Università degli Studi di Salerno

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Strumenti Formali per la Bioinformatica

SPAdes: un nuovo algoritmo di assemblaggio del genoma

Autore

Simone D'Assisi

mat. 0522502038

Anno Accademico 2024/2025

Indice

1 Introduzione

Il sequenziamento del DNA ha rivoluzionato il campo della biologia, permettendo di ottenere informazioni dettagliate sulla struttura e sul funzionamento dei genomi, prima inaccessibili. Grazie all'avvento del Sequenziamento di Nuova Generazione (NGS) è stato poi possibile ridurre drasticamente sia la durata del sequenziamento, passando da anni a pochi giorni per l'intero processo, sia i costi: se nel 2001 un sequenziamento condotto con il metodo di Sanger costava 100 milioni di dollari, adesso ne costa circa 1000. Tuttavia, condurre l'analisi e l'assemblaggio dei dati genomici in modo efficiente e in tempi brevi rappresenta una sfida ancora aperta, richiedendo algoritmi efficienti e robusti per ricostruire le sequenze a partire dai frammenti letti dai sequenziatori.
In questo scenario, dal 2012 **SPAdes** (St. Petersburg genome Assembler) si è affermato come uno degli strumenti più utilizzati per l'assemblaggio del genoma a partire da dati NGS. Basato su De Bruijn Graph, SPAdes è particolarmente adatto per l'assemblaggio di genomi batterici e piccoli eucarioti, con ottimizzazioni mirate per piattaforme come Illumina e IonTorrent. Grazie all'utilizzo dei De Bruijn Graph multidimensionali, è in grado di fornire risultati di ottima qualità, combinando i risultati migliori ottenuti da ogni iterazione.

1.1 Obiettivo del Progetto

Questo progetto si propone di fornire una panoramica dettagliata del funzionamento di SPAdes, con un confronto finale anche con un altro strumento di assemblaggio affermato, cioè Velvet. Dopo un'introduzione sui fondamenti del sequenziamento del DNA e sui metodi di rappresentazione dei dati genomici attraverso i De Bruijn Graph, verrà analizzato SPAdes nel dettaglio, delineandone il funzionamento e i vari protocolli d'uso in base ai diversi scenari applicativi, tra cui l'assemblaggio di genomi batterici, dataset metagenomici, trascrittomi e set di plasmidi putativi. Verrà, inoltre, descritto l'intero modulo principale del codice di SPAdes, effettuando un'analisi funzione per funzione.
Segue un'applicazione pratica di SPAdes su un sequenziamento Illumina del batterio *Escherichia coli*. Verranno dapprima presentati gli strumenti utilizzati per il preprocessing e per il controllo di qualità dei dati di sequenziamento, cioè *FASTQC*, *Trimmomatic* e *BUSCO*, per poi procedere all'esecuzione di SPAdes e dimostrarne le potenzialità in un contesto pratico. Infine, verranno confrontati i risultati ottenuti da SPAdes con quelli ottenuti da Velvet a partire dagli stessi dati, andando così ad esaminare i punti di forza e i limiti di SPAdes, fornendo una valutazione critica della sua efficacia.
L'obiettivo di questo lavoro è quindi fornire una guida chiara e dettagliata all'uso di SPAdes, analizzandone il funzionamento, i protocolli d'uso e i metodi di valutazione della qualità, per offrire un quadro completo sulle potenzialità di questo strumento nell'ambito dell'assemblaggio genomico.

2 Concetti Chiave per l'Assemblaggio

2.1 Sequenziamento del DNA e Single-Cell Sequencing

Il sequenziamento del DNA è la determinazione dell'ordine dei diversi nucleotidi (quindi delle quattro basi azotate che li differenziano, cioè adenina, citosina, guanina e timina) che costituiscono l'acido nucleico. La sequenza del DNA contiene tutte le informazioni genetiche ereditarie della cellula (ad eccezione di quelle presenti nei mitocondri) ed è alla base dello sviluppo di tutti gli organismi viventi. Grazie a questa sequenza, vengono codificati sia i geni di ogni organismo sia le istruzioni per la loro espressione, in un processo denominato **regolazione dell'espressione genica**.
I metodi di sequenziamento tradizionale forniscono un risultato medio derivato da molte cellule, risultando quindi inefficaci quando il numero di cellule è ridotto. Si consideri, ad esempio, che la maggior parte dei batteri presenti in ambienti molto diversi, dal corpo umano all'oceano, non può essere clonata in laboratorio.
Le tecnologie di *Single-Cell Sequencing* (sequenziamento a singola cellula, o SCS) permettono di sequenziare il genoma di una singola cellula, ottenendo informazioni genomiche, trascrittomiche o multi-omiche. Questo approccio consente di rivelare differenze tra popolazioni cellulari e di ricostruire le relazioni evolutive tra le cellule [1]. Risultano quindi più efficaci delle metodologie tradizionali, in quanto permettono di analizzare campioni con quantità limitate di materiale di partenza.

2.1.1 Sequenziamento di Nuova Generazione e Illumina

Il sequenziamento di nuova generazione (**Next Generation Sequencing**, NGS) è una tecnologia di sequenziamento del DNA che ha reso più rapida la ricerca genomica rispetto al passato: grazie all'NGS, è infatti possibile sequenziare un intero genoma umano in un solo giorno, mentre, al contrario, la precedente tecnologia di sequenziamento di Sanger, utilizzata per decifrare inizialmente il genoma umano, ha richiesto oltre un decennio per produrre la bozza finale [2].
La tecnologia di sequenziamento Illumina, chiamata **Sequencing by Synthesis** (SBS), è una delle tecnologie di sequenziamento di nuova generazione (NGS) più ampiamente adottate a livello mondiale. Il workflow di sequenziamento di Illumina si divide in quattro fasi principali [3]:

1. **Preparazione del Campione**: vengono aggiunti adattatori alle estremità dei frammenti di DNA. Attraverso un'amplificazione a ciclo ridotto, vengono incorporati siti di legame per i sequenziatori, indici e regioni complementari ai due tipi di oligonucleotidi della flow cell[1].

2. **Clustering**: ogni frammento viene amplificato isotermicamente. Uno dei due oligonucleotidi presenti nella flow cell si lega a uno specifico adattatore complementare, dopodiché la polimerasi sintetizza il resto del frammento. Successivamente, il frammento a doppio filamento viene denaturato e il template originale viene scartato. Il frammento ottenuto viene amplificato tramite bridge amplification, in cui l'altra estremità del filamento aderisce al secondo adattatore e la polimerasi ne crea il filamento

[1] La flow cell è un vetrino con dei canali intermedi e ricoperto da ologonucleotidi.

complementare, generando un "ponte". Anche in questo caso, si procede con la denaturazione, ottenendo due filamenti complementari. Il processo permette di ottenere numerosi cloni che si formano simultaneamente sull'intera superficie della flow cell. I frammenti invertiti vengono scartati, mantenendo solo i cloni del frammento originale.

3. **Sequenziamento**: i nucleotidi competono nella costruzione della read a partire da un primer, ma solo uno per volta viene incorporato nella catena. Nel momento in cui il nucleotide viene aggiunto, un segnale luminoso caratteristico per ogni nucleotide viene emesso (Sequencing by Synthesis, SBS). Il segnale viene poi rilevato e il nucleotide appena incorporato viene registrato. Dopo l'incorporazione, il filamento viene separato da quello originale, e il processo viene ripetuto per milioni di read, con ogni ciclo che aggiunge un nucleotide specifico al filamento in crescita. Questo permette di ottenere milioni di read simultaneamente.

4. **Analisi dei Dati**: dopo il sequenziamento, i dati grezzi (file FASTQ) vengono sottoposti a un processo di qualità e trimming per rimuovere eventuali contaminazioni, errori o sequenze di bassa qualità.

2.1.2 Phred Quality Score

Il **Phred Quality Score** (Q) è una misura della qualità di identificazione delle basi generate dal sequenziamento del DNA. Associa a ogni base della read la probabilità p_e tale che $0 \leq p_e \leq 1$ che la base sia errata. È definito come:

$$Q = -10 \log_{10} p_e$$

2.1.3 Formato FASTQ

Il formato FASTQ è un formato di puro testo adottato come standard dagli strumenti di sequenziamento NGS. Ci sono quattro tipi di righe nel formato FASTQ [4]:

1. La prima è la riga di titolo che inizia con un "@" e spesso contiene solo un identificatore del record. Questo è un campo a formato libero, senza limiti di lunghezza, che consente di includere annotazioni o commenti arbitrari.

2. La seconda riga è quella della sequenza, che come nel formato FASTA, può essere suddivisa su più righe. Non c'è una limitazione esplicita sui caratteri attesi, ma è consigliato limitarsi ai codici a una lettera IUPAC (A per l'adenina, G per la guanina, C per la citosina, T per la timina, U per l'uracile) per il DNA o RNA e si usa convenzionalmente la lettera maiuscola. In alcuni contesti, l'uso di lettere minuscole o miste, o l'inclusione di un carattere di lacuna, può avere senso. Gli spazi bianchi, come tabulazioni o spazi, non sono ammessi.

3. La terza riga viene utilizzata per segnare la fine delle righe della sequenza e l'inizio della stringa di qualità, c'è la riga con il segno "+". Originariamente, questa includeva una ripetizione completa del testo della riga di titolo; tuttavia, per convenzione comune, questa è opzionale e la riga con il "+" può contenere solo questo carattere, riducendo significativamente la dimensione del file.

4. La quarta riga, infine, indica la qualità attraverso una codifica dei Phred Quality Score associati ad ogni base, e può anch'essa essere suddivisa su più righe. Illumina utilizza un sottoinsieme dei caratteri ASCII stampabili (al massimo i caratteri ASCII 33–126 inclusi) con una mappatura basata su un semplice offset. Dopo la concatenazione (rimuovendo i ritorni a capo), la stringa di qualità deve avere la stessa lunghezza della stringa della sequenza.

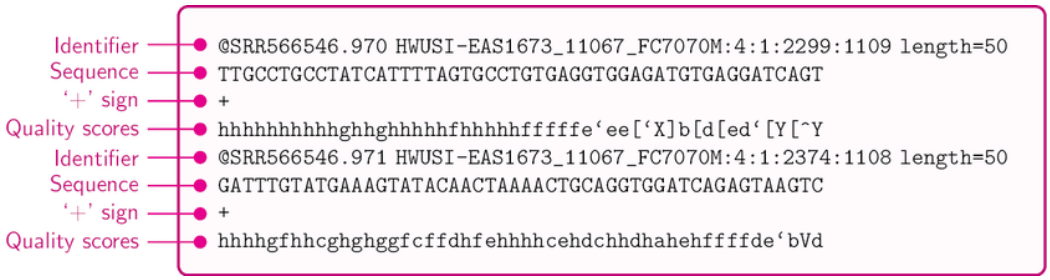


Figura 1: Esempio di sequenza FASTQ.

2.2 De Bruijn Graph

Un *grafo di De Bruijn* (o DBG) è un tipo di grafo orientato che rappresenta una sequenza di simboli in cui ogni sottostringa di lunghezza k (nota come *k-mer*) è associata a un nodo del grafo. In particolare, sia Σ un alfabeto di simboli e sia k un intero positivo, un grafo di De Bruijn DB(Σ, k) è definito da un insieme di **nodi** $V = \{v_1, v_2, \ldots, v_n\}$ che contiene tutte le sottostringhe di lunghezza k che possono essere formate combinando i simboli dell'alfabeto Σ, e da un insieme di **archi** $E = \{e_1, e_2, \ldots, e_m\}$, dove, dati due nodi A e B, esiste un arco da A a B se il suffisso di lunghezza $k-1$ di A corrisponde al prefisso di lunghezza $k-1$ di B.
Nel contesto del sequenziamento del DNA, un grafo di De Bruijn è utilizzato per rappresentare le letture (o "reads") di DNA come k-mers, dove per "letture" si intende un insieme di sequenze di nucleotidi ottenute tramite sequenziamento del DNA, provenienti dall'alfabeto $\{A, C, G, T\}$.

2.2.1 Terminologia

Nel parlare di grafi di De Bruijn (DBG), risulta utile riportare le seguenti definizioni:

- **hub**: nodo che presenta un solo arco entrante oppure un solo arco uscente.
- **h-path**: percorso che inizia e termina in un nodo hub e i cui archi intermedi sono non-hub.
- **h-edge**: arco che ha come primo nodo un nodo hub. Il primo arco di un h-path è sempre un h-edge, che risulta sempre unico per quello specifico h-path.
- **h-read** o **contig**: l'intera sequenza formata percorrendo un h-path.

2.2.2 De Bruijn Graph Multidimensionale

La scelta di k influisce sulla costruzione del grafo di De Bruijn. Valori di k piccoli comprimono più ripetizioni, rendendo il grafo più interconnesso e complesso, mentre valori elevati di k potrebbero non riuscire a rilevare sovrapposizioni tra le letture, in particolare nelle regioni a bassa copertura (cioè, nelle aree del genoma che sono state sequenziate meno frequentemente), rendendo il grafo più frammentato. Idealmente, si dovrebbero utilizzare valori più piccoli di k nelle regioni a bassa copertura (per ridurre la frammentazione) e valori più grandi di k nelle regioni ad alta copertura (per ridurre la compressione delle ripetizioni). Il grafo di De Bruijn multidimensionale consente di variare il valore di k in base alla copertura nelle diverse regioni. È definito come $DB(Reads, k-\sigma, k)$ dove $\sigma < k$ rappresenta la differenza tra la dimensione del k-mer adattato alle regioni a bassa copertura e quella utilizzata per le regioni ad alta copertura [5].

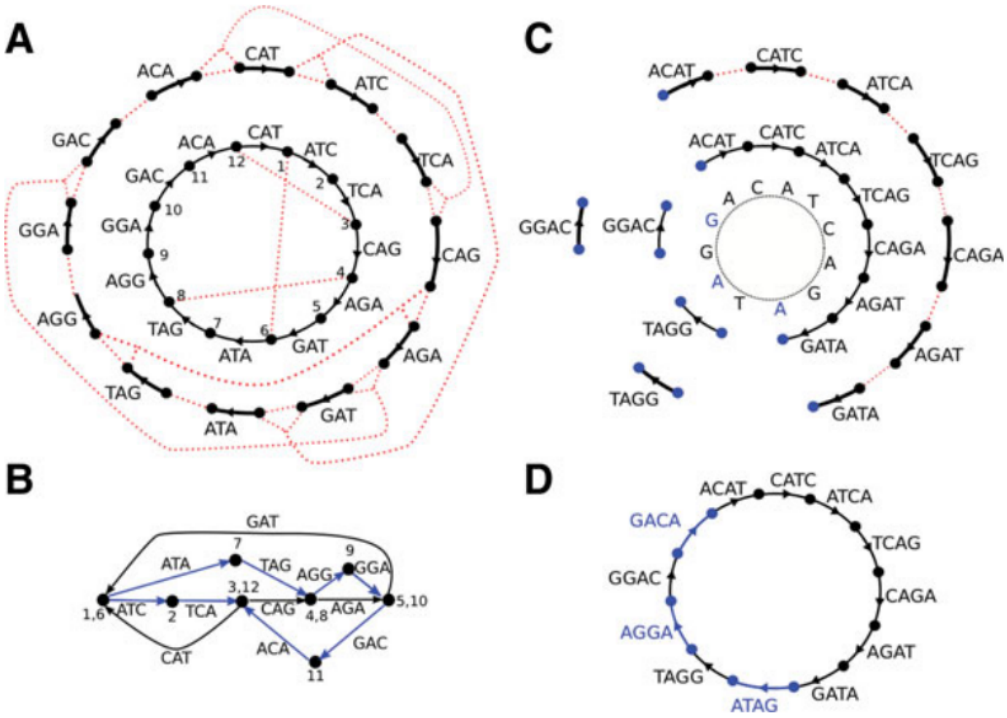


Figura 2: In questo esempio è mostrato com'è possibile utilizzare i DBG multidimensionali per collegare al grafo costituito dai 4-mer i tre frammenti presenti nell'immagine in (**C**), cioè {ATAG, AGGA, GACA}. (**A**) mostra com'è possibile collegare tutti i 3-mer presenti nell'insieme *Reads* attraverso le linee rosse tratteggiate. In (**B**) è possibile osservare il grafo $DB(Reads, 3)$, con particolare enfasi sui tre h-path di lunghezza 2 corrispondenti alle stringhe ATAG, AGGA, GACA. L'immagine in (**C**) mostra invece il grafo $DB(Reads, 4)$. Grazie al grafo $DB(Reads, 3)$ risulta ora possibile includere i frammenti privi di collegamenti. (**D**) mostra il grafo multidimensionale $DB(Reads, 3, 4)$.

2.2.3 Problematiche dei De Bruijn Graph

Nel contesto dell'assemblaggio di sequenze o di grafici di assemblaggio, i concetti di *bulge*, *tip* e *bubble* rivestono un'importanza fondamentale per l'analisi e la semplificazione della

struttura del grafo. Questi termini si riferiscono a particolari configurazioni topologiche degli archi e dei vertici, che hanno un impatto significativo sul processo di analisi e sulla qualità dell'assemblaggio. Nel processo di assemblaggio delle sequenze, è essenziale trattare e, in alcuni casi, rimuovere queste configurazioni.

Bulge. Una *bulge* è una regione in cui un ramo del grafo si separa temporaneamente in più ramificazioni, per poi convergere di nuovo. Dal punto di vista topologico, una bulge può essere vista come una configurazione di vertici e archi in cui uno o più percorsi si staccano, creando una divergenza temporanea. Le bulge possono essere dovute a variazioni nelle sequenze biologiche o a errori di sequenziamento, che devono essere corretti o trattati per evitare artefatti nell'assemblaggio.

Tip. Una *tip* è una configurazione del grafo in cui un vertice ha solo un singolo arco (o edge) in ingresso o in uscita, e quindi si trova alla fine di un percorso. Le tip sono tipicamente associate a sequenze corte o incomplete, che possono essere rimosse senza compromettere l'accuratezza complessiva dell'assemblaggio. Le tip possono essere il risultato di sequenziamenti parziali o incompleti, e la loro rimozione o la fusione con altre porzioni del grafo può aiutare a semplificare la struttura complessiva, riducendo il numero di ramificazioni inutili o erronee.

Bubble. Una *bubble* è una configurazione del grafo in cui due o più percorsi divergono da un vertice comune e si riconvertono in un altro vertice comune. Le bubble possono rappresentare varianti reali nella sequenza (come polimorfismi o ripetizioni) oppure errori introdotti durante il sequenziamento. A differenza delle bulge, le bubble hanno una struttura più chiusa e ben definita, che permette di identificarle in modo più sistematico.

3 Panoramica di SPAdes

SPAdes (St. Petersburg genome Assembler) è un assemblatore *de novo* di dati di sequenziamento genomico ottenuti dal sequenziamento del DNA genomico e da isolati microbici. In modo simile ad altri assemblatori, SPAdes mira a costruire sequenze (anche dette *contigs*) continue e accurate a partire dalle brevi read ottenute attraverso le tecnologie di sequenziamento Illumina o IonTorrent. Mentre inizialmente era stato progettato per il solo assemblaggio di genomi batterici, con il tempo è stato adattato anche per l'assemblaggio di metagenomi batterici, trascrittomi eucariotici e piccoli genomi eucariotici. Inoltre sono incluse pipeline per l'assemblaggio di plasmidi e cluster di geni biosintetici a partire dai dati di sequenziamento genomici o metagenomici.

3.1 Funzionamento

Per gestire le grandi differenze di copertura che si possono osservare nel genoma e che rappresentano un rilevante problema del SCS, SPAdes sfrutta i DBG multidimensionali, adattando la dimensione dei k-mer in base alla copertura.

Fase 1. Inizialmente, SPAdes si concentra sulla costruzione del grafo di De Bruijn multidimensionale, andando ad estrarre i k-mer dalle letture. Propone anche nuovi algoritmi per la rimozione di bulge e tip, rileva e rimuove letture chimeriche, aggrega le informazioni delle biread[2] in istogrammi di distanza e permette di tracciare le operazioni svolte sul grafo a ritroso.

Fase 2. Vengono estratti i k-bimer dalle biread, dove un k-bimer$(a|b,d)$ è costituito da una coppia di k-mer (a,b) che hanno una distanza genomica d tra loro pari al numero di basi che separano l'inizio del k-mer a dall'inizio del k-mer b all'interno della sequenza di DNA originale. La stima della distanza può, tuttavia, risultare imprecisa a causa di inserzioni/delezioni nel DNA. Il processo di *k-bimer adjustment* permette di trasformare il set di k-bimer in modo che la distanza stimata sia esatta o quasi esatta. I k-bimer regolati vengono poi sostituiti agli originali. Nella selezione dei k-bimer regolati, il sistema può decidere di sostituire un k-bimer con uno nuovo, anche se i k-mer considerati non erano nella stessa biread. L'approccio di regolazione prevede quattro trasformazioni:

- **B-tranformation**: decompone i bireads in k-bimer.
- **H-transformation**: modifica la stima della distanza genomica di un k-bimer che collega due h-path, convertendola in una stima della distanza tra le coppie di h-edge corrispondenti a queste h-path. Queste stime vengono raccolte in un istogramma.
- **A-transformation**: analizza l'istogramma delle stime imprecise di distanza tra due h-path per ottenere stime più precise.
- **E-transformation**: trasforma le distanze genomiche stimate tra h-path in set di k-bimer con distanze genomiche giuste, andando così a formare i k-bimer regolati.

[2]Le biread sono coppie di letture provenienti da tecnologie di sequenziamento di tipo *paired-end*, dove il DNA viene sequenziato alle estremità opposte con una data distanza tra di loro.

Fase 3. Viene costruito il *paired assembly graph*: le informazioni ottenute dalla precedente analisi vengono integrate nel grafo di De Bruijn, in modo che siano presenti solo h-edge con distanze corrette.

Fase 4. Grazie alle informazioni ottenute, SPAdes è ora in grado di costruire i contig finali utilizzando il paired assembly graph.

3.2 Protocolli d'Uso

SPAdes utilizza cinque protocolli di base per le principali pipeline implementate, oltre a diversi protocolli di supporto, come verrà illustrato di seguito [6].

3.2.1 Requisiti Necessari

Al fine di ottimizzare i risultati ottenuti dall'esecuzione di SPAdes per l'assemblaggio, sono consigliate le seguenti risorse:

- **Hardware**: sistema con Linux a 64-bit o MacOS con il maggior quantitativo di memoria fisica possibile (dipende fortemente dai dati di input). Per i protocolli di base 4 e 5 è consigliato un minimo di 8 GB di RAM.
- **Software**: Phyton e SPAdes.
- **File di Input**: Read nel formato FASTA o FASTQ. Sono consigliate letture di almeno 100bp.

3.2.2 Protocollo di Base 1: Assemblaggio di Dataset di Batteri Isolati

Questo protocollo descrive il processo di assemblaggio di base per un dataset di un isolato multi-cellulare utilizzando **SPAdes**. Si assume che l'input sia un insieme di letture ottenute da un singolo genoma batterico sequenziato con un'elevata profondità di copertura. Inoltre, si presume che la copertura sia uniforme lungo tutto il genoma. È strutturato in cinque fasi:

1. Classificare i dati di input: le letture non accoppiate possono essere fornite in ordine arbitrario, ma per ottenere le migliori prestazioni è consigliato fornire delle librerie *paired-end* e *mate-pair* in ordine di grandezza, dalla più piccola alla più grande. Se più librerie hanno dimensioni simili, è consigliabile unirle in un'unica libreria.
2. Specificare correttamente i dati di input tramite opzioni da riga di comando;
3. Impostare parametri aggiuntivi: anche se non necessari, è possibile configurare parametri avanzati per migliorare la qualità dell'assemblaggio.
4. Eseguire la pipeline di SPAdes attraverso la seguente riga di comando:

```
spades.py --isolate input_data_parameters
additional_parameters -o output_folder
```

5. Verificare che la pipeline sia stata eseguita con successo: nel caso si siano verificati errori, verranno riportati nel log. La cartella di output conterrà i seguenti file:

- `contigs.fasta`: contiene i contigs in formato FASTA;
- `scaffolds.fasta`: contiene le sequenze scaffold in formato FASTA;
- `assembly_graph.gfa`: contiene il grafo di assemblaggio e percorsi degli scaffold in formato GFA 1.0;
- `assembly_graph.fastg`: contiene il grafo di assemblaggio in formato FASTG;
- `contigs.paths`: contiene i percorsi nel grafo di assemblaggio corrispondenti a *contigs.fasta.*;
- `scaffolds.paths`: contiene i percorsi nel grafo di assemblaggio corrispondenti a *scaffolds.fasta.*;
- `spades.log`: file contenente tutti i messaggi di log;

3.2.3 Protocollo di Base 2: Assemblaggio di Dataset Metagenomici

Questo protocollo descrive il processo di assemblaggio di dataset metagenomici mediante **metaSPAdes**, che risulta in grado di assemblare dati di sequenziamento provenienti da una miscela di batteri provenienti da specie strettamente correlate in quantità diverse. L'obiettivo è quello di ricostruire un "core metagenome", cioè l'insieme di geni condivisi dai vari batteri delle specie considerate.

1. Specificare correttamente i dati di input attraverso riga di comando, oppure con un file di configurazione YAML.
2. Impostare parametri aggiuntivi: anche in questo caso i parametri aggiuntivi non sono necessari, ma è comunque possibile configurare parametri avanzati per migliorare la qualità dell'assemblaggio.
3. Eseguire la pipeline di metaSPAdes attraverso la seguente riga di comando:

   ```
   metaspades.py input_data_parameters
   additional_parameters -o output_folder
   ```

   La cartella di output verrà creata automaticamente nel caso in cui non venga specificata una cartella di destinazione.
4. Verificare che la pipeline sia stata eseguita con successo: nel caso si siano verificati errori, verranno riportati nel log. La cartella di output conterrà i seguenti file:

   - `contigs.fasta`: contiene i contigs in formato FASTA;
   - `scaffolds.fasta`: contiene le sequenze scaffold in formato FASTA;
   - `assembly_graph.gfa`: contiene il grafo di assemblaggio e percorsi degli scaffold in formato GFA 1.0;
   - `assembly_graph.fastg`: contiene il grafo di assemblaggio in formato FASTG;
   - `contigs.paths`: contiene i percorsi nel grafo di assemblaggio corrispondenti a *contigs.fasta.*;
   - `scaffolds.paths`: contiene i percorsi nel grafo di assemblaggio corrispondenti a *scaffolds.fasta.*;
   - `spades.log`: file contenente tutti i messaggi di log;

3.2.4 Protocollo di Base 3: Assemblaggio di Set di Plasmidi Putativi

Questo protocollo descrive gli algoritmi di assemblaggio di plasmidi[3] a partire da dataset genomici utilizzando **plasmidSPAdes** e **metaplasmidSPAdes**. Questi si basano principalmente su SPAdes e metaSPAdes, ma con un'ulteriore fase che permette la rimozione dei contig corrispondenti ai cromosomi batterici. Sono consigliati principalmente per plasmidi ad alto numero di copie, su cui forniscono i risultati migliori; per i plasmidi a basso numero di copie è possibile utilizzare i protocolli di base 1 e 2.

1. Specificare correttamente i dati di input attraverso riga di comando, oppure con un file di configurazione YAML.

2. Impostare parametri aggiuntivi: non sono necessari, ma è comunque possibile configurare parametri avanzati per migliorare la qualità dell'assemblaggio.

3. Eseguire la pipeline di plasmidSPAdes attraverso la seguente riga di comando:

   ```
   plasmidspades.py input_data_parameters
   additional_parameters -o output_folder
   ```

   oppure quella di metaplasmidSPAdes attraverso:

   ```
   metaplasmidspades.py input_data_parameters
   additional_parameters -o output_folder
   ```

   La cartella di output verrà creata automaticamente nel caso in cui non venga specificata una cartella di destinazione.

4. Verificare che la pipeline sia stata eseguita con successo: nel caso si siano verificati errori, verranno riportati nel log. La cartella di output conterrà i seguenti file:

   - `contigs.fasta`: contiene i contigs in formato FASTA;
   - `scaffolds.fasta`: contiene le sequenze scaffold in formato FASTA;
   - `assembly_graph.gfa`: contiene il grafo di assemblaggio e percorsi degli scaffold in formato GFA 1.0;
   - `assembly_graph.fastg`: contiene il grafo di assemblaggio in formato FASTG;
   - `contigs.paths`: contiene i percorsi nel grafo di assemblaggio corrispondenti a *contigs.fasta.*;
   - `scaffolds.paths`: contiene i percorsi nel grafo di assemblaggio corrispondenti a *scaffolds.fasta.*;
   - `spades.log`: file contenente tutti i messaggi di log;

[3] I plasmidi sono elementi genetici extracromosomici stabilmente mantenuti, che si replicano indipendentemente dai cromosomi della cellula ospite.

3.2.5 Protocollo di Base 4: Assemblaggio di Trascrittomi

Questo protocollo descrive il processo di assemblaggio del trascrittoma con **rnaSPAdes**, una pipeline implementata all'interno del pacchetto SPAdes. I dati derivati dal sequenziamento dell'RNA vengono tipicamente utilizzati per l'analisi dell'espressione genica, ma per organismi privi di un genoma di qualità, un'alternativa valida è rappresentata dall'assemblaggio de novo, che permette di ricostruire le sequenze dei trascritti direttamente dalle read di RNA senza la necessità di un genoma di riferimento.
In generale, rnaSPAdes presenta un workflow molto simile a quello di SPAdes, ma con alcuni algoritmi semplificati progettati specificamente per i trascrittomi.

1. Classificare i dati di input: le letture non accoppiate possono essere fornite in ordine arbitrario, ma per ottenere le migliori prestazioni è consigliato fornire delle librerie *paired-end* in ordine di grandezza, dalla più piccola alla più grande. Se più librerie hanno dimensioni simili, è consigliabile unirle in un'unica libreria. Questa strategia è utile anche quando si vuole assemblare un trascrittoma totale a partire da campioni diversi (ad esempio partendo da tessuti diversi).

2. Determinare la strand-specificity nelle librerie delle read corte: generalmente i kit delle librerie specificano se i filamenti ottenuti sono strand-specific, cioè se mantengono l'informazione sull'orientamento del filamento d'origine. Può tuttavia capitare che questa informazione può mancare, e risulta dunque utile determinarla.

3. Specificare correttamente i dati di input attraverso riga di comando, oppure con un file di configurazione YAML.

4. Impostare parametri aggiuntivi: non sono necessari, ma è comunque possibile configurare parametri avanzati per migliorare la qualità dell'assemblaggio.

5. Eseguire la pipeline di rnaSPAdes attraverso la seguente riga di comando:

   ```
   rnaspades.py input_data_parameters
   additional_parameters -o output_folder
   ```

   La cartella di output verrà creata automaticamente nel caso in cui non venga specificata una cartella di destinazione.

6. Verificare che la pipeline sia stata eseguita con successo: nel caso si siano verificati errori, verranno riportati nel log. La cartella di output conterrà i seguenti file:

   - `transcripts.fasta`: contiene i trascritti in formato FASTA;
   - `assembly_graph.gfa`: contiene il grafo di assemblaggio e percorsi degli scaffold in formato GFA 1.0;
   - `assembly_graph.fastg`: contiene il grafo di assemblaggio in formato FASTG;
   - `transcripts.paths`: contiene i percorsi nel grafo di assemblaggio corrispondenti a *transcripts.fasta.*;
   - `spades.log`: file contenente tutti i messaggi di log;

3.2.6 Protocollo di Base 5: Assemblaggio di Cluster di Geni Biosintetici Putativi

Questo protocollo di base descrive il processo di assemblaggio dei cluster di geni biosintetici (biosynthetic gene clusters, o BGC)[4] con **biosyntheticSPAdes**, che utilizza una singola libreria di letture accoppiate, con la possibilità di includere librerie di letture lunghe opzionali, e produce un assemblaggio delle sequenze nucleotidiche di potenziali biosynthetic gene clusters.
La pipeline di biosyntheticSPAdes si basa su quella di metaSPAdes, ma con moduli aggiuntivi che permettono di annotare i domini BGC nel grafo di assemblaggio, ridurre il numero di mismatch e enumerare tutti i potenziali BGC.

1. Specificare correttamente i dati di input attraverso riga di comando, oppure con un file di configurazione YAML.

2. Impostare parametri aggiuntivi: non sono necessari, ma è comunque possibile configurare parametri avanzati per migliorare la qualità dell'assemblaggio.

3. Eseguire la pipeline di biosyntheticSPAdes attraverso la seguente riga di comando:

   ```
   spades.py --bio input_data_parameters
   additional_parameters -o output_folder
   ```

   La cartella di output verrà creata automaticamente nel caso in cui non venga specificata una cartella di destinazione.

4. Verificare che la pipeline sia stata eseguita con successo: nel caso si siano verificati errori, verranno riportati nel log. La cartella di output conterrà i seguenti file:

   - `gene_clusters.fasta` : contiene le sequenze di cluster dei geni in formato FASTA;
   - `bgc_statistics.txt`: contiene le informazioni di base in riferimento ai cluster di geni;
   - `domain_graph.dot`: grafo di dominio in formato dot;
   - `spades.log`: file contenente tutti i messaggi di log;

3.3 Fattori di Rischio nell'Esecuzione di SPAdes

L'assemblaggio con SPAdes può essere influenzato da diversi fattori di rischio che possono compromettere la qualità dell'assemblaggio. Questi fattori devono essere monitorati e gestiti per evitare errori o inefficienze. Di seguito sono riportati i principali fattori di rischio.

[4] I biosynthetic gene clusters sono operoni, cioè geni adiacenti codificati da un unico primer, che codificano per metaboliti secondari e sono spesso presenti nei genomi batterici e fungini.

Scelta della dimensione dei k-mer. Per tutti gli assemblatori basati sui grafi di De Bruijn, la scelta della dimensione dei k-mer rappresenta un fattore critico: una scelta errata può compromettere la qualità dell'assemblaggio. L'utilizzo dei DBG multidimensionali permette tuttavia di arginare notevolmente questo problema, in quanto non viene utilizzato il miglior valore di k ottenuto nelle varie iterazioni, ma utilizza tutte le informazioni ottenute per tutti i valori di k utilizzati.
Ogni pipeline di SPAdes rileva autonomamente un insieme di k ottimali o sub-ottimali a seconda del tipo di dato e della lunghezza delle read considerate. Generalmente, vengono selezionati valori di $k = 21, 33, 55$ se la lunghezza delle letture è inferiore a 150 bp, $k = 21, 33, 55, 77$ per read comprese tra 150 bp e 249 bp, e $k = 21, 33, 55, 77, 99, 127$ per letture superiori a 250 bp.
È consigliabile non modificare l'insieme di default di valori di k utilizzati da SPAdes e di non utilizzare tool di selezione della lunghezza dei k-mer come *KmerGenie*, in quanto non progettati per assemblatori multi-k-mer [6], e non utilizzare metriche di valutazione come *N50*, perché valori di k maggiori potrebbero fornire un N50 migliore a costo di un tasso elevato di mis-assembly, a causa di lacune di copertura nelle regioni a bassa copertura del genoma.

Presenza di errori nelle read date in input. La presenza di errori nelle read prima dell'assemblaggio può ridurre le prestazioni di SPAdes e la qualità dell'assemblaggio stesso. Le procedure di correzione degli errori possono essere avviate dallo stesso SPAdes, grazie nello specifico a due moduli: **BayesHammer** [7], pensato per la correzione degli errori di sequenziamento Illumina, e **IonHammer** [8], pensato per la correzione degli errori di sequenziamento IonTorrent. Se le operazioni di trimming e correzione sono state eseguite esternamente, è possibile disabilitare questi strumenti con il comando `--only-assembler`.
È possibile procedere anche alla sola correzione degli errori (andando quindi ad evitare l'assemblaggio), utilizzando la flag `--only-error-correction`, che di default utilizzerà il modulo BayesHammer. Per utilizzare IonHammer é necessario utilizzare anche la flag `--iontorrent`.

Presenza di errori di mismatch e di indel brevi nell'assemblaggio. I mismatch e gli indel sono tipici problemi che possono verificarsi durante un assemblaggio: un mismatch si verifica quando una base nella sequenza del read non corrisponde alla sequenza di riferimento o a quella assemblata, mentre un indel si verifica quando una base o un gruppo di basi è stato aggiunto (insertion) o rimosso (deletion) da una sequenza di una read rispetto alla sequenza di riferimento o assemblata.
SPAdes integra il modulo **MisMatchCorrector** per migliorare gli assemblaggi ottenuti e correggere gli errori. Per abilitare il polishing è necessario utilizzare la flag `--careful`, ma può essere utilizzata solo se viene usata la pipeline standard di assemblaggio del genoma di SPAdes.

File di input corrotti. Affinchè l'input di SPAdes si a corretto sia a livello strutturale che logico, è necessario che i file utilizzato siano nel corretto formato FASTA/FASTQ. Trai più comuni errori che possono verificarsi c'è lo scorretto allineamento della linea di sequenza con la linea di qualità, oppure l'impossibilità del software di rilevare correttamente

l'offset di Phred nel file. Un altro errore comune è utilizzare dataset paired-end senza una corrispondenza tra destra e sinistra a causa di trimmer non paired-end o altri strumenti di pre-elaborazione. In questo caso vengono di solito lanciati messaggi di errore, e gli errori di questo tipo sono fatali: l'unica soluzione è correggere l'input.

Memoria insufficiente. Rappresentano il tipo di errore più frequente. Anche se SPAdes cerca di stimare lo spazio necessario a partire dal file di input scelto, tuttavia questa stima non risulta sempre corretta e il software terminerà prematuramente la propria esecuzione.
In questi casi non è sempre possibile una segnalazione efficacie, perché anche il lancio di un messaggio di errore richiederebbe l'utilizzo di memoria aggiuntiva.

Problemi di I/O. SPAdes alloca su disco alcuni file scratch a cui accede più volte durante i vari passaggi della pipeline. Se le operazioni di lettura e scrittura di questi file non avvengono correttamente, questo generalmente porta a errori fatali e all'interruzione del processo.
Tuttavia, spesso questo problema può essere risolto con un semplice riavvio del sistema.

Problemi con l'orientamento delle paired-end read. A volte l'orientamento delle paired-end reads non è standard e deve essere specificato esplicitamente. Questo accade più frequentemente con i mate-pairs, poiché il loro orientamento potrebbe essere specifico per un particolare protocollo di preparazione della libreria e di sequenziamento.
Mentre SPAdes fornisce un valore predefinito, cerca anche di rilevare se l'orientamento delle paired-end reads o dei mate-pairs è stato specificato correttamente. Se non lo è, di solito viene visualizzato un avviso riguardo a una dimensione dell'inserto negativa, e all'utente viene consigliato di verificare se l'orientamento è stato specificato correttamente.

Problemi con la distribuzione della lunghezza dei frammenti. SPAdes si affida alla distribuzione della lunghezza dei frammenti delle paired-end reads per il processo di risoluzione delle ripetizioni e di scaffolding. Tuttavia, a causa di un assemblaggio molto frammentato e della corruzione delle letture di input, la stima della lunghezza delle read può non essere eseguita correttamente.
Viene in questo caso suggerito all'utente di verificare i parametri utilizzati ed i dati di input.

Problemi legati alla variabilità della coverage. Poiché quasi tutti i protocolli di SPAdes (ad eccezione della modalità single-cell e di metaSPAdes) si basano sull'ipotesi di coverage uniforme, un'elevata variabilità della coverage può portare ad un risultato scadente dell'assemblaggio.
SPAdes cerca di rilevare se l'ipotesi di coverage uniforme è stata violata in un qualsiasi punto della pipeline, sia prima che dopo l'assemblaggio. Se tale ipotesi viene rilevata, ciò può avere diversi risultati, dall'emissione di un semplice warning all'interruzione del processo.

3.4 Analisi del Codice

Il codice analizzato farà riferimento alla versione 4.1.0 di SPAdes, rilasciata il 21/02/2025 [9].

3.4.1 Gestione del workflow di SPAdes

Il file `spades.py` è il file principale del software SPAdes e gestisce l'intero workflow del processo di assemblaggio, dalla lettura degli input alla generazione degli output. Di seguito ne verrà presentata l'intera struttura, con un particolare focus sulle funzioni che lo costituiscono.

print_used_values. La funzione `print_used_values(cfg , log)` serve a stampare i parametri e le configurazioni usate da SPAdes durante l'esecuzione.
Viene in primo luogo definita la funzione `print_value(cfg, section, param, pretty_param="", margin=" ")`, che servirà in seguito a stampare i valori dei parametri di configurazione in modo leggibile. Se il parametro `pretty_param` non è specificato, modifica i parametri aumentandone la leggibilità. Dopodiché recupera il valore dei parametri di configurazione da `cfg[section]`: se `param` è presente, lo aggiunge alla stringa di output; altrimenti viene verificato se è presente "offset", ed in quel caso sarà stampato "will be auto-detected".

```
def print_value(cfg, section, param, pretty_param="", margin="  "):
    if not pretty_param:
        pretty_param = param.capitalize().replace('_', ' ')
    line = margin + pretty_param
    if param in cfg[section].__dict__:
        line += ": " + str(cfg[section].__dict__[param])
    else:
        if "offset" in param:
            line += " will be auto-detected"
    log.info(line)

log.info("")
```

Codice 1: `print_used_values`, funzione `print_value`.

La seconda parte del codice ha l'obiettivo di stampare le informazioni di sistema nel log di SPAdes: vengono indicate la versione di SPAdes, di Python e il sistema operativo in uso al momento dell'esecuzione.

```
log.info("System information:")
try:
    log.info("  SPAdes version: " + str(spades_version).strip())
    log.info("  Python version: " + ".".join(map(str, sys.version_info[0:3]))
    )
    # for more details: '[' + str(sys.version_info) + ']'
    log.info("  OS: " + platform.platform())
    # for more details: '[' + str(platform.uname()) + ']'
except Exception:
    log.info("  Problem occurred when getting system information")
log.info("")
```

Codice 2: `print_used_values`, informazioni di sistema.

La terza parte del codice permette di visualizzare la configurazione specifica selezionata durante l'esecuzione di SPAdes. Si potrà infatti scegliere di eseguire una o più delle modalità proposte da SPAdes, tra cui correzione degli errori, assemblaggio, modalità sviluppatore e altro.

```
print_value(cfg, "common", "output_dir", "", "")
if not options_storage.args.rna:

    if ("error_correction" in cfg) and (not "assembly" in cfg):
        log.info("Mode: ONLY read error correction (without assembling)")
    elif (not "error_correction" in cfg) and ("assembly" in cfg):
        log.info("Mode: ONLY assembling (without read error correction)")
    else:
        log.info("Mode: read error correction and assembling")

if ("common" in cfg) and ("developer_mode" in cfg["common"].__dict__):
    if cfg["common"].developer_mode:
        log.info("Debug mode is turned ON")
    else:
        log.info("Debug mode is turned OFF")
log.info("")
```

Codice 3: `print_used_values`, contenuto principale.

La quarta parte del codice permette di visualizzare le informazioni specifiche relative al dataset utilizzato, andandosi in particolare a concentrarsi sui dati di input e sul loro formato. Dà, inoltre, informazioni aggiuntive all'utente sulla scelta corretta della modalità di esecuzione in base al tipo di sequenziamento.

```
if "dataset" in cfg:
    log.info("Dataset parameters:")

    if options_storage.args.iontorrent:
        log.info("  IonTorrent data")
    if options_storage.args.sewage:
        log.info("  Sewage mode")
    if options_storage.args.bio:
        log.info("  BiosyntheticSPAdes mode")
    if options_storage.args.rnaviral:
        log.info("  RNA virus assembly mode")
    if options_storage.args.meta:
        log.info("  Metagenomic mode")
    elif options_storage.args.large_genome:
        log.info("  Large genome mode")
    elif options_storage.args.isolate:
        log.info("  Isolate mode")
    elif options_storage.args.rna:
        log.info("  RNA-seq mode")
    elif options_storage.args.single_cell:
        log.info("  Single-cell mode")
    else:
        log.info("  Standard mode")
        log.info("  For multi-cell/isolate data we recommend to use '--
    isolate' option;" \
                     " for single-cell MDA data use '--sc';" \
                     " for metagenomic data use '--meta';" \
                     " for RNA-Seq use '--rna'.")
```

```
28
29      log.info("  Reads:")
30      dataset_data = pyyaml.load(open(cfg["dataset"].yaml_filename))
31      dataset_data = support.relative2abs_paths(dataset_data, os.path.dirname(
        cfg["dataset"].yaml_filename))
32      support.pretty_print_reads(dataset_data, log)
```

Codice 4: `print_used_values`, dataset.

La quinta parte del codice controlla se la correzione degli errori è attiva (la chiave "error-correction" è presente in cfg). Si occuperà poi di stampare il numero massimo di iterazioni che verranno eseguite per la correzione degli errori e il valore dell'offset Phred (`qvoffset`). Infine, specifica se le read corrette saranno o meno compresse.

```
1  if "error_correction" in cfg:
2      log.info("Read error correction parameters:")
3      print_value(cfg, "error_correction", "max_iterations", "Iterations")
4      print_value(cfg, "error_correction", "qvoffset", "PHRED offset")
5
6      if cfg["error_correction"].gzip_output:
7          log.info("  Corrected reads will be compressed")
8      else:
9          log.info("  Corrected reads will NOT be compressed")
```

Codice 5: `print_used_values`, correzione errori.

L'ultima parte del codice stampa i parametri relativi all'assemblaggio. Verifica, ad esempio, se la lunghezza dei k-mer può essere definita automaticamente in base alla lunghezza delle read rilevata dalla lettura del dataset, se il programma deve lavorare con plasmidi (opzione `plasmid`), se è necessario risolvere i problemi relativi alle sequenze ripetute (opzione `disable_rr`), se è necessario ridurre il numero di errori da mismatch (opzione `careful`), e altro ancora. Inoltre, viene specificato il formato del grafo di assemblaggio (GFA v1.1 o v1.2).

```
1  if "assembly" in cfg:
2      log.info("Assembly parameters:")
3      if options_storage.auto_K_allowed():
4          log.info("  k: automatic selection based on read length")
5      else:
6          print_value(cfg, "assembly", "iterative_K", "k")
7      if options_storage.args.plasmid:
8          log.info("  Extrachromosomal mode is turned ON")
9      if cfg["assembly"].disable_rr:
10         log.info("  Repeat resolution is DISABLED")
11     else:
12         log.info("  Repeat resolution is enabled")
13     if options_storage.args.careful:
14         log.info("  Mismatch careful mode is turned ON")
15     else:
16         log.info("  Mismatch careful mode is turned OFF")
17     if "mismatch_corrector" in cfg:
18         log.info("  MismatchCorrector will be used")
19     else:
20         log.info("  MismatchCorrector will be SKIPPED")
21     if cfg["assembly"].cov_cutoff == "off":
22         log.info("  Coverage cutoff is turned OFF")
23     elif cfg["assembly"].cov_cutoff == "auto":
```

```
24          log.info("  Coverage cutoff is turned ON and threshold will be auto-
        detected")
25      else:
26          log.info("  Coverage cutoff is turned ON and threshold is %f" % cfg["
        assembly"].cov_cutoff)
27      if cfg["assembly"].gfa11:
28          log.info("  Assembly graph output will use GFA v1.1 format")
29      else:
30          log.info("  Assembly graph output will use GFA v1.2 format")
31
32  log.info("Other parameters:")
33  print_value(cfg, "common", "tmp_dir", "Dir for temp files")
34  print_value(cfg, "common", "max_threads", "Threads")
35  print_value(cfg, "common", "max_memory", "Memory limit (in Gb)", "  ")
36  log.info("")
```

Codice 6: `print_used_values`, assembly.

create_logger. La funzione `create_logger()` crea e configura un logger per il sistema, specificando il formato del log ed il suo livello (che in questo caso è `DEBUG`), e aggiungendo l'handler del log alla console.

```
1  def create_logger():
2      log = logging.getLogger("spades")
3      log.setLevel(logging.DEBUG)
4
5      console = logging.StreamHandler(sys.stdout)
6      console.setFormatter(logging.Formatter("%(message)s"))
7      console.setLevel(logging.DEBUG)
8      log.addHandler(console)
9      return log
```

Codice 7: `create_logger`

check_cfg_for_partial_run. La funzione `check_cfg_for_partial_run()` verifica se una specifica fase di un processo, come ad esempio il "restart" o uno "stop" parziale, è correttamente configurata nel file di configurazione cfg. In caso contrario, l'esecuzione viene interrotta.

```
1  def check_cfg_for_partial_run(cfg, partial_run_type="restart-from"):  #
       restart-from ot stop-after
2      if partial_run_type == "restart-from":
3          check_point = options_storage.args.restart_from
4          action = "restart from"
5          verb = "was"
6      elif partial_run_type == "stop-after":
7          check_point = options_storage.args.stop_after
8          action = "stop after"
9          verb = "is"
10     else:
11         return
12
13     if check_point == "ec" and ("error_correction" not in cfg):
14         support.error(
```

```
            "failed to %s 'read error correction' ('%s') because this stage %
    s not specified!" % (action, check_point, verb))
    if check_point == "mc" and ("mismatch_corrector" not in cfg):
        support.error(
            "failed to %s 'mismatch correction' ('%s') because this stage %s
    not specified!" % (action, check_point, verb))
    if check_point == "as" or check_point.startswith('k'):
        if "assembly" not in cfg:
            support.error(
                "failed to %s 'assembling' ('%s') because this stage %s not
    specified!" % (action, check_point, verb))
```

Codice 8: `check_cfg_for_partial_run`

get_options_from_params. La funzione `get_options_from_params(params_filename, running_script)` recupera le informazioni sulla precedente configurazione di SPAdes, come la riga di comando, le opzioni e lo script eseguibile.
All'inizio, controlla se esiste un file contenente i parametri delle precedenti iterazioni (`params_filename`): se non esiste, viene restituito un messaggio d'errore. Il file con i parametri viene aperto e viene letta la prima riga di comando. Dopodiché, vengono lette le righe successive in cerca della versione di SPAdes e successivamente viene verificato se la versione specificata sia compatibile con quella dell'attuale esecuzione.
Viene in seguito verificata la riga di comando per estrarre il nome dello script eseguibile e le opzioni. Anche in questo caso viene verificata la compatibilità dello script con la versione utilizzata di SPAdes.
Vengono restituite la riga di comando (`command_line`), le opzioni (`options`), il nome dello script eseguibile della precedente esecuzione (`prev_running_script`), ed un messaggio vuoto.

```
def get_options_from_params(params_filename, running_script):
    command_line = None
    options = None
    prev_running_script = None
    if not os.path.isfile(params_filename):
        return command_line, options, prev_running_script, \
               "failed to parse command line of the previous run (%s not
    found)!" % params_filename

    with open(params_filename) as params:
        command_line = params.readline().strip()
        spades_prev_version = None
        for line in params:
            if "SPAdes version:" in line:
                spades_prev_version = line.split("SPAdes version:")[1]
                break

    if spades_prev_version is None:
        return command_line, options, prev_running_script, \
               "failed to parse SPAdes version of the previous run!"
    if spades_prev_version.strip() != spades_version.strip():
        return command_line, options, prev_running_script, \
               "SPAdes version of the previous run (%s) is not equal to the
    current version of SPAdes (%s)!" \
```

```
23                % (spades_prev_version.strip(), spades_version.strip())
24    if "Command line: " not in command_line or '\t' not in command_line:
25        return command_line, options, prev_running_script, "failed to parse
    executable script of the previous run!"
26    options = command_line.split('\t')[1:]
27    prev_running_script = command_line.split('\t')[0][len("Command line: "):]
28    prev_running_script = os.path.basename(prev_running_script)
29    running_script = os.path.basename(running_script)
30    # we cannot restart/continue spades.py run with metaspades.py/rnaspades.
    py/etc and vice versa
31    if prev_running_script != running_script:
32        message = "executable script of the previous run (%s) is not equal "
    \
33                  "to the current executable script (%s)!" % (
    prev_running_script, running_script)
34        return command_line, options, prev_running_script, message
35    return command_line, options, prev_running_script, ""
```

Codice 9: `get_options_from_params`

parse_args. La funzione `parse_args(args, log)` analizza e gestisce gli elementi passati da riga di comando durante l'esecuzione di SPAdes.
In particolare, la funzione analizza gli argomenti passati da riga di comando e li inserisce in `options`, `cfg` e `dataset_data`. Se l'argomento `continue_mode` è attivo, significa che l'utente vuole riprendere un'analisi interrotta in precedenza: verranno così recuperati i dati relativi alla precedente esecuzione grazie alla funzione `get_options_from_params` (Codice 9). L'utente può anche decidere di eseguire una rianalisi (`secondary_filling`), se riprendere da un punto specifico (`restart_from`), oppure se interrompere l'esecuzione in un determinato punto (`stop_after`). Sia nel caso della ripresa che nel caso dell'interruzione in un punto specifico, viene verificata la compatibilità delle scelte dell'utente con quanto stabilito nel file cfg attraverso l'esecuzione della funzione `check_cfg_for_partial_run` (Codice 8).

```
1  def parse_args(args, log):
2      options, cfg, dataset_data = options_parser.parse_args(log, bin_home,
       spades_home, secondary_filling=False, restart_from=False)
3
4      command_line = ""
5
6      if options_storage.args.continue_mode:
7          restart_from = options_storage.args.restart_from
8          command_line, options, script_name, err_msg = get_options_from_params
       (
9              os.path.join(options_storage.args.output_dir, "params.txt"),
10             args[0])
11         if err_msg:
12             support.error(err_msg + " Please restart from the beginning or
       specify another output directory.")
13         options, cfg, dataset_data = options_parser.parse_args(log, bin_home,
        spades_home, secondary_filling=True, restart_from=(options_storage.args.
       restart_from is not None),
14             options=options)
15
16         options_storage.args.continue_mode = True
17         options_storage.args.restart_from = restart_from
```

```
18
19          if options_storage.args.restart_from:
20              check_cfg_for_partial_run(cfg, partial_run_type="restart-from")
21
22      if options_storage.args.stop_after:
23          check_cfg_for_partial_run(cfg, partial_run_type="stop-after")
24
25      support.check_single_reads_in_options(log)
26      return cfg, dataset_data, command_line
```

Codice 10: `parse_args`

add_file_to_log. La funzione `add_file_to_log(cfg, log)` aggiunge un handler al logger di Python, in modo che tutte le informazioni vengano scritte su uno specifico file di log. Se la `continue_mode` è attiva, cioè se l'utente sta riprendendo un'analisi precedentemente interrotta, il file di log sarà aperto in modalità *append*, altrimenti in modalità *write*.

```
1  def add_file_to_log(cfg, log):
2      log_filename = os.path.join(cfg["common"].output_dir, "spades.log")
3      if options_storage.args.continue_mode:
4          log_handler = logging.FileHandler(log_filename, mode='a')
5      else:
6          log_handler = logging.FileHandler(log_filename, mode='w')
7      log.addHandler(log_handler)
8      return log_filename, log_handler
```

Codice 11: `add_file_to_log`

get_restart_from_command_line. La funzione `get_restart_from_command_line(args)` aggiorna i parametri della linea di comando.
Vengono filtrati gli elementi contenuti nel parametro args dato in input, andando ad escludere i parametri che corrispondono a `-o` (che indica una directory di output) e ad `--restart-form` che indica da quale punto riprendere l'esecuzione. Viene infine creato un messaggio che indica da quale punto si sta riprendendo, che include anche il valore dei parametri aggiornati.

```
1  def get_restart_from_command_line(args):
2      updated_params = ""
3      for i in range(1, len(args)):
4          if not args[i].startswith("-o") and not args[i].startswith("--restart
       -from") and \
5                          args[i - 1] != "-o" and args[i - 1] != "--restart-
       from":
6              updated_params += "\t" + args[i]
7
8      updated_params = updated_params.strip()
9      restart_from_update_message = "Restart-from=" + options_storage.args.
       restart_from + "\n"
10     restart_from_update_message += "with updated parameters: " +
       updated_params
11     return updated_params, restart_from_update_message
```

Codice 12: `get_restart_from_command_line`

get_command_line. La funzione `get_command_line(args)` costruisce una stringa di comando a partire dagli argomenti contenuti in `args`. Nell'eseguire questa operazione, tutti i percorsi relativi vengono sostituiti con quelli assoluti.

```
def check_dir_is_empty(dir_name):
    if dir_name is not None and \
            os.path.exists(dir_name) and \
            os.listdir(dir_name):
        support.warning("output dir is not empty! Please, clean output
    directory before run.")
```

Codice 13: `get_command_line`

print_params. La funzione `print_params(log, log_filename, command_line, args, cfg)` stampa le informazioni relative alla configurazione e alla riga di comando.
Nel caso di `continue_mode` attiva, viene indicata la ripresa della pipeline e il punto specifico in cui essa avviene. Tramite la funzione `get_restart_from_command_line` (Codice 12) la riga di comando viene aggiornata ed i nuovi valori vengono stampati. Se tale modalità non è attiva, verranno stampate unicamente le informazioni relative alla riga di comando, ottenute grazie alla funzione `get_command_line` (Codice 13).
La funzione si occupa inoltre della creazione del file di log "params.txt" nel quale verranno scritte le informazioni riguardanti la configurazione.
Tutti i valori saranno stampati attraverso la funzione `print_used_values` (Codici da 1 a 6).

```
def print_params(log, log_filename, command_line, args, cfg):
    if options_storage.args.continue_mode:
        log.info("\n======= SPAdes pipeline continued. Log can be found here:
     " + log_filename + "\n")
        log.info("Restored from " + command_line)
        log.info("")

    params_filename = os.path.join(cfg["common"].output_dir, "params.txt")
    params_handler = logging.FileHandler(params_filename, mode='w')
    log.addHandler(params_handler)

    if not options_storage.args.continue_mode:
        log.info("Command line: " + get_command_line(args))
    elif options_storage.args.restart_from:
        update_params, restart_from_update_message =
    get_restart_from_command_line(args)
        command_line += "\t" + update_params
        log.info(command_line)
        log.info(restart_from_update_message)
    else:
        log.info(command_line)


    print_used_values(cfg, log)
    log.removeHandler(params_handler)
```

Codice 14: `print_params`

clear_configs. La funzione `clear_configs(cfg, log, command_before_restart_from, stage_id_before_restart_from))` rimuove i file di configurazione non più necessari quando

viene eseguito un riavvio.
Viene inizialmente definita la funzione interna `matches_with_restart_from_arg(stage, restart_from_arg)`, che controlla se il nome abbreviato di uno stadio della pipeline "short_name" corrisponde al valore contenuto in `restart_from_arg`.
La funzione carica successivamente la vecchia pipeline, attraverso il file `run_spades.yaml`, che contiene l'elenco degli stage di SPAdes. Attraverso un ciclo che itera sulla vecchia pipeline si cerca lo stage nel quale si era interrotta l'esecuzione: il suo indice verrà salvato in `restart_from_stage_id`. Viene inoltre controllato che la vecchia e la nuova pipeline siano coerenti fino al punto di restart tramite la variabile `command_before_restart_from`.
Infine, la funzione elimina il file di checkpoint necessario alla ripresa dell'esecuzione, ed elimina tutti i file di checkpoint successivi allo stadio scelto per la ripartenza, per evitare conflitti tra la vecchia e la nuova esecuzione. Rimuove, inoltre, le cartelle di configurazione degli stage eliminati.

```
def clear_configs(cfg, log, command_before_restart_from,
    stage_id_before_restart_from):
    def matches_with_restart_from_arg(stage, restart_from_arg):
        return stage["short_name"].startswith(restart_from_arg.split(":")[0])

    spades_commands_fpath = os.path.join(cfg["common"].output_dir, "
    run_spades.yaml")
    with open(spades_commands_fpath) as stream:
        old_pipeline = pyyaml.load(stream)

    restart_from_stage_id = None
    for num in range(len(old_pipeline)):
        stage = old_pipeline[num]
        if matches_with_restart_from_arg(stage, options_storage.args.
    restart_from):
            restart_from_stage_id = num
            break

    if command_before_restart_from is not None and \
        old_pipeline[stage_id_before_restart_from]["short_name"] !=
    command_before_restart_from.short_name:
        support.error("new and old pipelines have difference before %s" %
    options_storage.args.restart_from, log)

    if command_before_restart_from is None:
        first_del = 0
    else:
        first_del = stage_id_before_restart_from + 1

    if restart_from_stage_id is not None:
        stage_filename = options_storage.get_stage_filename(
    restart_from_stage_id, old_pipeline[restart_from_stage_id]["short_name"])
        if os.path.isfile(stage_filename):
            os.remove(stage_filename)

    for delete_id in range(first_del, len(old_pipeline)):
        stage_filename = options_storage.get_stage_filename(delete_id,
    old_pipeline[delete_id]["short_name"])
        if os.path.isfile(stage_filename):
            os.remove(stage_filename)
```

```
34
35          cfg_dir = old_pipeline[delete_id]["config_dir"]
36          if cfg_dir != "" and os.path.isdir(os.path.join(cfg["common"].
    output_dir, cfg_dir)):
37              shutil.rmtree(os.path.join(cfg["common"].output_dir, cfg_dir))
```

Codice 15: `clear_configs`

get_first_incomplete_command. La funzione `get_first_incomplete_command(filename)` individua il primo stage incompleto di un'analisi di SPAdes, basandosi sui file di checkpoint generati dalla pipeline. In particolare, legge gli stadi della pipeline da un file YAML, li scansiona, verifica se il checkpoint esiste e restituisce il primo stadio di cui il checkpoint è assente.

```
1  def get_first_incomplete_command(filename):
2      with open(filename) as stream:
3          old_pipeline = pyyaml.load(stream)
4
5      first_incomplete_stage_id = 0
6      while first_incomplete_stage_id < len(old_pipeline):
7          stage_filename = options_storage.get_stage_filename(
    first_incomplete_stage_id, old_pipeline[first_incomplete_stage_id]["
    short_name"])
8          if not os.path.isfile(stage_filename):
9              return old_pipeline[first_incomplete_stage_id]
10         first_incomplete_stage_id += 1
11
12     return None
```

Codice 16: `get_first_incomplete_command`

get_command_and_stage_id_before_restart_from. La funzione `get_command_and_stage_id_before_restart_f` `cfg, log)` determina da quale fase della pipeline riprendere in fase di riavvio. In particolare, la funzione recupera il nome della fase da cui far ripartire l'esecuzione (`restart_from_stage_name`), che viene determinato automaticamente nel caso in cui il riavvio debba riprendere dall'ultima fase incompleta (grazie alla funzione `get_first_incomplete_command`, Codice 16). Se necessario, viene verificato che la fase precedente sia stata completata prima di riprendere. Infine, viene restituita la fase corretta da riprendere ed il suo ID.

```
1  def get_command_and_stage_id_before_restart_from(draft_commands, cfg, log):
2      restart_from_stage_name = options_storage.args.restart_from.split(":")[0]
3
4      if options_storage.args.restart_from == options_storage.LAST_STAGE:
5          last_command = get_first_incomplete_command(os.path.join(get_stage.
    cfg["common"].output_dir, "run_spades.yaml"))
6          if last_command is None:
7              restart_from_stage_name = draft_commands[-1].short_name
8          else:
9              restart_from_stage_name = last_command["short_name"]
10
11     restart_from_stage_id = None
12     for num in range(len(draft_commands)):
13         stage = draft_commands[num]
14         if stage.short_name.startswith(restart_from_stage_name):
```

```
            restart_from_stage_id = num
            break

    if restart_from_stage_id is None:
        support.error(
            "failed to restart from %s because this stage was not specified!"
     % options_storage.args.restart_from,
            log)

    if ":" in options_storage.args.restart_from or options_storage.args.
    restart_from == options_storage.LAST_STAGE:
        return draft_commands[restart_from_stage_id], restart_from_stage_id

    if restart_from_stage_id > 0:
        stage_filename = options_storage.get_stage_filename(
    restart_from_stage_id - 1, draft_commands[restart_from_stage_id - 1].
    short_name)
        if not os.path.isfile(stage_filename):
            support.error("cannot restart from stage %s: previous stage was
    not complete." % options_storage.args.restart_from, log)
        return draft_commands[restart_from_stage_id - 1],
    restart_from_stage_id - 1
    return None, None
```

Codice 17: `get_command_and_stage_id_before_restart_from`

print_info_about_output_files. La funzione `print_info_about_output_files(cfg, log, output_files)` stampa le informazioni relative ai file di output generati dal processo di assemblaggio di SPAdes.
Nello specifico, viene inizialmente definita la funzione interna `check_and_report_output_file` che controlla se il file di output esista e, nel caso, stampa un messaggio specifico.

```
def check_and_report_output_file(output_file_key, message_prefix_text,
    error_message = ""):
    if os.path.isfile(output_files[output_file_key]):
        message = message_prefix_text + support.process_spaces(output_files[
    output_file_key])
            log.info(message)
    else:
        if error_message != "":
            log.info(error_message)
```

Codice 18: `print_info_about_output_files`, funzione interna `check_and_report_output_file`

Se l'assemblaggio ha una fase di correzione errori, vengono stampate le posizioni delle letture corrette.

```
if "error_correction" in cfg and os.path.isdir(os.path.dirname(output_files["
    corrected_dataset_yaml_filename"])):
    log.info(" * Corrected reads are in " + support.process_spaces(os.path.
    dirname(output_files["corrected_dataset_yaml_filename"]) + "/"))
```

Codice 19: `print_info_about_output_files`, error correction.

Se l'assemblaggio è presente nella pipeline, vengono stampate diverse informazioni, tra cui: il percorso dei contigs assemblati; i grafi, i cluster genici e i BGC; le informazioni sull'assemblaggio dei trascritti (nel caso in cui l'utente abbia attivato l'opzione per l'analisi dell'RNA); gli scaffold ed i loro percorsi nel grafo nel caso in cui l'analisi non sia stata condotta su RNA). Il grafo di assemblaggio viene stampato in due formati, `.fastg` e `.gfa`.

```
if "assembly" in cfg:
    error_message = ""
    if options_storage.args.plasmid:
        error_message = "No plasmid contigs assembled!!"
        if options_storage.args.meta:
            error_message = "No complete extrachromosomal contigs assembled!!
    "
    check_and_report_output_file("result_contigs_filename", " * Assembled
    contigs are in ", error_message)

    if options_storage.args.bio or options_storage.args.custom_hmms or
    options_storage.args.corona:
        check_and_report_output_file("result_domain_graph_filename", " *
    Domain graph is in ")
        check_and_report_output_file("result_gene_clusters_filename", " *
    Gene cluster sequences are in ")
        check_and_report_output_file("result_bgc_stats_filename", " * BGC
    cluster statistics ")

    if options_storage.args.rna:
        check_and_report_output_file("result_transcripts_filename", " *
    Assembled transcripts are in ")
        check_and_report_output_file("result_transcripts_paths_filename",
            " * Paths in the assembly graph corresponding to the transcripts
    are in ")

        for filtering_type in options_storage.filtering_types:
            result_filtered_transcripts_filename = os.path.join(cfg["common"
    ].output_dir,
                filtering_type + "_filtered_" +
                options_storage.transcripts_name)
            if os.path.isfile(result_filtered_transcripts_filename):
                message = " * " + filtering_type.capitalize() + " filtered
    transcripts are in " + \
                support.process_spaces(result_filtered_transcripts_filename)
                log.info(message)
    else:
        check_and_report_output_file("result_scaffolds_filename", " *
    Assembled scaffolds are in ")
        check_and_report_output_file("result_contigs_paths_filename",
            " * Paths in the assembly graph corresponding to the contigs are
    in ")
        check_and_report_output_file("result_scaffolds_paths_filename",
            " * Paths in the assembly graph corresponding to the scaffolds
    are in ")

    check_and_report_output_file("result_assembly_graph_filename", " *
    Assembly graph is in ")
    check_and_report_output_file("result_assembly_graph_filename_gfa", " *
```

```
    Assembly graph in GFA format is in ")
```

Codice 20: `print_info_about_output_files`, stampa delle informazioni.

get_output_files. La funzione `get_output_files(cfg)` genera e restituisce un dizionario con i file di output prodotti da SPAdes in base alle impostazioni di configurazione. Viene, in primo luogo, inizializzato un dizionario vuoto e impostati i percorsi principali per i contig, gli scaffold e i grafi di assemblaggio nei due formati `.fastg` e `.gfa`. Se l'analisi riguarda trascritti di RNA, statistiche BGC, cluster genici o grafi di dominio, vengono salvati anche i relativi percorsi. Se la modalità HMM è attiva, usa i nomi di file secondari per scaffold e contig.

```
def get_output_files(cfg):
    output_files = dict()
    output_files["corrected_dataset_yaml_filename"] = ""
    output_files["result_contigs_filename"] = os.path.join(cfg["common"].
    output_dir, options_storage.contigs_name)
    output_files["result_scaffolds_filename"] = os.path.join(cfg["common"].
    output_dir, options_storage.scaffolds_name)
    output_files["result_assembly_graph_filename"] = os.path.join(cfg["common
    "].output_dir, options_storage.assembly_graph_name)
    output_files["result_assembly_graph_filename_gfa"] = os.path.join(cfg["
    common"].output_dir, options_storage.assembly_graph_name_gfa)

    output_files["result_contigs_paths_filename"] = os.path.join(cfg["common"
    ].output_dir, options_storage.contigs_paths)
    output_files["result_scaffolds_paths_filename"] = os.path.join(cfg["
    common"].output_dir, options_storage.scaffolds_paths)

    output_files["result_transcripts_filename"] = os.path.join(cfg["common"].
    output_dir, options_storage.transcripts_name)
    output_files["result_transcripts_paths_filename"] = os.path.join(cfg["
    common"].output_dir, options_storage.transcripts_paths)

    output_files["result_bgc_stats_filename"] = os.path.join(cfg["common"].
    output_dir, options_storage.bgc_stats_name)

    output_files["result_domain_graph_filename"] = os.path.join(cfg["common"
    ].output_dir, options_storage.domain_graph_name)
    output_files["result_sewage_lineages_filename"] = os.path.join(cfg["
    common"].output_dir, options_storage.sewage_lineages)

    output_files["result_gene_clusters_filename"] = os.path.join(cfg["common"
    ].output_dir, options_storage.scaffolds_name)

    output_files["misc_dir"] = os.path.join(cfg["common"].output_dir, "misc")

    ### if mismatch correction is enabled then result contigs are copied to
    misc directory
    output_files["assembled_contigs_filename"] = os.path.join(output_files["
    misc_dir"], "assembled_contigs.fasta")

    output_files["assembled_scaffolds_filename"] = os.path.join(output_files[
    "misc_dir"], "assembled_scaffolds.fasta")

```

```
    if options_storage.hmm_mode():
        output_files["result_scaffolds_filename"] = os.path.join(cfg["common"].output_dir, options_storage.secondary_scaffolds_name)
        output_files["result_scaffolds_paths_filename"] = os.path.join(cfg["common"].output_dir, options_storage.secondary_scaffolds_paths)

        output_files["result_contigs_filename"] = os.path.join(cfg["common"].output_dir, options_storage.secondary_contigs_name)

    return output_files
```

Codice 21: `get_output_files`

get_stage. La funzione `get_stage(iteration_name)` determina e restituisce la fase dalla quale riprendere l'esecuzione di SPAdes. Se non siamo in "continue_mode", viene restituito il "BASE_STAGE" per la ripartenza, mentre se invece è specificato tramite la variabile "restart_from" allora viene recuperato il primo comando incompleto (grazie alla funzione `get_first_incomplete_command`, Codice 16), da cui si riprenderà l'esecuzione. Se il nome dell'iterazione corrisponde al punto di ripartenza, viene restituito il "LAST_STAGE", altrimenti il "BASE_STAGE".

```
def get_stage(iteration_name):
    if not options_storage.args.continue_mode:
        return options_storage.BASE_STAGE

    if options_storage.args.restart_from is not None and \ options_storage.args.restart_from != options_storage.LAST_STAGE:
        if ":" in options_storage.args.restart_from and \ iteration_name == options_storage.args.restart_from.split(":")[0]:
            return options_storage.args.restart_from.split(":")[-1]
        else:
            return options_storage.BASE_STAGE

    if get_stage.restart_stage is None:
        last_command = get_first_incomplete_command(os.path.join(get_stage.cfg["common"].output_dir, "run_spades.yaml"))

        if last_command is not None:
            get_stage.restart_stage = last_command["short_name"]
        else:
            get_stage.restart_stage = "finish"

    if iteration_name == get_stage.restart_stage:
        return options_storage.LAST_STAGE
    else:
        return options_storage.BASE_STAGE
```

Codice 22: `get_stage`

build_pipeline. La funzione `build_pipeline(pipeline, cfg, output_files, tmp_configs_dir, dataset_data, log, bin_home, ext_python_modules_home, python_modules_home` costruisce la pipeline di esecuzione di spades, aggiungendo in sequenza tutte le fasi di elaborazione necessarie.
Vengono inizialmente importate tutte le fasi necessarie, cioè:

- `before_start_stage`: inizializzazione;
- `preprocess_reads_stage`: correzione degli errori;
- `spades_stage`: assemblaggio principale;
- `correction_stage`: correzione post-assemblaggio;
- `check_test_stage`: controllo di qualità;
- `breaking_scaffolds_stage`: scomposizione degli scaffold;
- `terminating_stage`: chiusura del processo.

Dopodiché, ogni fase viene aggiunta alla pipeline principale tramite il metodo `add_to_pipeline`, che riceve come parametri:

- `pipeline`: l'oggetto che rappresenta la pipeline;
- `cfg`: il file di configurazione;
- `output_files`: i file di output generati;
- `tmp_configs_dir`: directory temporanea per le configurazioni;
- `dataset_data`: i dati in ingresso;
- `log`: i logger;
- `bin_home, ext_python_modules_home, python_modules_home`: i percorsi dei moduli Python necessari.

```
def build_pipeline(pipeline, cfg, output_files, tmp_configs_dir, dataset_data
    , log, bin_home, ext_python_modules_home, python_modules_home):
    from stages import before_start_stage
    from stages import error_correction_stage
    from stages import spades_stage
    from stages import correction_stage
    from stages import check_test_stage
    from stages import breaking_scaffolds_stage
    from stages import preprocess_reads_stage
    from stages import terminating_stage

    before_start_stage.add_to_pipeline(pipeline, cfg, output_files,
    tmp_configs_dir, dataset_data, log, bin_home, ext_python_modules_home,
    python_modules_home)
    preprocess_reads_stage.add_to_pipeline(pipeline, cfg, output_files,
    tmp_configs_dir, dataset_data, log, bin_home, ext_python_modules_home,
    python_modules_home)
    error_correction_stage.add_to_pipeline(pipeline, cfg, output_files,
    tmp_configs_dir, dataset_data, log, bin_home, ext_python_modules_home,
    python_modules_home)

    get_stage.cfg, get_stage.restart_stage = cfg, None
    spades_stage.add_to_pipeline(pipeline, get_stage, cfg, output_files,
    tmp_configs_dir, dataset_data, log, bin_home, ext_python_modules_home,
    python_modules_home)
```

```
    correction_stage.add_to_pipeline(pipeline, cfg, output_files,
    tmp_configs_dir, dataset_data, log, bin_home, ext_python_modules_home,
    python_modules_home)
    check_test_stage.add_to_pipeline(pipeline, cfg, output_files,
    tmp_configs_dir, dataset_data, log, bin_home, ext_python_modules_home,
    python_modules_home)
    breaking_scaffolds_stage.add_to_pipeline(pipeline, cfg, output_files,
    tmp_configs_dir, dataset_data, log, bin_home, ext_python_modules_home,
    python_modules_home)
    terminating_stage.add_to_pipeline(pipeline, cfg, output_files,
    tmp_configs_dir, dataset_data, log, bin_home, ext_python_modules_home,
    python_modules_home)
```

Codice 23: `build_pipeline`

check_dir_is_empty. La funzione `check_dir_is_empty(dir_name)` serve a verificare se la directory di output è vuota prima di eseguire SPAdes.

```
def check_dir_is_empty(dir_name):
    if dir_name is not None and \
            os.path.exists(dir_name) and \
            os.listdir(dir_name):
        support.warning("output dir is not empty! Please, clean output
    directory before run.")
```

Codice 24: `check_dir_is_empty`

Viene verificato che: il nome della directory scelta dall'utente sia valido; la directory esista; la directory scelta contenga il file specificato dall'utente. Nel caso in cui le tre condizioni vengano verificate, l'utente viene avvisato che la directory di destinazione non è vuota e che deve essere ripulita prima di proseguire.

init_parser. La funzione `init_parser(args)` si occupa di gestire e validare i parametri dati in input dall'utente.
Inizialmente, viene verificato che i parametri utilizzati dall'utente su SPAdes siano stati eseguiti per la prima volta e, nel caso, salva la linea di comando. Inoltre, viene controllato che la directory di output sia vuota (Codice 24). Se non è la prima esecuzione, viene recuperata la cartella di output dai parametri. da continuare

```
def init_parser(args):
    if options_parser.is_first_run():
        options_storage.first_command_line = args
        check_dir_is_empty(options_parser.get_output_dir_from_args())
    else:
        output_dir = options_parser.get_output_dir_from_args()
        if output_dir is None:
            support.error("the output_dir is not set! It is a mandatory
    parameter (-o output_dir).")

        command_line, options, script, err_msg = get_options_from_params(
            os.path.join(output_dir, "params.txt"),
            args[0])

        if err_msg != "":
            support.error(err_msg)
```

```
16
17          options_storage.first_command_line = [script] + options
```

Codice 25: `init_parser`

main. Costituisce il punto di ingresso principale di SPAdes, gestendone l'intero flusso, dalla configurazione all'esecuzione della pipeline.
Inizialmente, il main imposta l'ambiente, forzando l'uso della localizzazione "C" per evitare problemi con il parsing di numeri e date. Viene creato un oggetto `Pipeline` per gestire le fasi del processo, e vengono inoltre inizializzate le variabili che conterranno le informazioni riguardanti la configurazione, il dataset, la linea di comando (funzione *parse_args*, Codice 10), ed il logger (funzione `add_file_to_log`, Codice 11). Queste informazioni vengono poi stampate attraverso la funzione `print_params` (Codice 14).

```
1  os.environ["LC_ALL"] = "C"
2
3  init_parser(args)
4
5  if len(args) == 1:
6      options_parser.usage(spades_version)
7      sys.exit(0)
8
9  pipeline = Pipeline()
10
11 log = create_logger()
12 cfg, dataset_data, command_line = parse_args(args, log)
13 log_filename, log_handler = add_file_to_log(cfg, log)
14 print_params(log, log_filename, command_line, args, cfg)
15
16 if not options_storage.args.continue_mode:
17     log.info("\n======= SPAdes pipeline started. Log can be found here: " +
       log_filename + "\n")
18 support.check_binaries(bin_home, log)
```

Codice 26: `main`, inizializzazione delle variabili.

Vengono poi recuperati i file di output prodotti da SPAdes attraverso la funzione `get_output_files` (Codice 21). Ora è possibile utilizzare la funzione `build_pipeline` (Codice 23) per generare la pipeline con tutte le fasi di elaborazione dei dati. SPAdes può riprendere anche da una fase specifica attraverso la funzione `get_command_and_stage_id_before_restart_from` (Codice 17), eliminando i dati non necessari a seguito del riavvio con `clear_configs` (Codice 15). Viene di seguito generato il file di configurazione YAML, e la pipeline viene eseguita. Se l'utente non vuole solo generare la configurazione, allora esegue la pipeline completa e stampa i file di output.

```
1  try:
2      output_files = get_output_files(cfg)
3      tmp_configs_dir = os.path.join(cfg["common"].output_dir, "configs")
4
5      build_pipeline(pipeline, cfg, output_files, tmp_configs_dir, dataset_data
       , log, bin_home, ext_python_modules_home, python_modules_home)
6
7      if options_storage.args.restart_from:
8          draft_commands = pipeline.get_commands(cfg)
```

```
        command_before_restart_from, stage_id_before_restart_from = \
            get_command_and_stage_id_before_restart_from(draft_commands, cfg, log)
        clear_configs(cfg, log, command_before_restart_from, stage_id_before_restart_from)

    pipeline.generate_configs(cfg, spades_home, tmp_configs_dir)
        commands = pipeline.get_commands(cfg)

    executor = executor_save_yaml.Executor(log)
    executor.execute(commands)

    if not options_storage.args.only_generate_config:
        executor = executor_local.Executor(log)
        executor.execute(commands)
        print_info_about_output_files(cfg, log, output_files)

    if not support.log_warnings(log):
        log.info("\n======= SPAdes pipeline finished.")
```

Codice 27: `main`, costruzione ed esecuzione della pipeline.

Il main gestisce anche errori generici e specifici, controlla se si sta usando la versione sbagliata dei binari e registra gli errori nel log.

```
except Exception:
    exc_type, exc_value, _ = sys.exc_info()
    if exc_type == SystemExit:
        sys.exit(exc_value)
    else:
        import errno
        if exc_type == OSError and exc_value.errno == errno.ENOEXEC:  # Exec format error
            support.error("it looks like you are using SPAdes binaries for another platform.\n" + support.get_spades_binaries_info_message())
            else:
            log.exception(exc_value)
            support.error("exception caught: %s" % exc_type, log)
except BaseException:  # since python 2.5 system-exiting exceptions (e.g. KeyboardInterrupt) are derived from BaseException
    exc_type, exc_value, _ = sys.exc_info()
    if exc_type == SystemExit:
        sys.exit(exc_value)
    else:
        log.exception(exc_value)
        support.error("exception caught: %s" % exc_type, log)
```

Codice 28: `main`, gestione delle eccezioni.

Infine, l'ultima sezione del main è dedicata alla stampa dei riferimenti bibliografici, in base alla specifica versione di SPAdes che si è utilizzata per le analisi condotte.

```
finally:
    mode = options_parser.get_mode()
    log.info("\nSPAdes log can be found here: %s" % log_filename)
    log.info("")

    if mode == "plasmid":
```

```
        spades_name = "plasmidSPAdes"
        cite_message = ("Antipov, D., Hartwick, N., Shen, M., Raiko, M., Lapidus, A. and Pevzner, P.A., 2016. "
        "plasmidSPAdes: assembling plasmids from whole genome sequencing data. "
        "Bioinformatics, 32(22), pp.3380-3387.")
        doi = "doi.org/10.1093/bioinformatics/btw493"
    elif mode == "bgc":
        spades_name = "BiosyntheticSPAdes"
        cite_message = ("Meleshko, D., Mohimani, H., Tracanna, V., Hajirasouliha, I., Medema, M.H., "
        "Korobeynikov, A.  and Pevzner, P.A., 2019. BiosyntheticSPAdes: reconstructing "
        "biosynthetic gene clusters from assembly graphs. Genome research, 29(8), pp.1352-1362.")
        doi = "doi.org/10.1101/gr.243477.118"
    elif mode == "meta":
        spades_name = "metaSPAdes"
        cite_message = ("Nurk, S., Meleshko, D., Korobeynikov, A. and Pevzner, P.A., 2017. "
        "metaSPAdes: a new versatile metagenomic assembler. "
        "Genome research, 27(5), pp.824-834.")
        doi = "doi.org/10.1101/gr.213959.116"
    elif mode == "metaplasmid":
        spades_name = "metaplasmidSPAdes"
        cite_message = ("Antipov, D., Raiko, M., Lapidus, A. and Pevzner, P.A., 2019. "
        "Plasmid detection and assembly in genomic and metagenomic data sets. "
        "Genome research, 29(6), pp.961-968.")
        doi = "doi.org/10.1101/gr.241299.118"
    elif mode == "metaviral":
        spades_name = "metaviralSPAdes"
        cite_message = ("Antipov, D., Raiko, M., Lapidus, A. and Pevzner, P.A., 2020. "
        "Metaviral SPAdes: assembly of viruses from metagenomic data. "
        "Bioinformatics, 36(14), pp.4126-4129.")
        doi = "doi.org/10.1093/bioinformatics/btaa490"
    elif mode == "corona":
        spades_name = "coronaSPAdes"
        cite_message = ("Meleshko, D., Hajirasouliha, I. and Korobeynikov, A., 2022. "
        "coronaSPAdes: from biosynthetic gene clusters to RNA viral assemblies. "
        "Bioinformatics, 38(1), pp.1-8.")
        doi = "doi.org/10.1093/bioinformatics/btab597"
    elif mode == "rnaviral":
        spades_name = "rnaviralSPAdes"
        cite_message = ("Korobeynikov, A. and Meleshko, D., 2022. Benchmarking state-of-the-art approaches "
        "for norovirus genome assembly in metagenome sample.")
        doi = "doi.org/10.21203/rs.3.rs-1827448/v1"
    elif mode == "rna":
        spades_name = "rnaSPAdes"
        cite_message = ("Bushmanova, E., Antipov, D., Lapidus, A. and Prjibelski, A.D., 2019. "
        "rnaSPAdes: a de novo transcriptome assembler and its application to
```

```
    RNA-Seq data. "
51          "GigaScience, 8(9), p.giz100.")
52          doi = "doi.org/10.1093/gigascience/giz100"
53          else:
54          spades_name = "SPAdes"
55          cite_message = ("Prjibelski, A., Antipov, D., Meleshko, D., Lapidus,
    A. and Korobeynikov, A., 2020. "
56          "Using SPAdes de novo assembler. Current protocols in bioinformatics,
     70(1), p.e102.")
57          doi = "doi.org/10.1002/cpbi.102"
58
59      log.info("Thank you for using %s! If you use it in your research, please
    cite:" % spades_name)
60      log.info("")
61      log.info("  %s" % cite_message)
62      log.info("  %s" % doi)
63      log.info("")
64
65      log.removeHandler(log_handler)
```

Codice 29: `main`, riferimemti bibliografici.

3.4.2 Costruzione dei grafi di de Bruijn

Un passo fondamentale presente nella pipeline di SPAdes è la costruzione dei grafi di De Bruijn, utilizzati per l'assemblaggio del genoma. Ad occuparsi della costruzione dei grafi è il file `debruijn_graph_constructor.hpp`, che definisce la classe `DeBruijnGraphConstructor`, il principale responsabile della costruzione dei grafi.
Il DeBruijnConstructor utilizza una struttura dati chiamata `Index`: prima della costruzione del grafo, tutti i k-mer vengono estratti dalle read ottenute dai dati di sequenziamento e vengono inseriti nell'Index, che è una struttura simile ad una hash table. Ogni k-mer viene salvato in un nodo, mentre ogni (k+1)-mer rappresenterà l'arco che unisce due k-mer. In SPAdes, l'Index utilizzato prende il nome di `origin_`, e fornisce l'accesso a tutti i k-mer.
La costruzione dei grafi di de Bruijn utilizza numerose funzioni, di cui verranno analizzate le principali. La prima funzione presa in considerazione, cioè `StepRightIfPossible`, verifica se è possibile spostarsi a destra nel grafo, cioè se c'è un arco che collega il nodo rappresentato da *kwh* (una chiave di un k-mer) a un altro nodo.

```
bool StepRightIfPossible(KeyWithHash &kwh) {
    if (origin_.RivalEdgeCount(kwh) == 1
            && origin_.NextEdgeCount(kwh) == 1) {
        kwh = origin_.NextEdge(kwh);
        return true;
    }
    return false;
}
```

Codice 30: `stepRightIfPossible`

La funzione `GoRight` permette di spostarsi a destra nel grafo finché è possibile, cioè fino a quando non ritorna al nodo iniziale o non trova più archi, eseguendo verifiche tramite la funzione precedente (codice 30).

```
KeyWithHash &GoRight(KeyWithHash &kwh) {
```

```
    KeyWithHash initial = kwh;
    while (StepRightIfPossible(kwh) && kwh != initial) {
        ;
    }
    return kwh;
}
```

Codice 31: `GoRight`.

La funzione `GoLeft`, analogamente a `GoRight`, permette di spostarsi nel grafo, tuttavia procedendo a ritroso, seguendo i contig.

```
KeyWithHash &GoRight(KeyWithHash &kwh) {
    KeyWithHash initial = kwh;
    while (StepRightIfPossible(kwh) && kwh != initial) {
        ;
    }
    return kwh;
}
```

Codice 32: `GoLeft`.

La funzione `ConstructSeqGoingRight` costruisce una sequenza spostandosi a destra nel grafo. La sequenza viene costruita con un oggetto `SequenceBuilder`, che aggiunge k-mer finché non ci sono più archi o non si torna al nodo iniziale. Anche questo codice effettua controlli sul prossimo k-mer della sequenza, utilizzando il codice 30.

```
Sequence ConstructSeqGoingRight(KeyWithHash &kwh) {
    SequenceBuilder s;
    s.append(kwh.key());
    KeyWithHash initial = kwh;
    while (StepRightIfPossible(kwh) && kwh != initial) {
        s.append(kwh[kmer_size_]);
    }
    return s.BuildSequence();
}
```

Codice 33: `ConstructSeqGoingRight`.

La funzione `ConstructSequenceWithEdge` permette di ricostruire una sequenza a partire da un arco: dato un arco kwh, la funzione ritorna a sinistra (tramite la funzione `GoLeft`, codice 32) fino a raggiungere l'origine. Dopodiché, la sequenza viene costruita a partire dall'origine spostandosi verso destra con `ConstructSeqGoingRight` (codice 33).

```
Sequence ConstructSequenceWithEdge(const KeyWithHash &kwh) {
    KeyWithHash tmp = kwh;
    return ConstructSeqGoingRight(GoLeft(tmp));
}
```

Codice 34: `ConstructSequenceWithEdge`.

La funzione `FindVertexByOutgoingEdges` permette di cercare il nodo corrispondente ad un k-mer con una specifica sequenza di nucleotidi, esplorando gli archi uscenti.

```
VertexId FindVertexByOutgoingEdges(Kmer kmer) {
    for (char c = 0; c < 4; ++c) {
        KeyWithHash edge = origin_.ConstructKWH(kmer.pushBack(c));
        if (origin_.contains(edge))
            return graph_.EdgeStart(origin_.get_value(edge).edge_id);
```

```
    }
    return VertexId(NULL);
}
```

Codice 35: `FindVertexByOutgoingEdges`.

Di contro, la funzione `FindVertexByIncomingEdges` permette di cercare il nodo corrispondente ad un k-mer con una specifica sequenza di nucleotidi, esplorando gli archi entranti.

```
VertexId FindVertexByIncomingEdges(Kmer kmer) {
    for (char c = 0; c < 4; ++c) {
        KeyWithHash edge = origin_.ConstructKWH(kmer.pushFront(c));
        if (origin_.contains(edge)) {
            return graph_.EdgeEnd(origin_.get_value(edge).edge_id);
        }
    }
    return VertexId(NULL);
}
```

Codice 36: `FindVertexByIncomingEdges`.

La funzione `FindVertex` permette di restituire il nodo corrispondente a un k-mer specifico, esplorando prima tutti i nodi con degli specifici archi uscenti (codice 35) e poi entranti (codice 36).

```
VertexId FindVertex(Kmer kmer) {
    VertexId v = FindVertexByOutgoingEdges(kmer);
    return v == VertexId(NULL) ? FindVertexByIncomingEdges(kmer) : v;
}
```

Codice 37: `FindVertex`.

La funzione `FindVertexMaybeMissing` permette di aggiungere un nuovo nodo nel caso in cui la funzione `FindVertex` (codice 37) dia esito negativo.

```
VertexId FindVertex(Kmer kmer) {
    VertexId v = FindVertexByOutgoingEdges(kmer);
    return v == VertexId(NULL) ? FindVertexByIncomingEdges(kmer) : v;
}
```

Codice 38: `FindVertexMaybeMissing`.

La funzione `FindEndMaybeMissing` trova un nodo finale, verificando se il k-mer di fine è uguale al k-mer di inizio, se è il coniugato o se è mancante.

```
VertexId FindEndMaybeMissing(const ConjugateDeBruijnGraph& graph,
        VertexId start, Kmer start_kmer, Kmer end_kmer) {
    if (start_kmer == end_kmer) {
        return start;
    } else if (start_kmer == !end_kmer) {
        return graph.conjugate(start);
    } else {
        return FindVertexMaybeMissing(end_kmer);
    }
}
```

Codice 39: `FindEndMaybeMissing`.

La funzione `ConstructPart` costruisce solo una parte del grafo di De Bruijn, cercando i nodi di inizio (codice 38) e di fine (codice 39) e aggiungendo i corrispondenti archi alla sequenza di k-mer.

```
void ConstructPart(const std::vector<KeyWithHash>& kwh_list,
        std::vector<Sequence>& sequences) {
    for (size_t i = 0; i < sequences.size(); ++i) {
        if (origin_.contains(kwh_list[i])) {
            continue;
        }

        Kmer start_kmer = sequences[i].start < Kmer > (kmer_size_);
        Kmer end_kmer = sequences[i].end < Kmer > (kmer_size_);

        VertexId start = FindVertexMaybeMissing(start_kmer);
        VertexId end = FindEndMaybeMissing(graph_, start, start_kmer,
                end_kmer);

        graph_.AddEdge(start, end, sequences[i]);
    }
}
```

Codice 40: `ConstructPart`.

La funzione `AddKmers` aggiunge nuovi k-mer all'elenco dei k-mer presenti in un `Index`.

```
void AddKmers(kmer_iterator &it, kmer_iterator &end, size_t queueSize,
              std::vector<KeyWithHash>& kwh_list) {
    for (; kwh_list.size() != queueSize && it != end; ++it) {
        KeyWithHash kwh = origin_.ConstructKWH(Kmer(unsigned(kmer_size_ + 1),
     (*it).data()));

        if (!origin_.contains(kwh))
            kwh_list.push_back(kwh);
    }
}
```

Codice 41: `AddKmers`.

La funzione `CalculateSequences` costruisce le sequenze di k-mer in parallelo, ottimizzando le risorse e velocizzando l'operazione utilizzando *OpenMP*.

```
void CalculateSequences(std::vector<KeyWithHash> &kwh_list,
                        std::vector<Sequence> &sequences) {
    size_t size = kwh_list.size();
    sequences.resize(size);

#   pragma omp parallel for schedule(guided)
    for (size_t i = 0; i < size; ++i) {
        sequences[i] = ConstructSequenceWithEdge(kwh_list[i]);
    }
}
```

Codice 42: `CalculateSequences`.

Infine, la funzione `ConstructGraph` permette di costruire l'intero grafo di De Bruijn. Sono necessari tre parametri:

- **queueMinSize**: dimensione iniziale della coda di k-mer da elaborare;
- **queueMaxSize**: dimensione massima che la coda può raggiungere;
- **queueGrowthRate**: fattore di crescita della coda dopo ogni iterazione del ciclo. Ad esempio, un valore di 1,5 implica una crescita del 50%.

Inizialmente, vengono dichiarati due iteratori per scorrere tra tutti i (k+1)-mer presenti nell'indice `origin_`: uno corrispondente all'inizio (`kmer_begin`) e uno alla fine (`kmer_end`). Vengono inoltre inizializzate due strutture dati:

- `kwh_list`, una lista contenente oggetti `KeyWithHash`, ovvero i (k+1)-mer da elaborare;
- `sequences`, una lista che conterrà le sequenze costruite a partire dai (k+1)-mer presenti in `kwh_list`.

A entrambe le strutture viene riservata memoria tramite la funzione `reserve`, al fine di evitare riallocazioni costose durante il ciclo.
Successivamente, inizia un ciclo `while` che prosegue fino a quando non vengono analizzati tutti i (k+1)-mer dell'indice (cioè finché `it != end`). All'interno del ciclo:

1. Viene chiamata `AddKmers` (codice 41), che aggiunge a `kwh_list` fino a `queueSize` nuovi (k+1)-mer che non sono ancora stati elaborati (cioè non presenti in `origin_`). Questi rappresentano potenziali nuovi cammini da esplorare nel grafo.
2. Viene invocata `CalculateSequences` (codice 42), che costruisce per ogni elemento di `kwh_list` una sequenza lineare, ovvero un cammino non ambiguo nel grafo.
3. Con `ConstructPart` (codice 40) viene determinato il nodo iniziale e quello finale per ogni sequenza ottenuta, e viene aggiunto un arco corrispondente al cammino lineare compatto rappresentato dalla sequenza stessa.
4. Al termine dell'elaborazione della coda corrente, la lista `kwh_list` viene svuotata per prepararsi alla coda successiva.
5. Infine, la dimensione della coda viene aggiornata moltiplicandola per `queueGrowthRate`, fino a raggiungere al massimo `queueMaxSize`. In questo modo si incrementa progressivamente il carico di lavoro per ogni iterazione.

```
1  void ConstructGraph(size_t queueMinSize, size_t queueMaxSize,
2                      double queueGrowthRate) {
3      kmer_iterator it = origin_.kmer_begin();
4      kmer_iterator end = origin_.kmer_end();
5      size_t queueSize = queueMinSize;
6      std::vector<KeyWithHash> kwh_list;
7      std::vector<Sequence> sequences;
8      kwh_list.reserve(queueSize);
9      sequences.reserve(queueMaxSize);
10     while (it != end) {
11         AddKmers(it, end, queueSize, kwh_list);
12         CalculateSequences(kwh_list, sequences);
13         ConstructPart(kwh_list, sequences);
14         kwh_list.clear();
15         queueSize = std::min(size_t(double(queueSize) * queueGrowthRate),
       queueMaxSize);
16     }
17 }
```

Codice 43: `ConstructGraph`.

3.4.3 Gestione di Bulge, Tip e Bubble

Rimozione delle Bulge con *bulge_remover.hpp*. All'interno del file `bulge_remover.hpp` è possibile trovare la classe che si occupa della rimozione delle bulge nei grafi di De Bruijn, cioè `ParallelBulgeRemover`. Tale classe implementa numerose funzioni coinvolte nell'identificazione e nella rimozione delle bulge, e di seguito verranno presentate quelle principali. La funzione `FillEdgeBuffer` si occupa di riempire un buffer con gli edge da processare, fino a una dimensione massima. Controlla che gli edge da processare rispettino certe condizioni, come il limite di copertura. Inoltre, gestisce l'interruzione del processo quando non ci sono più edge da elaborare o quando le condizioni non sono soddisfatte.

```
bool FillEdgeBuffer(std::vector<EdgeId> &buffer, func::TypedPredicate<EdgeId>
     proceed_condition) {
    VERIFY(buffer.empty());
    DEBUG("Filling edge buffer of size " << buff_size_);
    utils::perf_counter perf;
    double max_cov = std::numeric_limits<double>::min();
    bool exhausted = false;
    while (!it_.IsEnd() && buffer.size() < buff_size_) {
        EdgeId e = *it_;
        TRACE("Current edge " << this->g().str(e));

        double cov = this->g().coverage(e);
        if (buffer.empty()) {
            max_cov = cov + std::max(buff_cov_diff_, buff_cov_rel_diff_ * cov
    );
            DEBUG("Coverage interval [" << cov << ", " << max_cov << "]");
        }
        if (!proceed_condition(e)) {
            DEBUG("Stop condition was reached.");
            exhausted = true;
            break;
        }
        if (math::gr(cov, max_cov)) {
            DEBUG("Coverage exceeded " << cov << " > " << max_cov);
            break;
        }
        TRACE("Potential bulge edge");
        buffer.push_back(e);
        ++it_;
    }
    exhausted |= it_.IsEnd();
    it_.ReleaseCurrent();
    DEBUG("Filled in " << perf.time() << " seconds");
    DEBUG("Candidate queue exhausted " << exhausted);
    return !exhausted;
}
```

Codice 44: `FillEdgeBuffer`.

La funzione `FindBulges` analizza ogni edge per cercare possibili configurazioni che possono formare una bulge. Le bulge vengono inserite nei buffer dalla funzione `FillEdgeBuffer` (codice 44). Le operazioni vengono eseguite in parallelo grazie all'uso di OpenMP, migliorando l'efficienza nel caso di grandi grafi.

```
std::vector<std::vector<BulgeInfo>> FindBulges(const std::vector<EdgeId>&
    edge_buffer) const {
    DEBUG("Looking for bulges in parallel");
    utils::perf_counter perf;
    std::vector<std::vector<BulgeInfo>> bulge_buffers(omp_get_max_threads());
    const size_t n = edge_buffer.size();
    //order is in agreement with coverage
    DEBUG("Edge buffer size " << n);
    #pragma omp parallel for schedule(guided)
    for (size_t i = 0; i < n; ++i) {        EdgeId e = edge_buffer[i];
        auto alternative = alternatives_analyzer_(e);
        if (!alternative.empty()) {
            bulge_buffers[omp_get_thread_num()].push_back(BulgeInfo(i, e, std
    ::move(alternative)));
        }
    }
    DEBUG("Bulges found (in parallel) in " << perf.time() << " seconds");
    return bulge_buffers;
}
```

Codice 45: `FindBulges`.

La funzione `MergeBuffers` unisce i vari buffer di bulge trovati dai diversi thread di esecuzione parallela in un unico vettore. Parallelamente, ordina le bulge in base alla priorità, che di solito è legata alla copertura (più alta copertura, maggiore priorità).

```
std::vector<BulgeInfo> MergeBuffers(std::vector<std::vector<BulgeInfo>>&&
    buffers) const {
    DEBUG("Merging bulge buffers");
    utils::perf_counter perf;

    std::vector<BulgeInfo> merged_bulges;
    for (auto& bulge_buffer : buffers) {
        std::copy(std::make_move_iterator(bulge_buffer.begin()), std::
    make_move_iterator(bulge_buffer.end()), std::back_inserter(merged_bulges))
    ;
    }

    DEBUG("Sorting");
    //order is in agreement with coverage
    std::sort(merged_bulges.begin(), merged_bulges.end());
    DEBUG("Total bulges " << merged_bulges.size());
    DEBUG("Buffers merged in " << perf.time() << " seconds");
    return merged_bulges;
}
```

Codice 46: `MergeBuffers`.

La funzione `MergeBuffers` filtra le bulge per mantenere solo quelle che non interagiscono con altre bulge o archi, ovvero quelle indipendenti. Le bulge che interagiscono con altre vengono segnate come ”interacting”, o interagenti, e saranno trattate separatamente.

```
SmartEdgeSet RetainIndependentBulges(std::vector<BulgeInfo>& bulges) const {
    DEBUG("Looking for independent bulges");
    size_t total_cnt = bulges.size();
    utils::perf_counter perf;

    std::vector<BulgeInfo> filtered;
```

```
    filtered.reserve(bulges.size());
    //fixme switch to involved vertices to bring fully parallel glueing
    closer
    EdgeSet involved_edges;
    SmartEdgeSet interacting_edges(this->g(), false, CoverageComparator<Graph
    >(this->g()));

    for (BulgeInfo& info : bulges) {
        TRACE("Analyzing interactions of " << info.str(this->g()));
        if (CheckInteracting(info, involved_edges)) {
            TRACE("Interacting");
            interacting_edges.push(info.e);
        } else {
            TRACE("Independent");
            AccountEdges(info, involved_edges);
            filtered.push_back(std::move(info));
        }
    }
    bulges = std::move(filtered);

    DEBUG("Independent bulges identified in " << perf.time() << " seconds");
    DEBUG("Independent cnt " << bulges.size());
    DEBUG("Interacting cnt " << interacting_edges.size());
    VERIFY(bulges.size() + interacting_edges.size() == total_cnt);

    return interacting_edges;
}
```

Codice 47: `RetainIndependentBulges`.

La funzione `ProcessBulges` va ad "incollare" le bulge usando il BulgeGluer: grazie a questo processo, se due o più archi formano una bulge, possono essere sostituiti con un singolo arco che unisce direttamente i nodi di partenza e di arrivo, senza passare attraverso gli archi intermedi. Le bulge indipendenti vengono processate prima, quindi si passa a quelle interagenti, che richiedono una gestione più complessa.

```
size_t ProcessBulges(const std::vector<BulgeInfo>& independent_bulges,
    SmartEdgeSet& interacting_edges) {
    DEBUG("Processing bulges");
    utils::perf_counter perf;

    size_t triggered = 0;

    for (const BulgeInfo& info : independent_bulges) {
        TRACE("Processing bulge " << info.str(this->g()));
        triggered++;
        gluer_(info.e, info.alternative);
    }

    DEBUG("Independent bulges glued in " << perf.time() << " seconds");
    perf.reset();

    DEBUG("Processing remaining interacting bulges " << interacting_edges.
    size());
    triggered += BasicProcessBulges(interacting_edges);
    DEBUG("Interacting edges processed in " << perf.time() << " seconds");
    return triggered;
```

```
}
```

Codice 48: `ProcessBulges`.

La funzione `Run` è la principale funzione che coordina l'intero processo di rimozione delle bulge. Inizialmente, verifica se l'operazione di rimozione debba essere eseguita ex novo o se debba riprendere da iterazioni precedenti. Successivamente, si procede con il loop principale, che continua fino a quando ci sono bulge da rimuovere o archi da trattare. La funzione riempie il buffer `edge_buffer` grazie alla funzione `FillEdgeBuffer` (codice 44). Se la dimensione del buffer è inferiore a una soglia definita `SMALL_BUFFER_THR`, viene eseguito un trattamento per buffer piccoli; altrimenti si procede con la funzione `FindBulges` (codice 45) per l'identificazione delle bulge, che verranno poi unite tramite la funzione `MergeBuffers` (codice 46). Le bulge interagenti e indipendenti vengono filtrate tramite la funzione `RetainIndependentBulges` (codice 47) e infine vengono processate con `ProcessBulges` (codice 48).

```
size_t Run(bool force_primary_launch, double /*iter_run_progress*/) override {
    //todo remove if not needed;
    //potentially can vary coverage threshold in coordination with ec threshold
    auto proceed_condition = func::AlwaysTrue<EdgeId>();

    bool primary_launch = force_primary_launch ;
    if (!it_.IsAttached()) {
        it_.Attach();
        primary_launch = true;
    }

    if (primary_launch) {
        it_.clear();
        DEBUG("Primary launch.");
        DEBUG("Start search for interesting edges");
        interesting_edge_finder_->Run(this->g(), [&](EdgeId e) {it_.push(e);});
        DEBUG(it_.size() << " interesting edges to process");
    } else {
        VERIFY(tracking_);
        DEBUG(it_.size() << " edges to process");
    }

    size_t triggered = 0;
    bool proceed = true;
    while (proceed) {
        std::vector<EdgeId> edge_buffer;
        DEBUG("Filling edge buffer");
        edge_buffer.reserve(buff_size_);
        proceed = FillEdgeBuffer(edge_buffer, proceed_condition);
        DEBUG("Edge buffer filled");

        DEBUG("Edge buffer size " << edge_buffer.size());
        size_t inner_triggered = 0;
        //FIXME magic constant
        if (edge_buffer.size() < SMALL_BUFFER_THR) {
            DEBUG("Processing small buffer");
```

```
            utils::perf_counter perf;
            //TODO implement via moves?
            auto edges = AsSmartSet(edge_buffer);
            inner_triggered = BasicProcessBulges(edges);
            DEBUG("Small buffer processed in " << perf.time() << " seconds");
        } else {
            auto bulges = MergeBuffers(FindBulges(edge_buffer));
            auto interacting_edges = RetainIndependentBulges(bulges);
            inner_triggered = ProcessBulges(bulges, interacting_edges);
        }

        proceed |= (inner_triggered > 0);
        triggered += inner_triggered;
        DEBUG("Buffer processed");
    }

    DEBUG("Finished processing. Triggered = " << triggered);
    if (!tracking_)
        it_.Detach();

    return triggered;
}
```

Codice 49: `Run`.

Rimozione dei Tip con *tip_clipper.hpp*. Il file `tip_clipper.hpp` presenta diverse classi che si occupano di verificare se un particolare arco debba essere o meno rimosso.
La classe `RelativeCoverageTipCondition` definisce una condizione per determinare se un arco può essere considerato una tip in base alla copertura relativa rispetto ad altri archi concorrenti. La funzione principale è `MaxCompetitorCoverage`, che trova la copertura massima degli archi concorrenti, cioè quelli che partono o arrivano agli stessi archi: se un arco ha una copertura che è troppo alta rispetto agli altri archi che partono dal suo vertice di inizio o finiscono nel suo vertice di fine, viene ritenuto non una punta e quindi può essere eliminato.

```
double MaxCompetitorCoverage(EdgeId tip, IteratorType begin, IteratorType end
    ) const {
    const Graph &g = this->g();
    double result = 0;
    for (auto it = begin; it != end; ++it) {
        EdgeId e = *it;
        //update if competitor edge is not loop
        if (e != tip && g.EdgeStart(e) != g.EdgeEnd(e))
            result = std::max(result, g.coverage(*it));
    }
    return result;
}
```

Codice 50: `MaxCompetitorCoverage`.

La classe `TipCondition` determina se, dato un arco o un vertice, questi rappresentano una tip. Il metodo principale di questa classe è `IsTip`, che restituisce *true* se, dato un vertice, questo ha un solo arco in ingresso o in uscita, e quindi se è una tip.

```
bool IsTip(VertexId v) const {
```

```
    return this->g().IncomingEdgeCount(v) + this->g().OutgoingEdgeCount(v) ==
     1;
}
```

Codice 51: `IsTip`.

La classe `MismatchTipCondition` verifica se un arco è una tip basandosi su un mismatch (mancanza di corrispondenza) nelle sequenze nucleotidiche tra gli archi concorrenti. I metodi principali sono due: il primo, `Hamming`, calcola la distanza di Hamming (numero di differenze) tra le sequenze di due archi, mentre il secondo, `InnerCheck` controlla se un arco è simile a un altro arco concorrente, in base alla distanza di Hamming e alla lunghezza dell'arco stesso.

```
size_t Hamming(EdgeId edge1, EdgeId edge2) const {
    size_t cnt = 0;
    Sequence seq1 = this->g().EdgeNucls(edge1);
    Sequence seq2 = this->g().EdgeNucls(edge2);
    VERIFY(seq1.size() < seq2.size());
    size_t len = std::min(seq1.size(), seq2.size());
    for (size_t i = this->g().k(); i < len; i++) {
        if (seq1[i] != seq2[i])
            cnt++;
    }
    return cnt;
}

bool InnerCheck(EdgeId e) const {
    size_t len = this->g().length(e);
    for (auto alt : this->g().OutgoingEdges(this->g().EdgeStart(e))) {
        if (e != alt && len < this->g().length(alt)) {
            auto diff_bound = math::ge(max_diff_, 1.) ? max_diff_ : max_diff_
     * double(len);
            if (Hamming(e, alt) <= size_t(math::round(diff_bound)))
                return true;
        }
    }
    return false;
}
```

Codice 52: `Hamming`.

La classe `ATCondition` determina se un arco è una tip in base alla composizione della sequenza. Viene, in particolare, applicata una condizione basata sulla percentuale di adenina e timina presente in un arco. Il metodo principale è `Check`, che controlla se la percentuale di A/T è maggiore di una soglia massima definita dalla variabile `max_AT_percentage_` e lo elimina di conseguenza.

```
bool Check(EdgeId e) const {
    //+1 is a trick to deal with edges of 0 coverage from iterative run
    size_t start = 0;
    size_t end = this->g().length(e) + this->g().k();
    if (check_tip_) {
        if (this->g().OutgoingEdgeCount(this->g().EdgeEnd(e)) == 0)
            start = this->g().k();
        else if (this->g().IncomingEdgeCount(this->g().EdgeStart(e)) == 0)
            end = this->g().length(e);
        else return false;
```

```
    }
    std::array<size_t, 4> counts = std::array<size_t, 4>();
    const Sequence &s_edge = this->g().EdgeNucls(e);

    for (size_t position = start; position < end; position ++) {
        counts[s_edge[position]]++;
    }
    size_t curm = *std::max_element(counts.begin(), counts.end());
    if (math::gr(double(curm), max_AT_percentage_ * double(end - start))) {
        DEBUG("deleting edge" << s_edge.str());;
    DEBUG("curm: " << curm);
        DEBUG("start end cutoff" << start << " " << end << " " <<
    max_AT_percentage_ * double(this->g().length(e)));

        return true;
    } else {
        return false;
    }
}
```

Codice 53: `ATCondition`.

La classe `DeadEndCondition` verifica se un arco rappresenta un "dead end" (punto morto), ossia un vertice che non ha altre uscite o ingressi. Il metodo principale, cioè `IsDeadEnd`, determina se un vertice è un dead end, verificando se questo non ha archi in ingresso o uscita. In caso il vertice sia un punto morto, la funzione restituisce *true*.

```
bool IsDeadEnd(VertexId v) const {
    return this->g().IncomingEdgeCount(v) * this->g().OutgoingEdgeCount(v) ==
     0;
}
```

Codice 54: `DeadEndCondition`.

La classe `IsAllowedCondition` controlla se un arco può essere rimosso, in base a un set di vertici proibiti (forbidden). Anche in questo caso, la logica principale della classe è implementata nella funzione `Check`, che verifica se l'arco dato in input può essere rimosso basandosi sulla lista `forbidden_`, che contiene gli archi proibiti.

```
bool Check(EdgeId e) const {
    return (!forbidden_.count(this->g().EdgeStart(e)) && !forbidden_.count(
    this->g().EdgeEnd(e)));
}
```

Codice 55: `IsAllowedCondition`.

In generale, ogni classe implementa il metodo `Check(EdgeId e)`, che, dato un arco, è in grado di verificare se questo sia una tip, o richiamando la funzione principale evidenziata (nel caso delle classi `RelativeCoverageTipCondition`, `TipCondition`, `MismatchTipCondition`, DeadEndCondition), oppure implementandola direttamente (come nel caso delle classi `ATCondition` e `IsAllowedCondition`).

Rimozione delle Bubble con *superbubble_finder.hpp*. Il file `superbubble_finder.hpp` definisce la classe `SuperbubbleFinder`, che cerca di identificare le bubble a partire da un vertice iniziale (`start_vertex_`) nel grafo. Il processo si ferma se vengono raggiunti i limiti di lunghezza o di numero di nodi (impostati tramite `max_length_` e `max_count_`), oppure se

non è più possibile proseguire la ricerca.
La principale funzione implementata all'interno della classe `SuperbubbleFinder` è `FindSuperbubble`. Inizialmente viene controllato che il numero di archi in uscita di un nodo dato sia maggiore di 2, condizione necessaria affinché possa essere presente una bubble nel grafo.

```
if (g_.OutgoingEdgeCount(start_vertex_) < 2) {
    return false;
}
```

Codice 56: `FindSuperbubble`, controllo iniziale nel grafo.

Successivamente, si va a inizializzare il vertice di partenza (`start_vertex_`). A tutti i vertici, partendo da quello iniziale, viene assegnato un peso e un range. Per il vertice iniziale avremo un peso pari a 0 e un range pari a (0,0). Queste informazioni vengono salvate nella mappa `superbubble_vertices_`.
`heaviest_backtrace_` è un'altra mappa che tiene traccia dell'arco che ha il percorso più pesante fino a ciascun vertice. Per il vertice di partenza, non c'è arco precedente, quindi si assegna un valore vuoto (EdgeId()). Infine `cnt_++`: Incrementa un contatore che tiene traccia del numero di vertici processati.

```
DEBUG("Adding starting vertex " << g_.str(start_vertex_) << " to dominated
    set");
superbubble_vertices_[start_vertex_] = std::make_pair(0, Range(0, 0));
heaviest_backtrace_[start_vertex_] = EdgeId();
cnt_++;
```

Codice 57: `FindSuperbubble`, inizializzazione.

Vengono, poi, inizializzate due variabili per contenere i vertici da processare (`can_be_processed`) e quelli sul confine tra due bubble (`border`), cioè vicini ma non inclusi nel percorso principale.

```
std::unordered_set<VertexId> can_be_processed;
std::unordered_set<VertexId> border;
UpdateCanBeProcessed(start_vertex_, can_be_processed, border);
```

Codice 58: `FindSuperbubble`, gestione dei vertici da processare.

I vertici presenti in `can_be_processed` vengono poi processati: ogni volta che un vertice viene processato, il contatore `cnt_` viene incrementato, fino al raggiungimento di una soglia (per evitare loop infiniti).

```
while (!can_be_processed.empty()) {
    //finish after checks and adding the vertex
    bool final = (border.size() == 1 && can_be_processed.size() == 1);
    DEBUG("Final: " << final);
    if (++cnt_ > max_count_) {
        return false;
    }
    VertexId v = *can_be_processed.begin();
    can_be_processed.erase(can_be_processed.begin());
```

Codice 59: `FindSuperbubble`, ciclo di elaborazione dei vertici.

Per ogni vertice, viene calcolato il range di distanza e il peso massimo degli archi in ingresso. Il range è un intervallo, di cui il primo valore rappresenta il punto di partenza (`start_pos`) ed il secondo la fine del percorso (`end_pos`): la loro differenza rappresenta l'intera lunghezza

del percorso. Il peso è un valore che indica quante volte un k-mer appare nel grafo. Se il peso di un arco è maggiore di quello trovato fino a quel punto, questo viene aggiornato.

```
    DEBUG("Counting distance range for vertex " << g_.str(v));
    size_t min_d = std::numeric_limits<size_t>::max();
    size_t max_d = 0;
    size_t max_w = 0;
    EdgeId entry;

    VERIFY(g_.IncomingEdgeCount(v) > 0);
    VERIFY(CheckCanBeProcessed(v));
    for (EdgeId e : g_.IncomingEdges(v)) {
        //in case of dominated_only == false
        if (superbubble_vertices_.count(g_.EdgeStart(e)) == 0)
            continue;
        size_t weight;
        Range range;
        std::tie(weight, range) = utils::get(superbubble_vertices_, g_.
    EdgeStart(e));
        range.shift((int) g_.length(e));
        DEBUG("Edge " << g_.str(e) << " provide distance range " << range);
        if (range.start_pos < min_d)
            min_d = range.start_pos;
        if (range.end_pos > max_d)
            max_d = range.end_pos;

        weight += size_t(math::round(double(g_.length(e)) * g_.coverage(e)));
        if (weight > max_w) {
            max_w = weight;
            entry = e;
        }
    }
```

Codice 60: `FindSuperbubble`, calcolo del peso e del range.

In seguito, si verifica che il percorso trovato non superi i limiti e che non ci siano condizioni non valide (come archi che tornano al vertice di partenza).

```
    VERIFY((max_d > 0) && (min_d < std::numeric_limits<size_t>::max()) && (
    min_d <= max_d));
    DEBUG("Range " << Range(min_d, max_d));
    Range r(min_d, max_d);
    \\return std::make_pair(std::make_pair(max_w, entry), Range(min_d, max_d)
    );
    if (r.start_pos > max_length_) {
        return false;
    }
    //Inner vertices cannot have edge to start vertex
    //TODO Also all added edges have to have an outgoing edge
    if (!final && (!CheckNoEdgeToStart(v) || g_.OutgoingEdgeCount(v) == 0)) {
        return false;
    }
```

Codice 61: `FindSuperbubble`, verifica dei limiti di lunghezza.

Infine, il vertice viene aggiunto alla mappa `superbubble_vertices_` in elaborazione, con il peso massimo ed il range calcolato. Inoltre, si aggiorna l'arco che ha il percorso più pesante con l'arco corrente. Se il vertice è l'ultimo della bubble (`final`), viene impostato

come *end_vertex_* e la funzione restituisce *true*, indicando che una bubble è stata trovata con successo. Altrimenti, il processo continua aggiornando i vertici da processare. Se la funzione esce dal ciclo senza aver trovato una bubble valida, restituisce *false*, indicando che la ricerca non ha avuto successo.

```
    DEBUG("Adding vertex " << g_.str(v) << " to dominated set");
    superbubble_vertices_[v] = std::make_pair(max_w, r);
    heaviest_backtrace_[v] = entry;
    border.erase(v);
    if (final) {
        end_vertex_ = v;
        return true;
    } else {
        UpdateCanBeProcessed(v, can_be_processed, border);
    }
}
DEBUG("Finished search for starting vertex " << g_.str(start_vertex_));
return false;
}
```

Codice 62: `FindSuperbubble`, aggiornamento della bubble.

4 Assemblaggio dei Plasmidi

I plasmidi sono elementi genetici extra-cromosomici che si replicano in modo indipendente dal cromosoma principale della cellula. Possono contenere geni importanti per la virulenza e la resistenza agli antibiotici, e sono strumenti fondamentali per l'ingegneria genetica. L'identificazione dei plasmidi nei dati di sequenziamento dell'intero genoma è complessa, poiché spesso non vengono distinti dai frammenti genomici durante l'assemblaggio. Per questa ragione, *plasmidSPAdes* [10] viene proposto per ricostruire automaticamente i plasmidi a partire da dati dell'intero genoma (*Whole Genome Sequencing*, o WGS), anche quando i plasmidi non sono stati isolati prima del sequenziamento.

4.1 Come PlasmidSpades Separa i Plasmidi dal Genoma

PlasmidSPAdes utilizza la coverage come mezzo per distinguere tra i plasmidi ed i cromosomi: mentre nelle read Illumina la coverage si presenta molto uniforme, nei plasmidi può risultare variabile, a seconda di alcuni fattori, come ad esempio il numero di copie presenti nella cellula, oppure il numero di cellule nel campione che possiedono quel plasmide.
Ma la semplice copertura media non può essere utilizzata per distinguere i plasmidi dai cromosomi, perché i plasmidi presenti in copie numerose falsano quel valore. Per questa ragione, viene calcolata la **copertura mediana** ma utilizzando solo i contig più lunghi (10000 bp o più) per due ragioni: in questo modo vengono evitati i ripetuti (che sono in genere brevi); inoltre i contig lunghi hanno meno variazione (coverage più stabile).

$$\text{medianCoverage} = \max\left\{ c \;\middle|\; \sum_{\substack{e \in \text{LongEdges} \\ \text{coverage}(e) \geq c}} \text{length}(e) \geq \frac{1}{2} \sum_{e \in \text{LongEdges}} \text{length}(e) \right\}$$

Per il calcolo della copertura mediana, si guarda il grafico di assemblaggio costruito da SPAdes (basato sul grafo di De Bruijn), si prende l'insieme di tutti i contig lunghi (oltre 10 kb) ed, in seguito, si trova il valore di copertura per cui i contig con quella copertura o superiore coprono almeno metà della lunghezza totale dei contig lunghi.
Per classificare un contig come cromosomico, questo deve rientrare all'interno del seguente intervallo:

$$1 - \text{maxDeviation} < \frac{\text{Coverage(e)}}{medianCoverage} < 1 + \text{maxDeviation}$$

Dove *maxDeviation* in genere è fissato a 0.3.

4.2 Algoritmo di PlasmidSPAdes

L'algoritmo di PlasmidSPAdes segue quello di SPAdes fino alla costruzione dell'assembly graph, ma con la creazione finale di un sottografo, anche definito *plasmid graph* che distingue e genera contig plasmidici. L'algoritmo si sviluppa nel seguente modo:

1. Costruzione del grafo di De Bruijn, utilizzando i k-mer presi dalle read a partire dai dati di sequenziamento;

2. Calcolo della copertura mediana;

3. Fase di pulizia del grafo, che si occupa di:
    - rimuovere i contig cromosomici lunghi e le dead end (poiché un plasmide ha una struttura circolare, non può terminare improvvisamente);
    - trasformare ogni percorso non ramificato in un singolo arco, semplificando ulteriormente il grafo.
4. Rimozione di tutte le componenti non plasmidiche dal grafo;
5. Gestione dei ripetuti nel grafo plasmidico con il tool *exSPAnder*;
6. Creazione del grafo plasmidico.

PlasmidSPAdes dovrebbe idealmente assemblare ogni plasmide in un contig separato, ma ciò non avviene sempre a causa delle lunghe ripetizioni nei plasmidi e della lunghezza che eccede quella stabilita, andando quindi a separare un singolo plasmide.

5 Strumenti di Analisi

5.1 FASTQC

FASTQC è un software che mira ad eseguire controlli di qualità sui dati grezzi provenienti dal sequenziamento. Genera un report finale che sintetizza alcune delle principali caratteristiche dei dati, tra cui:

- **Basic Statistics**: fornisce un riepilogo delle principali caratteristiche del file analizzato, riportando ad esempio informazioni sul nome, sul formato, sul numero e sulla lunghezza delle sequenze, e sulla percentuale di G e C nel dataset.
- **Per Base Sequence Quality**: mostra il variare della qualità del sequenziamento al variare della posizione della read. I dati sono rappresentati attraverso un diagramma a scatola e baffi, di cui la mediana è indicata da una linea rossa, l'intervallo interquartile da un rettangolo giallo, mentre i baffi segnano il 10° e 90° percentile. Una linea blu rappresenta la media della qualità, e lo sfondo colorato indica se le basi hanno qualità buona (verde), accettabile (arancione) o scarsa (rossa). La qualità tende normalmente a diminuire verso la fine delle letture.
- **Per Sequence Quality Score**: aiuta a identificare se alcune sequenze hanno qualità sistematicamente bassa, indice di perdita di qualità durante il sequenziamento. Per lunghi run (cicli singoli di sequenziamento) la rimozione delle basi di bassa qualità, che prende il nome di **trimming**, può risolvere il problema.
- **Per Base Sequence Content**: grafico che mostra la proporzione di ogni base (A, T, G, C) per ciascuna posizione nelle read. Generalmente ci si aspetta che le proporzioni di ciascuna base siano simili alle altre. Tuttavia, esistono alcuni fenomeni, come sequenze sovra rappresentate o trimming aggressivo degli adattatori, che possono alterare tali proporzioni.
- **Per Sequence GC Content**: misura il contenuto di guanina e citosina lungo l'intera sequenza e confronta la loro distribuzione con una distribuzione normale. Se la distribuzione è insolita o spostata, la libreria potrebbe essere contaminata o un sottogruppo potrebbe avere un bias di sequenziamento.
- **Per Base N Content**: indica la percentuale di "N" inseriti durante il sequenziamento, dove "N" indica un'incertezza del sequenziatore: esso viene inserito quando il sequenziatore non è sicuro di quale base inserire in una posizione data. Ci si aspetta un leggero incremento di "N" verso la fine della sequenza, ma un significativo incremento potrebbe indicare una bassa qualità complessiva.
- **Sequence Length Distribution**: mostra la distribuzione delle lunghezze dei frammenti di sequenza nel file analizzato. Alcuni sequenziatori generano frammenti di lunghezza uniforme, mentre altri producono letture di lunghezze molto variabili. Anche nelle librerie con lunghezza uniforme, alcune pipeline possono effettuare il trimming delle sequenze per rimuovere le basi di bassa qualità dalla fine.
- **Duplicate Sequences**: mostra la distribuzione delle sequenze con diversi livelli di duplicazione. Le sequenze identiche vengono tracciate e, per ridurre i requisiti di

memoria, solo le prime 100.000 sequenze vengono analizzate, mentre le sequenze con più di 10 duplicati vengono raggruppate in intervalli. Nel grafico sono presenti due linee, una linea blu, che mostra la distribuzione dei livelli di duplicazione per l'intero set di sequenze, includendo tutte le duplicazioni, e una linea rossa, che mostra la distribuzione dei livelli di duplicazione dopo che le sequenze sono state deduplicate (ovvero a seguito di rimozione delle duplicazioni), evidenziando la proporzione delle sequenze uniche rispetto ai livelli di duplicazione originali.

- **Overrepresented Sequences**: elenca tutte le sequenze che rappresentano più dello 0,1% del totale. Poiché solo le sequenze che appaiono nelle prime 100.000 sequenze vengono tracciate fino alla fine del file, è possibile che una sequenza sovra-rappresentata che non appare all'inizio del file venga persa. La presenza di sequenze sovra-rappresentate può indicare contaminazione da adattatori, primer o altre sequenze non target.

- **Adapter Content**: esamina la proporzione di letture che contengono specifici k-mer di adattatori, mostrando un grafico cumulativo della loro presenza nelle letture. Le librerie con inserti più corti rispetto alla lunghezza delle letture potrebbero necessitare di un trimming degli adattatori prima di ulteriori analisi.

5.2 BUSCO

BUSCO, acronimo di *Benchmarking Universal Single-Copy Orthologs* [11], è una metodologia proposta per valutare la qualità di un assemblaggio. Mentre metriche come l'N50 possono risultare utili, non valutano appieno quanto un assemblaggio rappresenti effettivamente i geni in un genoma. BUSCO, invece, utilizza specifici geni, i geni ortologhi, per colmare questa lacuna: questi sono geni che si trovano in singola copia nella maggior parte delle specie. In particolare, BUSCO:

- Seleziona dei geni ortologhi: questi geni vengono selezionati in quanto essenziali per la funzione cellulare o evolutivamente stabili, e dovrebbero essere presenti in tutte le specie appartenenti ad un determinato gruppo;

- Definisce un set di geni ortologhi in singola copia: questi geni verranno confrontati con i genomi assemblati;

- Ricerca i geni ortologhi nell'assemblaggio: verifica cioè se il genoma assemblato presenta i geni ortologhi, se sono completi, parzialmente frammentati o assenti.

- Classifica i geni in base ai risultati dell'analisi.

5.2.1 Analisi e interpretazione dei risultati

I risultati della valutazione della completezza tramite BUSCO hanno senso solo se contestualizzati nella biologia dell'organismo: la mancanza o la duplicazione dei geni potrebbero non essere dovute a un errore nell'assemblaggio (e quindi di natura tecnica), ma potrebbero invece dipendere da un evento di natura biologica, come una recente duplicazione genomica. BUSCO divide i risultati in quattro categorie distinte:

- **Completo**: Se un gene è classificato come *Completo*, allora ha una corrispondenza che soddisfa i criteri di punteggio e la lunghezza allineata prevista.
- **Frammentato**: Se un gene è classificato come *Frammentato*, allora la corrispondenza soddisfa i punteggi attesi ma non la lunghezza prevista dal profilo BUSCO. Ciò significa che i modelli di geni sono incompleti, il che potrebbe significare che il gene è parzialmente presente o che i passaggi di previsione del gene non hanno prodotto un modello completo, anche se il gene potrebbe essere presente nel genoma.
- **Mancante**: Se un gene è classificato come *Mancante* potrebbero non essere stati identificati match significativi, oppure il punteggio è al di sotto della soglia prevista dal profilo BUSCO. Ciò significa che i geni ortologhi potrebbero essere effettivamente mancanti o che il processo di ricerca non è riuscito ad identificarli.

5.3 Trimmomatic

Trimmomatic [12] rappresenta uno dei principali tool per l'esecuzione delle operazioni di trimming, processo di rimozione delle read di bassa qualità, o di sequenze indesiderate, come ad esempio gli adattatori, al fine di migliorare la qualità e ridurre il rischio di errori in fase di assemblaggio. Esistono diverse strategie di trimming:

- **Trimming della qualità**: tenendo in considerazione i valori di Phred legati alle sequenze nel formato FASTQ, questa operazione permette di eliminare le sequenze alle estremità delle read (dove in genere la qualità si abbassa, soprattutto nelle read più lunghe) nel caso in cui la loro qualità scenda al di sotto di una certa soglia;
- **Rimozione di adattatori**: è comune utilizzare adattatori alle estremità dei filamenti in fase di sequenziamento. Tuttavia gli adattatori non appartengono al genoma di riferimento e quindi è possibile eliminarle;
- **Trimming di lunghezza minima**: a seguito delle altre operazioni di trimming, la lunghezza delle read può essere compromessa. Vengono eliminate le read che non superano una lunghezza minima.

6 Caso d'Uso: DNA di E. Coli

Verrà di seguito riportato un caso di utilizzo di SPAdes applicato a dei dati di sequenziamento Illumina ottenuti a partire da un campione di Escherichia coli.

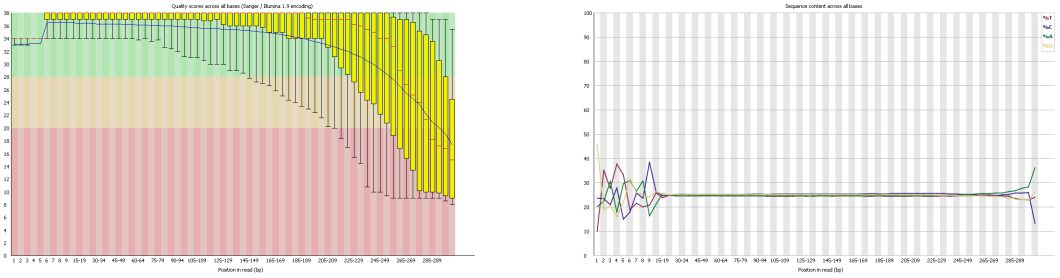
6.1 Escherichia Coli

L'Escherichia coli è la principale flora facoltativa non patogena dell'intestino umano [13]. Tuttavia, alcuni ceppi di E. coli hanno sviluppato la capacità di causare malattie a carico del sistema gastrointestinale, urinario o nervoso centrale anche negli ospiti umani più sani.

6.1.1 Dati di Sequenziamento e Controllo della Qualità

I dati di sequenziamento dell'intero genoma di E. coli sono stati ottenuti dalla *National Library of Medicine* [14]. Il sequenziamento è stato eseguito nell'ambito del programma *National Antimicrobial Resistance Monitoring System* (NARMS), che utilizza l'analisi genomica di patogeni di origine alimentare, inclusi i ceppi di E. coli, per monitorare la resistenza antimicrobica e identificare potenziali minacce alla salute pubblica. Il dataset utilizzato per l'assemblaggio è stato ottenuto da un sequenziamento Illumina e corrisponde al file: **SRR32812502.fastq**. Il file presenta i filamenti *forward* e *reverse* in modo alterno.
Il controllo della qualità delle sequenze è stato effettuato tramite il software *FASTQC*. Nel dataset sono state rilevate 2.584.822 sequenze, per un totale di 570,7 Mpb. Tutte le sequenze presentano una lunghezza compresa tra le 35 e le 301 basi. La percentuale di GC è del 50%.

Moduli con lo stato di *Failure*. Durante l'esecuzione di FASTQC, due moduli hanno riportato lo stato di *Failure* (fig 3). Il primo modulo è *Per base sequence quality*, che segnala una notevole diminuzione nella qualità delle read più lunghe: dopo le 150 bp si osserva un progressivo calo della qualità, che culmina dopo le 200-250 bp, indicando una possibile necessità di trimming. Il secondo modulo è *Per base sequence content*, che indica un'alta variabilità nella composizione delle basi nelle prime posizioni (1-12 bp). Tuttavia, come indicato anche dalla documentazione ufficiale, un'alta variabilità iniziale suggerisce un bias di frammentazione comune, che quindi non compromette le analisi. La concentrazione delle basi si mantiene poi stabile fino alla posizione 280, indice di una perdita di informazione nelle read più lunghe. Ciò conferma la necessità di trimming.

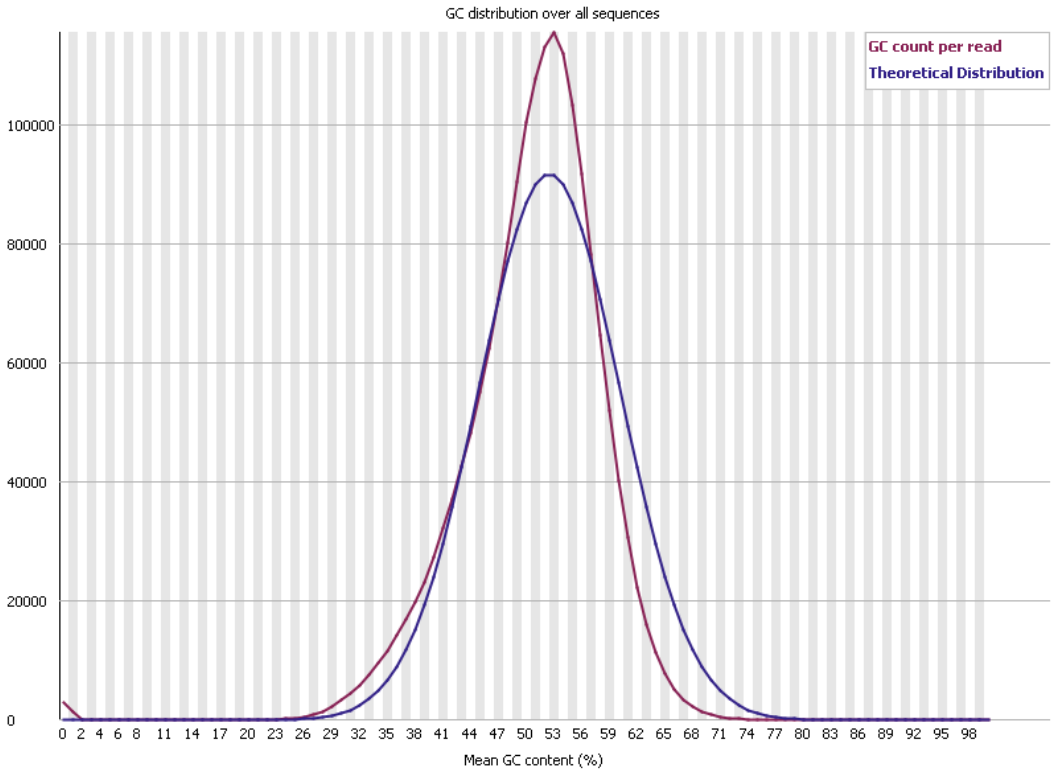


(a) Per base sequence quality

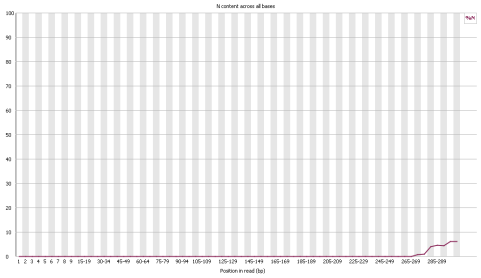
(b) Per base sequence content

Figura 3: Moduli di FASTQC che hanno riportato lo stato di *Failure*.

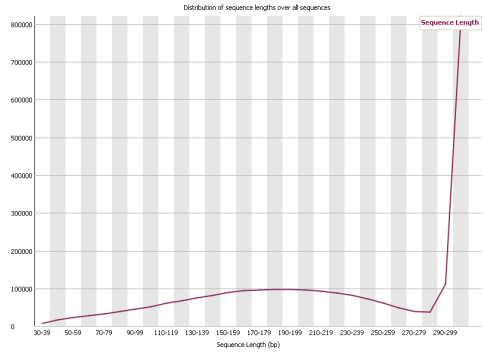
Moduli con lo stato di *Warning*. *FASTQC* ha riportato tre moduli con lo stato di *Warning* (fig 4). Un Warning in *Per sequence GC content*: la distribuzione di GC assume una forma simile a una normale, ma con un picco intorno al 50%, mostrando quindi una leggera discrepanza con la distribuzione teorica. Questo indica che la somma delle deviazioni supera il 15% delle letture, indicando un potenziale bias o contaminazione. Il secondo modulo che presenta un Warning è *Per base N content*, che presenta un incremento delle basi N intorno alle 270 bp, in linea con l'abbassamento della qualità del sequenziamento verso la fine della read. L'ultimo modulo è *Sequence length distribution*, che mostra uno sbilanciamento del dataset verso le read più lunghe, con una dimensione maggiore di 280 bp.



(a) Per sequence GC content



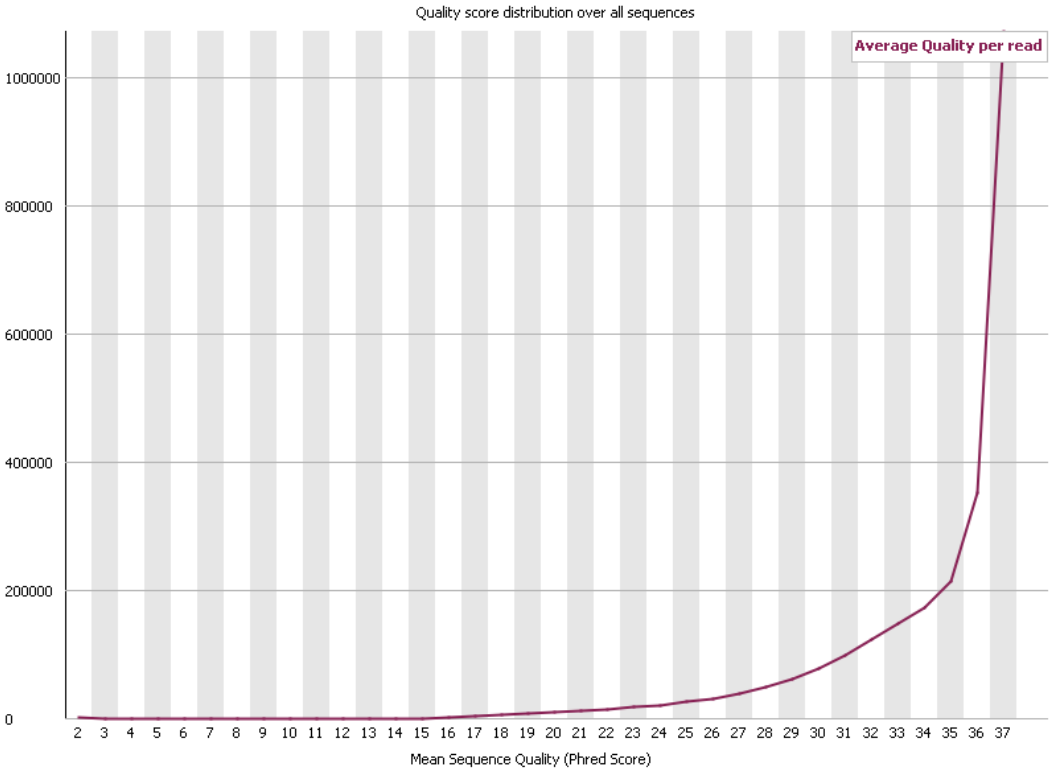
(b) Per base N content



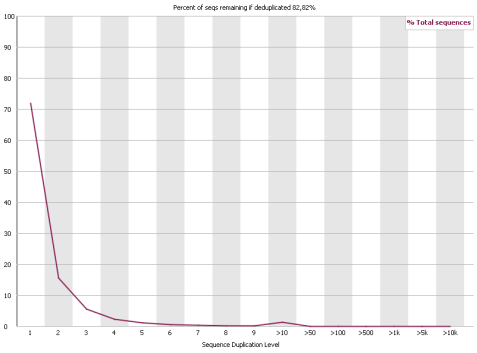
(c) Sequence length distribution

Figura 4: Moduli di FASTQC che hanno riportato lo stato di *Warning*.

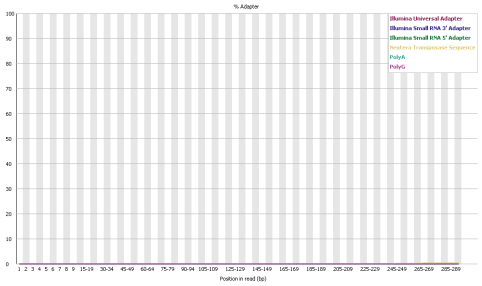
Altri moduli. Gli altri moduli di *FASTQC* non hanno rilevato anomalie; il modulo *Per sequence quality score* indica che, in generale, le sequenze hanno un'ottima qualità; il modulo *Sequence duplication levels* indica un'alta coverage ottenuta con il sequenziamento; il modulo *Overrepresented sequences* mostra l'assenza di sequenze sovrarappresentate; infine, il modulo *Adapter content* indica che la presenza di adattatori nelle letture di sequenziamento è bassa o assente, il che è un buon segnale di qualità della libreria.



(a) Per sequence quality score



(b) Sequence duplication levels



(c) Adapter content

Figura 5: Moduli di FASTQC che hanno riportato lo stato di *Warning*.

6.2 Esecuzione di SPAdes e Valutazione dei Risultati Ottenuti

L'esecuzione di SPAdes sui dati di sequenziamento Illumina **SRR32812502.fastq** è stata preceduta da una fase di pre-processing che ha portato anche alla separazione dei filamenti forward e reverse in due file distinti **forward.fasta** e **reverse.fasta** tramite il codice, eseguito da linea di comando:

```
awk 'NR%8==1 || NR%8==2 || NR%8==3 || NR%8==4 {print > "forward.fastq"} NR
    %8==5 || NR%8==6 || NR%8==7 || NR%8==0 {print > "reverse.fastq"}'
    SRR32812502.fastq
```

Poiché ogni read è composta da quattro righe, in accordo con la struttura del formato FASTQ, e poiché ad ogni read del filamento forward ne segue una del filamento reverse, l'esecuzione del codice precedente ha permesso di separare facilmente i due filamenti nei rispettivi file. Ciò ha reso possibile una corretta esecuzione di SPAdes sulle paired-end read tramite la riga di comando:

```
./<path>/SPAdes-4.1.0-Linux/bin/spades.py -1 forward.fastq -2 reverse.fastq -
    o <path>/spades_output
```

Poiché dall'analisi condotta con FASTQC sono emerse diverse "Failure" e "Warning" riguardo alla qualità del sequenziamento, SPAdes è stato eseguito anche in modalità "Correzione degli errori", che utilizza di default lo strumento BayesHammer [7], specifico per i dati di sequenziamento Illumina, per eseguire operazioni di pulizia, come il trimming delle basi.

6.2.1 Creazione dell'ambiente virtuale ed esecuzione di BUSCO

Per eseguire BUSCO, necessario per la valutazione dei risultati dell'assemblaggio ottenuti con SPAdes, è stato utilizzato un ambiente virtuale Python. In particolare, è stata scelta la versione 3.10 per garantire la compatibilità con le librerie richieste.
Per semplificare la creazione e la gestione dell'ambiente virtuale, è stata installata **Anaconda** (la documentazione ufficiale di BUSCO [15] suggerisce qualsiasi versione successiva alla 4.8.4). Per una gestione più rapida degli ambienti e l'installazione dei pacchetti è stato installato **mamba** (versione 2.0.8) dal canale **conda-forge**, che contiene numerosi pacchetti comunitari aggiornati di frequente, attraverso la riga di comando:

```
conda install -c conda-forge mamba
```

La versione 5.8.2 di BUSCO è stata invece installata dal canale **bioconda**, che è specializzato in software per la bioinformatica, attraverso la riga:

```
mamba install -c conda-forge -c bioconda busco=5.8.2
```

Ottenute tutte le componenti necessarie, è stato possibile procedere alla creazione e attivazione dell'ambiente **busco_env**:

```
mamba create --name busco_env python=3.10
conda activate busco_env
```

Dopo aver configurato l'ambiente con tutti gli strumenti necessari, è adesso possibile eseguire BUSCO sui dati prodotti da SPAdes a seguito dell'assemblaggio. In particolare, verranno analizzati i dati riportati nel file `contigs.fasta`:

```
busco -m genome -i /<path>/spades_output/contigs.fasta -o /<path>/
    BUSCO_output
```

6.2.2 Valutazione dei risultati

I risultati ottenuti (tab 1) vengono riassunti da BUSCO nei file:

- **short_summary.generic.bacteria_odb12.BUSCO_output.txt**;
- **short_summary.generic.bacteria_odb12.BUSCO_output.txt**.

È possibile osservare un'elevata completezza del genoma, con un 99.1% di completezza totale, il che suggerisce che vengono conservati la maggior parte dei geni presenti nei batteri. Tutti i BUSCO completi sono a singola copia: l'assenza di copie multiple indica che non vi sono duplicazioni anomale nei geni conservati. Inoltre, un solo BUSCO su 116 risulta mancante, e nessun BUSCO risulta frammentato. In totale, il genoma risulta lungo 5,169,832 bp, in linea con la dimensione di molti genomi batterici. Il tutto indica un'ottima qualità ottenuta dall'assemblaggio.

Parametro	**Valore**
Dataset di riferimento	bacteria_odb12
Completezza totale (%)	99.1
BUSCO completi	115
BUSCO a copia singola (%)	99.1
BUSCO a copia singola	115
BUSCO a copie multiple (%)	0
BUSCO a copie multiple	0
BUSCO frammentati (%)	0
BUSCO frammentati	0
BUSCO mancanti (%)	0.9
BUSCO mancanti	1
Numero di marker totali	116
Identità media	null
Dominio	prokaryota
Conteggio codoni di stop interni	0
Percentuale codoni di stop interni	0
Numero di scaffolds	232
Numero di contigs	232
Lunghezza totale	5169832
Scaffold N50	80627
Contig N50	80627

Tabella 1: Riassunto dei risultati dell'analisi BUSCO.

7 Confronto dei Risultati di SPAdes

Per validare l'efficacia di SPAdes rispetto ai risultati ottenuti nei capitoli precedenti, è utile confrontarlo con un altro software di assemblaggio, Velvet. L'esecuzione di Velvet sarà effettuata tramite Galaxy EU, mentre la qualità dell'assemblaggio verrà valutata utilizzando nuovamente BUSCO.

7.1 Velvet

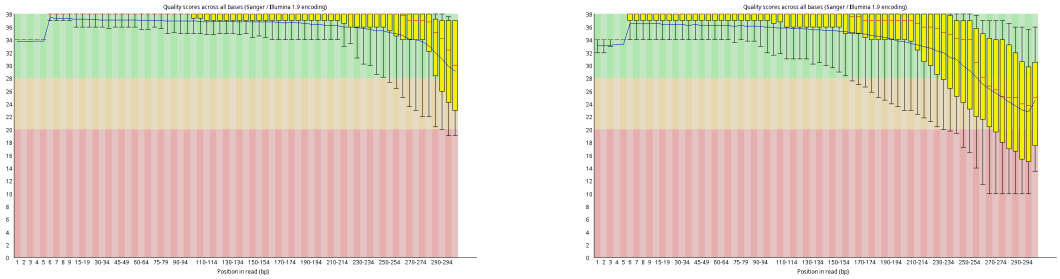
Velvet è uno dei software più utilizzati per l'assemblaggio delle reads [16]. Come SPAdes, Velvet utilizza i grafi de Bruijn, superando le soluzioni tradizionali basate su grafi di sovrapposizione, che comporterebbero un'eccessiva complessità computazionale e maggiore ambiguità nei dati ripetitivi. Tuttavia, la principale differenza tra Velvet e SPAdes risiede nella scelta della lunghezza dei k-mer: mentre SPAdes utilizza grafi de Bruijn multidimensionali, combinando le informazioni ottenute da iterazioni dell'algoritmo con diversi valori di k, Velvet richiede invece l'inserimento manuale del valore di k prima dell'esecuzione dell'algoritmo. Di conseguenza, per determinare il valore ottimale di k, è necessario eseguire l'algoritmo più volte e scegliere il valore che massimizza la qualità dell'assemblaggio. Inoltre, Velvet ha un limite di dimensione per i k-mer pari a 31.

7.1.1 Esecuzione di Velvet

Poiché, a differenza di SPAdes, Velvet non possiede strumenti di eliminazione degli errori, è stato necessario effettuare una fase di trimming preliminare attraverso il tool *Trimmomatic* (versione 0.39) sui due file `forward.fastq` e `reverse.fastq`, che ha portato alla creazione di quattro file di output:

- `output_forward_paired.fastq` e `output_reverse_paired.fastq`, che contengono le letture forward e reverse pulite e accoppiate;
- `output_forward_unpaired.fastq` e `output_reverse_unpaired.fastq`, che contengono le letture pulite, ma senza la controparte corrispondente sul filamento opposto.

A seguito dell'operazione di trimming, ne è stata valutata la bontà attraverso il software FASTQC: in generale, possiamo osservare un miglioramento nel modulo *Per base sequence quality* sia per la sequenza forward, che ha risolto lo stato di *Failure*, sia per la sequenza reverse, che è passata dallo stato di *Failure* a quello di *Warning*.



(a) Forward (b) Reverse

Figura 6: Modulo *Per base sequence quality* per le sequenze forward e reverse.

Avendo ripulito le sequenze forward e reverse, è possibile procedere con l'esecuzione di Velvet (di cui è stata utilizzata la versione 1.2.10), e la valutazione successiva dei risultati. Velvet offre due moduli principali per l'assemblaggio: `velveth`, che permette la divisione dei dati nei k-mer della lunghezza scelta, e `velvetg`, che, prendendo in input i dati prodotti da `velveth`, si occupa dell'assemblaggio dei k-mer in contigs. Poiché Velvet richiede l'inserimento manuale di un valore di k, e non è in grado di combinare automaticamente le informazioni ottenute da diversi valori di k per ottenere un risultato ottimale, è stato deciso di eseguire Velvet utilizzando più valori di k, nello specifico 21 e 31 (considerando il limite dimensionale imposto dal software).
Per velocizzare il workflow, il processo è stato automatizzato tramite lo script `run_velvet_busco.sh` (Codice 63), che esegue in sequenza i comandi `velveth`, `velvetg` e `busco`.

```
# Controlla che un numero k sia stato fornito come parametro
if [ $# -eq 0 ]; then
    echo "Fornire un valore valido per k"
    exit 1
fi

# Prendi il parametro k dalla linea di comando
KMER=$1

# Esegui il comando velveth
echo "Eseguo velveth con k = $KMER"
velveth velveth$KMER $KMER -shortPaired -fastq -separate /<path>/
    output_forward_paired.fastq /<path>/output_reverse_paired.fastq

# Esegui il comando velvetg
echo "Eseguendo velvetg con k = $KMER"
velvetg velveth$KMER -exp_cov auto -cov_cutoff auto

# Esegui il comando BUSCO
echo "Eseguendo BUSCO per contigs.fa con k = $KMER"
busco -m genome -i /<path>/velvet-1.2.10/velveth$KMER/contigs.fa -o /results/
    BUSCO_output_velveth$KMER

echo "Completato il processo per k = $KMER"
```

Codice 63: run_velvet_busco.sh

7.2 Confronto tra SPAdes e Velvet

Dopo aver eseguito BUSCO sui dati di output prodotti da velvetg, è possibile confrontare i risultati ottenuti con quelli di SPAdes. L'analisi BUSCO (tab 2) mostra risultati molto simili per SPAdes, Velvet_21 e Velvet_31 in termini di completezza totale (99.1%) e BUSCO completi (115 su 116), con il 100% dei marker presenti come copia singola e senza segnalare copie multiple o frammentazioni. La percentuale di BUSCO mancanti è altrettanto bassa, pari a 0.9% in tutti e tre gli assemblaggi.
Le principali differenze tra gli assemblaggi emergono dai parametri di assemblaggio: in generale, SPAdes ha prodotto un numero significativamente inferiore di contigs e scaffolds rispetto alle iterazioni di Velvet. Questo suggerisce che Velvet tende a frammentare maggiormente l'assemblaggio, sebbene tale comportamento possa essere influenzato dal limite

imposto sulla dimensione dei k-mer. Inoltre, la lunghezza totale degli assemblaggi ottenuti con Velvet è leggermente inferiore a quella di SPAdes, confermando che Velvet ha generato contigs mediamente più corti. Nonostante ciò, la completezza degli assemblaggi di Velvet rimane simile a quella di SPAdes, indicando che la frammentazione non ha compromesso significativamente la qualità complessiva dell'assemblaggio.

Parametro	**SPAdes**	**Velvet_21**	**Velvet_31**
Dataset di riferimento	bacteria_odb12	bacteria_odb12	bacteria_odb12
Completezza totale (%)	99.1	99.1	99.1
BUSCO completi	115	115	115
BUSCO a copia singola (%)	99.1	99.1	99.1
BUSCO a copia singola	115	115	115
BUSCO a copie multiple (%)	0	0	0
BUSCO a copie multiple	0	0	0
BUSCO frammentati (%)	0	0	0
BUSCO frammentati	0	0	0
BUSCO mancanti (%)	0.9	0.9	0.9
BUSCO mancanti	1	1	1
Numero di marker totali	116	116	116
Dominio	prokaryota	prokaryota	prokaryota
Conteggio codoni di stop interni	0	0	0
Percentuale codoni di stop interni	0	0	0
Numero di scaffolds	**232**	1443	882
Numero di contigs	**232**	1534	948
Lunghezza totale	**5169832**	5046871	5080707
Scaffold N50	80627	31157	41220
Contig N50	80627	20897	30772

Tabella 2: Riassunto dei risultati dell'analisi BUSCO e confronto con SPAdes.

8 Conclusioni

Dall'analisi dei risultati ottenuti, è possibile affermare che SPAdes riesca a produrre un assemblaggio di ottima qualità, sia a livello di completezza, raggiungendo il 99.1%, sia in termini di frammentazione, andando a generare un numero di contigs e scaffolds di gran lunga minore rispetto a Velvet. Quest'ultimo fattore risulta particolarmente rilevante, perché un assemblaggio più coerente e meno disperso facilita l'interpretazione biologica dei risultati. Tuttavia, non possiamo trascurare la differenza tra i costi computazionali dei due algoritmi: SPAdes tende a richiedere risorse di memoria e tempo di esecuzione superiori rispetto a Velvet, il che potrebbe rappresentare un problema nell'analisi di dataset particolarmente grandi o complessi.

8.1 Sviluppi Futuri

Un'analisi futura di SPAdes potrebbe concentrarsi sulla valutazione della qualità dell'assemblaggio su genomi più complessi, come quelli di organismi eucariotici, o su dataset metagenomici, caratterizzati da un'elevata eterogeneità. Ciò permetterebbe anche di sfruttare i moduli più recenti e avanzati integrati in SPAdes.
Inoltre, la larga diffusione di modelli di machine learning potrebbe offrire nuove opportunità anche per SPAdes: un possibile sviluppo potrebbe riguardare l'implementazione di algoritmi di apprendimento per la selezione automatica dei parametri di assemblaggio, come la dimensione dei k-mer, andando a velocizzare l'assemblaggio e migliorare l'efficienza nell'uso delle risorse computazionali, rendendo SPAdes ancora più performante.

Riferimenti bibliografici

[1] Xiaoning Tang et al. «The single-cell sequencing: new developments and medical applications». In: *Cell & bioscience* 9 (2019), pp. 1–9.

[2] Sam Behjati e Patrick S Tarpey. «What is next generation sequencing?» In: (2013).

[3] Illumina. *Next-Generation Sequencing Technology*. 2025. URL: `https://www.illumina.com/science/technology/next-generation-sequencing/sequencing-technology.html`.

[4] Peter JA Cock et al. «The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants». In: *Nucleic acids research* 38.6 (2010), pp. 1767–1771.

[5] Anton Bankevich et al. «SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing». In: *Journal of computational biology* 19.5 (2012), pp. 455–477.

[6] Andrey Prjibelski et al. «Using SPAdes de novo assembler». In: *Current protocols in bioinformatics* 70.1 (2020), e102.

[7] Sergey I Nikolenko, Anton I Korobeynikov e Max A Alekseyev. «BayesHammer: Bayesian clustering for error correction in single-cell sequencing». In: *BMC genomics* 14.Suppl 1 (2013), S7.

[8] Vasily Ershov et al. «IonHammer: Homopolymer-space hamming clustering for IonTorrent read error correction». In: *Journal of Computational Biology* 26.2 (2019), pp. 124–127.

[9] SPAdes Development Team. *SPAdes Genome Assembler*. Accessed: 29 March 2025. 2024. URL: `https://github.com/ablab/spades`.

[10] Dmitry Antipov et al. «plasmidSPAdes: assembling plasmids from whole genome sequencing data». In: *Bioinformatics* 32.22 (2016), pp. 3380–3387.

[11] Felipe A Simão et al. «BUSCO: assessing genome assembly and annotation completeness with single-copy orthologs». In: *Bioinformatics* 31.19 (2015), pp. 3210–3212.

[12] Anthony M Bolger, Marc Lohse e Bjoern Usadel. «Trimmomatic: a flexible trimmer for Illumina sequence data». In: *Bioinformatics* 30.15 (2014), pp. 2114–2120.

[13] James P Nataro e James B Kaper. «Diarrheagenic escherichia coli». In: *Clinical microbiology reviews* 11.1 (1998), pp. 142–201.

[14] National Center for Biotechnology Information. *NCBI Sequence Read Archive - SRR32812502*. Accessed: 31 Mar. 2025. 2025. URL: `https://trace.ncbi.nlm.nih.gov/Traces/?view=run_browser&page_size=10&acc=SRR32812502&display=metadata`.

[15] Simão, F. A. et al. *BUSCO User Guide*. Accessed: 2025-04-01. 2024. URL: `https://busco.ezlab.org/busco_userguide.html#installation-with-conda`.

[16] Daniel R Zerbino e Ewan Birney. «Velvet: algorithms for de novo short read assembly using de Bruijn graphs». In: *Genome research* 18.5 (2008), pp. 821–829.