



UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

STRUMENTI FORMALI PER LA BIOINFORMATICA

PROGETTO

Metodi Efficienti per il Trattamento di Sequenze Genomiche: Un'Analisi di Kallisto, Minimap e Miniasm

PROFESSORI

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

CANDIDATO

Alessandro Carnevale
MATRICOLA: 0522501994

Indice

1	Introduzione	2
1.1	Panoramica e Motivazioni	2
1.2	Contesto biologico	3
1.2.1	Quantificazione dei Trascritti da RNA-seq	3
1.2.2	Assemblaggio del genoma	4
2	Revisione della Letteratura	5
2.1	Metodi Alignment-based	5
2.2	Metodi Alignment-free	6
2.2.1	Metodi basati sulla frequenza di parole (word-based)	6
2.2.2	Metodi basati sulla teoria dell'informazione	7
3	Metodologia	10
3.1	Kallisto	11
3.2	Minimap	17
3.3	Miniasm	24
4	Risultati	30
4.1	Kallisto	31
4.2	Minimap	38
4.3	Miniasm	41
5	Discussione	43
6	Conclusioni	45

Capitolo 1

Introduzione

1.1 Panoramica e Motivazioni

L’analisi delle sequenze genomiche è un campo fondamentale della bioinformatica, essenziale per comprendere la struttura e la funzione del DNA e dell’RNA. Con l’aumento della quantità di dati prodotti dalle moderne tecnologie di sequenziamento, è diventato sempre più necessario sviluppare strumenti computazionali efficienti in grado di elaborare e interpretare tali dati in tempi rapidi e con elevata accuratezza. Tradizionalmente, il confronto e l’analisi delle sequenze sono stati condotti attraverso metodi alignment-based, come *BWA* [10] e *Bowtie2* [8]. Tuttavia, questi approcci possono risultare onerosi in termini di risorse computazionali, soprattutto quando si analizzano dataset di grandi dimensioni. Per superare tali limiti, sono stati sviluppati metodi *alignment-free* [15, 14], che offrono un’alternativa più efficiente per alcune applicazioni bioinformatiche. Gli approcci alignment-free, tra cui quelli basati su k-mer e minimizer, riducono la necessità di calcolare corrispondenze base per base, migliorando così le prestazioni in termini di velocità e utilizzo di memoria. Questa tesi si propone di analizzare tre strumenti innovativi che sfruttano tali tecniche: **Kallisto** [2], **Minimap** e **Miniasm** [9]. Kallisto è un software per la quantificazione dell’espressione genica basato sul concetto di *pseudoallineamento*, che consente una rapida analisi dei dati RNA-seq [12]. Minimap è un mapper di letture lunghe che utilizza minimizer per un’efficace individuazione di regioni omologhe nel genoma. Miniasm, invece, è un assembler de novo che, a differenza delle pipeline tradizionali, evita la fase di correzione degli errori, accelerando notevolmente il processo di assemblaggio. L’obiettivo della tesi è valutare l’efficienza e l’accuratezza di questi strumenti, confrontandoli con approcci tradizionali e discutendone i potenziali sviluppi futuri.

1.2 Contesto biologico

1.2.1 Quantificazione dei Trascritti da RNA-seq

L’analisi del trascrittoma (l’insieme di tutti gli RNA presenti in un campione) è fondamentale per comprendere come le differenze nell’espressione genica contribuiscano a variazioni fenotipiche, come malattie o adattamenti biologici. Per decenni, i *microarray* sono stati lo strumento principale per questo scopo, misurando l’abbondanza di mRNA tramite ibridazione con sonde specifiche. Tuttavia, con l’avvento del sequenziamento di nuova generazione (NGS), l’RNA-seq è emerso come alternativa più potente e versatile. L’analisi dell’RNA-seq si basa sul sequenziamento di frammenti di RNA convertiti in cDNA, generando milioni di letture corte (*short reads*) che devono essere successivamente analizzate per determinare il livello di espressione dei trascritti [12]. Tuttavia, la quantificazione accurata dell’abbondanza dei trascritti rappresenta una sfida complessa a causa di diversi fattori. Uno dei problemi principali è la frammentazione delle sequenze, che rende necessario il riconoscimento dell’origine di ciascuna lettura. Questo diventa particolarmente difficile in presenza di splicing alternativo, dove lo stesso gene può produrre più isoforme che condividono regioni comuni. Di conseguenza, l’assegnazione delle letture a una specifica isoforma è intrinsecamente ambigua. Un ulteriore ostacolo è rappresentato dalla variazione nella lunghezza dei trascritti. Trascritti più lunghi hanno una probabilità maggiore di essere coperti da più letture rispetto a trascritti più corti, anche se questi ultimi hanno livelli di espressione simili. Questo può introdurre bias nei conteggi e rendere necessaria una normalizzazione adeguata per evitare sovrastime o sottostime nell’analisi. Tradizionalmente, i metodi per la quantificazione dell’abbondanza dei trascritti si basano sull’allineamento delle letture a un genoma o a una collezione di trascritti di riferimento, utilizzando strumenti come *BWA* [10], o *Bowtie2* [8]. Sebbene questi approcci garantiscono un’accurata mappatura delle sequenze, risultano computazionalmente onerosi, richiedendo ingenti risorse di memoria e tempo di elaborazione. In alternativa, metodi più recenti basati su pseudoallineamento offrono una soluzione più efficiente. Strumenti come *Kallisto* [2] utilizzano un approccio basato su k-mer, evitando l’allineamento esplicito delle letture e assegnandole direttamente ai trascritti di riferimento in base alla compatibilità probabilistica. Questo consente una quantificazione rapida e accurata dell’abbondanza dei trascritti, riducendo drasticamente il tempo di analisi e il consumo di risorse computazionali. L’efficacia di *Kallisto* nel risolvere il problema della quantificazione dell’espressione genica lo rende una scelta ideale per l’analisi di grandi dataset RNA-seq, in cui velocità e scalabilità sono fattori critici. La sua accuratezza nel distinguere trascritti sovrapposti lo pone come un’alternativa valida ai metodi tradizionali basati su allineamento.

1.2.2 Assemblaggio del genoma

L’assemblaggio del genoma è un processo finalizzato alla ricostruzione di un’intera sequenza genomica a partire da frammenti più piccoli prodotti dalle moderne tecnologie di sequenziamento, come *PacBio* e *Oxford Nanopore*, che generano letture lunghe con un alto tasso di errori. Poiché queste tecnologie suddividono il DNA in sequenze brevi, chiamate "reads", è necessario un processo di riassemblaggio per ricostruire la sequenza completa.

L’assemblaggio genomico si basa su due paradigmi principali: il *grafo di De Bruijn (DBG)* e il metodo *Overlap-Layout-Consensus (OLC)* [11]. Gli algoritmi basati su DBG frammentano le sequenze in k-mer e costruiscono un grafo in cui i nodi rappresentano i k-mer e gli archi rappresentano la loro sovrapposizione. Questo approccio è altamente scalabile ed efficiente per il sequenziamento di letture corte, poiché riduce il consumo di memoria e velocizza l’elaborazione dei dati. Tuttavia, può risultare meno preciso nella gestione di regioni altamente ripetitive, in quanto il grafo di De Bruijn può creare percorsi ambigui tra sequenze simili, e nella gestione di errori di sequenziamento, poiché questi possono frammentare il grafo e introdurre artefatti nell’assemblaggio. D’altra parte, il paradigma OLC è più adatto per il trattamento di letture lunghe. Questo metodo identifica sovrapposizioni tra le letture per costruire un layout dell’assemblaggio, seguito da una fase di raffinamento per ottenere la sequenza finale. Sebbene gli algoritmi OLC siano più accurati nella gestione delle ripetizioni e della struttura genomica complessa, risultano più dispendiosi in termini di tempo e memoria perché richiedono il confronto diretto di tutte le letture per identificare sovrapposizioni, un processo che cresce esponenzialmente con la dimensione del dataset. Entrambi i paradigmi sono fondamentali nell’assemblaggio genomico e la scelta del metodo dipende dal tipo di dati disponibili e dall’obiettivo specifico dell’analisi. L’assemblaggio genomico riveste un ruolo cruciale in ambiti quali la medicina genetica, l’oncologia e la biologia evolutiva, contribuendo alla comprensione della diversità genetica e allo sviluppo di nuove terapie.

Capitolo 2

Revisione della Letteratura

2.1 Metodi Alignment-based

Gli approcci di allineamento tradizionali utilizzano algoritmi avanzati per mappare le sequenze rispetto a un riferimento noto. Tra gli strumenti più noti vi sono *BWA* (*Burrows-Wheeler Aligner*) [10] e *Bowtie2* [8], che sfruttano la **Burrows-Wheeler Transform (BWT)** [3] per comprimere l'indice della sequenza di riferimento e ottimizzare le operazioni di ricerca. Questo approccio permette di eseguire allineamenti efficienti anche con la gestione di inserzioni e delezioni (*indel*), garantendo un'elevata accuratezza rispetto ai metodi basati su hashing.

Nonostante l'elevata precisione, i metodi di allineamento presentano limiti significativi. In primo luogo, assumono che le sequenze omologhe abbiano regioni conservate disposte linearmente, una condizione che spesso non si verifica nei genomi virali, caratterizzati da una grande variabilità e frequenti eventi di riarrangiamento genetico. Inoltre, quando il livello di similarità tra le sequenze scende al di sotto di una soglia critica, tipicamente compresa tra il 20% e il 35%, la distinzione tra omologia e casualità diventa incerta, compromettendo l'affidabilità dell'allineamento.

Un ulteriore ostacolo è il costo computazionale, poiché l'allineamento di sequenze lunghe e complesse richiede un significativo impiego di risorse, sia in termini di tempo che di memoria. In particolare, il problema dell'allineamento multiplo di sequenze è considerato computazionalmente difficile, il che implica la necessità di adottare strategie euristiche che, pur accelerando il processo, possono ridurre l'accuratezza dei risultati. Infine, gli allineamenti dipendono da parametri impostati a priori, come le matrici di sostituzione e le penalità per gap, che possono introdurre bias e influenzare l'interpretazione dei dati.

2.2 Metodi Alignment-free

I metodi alignment-free offrono un’alternativa più veloce e flessibile, consentendo il confronto tra sequenze genomiche senza la necessità di un riferimento esplicito. Questi approcci sfruttano diverse strategie per determinare la similarità tra sequenze, evitando il calcolo esplicito dell’allineamento e risultando più robusti contro riorganizzazioni genomiche e variazioni nelle sequenze.

2.2.1 Metodi basati sulla frequenza di parole (word-based)

Questi metodi si basano sull’idea che sequenze simili contengano frammenti comuni di nucleotidi o amminoacidi. Il processo si articola in tre fasi principali:

1. Segmentazione della sequenza in parole di lunghezza k .
2. Conversione in vettori numerici, conteggiando la frequenza di ciascuna parola.
3. Misurazione della similarità, spesso tramite distanza euclidea tra i vettori:

$$d_E(X, Y) = \sqrt{\sum_{i=1}^K (c_{X,i} - c_{Y,i})^2} \quad (2.1)$$

dove $c_{X,i}$ e $c_{Y,i}$ rappresentano le frequenze del k -mer i -esimo rispettivamente nelle sequenze X e Y . Questa formula calcola la distanza tra i due vettori di frequenze, fornendo una misura della loro somiglianza.

Esempi di questi metodi includono:

- **K-mers counting:** fondamentale per diversi strumenti di sequenziamento, consente di individuare sequenze comuni e costruire indici di riferimento per il mapping e l’assemblaggio del genoma.
- **Google distance:** misura la somiglianza tra gruppi di parole analizzando quante volte compaiono congiuntamente in un ampio dataset. Questo metodo è utile per classificare e raggruppare le sequenze genomiche in base alla loro affinità.
- **Chaos game representation:** un metodo grafico per visualizzare strutture genomiche, che rappresenta le sequenze come traiettorie in uno spazio bidimensionale, evidenziando pattern ricorrenti e differenze nella composizione delle sequenze.

Strumenti come Kallisto, Minimap e Miniasm sfruttano queste strategie per il mapping e l’assemblaggio genomico.

Query sequences	x	ATGTGTG	y	CATGTG
Word size: 3	W_3^x	ATG TGT GTG TGT GTG	W_3^y	CAT ATG TGT GTG
Union of two sets	$W_3 = W_3^x \cup W_3^y$	CAT ATG TGT GTG		
Word counts	c_3^x	0 1 2 2	c_3^y	1 1 1 1
Euclidean distance		$\ c_3^x - c_3^y\ = \sqrt{(0-1)^2 + (1-1)^2 + (2-1)^2 + (2-1)^2} = \sqrt{3} = 1.73$		

Figura 2.1: Esempio di confronto tra sequenze genomiche basato su k-mers e distanza euclidea [15].

2.2.2 Metodi basati sulla teoria dell'informazione

Questi approcci utilizzano misure di complessità ed entropia per valutare la similarità tra sequenze genomiche:

- **Kolmogorov Complexity:** misura la lunghezza della rappresentazione più breve di una sequenza, valutando la minima quantità di informazioni necessarie per descriverla senza perdita di dati. Questo concetto è strettamente legato alla compressibilità delle sequenze e alla loro complessità strutturale.
- **Compression-based distances:** si basano su algoritmi di compressione (es. *Lempel-Ziv 2.2*) per stimare la complessità della sequenza. Questi metodi sfruttano la ridondanza nei dati per ottenere una misura della loro somiglianza e sono utili per confrontare sequenze genomiche senza necessità di un allineamento esplicito.
- **Shannon Entropy:** quantifica l'incertezza in una sequenza in base alla distribuzione dei suoi caratteri e rappresenta una misura fondamentale per la valutazione della diversità e della casualità nelle sequenze genomiche:

Query sequences

x ATGTGTG

y CATGTG

xy ATGTGTGCATGTG

Lempel-Ziv complexity

		
$c(x)=4$	$c(y)=5$	$c(xy)=7$

Normalized compression distance

$$\frac{C(xy)-\min\{C(x), C(y)\}}{\max\{C(x), C(y)\}} \quad \boxed{\frac{7-4}{5} = 0.6}$$

Figura 2.2: Calcolo alignment-free della distanza di compressione normalizzata utilizzando l'algoritmo di stima della complessità di Lempel-Ziv. La complessità di Lempel-Ziv conta il numero di parole distinte in una sequenza quando viene letta da sinistra a destra (ad esempio, per $s = \text{ATGTGTG}$, la complessità di Lempel-Ziv è 4: A|T|G|TG) [15].

$$H(X) = - \sum_{i=1}^K p_i \log_2 p_i \quad (2.2)$$

dove p_i rappresenta la probabilità di occorrenza del simbolo i nella sequenza X . Questa misura descrive la quantità di informazione contenuta nella sequenza.

- **Kullback-Leibler divergence:** confronta le distribuzioni di frequenza di due sequenze per determinarne la divergenza informativa:

$$D_{KL}(P||Q) = \sum_{i=1}^K p_i \log_2 \frac{p_i}{q_i} \quad (2.3)$$

dove p_i e q_i sono le probabilità di osservare il carattere i nelle distribuzioni P e Q . Questa formula misura quanto una distribuzione differisce da un'altra.

Recenti studi hanno dimostrato che l'uso di misure derivate dalla teoria dell'informazione può offrire vantaggi significativi rispetto ai metodi basati su k -mers, specialmente per la classificazione tassonomica e la filogenesi.

Inoltre, metodi basati sulla distanza angolare tra vettori di frequenze stanno emergendo come alternative promettenti, capaci di ridurre gli effetti della lunghezza della sequenza sulla misura della similarità:

$$d_{cos}(X, Y) = 1 - \frac{\sum_{i=1}^K c_{X,i} c_{Y,i}}{\sqrt{\sum_{i=1}^K c_{X,i}^2} \sqrt{\sum_{i=1}^K c_{Y,i}^2}} \quad (2.4)$$

dove il numeratore rappresenta il prodotto scalare tra i due vettori di frequenze, mentre il denominatore normalizza il valore sulla lunghezza dei vettori. Questa misura descrive la similarità tra due sequenze basandosi sulla loro orientazione nello spazio multidimensionale.

I metodi alignment-free non sostituiscono completamente i metodi alignment-based, ma offrono strumenti complementari per filogenesi, metagenomica e analisi NGS. La sfida attuale è creare benchmark affidabili e valutare l'efficacia dei vari metodi in scenari biologici reali. Attualmente, molte metriche di valutazione sono influenzate dalla lunghezza della sequenza e dalla scelta dei parametri computazionali, rendendo difficile un confronto equo tra metodi diversi. Inoltre, la mancanza di dataset di riferimento ben curati e standardizzati complica ulteriormente la validazione delle prestazioni degli strumenti alignment-free. Sebbene i metodi basati su k -mers siano dominanti, quelli basati sulla teoria dell'informazione e sulle metriche angolari potrebbero migliorare con nuove tecniche, contribuendo ulteriormente all'analisi genomica moderna. L'integrazione di metodologie diverse potrebbe portare a strumenti ibridi che combinano i vantaggi di entrambi gli approcci, migliorando l'accuratezza e l'efficienza dell'analisi genomica.

Capitolo 3

Metodologia

Questo capitolo esaminerà tre strumenti bioinformatici avanzati: Kallisto, Minimap e Miniasm, sviluppati per ottimizzare diverse fasi del trattamento delle sequenze genomiche. Grazie a tecniche innovative, questi strumenti migliorano sia l'efficienza computazionale che la velocità di analisi.

Kallisto [2], progettato per la quantificazione dell'espressione genica da dati RNA-seq, utilizza un approccio di *pseudoallineamento* che consente di ottenere risultati accurati riducendo significativamente il tempo di calcolo e il consumo di risorse rispetto ai metodi di allineamento tradizionali. Minimap [9], invece, si distingue per la sua capacità di eseguire il mapping di long reads, sfruttando *minimizer* per individuare rapidamente regioni omologhe nel genoma e ottimizzando così l'efficienza del processo di allineamento, specialmente per dataset complessi. Infine, Miniasm [9] è un assembler de novo estremamente veloce che accelera il processo di assemblaggio genomico eliminando la fase di correzione degli errori, sebbene questo approccio comporti un compromesso sulla qualità delle sequenze ottenute.

L'obiettivo principale di questo capitolo è fornire un'analisi dettagliata di questi tre strumenti. Verranno approfonditi il loro principio di funzionamento e le tecniche computazionali adottate per migliorarne le prestazioni, oltre a un confronto con metodi alternativi per evidenziarne vantaggi e limitazioni. Inoltre, sarà proposta un'analisi del codice sorgente per comprendere le ottimizzazioni implementate e il loro impatto sulle prestazioni.

3.1 Kallisto

Descrizione Generale

Kallisto [2] è uno strumento alignment-free progettato per la quantificazione dell'espressione genica a partire da dati RNA-seq [12]. Il suo obiettivo principale è fornire una soluzione efficiente e veloce per stimare l'abbondanza dei trascritti, evitando la necessità di un allineamento esplicito delle letture.

A differenza dei metodi di quantificazione alignment-based, come *STAR* [4] o *HISAT2* [6], che richiedono una mappatura completa delle letture al genoma di riferimento, Kallisto sfrutta un approccio di pseudoallineamento, che permette di individuare rapidamente i trascritti compatibili senza la necessità di calcolare allineamenti base per base. Ciò consente un significativo risparmio di tempo e di risorse computazionali, rendendolo particolarmente adatto per l'analisi di grandi dataset di RNA-seq.

Principi di funzionamento

Lo pseudoallineamento, che costituisce il principio fondamentale di Kallisto, si distingue dagli approcci tradizionali perché esso si concentra esclusivamente sull'identificazione dei trascritti da cui una lettura potrebbe derivare, senza specificare le coordinate esatte delle basi all'interno delle sequenze di riferimento. Questo metodo permette una significativa riduzione del costo computazionale nell'analisi RNA-seq, migliorando l'efficienza senza compromettere in modo rilevante l'accuratezza della quantificazione.

Formalmente, dato un insieme di trascritti T , lo pseudoallineamento di una lettura r a T è definito come un sottoinsieme $S \subseteq T$, che raccoglie tutti i trascritti che possono essere plausibili fonti di origine per la lettura. In altre parole, S include tutti i trascritti $t \in T$ per i quali la frazione di k -mer condivisi con r supera una soglia prefissata τ . Questo approccio consente di identificare rapidamente i trascritti compatibili con una lettura senza la necessità di una mappatura esplicita delle basi. Inoltre, non tutti i k -mer di una lettura devono necessariamente essere trovati in un trascritto affinché venga considerato compatibile: la selezione dei trascritti si basa su una soglia prefissata, che determina la frazione minima di k -mer condivisi necessaria per l'inclusione nel sottoinsieme dei trascritti plausibili.

Per realizzare questa identificazione in modo efficiente, Kallisto costruisce un *indice del trascrittoma* basato su un **Transcriptome De Bruijn Graph (T-DBG)**. In questa struttura, i trascritti sono rappresentati come percorsi in un grafo in cui i nodi corrispondono ai k -mer e gli archi rappresentano la loro connessione sequenziale. Una volta costruito il grafo e i contig, Kallisto memorizza una tabella hash che associa ogni k -mer al contig in cui è contenuto, insieme alla posizione

all'interno del contig. Questa struttura consente di comprimere efficacemente le informazioni del trascrittoma, riducendo l'ingombro di memoria e velocizzando la ricerca dei k -mer durante lo pseudoallineamento.

Quando una lettura viene analizzata, i suoi k -mer vengono estratti e confrontati con quelli presenti nell'indice del T-DBG. Kallisto esegue questa operazione utilizzando una strategia di hashing dei k -mer, che consente di individuare rapidamente la *classe di compatibilità k -mer* associata a ciascun k -mer della lettura.

Poiché tutti i k -mer in un contig condividono la stessa classe di compatibilità, invece di confrontare ogni singolo k -mer con l'indice, Kallisto calcola la distanza dalle giunzioni ai confini del contig. Se una lettura proviene effettivamente da un trascritto presente nel T-DBG, i k -mer fino a tali distanze possono essere ignorati senza influenzare il risultato finale dell'intersezione delle classi di compatibilità. Questo meccanismo consente di ridurre significativamente il numero di lookup di hash, migliorando l'efficienza dello pseudoallineamento. Infine, Kallisto utilizza un algoritmo chiamato **Expectation-Maximization (EM)** per stimare quante volte ciascun trascritto è rappresentato nei dati RNA-seq. Impiega un metodo di campionamento ripetuto (*bootstrapping*) per valutare l'incertezza nelle stime di abbondanza dei trascritti. Questo processo genera multiple ripetizioni del dataset originale, simulando la variabilità naturale dei dati per stimare la distribuzione delle abbondanze. Kallisto fornisce intervalli di confidenza più robusti sulle quantificazioni, migliorando la stima dell'espressione genica. L'algoritmo lavora in modo iterativo, aggiornando progressivamente le probabilità di assegnazione delle letture ai trascritti, migliorando così la precisione della stima dell'abbondanza dei trascritti. Grazie alla sua efficienza, Kallisto permette di analizzare grandi dataset RNA-seq in tempi ridotti rispetto ai metodi basati su allineamento, rendendolo uno strumento altamente scalabile ed efficace per l'analisi dell'espressione genica.

Analisi del codice

Costruzione del Transcriptome de Bruijn Graph

La struttura `KmerEntry` è una delle componenti fondamentali di Kallisto, utilizzata per rappresentare i k -mer all'interno del **Transcriptome de Bruijn Graph (TDBG)**. Essa memorizza informazioni relative al contig di appartenenza, alla posizione del k -mer e alla lunghezza del contig. Questo consente di determinare la relazione tra un dato k -mer e i suoi equivalenti all'interno del grafo.

```
struct KmerEntry {
    int32_t contig; // id del contig
    uint32_t _pos; // distanza dalla giunzione dell'Equivalence
    Class (EC)
    int32_t contig_length;
};
```

Listing 3.1: Struttura KmerEntry

```
inline int getDist(bool fw) const {
    // Se la direzione del k-mer è la stessa di quella richiesta (fw)
    if (isFw() == fw) {
        // Calcola la distanza dalla fine del contig
        return (contig_length - 1 - getPos());
    } else {
        // Se la direzione è opposta, restituisce semplicemente la posizione
        return getPos();
    }
}
```

Listing 3.2: La funzione getDist

La funzione `getDist` calcola la distanza di un k -mer dal bordo del contig in cui si trova, tenendo conto della sua direzione. Se il k -mer è nella stessa direzione specificata dal parametro `fw`, la funzione restituisce la distanza dalla fine del contig, calcolata sottraendo la posizione del k -mer dalla lunghezza totale del contig. Se invece il k -mer è in direzione opposta, la funzione restituisce semplicemente la sua posizione, che rappresenta la distanza dall'inizio del contig. In pratica, questa funzione aiuta a capire quanto un k -mer si trova vicino ai confini del contig, sia in avanti che all'indietro. Questo è utile in Kallisto per determinare la posizione dei k -mer nel Transcriptome de Bruijn Graph e per migliorare l'efficienza dello pseudoallineamento.

```

void KmerIndex::BuildDeBruijnGraph(const ProgramOptions& opt,
                                    const std::string& tmp_file,
                                    std::ofstream& out) {
    CDBG_Build_opt c_opt;
    c_opt.k = k;
    c_opt.nb_threads = opt.threads;
    c_opt.build = true;
    c_opt.clipTips = false;
    c_opt.deleteIsolated = false;
    c_opt.verbose = true;
    c_opt.filename_ref_in.push_back(tmp_file);
    dbg = CompactedDBG<Node>(k, c_opt.g);
    dbg.build(c_opt);
}

```

Listing 3.3: La funzione `BuildDeBruijnGraph`

La funzione `BuildDeBruijnGraph` prende in input un file contenente le sequenze di riferimento (`tmp_file`) e un oggetto di configurazione (`opt`) che specifica i parametri di costruzione. Il primo passo consiste nell'inizializzare una struttura di configurazione (`CDBG_Build_opt c_opt`) e successivamente, viene creato un oggetto `CompactedDBG<Node>` (`dbg`), che rappresenta il grafo de Bruijn compatto con parametri definiti nella configurazione. Infine, il grafo viene costruito invocando il metodo `dbg.build(c_opt)`.

Expectation-Maximization

La classe `EMAlgorithm` implementa l'algoritmo EM e gestisce i dati relativi alla quantificazione. L'algoritmo inizia con un'assegnazione uniforme delle probabilità ai trascritti:

```

alpha_(num_trans_, 1.0/num_trans_) // Distribuzione uniforme sui
target

```

Listing 3.4: Inizializzazione delle abbondanze

L'algoritmo si sviluppa in due fasi iterative:

E-Step Si calcola la probabilità normalizzata che una lettura derivi da ciascun trascritto associato al cluster:

```
denom = 0.0;
for (auto t_it = 0; t_it < numEC; ++t_it) {
    denom += alpha_[trs[t_it]] * wv[t_it];
}
if (denom < TOLERANCE) {
    continue;
}
```

Listing 3.5: Calcolo della normalizzazione dell'abbondanza

Se il denominatore è troppo piccolo, il cluster viene ignorato per evitare instabilità numeriche.

M-Step Si aggiorna l'abbondanza stimata dei trascritti:

```
auto countNorm = counts_[it.second] / denom;
for (auto t_it = 0; t_it < numEC; ++t_it) {
    next_alpha[trs[t_it]] += (wv[t_it] * alpha_[trs[t_it]]) *
        countNorm;
}
```

Listing 3.6: Aggiornamento dell'abbondanza

Questa fase tiene conto della probabilità di appartenenza di una lettura a un trascritto, pesata rispetto alla distribuzione attuale delle abbondanze. L'algoritmo si arresta quando il cambiamento relativo tra due iterazioni consecutive è trascurabile:

```
if (chcount == 0 && i > min_rounds) {
    stopEM = true;
}
```

Listing 3.7: Condizione di arresto dell'EM

Al termine della convergenza, le abbondanze molto piccole vengono azzerate:

```
for (int ec = 0; ec < num_trans_; ec++) {
    if (alpha_[ec] < alpha_limit/10.0) {
        alpha_[ec] = 0.0;
    }
}
```

Listing 3.8: Filtraggio delle abbondanze trascurabili

Dopo la convergenza, le abbondanze normalizzate vengono convertite in unità TPM (Transcripts Per Million) e scritte in output:

```
out << "target_id" << "\t" << "rho" << "\t" << "tpm" << "\t" << "
est_counts" << std::endl;
for (auto i = 0; i < rho_.size(); ++i) {
    out << target_names_[i] << "\t" << rho_[i] << "\t" << rho_[i]
        * MILLION << "\t" << alpha_[i] << std::endl;
}
```

Listing 3.9: Output dell'algoritmo EM

3.2 Minimap

Descrizione Generale

Minimap [9] è un software progettato per il mapping di long reads, in particolare per dati prodotti da tecnologie di sequenziamento di terza generazione come *Pac-Bio* e *Oxford Nanopore*. Grazie a un algoritmo basato sui *minimizer*, Minimap è in grado di individuare rapidamente le regioni omologhe tra le letture e il genoma di riferimento, migliorando significativamente le prestazioni rispetto agli approcci tradizionali. Il suo utilizzo si estende anche al confronto di interi genomi e alla generazione di allineamenti veloci e approssimati.

Minimap si concentra sull'identificazione rapida delle corrispondenze tra sequenze, rendendolo particolarmente utile per applicazioni che richiedono elevata velocità di esecuzione e una gestione efficiente di grandi volumi di dati.

Principi di Funzionamento

I minimizer sono un sottoinsieme selezionato dei k -mers di una sequenza, utilizzati per rappresentarla in modo compatto ed efficiente. In una finestra scorrevole di lunghezza w , che contiene $w - k + 1$ k -mers, il **minimizer** è il k -mer con il valore hash più basso nella finestra. Man mano che la finestra avanza, il minimizer viene aggiornato solo se il nuovo k -mer ha un valore hash inferiore o se il minimizer attuale esce dalla finestra. Questo meccanismo consente di ottenere un sottoinsieme ridotto di k -mers che preserva le informazioni più rappresentative della sequenza originale.

L'uso dei minimizer consente di ridurre il numero di operazioni di confronto necessarie per identificare regioni omologhe tra le sequenze, senza compromettere significativamente la sensibilità.

Il software adotta un approccio greedy per scegliere i minimizer, assicurando una copertura uniforme della sequenza con un basso costo computazionale. Questo equilibrio tra velocità e accuratezza migliora il mapping delle letture lunghe senza ridurre la sensibilità nel rilevamento delle omologie. Rispetto ai grafi di De Bruijn, che frammentano le sequenze in tutti i possibili k -mers, i minimizer permettono una riduzione del consumo di memoria e un'accelerazione della ricerca delle corrispondenze, riducendo drasticamente il numero di confronti necessari, mantenendo comunque un alto grado di sensibilità.

Il flusso di lavoro di Minimap si articola nei seguenti passaggi:

1. **Indicizzazione delle sequenze target:** Le sequenze di riferimento vengono convertite in una rappresentazione compatta basata sui minimizer riducendo così drasticamente il consumo di memoria.
2. **Estrazione dei minimizer dalle letture:** Ogni lettura viene processata per identificare i minimizer che verranno confrontati con quelli della sequenza di riferimento.
3. **Matching approssimato:** I minimizer estratti dalle letture vengono confrontati con quelli della sequenza target per identificare regioni di potenziale omologia.
4. **Filtraggio e clustering:** I risultati vengono filtrati e raggruppati per migliorare la precisione del mapping, riducendo gli errori e l'ambiguità.

Grazie a questa strategia, Minimap è estremamente veloce ed efficiente dal punto di vista computazionale, permettendo il mapping di milioni di letture in pochi minuti. Tuttavia, poiché non esegue un allineamento dettagliato, può essere meno accurato rispetto a metodi più raffinati nei casi in cui sia necessaria una precisa determinazione delle corrispondenze tra sequenze.

Analisi del codice

Costruzione dei minimizer

L'algoritmo per il calcolo dei minimizer (1) funziona come segue:

1. **Scorrimento della sequenza** Si analizza una sequenza di DNA suddividendola in finestre sovrapposte di lunghezza w , in modo da coprire progressivamente l'intera sequenza di input.
2. **Generazione dei k-mer** Per ogni finestra, si estraggono tutti i k-mer, ovvero tutte le sottosequenze di lunghezza k contenute in quella finestra.
3. **Calcolo delle rappresentazioni hash** Ogni k-mer viene trasformato in un valore numerico attraverso una funzione di hashing. Questo valore serve a rappresentare il k-mer in modo compatto ed efficiente. Inoltre, si calcola il valore hash del complemento Watson-Crick del k-mer (ossia la sua sequenza complementare inversa).
4. **Selezione del minimizer** Tra tutti i valori hash ottenuti nella finestra, si seleziona il più piccolo. Se vi sono più valori minimi uguali, la finestra viene ignorata per evitare ambiguità.
5. **Memorizzazione del minimizer** Dopo aver individuato il valore hash minimo, si verifica quale k-mer in quella finestra lo ha generato. Il k-mer corrispondente viene registrato come minimizer, memorizzando anche la sua posizione nella sequenza e il relativo strand (orientamento).
6. **Proseguimento e ottimizzazione** L'algoritmo avanza lungo la sequenza, aggiornando la finestra scorrevole e ripetendo il processo. Per migliorare l'efficienza, si utilizza una struttura dati (come una coda) che evita il ricalcolo completo dei minimizer in ogni nuova finestra.
7. **Output** Alla fine del processo, si ottiene un insieme di minimizer che rappresentano in modo compatto la sequenza originale. Questo insieme viene restituito come output e può essere utilizzato per l'allineamento o la compressione delle sequenze genomiche.

Algorithm 1 Calcolo dei minimizer

```
1: Input: Parametri  $w$  e  $k$  e sequenza  $s$  con  $|s| \geq w + k - 1$ 
2: Output:  $(w, k)$ -minimizer, le loro posizioni e i relativi strand
3: function MINIMIZERSKETCH( $w, k, s$ )
4:    $\mathcal{M} \leftarrow \emptyset$                                  $\triangleright \mathcal{M}$  è un insieme; niente duplicati
5:   for  $i \leftarrow 1$  to  $|s| - w - k + 1$  do
6:      $m \leftarrow \infty$ 
7:     for  $j \leftarrow 0$  to  $w - 1$  do                       $\triangleright$  Trova il valore minimo
8:        $(u, v) \leftarrow (\phi(s_{i+j}^+), \phi(s_{i+j}^-))$ 
9:       if  $u = v$  then                                 $\triangleright$  Salta se lo strand è ambiguo
10:        continue
11:      end if
12:       $m \leftarrow \min(m, \min(u, v))$ 
13:    end for
14:    for  $j \leftarrow 0$  to  $w - 1$  do                   $\triangleright$  Raccogli i minimizer
15:       $(u, v) \leftarrow (\phi(s_{i+j}^+), \phi(s_{i+j}^-))$ 
16:      if  $m = u$  e  $u \neq v$  then
17:         $\mathcal{M} \leftarrow \mathcal{M} \cup \{(m, i + j)\}$ 
18:      else if  $m = v$  e  $u \neq v$  then
19:         $\mathcal{M} \leftarrow \mathcal{M} \cup \{(m, i + j, 1)\}$ 
20:      end if
21:    end for
22:  end for
23:  return  $\mathcal{M}$ 
24: end function
```

Indicizzazione delle sequenze target

L'algoritmo di indicizzazione delle sequenze target (2) ha l'obiettivo di costruire una tabella hash contenente i minimizer di un insieme di sequenze target. Questa struttura dati sarà utile per cercare rapidamente corrispondenze tra le sequenze, facilitando operazioni come l'allineamento o l'assemblaggio.

Algorithm 2 Indicizzazione delle sequenze target

```

1: Input: Insieme di sequenze target  $\mathcal{T} = \{s_1, \dots, s_T\}$ 
2: Output: Tabella hash dei minimizer  $\mathcal{H}$ 
3: function INDEX( $\mathcal{T}, w, k$ )
4:    $\mathcal{H} \leftarrow$  tabella hash vuota
5:   for  $t \leftarrow 1$  to  $T$  do                                 $\triangleright$  Itera su tutte le sequenze target
6:      $\mathcal{M} \leftarrow$  MinimizerSketch( $s_t, w, k$ )
7:     for all  $(h, i, r) \in \mathcal{M}$  do                 $\triangleright$  Scansiona i minimizer trovati
8:        $\mathcal{H}[h] \leftarrow \mathcal{H}[h] \cup \{(t, i, r)\}$ 
9:     end for
10:   end for
11:   return  $\mathcal{H}$ 
12: end function

```

Nel dettaglio, l'algoritmo per l'indicizzazione delle sequenze target funziona nel seguente modo:

1. **Inizializzazione della tabella hash** L'algoritmo parte creando una tabella hash vuota H , che verrà popolata progressivamente con i minimizer estratti dalle sequenze.
2. **Iterazione sulle sequenze target** Si prende in esame ciascuna sequenza dell'insieme target T , che può contenere numerose sequenze genomiche. L'algoritmo esegue un'operazione di scansione su ogni sequenza.
3. **Calcolo dei minimizer** Per ciascuna sequenza, si esegue la procedura di estrazione dei minimizer. Questo avviene attraverso la funzione `MinimizerSketch`, che scorre la sequenza utilizzando una finestra di lunghezza w e individua i minimizer tra i k -mer contenuti in essa. Ogni minimizer è rappresentato come una tupla contenente il valore hash h del minimizer, la posizione i nella sequenza e l'orientamento r (strand forward o reverse).

4. **Inserimento nella tabella hash** Una volta individuati i minimizer, essi vengono inseriti nella tabella hash. Per ogni minimizer, si controlla la chiave corrispondente (l'hash h) nella tabella H . Se già presente, si aggiunge la nuova informazione alla lista associata a quella chiave. Se non esiste ancora, si crea una nuova voce nella tabella.
5. **Restituzione della tabella hash** Dopo aver elaborato tutte le sequenze target, l'algoritmo restituisce H , che ora contiene un'indicizzazione efficiente dei minimizer. Questa struttura consente di cercare rapidamente corrispondenze tra le sequenze, facilitando operazioni di allineamento e confronto.

L'uso della tabella hash dei minimizer permette di ridurre la complessità computazionale rispetto a un confronto diretto di tutte le sottosequenze e di accedere rapidamente alle corrispondenze tra minimizer nelle varie sequenze, un'operazione fondamentale per algoritmi di allineamento e assemblaggio. Inoltre permette di migliorare la gestione dei dati genomici di grandi dimensioni rendendo più efficienti le operazioni su sequenze lunghe.

Mapping

Il seguente algoritmo mappa una sequenza di query rispetto a un insieme di sequenze target già indicizzate. In altre parole, cerca regioni di corrispondenza tra la query e le sequenze target utilizzando i minimizer. L'algoritmo fa parte della fase di **Matching approssimato** e **Filtraggio e clustering**, nel flusso di lavoro di Minimap.

Minimap cerca corrispondenze tra i minimizer della query e quelli del target utilizzando una tabella hash H . Se un minimizer m è presente in H , vengono recuperati i relativi hit, ossia le posizioni in cui lo stesso minimizer appare nel target. Ogni hit h rappresenta una potenziale sovrapposizione tra query e target e viene memorizzato in una lista L con la relativa posizione e orientamento.

Gli hit vengono poi ordinati in base alla posizione nel target e raggruppati in cluster per individuare regioni contigue di somiglianza, filtrando eventuali corrispondenze spurious. Viene quindi determinato il **sottoinsieme colineare massimo**, ovvero la più lunga sequenza di minimizer che segue un ordine crescente sia nella query che nel target, riducendo i falsi positivi e migliorando la qualità del mapping.

Una volta individuate le corrispondenze più robuste, vengono generati gli intervalli di mappatura, rappresentando le regioni di massima similarità tra query e target.

L'algoritmo 3 descrive in dettaglio questo processo: dopo il calcolo dei minimizer con **MinimizerSketch**, si cercano corrispondenze nella tabella hash e si registrano gli hit, distinguendo tra minimizer sullo stesso strand o su strand opposti. Gli hit vengono poi ordinati, raggruppati in cluster e filtrati per garantire un mapping coerente.

Algorithm 3 Mappatura di una sequenza di query

```

1: Input: Tabella hash  $\mathcal{H}$  e sequenza di query  $q$ 
2: Output: Stampa gli intervalli di query e target corrispondenti
3: function MAP( $\mathcal{H}, q, w, k, \epsilon$ )
4:    $\mathcal{A} \leftarrow$  array vuoto
5:    $\mathcal{M} \leftarrow$  MinimizerSketch( $q, w, k$ )
6:   for all  $(h, i, r) \in \mathcal{M}$  do                                 $\triangleright$  Raccoglie gli hit dei minimizer
7:     for all  $(t, i', r') \in \mathcal{H}[h]$  do
8:       if  $r = r'$  then                                 $\triangleright$  Minimizer sullo stesso strand
9:         Aggiungi  $(t, 0, i - i', i')$  a  $\mathcal{A}$ 
10:      else                                          $\triangleright$  Minimizer su strand opposti
11:        Aggiungi  $(t, 1, i + i', i')$  a  $\mathcal{A}$ 
12:      end if
13:    end for
14:  end for
15:  Ordina  $\mathcal{A} = [(t, r, c, i')]$  in base ai quattro valori nelle tuple
16:   $b \leftarrow 1$ 
17:  for  $e \leftarrow 1$  to  $|\mathcal{A}|$  do                                 $\triangleright$  Raggruppa gli hit dei minimizer
18:    if  $e = |\mathcal{A}|$  o  $\mathcal{A}[e + 1].t \neq \mathcal{A}[e].t$  o  $\mathcal{A}[e + 1].r \neq \mathcal{A}[e].r$  then
19:      o  $\mathcal{A}[e + 1].c - \mathcal{A}[e].c \geq \epsilon$ 
20:       $\mathcal{C} \leftarrow$  il sottoinsieme colineare massimo di  $\mathcal{A}[b..e]$ 
21:      Stampa le posizioni di query/target più a sinistra e a destra in  $\mathcal{C}$ 
22:       $b \leftarrow e + 1$ 
23:    end if
24:  end for
25: end function

```

3.3 Miniasm

Descrizione Generale

L’assemblaggio delle sequenze genomiche può essere affrontato con due principali paradigmi: il modello *Overlap-Layout-Consensus* (*OLC*) e il modello basato su *grafi di De Bruijn* (*DBG*) [11]. Gli algoritmi basati su OLC iniziano identificando le sovrapposizioni tra le letture, costruendo successivamente un grafo di layout e infine producendo un’eventuale sequenza di consenso. Questo metodo è particolarmente efficace per le letture lunghe, permettendo di gestire meglio le regioni ripetitive del genoma rispetto ai metodi basati su grafi di De Bruijn.

Un tipico assemblatore OLC segue tre fasi principali:

- **Overlap:** Vengono identificate le regioni di sovrapposizione tra le diverse letture.
- **Layout:** Le letture vengono organizzate in un grafo di sovrapposizione, eliminando le sequenze ridondanti e costruendo una rappresentazione strutturata del genoma. Il grafo ottenuto consente di stabilire una connettività chiara tra le letture per ottimizzare il processo di assemblaggio.
- **Consensus:** Si applicano algoritmi di consenso per generare una sequenza finale accurata, riducendo gli errori di sequenziamento attraverso strategie di correzione degli errori (*polishing*). Questa fase è fondamentale per ottenere una sequenza di alta qualità, specialmente quando si lavora con letture con un alto tasso di errore provenienti da tecnologie di terza generazione.

Alcuni assembler, come *Canu* [7], implementano tutte e tre queste fasi, mentre Miniasm omette la fase di consenso per massimizzare la velocità di elaborazione. Questo approccio consente di ottenere rapidamente un assemblaggio preliminare del genoma, riducendo drasticamente i tempi di calcolo rispetto agli assembler tradizionali. L’assenza della fase di consenso implica che la qualità della sequenza ottenuta dipenda fortemente dalla precisione delle letture di input, rendendo spesso necessaria una fase successiva di rifinitura delle sequenze per correggere eventuali errori.

Principi di Funzionamento

Miniasm è un assembler de novo ottimizzato per la velocità: il suo approccio consente di generare rapidamente una bozza dell’assemblaggio, ma senza una correzione degli errori integrata, la qualità delle sequenze risultanti è inferiore rispetto ad altri assembler OLC. Sebbene garantisca un’alta velocità di assemblaggio, Miniasm produce sequenze che spesso richiedono una rifinitura con strumenti come *Racon* [13] per migliorarne la qualità. Miniasm è stato sviluppato per funzionare in combinazione con Minimap, che viene utilizzato per individuare le sovrapposizioni tra le letture.

Il processo di assemblaggio si basa sulle seguenti fasi principali:

1. **Identificazione delle Sovrapposizioni tra le Letture:** L’assemblaggio inizia con la ricerca delle sovrapposizioni tra le letture di sequenziamento utilizzando Minimap. Questa strategia consente di individuare rapidamente le regioni di sovrapposizione tra le letture e di ridurre la complessità computazionale dell’allineamento all-vs-all. L’output di questa fase è un insieme di mappature tra coppie di letture che mostrano una sovrapposizione significativa.
2. **Costruzione del Grafo di Assemblaggio:** Dopo il filtraggio delle letture, Miniasm costruisce un grafo di sovrapposizione (*overlap graph*) che rappresenta le connessioni tra le sequenze. Ogni lettura è un nodo del grafo, mentre un arco tra due nodi viene creato se tra di essi esiste una sovrapposizione.
3. **Pulizia del Grafo di Assemblaggio:** Una volta costruito il grafo di sovrapposizione, vengono applicate diverse operazioni di pulizia per migliorarne la struttura e rimuovere le ambiguità. Innanzitutto, vengono eliminate le connessioni ridondanti che possono essere dedotte da altre relazioni nel grafo. Successivamente, vengono identificati ed eliminati i cosiddetti “*tips*”, ovvero sequenze terminali isolate composte da poche letture, che spesso derivano da errori o frammenti non informativi. Infine, vengono individuate e collassate le “*bubbles*”, ovvero percorsi alternativi nel grafo causati da varianti genomiche o errori di sequenziamento. Questa fase è fondamentale per migliorare la continuità dell’assemblaggio e ridurre il numero di contigs generati.
4. **Generazione delle Unitigs:** Dopo la pulizia del grafo, Miniasm estrae le *unitigs*, ovvero sequenze contigue ottenute direttamente dalla topologia del grafo. Ogni unitig rappresenta una concatenazione di letture che formano un percorso univoco nel grafo di sovrapposizione. A causa dell’assenza di una fase di consenso, le unitigs generate mantengono lo stesso tasso di errore delle letture originali. Sebbene questa scelta semplifichi il processo di assemblaggio

e riduca drasticamente i tempi di esecuzione, implica che l'output finale presenti ancora errori di sequenziamento che dovranno essere corretti in una fase successiva con strumenti specifici di consensus polishing.

5. **Esportazione del Grafo in Formato GFA:** Il risultato finale dell'assemblaggio è un file in formato GFA (Graphical Fragment Assembly), che descrive la struttura del grafo generato. Questo file contiene informazioni sui nodi del grafo, le connessioni tra le letture e le sequenze delle unitigs assemblate.

Miniasm è particolarmente utile per assemblaggi rapidi di dati provenienti da tecnologie di sequenziamento di terza generazione, come studi su microbiomi, analisi di varianti strutturali o genomi batterici e virali. Fornisce una struttura preliminare del genoma, che può essere successivamente migliorata con strumenti dedicati per aumentare la precisione delle sequenze.

Analisi del codice

Classificazione dei Mapping

Il seguente algoritmo si occupa di classificare la relazione tra due letture genomiche basandosi sulle loro coordinate di inizio e fine. Identifica se una lettura è contenuta nell'altra, se si sovrappongono parzialmente o se sono corrispondenze interne, facilitando il processo di allineamento e assemblaggio delle sequenze.

L'algoritmo di mapping classification (4) è utilizzato per classificare la relazione tra due letture genomiche basandosi sulle loro coordinate di inizio e fine, permettendo di determinare il tipo di sovrapposizione tra le sequenze. Il processo inizia calcolando l'overhang, che rappresenta la somma delle regioni non allineate all'inizio e alla fine delle due letture. Questo valore è determinato considerando le posizioni iniziali e finali delle due letture, sommando le porzioni non allineate ai margini. Contemporaneamente, si calcola la lunghezza massima del mapping, ossia la massima distanza coperta tra le due sequenze.

Se l'overhang supera una soglia prestabilita, la lettura viene classificata come **INTERNAL_MATCH**. In caso contrario, viene effettuata una verifica per determinare se una lettura è completamente contenuta nell'altra. Se la prima lettura include interamente la seconda, la classificazione risultante sarà **FIRST_CONTAINED**; viceversa, se la seconda lettura ingloba completamente la prima, viene assegnata la categoria **SECOND_CONTAINED**. Se nessuno di questi criteri è soddisfatto, si analizzano le sovrapposizioni parziali tra le due letture. Se la prima lettura inizia dopo la seconda, il mapping viene categorizzato come **FIRST_TO_SECOND_OVERLAP**; altrimenti, viene etichettato come **SECOND_TO_FIRST_OVERLAP**.

Questa classificazione è essenziale per filtrare e organizzare le letture in modo che contribuiscano efficacemente alla costruzione del grafo di assemblaggio.

Algorithm 4 Mapping classification

```

1: Input: Lunghezza della lettura  $l$ , coordinate di inizio  $b$  e fine  $e$  della mappatura
   per due letture; valore massimo di overhang  $o$  (1000 di default) e rapporto
   massimo tra overhang e lunghezza della mappatura  $r$  (0.8 di default)
2: Output: Intero hashed  $p$ -bit
3: function CLASSIFYMAPPING( $(l[2], b[2], e[2], o, r)$ )
4:    $overhang \leftarrow \min(b[1], b[2]) + \min(l[1] - e[1], l[2] - e[2])$ 
5:    $maplen \leftarrow \max(e[1] - b[1], e[2] - b[2])$ 
6:   if  $overhang > \min(o, maplen \cdot r)$  then
7:     return INTERNAL_MATCH
8:   else if  $b[1] \leq b[2]$  e  $l[1] - e[1] \leq l[2] - e[2]$  then
9:     return FIRST_CONTAINED
10:   else if  $b[1] \geq b[2]$  e  $l[1] - e[1] \geq l[2] - e[2]$  then
11:     return SECOND_CONTAINED
12:   else if  $b[1] > b[2]$  then
13:     return FIRST_TO_SECOND_OVERLAP
14:   else
15:     return SECOND_TO_FIRST_OVERLAP
16:   end if
17: end function

```

Pulizia del Grafo

Uno dei problemi principali nei grafi di assemblaggio è la presenza di **bolle**. Esse possono derivare da errori di sequenziamento o da reali varianti genomiche. Miniasm rileva e rimuove queste strutture utilizzando l'algoritmo di bubble detection (5).

In particolare, l'algoritmo prende in input un grafo diretto G , un nodo di partenza v_0 e un parametro d che definisce la distanza massima di esplorazione. Il processo inizia verificando se v_0 ha almeno due archi uscenti; in caso contrario, non può essere la fonte di una bolla e l'algoritmo termina. Successivamente, vengono inizializzate le strutture dati, assegnando una distanza infinita a tutti i nodi tranne v_0 , che viene impostato a 0. Si utilizza uno stack S per memorizzare i nodi con archi entranti già esplorati e un contatore p per tracciare i nodi ancora da visitare. L'algoritmo procede estraendo iterativamente un nodo v dallo stack e analizzando i suoi archi uscenti. Se viene rilevato un ciclo che coinvolge v_0 , l'algoritmo termina restituendo *nil*. Se la distanza supera d , il cammino viene escluso. I nodi visitati per la prima volta vengono aggiornati, mentre quelli con tutti gli ingressi esplorati vengono aggiunti allo stack.

Infine, se nello stack rimane un solo elemento e $p = 0$, significa che è stato trovato il nodo di chiusura della bolla e viene restituito. Se nessun nodo di chiusura viene identificato, l'algoritmo restituisce *nil*. Questo processo permette di individuare e rimuovere strutture ridondanti nel grafo, migliorando l'accuratezza dell'assemblaggio e preservando solo i percorsi più affidabili.

Algorithm 5 Bubble detection

```
1: Input: Grafo  $G = (V, E)$ , vertice iniziale  $v_0$  e distanza massima di esplorazione  
    $d$   
2: Output: Il vertice sink di una bolla entro  $d$ , oppure nil se non trovato  
3: function DETECTBUBBLE( $V, E, v_0, d$ )  
4:   if  $\deg^+(v_0) < 2$  then            $\triangleright$  Non può essere l'origine di una bolla  
5:     return nil  
6:   end if  
7:   for all  $v \in V$  do            $\triangleright$  Inizializza le distanze minime  
8:      $\delta[v] \leftarrow \infty$   
9:   end for  
10:   $\delta[v_0] \leftarrow 0$   
11:   $S \leftarrow$  stack vuoto            $\triangleright$  Vertici con tutti gli archi entranti visitati  
12:  Push( $S, v_0$ )  
13:   $p \leftarrow 0$             $\triangleright$  Numero di vertici visitati mai aggiunti a  $S$   
14:  while  $S$  non è vuoto do  
15:     $v \leftarrow \text{Pop}(S)$   
16:    for all  $(v \rightarrow w) \in E$  do  
17:      if  $w = v_0$  then            $\triangleright$  Un ciclo che coinvolge il vertice iniziale  
18:        return nil  
19:      end if  
20:      if  $\delta[v] + \ell(v \rightarrow w) > d$  then            $\triangleright$  Esplorazione oltre il limite  
21:        return nil  
22:      end if  
23:      if  $\delta[w] = \infty$  then            $\triangleright$  Vertice mai visitato prima  
24:         $\gamma[w] \leftarrow \deg^-(w)$             $\triangleright$  Numero di archi entranti non visitati  
25:         $p \leftarrow p + 1$   
26:      end if  
27:      if  $\delta[v] + \ell(v \rightarrow w) < \delta[w]$  then  
28:         $\delta[w] \leftarrow \delta[v] + \ell(v \rightarrow w)$   
29:      end if  
30:       $\gamma[w] \leftarrow \gamma[w] - 1$             $\triangleright$  Aggiorna il conteggio degli archi entranti  
31:      if  $\gamma[w] = 0$  then            $\triangleright$  Tutti gli archi entranti visitati  
32:        if  $\deg^+(w) \neq 0$  then            $\triangleright$  Non è un tip  
33:          Push( $S, w$ )  
34:        end if  
35:         $p \leftarrow p - 1$   
36:      end if  
37:    end for  
38:    if  $|S| = 1$  e  $p = 0$  then            $\triangleright$  Nodo di chiusura della bolla trovato  
39:      return Pop( $S$ )  
40:    end if  
41:  end while                                29  
42:  return nil  
43: end function
```

Capitolo 4

Risultati

In questo capitolo vengono presentati i risultati sperimentali ottenuti dall’analisi delle sequenze genomiche utilizzando i tool Kallisto, Minimap e Miniasm. I risultati sono stati ottenuti mediante il framework **Galaxy** [1], utilizzando dataset pubblici di *Escherichia coli* e lasciando invariati i parametri di default per ognuno dei tool.

L’analisi è stata eseguita in un ambiente virtualizzato mediante container *Singularity*, garantendo la riproducibilità e l’isolamento del processo. Il sistema operativo su cui è stato eseguito è Linux (kernel 5.14.0-427.31.1.e19.x86_64), assicurando una gestione efficiente delle risorse e un’ottima compatibilità con gli strumenti di bioinformatica.

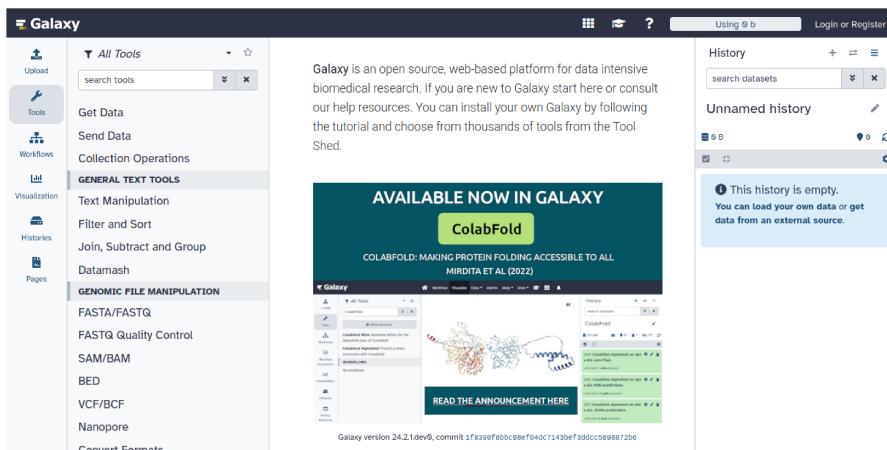


Figura 4.1: Interfaccia della piattaforma Galaxy.

4.1 Kallisto

Risultati ottenuti - Primo test

Dati utilizzati

- **Trascrittoma di riferimento:**

Escherichia_coli_str_k_12_substr_mg1655_gca_000005845.ASM584v2.cdna.all.fa

- **Lettura forward:** ERR2686027_1.fastq.gz

- **Lettura reverse:** ERR2686027_2.fastq.gz

Più nello specifico, oltre al trascrittoma di riferimento e le letture date in input, Kallisto è stato eseguito lasciando i seguenti parametri al valore di default (4.2):

Tool Parameters	
Input Parameter	Value
Reference transcriptome for quantification	history
FASTA reference transcriptome	3: Escherichia_coli_str_k_12_substr_mg1655_gca_000005845.ASM584v2.cdna.all.fa uncompresssed
Single-end or paired reads	paired_single
Forward reads	1: ERR2686027_1.fastq.gz
Reverse reads	2: ERR2686027_2.fastq.gz
Library strandness information	Unstranded
Perform sequence based bias correction	false
Number of bootstrap samples	0
Search for fusions	false
Single overhang	false
Output pseudoalignments in BAM format	true
Project pseudoalignments to genome	
Seed for the bootstrap sampling	42

Figura 4.2: Parametri di esecuzione di Kallisto per la quantificazione dell'espressione genica.

Descrizione dei parametri

Tra i parametri più significativi vi sono i seguenti:

- **Single-end or paired reads:** `paired_single`
Indica che il sequenziamento è paired-end, quindi ci sono due file (`_1.fastq.gz` e `_2.fastq.gz`).
- **Forward reads:** `ERR2686027_1.fastq.gz`
File contenente le letture forward.
- **Reverse reads:** `ERR2686027_2.fastq.gz`
File contenente le letture reverse.
- **Library strandness information:** `Unstranded`
Indica che la libreria di RNA-seq non ha una direzione specifica (**non stranded**), quindi non è noto da quale filamento del DNA derivi ciascun trascritto.
- **Perform sequence based bias correction:** `false`
Se impostato su `true`, Kallisto correggerebbe i bias derivanti dalla composizione nucleotidica locale nelle letture. In questo caso, la correzione non è stata applicata.
- **Number of bootstrap samples:** `0`
Il bootstrap è un metodo per stimare la variabilità dei dati ripetendo la quantificazione più volte con sottocampioni. Qui è stato impostato a 0, quindi non viene eseguito.
- **Single overhang:** `false`
Se `true`, permetterebbe la quantificazione anche di letture che si sovrappongono a un solo estremo di un trascritto. Qui è disabilitato.
- **Seed for the bootstrap sampling:** `42`
Numero utilizzato come seed per la generazione casuale dei bootstrap sample. Dato che il numero di bootstrap è 0, questo parametro non ha effetto in questa analisi.

Output ottenuto

L'output di Kallisto consiste in un file tabulare (4.1) contenente le seguenti informazioni:

- **ID del trascritto (target_id)**
- **Lunghezza totale del trascritto (length)** in nucleotidi, come riportata nel trascrittoma di riferimento
- **Lunghezza effettiva (eff_length)** calcolata da Kallisto per compensare il bias dovuto alla frammentazione delle letture, tenendo conto della probabilità che un frammento di RNA si sovrapponga a un trascritto durante il sequenziamento
- **Stima del numero di letture assegnate (est_counts)**
- **TPM** (Transcripts Per Million), una misura normalizzata dell'abbondanza relativa

target_id	length	eff_length	est_counts	tpm
AAD13456	3051	2570.4	3323	342.324
AAD13462	2148	1667.4	218	34.6198
AAD13438	3048	2567.4	8	8.14782
AAC76904	300	50.6492	0	0
AAC75356	1203	722.396	111	40.6869
AAC74435	489	68.4703	22	85.08
AAC76238	273	41.9355	2	0
AAC74021	1146	665.396	11	4.37743
AAC75128	3318	2837.4	131	12.2253
AAC74857	504	76.4	124	429.769
AAC74091	1329	848.396	44	13.7329
AAC75784	1140	659.396	214	85.9359
AAC75546	1062	581.396	396	180.356

Tabella 4.1: Output di Kallisto (alcuni record): informazioni sui trascritti, lunghezza effettiva, conteggi stimati e TPM.

Un'analisi preliminare dei risultati mostra una grande variabilità nei livelli di espressione dei trascritti. In particolare, alcuni trascritti presentano valori di TPM significativamente più elevati rispetto agli altri, indicando una forte espressione nei campioni analizzati.

Il seguente grafico è stato ricavato dal file tabulare ottenuto in output, ordinando tutti i record in ordine decrescente in base al TPM e prelevando i primi dieci risultati. In particolare nel grafico sono riportati i 10 trascritti con TPM più elevato, evidenziando le differenze nei livelli di espressione.

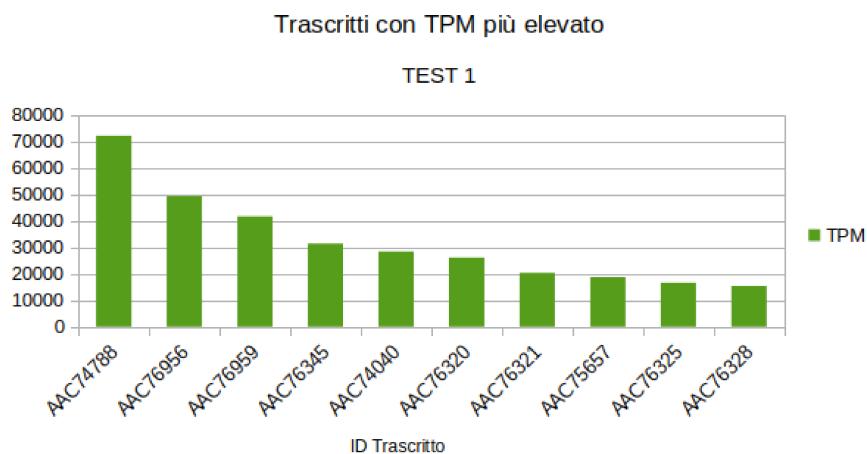


Figura 4.3: I 10 trascritti con TPM più elevato (test 1).

Questo grafico mostra come alcuni trascritti siano altamente espressi rispetto ad altri.

Analisi delle prestazioni

L'analisi ha richiesto un tempo totale di CPU di **5 minuti**. Dal punto di vista della memoria, sono stati allocati **20.48 GB** di RAM, mentre il consumo massimo registrato è stato di **4.4 GB**. Questo suggerisce un uso efficiente delle risorse, poiché Kallisto ha gestito l'elaborazione senza richiedere un utilizzo intensivo della RAM, mantenendo un buon equilibrio tra prestazioni e risorse disponibili.

L'analisi è stata eseguita su un sistema con **23 core** disponibili, di cui **8** sono stati allocati per il processo, permettendo un'esecuzione parallela del calcolo e riducendo i tempi di elaborazione. Il sistema dispone di **86.6 GB** di memoria totale, senza utilizzo di memoria di swap, confermando che il carico di lavoro è stato gestito interamente dalla RAM fisica, evitando rallentamenti dovuti all'accesso al disco.

Job Metrics	
cgroup	
CPU usage time	5 minutes
CPU user time	5 minutes
CPU system time	7.1429480 seconds
Number of processes belonging to this cgroup killed by any kind of OOM killer	0
Max memory usage recorded	4.4 GB
core	
Container ID	/cvmfs/singularity.galaxyproject.org/all/mulled-v2-143e618e2d6eeef454186ce14739a2cabc5ecf64573585c71af93a1f54d7c49f0c36f37d638946b1e-0
Container Type	singularity
Cores Allocated	8
Memory Allocated (MB)	20480
Job Start Time	2025-02-10 03:48:11
Job End Time	2025-02-10 03:53:37
Job Runtime (Wall Clock)	5 minutes
cpuinfo	
Processor Count	23
meminfo	
Total System Memory	86.6 GB
Total System Swap	0 bytes
uname	
Operating System	Linux galaxy-main-set03-2.novalocal 5.14.0-427.31.1.el9_4.x86_64 #1 SMP PREEMPT_DYNAMIC Wed Aug 14 16:15:25 UTC 2024 x86_64 x86_64 GNU/Linux

Figura 4.4: Metriche di esecuzione di Kallisto su Galaxy, includendo il tempo di utilizzo della CPU, la memoria allocata e il numero di core utilizzati.

Risultati ottenuti - Secondo test

Dati utilizzati

- **Trascrittoma di riferimento:**
`Escherichia_coli_str_k_12_substr_mg1655_gca_000005845_ASM584v2.cdna.all.fa`
- **Letture forward:** `ERR2686027_1.fastq.gz`
- **Letture reverse:** `ERR2686027_2.fastq.gz`

Descrizione dei parametri

In questo secondo test sono stati modificati i seguenti parametri:

- **Perform sequence-based bias correction:** `true`

Se impostato su `true`, Kallisto applica una correzione per ridurre il bias di sequenziamento, migliorando la precisione della quantificazione. Nel test precedente era impostato su `false`, quindi la correzione non era stata eseguita.

- **Number of bootstrap samples:** `100`

Il *bootstrap* è un metodo per stimare la variabilità delle quantificazioni, ripetendo più volte il processo su dati casualizzati. Qui è stato impostato a 100, mentre nel test precedente era 0, quindi non veniva effettuata alcuna stima della variabilità.

- **Single overhang:** `true`

Se impostato su `true`, consente la quantificazione anche di letture che si sovrappongono parzialmente a un trascritto. Nel test precedente era `false`, quindi solo le letture completamente allineate venivano considerate. L'attivazione di questo parametro può migliorare la copertura dell'analisi.

Output ottenuto

Di seguito il grafico dei 10 trascritti con TPM più elevato, ottenuto dall'output di Kallisto.

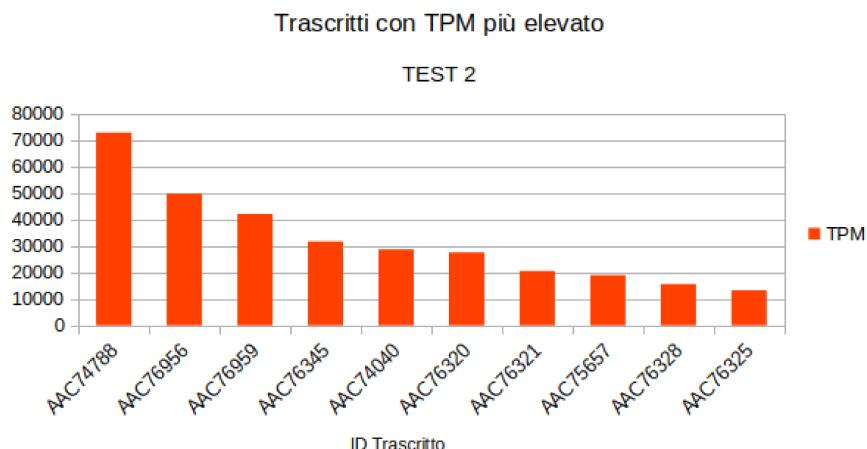


Figura 4.5: I 10 trascritti con TPM più elevato (test 2).

Dal grafico 4.5 si nota una lieve variazione nella distribuzione dei TPM nel Test 2, ma i trascritti con i valori più elevati rimangono gli stessi in entrambi i test. Questo suggerisce che la struttura complessiva dell'espressione genica non è cambiata significativamente.

Analisi delle prestazioni

L'analisi ha richiesto un tempo totale di CPU di **6 minuti**. Dal punto di vista della memoria, sono stati allocati **3.78 GB** di RAM, mentre il consumo massimo registrato è stato di **3.4 GB**. L'analisi è stata eseguita su un sistema con **16 core** disponibili, ma solo **1 core** è stato allocato per il processo.

4.2 Minimap

Risultati ottenuti

Dati utilizzati

- **Genoma di riferimento:** GCF_000005845.2_ASM584v2_genomic.fna
- **Dataset delle letture:** SRR32026183_1.fastq.gz
- **Tipo di lettura:** *single-end* (sequenziamento a lettura singola)

Descrizione dei parametri

Tra i parametri più significativi vi sono i seguenti:

- **K-mer size (-k):**
Definisce la lunghezza dei *k-mer* utilizzati per l'indicizzazione e il mapping. Valori più piccoli aumentano la sensibilità, mentre valori più grandi migliorano la velocità. Per dati **Oxford Nanopore**, un valore tipico è $k = 15$.
- **Minimizer window size (-w):**
Determina la selezione dei *k-mer* più informativi riducendo il numero di confronti durante il mapping. Per letture *long-read*, un valore tipico è $w = 10-11$.
- **Score for a sequence match (-score-match):**
Definisce il punteggio assegnato per basi correttamente allineate tra lettura e riferimento. Qui il valore è predefinito dal *preset* selezionato.
- **Penalty for a mismatch (-pen-mismatch):**
Penalizza le discordanze tra la lettura e il riferimento per ridurre falsi positivi nell'allineamento.
- **Gap open penalties for deletions (-gap-open-del):**
Definisce la penalità per l'inizio di una delezione (base presente nella lettura ma assente nel riferimento).
- **Gap open penalties for insertions (-gap-open-ins):**
Controlla il costo dell'introduzione di un'inserzione (base presente nel riferimento ma assente nella lettura).
- **Gap extension penalties (-gap-ext):**
Penalità per l'estensione di un gap (*indel*). Valori più alti scoraggiano la formazione di gap lunghi nell'allineamento.

Output ottenuto

L'output di Minimap è un file in formato PAF (Pairwise Alignment Format), contenente le seguenti informazioni per ciascuna lettura:

- **ID della lettura**
- **Lunghezza della lettura**
- **Posizione di inizio e fine sull'assemblaggio**
- **Strand (+/-)**
- **Genoma di riferimento** (NC_000913.3 per *E. coli*)
- **Percentuale di identità e score di allineamento**

Questi dati permettono di valutare l'accuratezza dell'allineamento e la copertura del genoma. L'analisi mostra che la maggior parte delle letture si allinea correttamente al genoma di riferimento, con allineamenti di alta qualità e un buon grado di copertura. Tuttavia, alcune letture presentano regioni di bassa qualità, probabilmente dovute a errori di sequenziamento o variazioni strutturali.

Il grafico seguente (4.6) è stato generato a partire dal file PAF di output, ordinando i record in ordine decrescente in base allo score di allineamento e al numero di basi mappate sul genoma di riferimento. In particolare, esso visualizza le 10 letture con il miglior allineamento, confrontando la lunghezza totale delle sequenze di lettura con il numero di basi effettivamente mappate sul genoma di riferimento.

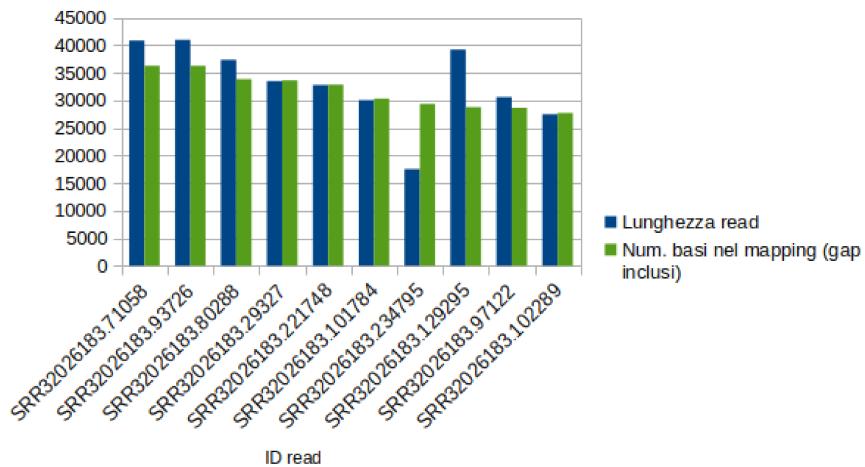


Figura 4.6: Confronto tra la lunghezza delle sequenze di lettura e il numero di basi effettivamente mappate sul genoma di riferimento.

Si nota che la maggior parte delle sequenze presenta un'alta percentuale di basi effettivamente mappate, indicando una buona qualità dell'allineamento. Tuttavia, alcune letture mostrano una discrepanza significativa tra la lunghezza della sequenza e il numero di basi mappate, suggerendo possibili regioni problematiche nell'assemblaggio o variazioni genomiche nel dataset analizzato.

Analisi delle prestazioni

L'esecuzione ha richiesto un tempo totale di CPU di **2 minuti**. L'uso della CPU è stato limitato, con **3 core** allocati su un totale di **32 core** disponibili, garantendo un'esecuzione parallela con un basso overhead e una riduzione dei tempi di elaborazione.

Dal punto di vista della memoria, sono stati allocati **14.3 GB** di RAM, con un consumo massimo registrato di **2.9 GB**. Questo dimostra che Minimap è altamente efficiente e non richiede elevate risorse di sistema per completare il processo di allineamento. Inoltre, il sistema disponeva di **119.5 GB** di memoria totale, senza utilizzo di memoria di swap, confermando che l'intero carico di lavoro è stato gestito direttamente dalla RAM fisica, evitando rallentamenti dovuti all'accesso al disco.

4.3 Miniasm

Risultati ottenuti

Dati utilizzati

- **Sequence Reads:** `reads.fq`
File contenente le letture di sequenziamento in formato FASTQ.
- **PAF file:** `minimap.paf`
File generato da Minimap, contenente le informazioni di sovrapposizione tra le letture. Questo file viene utilizzato da Miniasm per costruire il grafo di sovrapposizione senza necessità di un allineamento esplicito.

Descrizione dei parametri

Tra i parametri più significativi vi sono i seguenti:

- **Drop mappings having less than this number of matching bases:** 190
Esclude i mapping che contengono meno di 190 basi corrispondenti tra le letture. Questo filtro aiuta a eliminare allineamenti poco affidabili.
- **Drop mappings shorter than this number bp:** 1090
Rimuove i mapping più corti di 1090 bp, riducendo il rumore nell'assemblaggio.
- **Minimal coverage by other reads:** 3
Ogni lettura deve essere supportata da almeno 3 altre letture per essere inclusa nell'assemblaggio.
- **Minimal overlap length:** 1090
La lunghezza minima di sovrapposizione tra due letture è 1090 bp. Letture con sovrapposizioni più corte non verranno considerate nel grafo.
- **Maximum overhang length:** 1090
Definisce la lunghezza massima di *overhang* (regioni non allineate ai bordi delle letture) accettata nel grafo di assemblaggio.
- **Minimal ratio of mapping length to mapping + overhang length:** 0.98
Almeno il 98% della lunghezza del mapping deve corrispondere effettivamente alla lettura. Riduce il rischio di includere false sovrapposizioni.

- **Maximal gap differences between two reads in a mapping:** 1090
Definisce il gap massimo tra due letture nel grafo.
- **Maximal probing distance for bubble popping:** 50000
Distanza massima per la risoluzione di *bolle* nel grafo di sovrapposizione.
- **A unitig is considered small if it is composed of less than this many reads:** 4
Un *unitig* (segmento contiguo dell’assemblaggio) è considerato piccolo se è supportato da meno di 4 letture.

Output ottenuto

L’assemblaggio ottenuto con Miniasm è stato valutato utilizzando *QUAST* (*Quality Assessment Tool for Genome Assemblies*) [5]. I risultati mostrano che l’assemblaggio è composto da un unico contig con una lunghezza totale di **4.821.921 bp** e un valore di **N50** pari a **4.821.921 bp**, confermando l’assenza di frammentazione. Il contenuto di **GC** è del **49,73%**, un valore atteso per *E. coli* e indicativo di un’adeguata rappresentazione del genoma.

La presenza di un singolo contig che copre l’intero genoma suggerisce un’alta continuità dell’assemblaggio, un risultato tipico dell’uso di *long reads*, che permettono di superare le problematiche di frammentazione tipiche dei metodi basati su *short reads*.

Analisi delle prestazioni

Dall’esecuzione di Miniasm, è emerso che il tempo totale di utilizzo della CPU è stato di **6.71 secondi**, con un tempo di esecuzione effettivo di **10 secondi**. Durante l’elaborazione, Miniasm ha utilizzato un picco massimo di **663.9 MB** di memoria, con una memoria allocata di **32.7 GB** su un sistema che disponeva di **119.5 GB** di RAM totale. L’esecuzione è avvenuta su un **singolo core**, nonostante il sistema avesse **32 core** disponibili.

L’uso della memoria è stato contenuto, con un picco massimo di **663.9 MB**. Nel dettaglio, il processo ha coinvolto un totale di **14.361 sequenze** per una lunghezza complessiva di **135.3 milioni di bp**, con un filtraggio progressivo che ha portato a una copertura grezza finale di **23.81**. Il grafo di assemblaggio generato ha subito una serie di operazioni di pulizia, tra cui taglio di tips e rimozione di sovrapposizioni corte. Complessivamente, sono stati rimossi **10.265 archi transitivi**, **82 bolle** e diversi archi di overlap ridondanti, garantendo una struttura ottimizzata del grafo di assemblaggio.

Capitolo 5

Discussione

Kallisto

L'uso di Kallisto per la quantificazione dell'espressione genica ha confermato le aspettative in termini di velocità ed efficienza computazionale. Esso ha permesso di ottenere risultati in tempi significativamente ridotti, grazie al suo approccio di *pseudoallineamento* basato su *k-mers*.

Uno degli aspetti più rilevanti emersi dai risultati è che Kallisto fornisce stime accurate dell'abbondanza dei trascritti, con un basso consumo di memoria e una gestione efficiente di grandi dataset *RNA-seq*. Tuttavia, il metodo presenta alcune limitazioni intrinseche:

- **Sensibilità alla qualità dei dati di input:** Poiché Kallisto si basa su una rappresentazione probabilistica dell'allineamento, la qualità delle letture influisce sulla precisione della quantificazione. Letture contenenti errori o bias di sequenziamento possono portare a stime meno affidabili.
- **Limitazioni nell'identificazione di nuovi trascritti:** Poiché Kallisto richiede un trascrittoma di riferimento, non è adatto per l'identificazione di nuovi trascritti o varianti di *splicing* non annotate, limitando il suo uso in contesti di scoperta.

Nonostante questi limiti, l'efficienza computazionale e l'accuratezza delle stime rendono Kallisto un'ottima scelta per studi di espressione genica differenziale, specialmente in contesti in cui le risorse computazionali sono limitate.

Minimap

Minimap ha dimostrato di essere uno strumento estremamente efficace per il mapping di *long reads* su genomi di riferimento, grazie all'uso dei *minimizer* per ridurre il numero di confronti necessari. Dai test eseguiti, Minimap ha permesso di ottenere un mapping rapido e con un buon livello di accuratezza, rendendolo particolarmente adatto per l'analisi di dati generati da tecnologie come PacBio e Oxford Nanopore.

Tra i punti di forza di Minimap si evidenziano:

- **Alta velocità di esecuzione**, grazie all'uso dei *minimizer*.
- **Basso consumo di memoria**, che lo rende scalabile per dataset molto grandi.
- **Buona accuratezza** per il mapping di *long reads*, in particolare rispetto a metodi più tradizionali basati su allineamento completo.

Minimap, pur essendo efficiente e veloce, presenta alcune difficoltà nel gestire regioni genomiche altamente ripetitive, dove un allineamento più preciso sarebbe necessario.

Miniasm

L'assemblaggio genomico con Miniasm ha prodotto risultati interessanti, specialmente in termini di continuità dell'assemblaggio.

Miniasm offre un'alta velocità di esecuzione grazie all'eliminazione della fase di consenso tipica degli algoritmi basati sul paradigma OLC, accelerando il processo di assemblaggio e producendo un output compatto con un unico contig di grandi dimensioni, garantendo una buona continuità dell'assemblaggio.

Tuttavia, Miniasm presenta alcune limitazioni: mantenendo gli errori delle letture originali a causa della mancanza di una fase di consenso, rende spesso necessaria una successiva correzione con strumenti di polishing come Racon [13]. Inoltre, la difficoltà nella gestione di regioni ripetitive può portare a errori di collineamento delle letture, influenzando la qualità dell'assemblaggio.

Nel complesso, Miniasm è ideale per ottenere rapidamente un assemblaggio preliminare, che può essere poi migliorato con ulteriori raffinamenti.

Capitolo 6

Conclusioni

L’analisi condotta su Kallisto, Minimap e Miniasm ha evidenziato il ruolo chiave di questi strumenti nel trattamento efficiente di sequenze genomiche. Ciascuno di essi si distingue per specifiche caratteristiche che ne determinano l’applicabilità in diversi contesti della bioinformatica. **Kallisto** si è confermato un metodo estremamente veloce ed efficiente per la quantificazione dell’espressione genica, grazie al suo approccio di *pseudoallineamento* basato su k-mer. Pur presentando limitazioni nell’identificazione di nuovi trascritti e nella sensibilità alla qualità dei dati di input, la sua rapidità e accuratezza lo rendono un’opzione preferibile per studi di espressione genica differenziale, soprattutto in contesti con grandi dataset RNA-seq.

Minimap ha dimostrato di essere un efficace mapper per *long reads*, permettendo un allineamento rapido e con un uso limitato delle risorse computazionali. La sua capacità di identificare corrispondenze tra sequenze in tempi brevi lo rende ideale per l’analisi di dati generati da PacBio e Oxford Nanopore. Tuttavia, presenta alcune difficoltà nell’allineamento di regioni altamente ripetitive.

Miniasm si distingue per la sua velocità nell’assemblaggio genomico, essendo uno degli strumenti più efficienti nel generare rapidamente una bozza dell’assemblaggio senza includere una fase di consenso. Questa caratteristica accelera notevolmente il processo, ma comporta la necessità di una successiva fase di polishing per correggere gli errori derivanti dal sequenziamento. Nonostante questa limitazione, Miniasm rappresenta un’ottima scelta per ottenere un assemblaggio preliminare di alta continuità.

Il progresso delle tecnologie di sequenziamento e l’innovazione degli algoritmi bioinformatici continueranno a rendere sempre più accessibile e veloce l’analisi di grandi volumi di dati biologici, aprendo nuove opportunità per la comprensione dei meccanismi molecolari alla base della vita.

Bibliografia

- [1] Enis Afgan et al. “The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update”. In: *Nucleic Acids Research* 46.W1 (2018), W537–W544. DOI: 10.1093/nar/gky379. URL: <https://doi.org/10.1093/nar/gky379>.
- [2] Nicolas L. Bray et al. “Near-optimal probabilistic RNA-seq quantification”. In: *Nature Biotechnology* 34 (2016), pp. 525–527. DOI: 10.1038/nbt.3519.
- [3] Michael Burrows e David J. Wheeler. *A block-sorting lossless data compression algorithm*. Rapp. tecn. 124. Palo Alto, CA: Digital Equipment Corporation, 1994.
- [4] Alexander Dobin et al. “STAR: ultrafast universal RNA-seq aligner”. In: *Bioinformatics* 29.1 (gen. 2013), pp. 15–21. DOI: 10.1093/bioinformatics/bts635.
- [5] Alexey Gurevich et al. “QUAST: quality assessment tool for genome assemblies”. In: *Bioinformatics* 29.8 (2013), pp. 1072–1075. DOI: 10.1093/bioinformatics/btt086.
- [6] Daehwan Kim et al. “Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype”. In: *Nature Biotechnology* 37 (2019), pp. 907–915. DOI: 10.1038/s41587-019-0201-4.
- [7] Sergey Koren et al. “Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation”. In: *Nature Methods* (2017). DOI: 10.1038/nmeth.2474.
- [8] Ben Langmead e Steven L. Salzberg. “Fast gapped-read alignment with Bowtie 2”. In: *Nature Methods* 9 (2012), pp. 357–359. DOI: 10.1038/nmeth.1923.
- [9] Heng Li. “Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences”. In: *Bioinformatics* 32.14 (2016), pp. 2103–2110. DOI: 10.1093/bioinformatics/btw152.

- [10] Heng Li e Richard Durbin. “Fast and accurate short read alignment with Burrows–Wheeler transform”. In: *Bioinformatics* 25.14 (2009), pp. 1754–1760. DOI: [10.1093/bioinformatics/btp324](https://doi.org/10.1093/bioinformatics/btp324).
- [11] Zhenyu Li et al. “Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph”. In: *Briefings in Functional Genomics* 11.1 (2011), pp. 25–37. DOI: [10.1093/bfgp/elr035](https://doi.org/10.1093/bfgp/elr035).
- [12] Charlotte Soneson e Mauro Delorenzi. “A comparison of methods for differential expression analysis of RNA-seq data”. In: *BMC Bioinformatics* 14.1 (2013), p. 91. DOI: [10.1186/1471-2105-14-91](https://doi.org/10.1186/1471-2105-14-91).
- [13] Robert Vaser et al. “Fast and accurate de novo genome assembly from long uncorrected reads”. In: *Nature Methods* (2019). DOI: [10.1038/s41592-019-0052-2](https://doi.org/10.1038/s41592-019-0052-2).
- [14] Susana Vinga e Jonas Almeida. “Alignment-free sequence comparison—a review”. In: *Bioinformatics* 19.4 (2003), pp. 513–523. DOI: [10.1093/bioinformatics/btg005](https://doi.org/10.1093/bioinformatics/btg005).
- [15] Andrzej Zielezinski et al. “Alignment-free sequence comparison: benefits, applications, and tools”. In: *Genome Biology* 18.1 (2017), p. 186. DOI: [10.1186/s13059-017-1319-7](https://doi.org/10.1186/s13059-017-1319-7).