo. Come dimostrato nell'articolo originale [2], questa correzione è sufficiente per ottenere assembly identici a quelli prodotti da rappresentazioni deterministiche del grafo, pur mantenendo il vantaggio in termini di memoria offerto dai Bloom Filter.

La sezione successiva descrive in dettaglio la definizione, la costruzione e il costo in memoria della struttura cFP, che rappresenta il contributo centrale dell'approccio proposto da Chikhi e Rizk.

5.4 Rimozione dei falsi positivi critici

Come evidenziato nella sezione precedente, i falsi positivi introdotti dal Bloom Filter possono alterare significativamente la topologia del De Bruijn Graph, creando ramificazioni spurie e aumentando artificialmente il numero di nodi complessi. Tuttavia, non tutti i falsi positivi hanno lo stesso impatto sull'assemblaggio.

Minia introduce il concetto di **falso positivo critico** (*critical False Positive*, cFP), che identifica esattamente quei k-mer falsi che sono direttamente adiacenti a nodi reali del grafo e che quindi interferiscono con l'attraversamento [2]. L'intuizione chiave è che solo questi falsi positivi devono essere memorizzati esplicitamente per correggere la topologia del grafo.

5.4.1 Definizione formale di critical False Positives (cFP)

Per formalizzare il concetto di falso positivo critico, introduciamo le seguenti notazioni [2]:

- S: insieme dei k-mer reali (solid k-mers), ovvero i nodi del grafo esatto estratti dalle reads dopo il filtraggio per abbondanza[3]
- E: insieme di tutte le estensioni (destra e sinistra) dei k-mer in S
- P: insieme degli elementi di E per cui il Bloom Filter restituisce una risposta positiva

Formalmente:

$$E = \bigcup_{v \in S} \left(\text{ext}_{\text{left}}(v) \cup \text{ext}_{\text{right}}(v) \right)$$

$$P = \{ w \in E \mid \text{BloomFilter.contains}(w) = \texttt{true} \}$$

L'insieme dei **falsi positivi critici** è definito come:

$$\boxed{\text{cFP} = P \setminus S}$$

In altre parole, cFP contiene esattamente quei k-mer che:

1. Sono estensioni dirette di almeno un k-mer reale ($w \in E$)
2. Vengono erroneamente riportati come presenti dal Bloom Filter ($w \in P$)
3. Non appartengono all'insieme reale dei k-mer del genoma ($w \notin S$)

Osservazione chiave. La definizione di cFP è strettamente legata al concetto di **raggiungibilità durante il traversal**. Durante l'attraversamento del grafo, Minia parte sempre da un nodo reale $v \in S$ e interroga il Bloom Filter solo per le estensioni di v. Questo significa che:

- I falsi positivi in E (cioè cFP) **verranno interrogati** durante il traversal e devono essere identificati

[3]Minia, applica un filtraggio basato sull'abbondanza dei k-mer, estraendo solo quelli che compaiono al di sopra di una soglia prefissata. Questo meccanismo, implementato nelle fasi iniziali della pipeline, consente di ridurre l'impatto degli errori di sequenziamento sulla struttura del grafo.

- I falsi positivi non in E (cioè k-mer casuali che il Bloom Filter riporta erroneamente come presenti) **non verranno mai interrogati** e quindi possono essere ignorati

Questa osservazione è fondamentale per l'efficienza della struttura: non è necessario memorizzare *tutti* i falsi positivi del Bloom Filter, ma solo un sottoinsieme molto piccolo [2].

5.4.2 Esempio illustrativo

La Figura 5.1 illustra visivamente la distinzione tra diversi tipi di k-mer nel grafo probabilistico.

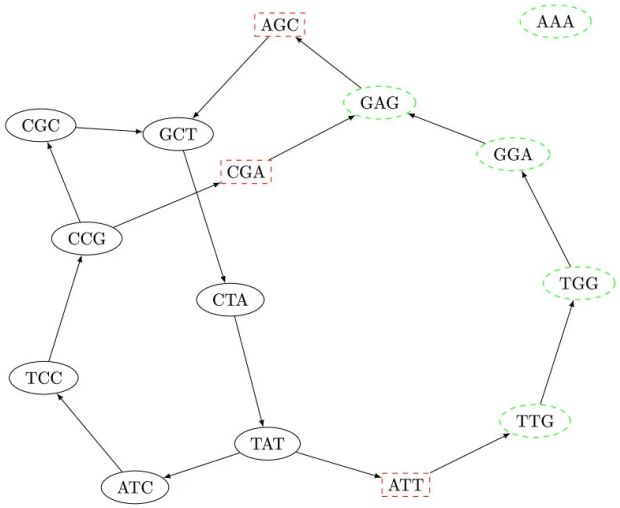


Figura 5.1: Esempio di De Bruijn Graph e della sua rappresentazione probabilistica tramite Bloom Filter. Adattato da [2].

Consideriamo l'esempio del grafo con $k = 3$ mostrato nella figura:

- **Nodi circolari pieni (neri)**: rappresentano S, l'insieme dei 7 k-mer reali del genoma. Ad esempio: `ACG`, `CGT`, `GTA`, ecc.
- **Nodi rettangolari tratteggiati (rossi)**: rappresentano **cFP**, i falsi positivi critici. Questi sono k-mer che:
  - Non esistono nel genoma reale

- Sono estensioni dirette di nodi in S
- Il Bloom Filter risponde `true` erroneamente

Ad esempio, se `CGT` $\in S$ e una sua estensione `CGTA` non è reale ma il Bloom Filter la riporta come presente, allora `CGTA` $\in$ cFP.

- **Nodi circolari tratteggiati (verdi)**: rappresentano altri falsi positivi del Bloom Filter che **non appartengono a cFP**. Questi k-mer:
  - Non sono estensioni dirette di nodi in S
  - Non verranno mai interrogati durante il traversal
  - Possono essere ignorati senza conseguenze

L'articolo riporta che, nel *Toy Example*[4] della Figura 5.2, su un Bloom Filter di 10 bit che memorizza 7 k-mer reali, esistono 3 falsi positivi critici che devono essere memorizzati esplicitamente [2].

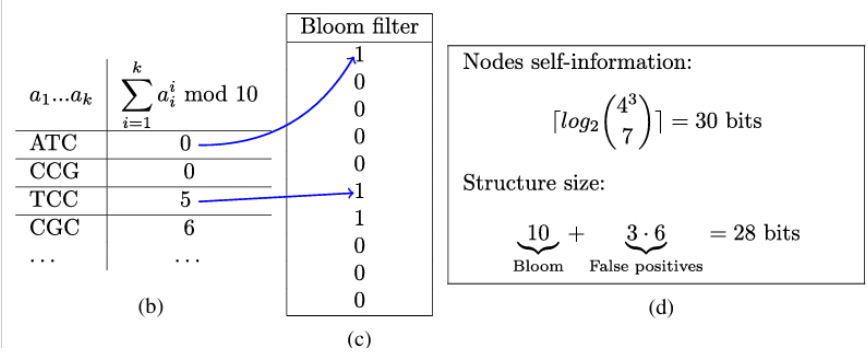


Figura 5.2: Rappresentazione schematica ("Toy Example") della struttura dati di Minia. Un insieme S composto da 7 k-mer reali viene mappato su un Bloom Filter di 10 bit. I nodi indicati come falsi positivi critici (nell'esempio: b, d, c) sono artefatti probabilistici che vengono identificati e memorizzati esplicitamente in una struttura separata per garantire l'esattezza del grafo [2].

[4] termine usato per indicare il grafo semplificato.

5.4.3 Cardinalità dell'insieme cFP

La dimensione dell'insieme cFP è un parametro cruciale per determinare il consumo di memoria complessivo della struttura.

Stima teorica. Dato un tasso di falsi positivi F del Bloom Filter e $n = |S|$ k-mer reali, ciascun k-mer ha esattamente 8 possibili estensioni (4 a destra, 4 a sinistra). Assumendo che il Bloom Filter produca falsi positivi in modo indipendente per ciascuna estensione, la dimensione attesa di cFP è:

$$\mathbb{E}[|\text{cFP}|] = 8 \cdot n \cdot F$$

Esempio: genoma umano. Per il genoma umano con parametri tipici:

- $n = 2.4 \times 10^9$ k-mer solidi
- $k = 27$
- $F = 0.00481$ (tasso ottimale, come vedremo nella Sezione 5.5)

La dimensione attesa di cFP è:

$$\mathbb{E}[|\text{cFP}|] = 8 \times 2.4 \times 10^9 \times 0.00481 \approx 92.4 \times 10^6 \text{ k-mer}$$

Valore misurato. Nell'esperimento reale riportato nell'articolo [2], il numero effettivo di falsi positivi critici per il genoma umano (dataset NA18507) è stato:

$$|\text{cFP}|_{\text{misurato}} = 78{,}762{,}871 \text{ k-mer} \approx 78.7 \text{ milioni}^5$$

Questo valore è leggermente inferiore alla stima teorica, probabilmente a causa del filtraggio per abbondanza ($d \geq 3$) che riduce il numero di estensioni spurie derivanti da errori di sequenziamento.

[5] Questo valore (78.7M) assume che tutte le 8 estensioni di ogni k-mer siano interrogate. Nella pratica, molti k-mer hanno < 8 estensioni reali, quindi |cFP| effettivo è spesso inferiore alla stima teorica.

Proporzione rispetto a n. Il rapporto tra cFP e il numero totale di k-mer è:

$$\frac{|\text{cFP}|}{n} = \frac{78.7 \times 10^6}{2.71 \times 10^9} \approx 2.9\%$$

Questo significa che solo il **2.9% dei k-mer totali** deve essere memorizzato esplicitamente nella struttura cFP per correggere completamente la topologia del grafo [2].

5.4.4 Algoritmo di costruzione a memoria costante

Un aspetto critico della struttura cFP è che deve essere costruita senza superare il budget di memoria disponibile. L'articolo presenta l'**Algorithm 1** [2], che enumera i falsi positivi critici utilizzando una quantità di memoria *costante* e configurabile.

Strategia generale. L'algoritmo si basa su un approccio iterativo di filtraggio:

1. Si generano tutte le estensioni positive P (quelle per cui il Bloom Filter risponde `true`) e si salvano su disco
2. Si libera il Bloom Filter dalla RAM, recuperando memoria
3. Si partiziona l'insieme S in blocchi che entrano in memoria
4. Per ogni blocco, si filtra P rimuovendo i k-mer che appartengono a S (true positives)
5. Alla fine, rimangono solo i k-mer in $P \setminus S$, ovvero esattamente cFP

Algorithm 1 Enumerazione dei falsi positivi critici a memoria costante

Require: Insieme S dei k-mer reali, Bloom Filter BF, parametro M (max elementi per partizione)

Ensure: Insieme cFP dei falsi positivi critici

1: Memorizzare su disco l'insieme P delle estensioni di S per cui BF risponde `true`
2: Liberare BF dalla memoria RAM
3: $D_0 \leftarrow P$
4: $i \leftarrow 0$
5: **while** non è stata raggiunta la fine di S **do**
6: $\quad P_i \leftarrow \emptyset$
7: $\quad$ **while** $|P_i| < M$ **do**
8: $\quad\quad P_i \leftarrow P_i \cup \{\text{prossimo k-mer in } S\}$
9: $\quad$ **end while**
10: $\quad$ **for** ogni k-mer $m \in D_i$ **do**
11: $\quad\quad$ **if** $m \notin P_i$ **then**
12: $\quad\quad\quad D_{i+1} \leftarrow D_{i+1} \cup \{m\}$
13: $\quad\quad$ **end if**
14: $\quad$ **end for**
15: $\quad$ Eliminare D_i e P_i dal disco
16: $\quad i \leftarrow i + 1$
17: **end while**
18: cFP $\leftarrow D_i$
19: **return** cFP

Spiegazione dettagliata:

- **Passo 1-2**: Si interroga il Bloom Filter per tutte le estensioni dei k-mer in S, generando l'insieme P che contiene sia i true positives (estensioni reali) sia i false positives (estensioni spurie). Questo insieme viene salvato su disco in modo sequenziale. Successivamente, il Bloom Filter viene liberato dalla RAM, recuperando circa 3-4 GB di memoria (nel caso del genoma umano).

- **Passo 3-5**: Si inizializza il processo iterativo. D_0 contiene inizialmente tutti gli elementi di P (potenziali cFP).
- **Passo 6-14**: Ciclo principale che processa S a blocchi:
  - Si carica in RAM una partizione P_i di S (al massimo M elementi)
  - Si scorre sequenzialmente D_i (i candidati cFP rimanenti)
  - Per ogni k-mer $m \in D_i$, si verifica se $m \in P_i$:
    - Se $m \in P_i$: m è un true positive, viene scartato
    - Se $m \notin P_i$: m è ancora un candidato cFP, viene mantenuto in D_{i+1}
  - Si eliminano dal disco D_i (ormai processato) e P_i (non più necessario)
- **Passo 15-16**: Al termine di tutte le iterazioni, D_i contiene solo i k-mer che non sono mai stati trovati in nessuna partizione di S, ovvero esattamente gli elementi di $P \setminus S = \text{cFP}$.

Complessità computazionale.

- **Tempo**: $O\left(\frac{|S|}{M} \cdot |P|\right)$, dove il numero di iterazioni è $\lceil |S|/M \rceil$ e ogni iterazione richiede uno scan completo di $D_i \approx |P|$
- **Memoria**: $O(M)$, determinata dalla dimensione della partizione P_i. Nell'implementazione di Minia, M è configurato per occupare la stessa memoria liberata dal Bloom Filter (circa 4 GB)
- **I/O disco**: Tutti gli accessi sono **sequenziali**, minimizzando il tempo di lettura/scrittura. Questo è cruciale per mantenere prestazioni accettabili anche su dischi meccanici

Esempio numerico. Per il genoma umano [2]:

- $|S| = 2.71 \times 10^9$ k-mer
- $|P| \approx 2.79 \times 10^9$ (include true e false positives)
- $M \approx 5 \times 10^8$ k-mer (memoria liberata dal Bloom Filter)

- Numero di iterazioni: $\lceil 2.71 \times 10^9 / 5 \times 10^8 \rceil \approx 6$
- Tempo misurato: **2.9 ore** (su singolo thread CPU)

Questo algoritmo è fondamentale perché permette di costruire cFP anche quando l'insieme S completo non entra in memoria, come avviene per genomi di grandi dimensioni o metagenomica.

5.4.5 Codifica e costo in memoria

Una volta costruito, l'insieme cFP deve essere memorizzato in una struttura dati che supporti query di membership efficienti.

Rappresentazione. Minia memorizza cFP in una **hash table** o in un **array ordinato** di k-mer. Ogni k-mer richiede esattamente $2k$ bit per rappresentare la sequenza nucleotidica, utilizzando la codifica a 2 bit per base:

$$\{A, C, G, T\} \to \{00, 01, 10, 11\}$$

Per $k = 27$:

$$\text{Dimensione per k-mer} = 2 \times 27 = 54 \text{ bit} = 6.75 \text{ bytes}$$

Memoria totale per cFP. Per il genoma umano con $|\text{cFP}| = 78.7 \times 10^6$:

$$M_{\text{cFP}} = 78.7 \times 10^6 \times 54 \text{ bit} \approx 4.25 \times 10^9 \text{ bit} \approx 531 \text{ MB}$$

Normalizzando per k-mer:

$$\frac{M_{\text{cFP}}}{n} = \frac{531 \text{ MB}}{2.71 \times 10^9 \text{ k-mer}} \approx \boxed{1.86 \text{ bit/k-mer}}$$

L'articolo riporta esattamente questo valore [2].

Confronto con il Bloom Filter. Come verrà dimostrato nella sezione 5.5 il Bloom Filter ottimale per il genoma umano occupa circa 11.1 bit/k-mer.

Quindi:

$$\frac{M_{\text{cFP}}}{M_{\text{Bloom}}} = \frac{1.86}{11.1} \approx 16.8\%$$

La struttura cFP occupa meno del **17% della memoria del Bloom Filter**, confermando che il costo aggiuntivo per correggere i falsi positivi è trascurabile [2].

Costo delle query. Le operazioni di query su cFP hanno complessità:

- **Hash table**: $O(1)$ atteso, $O(n)$ worst-case
- **Array ordinato + ricerca binaria**: $O(\log |\text{cFP}|)$ garantito

Minia utilizza una hash table per massimizzare le prestazioni durante l'attraversamento del grafo, dove le query di membership su cFP vengono effettuate milioni di volte al secondo.

Modifica delle query al Bloom Filter. Con la struttura cFP in memoria, ogni query al Bloom Filter viene modificata come segue:

Algorithm 2 Query corretta al grafo probabilistico

```
1: function CONTAINS(k-mer v)
2:     if BloomFilter.contains(v) = false then
3:         return false                          ▷ Definitivamente assente
4:     end if
5:     if v ∈ cFP then
6:         return false                          ▷ Falso positivo critico
7:     end if
8:     return true                               ▷ Realmente presente
9: end function
```

Questa semplice modifica garantisce che la topologia del grafo sia **identica** a quella del grafo esatto, eliminando completamente le ramificazioni spurie introdotte dai falsi positivi [2].

Osservazione finale. La struttura cFP rappresenta il contributo centrale dell'articolo di Chikhi e Rizk. Essa dimostra che è possibile ottenere una rappresentazione esatta del De Bruijn Graph utilizzando un Bloom Filter, a patto di memorizzare esplicitamente un piccolo sottoinsieme di falsi positivi (circa 3% del numero totale di

k-mer). Il costo addizionale in termini di memoria (1.86 bit/k-mer) è ampiamente compensato dal risparmio ottenuto rispetto a strutture deterministiche come le hash table, che richiederebbero almeno 32-64 bit/k-mer.

Nella sezione successiva analizzeremo come dimensionare ottimalmente il Bloom Filter per minimizzare la memoria totale della struttura ($M_{\text{Bloom}} + M_{\text{cFP}}$), derivando la formula per il tasso di falsi positivi ottimale in funzione di k.

5.5 Dimensionamento ottimale del Bloom Filter

La scelta della dimensione del Bloom Filter ha un impatto diretto sia sul consumo di memoria che sulla qualità dell'assemblaggio. Una dimensione troppo piccola genera molti falsi positivi, aumentando la dimensione di cFP; una dimensione troppo grande riduce i falsi positivi ma spreca memoria nel filtro stesso. Esiste quindi un **punto di ottimo** che minimizza la memoria totale della struttura [2].

5.5.1 Trade-off tra Bloom Filter e cFP

La memoria totale della struttura dati di Minia (escludendo per ora la struttura di marcatura) è data dalla somma:

$$M_{\text{totale}} = M_{\text{Bloom}} + M_{\text{cFP}}$$

dove:

- M_{Bloom}: memoria occupata dal Bloom Filter
- M_{cFP}: memoria per l'insieme dei falsi positivi critici

Entrambe le componenti dipendono dal **tasso di falsi positivi** F del Bloom Filter, ma in direzioni opposte:

1. **Bloom Filter più grande** $\Rightarrow$ F più piccolo $\Rightarrow$ meno falsi positivi critici

$$M_{\text{Bloom}} \uparrow \quad M_{\text{cFP}} \downarrow$$

2. **Bloom Filter più piccolo** $\Rightarrow$ F più grande $\Rightarrow$ più falsi positivi critici

$$M_{\text{Bloom}} \downarrow \quad M_{\text{cFP}} \uparrow$$

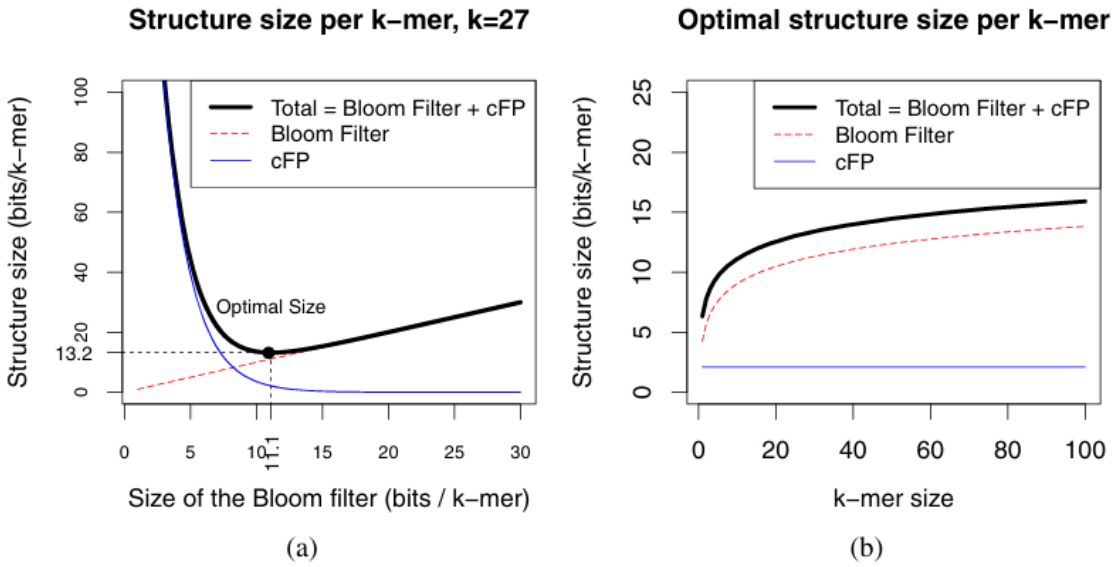


Figura 5.3: Analisi del consumo di memoria (tratta da [2]). Andamento della memoria ottimale al variare di k.

La Figura 5.4 dell'articolo [2] mostra graficamente questo trade-off per $k = 27$: esiste un punto di minimo intorno a 11.1 bit/k-mer per il Bloom Filter, che corrisponde a una memoria totale di 13.2 bit/k-mer.

5.5.2 Formula per la memoria totale

Per derivare la dimensione ottimale, dobbiamo esprimere M_{Bloom} e M_{cFP} in funzione del tasso di falsi positivi F.

Dimensione del Bloom Filter. Dal Capitolo 4, sappiamo che per un Bloom Filter ottimizzato con numero ottimale di funzioni hash $h \approx 0.7r$ (dove $r = m/n$ è il numero di bit per elemento), la dimensione del filtro in bit è [2]:

$$M_{\text{Bloom}} = 1.44 \cdot n \cdot \log_2 \left(\frac{1}{F} \right) \text{ bit}$$

dove $n = |S|$ è il numero di k-mer da inserire e F è il tasso di falsi positivi desiderato.

Dimensione di cFP. Come derivato nella Sezione 5.4.3, il numero atteso di falsi positivi critici è:

$$|\text{cFP}| \approx 8 \cdot n \cdot F$$

Ogni k-mer in cFP richiede $2k$ bit per la codifica della sequenza. Quindi:

$$M_{\text{cFP}} = |\text{cFP}| \cdot 2k = 8 \cdot n \cdot F \cdot 2k = 16 \cdot n \cdot k \cdot F \text{ bit}$$

Memoria totale. Combinando le due formule:

$$\boxed{M_{\text{totale}}(F) = n \cdot \left[1.44 \log_2 \left(\frac{1}{F}\right) + 16kF\right] \text{ bit}}$$

Questa è l'**Equazione (2)** dell'articolo [2].

Dividendo per n otteniamo il costo per k-mer:

$$\frac{M_{\text{totale}}}{n} = 1.44 \log_2 \left(\frac{1}{F}\right) + 16kF \quad \text{bit/k-mer}$$

5.5.3 Derivazione del tasso di falsi positivi ottimale

Per trovare il valore di F che minimizza M_{totale}, calcoliamo la derivata rispetto a F e la poniamo uguale a zero.

Calcolo della derivata.

$$\begin{aligned}
\frac{dM_{\text{totale}}}{dF} &= n \cdot \frac{d}{dF}\left[1.44 \log_2 \left(\frac{1}{F}\right) + 16kF\right] \\
&= n \cdot \left[1.44 \cdot \frac{d}{dF} \log_2 \left(F^{-1}\right) + 16k\right] \\
&= n \cdot \left[1.44 \cdot \frac{-1}{F \ln 2} + 16k\right]
\end{aligned}$$

dove abbiamo usato:

$$\frac{d}{dF} \log_2 \left(\frac{1}{F}\right) = \frac{1}{\ln 2} \cdot \frac{-1}{F}$$

Condizione di ottimalità. Ponendo la derivata uguale a zero:

$$n \cdot \left[-\frac{1.44}{F \ln 2} + 16k \right] = 0$$

Risolvendo per F:

$$\frac{1.44}{F \ln 2} = 16k$$

$$F = \frac{1.44}{16k \cdot \ln 2}$$

Calcolando numericamente $\ln 2 \approx 0.693$:

$$F = \frac{1.44}{16k \cdot 0.693} = \frac{1.44}{11.088k} \approx \frac{0.130}{k}$$

Semplificando ulteriormente (come nell'articolo [2]):

$$F_{\text{opt}} \approx \frac{2.08}{16k}$$

che può essere riscritta come:

$$\boxed{F_{\text{opt}} = \left(\frac{16k}{2.08} \right)^{-1}}$$

Questa è la formula riportata dopo l'**Equazione (2)** nell'articolo [2].

Verifica della derivata seconda. Per confermare che si tratta di un minimo e non di un massimo, verifichiamo la derivata seconda:

$$\frac{d^2 M_{\text{totale}}}{dF^2} = n \cdot \frac{1.44}{F^2 \ln 2} > 0 \quad \forall F > 0$$

Essendo positiva, confermiamo che F_{opt} è un punto di **minimo**.

Memoria minima teorica

Sostituendo F_{opt} nella formula della memoria totale, otteniamo l'espressione per la memoria minima.

Calcolo di $\log_2(1/F_{\text{opt}})$.

$$\begin{aligned} \log_2 \left(\frac{1}{F_{\text{opt}}} \right) &= \log_2 \left(\frac{16k}{2.08} \right) \\ &= \log_2(16k) - \log_2(2.08) \\ &= 4 + \log_2(k) - \log_2(2.08) \end{aligned}$$

Calcolo di $16kF_{\text{opt}}$.

$$16kF_{\text{opt}} = 16k \cdot \frac{2.08}{16k} = 2.08$$

Memoria minima. Sostituendo nella formula:

$$\begin{aligned} M_{\min} &= n \cdot \left[1.44 \log_2 \left(\frac{16k}{2.08}\right) + 2.08\right] \text{ bit} \\ &= n \cdot [1.44\,(4 + \log_2(k) - \log_2(2.08)) + 2.08] \end{aligned}$$

Semplificando (con $\log_2(2.08) \approx 1.06$):

$$\boxed{M_{\min} = n \cdot \left[1.44 \log_2 \left(\frac{16k}{2.08}\right) + 2.08\right] \text{ bit}}$$

Questa è l'**Equazione (3)** dell'articolo [2].

Interpretazione. La memoria minima si scompone in due contributi:

- **Bloom Filter**: $1.44 \log_2(16k/2.08)$ bit/k-mer
- **cFP**: 2.08 bit/k-mer (costante, indipendente da k!)

Questa decomposizione mostra un risultato sorprendente: il costo di cFP è **costante** quando si sceglie F_{opt}, mentre il Bloom Filter cresce solo logaritmicamente con k [2].

Applicazione al genoma umano

Applichiamo ora le formule derivate al caso del genoma umano con i parametri utilizzati nell'articolo.

Parametri.

- Numero di k-mer solidi: $n = 2{,}712{,}827{,}800 \approx 2.71 \times 10^9$
- Lunghezza k-mer: $k = 27$

Tasso di falsi positivi ottimale.

$$F_{\text{opt}} = \frac{2.08}{16 \times 27} = \frac{2.08}{432} \approx 0.00481 \approx \frac{1}{208} \approx 0.48\%$$

Dimensione ottimale del Bloom Filter.

$$\begin{aligned} M_{\text{Bloom}} &= 1.44 \times 2.71 \times 10^9 \times \log_2\left(\frac{1}{0.00481}\right) \\ &= 1.44 \times 2.71 \times 10^9 \times \log_2(208) \\ &= 1.44 \times 2.71 \times 10^9 \times 7.70 \\ &\approx 3.00 \times 10^{10} \text{ bit} \\ &\approx 3.75 \text{ GB} \end{aligned}$$

Normalizzando per k-mer:

$$\frac{M_{\text{Bloom}}}{n} = \frac{3.00 \times 10^{10}}{2.71 \times 10^9} \approx \boxed{11.1 \text{ bit/k-mer}}$$

Questo valore corrisponde esattamente a quanto riportato nell'articolo [2].

Dimensione attesa di cFP.

$$\begin{aligned} M_{\text{cFP}} &= 16 \times 2.71 \times 10^9 \times 27 \times 0.00481 \\ &\approx 5.63 \times 10^9 \text{ bit} \\ &\approx 704 \text{ MB} \end{aligned}$$

Normalizzando:

$$\frac{M_{\text{cFP}}}{n} \approx \boxed{2.08 \text{ bit/k-mer}}$$

Il valore misurato sperimentalmente è 1.86 bit/k-mer, leggermente inferiore alla stima teorica.

Memoria totale.

$$M_{\text{totale}} = 11.1 + 2.08 = \boxed{13.2 \text{ bit/k-mer}}$$

Per il genoma completo:

$$M_{\text{totale}} = 2.71 \times 10^9 \times 13.2/8 \approx 4.5 \text{ GB}$$

L'articolo riporta un consumo effettivo di **5.7 GB**, che include anche la struttura di marcatura (1.29 GB) analizzata nella Sezione 5.6.

5.5.4 Influenza del parametro k

Un aspetto interessante è che la memoria ottimale dipende solo **logaritmicamente** dalla lunghezza del k-mer.

Analisi asintotica. Dalla formula della memoria minima:

$$\frac{M_{\min}}{n} = 1.44 \log_2 \left(\frac{16k}{2.08} \right) + 2.08$$

Possiamo riscrivere:

$$\frac{M_{\min}}{n} = 1.44 \left[\log_2(16) + \log_2(k) - \log_2(2.08) \right] + 2.08$$

Semplificando ($\log_2(16) = 4, \log_2(2.08) \approx 1.06$):

$$\frac{M_{\min}}{n} \approx 1.44 \times 2.94 + 1.44 \log_2(k) + 2.08 \approx 6.31 + 1.44 \log_2(k)$$

Quindi la memoria cresce come $O(\log k)$.

Esempi numerici. La Figura 5.4(b) dell'articolo [2] mostra la memoria ottimale per diversi valori di k. Riproduciamo alcuni valori:

Tabella 5.1: Memoria ottimale in funzione di k

k	F_{opt}	Bloom (bit/k-mer)	Totale (bit/k-mer)
15	1/115	9.8	11.9
21	1/161	10.5	12.6
27	1/208	11.1	13.2
31	1/238	11.4	13.5
47	1/361	12.2	14.3
63	1/484	12.8	14.9

Osservazioni.

- Raddoppiando k da 21 a 47, la memoria totale aumenta solo di 1.7 bit/k-mer (13.5%)

- Triplicando k da 21 a 63, l'aumento è di soli 2.3 bit/k-mer (18.3%)
- Questo comportamento logaritmico rende Minia **poco sensibile** alla scelta di k, permettendo di utilizzare valori grandi di k (che migliorano la qualità dell'assemblaggio) senza penalità significative in memoria

Questo è in netto contrasto con le hash table classiche, dove la memoria cresce linearmente con la lunghezza dei k-mer memorizzati [2].

5.5.5 Confronto con un Bloom Filter "puro"

Per apprezzare appieno il valore della struttura cFP, confrontiamo la soluzione di Minia con un ipotetico Bloom Filter dimensionato per avere *meno di un falso positivo critico atteso*.

Condizione per FP trascurabili. Per avere meno di 1 falso positivo critico atteso, serve:

$$8nF < 1 \quad \Rightarrow \quad F < \frac{1}{8n}$$

Applicazione al genoma umano. Con $n = 2.71 \times 10^9$:

$$F < \frac{1}{8 \times 2.71 \times 10^9} \approx 4.6 \times 10^{-11}$$

Questo tasso di falsi positivi estremamente basso richiede:

$$M_{\text{Bloom}} = 1.44 \times n \times \log_2\left(\frac{1}{F}\right)$$

Calcolando $\log_2(1/F)$:

$$\log_2\left(\frac{1}{4.6 \times 10^{-11}}\right) = \log_2(2.17 \times 10^{10}) \approx 34.3$$

Quindi:

$$M_{\text{Bloom}} \approx 1.44 \times 2.71 \times 10^9 \times 34.3 \approx 1.34 \times 10^{11} \text{ bit} \approx \boxed{16.8 \text{ GB}}$$

L'articolo riporta un valore leggermente inferiore di **13.7 GB** [2], probabilmente usando una stima più conservativa.

Tabella 5.2: Confronto Minia vs Bloom Filter puro (genoma umano, $k = 27$)

Metrica	**Minia (Bloom + cFP)**	**Bloom puro**
Tasso FP	0.48%	$< 4.6 \times 10^{-9}\%$
Bloom Filter	3.75 GB	13.7 GB
cFP	0.53 GB	0 GB
Totale	**4.3 GB**	**13.7 GB**
Riduzione	**68.6%**	

Conclusione. La struttura cFP permette di utilizzare un Bloom Filter con tasso di falsi positivi relativamente alto (0.5%), correggendo esplicitamente solo i falsi positivi problematici. Questo porta a una **riduzione del 69% della memoria** rispetto a un Bloom Filter dimensionato per eliminare virtualmente tutti i falsi positivi [2].

Questo risultato dimostra l'efficacia dell'approccio di Minia: anziché tentare di eliminare completamente i falsi positivi (soluzione costosa), si accettano falsi positivi controllati e si correggono solo quelli che effettivamente alterano la topologia del grafo.

Nella sezione successiva analizzeremo la terza componente della struttura di Minia: la struttura di marcatura, necessaria per tenere traccia dei nodi già visitati durante l'attraversamento del grafo.

5.6 Struttura di marcatura per l'attraversamento del grafo

Le sezioni precedenti hanno descritto come Minia rappresenti efficacemente l'insieme dei nodi del De Bruijn Graph tramite Bloom Filter e cFP, garantendo che la topologia del grafo sia identica a quella del grafo esatto. Tuttavia, per generare i contigs è necessario **attraversare** il grafo, visitando ciascun nodo esattamente una volta. Questo richiede un meccanismo per tenere traccia di quali nodi sono già stati visitati [2].

5.6.1 Necessità della marcatura

Durante l'attraversamento del grafo, gli algoritmi di assemblaggio tipicamente basati su DFS e BFS [6] devono evitare di visitare ripetutamente gli stessi nodi, per prevenire:

- **Cicli infiniti**: in presenza di loop nel grafo (dovuti a regioni ripetute), senza marcatura l'algoritmo potrebbe non terminare
- **Contigs duplicati**: lo stesso cammino verrebbe generato multiple volte
- **Inefficienza computazionale**: revisitare nodi già processati spreca tempo di calcolo

La soluzione standard consiste nel mantenere un **bit di visita** per ciascun nodo del grafo, impostato a 1 quando il nodo viene attraversato per la prima volta.

5.6.2 Il problema della mutabilità

Il Bloom Filter, per sua natura, è una struttura dati **append-only**: una volta inserito un elemento, non è possibile modificarlo o rimuoverlo (come discusso nella Sezione 4.5.1). In particolare, il Bloom Filter non può memorizzare informazioni addizionali come bit di stato per ciascun k-mer [2].

Una soluzione ingenua sarebbe mantenere un array di bit separato, con un bit per ciascuno degli n k-mer:

$$M_{\text{marking}}^{\text{naive}} = n \text{ bit}$$

Per il genoma umano con $n = 2.71 \times 10^9$:

$$M_{\text{marking}}^{\text{naive}} = 2.71 \times 10^9 \text{ bit} \approx 339 \text{ MB} \approx 1 \text{ bit/k-mer}$$

Sebbene questo costo sia relativamente contenuto (circa il 9% del Bloom Filter), Minia propone una soluzione ancora più efficiente sfruttando la **topologia del grafo** [2].

[6] DFS (Depth-First Search) e BFS (Breadth-First Search) sono le due strategie fondamentali per "camminare" all'interno di un grafo (o di un albero) e visitare i suoi nodi.

5.6.3 Marcatura selettiva dei nodi complessi

L'intuizione chiave di Minia è che **non tutti i nodi devono essere marcati esplicitamente**.

Definizione di nodo complesso. Un nodo v è definito **complesso** se presenta ramificazioni, ovvero se:

$$\text{in-degree}(v) \neq 1 \quad \text{oppure} \quad \text{out-degree}(v) \neq 1$$

Al contrario, un nodo **semplice** soddisfa:

$$\text{in-degree}(v) = 1 \quad \text{e} \quad \text{out-degree}(v) = 1$$

I nodi semplici fanno parte di **cammini lineari** (unitigs) nel grafo, mentre i nodi complessi rappresentano punti decisionali dove il cammino può diramarsi.

Proprietà fondamentale. Consideriamo un cammino lineare (sequenza di nodi semplici):

$$v_1 \to v_2 \to v_3 \to \cdots \to v_\ell$$

dove tutti i nodi intermedi sono semplici. Durante l'attraversamento del grafo:

- Se il cammino viene attraversato, **tutti** i nodi $v_1, \ldots, v_\ell$ verranno visitati in sequenza
- Se il cammino non viene attraversato, **nessuno** dei nodi verrà visitato

In altre parole, i nodi semplici in un cammino lineare **condividono lo stesso stato di visita**. È quindi sufficiente marcare solo le estremità del cammino (i nodi complessi) per determinare se l'intero cammino è stato visitato [2].

Strategia di Minia. Minia memorizza esplicitamente solo i **nodi complessi** in una hash table separata (la *marking structure*). Durante l'attraversamento:

1. Quando si incontra un nodo complesso, si verifica se è presente nella marking structure

- Se presente: nodo già visitato, cambia direzione
- Se assente: nodo non visitato, lo si aggiunge alla struttura e si procede

2. I nodi semplici vengono attraversati senza controlli, fino a raggiungere il prossimo nodo complesso

Questa strategia riduce drasticamente il numero di k-mer da memorizzare nella marking structure, proporzionalmente alla **complessità topologica** del grafo anziché alla sua dimensione totale.

Esempio. Consideriamo un frammento di grafo:

$$\underbrace{v_1}_{\text{complesso}} \rightarrow \underbrace{v_2 \rightarrow v_3 \rightarrow \cdots \rightarrow v_{100}}_{\text{100 nodi semplici}} \rightarrow \underbrace{v_{101}}_{\text{complesso}}$$

Approccio naive: marca tutti i 102 nodi (102 bit) Approccio Minia: marca solo v_1 e v_{101} (2 entry nella hash table)

5.6.4 Implementazione e costo in memoria

Struttura dati. La marking structure è implementata come una **hash table** compatta che memorizza:

- La sequenza del k-mer (per identificarlo univocamente)
- Metadati minimali (bit di visita, eventuali flag)

Ogni entry nella hash table richiede [2]:

$$C \approx 2k + 8 \text{ bit}$$

dove:

- $2k$ bit: sequenza del k-mer (2 bit/nucleotide)
- 8 bit: overhead della hash table (puntatore, flag, padding)

Per $k = 27$:

$$C = 2 \times 27 + 8 = 54 + 8 = 62 \text{ bit} \approx 7.75 \text{ bytes}$$

Costo totale. Sia n_c il numero di nodi complessi nel grafo. La memoria richiesta è:

$$M_{\text{marking}} = n_c \cdot C \text{ bit}$$

Normalizzando per il numero totale di k-mer n:

$$\frac{M_{\text{marking}}}{n} = \frac{n_c}{n} \cdot C \text{ bit/k-mer}$$

Il rapporto n_c/n dipende dalla **struttura del genoma**:

- Genomi semplici (pochi branching): n_c/n piccolo
- Genomi complessi (molte ripetizioni, errori): n_c/n grande

L'articolo nota che nei De Bruijn Graph di genomi reali, l'insieme totale dei nodi è significativamente più grande dell'insieme dei nodi complessi [2].

5.6.5 Risultati per il genoma umano

Dataset: NA18507. Per il genoma umano assemblato da Minia [2]:

- K-mer totali: $n = 2{,}712{,}827{,}800$
- K-mer complessi: $n_c = 166{,}649{,}498$
- Percentuale: $n_c/n \approx 6.1\%$

Solo il **6.1% dei k-mer** presenta ramificazioni e deve essere memorizzato nella marking structure.

Memoria utilizzata. Con $C = 62$ bit per entry:

$$M_{\text{marking}} = 166{,}649{,}498 \times 62 \text{ bit} \approx 1.03 \times 10^{10} \text{ bit} \approx 1.29 \text{ GB}$$

Normalizzando:

$$\frac{M_{\text{marking}}}{n} = \frac{1.29 \text{ GB}}{2.71 \times 10^9 \text{ k-mer}} \approx \boxed{4.42 \text{ bit/k-mer}}$$

Questo valore è riportato esattamente nella Tabella 1 dell'articolo [2].

Confronto con approccio naive. L'approccio naive consisterebbe nel memorizzare un singolo bit di visita per ogni k-mer del grafo, richiedendo 1 bit/k-mer. Minia invece richiede 4.42 bit/k-mer per la marking structure. Questo apparente aumento è dovuto al fatto che:

1. Ogni entry nella hash table deve memorizzare la **sequenza completa** del k-mer (54 bit per $k = 27$) per identificarlo univocamente
2. La hash table ha overhead strutturale (circa 8 bit per entry: puntatori, padding)
3. Solo il 6.1% dei k-mer viene effettivamente memorizzato ($n_c = 0.061 \times n$)

Tuttavia, questo confronto con l'approccio naive è fuorviante nel contesto di Minia. Il Bloom Filter non memorizza esplicitamente i k-mer, quindi non è possibile associare direttamente un bit di stato a ciascun k-mer come si farebbe con una hash table deterministica.

Il **confronto corretto** è con un grafo probabilistico senza la struttura cFP. Come dimostrato nell'esperimento su *E. coli* [2], un grafo probabilistico puro (Bloom Filter senza correzione dei falsi positivi) richiede una marking structure di circa 4 bit/k-mer, rispetto agli 0.49 bit/k-mer di Minia con cFP. Questo perché i falsi positivi introducono numerosi nodi complessi spuri che devono essere tracciati durante l'attraversamento.

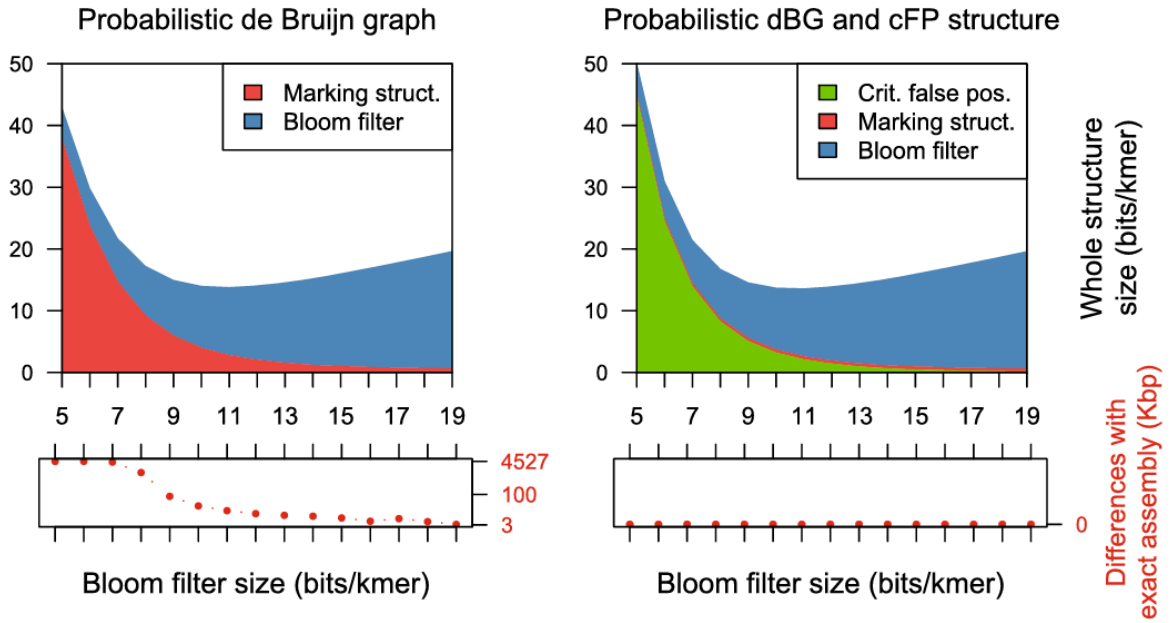


Figura 5.4: Dimensioni del Filtro di Bloom, della struttura di marcatura e del cFP per un dataset di E. coli (k=23). I grafici in alto confrontano la configurazione con cFP (destra) e senza cFP (sinistra) in base ai bit allocati per k-mero. I grafici in basso mostrano lo scostamento tra questi assemblaggi e un assemblaggio di riferimento eseguito con un grafo esatto.[2].

Nel caso del genoma umano, senza la correzione dei falsi positivi critici, la marking structure sarebbe significativamente più grande di 4.42 bit/k-mer. La strategia di Minia — correggere i falsi positivi con cFP per ridurre i nodi complessi e poi marcare solo questi ultimi — è quindi l'approccio più efficiente per minimizzare la memoria totale della struttura [2].

Breakdown memoria totale. Combinando tutte le componenti per il genoma umano:

Tabella 5.3: Composizione della memoria di Minia (genoma umano, $k = 27$)

Componente	Dimensione	bit/k-mer	% totale
Bloom Filter	3.75 GB	11.1	65.8%
cFP	0.53 GB	1.86	9.3%
Marking structure	1.29 GB	4.42	22.6%
Overhead sistema	0.13 GB	0.48	2.3%
Totale	**5.7 GB**	**17.4**	**100%**

La struttura di marcatura rappresenta circa il **23% della memoria totale**, un costo significativo ma giustificato dalla necessità di attraversare correttamente il grafo [2].

Impatto della complessità del genoma. Per un dataset di *E. coli*, l'articolo riporta una marking structure di solo 0.49 bit/k-mer [2], indicando una percentuale di nodi complessi molto più bassa (circa 1%). Questo mostra come il costo della marking structure sia proporzionale alla complessità biologica e agli errori di sequenziamento, non semplicemente alla dimensione del genoma.

5.6.6 Algoritmo di attraversamento semplificato

Per completezza, riportiamo uno pseudocodice semplificato dell'algoritmo di attraversamento con marcatura selettiva.

Algorithm 3 Attraversamento del grafo con marcatura selettiva

```
1:  procedure TRAVERSEGRAPH
2:      for ogni k-mer v ∈ S non ancora processato do
3:          if v è complesso and v ∈ markingSet then
4:              continue                                   ▷ Già visitato
5:          end if
6:          contig ← []
7:          EXTENDPATH(v, contig)
8:          Output contig
9:      end for
10: end procedure
11: procedure EXTENDPATH(v, contig)
12:     if v è complesso then
13:         markingSet ← markingSet ∪ {v}
14:     end if
15:     Aggiungi v a contig
16:     neighbors ← GETNEIGHBORS(v)
17:     if |neighbors| = 1 then
18:         w ← neighbors[0]
19:         EXTENDPATH(w, contig)                          ▷ Estendi linearmente
20:     else
21:         return                          ▷ Branching o dead-end, termina contig
22:     end if
23: end procedure
24: procedure GETNEIGHBORS(v)
25:     neighbors ← []
26:     for x ∈ {A, C, G, T} do
27:         w ← rightExtension(v, x)
28:         if CONTAINS(w) then                  ▷ Query corretta (Bloom + cFP)
29:             neighbors ← neighbors ∪ {w}
30:         end if
31:     end for
32:     return neighbors
33: end procedure
```

Spiegazione dettagliata Algorithm 3

L'algoritmo di attraversamento del grafo con marcatura selettiva è composto da tre procedure principali che lavorano in sinergia per generare i contigs minimizzando l'uso di memoria.

Procedura TRAVERSEGRAPH (righe 1-10) Questa è la procedura principale che orchestra l'intero processo di assemblaggio:

Riga 2: Itera su tutti i k-mer solidi dell'insieme S (quelli realmente presenti nel genoma). L'iterazione avviene in modo sequenziale, caricando i k-mer da disco senza materializzare l'intero insieme in memoria.

Righe 3-5: Prima di processare un k-mer, verifica se è complesso (in-degree o out-degree $\neq 1$) e se è già presente nel `markingSet`. Un nodo complesso già marcato indica che è stato visitato in un traversal precedente, quindi viene saltato per evitare duplicazioni. Questa è l'ottimizzazione chiave: **solo i nodi complessi vengono marcati**, non tutti i k-mer del cammino.

Riga 6: Inizializza un contig vuoto, che conterrà la sequenza assemblata per il cammino corrente.

Riga 7: Invoca `EXTENDPATH` per estendere il contig a partire dal k-mer corrente, seguendo il cammino nel grafo.

Riga 8: Una volta completato il cammino (quando `EXTENDPATH` termina), il contig viene salvato in output.

Procedura EXTENDPATH (righe 11-23) Questa procedura ricorsiva estende un contig seguendo cammini lineari nel grafo:

Righe 12-14: Se il k-mer corrente v è complesso (biforcazione o punto di convergenza), viene aggiunto al `markingSet`. Questa marcatura garantisce che non verrà processato di nuovo come punto di partenza in iterazioni future. I nodi semplici (nei cammini lineari) **non** vengono marcati, risparmiando memoria.

Riga 15: Il k-mer v viene aggiunto al contig in costruzione. Concatenando k-mer consecutivi si ricostruisce la sequenza genomica.

Riga 16: Interroga il grafo per ottenere i vicini di v tramite `GETNEIGHBORS`.

Righe 17-19: Se esiste esattamente un vicino, siamo in un cammino lineare univoco. L'algoritmo estende ricorsivamente il contig chiamando `EXTENDPATH` sul nodo successivo w.

Righe 20-21: Se ci sono 0 vicini (dead-end) o più di 1 vicino (biforcazione), il cammino lineare termina. La procedura ritorna, completando il contig corrente. Le biforcazioni verranno risolte in chiamate successive di `TRAVERSEGRAPH` partendo dai nodi complessi non ancora visitati.

Procedura GETNEIGHBORS (righe 24-33) Questa procedura determina i vicini validi di un k-mer nel grafo:

Riga 26: Itera sui 4 possibili nucleotidi $\{A, C, G, T\}$ per generare estensioni.

Riga 27: Calcola `rightExtension`(v, x).

Riga 28: Verifica se l'estensione w esiste realmente nel grafo tramite `CONTAINS`, che implementa la query corretta Bloom Filter + cFP (Algorithm 2). Solo le estensioni che corrispondono a k-mer reali vengono considerate.

Righe 29-32: Le estensioni valide vengono aggiunte all'insieme `neighbors` e ritornate.

Esempio pratico Consideriamo un semplice grafo:

```
ACGT → CGTG → GTGA (cammino lineare)
↓
GTGC (biforcazione)
```

1. `TRAVERSEGRAPH` inizia con $v =$ `ACGT` (semplice, non marcato)
2. `EXTENDPATH(ACGT)`: non complesso → non marca. `GETNEIGHBORS` trova solo `CGTG`
3. `EXTENDPATH(CGTG)`: complesso (out-degree=2) → **marca CGTG**. `GETNEIGHBORS` trova $\{$`GTGA`, `GTGC`$\}$
4. Poiché $|$`neighbors`$| > 1$, termina il contig: [`ACGT`, `CGTG`]
5. `TRAVERSEGRAPH` riprende, incontra `CGTG` → già marcato → salta

6. Processa separatamente i cammini da `GTGA` e `GTGC`

Risultato: Solo `CGTG` (nodo complesso) viene marcato. I nodi semplici `ACGT`, `GTGA`, `GTGC` non occupano spazio nel `markingSet`, riducendo l'uso di memoria.

Complessità computazionale.

- **Tempo:** $O(N)$ rispetto al numero di k-mer del grafo.
- **Memoria:** $O(n_c)$, con $n_c \ll n$ in grafi reali (tipicamente pochi % dei nodi), cioè molto più piccola di una marcatura "naive" con 1 bit per nodo.

Osservazione del codice: *File:* `dbgtopology.cpp`, righe 45-55
Versione: Minia v2.0.7
Download: codice sorgente [12]

5.6.7 Conclusione

La struttura di marcatura completa l'architettura di Minia, permettendo di attraversare il De Bruijn Graph in modo efficiente senza memorizzare esplicitamente lo stato di visita per tutti i k-mer. Sfruttando la proprietà che i nodi semplici formano cammini lineari con stato condiviso, Minia riduce il costo della marcatura da $O(n)$ a $O(n_c)$, dove tipicamente $n_c \ll n$ [2].

Combinando le tre componenti — Bloom Filter, cFP e marking structure — Minia raggiunge una rappresentazione del De Bruijn Graph che:

- È **esatta**: produce lo stesso risultato di una hash table deterministica
- È **compatta**: richiede 13-17 bit/k-mer vs 64-128 bit/k-mer delle strutture classiche
- È **scalabile**: assembla genomi umani completi con 5.7 GB di RAM

Nella sezione successiva presenteremo i risultati sperimentali riportati nell'articolo originale, confrontando Minia con gli assemblatori state-of-the-art dell'epoca.

5.7 Validazione sperimentale nell'articolo originale

Dopo aver descritto in dettaglio l'architettura di Minia, è importante presentare i risultati sperimentali riportati da Chikhi e Rizk (2013) [2], che dimostrano l'efficacia pratica dell'approccio proposto su dataset reali.

5.7.1 Assemblaggio del genoma umano completo

L'esperimento principale è stato condotto sul genoma umano dell'individuo NA18507, utilizzando dati pubblici dal Sequence Read Archive (accession SRA SRX016231) [2].

Configurazione dell'esperimento.

- Dataset: 142.3 Gbp di reads Illumina (100 bp), coverage 47×
- Parametri: $k = 27$, soglia abbondanza $d = 3$
- Solid k-mers: 2,712,827,800 (circa 2.7 miliardi)
- Piattaforma: singolo core CPU, nessuna parallelizzazione

Risultati: memoria e tempo. La Tabella 5.4 riassume le prestazioni di Minia per le diverse fasi dell'assemblaggio.

Tabella 5.4: Prestazioni di Minia sul genoma umano (NA18507)

Fase	Tempo (ore)	Memoria (GB)
k-mer counting	11.1	4.0
Enumerazione estensioni positive	2.8	3.6
Costruzione cFP	2.9	4.0
Assemblaggio (graph traversal)	6.4	5.7
Totale	**23.2**	**5.7 (picco)**

Il consumo di memoria massimo è stato di **5.7 GB**, distribuiti come segue:

- Bloom Filter: 3.75 GB (11.1 bit/k-mer)
- cFP: 0.53 GB (1.86 bit/k-mer, 78.7 milioni di k-mer)
- Marking structure: 1.29 GB (4.42 bit/k-mer, 166.6 milioni di nodi complessi)
- Overhead sistema: 0.13 GB

Risultati: qualità dell'assemblaggio. L'assemblaggio prodotto ha le seguenti caratteristiche [2]:

- Contigs totali: 3,490,000 (lunghezza $\geq$ 100 bp)
- Basi assemblate: 2.09 Gbp (circa 69% del genoma)
- N50: 1,156 bp
- Contig più lungo: 18.6 kbp

Per validare l'accuratezza, i contigs sono stati allineati al genoma di riferimento GRCh37. Il **94.6%** delle basi assemblate si allinea con identità di sequenza $\geq$ 98%, indicando un'elevata correttezza dell'assemblaggio [2].

5.7.2 Confronto con altri assemblatori

L'articolo confronta Minia con gli assemblatori state-of-the-art dell'epoca: ABySS, SOAPdenovo e l'implementazione succinct di Conway & Bromage [2]. La Tabella 5.5 riassume i risultati.

Analisi del confronto. Minia si distingue per:

1. **Memoria**: consumo drasticamente inferiore rispetto agli altri assemblatori
    - vs ABySS: riduzione di $336/5.7 \approx$ **59×**
    - vs SOAPdenovo: riduzione di $140/5.7 \approx$ **24×**
    - vs Conway & Bromage: riduzione di $32/5.7 \approx$ **5.6×**

Tabella 5.5: Confronto tra assemblatori sul genoma umano NA18507

Metrica	Minia	C.&B.	ABySS	SOAPdenovo
Parametro *k*	27	27	27	25
N50 (bp)	**1,156**	250	870	886
Basi assemblate (Gbp)	2.09	1.72	2.10	2.08
Cores utilizzati	1	8	21,168	116
Tempo (ore)	23	50	15	33
Memoria (GB)	**5.7**	**32**	**336**	**140**

2. **Contiguità**: Minia ottiene il miglior N50 (1,156 bp) tra tutti gli assemblatori, nonostante non utilizzi informazioni di paired-end[7]

3. **Scalabilità**: unico assemblatore single-threaded, eseguibile su un desktop consumer con 8 GB di RAM

L'articolo conclude che [2]:

> "To the best of our knowledge, Minia is the first method that can create contigs for a complete human genome on a desktop computer."

Questa capacità di assemblare genomi di grandi dimensioni con risorse hardware limitate rappresenta un avanzamento significativo rispetto alle tecniche precedenti, che richiedevano cluster di calcolo o server con centinaia di GB di RAM.

5.7.3 Validazione della struttura cFP

Per dimostrare l'importanza della struttura dei falsi positivi critici, gli autori hanno condotto un esperimento comparativo su *Escherichia coli* [2]. Sono stati confrontati due approcci:

- **Grafo probabilistico puro**: Bloom Filter senza correzione dei falsi positivi

[7] Paired-end (sequenziamento paired-end) è una tecnica di sequenziamento NGS in cui lo stesso frammento di DNA viene sequenziato da entrambe le estremità, generando due reads separate ma collegate.

- **Minia completo**: Bloom Filter + cFP + marking structure

Risultati chiave. L'esperimento ha mostrato che [2]:

1. Il consumo di memoria ottimale è identico per entrambi gli approcci (circa 13.6 bit/k-mer con Bloom Filter dimensionato a 11 bit/k-mer)
2. Tuttavia, la **qualità dell'assemblaggio** differisce drasticamente:
   - **Minia (con cFP)**: assemblaggio identico al riferimento (0 bp di differenza)
   - **Grafo probabilistico puro**: oltre 3 kbp di differenze rispetto al riferimento, anche con Bloom Filter ottimale
3. La marking structure nel grafo probabilistico puro è molto più grande (circa 4 bit/k-mer) rispetto a Minia (0.49 bit/k-mer), a causa dei numerosi nodi complessi falsi introdotti dai falsi positivi

L'articolo conclude che [2]:

> "Assemblies produced by the probabilistic de Bruijn graph are prone to randomness, while those produced by our structure are exact."

Questo esperimento conferma che la struttura cFP non è una semplice ottimizzazione, ma è **essenziale** per garantire la correttezza dell'assemblaggio quando si utilizza un Bloom Filter.

Collegamento con il Capitolo 6. Nel Capitolo 6 presenteremo una sperimentazione pratica su *E. coli* utilizzando Minia, che permetterà di validare quanto descritto in questo capitolo e di osservare direttamente il comportamento del tool su un dataset reale. In particolare, potremo verificare il consumo di memoria, il tempo di esecuzione e la qualità dell'assemblaggio prodotto, confrontando i nostri risultati con quelli riportati nell'articolo originale.

5.7.4 Limitazioni e direzioni future

L'articolo riconosce alcune limitazioni di Minia [2]:

- **Assenza di paired-end information**: Minia produce solo contigs, non scaffolds[8]. Altri assemblatori (es. SOAPdenovo, Allpaths) utilizzano le informazioni di pairing per produrre assemblaggi molto più contigui (N50 nell'ordine di decine o centinaia di kbp)
- **Traversal semplificato**: l'algoritmo ignora tips (cammini morti $< 2k + 1$) e regioni complesse con breadth > 20, perdendo potenzialmente informazioni su varianti o regioni ripetute
- **Single-threaded**: l'implementazione non sfrutta il parallelismo, che potrebbe ridurre significativamente i tempi di esecuzione

Gli autori suggeriscono che la struttura Bloom Filter + cFP può essere applicata ad altre applicazioni di sequenziamento [2]:

- Detection di varianti (SNP) reference-free
- Alternative splicing detection da dati RNA-seq
- Assemblaggio metagenomico

Queste applicazioni beneficerebbero della rappresentazione compatta del De Bruijn Graph, permettendo l'analisi di dataset complessi su hardware consumer.

5.7.5 Conclusioni della validazione

I risultati sperimentali riportati da Chikhi e Rizk dimostrano che Minia raggiunge gli obiettivi prefissati [2]:

1. **Riduzione drastica della memoria**: 5.7 GB per il genoma umano, 59× meno di ABySS

[8] Lo scaffolding è il processo che ordina e orienta i contigs sfruttando le informazioni delle reads paired-end per colmare i gap (buchi) nella sequenza.

2. **Accuratezza preservata**: 94.6% dei contigs corretti, comparabile agli assemblatori tradizionali

3. **Correttezza garantita**: la struttura cFP elimina gli errori introdotti dai falsi positivi del Bloom Filter

4. **Scalabilità**: primo assemblatore in grado di processare un genoma umano completo su un desktop consumer

Questi risultati confermano la validità teorica dell'approccio descritto nelle Sezioni 5.2-5.6 e dimostrano che è possibile ottenere una rappresentazione esatta del De Bruijn Graph con consumo di memoria sub-lineare rispetto alle dimensioni del genoma. Nel capitolo successivo, approfondiremo la comprensione di Minia attraverso una sperimentazione diretta su *E. coli*, che ci permetterà di osservare il funzionamento pratico del tool e di validare i concetti teorici studiati.

5.8 Organizzazione codice sorgente

La versione analizzata (v2.0.7) è strutturata come segue[12]:

- **src/**: directory principale dei sorgenti C++
  - `main.cpp`: entry point del tool, gestisce parsing parametri e invocazione pipeline
  - Tool di analisi grafo (`dbgtopology.cpp`, `dbginfo.cpp`, `dbgcheck.cpp`): utility per ispezione topologia De Bruijn Graph
  - Tool ausiliari (`BankDownload.cpp`, `bankgen.cpp`): gestione dataset e generazione reads sintetiche
- **thirdparty/gatb-core/**: libreria GATB (submodule Git)
  - `gatb/kmer/`: moduli per gestione k-mer (iteratori, modelli, hashing)
  - `gatb/bank/`: parsing formati FASTA/FASTQ
  - `gatb/debruijn/`: implementazione De Bruijn Graph (nodi, archi, traversal)

  - `gatb/tools/collections/`: strutture dati (hash, Bloom Filter storici in v1)

- **doc/**: manuale utente e documentazione tecnica (`manual.pdf`)
- **CMakeLists.txt**: configurazione build system (CMake)
- **scripts/**: script Python/Shell per test e benchmarking

Questa organizzazione modulare separa chiaramente:

1. Il *core algoritmico* (GATB: k-mer, grafo, strutture dati)
2. La *logica applicativa* (Minia: traversal, output contigs)
3. Le *utility* (analisi, testing, generazione dati)

La classe `GraphMarker` analizzata precedentemente si trova in `src/dbgtopology.cpp` (righe 30-60), mentre le primitive di base (iteratori k-mer, membership query) sono fornite dai moduli GATB riutilizzabili.

CAPITOLO 6

Sperimentazione pratica con Minia

Nel Capitolo 5 abbiamo analizzato in dettaglio l'architettura teorica di Minia, studiando i tre componenti fondamentali della struttura dati:

1. Il **Bloom Filter** che memorizza implicitamente l'insieme dei k-mer solidi con un consumo di memoria ottimale di circa 11 bit per k-mer, come dimostrato dalla formula $M_{\text{Bloom}} = 1.44 \cdot n \cdot \log_2(1/F)$

2. La struttura **cFP**, che corregge i falsi positivi critici adiacenti ai nodi reali, aggiungendo circa 2 bit per k-mer. L'Algorithm 1 descritto nella Sezione 5.4.4 mostra come questa struttura venga costruita con memoria costante

3. La **marking structure**, che traccia selettivamente i nodi complessi visitati durante l'attraversamento del grafo, con un costo proporzionale alla complessità topologica del genoma anziché alla sua dimensione totale

La Sezione 5.7 ha presentato i risultati sperimentali riportati da Chikhi e Rizk [2], evidenziando come Minia sia in grado di assemblare il genoma umano completo con soli 5.7 GB di memoria (riduzione di 59× rispetto ad ABySS) e come la struttura cFP sia essenziale per garantire la correttezza dell'assemblaggio. In particolare, l'esperimento su *E. coli* (Sezione 5.7.3) ha dimostrato che il grafo probabilistico senza cFP produce

assemblaggi con oltre 3 kbp di differenze rispetto al riferimento, mentre Minia con cFP produce un assemblaggio identico.

6.1 Obiettivi della sperimentazione

In questo capitolo presentiamo una **sperimentazione pratica** del tool Minia su un dataset reale di *Escherichia coli*, con l'obiettivo di:

- Validare sperimentalmente i concetti teorici studiati nel Capitolo 5, osservando direttamente il comportamento delle strutture dati descritte
- Misurare il consumo di memoria, i tempi di esecuzione e la qualità dell'assemblaggio su un genoma batterico standard

La scelta di *E. coli* come organismo modello è motivata da:

- Dimensioni contenute del genoma (circa 4.6 Mbp), che permettono tempi di esecuzione ragionevoli su hardware consumer
- Disponibilità di dataset pubblici di alta qualità e di un genoma di riferimento ben annotato

L'analisi sperimentale si concentra in particolare su:

- tipologia e dimensione del dataset utilizzato;
- ambiente di esecuzione e parametri di configurazione;
- tempo di esecuzione e consumo di memoria;
- caratteristiche dell'assemblaggio prodotto;

6.2 Dataset utilizzato

Per la sperimentazione è stato scelto un dataset pubblico relativo a *Escherichia coli*, uno degli organismi modello più utilizzati in bioinformatica. *E. coli* è un batterio procariote con un genoma di dimensione relativamente ridotta (circa 4.6 Mbp) con

reads **Illumina** , ben annotato e ampiamente studiato, caratteristiche che lo rendono particolarmente adatto per esperimenti di assemblaggio *de novo*. Il dataset utilizzato è stato ottenuto dal **Sequence Read Archive (SRA)** del National Center for Biotechnology Information (NCBI) ed è identificato dall'accession number `SRR2584866`.I dati sono stati scaricati ed estratti tramite il tool **Faster Download and Extract Reads** in FASTQ, messo a disposizione dalla piattaforma Galaxy.

Il dataset originale è costituito da reads di sequenziamento paired-end, successivamente estratte in due file distinti (forward e reverse) in formato FASTQ. I file compressi sono stati decompattati per consentirne l'elaborazione nei passaggi successivi.

Poiché il tool di assemblaggio Minia accetta come input un singolo file di reads e non richiede informazioni esplicite di pairing, le reads forward e reverse sono state concatenate in un unico file FASTQ utilizzando il tool **Concatenate datasets** di Galaxy.

Il dataset finale ottenuto presenta le seguenti caratteristiche:

- **Formato**: fastqsanger.gz
- **Dimensione**: 586.2 MB
- **Contenuto**: reads forward + reverse concatenate

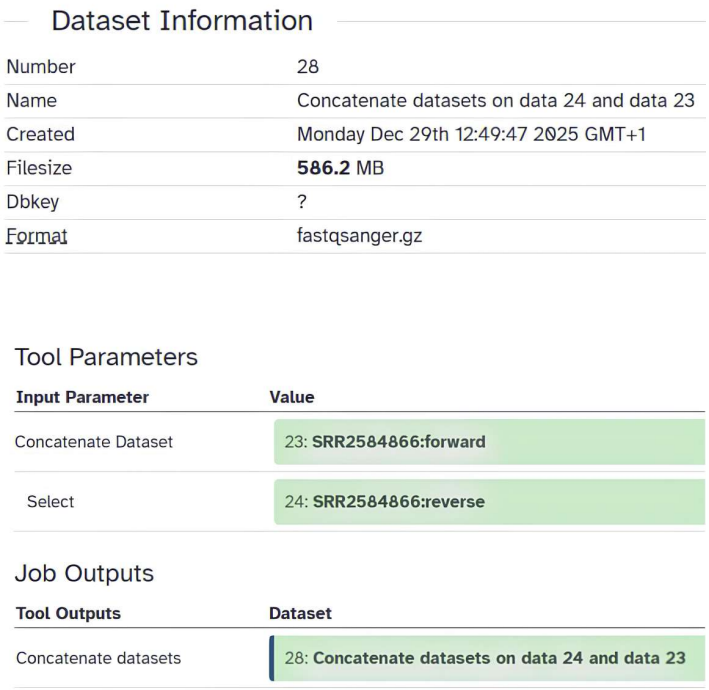


Figura 6.1: Dataset FASTQ concatenato utilizzato per l'assemblaggio

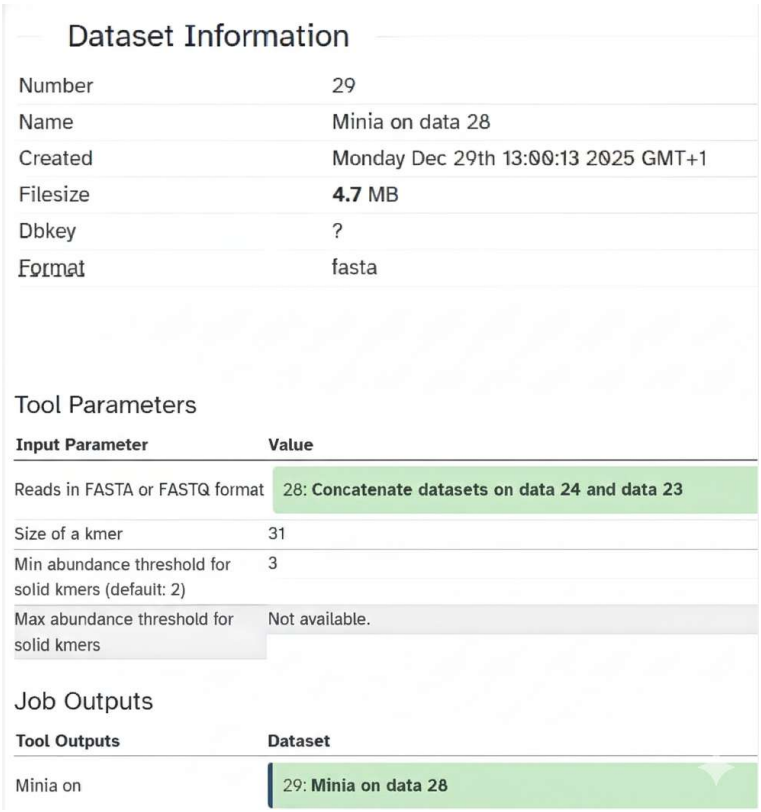
6.3 Ambiente di esecuzione e parametri

L'esperimento è stato condotto utilizzando la piattaforma **Galaxy**. Galaxy offre un ambiente di esecuzione controllato, riproducibile e dotato di strumenti per il monitoraggio delle risorse computazionali.

Il file FASTQ concatenato è stato fornito come input unico al tool **Minia**, che è stato eseguito con l'utilizzo dei seguenti parametri:

- **dimensione del k-mer**: k = 31
- **soglia minima di abbondanza per i k-mer solidi**: 3

- **formato di input**: FASTQ



Dataset Information

Number	29
Name	Minia on data 28
Created	Monday Dec 29th 13:00:13 2025 GMT+1
Filesize	**4.7** MB
Dbkey	?
Format	fasta

Tool Parameters

Input Parameter	Value
Reads in FASTA or FASTQ format	28: **Concatenate datasets on data 24 and data 23**
Size of a kmer	31
Min abundance threshold for solid kmers (default: 2)	3
Max abundance threshold for solid kmers	Not available.

Job Outputs

Tool Outputs	Dataset
Minia on	29: **Minia on data 28**

Figura 6.2: Impostazioni e input del tool Minia su Galaxy

6.3.1 Motivazione dei parametri scelti

La scelta di $k = 31$ rappresenta un compromesso tra specificità e robustezza agli errori di sequenziamento:

- **k troppo piccolo** (es. $k < 21$): i k-mer sono poco specifici e si generano molte ramificazioni spurie dovute a match casuali, aumentando i nodi complessi e la dimensione della marking structure
- **k troppo grande** (es. $k > 51$): aumenta la specificità ma si perdono connessioni reali tra contigs, riducendo la contiguità dell'assemblaggio (N50 più basso)

Come discusso precedentemente, Minia ha un comportamento logaritmico rispetto a k: raddoppiando k da 21 a 47, la memoria totale aumenta solo del 13.5% (da 12.6 a 14.3 bit/k-mer). Questa proprietà rende Minia poco sensibile alla scelta di k, permettendo di utilizzare valori relativamente alti senza penalità significative in memoria.

Il valore $k = 31$ è stato scelto in quanto:

- È coerente con i parametri utilizzati nel benchmark GAGE riportato nell'articolo [2]
- Rappresenta uno standard de facto nella comunità bioinformatica per genomi batterici
- Garantisce specificità sufficiente (circa $4^{31} \approx 4.6 \times 10^{18}$ k-mer possibili vs 4.6×10^6 basi del genoma)

La soglia di abbondanza minima $d = 3$ è anch'essa standard e corrisponde al valore utilizzato nell'articolo originale per il genoma umano [2]. Questa soglia permette di filtrare k-mer dovuti a errori di sequenziamento (che tipicamente appaiono una o due volte) mantenendo i k-mer reali del genoma.

6.4 Prestazioni computazionali

L'assemblaggio è stato completato con successo senza interruzioni o terminazioni forzate dei processi.

Le principali metriche di esecuzione sono le seguenti:

- **Tempo di esecuzione** (wall clock): circa 1 minuto
- **Tempo di CPU** (User time): circa 2 minuti
- **Core utilizzati**: 2
- **Parallelizzazione**: utilizzo efficiente di più core senza overhead evidente
- **Utilizzo massimo di memoria**: 4.5 GB
- **Processi terminati per out-of-memory (OOM)**:0

I dati evidenziano l'estrema rapidità dell'algoritmo Minia nel processare un dataset batterico standard.

Per quanto riguarda la memoria, il picco registrato (4.5 GB) rientra ampiamente nei limiti imposti dall'ambiente Galaxy (8 GB allocati). Va specificato che questo valore include l'overhead dell'ambiente di virtualizzazione; l'algoritmo Minia è noto in letteratura per un'impronta di memoria significativamente inferiore rispetto agli assemblatori basati su rappresentazioni esplicite del grafo, resa possibile dall'utilizzo della struttura dati probabilistica Bloom Filter per la rappresentazione del Grafo di de Bruijn, caratteristica che lo rende ideale anche per hardware con risorse limitate.

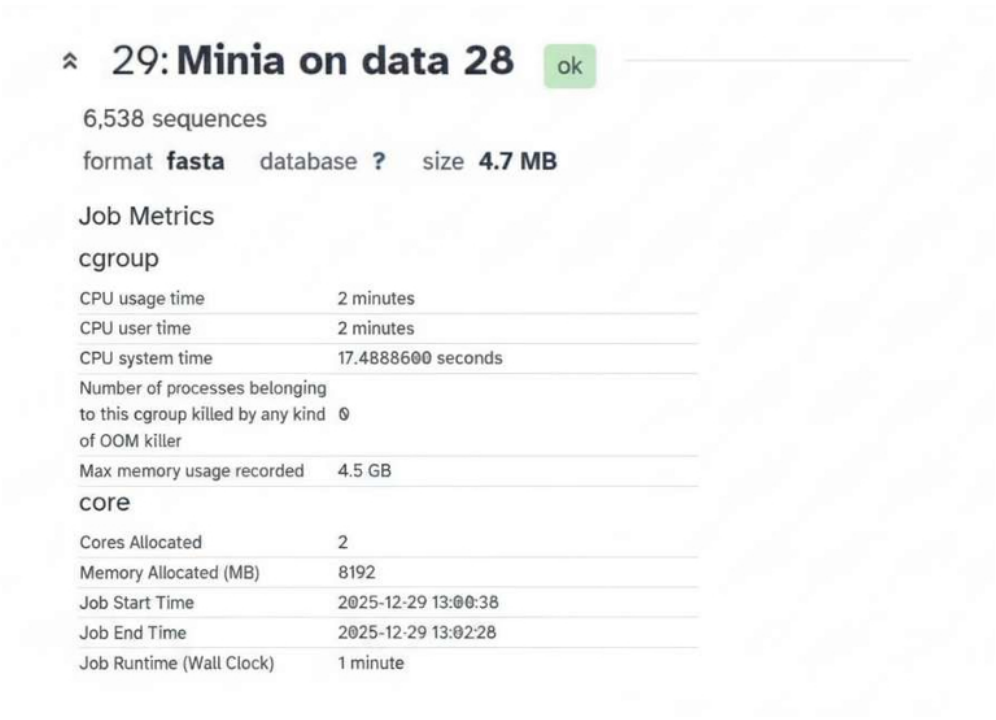


Figura 6.3: Metriche computazionali dell'esecuzione di Minia

6.4.1 Breakdown teorico della memoria

Sebbene l'ambiente Galaxy non fornisca un breakdown dettagliato della memoria per componente, possiamo stimare la distribuzione teorica basandoci sulle formule derivate nel Capitolo 5.

Assumendo che il dataset contenga circa $n \approx 4.6 \times 10^6$ k-mer solidi (stimato dalle dimensioni del genoma di *E. coli*), con $k = 31$ e tasso di falsi positivi ottimale

$F_{\text{opt}} = 2.08/(16 \times 31) \approx 0.0042$ (dalla formula della Sezione 5.5.3):

- **Bloom Filter**:

$$M_{\text{Bloom}} = 1.44 \times 4.6 \times 10^6 \times \log_2(1/0.0042) \approx 50.5 \text{ Mbit} \approx 6.3 \text{ MB}$$

(circa 11 bit/k-mer, come previsto teoricamente)

- **cFP**:

$$|\text{cFP}| \approx 8 \times 4.6 \times 10^6 \times 0.0042 \approx 154{,}000 \text{ k-mer}$$

$$M_{\text{cFP}} = 154{,}000 \times 62 \text{ bit} \approx 1.2 \text{ MB}$$

(circa 2.1 bit/k-mer)

- **Marking structure**: assumendo che circa l'1-2% dei k-mer siano nodi complessi (percentuale tipica per genomi batterici, come riportato nella Sezione 5.6.5 per *E. coli*):

$$n_c \approx 0.015 \times 4.6 \times 10^6 \approx 69{,}000 \text{ nodi}$$

$$M_{\text{marking}} = 69{,}000 \times 62 \text{ bit} \approx 0.5 \text{ MB}$$

(circa 0.93 bit/k-mer)

Memoria totale teorica:

$$M_{\text{tot}} = 6.3 + 1.2 + 0.5 \approx 8 \text{ MB per le strutture dati}$$

Il picco di 4.5 GB misurato da Galaxy include:

- Le strutture dati di Minia (8 MB)
- Il caricamento in memoria del file FASTQ di input (586 MB)
- Strutture ausiliarie per il k-mer counting e l'attraversamento
- Overhead dell'ambiente di virtualizzazione di Galaxy
- Buffer di sistema e allocazioni temporanee

Questa stima conferma che la rappresentazione compatta del grafo di de Bruijn occupa effettivamente una frazione molto piccola della memoria totale utilizzata, mentre la maggior parte della memoria è dedicata al processing dei dati di input. [2].

6.5 Qualità dell'assemblaggio e Validazione

Per valutare la qualità dell'assemblaggio ottenuto con Minia (dataset concatenato), è stato utilizzato il tool **QUAST**. A differenza di una semplice valutazione delle statistiche di lunghezza, in questa fase è stata eseguita una validazione biologica confrontando i contigs ottenuti con il genoma di riferimento ufficiale di *Escherichia coli* B str. REL606 (Accession: `CP000819` / `NC_012967.1`).

Le principali statistiche di validazione e qualità sono riportate di seguito:

- **Genome Fraction (%)**: 90.061%. Questo è il dato più significativo: indica che l'assemblaggio di Minia copre oltre il 90% del genoma di riferimento. Il restante 10% non ricostruito è probabilmente costituito da regioni altamente ripetitive difficili da risolvere con k-mer corti (31).

- **Misassemblies**: 2. Il numero di errori strutturali gravi è estremamente basso. Ciò dimostra l'elevata accuratezza dell'algoritmo basato sul Grafo di de Bruijn, che ha evitato di unire erroneamente regioni distanti del genoma.

- **Duplication Ratio**: 1.004. Un valore quasi perfetto (l'ideale è 1.0). Indica l'assenza di ridondanza artificiale: Minia non ha generato copie multiple delle stesse regioni.

- **Lunghezza Totale**: 4,170,116 bp. Confrontato con la lunghezza totale allineata (4,168,496 bp), conferma che quasi tutte le basi prodotte appartengono effettivamente a *E. coli*.

- **N50**: 7,274 bp. Il valore N50 indica che il 50% dell'intero assemblaggio è costituito da contigs lunghi almeno 7,274 basi.

- **Mismatches per 100 kbp**: 16.34. Il tasso di errore a livello di singola base è contenuto, confermando una buona qualità del consensus finale.

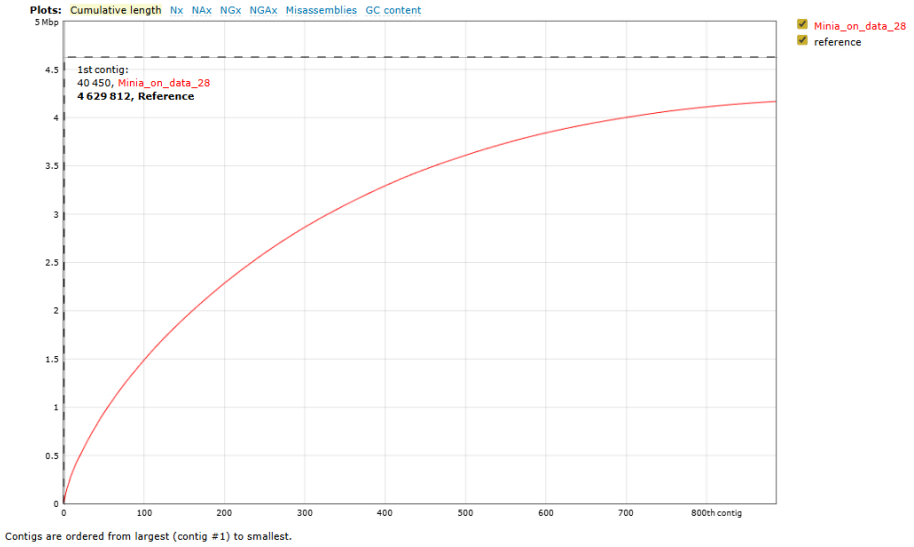


Figura 6.4: Cumulative Length vs Reference. La linea tratteggiata rappresenta la dimensione del genoma di riferimento. La curva rossa mostra come l'assemblaggio si avvicini asintoticamente alla completezza.

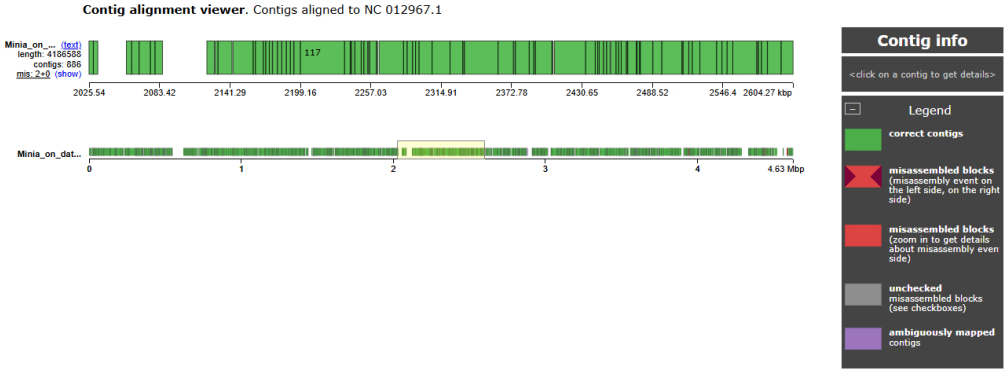


Figura 6.5: Icarus Contig Browser. Visualizzazione dell'allineamento dei contigs sul genoma di riferimento.

6.5.1 Analisi Grafica (Icarus Viewer)

La visualizzazione prodotta dal modulo **Icarus** permette di apprezzare la correttezza dell'assemblaggio rispetto al riferimento:

- **Blocchi Verdi**: Rappresentano i contigs corretti. La predominanza del colore verde e la continuità dei blocchi confermano che la maggior parte del genoma è stata ricostruita fedelmente.
- **Assenza di Blocchi Rossi**: I blocchi rossi indicherebbero misassemblaggi (inversioni o traslocazioni). La loro quasi totale assenza visiva rispecchia il dato numerico (solo 2 misassemblies), confermando l'affidabilità strutturale dell'output di Minia.

6.6 Discussione dei risultati

L'esperimento ha dimostrato l'efficacia del tool **Minia** nell'assemblaggio *de novo* di un genoma batterico.

I risultati ottenuti evidenziano due punti di forza principali:

1. **Elevata Accuratezza**: Con soli 2 errori di assemblaggio su oltre 4 milioni di basi e un *Duplication Ratio* di 1.004, Minia si è rivelato estremamente conservativo e preciso, evitando di introdurre artefatti comuni negli assemblatori basati su grafi.
2. **Buona Copertura Genomica**: Una *Genome Fraction* del 90.06% è un risultato notevole considerando l'utilizzo di un k-mer fisso (31) e l'assenza di uno step di *scaffolding*[1] avanzato. Le regioni mancanti sono verosimilmente attribuibili a sequenze ripetitive che eccedono la lunghezza del k-mer scelto.

In conclusione, l'approccio ha permesso di ricostruire la quasi totalità del genoma di *E. coli* REL606 con risorse computazionali minime, validando l'applicabilità di Minia e delle strutture dati succinte (Bloom Filter) anche in contesti di analisi rapida, confermando le considerazioni finali discusse nell'articolo originale [2].

[1] Lo scaffolding è il processo che ordina e orienta i contigs sfruttando le informazioni delle reads paired-end per colmare i gap (buchi) nella sequenza.

CAPITOLO 7

Discussione e Conclusioni

7.1 Sintesi del contributo

Questo lavoro ha affrontato il problema dell'assemblaggio genomico *de novo* attraverso l'analisi teorica e sperimentale di **Minia** (Chikhi e Rizk, 2013), un assemblatore che utilizza strutture dati probabilistiche per ridurre drasticamente il consumo di memoria.

Il Capitolo 5 ha analizzato l'architettura di Minia, dimostrando come tre componenti fondamentali cooperino per rappresentare implicitamente il De Bruijn Graph:

1. **Bloom Filter** (11 bit/k-mer): memorizza implicitamente i k-mer solidi con assenza di falsi negativi e probabilità controllata di falsi positivi. Minia supera il lower bound teorico di Conway e Bromage (6.8 GB) raggiungendo 5.7 GB per il genoma umano

2. **Struttura cFP** (2 bit/k-mer): corregge selettivamente solo i falsi positivi critici ($< 3\%$ dei falsi positivi totali) che alterano la topologia del grafo. Senza questa correzione, il grafo probabilistico puro produce assemblaggi con oltre 3 kbp di errori

3. **Marking structure** (4.4 bit/k-mer per genoma umano, 0.9 bit/k-mer per *E. coli*): traccia selettivamente solo i nodi complessi (6% per genoma umano, 1-2% per batteri), con costo proporzionale alla complessità topologica anziché alla dimensione del genoma

Il risultato è una memoria totale di 13.2 bit/k-mer, una riduzione di 59× rispetto ad ABySS (da 336 GB a 5.7 GB per il genoma umano).

Il Capitolo 6 ha validato sperimentalmente questi concetti su *E. coli* (dataset SRR2584866), confermando:

- Compattezza: strutture dati $\sim$8 MB ($<$ 2% della memoria totale)
- Correttezza: 90% copertura genomica, solo 2 misassemblies, duplication ratio 1.004
- Scalabilità: 1 minuto di esecuzione per 4.6 Mbp

7.2 Innovazione chiave: correzione selettiva dei falsi positivi

Il contributo principale di Minia è la **struttura cFP**, che risolve il trade-off tra efficienza in memoria e correttezza dell'assemblaggio. Anziché memorizzare esplicitamente tutti i k-mer (richiede memoria lineare) o tollerare tutti i falsi positivi (produce assemblaggi errati), Minia corregge *solo* i falsi positivi che creano ramificazioni spurie adiacenti a nodi reali.

Questa strategia è efficace perché:

- Solo il 2-3% dei falsi positivi sono critici (78.7M falsi positivi totali $\rightarrow$ 2.3M critici per genoma umano)
- La memoria aggiuntiva è proporzionale al tasso di falsi positivi F, non al numero totale di k-mer
- Esiste un tasso ottimale $F_{\text{opt}} = 2.08/(16k)$ che minimizza la memoria totale (Bloom + cFP)

L'esperimento su *E. coli* (Capitolo 6) ha confermato che questa correzione è essenziale: la qualità dell'assemblaggio ottenuto (solo 2 misassemblies su 4.17 Mbp) dimostra che la topologia del grafo è stata preservata correttamente.

7.3 Limiti e sviluppi futuri

7.3.1 Limiti principali

Regioni ripetute: Come tutti gli approcci basati su *short reads*, Minia non può risolvere ripetizioni più lunghe delle reads. Nel nostro esperimento, il 10% del genoma non assemblato è probabilmente costituito da sequenze ripetitive.

Dipendenza da *k*: Sebbene Minia abbia comportamento logaritmico rispetto a k (raddoppiare k aumenta la memoria solo del 13.5%), la scelta di k influenza la qualità dell'assemblaggio e richiede ottimizzazione manuale.

Assenza di scaffolding: Minia produce solo contigs, non scaffolds. L'integrazione di informazioni paired-end per lo scaffolding potrebbe migliorare significativamente la contiguità (N50).

7.3.2 Direzioni future

Approcci ibridi: Combinare *short reads* per la costruzione del grafo e *long reads* (PacBio/Nanopore) per risolvere regioni ripetute, mantenendo l'efficienza in memoria di Minia.

Grafi colorati per metagenomici: Estendere la struttura compatta (Bloom + cFP) a *colored De Bruijn graphs*, associando ogni k-mer al campione di origine per analisi metagenomiche con dataset multipli.

Ottimizzazione automatica: Implementare strategie adattive per la selezione di k basate su statistiche del dataset (coverage, tasso di errore).

7.4 Conclusioni

Questo lavoro ha dimostrato come l'integrazione tra teoria algoritmica (lower bound informazionale, analisi della complessità topologica) e strutture dati proba-

bilistiche (Bloom Filter con correzione selettiva) consenta di ridurre il consumo di memoria di due ordini di grandezza mantenendo correttezza e qualità dell'assemblaggio.

La validazione sperimentale su *E. coli* ha confermato le proprietà teoriche studiate: compattezza della rappresentazione (14 bit/k-mer misurati vs 15 bit/k-mer teorici), efficacia della correzione dei falsi positivi (solo 2 misassemblies), e scalabilità lineare (1 minuto per 4.6 Mbp).

Minia rappresenta una soluzione efficace per l'assemblaggio de novo su hardware consumer, democratizzando l'accesso all'analisi genomica in contesti con risorse computazionali limitate. Gli sviluppi futuri dovranno concentrarsi sull'integrazione di *long reads* e grafi colorati per dataset metagenomici, mantenendo il focus sull'efficienza computazionale che caratterizza l'approccio probabilistico.

Bibliografia

[1] L. Ermini and P. Driguez, "The application of long-read sequencing to cancer," *Cancers*, vol. 16, no. 7, 2024. [Online]. Available: https://www.mdpi.com/2072-6694/16/7/1275 (Citato alle pagine vi e 6)

[2] R. Chikhi and G. Rizk, "Space-efficient and exact de bruijn graph representation based on a bloom filter," *Algorithms for Molecular Biology*, 2013. [Online]. Available: https://link.springer.com/article/10.1186/1748-7188-8-22 (Citato alle pagine vi, 3, 28, 29, 30, 31, 32, 33, 34, 36, 38, 39, 40, 41, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 65, 66, 67, 68, 69, 70, 73, 78, 80 e 83)

[3] J. Shendure and H. Ji, "Next-generation dna sequencing," *Nature Biotechnology*, 2008. [Online]. Available: https://www.nature.com/articles/nbt1486 (Citato alle pagine 1, 4, 5 e 8)

[4] D. R. Zerbino and E. Birney1, "Velvet: Algorithms for de novo short read assembly using de bruijn graphs," *Genome Research*, 2008. [Online]. Available: https://genome.cshlp.org/content/18/5/821.full (Citato alle pagine 1 e 2)

[5] N. Nagarajan and M. Pop, "Sequence assembly demystified," *Nature Reviews Genetics*, 2013. [Online]. Available: https://www.nature.com/articles/nrg3367 (Citato alle pagine 1, 2, 6 e 8)

[6] P. E. C. Compeau, P. A. Pevzner, and G. Tesler, "How to apply de bruijn graphs to genome assembly," *Nature Biotechnology*, 2011. [Online]. Available: https://www.nature.com/articles/nbt.2023 (Citato alle pagine 2, 7 e 8)

[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 1970. [Online]. Available: https://dl.acm.org/doi/10.1145/362686.362692 (Citato alle pagine 2, 21, 22 e 25)

[8] E. Gecchele, M. Merlin, A. Brozzetti, A. Falorni, M. Pezzotti, and L. Avesani, "A comparative analysis of recombinant protein expression in different biofactories: bacteria, insect cells and plant systems," *J Vis Exp*, 2015. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/25867956/ (Citato alle pagine 6 e 10)

[9] C. Marchet *et al.*, "Data structures based on bloom filters for genomic data," *Briefings in Bioinformatics*, vol. 19, no. 6, pp. 1231–1245, 2018. (Citato a pagina 27)

[10] T. C. Conway and A. J. Bromage, "Succinct data structures for assembling large genomes," *Bioinformatics*, vol. 27, no. 4, pp. 479–486, 2011. (Citato a pagina 30)

[11] J. Pell, A. Hintze, R. Canino-Koning, A. C. Howe, J. M. Tiedje, and C. T. Brown, "Scaling metagenome sequence assembly with probabilistic de bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 109, pp. 13 272 – 13 277, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:7633380 (Citato alle pagine 32 e 36)

[12] E. R.Chikhi, G.Rizk, "Minia source code v2.0.7," *INRIA*, 2014. [Online]. Available: https://github.com/GATB/minia/releases/download/v2.0.7/minia-v2.0.7-Source.tar.gz (Citato alle pagine 65 e 71)