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 2024-2025

Abstract

Il sequenziamento genomico di nuova generazione (NGS) ha rivoluzionato la biologia molecolare, generando enormi 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 dell'approccio basato su De Bruijn Graph, con particolare attenzione alle strutture dati probabilistiche come i Bloom Filter.

L'assemblaggio genomico consiste nel ricostruire una sequenza completa a partire da milioni di frammenti di DNA (*reads*), affrontando sfide quali errori di sequenziamento, regioni ripetute e consumo elevato di memoria. L'approccio basato su De Bruijn Graph offre una complessità computazionale lineare $O(n \cdot L)$, superiore ai metodi tradizionali Overlap-Layout-Consensus, ma richiede memoria proporzionale al numero di k-mer unici, che può raggiungere ordini di grandezza impraticabili per genomi complessi.

Il progetto analizza in dettaglio l'assemblatore Minia, che utilizza i Bloom Filter per rappresentare implicitamente il grafo, riducendo il consumo di memoria del 90–95% rispetto alle strutture deterministiche. Viene esaminato il funzionamento dei Bloom Filter, strutture dati probabilistiche che garantiscono assenza di falsi negativi e probabilità controllata di falsi positivi, e le strategie implementate da Minia per gestire i falsi positivi critici che alterano la topologia del grafo.

La validazione sperimentale è stata condotta su un dataset reale di *Escherichia coli* (SRR2584866, 586.2 MB paired-end Illumina) utilizzando la piattaforma Galaxy. L'assemblaggio ha prodotto 4.17 Mbp con N50 di 7,274 bp, coprendo il 90.06% del genoma di riferimento con solo 2 misassemblaggi, in 1 minuto di esecuzione con un picco di 4.5 GB di RAM. I risultati confermano l'efficacia dell'approccio probabilistico nell'assemblaggio genomico, dimostrando come strutture dati teoricamente solide possano tradursi in strumenti pratici accessibili anche su hardware con risorse limitate.

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) [2]. 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 [3].

L'assemblaggio genomico rappresenta una delle principali sfide della bioinformatica moderna [4], 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 [3]. 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 [4]:

- 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 [5], 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 [5]. 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 [6], 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 [7];
- 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 [2].

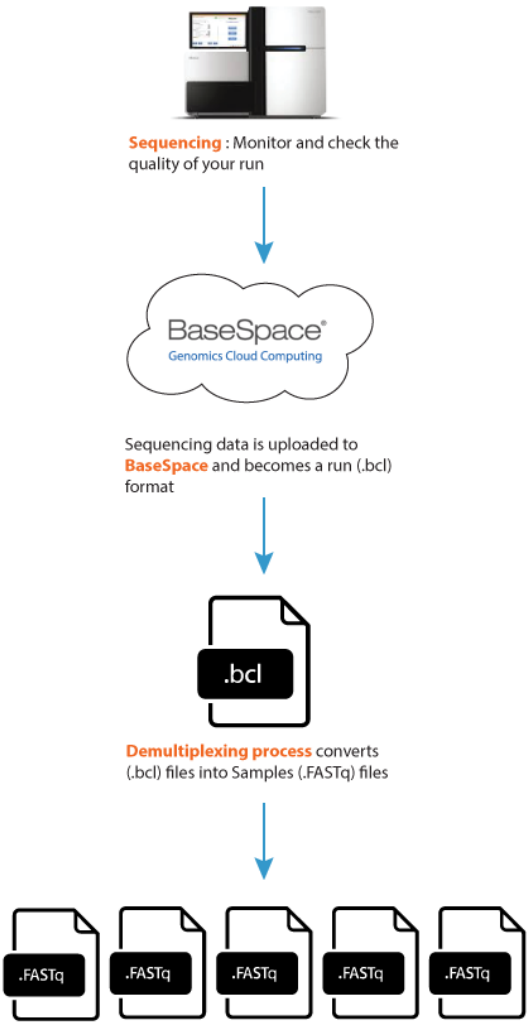


Figura 2.1: Schema generale del processo di NGS (dalla frammentazione alla generazione di reads).

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 [2]. 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 [4].

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].

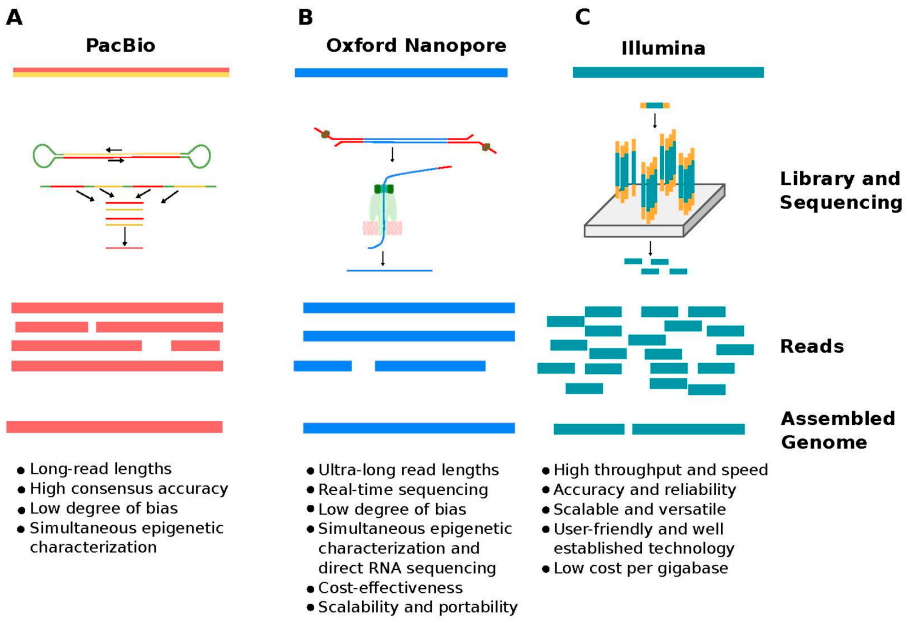


Figura 2.2: Confronto tra tecnologie di sequenziamento: lunghezza delle reads, accuratezza e applicazioni [1].

[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 [5].

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 [2].

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 [4].

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 [5].

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 (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 [6], 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[6].

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 [6]:

- 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.

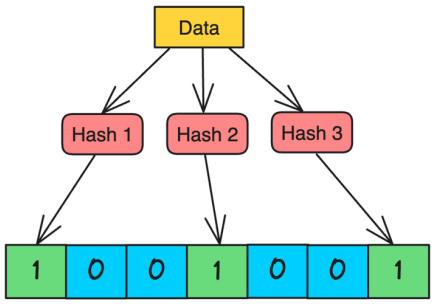


Figura 4.1: Struttura Bloom Filter.

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 [6]:

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

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 [6].

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) [7], 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 Che cos'è Minia

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**, 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.

5.2 De Bruijn Graph in Minia

Minia introduce una nuova rappresentazione del De Bruijn Graph progettata specificamente per ridurre drasticamente il consumo di memoria. L'idea centrale consiste nel codificare il grafo in modo implicito, evitando la memorizzazione esplicita di nodi e archi e sfruttando una combinazione di strutture dati probabilistiche e deterministiche.

5.2.1 Codifica implicita del grafo tramite Bloom Filter

Nel modello adottato da Minia, l'insieme dei nodi del De Bruijn Graph, ovvero l'insieme dei k-mer distinti estratti dalle reads, viene rappresentato tramite un Bloom Filter, dando vita a un **Probabilistic De Bruijn Graph** [7].

A livello implementativo:

- ogni k-mer viene convertito in una rappresentazione compatta (generalmente codifica binaria su 2 bit per base);
- il k-mer codificato viene inserito nel Bloom Filter utilizzando una famiglia di funzioni hash indipendenti;
- il Bloom Filter memorizza esclusivamente l'informazione di appartenenza, senza conservare il k-mer stesso.

In questo modo la presenza di un nodo nel grafo viene verificata interrogando il Bloom Filter, gli archi del grafo non sono memorizzati, ma ricostruiti dinamicamente verificando l'esistenza dei k-mer successivi ottenuti tramite shift della sequenza [7].

Questa rappresentazione consente di ridurre l'uso di memoria da decine di byte per k-mer a pochi bit per k-mer, rendendo possibile la gestione di dataset genomici molto grandi.

5.3 Ramificazioni spurie e falsi positivi critici

Nel contesto dei De Bruijn Graph, una **ramificazione** si verifica quando un nodo presenta più archi uscenti o entranti, indicando la presenza di percorsi alternativi nel grafo. Tali ramificazioni possono riflettere caratteristiche biologiche reali, come ripetizioni genomiche o variazioni strutturali, ma possono anche essere introdotte artificialmente da errori di sequenziamento o da approssimazioni nella rappresentazione del grafo.

Nel caso di Minia, l'utilizzo di un Bloom Filter per rappresentare implicitamente l'insieme dei k-mer introduce la possibilità di **falsi positivi**. Sebbene questi non compromettano la correttezza delle interrogazioni di appartenenza, essi possono generare **ramificazioni spurie**, ovvero archi del De Bruijn Graph che non esistono nel genoma reale ma derivano esclusivamente da k-mer erroneamente considerati presenti dal filtro.

In particolare, un falso positivo diventa problematico quando introduce una nuova ramificazione a partire da un nodo reale, alterando la topologia del grafo e ostacolando l'identificazione di cammini lineari durante l'assemblaggio. Tali falsi positivi vengono definiti **falsi positivi critici** (*critical false positives*) [7].

Per risolvere questo problema, Minia memorizza esplicitamente soltanto il sottoinsieme dei falsi positivi critici in una struttura ausiliaria separata. In questo modo, il grafo risultante mantiene una rappresentazione compatta grazie al Bloom Filter, ma preserva al contempo una topologia corretta, consentendo un attraversamento affidabile del De Bruijn Graph.

In particolare:

- durante la costruzione del grafo, per ogni nodo reale vengono analizzati i possibili successori;
- se un successore risulta presente nel Bloom Filter ma non nei dati reali esso introduce una biforcazione e viene identificato come **falso positivo critico**;
- questi k-mer vengono memorizzati esplicitamente in una struttura separata (tipicamente una hash table compatta).

Questa struttura è estremamente ridotta in dimensione, poiché il numero di falsi positivi critici è molto piccolo rispetto al numero totale di k-mer.

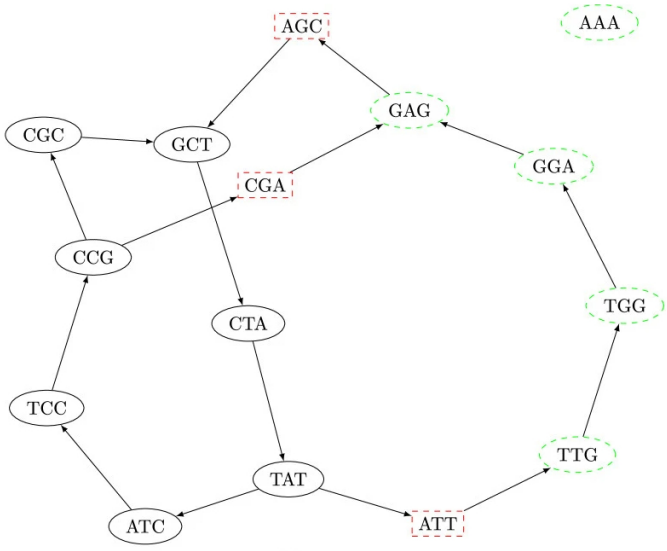


Figura 5.1: Esempio di De Bruijn Graph e della sua rappresentazione probabilistica tramite Bloom Filter.

I nodi circolari pieni rappresentano i k-mer realmente presenti nel grafo, mentre i nodi tratteggiati indicano k-mer che risultano presenti nel Bloom Filter a causa di

falsi positivi. Tra questi, i **falsi positivi critici** sono quelli adiacenti a nodi reali e che introducono **ramificazioni spurie** nel grafo probabilistico. L'identificazione e la memorizzazione separata di tali nodi consente a Minia di ottenere una rappresentazione del grafo corretta ed efficiente in termini di memoria. Immagine adattata da Chikhi e Rizk (2013) [7].

5.4 Struttura di marcatura

Un ulteriore problema riguarda l'attraversamento del grafo; durante l'assemblaggio è necessario evitare di visitare più volte gli stessi nodi, per prevenire cicli e ridondanze.

Minia introduce quindi una struttura di marcatura (*marking structure*)[7] progettata per essere anch'essa efficiente in termini di spazio.

A livello implementativo:

- non viene mantenuto un flag di visita per tutti i k-mer;
- la struttura di marcatura memorizza informazioni solo per un sottoinsieme dei nodi, tipicamente quelli coinvolti nei cammini di assemblaggio;
- la marcatura avviene dinamicamente durante il traversal[2], aggiornando la struttura solo quando necessario.

Questa scelta consente di evitare strutture di marcatura dense e costose e mantenere la complessità in memoria proporzionale al numero di nodi effettivamente attraversati.

5.5 Combinazione delle strutture

La combinazione delle tre strutture dati:

- **Bloom Filter** per rappresentare l'insieme dei k-mer;

[2]Con il termine attraversamento (traversal) si indica il processo mediante il quale Minia esplora il De Bruijn Graph, ricostruendo dinamicamente nodi e archi per generare l'output a partire dai k-mer presenti

- **Struttura dei falsi positivi critici** per eliminare ramificazioni spurie;
- **Struttura di marcatura** per controllare l'attraversamento,

consente a Minia di ottenere una rappresentazione del De Bruijn Graph che è probabilistica nella memorizzazione deterministica nel risultato finale.

Questa architettura permette interrogazioni rapide (tempo costante), un consumo di memoria ridotto di un ordine di grandezza rispetto agli approcci tradizionali e un assemblaggio corretto, equivalente a quello ottenuto con strutture deterministiche [7].

5.6 Analisi del codice di Minia

5.6.1 Linguaggio e dipendenze

Minia è implementato in **C++**, scelta che consente un controllo sull'utilizzo della memoria assicurando prestazioni elevate, requisiti fondamentali per l'assemblaggio di grandi dataset genomici.

Il tool fa parte dell'ecosistema **GATB** (Genome Assembly Tool Box), una libreria modulare progettata per lo sviluppo di algoritmi di analisi genomica efficienti in termini di tempo e spazio. GATB fornisce implementazioni ottimizzate di strutture dati fondamentali, tra cui Bloom Filter, gestione compatta dei k-mer e iterazione su grandi collezioni di sequenze.

L'uso di GATB consente a Minia di astrarre molti dettagli implementativi di basso livello, mantenendo al contempo un'implementazione aderente ai principi teorici descritti nell'articolo di riferimento [7].

5.6.2 Architettura del tool

Dal punto di vista software, Minia è strutturato come una pipeline di assemblaggio composta da fasi ben definite, ciascuna delle quali riflette un passaggio concettuale dell'algoritmo descritto in letteratura.

Le principali fasi dell'esecuzione sono:

- Lettura e parsing delle reads di input.

- Estrazione e filtraggio dei k-mer.
- Costruzione del grafo di De Bruijn probabilistico.
- Individuazione dei falsi positivi critici.
- Attraversamento del grafo.
- Generazione dell'output.

5.6.3 Dati in input

Minia accetta in input reads di sequenziamento in formato FASTA o FASTQ, provenienti da tecnologie di sequenziamento di nuova generazione.

A partire dalle reads, Minia estrae tutti i possibili k-mer di lunghezza fissata k, che costituiscono i nodi del De Bruijn Graph. La scelta del parametro k influenza direttamente la struttura del grafo e la qualità dell'assemblaggio finale.

5.6.4 Estrazione e gestione dei k-mer

L'estrazione dei k-mer è delegata alla libreria **GATB**, che fornisce iteratori efficienti per scorrere le reads e generare i k-mer sovrapposti di lunghezza k. In particolare, questa funzionalità è realizzata tramite le classi del modulo `kmer`[10] e viene utilizzata durante la fase di costruzione del grafo.

Un esempio semplificato del codice utilizzato per iterare sui k-mer è riportato di seguito:

```
typedef Kmer<>::ModelDirect Model;
Model model(k);

Iterator<Sequence>* itSeq = bank->iterator();
for (itSeq->first(); !itSeq->isDone(); itSeq->next()) {
    const Sequence& seq = itSeq->item();
    Model::Iterator itKmer(model, seq);

    for (itKmer.first(); !itKmer.isDone(); itKmer.next()) {
```

```
        const Model::Kmer& kmer = itKmer.item();
        // utilizzo del k-mer
    }
}
```

Il codice mostra come Minia utilizzi un iteratore sui k-mer senza mai materializzare l'intero insieme in memoria. Ogni k-mer viene generato dinamicamente a partire dalla sequenza corrente e reso immediatamente disponibile per le fasi successive dell'algoritmo.

Minia applica inoltre 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.

5.6.5 Costruzione del grafo di De Bruijn probabilistico

La gestione del Bloom Filter e delle operazioni di inserimento e interrogazione è implementata tramite le strutture fornite da GATB [10]. Una volta estratto, ciascun k-mer viene immediatamente inserito nel Bloom Filter che rappresenta l'insieme dei nodi del De Bruijn Graph.

Concettualmente, l'inserimento assume la forma seguente:

```
bloomFilter.insert(kmer);
```

La presenza di un nodo nel grafo è determinata esclusivamente dall'interrogazione del Bloom Filter.

Gli archi del De Bruijn Graph non sono memorizzati, ma ricostruiti dinamicamente interrogando il Bloom Filter sui k-mer adiacenti.

Uno schema semplificato per la verifica dell'esistenza degli archi:

```
for (char nucleotide : {'A','C','G','T'}) {
    Kmer next = current.extendRight(nucleotide);
    if (bloomFilter.contains(next)) {
        // arco implicito u -> next
```

```
    }
}
```

Questo frammento evidenzia un aspetto cruciale dell'approccio di Minia: la verifica dell'esistenza di un arco equivale a una semplice interrogazione del Bloom Filter, operazione a tempo costante.

5.6.6 Gestione dei falsi positivi critici

Poiché il Bloom Filter può restituire falsi positivi, la costruzione del grafo probabilistico è strettamente collegata alla fase di identificazione dei *falsi positivi critici*. Minia affronta questo problema identificando e memorizzando esplicitamente, quest'ultimi, all'intero di un'apposita struttura, garantendo così una topologia del grafo equivalente a quella deterministica.

```
for (char nucleotide : {'A','C','G','T'}) {
    Kmer neighbor = current.extendRight(nucleotide);
    if (bloomFilter.contains(neighbor)) {
        if (!solidKmers.contains(neighbor)) {
            criticalFalsePositives.insert(neighbor);
        }
    }
}
```

In questo frammento:

- `bloomFilter.contains()` verifica l'appartenenza probabilistica;
- `solidKmers` rappresenta l'insieme deterministico dei k-mer validi;
- `criticalFalsePositives` è la struttura che memorizza esclusivamente i falsi positivi critici.

I falsi positivi critici vengono memorizzati in una struttura compatta, tipicamente una hash table o un set ordinato, il cui costo in memoria è trascurabile rispetto alla dimensione del Bloom Filter. Come evidenziato nell'articolo [7], il numero di tali elementi è estremamente ridotto rispetto al numero totale di k-mer.

5.6.7 Struttura di marcatura e attraversamento del grafo

Successivamente, Minia procede con l'attraversamento del grafo per generare l'output. Per fare ciò viene utilizzata una **struttura di marcatura compatta**.

Nel codice di Minia[10], la struttura di marcatura è implementata come una collezione compatta di identificatori di k-mer, spesso basata su hash set.

Uno schema semplificato del comportamento è il seguente:

```
if (markedKmers.contains(currentKmer)) {
    return;
}
markedKmers.insert(currentKmer);
extendPath(currentKmer);
```

In questo frammento:

- `markedKmers` rappresenta la struttura di marcatura;
- il controllo di appartenenza impedisce visite multiple;
- la marcatura avviene solo quando il nodo è effettivamente attraversato.

Come evidenziato da Chikhi e Rizk [7], il numero di nodi effettivamente marcati è molto inferiore al numero complessivo di k-mer, rendendo il costo della struttura trascurabile rispetto al Bloom Filter.

5.6.8 Output

L'output principale di Minia consiste in una collezione di **contig**, ovvero sequenze genomiche ottenute mediante l'attraversamento dei cammini non ambigui del De Bruijn Graph.

I **contig** sono generalmente forniti in formato FASTA e rappresentano porzioni ricostruite del genoma di partenza. La qualità e la lunghezza dei contig dipendono da diversi fattori, tra cui la copertura del sequenziamento, la presenza di errori nelle reads e la scelta del parametro k.

CAPITOLO 6

Esperimenti e testing

In questo capitolo viene presentata una sperimentazione pratica del tool **Minia**, con l'obiettivo di valutare il comportamento dell'assemblatore su un dataset reale e verificare la coerenza dei risultati ottenuti con quanto discusso nei capitoli precedenti e riportato nell'articolo di riferimento.

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.

6.1 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.

La scelta di un dataset pubblico e standard consente inoltre di garantire la riproducibilità dell'esperimento e di confrontare i risultati ottenuti con quelli presenti in letteratura.

Il dataset finale ottenuto presenta le seguenti caratteristiche:

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

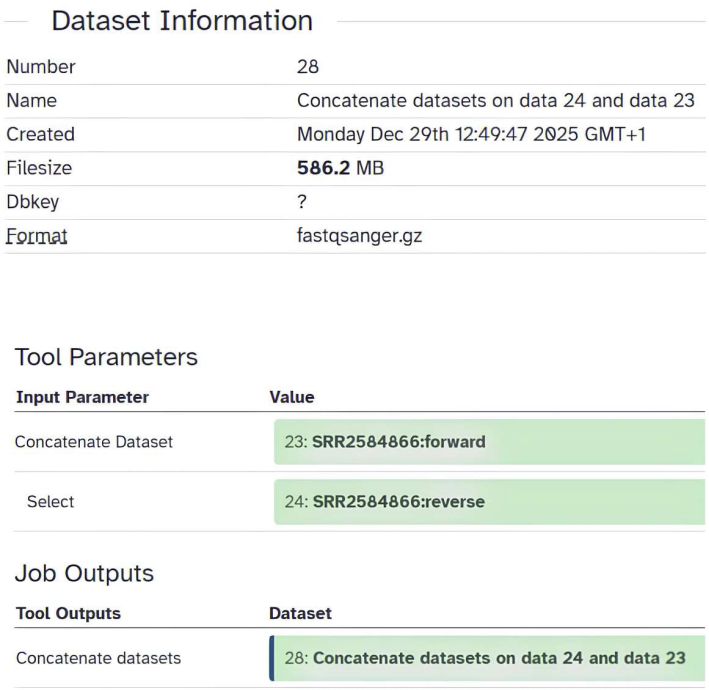


Figura 6.1: Dataset FASTQ concatenato utilizzato per l'assemblaggio

6.2 Ambiente di esecuzione e parametri

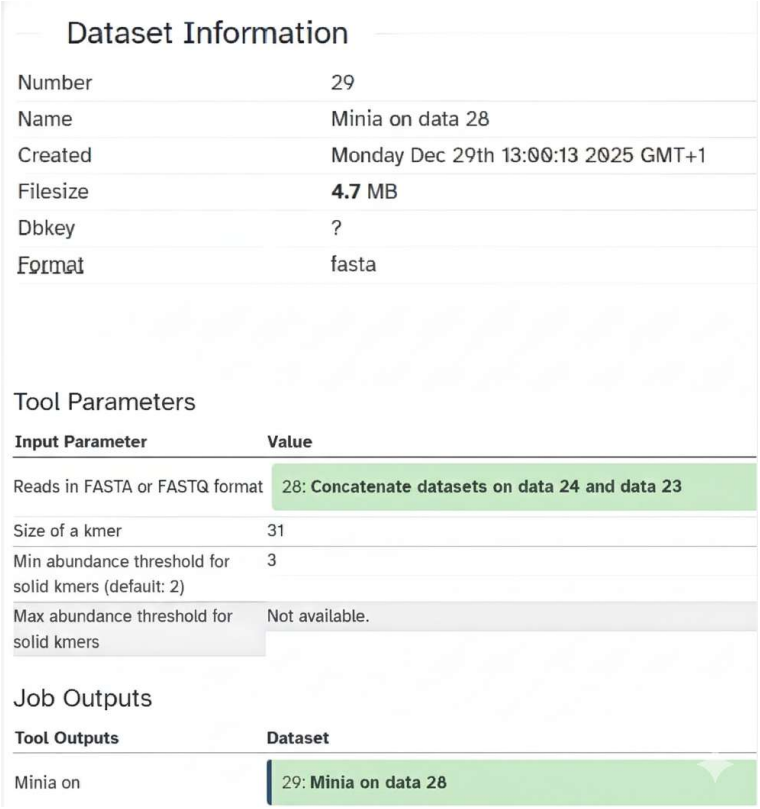
L'esperimento è stato condotto utilizzando la piattaforma **Galaxy**. Galaxy offre un ambiente di esecuzione controllato, riproducibile e dotato di strumenti per il monitoraggio delle risorse computazionali.

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

La scelta di $k = 31$ rappresenta un compromesso comune tra specificità e robustezza agli errori di sequenziamento ed è coerente con i parametri utilizzati nell'articolo originale di Minia[7].



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 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.

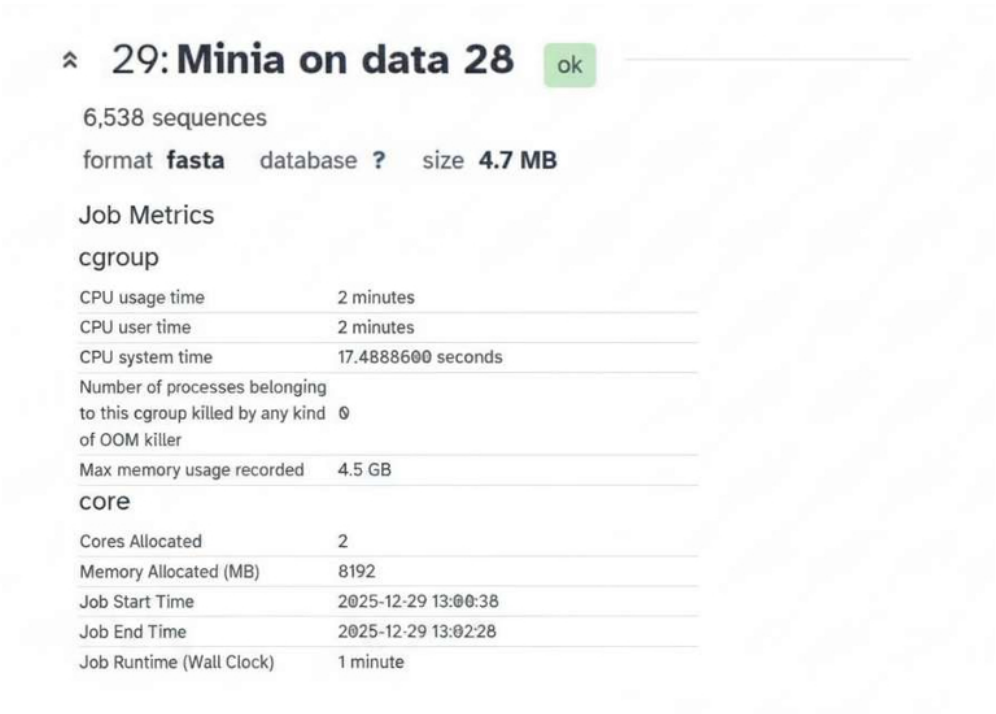


Figura 6.3: Metriche computazionali dell'esecuzione di Minia

6.4 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.

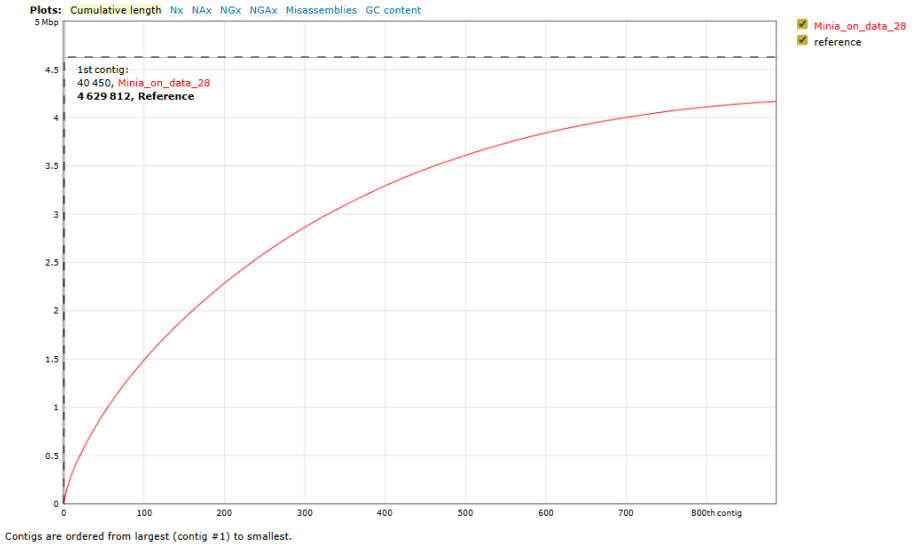


Figura 6.4: Cumulative Length vs Reference. La linea tratteggiata rappresenta la dimensione del genoma di riferimento. La curva rossa mostra come l'assemblaggio si avvicini asintoticamente alla completezza.

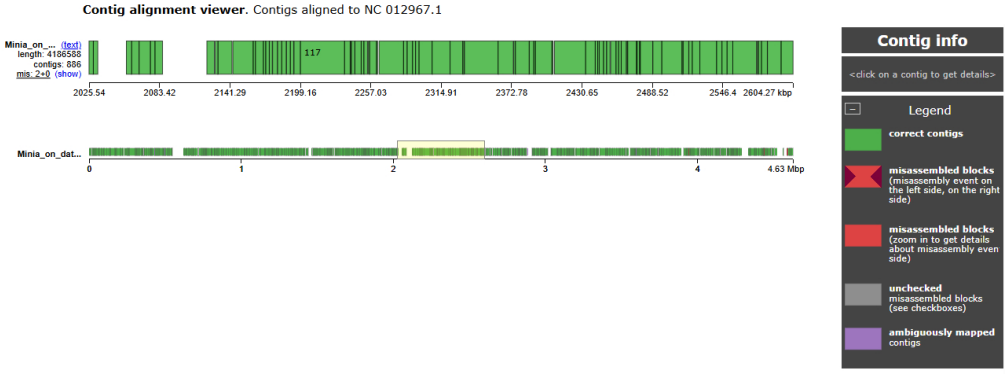


Figura 6.5: Icarus Contig Browser. Visualizzazione dell'allineamento dei contigs sul genoma di riferimento.

6.4.1 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.

6.5 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.

[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 Riassunto del lavoro svolto

Questo lavoro ha affrontato il problema dell'assemblaggio genomico *de novo* attraverso un'analisi sistematica che parte dalle tecnologie di sequenziamento e arriva all'implementazione pratica di soluzioni algoritmiche avanzate.

Il progetto si è articolato su più livelli. Inizialmente è stato introdotto il contesto del sequenziamento NGS, analizzando le differenze tra tecnologie *short-read* (Illumina) e *long-read* (PacBio, Nanopore), l'impatto degli errori di sequenziamento sui k-mer e le strategie di preprocessing. È stato dimostrato come un singolo errore in una read generi fino a k k-mer errati, evidenziando la criticità del quality filtering per ridurre la complessità del grafo.

Successivamente, il problema dell'assemblaggio è stato formalizzato come ricostruzione di una sequenza genomica da un multiset di reads affette da errori, regioni ripetute e copertura non uniforme. L'approccio basato su De Bruijn Graph è stato confrontato con OLC, dimostrando una complessità computazionale lineare $O(n \cdot L)$ contro quella quadratica $O(n^2)$ di OLC, rendendolo scalabile per dataset NGS. Tuttavia, l'analisi ha evidenziato il limite critico: la rappresentazione esplicita del grafo

richiede memoria proporzionale al numero di *k*-mer unici, stimata in circa 260 MB per *E. coli* e fino a 3.5 TB per dati metagenomici.

La soluzione al problema della scalabilità è stata individuata nei Bloom Filter, strutture dati probabilistiche che garantiscono assenza di falsi negativi, probabilità controllata di falsi positivi $p = (1 - e^{-kn/m})^k$, e complessità $O(k)$ per inserimento e query. Questa struttura consente una riduzione del 90–95% della memoria rispetto a hash table esplicite.

L'analisi si è poi concentrata su Minia, che implementa un De Bruijn Graph implicito tramite Bloom Filter. L'algoritmo introduce due strutture ausiliarie: una per gestire i falsi positivi critici (< 0.1% dei nodi) che alterano la topologia del grafo, e una struttura di marcatura compatta per controllare l'attraversamento. L'analisi del codice ha rivelato un'architettura modulare basata su GATB, con separazione netta tra costruzione del grafo e traversal.

La validazione sperimentale è stata condotta su dataset reale (SRR2584866, *E. coli*, 586.2 MB *paired-end* Illumina). L'assemblaggio ha prodotto 4.17 Mbp con N50 di 7,274 bp in 1 minuto con picco di 4.5 GB RAM. Il confronto con il genoma di riferimento (CP000819 NC_012967.1) ha rivelato una copertura del 90.06%, solo 2 misassemblaggi e un duplication ratio di 1.004, dimostrando elevata efficienza computazionale e correttezza biologica.

7.2 Punti di forza dell'approccio

L'approccio basato su Bloom Filter per la rappresentazione dei De Bruijn Graph presenta vantaggi significativi rispetto alle soluzioni deterministiche.

Efficienza in memoria: La riduzione del consumo di memoria del 90–95% rende possibile l'assemblaggio di genomi complessi su hardware consumer. Per *E. coli*, il picco misurato di 4.5 GB (incluso overhead di virtualizzazione) conferma la scalabilità dell'algoritmo anche per dataset di dimensioni superiori.

Correttezza algoritmica: L'assenza di falsi negativi nei Bloom Filter garantisce che nessun *k*-mer reale venga erroneamente escluso dal grafo. L'introduzione della struttura dei falsi positivi critici elimina le ramificazioni spurie, preservando una topologia equivalente a quella deterministica.

Velocità di esecuzione: La complessità lineare della costruzione del grafo ($O(n \cdot L)$) e la complessità costante $O(k)$ per le query sui k-mer consentono tempi di assemblaggio estremamente ridotti. L'esperimento su *E. coli* ha richiesto solo 1 minuto di wall-clock time, evidenziando l'efficienza pratica dell'implementazione.

Modularità e riproducibilità: L'integrazione con GATB fornisce un'architettura software ben strutturata, riutilizzabile per lo sviluppo di ulteriori tool bioinformatici. L'uso di dataset pubblici (SRA) e parametri standard ($k = 31$) garantisce la riproducibilità degli esperimenti.

7.3 Limiti del metodo

Nonostante i vantaggi, l'approccio presenta alcune limitazioni intrinseche.

Natura probabilistica: I Bloom Filter introducono falsi positivi con probabilità $p = (1 - e^{-kn/m})^k$. Sebbene la struttura ausiliaria dei falsi positivi critici risolva le ambiguità topologiche, il loro numero dipende dalla scelta dei parametri m (dimensione del bit array) e k (numero di funzioni hash). Una parametrizzazione non ottimale può aumentare sia la probabilità di falsi positivi sia il costo delle strutture ausiliarie.

Dipendenza dal parametro k: La scelta della lunghezza del k-mer influenza direttamente la qualità dell'assemblaggio. Valori piccoli di k generano grafi complessi con molte ambiguità, mentre valori grandi frammentano il grafo perdendo sovrapposizioni. Minia richiede una scelta manuale di k (es. $k = 31$), che potrebbe non essere ottimale per tutti i dataset.

Gestione delle regioni ripetute: Come tutti gli approcci basati su De Bruijn Graph con *short reads*, Minia fatica a risolvere ripetizioni più lunghe delle reads. Nel genoma umano, circa il 45% della sequenza è costituito da elementi ripetuti, limitando la continuità dei contig. L'assemblaggio di *E. coli*, pur raggiungendo il 90.06% di copertura genomica, ha mostrato frammentazione residua dovuta principalmente a ripetizioni.

Accuratezza strutturale: Nonostante l'elevata qualità complessiva (solo 2 misassemblaggi su 4.17 Mbp), la validazione ha evidenziato 16.34 mismatches per 100 kbp e alcune discontinuità strutturali. Questi errori, sebbene contenuti, possono limitare

l'applicabilità in contesti che richiedono assemblaggio di qualità *reference-grade*[1] .

7.4 Possibili sviluppi futuri

Diverse direzioni di ricerca possono estendere e migliorare l'approccio analizzato.

Integrazione con long reads: L'approccio ibrido, combinando *short reads* accurate (Illumina) per la costruzione del grafo e *long reads* (PacBio/Nanopore) per lo scaffolding, potrebbe risolvere regioni ripetute mantenendo l'efficienza in memoria di Minia. Questo richiederebbe l'estensione dell'algoritmo per gestire reads eterogenee.

Ottimizzazione automatica dei parametri: L'implementazione di strategie adattive per la selezione di k, basate su statistiche del dataset (copertura, tasso di errore, lunghezza delle reads), potrebbe migliorare la qualità dell'assemblaggio senza intervento manuale.

Strutture dati alternative: L'esplorazione di varianti dei Bloom Filter (es. Counting Bloom Filter per gestire eliminazioni, Quotient Filter per locality) o strutture più recenti come Bloom Filter Cascade potrebbe ulteriormente ridurre il consumo di memoria o migliorare la gestione dei falsi positivi.

Validazione su dataset complessi: Estendere la sperimentazione a genomi eucariotici (es. *Drosophila*, *Arabidopsis*) e dati metagenomici permetterebbe di valutare la scalabilità e robustezza dell'approccio in scenari più sfidanti.

7.5 Considerazioni finali

Questo lavoro ha dimostrato come l'integrazione tra teoria algoritmica (De Bruijn Graph, cammini euleriani) e strutture dati probabilistiche (Bloom Filter) consenta di affrontare il problema dell'assemblaggio genomico in modo scalabile ed efficiente. Minia rappresenta un esempio paradigmatico di come soluzioni teoricamente solide

[1]Con il termine *reference-grade assembly* si indica un assemblaggio genomico di qualità comparabile a un genoma di riferimento ufficiale, caratterizzato da un'elevata continuità, un numero minimo di misassemblaggi e un basso tasso di errori di sequenza, tale da consentirne l'utilizzo come riferimento biologico affidabile.

possano tradursi in strumenti pratici per l'analisi di dati biologici su hardware accessibile.

La sperimentazione condotta su *E. coli* ha confermato la capacità di Minia di ricostruire genomi batterici con alta fedeltà (4.17 Mbp, 90.06% di copertura, solo 2 misassemblaggi) e tempi di esecuzione ridotti (1 minuto, 4.5 GB RAM), validando l'approccio proposto da Chikhi e Rizk (2013). Tuttavia, i limiti evidenziati suggeriscono che l'assemblaggio genomico rimane un problema aperto, richiedente approcci ibridi e integrazione di tecnologie complementari.

In un'era in cui il sequenziamento ad alta produttività genera volumi di dati sempre crescenti, lo sviluppo di algoritmi *memory-efficient* come Minia risulta essenziale per democratizzare l'accesso all'analisi genomica, abilitando ricerche in contesti con risorse computazionali limitate. Gli sviluppi futuri dovranno concentrarsi sull'integrazione di *long reads*, ottimizzazione automatica dei parametri e validazione su dataset complessi, 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 v e 6)

[2] 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)

[3] 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)

[4] 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)

[5] 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)

[6] 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)

[7] 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 3, 28, 29, 30, 32, 33, 36, 37 e 41)

[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] R. Chikhi and G. Rizk, "Minia: source code," 2013. [Online]. Available: https://github.com/GATB/minia (Citato alle pagine 34, 35 e 37)