UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO D'INFORMATICA

Strumenti formali per la bioinformatica

ABySS 2.0

Professori:
De Felice Clelia
Zaccagnino Rocco
Zizza Rosalba

Studente:
Strianese Davide Benedetto Matr.0522501458

Anno Accademico: 2023/2024

1 Introduzione

In questa sezione verrà descritto, a grandi linee, in cosa consiste il problema del **de novo assembly** insieme ad uno dei primi modelli utilizzati per risolverlo, ovvero **ABySS**.

1.1 De Novo Assembly

Il **de novo assembly** è una tecnica utilizzata in bioinformatica per ricostruire un genoma a partire da frammenti di sequenze di DNA.
A differenza del **reference-based assembly**, che utilizza un genoma di riferimento come base di partenza per l'assemblaggio, nel de novo assembly non si è a conoscenza a priori della sequenza o dell'ordine corretto dei frammenti che compongono il genoma. Questi frammenti prendono il nome di **reads** e possono essere classificati in due categorie a seconda di quanti ne vengono raccolti ad ogni lettura:

- **short-reads**: il range dei campioni letti è tra i 35 bp e i 1000 bp
- **long-reads**: il range dei campioni letti è tra i 1000 bp e i 500000 bp

Dove bp indica il base pair.
La lunghezza della lettura dipende dal modello di sequenziatore e dal protocollo di preparazione della libreria e queste letture sono generalmente memorizzate in file **FastQ**.
Questo processo è fondamentale per lo studio del genoma di organismi che non sono stati precedentemente sequenziati o per i quali non esiste un genoma di riferimento disponibile. Grazie al de novo assembly, è possibile acquisire una maggiore comprensione della struttura e della funzione del genoma di un organismo e delle relazioni evolutive tra le specie.
Il de novo assembly consiste di tre fasi [Figura 1]:

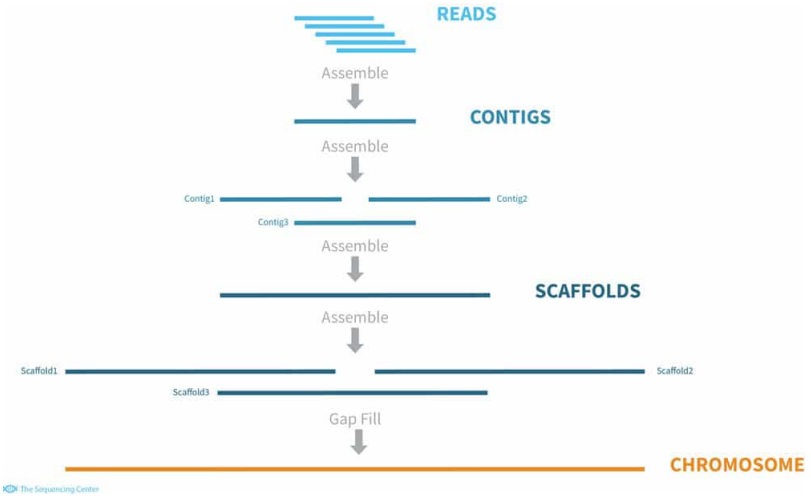

Figura 1: Fasi del De novo assembly

1. **Assemblaggio dei contigs**: la prima fase consiste nell'assemblaggio dei reads in contigs.
   Questa fase è solitamente eseguita mediante l'utilizzo di algoritmi che confrontano le sovrapposizioni tra i frammenti di sequenza per unirli in contigs di maggiore lunghezza. I metodi di assemblaggio possono variare a seconda del tipo di sequenziamento utilizzato, ma in generale tutti gli algoritmi di assemblaggio cercano di trovare la combinazione di contigs che massimizza la sovrapposizione tra i reads.
   Questa fase è solitamente la più intensiva in termini di risorse computazionali in quanto richiede l'elaborazione di grandi quantità di dati;
2. **Allineamento dei contigs in scaffolds**: una volta ottenuti i contigs, nella seconda fase del processo, questi vengono allineati tra di loro in modo da formare scaffolds, ovvero sequenze di maggiore lunghezza rispetto ai contigs che tengono conto della distanza e dell'orientamento relativo dei contigs. Questa fase è solitamente eseguita attraverso l'utilizzo di informazioni esterne, ad esempio mappature genetiche o sequenze di sondaggio, che permettono di determinare l'ordine e la posizione dei contigs sul genoma originale. In alternativa, è possibile utilizzare dati di sequenziamento di lunghezza elevata che consentono di ottenere un assemblaggio più preciso rispetto ai metodi di sequenziamento di corta lunghezza;

3. **Raffinamento degli scaffolds**: nell'ultima fase gli scaffolds vengono raffinati per migliorare la qualità dell'assemblaggio. In questa processo vengono utilizzate tecniche di validazione dei risultati, come la verifica della corretta mappatura degli scaffolds sulla mappa fisica del genoma originale, l'individuazione di eventuali errori di assemblaggio o di sequenziamento, l'utilizzo di dati di sequenziamento di diversi tipi per confermare l'accuratezza dell'assemblaggio. Questa fase è fondamentale per ottenere un assemblaggio di alta qualità e affidabilità del genoma originale.

1.2 ABySS

ABySS (Assembly By Short Sequences) è stato il primo strumento scalabile per il de novo assembly in grado di assemblare un genoma umano utilizzando letture brevi da una piattaforma di sequenziamento ad alto rendimento.
Il sistema è composto da tre fasi che rispecchiano le tre frasi del de novo assembly:

- **Unting**: in questa fase viene eseguito l'assemblaggio iniziale delle sequenza secondo il paradigma dell'assemblaggio a grafo di de Brujin;
- **Contig**: in questa fase vengono allineate le letture paired-end alle unitig (ottenute precedentemente) e tramite le informazioni di appaiamento vengono orientate e unite le unitig sovrapposte;
- **Scaffold**: in questa fase vengono allineate le letture mate-pare ai contig per poi orientarle e unirle in scaffold inserendo serie di caratteri "N" in corrispondenza di lacune nella copertura e per le ripetizioni non risolte.

La fase più dispendiosa in termini di risorse di ABySS è quella dell'assemblaggio di untig e rappresenta anche il picco di memoria richiesto.
Questa fase carica l'intero set di k-mer dalle letture di sequenziamento in ingresso in una tabella hash e memorizza dati ausiliari per ciascun k-mer, come il numero di occorrenze del k-mer nelle letture e la presenza/assenza di possibili k-mer vicini nel grafo di de Bruijn.
Il problema del grande utilizzo di memoria è stato risolto implementando una versione distribuita dell'approccio all'assemblaggio del grafo di de Bruijn, in cui la tabella hash dei k-mer è suddivisa tra i nodi del cluster e la comunicazione tra i nodi avviene tramite lo standard **MPI** (Message Passing Interface), protocollo per la comunicazione tra nodi appartenenti a un cluster di computer che eseguono un programma parallelo multi-nodo. Con questi mezzi, ABySS 1.0 consente l'assemblaggio di genomi di grandi dimensioni su cluster di hardware di base.

2 Prerequisiti

In questa sezione verranno descritti due importanti strumenti che sono alla base del modello ABySS 2.0, ovvero il **grafo di De Brujin** e il **Bloom Filter**.

2.1 Grafo di de Brujin

Un grafo di De Bruijn è un tipo di grafo diretto etichettato che viene utilizzato per rappresentare sequenze di simboli, come ad esempio sequenze di DNA o proteine.
In questa tipologia di grafo i nodi rappresentano sottostringhe di lunghezza k e gli archi rappresentano una relazione tra queste sottostringhe.
In particolare, ogni arco collega due nodi, $v1$ e $v2$, se e solo se la sottostringa rappresentata dal nodo $v2$ è ottenuta aggiungendo un simbolo alla fine della sottostringa rappresentata dal nodo $v2$. In questo modo, ogni sequenza di simboli può essere rappresentata come un percorso nel grafo di De Bruijn.
Formalmente, un grafo di De Bruijn è un grafo diretto etichettato $G = (V, E, \Sigma)$ dove:

- **V**: insieme dei nodi rappresentati da tutte le sottostringhe di lunghezza k su un alfabeto Σ;
- **E**: insieme degli archi tra i nodi. Ogni arco rappresenta la sovrapposizione di una base tra due sottostringhe adiacenti di lunghezza $k - 1$;
- Σ: alfabeto dei simboli.

In particolare, un arco $(u, v) \in E$ rappresenta la sottostringa di lunghezza k che ha un prefisso uguale alla sottostringa rappresentata dal nodo u e un suffisso uguale alla sottostringa rappresentata dal nodo v ma con l'aggiunta di un simbolo alla fine.
Il grafo di De Bruijn può essere usato per rappresentare qualsiasi sequenza di simboli dell'alfabeto Σ come un percorso attraverso il grafo, dove il percorso corrisponde alla sequenza stessa e ogni nodo corrisponde ad una sottostringa di lunghezza k presente nella sequenza.
Questa caratteristica ritorna utile nell'assemblaggio dei genomi, infatti questo tipo di grafo è molto utilizzato nell'ambito bioinformatico.

2.1.1 Esempio

Consideriamo la sequenza "ATGCTAGCAC" di lunghezza 10 e composta da 5 reads ognuno di lunghezza 6.
I reads vengono spezzati in frammenti più piccoli di una determinata dimensione k, ad esempio $k = 3$. Vengono identificate le k-mers e viene disegnato un grafo di de Bruijn con $(k - 1)$-mers come nodi e k-mers come archi. Successivamente viene identificato un percorso euleriano in modo da ottenere la ricostruzione della sequenza genomica originale [Figura 2].

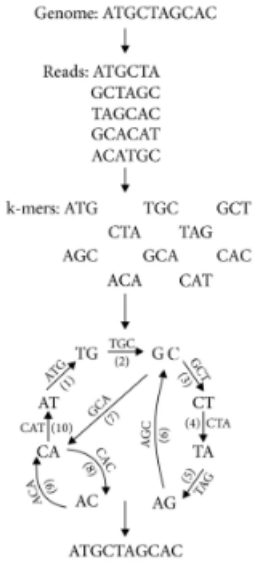

Figura 2: Esempio costruzione grafo di De Brujin

2.2 Bloom Filter

Un **bloom filter** è una struttura dati probabilistica efficiente dal punto di vista dello spazio, utilizzata per verificare se un elemento è presente in un insieme oppure no. Ad esempio, la verifica della disponibilità di un nome utente è un problema di appartenenza ad un insieme dove l'insieme è l'elenco di tutti i nomi utenti già utilizzati. L'efficienza deriva dall'essere una struttura dati probabilistica, questo significa che potrebbero verificarsi dei falsi positivi. Nell'esempio dei nomi utenti un falso positivo si verifica quando viene data come risposta la presenza del nome utente nel registro quando in realtà non è così.
Di seguito alcune delle proprietà fondamentali del Bloom filter:

- Diversamente da una normale tabella hash, un Bloom filter di una grandezza fissata può rappresentare un insieme con un numero arbitrario di elementi;
- L'aggiunta di un elemento non fallisce mai. Tuttavia, il tasso di falsi positivi aumenta costantemente con l'aggiunta di elementi fino a quando tutti i bit del filtro sono impostati su 1, a quel punto tutte le query danno un risultato positivo:
- I Bloom filter non generano mai falsi negativi, ovvero se si effettua una query con un nome utente presente nella lista allora la risposta non sarà mai negativa;
- L'eliminazione non è possibile, questo perché per eliminare un singolo elemento bisogna impostare i bit associati a 0 e per fare questo si rischia di impostare a 0 dei bit associati ad un altro elemento andando così ad aumentare la possibilità di falsi negativi.

2.2.1 Funzionamento

Inizialmente un Bloom filter è formato da un array di m bit settati a 0 [Figura 3].

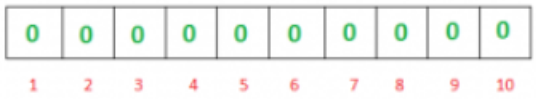

Figura 3: Stato iniziale del Bloom Filter

Abbiamo bisogno di k funzioni hash per calcolare i valori hash dell'input dato. Quando si vuole aggiungere un elemento al filtro vengono impostati a 1 i k indici calcolati tramite le funzioni hash.
Per esempio si vuole aggiungere il nome utente "Davide" al filtro, si dispone di 3 ($k = 3$) funzioni hash e un array di bit di lunghezza 10 ($m = 10$), tutti impostati inizialmente a 0.
Per prima cosa vengono calcolati gli indici tramite le funzioni hash come di seguito:

- h1("Davide") % 10 = 1
- h2("Davide") % 10 = 4
- h3("Davide") % 10 = 7

Successivamente i bit agli indici 1, 4 e 7 vengono settati a 1 [Figura 4] .

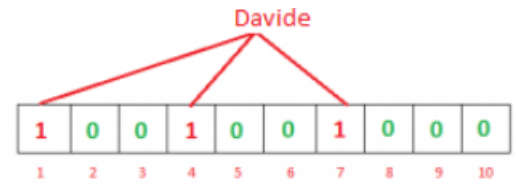

Figura 4: Aggiunta della stringa "Davide"

Successivamente si vuole aggiungere "Luca" e vengono calcolati gli indici come nel caso precedente [Figura 5]:

- h1("Luca") % 10 = 3
- h2("Luca") % 10 = 5
- h3("Luca") % 10 = 4

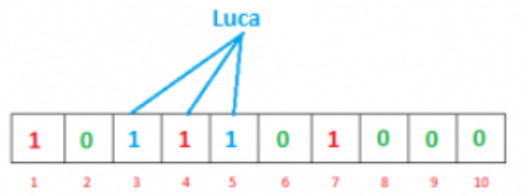

Figura 5: Aggiunta della stringa "Luca"

Per verificare se "Davide" è presente nel filtro si esegue lo stesso processo ma in ordine inverso. Vengono calcolati i rispettivi hash utilizzando $h1$, $h2$, $h3$ e si controlla se tutti gli indici ottenuti sono impostati a 1 nell'array di bit. Se tutti i bit sono impostati a 1 allora l'elemento è probabilmente presente altrimenti sicuramente non è presente.
Supponiamo di voler verificare se il nome utente "Francesco" è presente o meno nel filtro. Per prima cosa vengono calcolati gli indici tramite le funzioni hash $h1$, $h2$ e $h3$ [Figura 6]:

- h1("Francesco") % 10 = 1
- h2("Francesco") % 10 = 3
- h3("Francesco") % 10 = 7

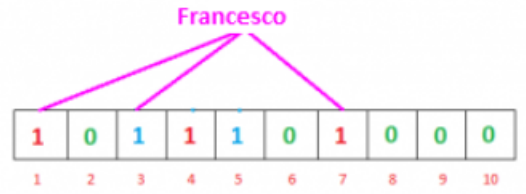

Figura 6: Ricerca della stringa "Francesco"

Controllando l'array di bit si nota che i bit che corrispondono a questi indici sono impostati a 1 ma il nome utente "Francesco" non è mai stato aggiunto.
Infatti i bit 1 e 7 sono stati impostati a 1 quando è stato inserito "Davide" mentre il bit 3 dall'inserimento di "Luca".
Dato che gli indici calcolati sono già stati impostati a 1 da qualche altro elemento il Bloom filter afferma erroneamente che "Francesco" è presente generando così un falso positivo.
La probabilità dei falsi positivi si basa sulla grandezza dell'array e sul numero di funzioni hash. Quindi se si vuole diminuire questa probabilità bisogna aumentare la dimensione dell'array e il numero di funzioni hash. Fare ciò però comporterebbe una maggiore latenza nelle operazioni.

2.3 Cascade Bloom Filter

Come detto nelle sezioni precedenti aggiungere elementi in un Bloom filter risulta semplice mentre rimuoverli no. Per ovviare a questo problema viene introdotto il **counting Bloom filter**.
Un counting Bloom Filter è un array composto da m counter, dove il valore iniziale di ogni counter è pari a 0. Il filtro funziona nel seguente modo:

- Per inserire un elemento e vengono computati gli indici $h_i(e)$ per $i = 1, ..., k$ e incrementato ogni counter di 1;
- Per eliminare un elemento e vengono computati gli indici $h_i(e)$ e decrementati i counter di 1;
- Per verificare l'appartenenza di un elemento e vengono calcolati gli indici $h_i(e)$ e viene verificato se il counter di ogni indice è almeno 1.

In questo modo il Bloom filter acquisisce la capacità di rimozione degli elementi a discapito dello spazio. Infatti, i counter possono essere implementati in spazio $O(log(C))$, dove C è il valore massimo che si prevede di memorizzare nel contatore, dove le operazioni di incremento, decremento e controllo dello 0 vengono fatte in tempo costante.

Per avere un'ulteriore miglioria anche sull'utilizzo dello spazio, si fa uso del **cascade Bloom filter**. Questa tipologia di filtro permette di rappresentare, in modo efficiente, un insieme di elementi senza incorrere nei falsi positivi.

Per definire un cascade Bloom filter introduciamo il concetto di profondità indicata con d e un Bloom filter per ogni livello $1, ..., d$. Denotiamo il Bloom filter al livello i come BF_i che rappresenta A_i (dove A_i è l'insieme di elementi che possono essere inseriti in BF_i).
Il Bloom filter BF_1 rappresenta l'insieme A dell'universo U, ovvero $A_1 = A$. A_2 è l'insieme degli elementi provenienti da $U - A_1$ che sono i falsi positivi di BF_1. Successivamente, ogni A_i comprende quegli elementi da A_{i-2} che risultato falsi positivi in BF_{i-1}. Infine, gli elementi di A_{d-1} che risultano falsi positivi in BF_d vengono memorizzati in un elenco, ad esempio una hashtable, denotando l'insieme come A_{d+1}.
In figura[6] viene mostrato un esempio di una struttura dell'insieme A con l'universo U rappresentata da un cascade Bloom filter con profondità pari a 4.

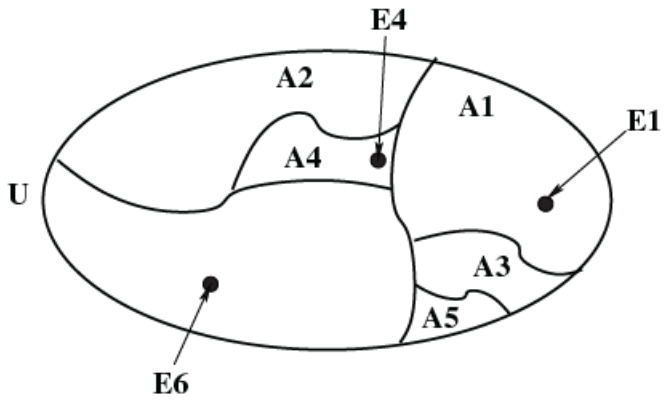

Figura 7: Esempio di un cascade Bloom filter con $d = 4$

In questo esempio si nota come l'universo U viene diviso negli insiemi A_i per un Cascade Bloom filter di profondità 4. Ogni A_{i+1} è l'insieme di falsi positivi da A_{i-1} nel Bloom filter che rappresenta A_i, con $A_0 = U$. Da notare:

- $A_1 \supseteq A_3 \supseteq A_5$, e $A_2 \supseteq A_4$;
- l'ultimo insieme, A_5, non ha un Bloom filter corrispondente quindi viene rappresentato esplicitamente (per esempio tramite hashtable);
- l'elemento $E_1 \in A_1$ e a nessun altro insieme, quindi un test di appartenenza risulta *true* per BF_1 e *false* per BF_2;
- l'elemento $E_4 \in A_4 \supseteq A_2$, quindi un test di appartenenza ha esito *true* per tutti i $BF_1, ..., BF_4$. Dato che $E_4 \notin A_5$ allora non è un falso positivo per BF_3 quindi $E_4 \notin A_1 = A$;
- l'elemento $E_6 \notin A_i$ per ogni A_i, quindi un test di appartenenza per BF_1 da come esito *false*.

Di seguito la defizione formare di un **Cascade Bloom filter**:
Un Cascade Bloom filter C è un coppia $< B, E >$, dove $B = \{BF_1, ..., BF_d\}$ è una lista di Bloom filter e $E \subseteq U$ è un insieme di elementi dell'universo U. d viene definito come profondità del filtro a cascata e ogni $l = 1, ..., d$ viene chiamato livello. Ogni BF_i rappresenta un insieme $A_i \subseteq U$, tale che per $i = 2, ..., d$ A_i è l'insieme dei falsi positivi appartenenti a A_{i-2} in BF_{i-1}, con $A_0 = U$. L'insieme E è l'insieme dei falsi positivi appartenenti a A_{d-1} in BF_d. Diremo che C rappresenta $A = A_1$ con universo U.

3 ABySS 2.0

In questa sezione si andrà ad analizzare ABySS 2.0, in particolare delle evolutive rispetto alla versione precedente e dei miglioramenti che si sono avuti per quanto riguarda le performance.

3.1 Perché nasce

Come detto precedentemente, ABySS 2.0 nasce dall'esigenza di migliorare le performance del suo predecessore specialmente per quanto riguarda l'utilizzo della memoria nella prima fase. Infatti, la fase più dispendiosa in termini di risorse di ABySS 1.0 è quella dell'assemblaggio di unitig e rappresenta anche il picco di memoria richiesto.
Questa fase della pipeline carica l'intero set di k-mer dalla lettura di sequenziamento in ingresso in una tabella hash e memorizza dati ausiliari per ciascun k-mer. ABySS 1.0 risolve il problema di uso di un'elevata memoria implementando una versione distribuita dell'approccio all'assemblaggio del grafo di de Brujin, in cui la tabella hash dei k-mer è suddivisa tra i nodi del cluster e la comunicazione tra di essi avviene tramite lo standard MPI (Message Passing Interface).
In ABySS 2.0 è stata eseguita una riprogettazione della vecchia versione seguendo il modello di Minia in cui non si ha più l'utilizzo di MPI per migliorare l'efficienza di calcolo ma si fa uso di algoritmi che impiegano un Bloom Filter, una struttura dati probabilistica, utile a rappresentare in maniera efficiente il grafo di de Brujin in modo tale da ridurre i requisiti di memoria.
Quindi la principale innovazione è stata quella di implementare la fase di assemblaggio di uniting usando un Bloom filter consentendo l'assemblaggio di genomi di grandi dimensioni su una singola macchina così da ridurre i requisiti di memoria.

3.2 Come funziona

La struttura dati Bloom filter consiste in un vettore di bit e in una o più funzioni di hash, dove le funzioni hash mappano ogni k-mer a un corrispondente insieme di posizioni all'interno del vettore di bit; l'insieme composto dalle posizioni di bit prende il nome di firma di bit per il k-mer. Un k-mer viene inserito nel Bloom filter impostando le posizioni della sua firma di bit a 1 e viene interrogato verificando se tutte le posizioni della sua firma di bit sono 1.
Sebbene il Bloom filter sia un mezzo molto efficiente in termini di memoria per rappresentare un insieme, ha il difetto di generare falsi positivi quando le firme dei bit di k-mer diversi si sovrappongono per caso.
Ciò significa che una certa frazione di interrogazioni di k-mer risulterà vera anche se i k-mer non esistono nei dati di sequenziamento in ingresso.
In dettaglio, la fase di assemblaggio di unitig consiste di due passaggi:

- Nel primo passaggio vengono estratti i k-mers dalle letture e vengono caricate in un Bloom filter. Per eliminare i k-mers generati da errori di sequenziamento vengono scartati i k-mers con un numero di occorrenze minore rispetto ad una soglia specificata dall'utente (tra 2 e 4). I k-mers restanti vengono definiti come **k-mers solidi**;
- Nel secondo passaggio vengono identificate le letture che consistono solo di k-mers solidi, chiamati **letture solide**, che vengono estesi sia a sinistra che a destra all'interno del grafo di De Brujin in modo tale da creare delle unitigs. Dato che nel Bloom filter sono memorizzati solo i nodi (k-mers) del grafo di De Brujin e non gli archi, durante la fase di attraversamento del grafo il Bloom filter viene interrogato con tutti i quattro possibili k-mers vicino al k-mer corrente. Questo passaggio permette di scoprire gli archi uscenti. Da notare che durante questa fase di estensione è possibile che più letture solide diano luogo alla stessa unitig. Per evitare queste sequenze duplicate viene utilizzato un secondo Bloom filter per tenere traccia dei k-mer inclusi in precedenti unitig in modo tale da estendere una lettura solida solo se ha almeno un k-mer non presente nel Bloom filter.

3.3 Caratteristiche comuni con Minia

L'uso dei Bloom filter per il de novo assembly è stato dimostrato per la prima volta in Minia e ABySS 2.0 si basa su molti aspetti di quell'approccio.
Le principali novità sono le seguenti:

- L'uso di letture solide per l'attraversamento dei contig;
- L'uso di look-ahead per la correzione degli errori e l'eliminazione dei falsi positivi dal Bloom filter;
- L'uso di un nuovo algoritmo di hashing, **ntHash**, progettato per elaborare in modo efficienti le sequenze di DNA/RNA

Come in Minia, il primo passo dell'algoritmo di assemblaggio consiste nel caricare tutti i k-mers dalle letture di sequenziamento in un Bloom filter, questi k-mers rappresentano l'insieme dei nodi che compongono il grafo di De Brujin. Anche la creazione degli archi avviene allo stesso modo di Mina, ovvero interrogando il Bloom filter con i quattro possibili predecessori/successori del k-mer corrente.
Ogni possibile successore (predecessore) corrisponde a un'estensione a base singola del k-mer corrente a destra (sinistra) di "A", "C", "G" o "T".
Un'ulteriore tecnica condivisa con Minia è l'uso di un Bloom filter a cascata per eliminare i k-mer a bassa frequenza causati da errori di sequenziamento.
In particolare, un Bloom filter a cascata è un array concatenato di Bloom filter in cui ogni Bloom filter memorizza k-mers con un conteggio superiore a quello del Bloom filter precedente. La procedura per inserire un k-mer in un Bloom filter a cascata consiste nell'interrogare ogni Bloom filter in successione e aggiungere il k-mer al primo Bloom filter in cui non è già presente. Dopo che tutti i k-mer delle letture sono stati inseriti, l'ultimo Bloom filter della catena viene mantenuto come insieme di k-mer solidi e i Bloom filter precedenti vengono scartati.

3.4 Differenze con Minia

Di seguito le principali differenze che AbySS 2.0 presenta rispetto a Minia:

- La prima differenza è il metodo usato per seminare le traversate del grafo per generare i contig.
  Mentre Minia identifica e memorizza i punti di ramificazione del grafo di de Bruijn da utilizzare come punti di partenza per l'attraversamento dei contig, ABySS 2.0 estende le letture solide a destra e a sinistra fino a quando non si incontra un vicolo cieco o un punto di ramificazione nel grafo.
  Una lettura è considerata solida se consiste interamente di k-mers solidi ed è quindi probabile che rappresenti un percorso corretto nel grafo di de Bruijn. La percentuale di letture solide nel set di dati dipende dalla soglia minima di presenza di k-mer specificata dall'utente. L'approccio alla generazione dei contig basato sull'estensione delle letture ha il vantaggio di essere semplice da implementare ma richiede alcune precauzioni per garantire che i contig ridondanti non vengano generati da letture solide situate nello stesso quartiere del grafico di de Bruijn. Per risolvere questo problema viene utilizzato un Bloom filter aggiuntivo di tracciamento per registrare l'insieme dei k-mers che sono stati precedentemente inclusi nei contig; se tutti i k-mers di una lettura solida sono già contenuti nel Bloom filter di tracciamento, non viene estesa in un contig ma viene saltata. Affinché questo schema funzioni correttamente le letture solide che coprono punti di ramificazione del grafo di De Bruijn devono essere divise nei punti di ramificazione e trattate come candidati separati per l'estensione;

- La seconda sostanziale differenza si ha sulla gestione dei falsi positivi generati dal Bloom filter.
  Infatti, Minia utilizza una struttura dati aggiuntiva non probabilistica utile a memorizzare i falsi positivi, invece ABySS 2.0 fa uso di un meccanismo di look-ahead durante l'attraversamento del grafo per eliminare i rami corti generati dai falsi positivi e dagli errori di sequenziamento.
  In dettaglio, il look-ahead viene invocato a ogni punto di ramificazione che si incontra durante l'estensione del contig fino a una distanza di k nodi. Se la fase di look-ahead rivela che un ramo ha una lunghezza inferiore o uguale a k nodi viene considerato un falso ramo e viene ignorato. Se invece il punto di diramazione ha due o più rami di lunghezza superiore a k nodi l'estensione unitaria viene interrotta.
  Purtroppo l'uso del look-ahead comporta un costo computazionale aggiuntivo per l'attraversamento del grafo ma evita l'uso di strutture dati aggiuntive per tenere traccia dei falsi positivi e dei k-mers erroneamente generati;

- L'ultima differenza riguarda la funzione hash utilizzata. In ABySS 2.0 viene utilizzata la funzione hash ntHash. Questa funzione consiste in un algoritmo efficiente per calcolare i valori hash di tutti i k-mer consecutivi in una sequenza di DNA in modo ricorsivo in cui il valore di hash di ogni k-mer è derivato dal valore di hash del k-mer precedente. Più specificatamente, ntHash è una versione adattata dell'hashing polinomiale ciclico e viene utilizzato per calcolare valori hash normali o canonici per tutti i k-mer di una sequenza di DNA. Un'ulteriore caratteristica della funzione hash è il calcolo rapido di più valori hash per lo stesso k-mer in modo da non ripetere l'intero calcolo dell'hashing.