STRUMENTI FORMALI PER LA BIOINFORMATICA

Abyss e Bloom Filters: Un caso di studio

Autori
Pietro Negri 0522501367
Francesco Maddaloni 0522501740

Contents

1 Introduzione **3**

2 Strumenti e strutture dati **4**
2.1 De Bruijn Graph 4
2.2 Bloom Filters 5

3 Minia **7**
3.1 Caratteristiche 7

4 Abyss **12**
4.1 Caratteristiche 12
4.2 ntHash 14
4.2.1 ntHash 1 14
4.2.2 ntHash 2 15

5 Benchmark NtHash2, MurmurHash3, CityHash e test di Kolmogorov-Smirnov **19**
5.1 Test di Kolmogorov-Smirnov 20
5.2 Benchmark Nthash2, MurmurHash3, CityHash 22
5.3 Implementazione e test di un Bloom Filter 26

6 Conclusioni **29**

Abstract

Il lavoro di questa tesina presenta una panoramica di alcune delle più popolari soluzioni per l'assemblaggio di sequenze genomiche, concentrandosi in particolare su ABySS 2.0 [5], uno dei primi assemblatori sufficientemente leggeri da essere utilizzati su un computer desktop consumer. Verranno brevemente introdotti concetti relativi all'assemblaggio delle sequenze genomiche [4], alle strutture dati utilizzate tipicamente dagli assemblatori [10], per poi fare una panoramica su Minia [3], uno dei primi assemblatori a utilizzare un Bloom Filter, per poi concentrare la nostra attenzione su ABySS, che segue le orme di Minia ma con alcune differenze. Infine, molto della tesina si concentra su una delle innovazioni di ABySS, ovvero una funzione hash chiamata nTHash [9] [6] che risulta essere particolarmente prestante e ottimizzata per l'uso in campo bioinformatico. Di essa verranno analizzate alcune soluzioni adottate e verrà mostrato del codice per spiegarne e testarne il funzionamento.

1 Introduzione

Il sequenziamento del DNA rappresenta una delle innovazioni più rivoluzionarie nel campo della biologia e della bioinformatica. La capacità di determinare l'ordine delle basi nucleotidiche all'interno di un frammento di DNA ha aperto nuove frontiere nella comprensione dei processi biologici, nello studio delle malattie genetiche e nello sviluppo di terapie personalizzate. Nel contesto della bioinformatica, il sequenziamento del DNA richiede sofisticati strumenti di assemblaggio per ricostruire accuratamente i genomi a partire da frammenti di sequenza generati dalle moderne tecnologie di sequenziamento ad alta velocità. Il sequenziamento ha applicazioni in numerosi ambiti, tra cui la genomica comparativa e la medicina personalizzata.
Tuttavia, l'analisi dei dati generati dal sequenziamento pone sfide significative, soprattutto in termini di gestione e interpretazione delle immense quantità di dati prodotti. L'assemblaggio del genoma è un processo cruciale che consiste nel ricostruire la sequenza originale di un genoma a partire dai frammenti di sequenza generati dal sequenziamento [4]. Questo compito è reso complesso dalla presenza di regioni ripetitive, errori di sequenziamento e la frammentazione dei dati. Strumenti di assemblaggio efficaci sono quindi fondamentali per ottenere sequenze genomiche accurate e complete.
L'accuratezza dell'assemblaggio influisce direttamente sulla qualità delle analisi successive, come l'annotazione genica, l'identificazione delle mutazioni e lo studio delle strutture genomiche. Pertanto, la ricerca e lo sviluppo di algoritmi e software di assemblaggio avanzati è un'area di interesse cruciale.
Questa tesi si propone di esaminare un particolare strumento di assemblaggio, Abyss, valutandone le caratteristiche e le particolarità, anche in riferimento a strumenti simili e già noti in letteratura. Nel seguito verranno analizzati i principi teorici alla base dello strumento, le sue maggiori ispirazioni, i suoi algoritmi e le sue implementazioni pratiche.

2 Strumenti e strutture dati

Prima di iniziare con l'analisi di Abyss, due concetti sono di fondamentale importanza per comprendere il seguito.

2.1 De Bruijn Graph

I grafi di De Bruijn sono delle particolari tipologie di grafo, introdotti dal matematico olandese Nicolaas de Bruijn, interessato al problema della "superstringa": trovare la più piccola superstringa circolare che contenesse tutte le possibili sottostringhe di lunghezza k (k-mer) su un certo alfabeto.
Esistono diverse definizioni del grafo di De Bruijn, che non dipendono necessariamente dalla Bioinformatica. In generale, assumendo di avere un alfabeto con n simboli, sappiamo di avere n^k k-mer. Per costruire il grafo, assegnamo ogni possibile (k-1)-mer a un nodo, e colleghiamo ogni (k-1)-mer a un altro (k-1)-mer se c'é qualche k-mer che ha come prefisso il primo e come suffisso il secondo. Gli archi rappresenteranno quindi tutti i possibili k-mer.
Più specificatamente, possiamo definire i grafi di de Bruijn in modo più formale usando sia una definizione nodo-centrica che una definizione arco-centrica. Di seguito riportiamo la seconda:
Definizione: Un de Bruijn Graph di una collezione di reads $\{r1, r2, \ldots, rn\}$ è il grafo $G = (V, A)$ dove:

- **A** rappresenta l'insieme degli elementi dello spettro di ordine k; cioè, l'arco dal nodo $v1$ al nodo $v2$ è tale per cui:
  - il suffisso di $v1$ lungo $k-2$ è uguale a prefisso di $v2$ lungo $k-2$ (overlap).
  - il $k-mer$ ottenuto assemblando $v1$ e $v2$ è sottostringa in almeno una read della collezione.
- L'arco è etichettato dall'estensione (lunga un carattere) di $v2$ che eccede l'overlap con $v1$.

Il motivo per cui i grafi di De Bruijn sono così importanti è che l'assemblaggio del genoma viene ricondotto a un problema di ricerca di cammini particolari all'interno di questo grafo [4]. Il problema è cercare di costruire un genoma assemblando miliardi di piccole reads in una singola, per così dire, parola.
Metodi molto semplici inizialmente non usavano il grafo di de Bruijn, ma un semplice grafo in cui ogni read era rappresentata da un nodo e l'overlap fra le reads era rappresentato da un arco diretto che univa due reads.
A questo punto si ricercava un particolare tipo di cammino in questo grafo, quello Hamiltoniano, ovvero un cammino che attraversasse ogni nodo esattamente una sola volta. Il problema è che non esiste un algoritmo efficiente per trovare un cammino Hamiltoniano, trattandosi di un problema NP-Completo [4]. Quello che invece si cerca di fare è trovare un cammino che attraversi tutti gli archi una e sola volta, chiamato cammino Euleriano.

2.2 Bloom Filters

Il Bloom Filter è una struttura dati probabilistica utilizzata per verificare in maniera efficiente l'appartenenza di un elemento a un insieme. [10] [5] [2] Introdotta negli anni 70 da Howard Bloom, ha avuto una grande popolarità negli anni in diversi domini applicativi, soprattutto per migliorare la complessità spaziale e temporale.
Dunque si tratta di una struttura dati molto efficiente, ma con dei problemi. Infatti, la sua natura probabilistica causa la presenza di falsi positivi e, laddove sia possibile anche cancellare elementi, dei falsi negativi. Come struttura dati ha un'interfaccia molto semplice che prevede inserimento, query e cancellazione, ma, come anticipato, talvolta si preferisce evitare di consentire la cancellazione per non rischiare falsi negativi. Nel nostro caso, la cancellazione non sarà considerata in quanto inutile ai fini della risoluzione del problema. In genere, vengono implementati come dei semplici array di bit, in cui ogni casella consiste di uno zero oppure di un uno. La particolarità del Bloom Filter è che per utilizzare le operazioni della sua interfaccia bisogna prima effettuare un hash dell'elemento da inserire o da controllare. Ogni Bloom Filter presenterà quindi k funzioni hash "a guardia" della struttura e verranno invocate ogni qualvolta bisognerà utilizzarlo. La complessità temporale di un Bloom Filter è O(k) per inserimento e membership query, assumendo che k sia il nostro numero di funzioni hash.[10]
Ogni qualvolta bisogna aggiungere un elemento al filtro, si fa l'hash dell'elemento con ognuna delle k funzioni hash. Il risultato sarà che k indici del bit array verranno messi a 1 per segnalare l'inserimento dell'elemento. Allo stesso modo, ogni qualvolta bisognerà controllare la presenza di un elemento verranno ricalcolate, usando le stesse k funzioni hash, i k indici da controllare.
Se tutti gli indici sono a 1, l'elemento è presente nel Bloom Filter. Altrimenti, l'elemento non è presente.
Assumendo che non sia possibile cancellare elementi, non esiste rischio di falsi negativi [10]. A prescindere da questa possibilità è invece sempre possibile avere falsi positivi.
Infatti, il fatto che sia presente indica soltanto che tutti gli indici della sua "bit signature" sono stati posti a 1, ma ciò potrebbe essere avvenuto perché altri elementi condividono parte di questa signature e hanno messo gli stessi bit a 1 in momenti diversi del ciclo di vita della struttura dati.

Volendo essere più rigorosi individuiamo 4 casistiche principali:

a) **True positive**: Oggetto inserito effettivamente e risposta del filtro pari a true durante la query. L'elemento è davvero presente nel filtro e i bit della sua bit signature contengono 1.

b) **True negative**: Oggetto non inserito e risposta ottenuta dal filtro durante la query pari a false. L'oggetto effettivamente non è presente, perché almeno una posizione della bit signature contiene 0.

c) **False positive**: L'oggetto non è inserito, ma la risposta data dal Bloom Filter durante la query è true. L'oggetto non c'è, però per qualche motivo tutti i k indici della bit signature contengono 1. Questo accade quando altri elementi hanno parte della loro bit signature che coincide con l'elemento in questione.

d) **False negative**: L'oggetto è inserito, ma la risposta data dal Bloom Filter è false. Significa che l'oggetto dovrebbe esserci, ma alcuni indici della sua bit signature contengono 0. Questo accade se è consentita l'operazione di cancellazione.

Dunque il problema principale è rappresentato dai falsi positivi, che però sono molto difficili da eliminare. Esistono diverse varianti dei Bloom Filter che provano a ridurre la percentuale di falsi positivi attraverso diverse tecniche, e anche la tipologia di funzione hash utilizzata in relazione alla grandezza del filtro può aiutare a migliorarne le performance.
Ricordando che il Bloom Filter è una struttura dati probabilistica potremmo essere interessati a conoscere quanto sia probabile avere, per esempio, dei falsi positivi. Sia $\mathbb{B}$ un Bloom Filter, $\mathbb{S}$ un insieme, n il totale degli elementi inseriti nel Bloom Filter, m la dimensione di $\mathbb{B}$, e k il numero totale di funzioni di hash. La probabilità di avere dei bit pari a 1 è:

$$(1 - (1 - \frac{1}{m})^{kn})$$

Sia X la variabile aleatoria che rappresenta il numero totale di bit messi a '1' che è condizionata da X = x, allora la probabilità di falso positivo sarà:

$$Pr(FP|X = x) = \binom{x}{m}^k$$

A questo punto la probabilità di falsi positivi complessiva di un Bloom Filter sarà:

$$FPP = m \sum_{x=0}^{m} Pr(FP|X = x) \cdot Pr(X = x) = m \sum_{x=0}^{m} \binom{x}{m}^k \cdot f(x)$$

Da un punto di vista pratico, il Bloom Filter è una struttura dati molto semplice per controllare l'appartenenza di elementi a un insieme, in modo efficiente e utilizzando poca memoria rispetto a un Set tradizionale. Per esempio, alcune sue applicazioni pratiche le riscontriamo nelle memorie flash, o nell'ambito dei Big Data. Infine, non solo i falsi positivi ma anche la scalabilità e l'hashing sono due problemi molto importanti per questa struttura dati. Infatti, il Bloom Filter standard non è scalabile, ma è di larghezza fissata. Inoltre un Bloom Filter molto grande occupa molta memoria, che non sempre è possibile utilizzare. Esistono soluzioni, come Bloom Filter gerarchici, che migliorano la scalabilità. Infine, come accennato precedentemente, l'efficienza del filtro dipende principalmente dalla tecnica di hashing utilizzata. Ovviamente, funzioni hash più potenti possono migliorare le performance ma anche aumentare l'overhead. In particolare, nel seguito vedremo una funzione hash che è in grado di dare ottimi risultati sia di efficienza che di FPP e toccheremo anche brevemente il caso in cui ci siano più funzioni hash per lo stesso filtro.

3 Minia

L'assemblatore Abyss basa la sua architettura su Minia [3], un assemblatore il quale si sviluppa su di una particolare rappresentazione del De Brujin Graph.
L'obbiettivo principale dell'assemblatore è quello di ottimizzare lo spazio occupato da questa struttura dati attraverso l'utilizzo di un Bloom Filter per memorizzare il grafo. Questa nuova struttura così ottenuta è denominata De Brujin Graph Probabilistico. Il De Brujin Graph Probabilistico è ottenuto inserendo tutti i nodi del grafo originale (tutti i k-mers) in un Bloom Filter. Gli archi sono dedotti implicitamente cercando nel Bloom Filter l'appartenenza di tutte le possibili estensioni di un K-Mer. Per essere più precisi, un'estensione di un k-mer v è la concatenazione di:

- il suffiso $k-1$ di v con una delle possibili quattro basi nucleotidiche.
- Una delle quattro possibili basi nucleotidiche con il suffisso $k-1$ di v.

Il De Brujin Graph Probabilistico contiene, quindi, un'approssimazione per eccesso del grafo originale. Ricercare nel bloom filter l'esistenza di un nodo arbitrario potrebbe restituire un falso positivo. Ciò introduce al problema del false branching tra veri positivi e falsi positivi.

3.1 Caratteristiche

Una delle caratteristiche fondamentali di Minia è una struttura dati denominata `cFP` (Critical False Positives) che ha lo scopo di evitare il **false branching**. Durante il traversing, i falsi positivi vengono memorizzati all'interno di questa struttura. Ogni query al bloom filter restituirà quindi true se e solo se l'esito sarà positivo nel bloom filter e negativo nel `cFP`.

E' importante notare che se il set cFP contenesse tutti gli elementi falsi positivi, i benefici dell'utilizzo di un Bloom Filter per l'efficienza della memoria andrebbero persi [3].
L'osservazione chiave è che i k-mer che verranno interrogati durante il traversing del grafo non sono tutti i possibili k-mer.
Sia S il set di nodi veri positivi e E il set delle estensioni dei nodi di S. Supponendo di attraversare il grafo partendo solo da un nodo in S, i falsi positivi che non appartengono a E non verranno mai interrogati. Pertanto, il set `cFP` sarà un sottoinsieme di E.
Sia P il set di tutti gli elementi di E per i quali il filtro di Bloom risponde sì. Il set `cFP` è quindi definito formalmente come $cFP = P/S$.

Il set `cFP` può essere costruito utilizzando un algoritmo che limita il suo utilizzo di memoria, ad esempio alle dimensioni del Bloom Filter.
Il set P viene creato su disco, da cui `cFP` viene gradualmente costruito filtrando iterativamente P con partizioni di S caricate in una tabella hash.

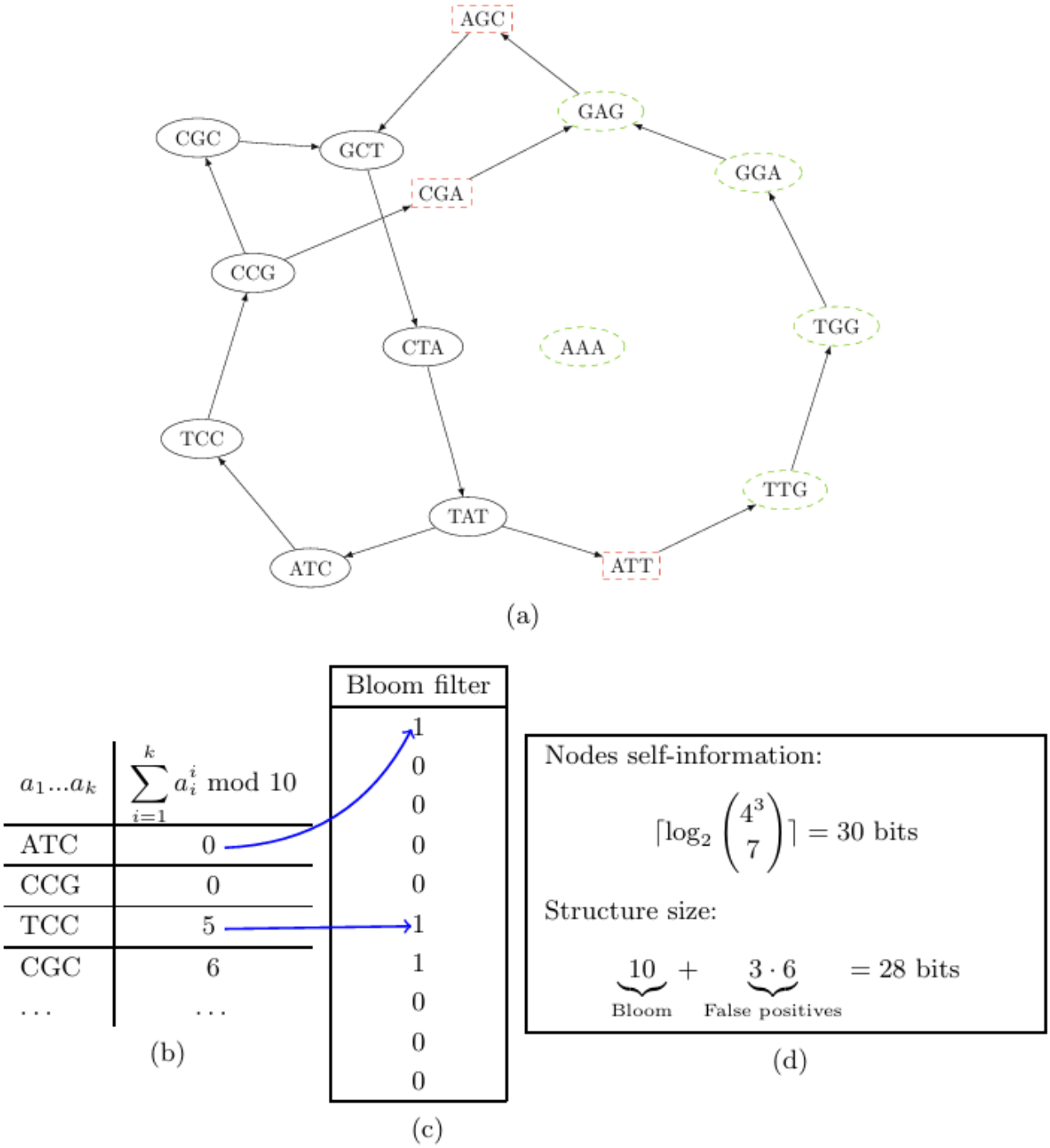


Figure 1: Esempio di gestione della struttura cFP

L'immagine sopra riportata esplica un esempio completo di rimozione dei falsi positivi nel grafo probabilistico di de Bruijn. Per andare nel dettaglio:

- (a) Mostra il grafo di de Bruijn originale (con 7 nodi non tratteggiati) e la sua rappresentazione prob-

abilistica tramite un Bloom Filter (prendendo l'unione di tutti i nodi). I nodi rettangolari tratteggiati (in rosso nella versione elettronica) sono i vicini immediati di S nel grafo probabilistico. Questi nodi sono i falsi positivi critici.

- (b) Mostra un campione dei valori hash associati ai nodi di S (è stato utilizzato un hash di prova).
- (c) Mostra il Bloom Filter completo associato a S; incidentalmente, i nodi di B corrispondono esattamente a quelli a cui il Bloom Filter risponde positivamente.
- (d) Descrive il limite inferiore per la codifica esatta dei nodi di S (autoinformazione) e lo spazio richiesto per codificare struttura (Bloom Filter, 10 bit, e 3 falsi positivi critici, 6 bit per 3-mer)

Per comprendere meglio come vengono empiricamente definite le dimensioni di cFP è doveroso introdurre una metrica per il tasso dei falsi positivi.

Quando si considerano le funzioni hash che producono posizioni ugualmente probabili nell'array di bit, e per una dimensione dell'array m e un numero di elementi inseriti n abbastanza grandi, il tasso di falsi positivi F è:

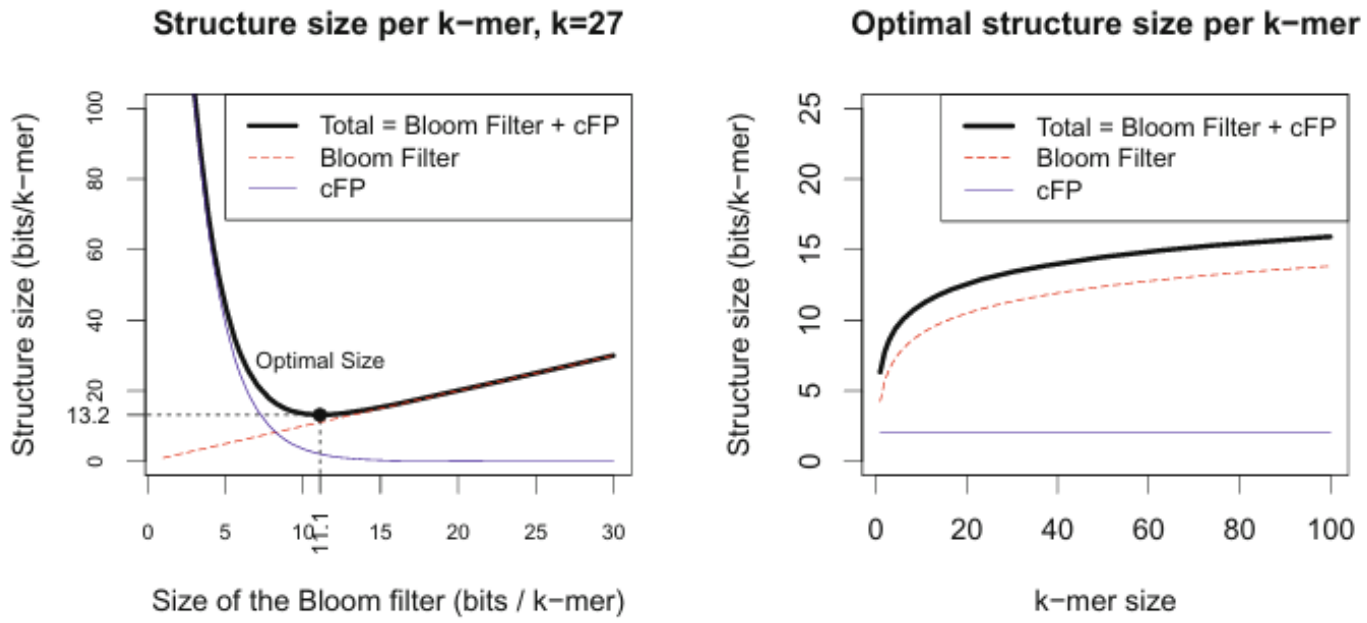
$$F \approx (1 - e^{-hn/m}) = (1 - e^{-h/r})^h \tag{1}$$

Usando le stesse notazioni previamente descritte, dato che $n = |S|$, la dimensione del filtro m e il tasso di falsi positivi F sono legati tramite l'Equazione 1.
La dimensione attesa di cFP è $8n \cdot F$, poiché ogni nodo ha solo otto possibili estensioni, che potrebbero essere falsi positivi. Nella codifica di cFP, ogni k-mer occupa $2 \cdot k$ bit. Ricorda che, per un dato tasso di falsi positivi F, la dimensione ottimale attesa del Bloom Filter è $1.44n \log_2 \left(\frac{1}{F}\right)$. La dimensione totale della struttura dovrà presumibilmente essere:

$$1.44n \log_2 \left(\frac{1}{F}\right) \quad \text{(filtro di Bloom)} \quad + \quad (16 \cdot Fnk) \quad \text{(cFP)} \quad \text{bits} \tag{2}$$

La dimensione è minima per $F \approx \left(\frac{16k}{2.08}\right)^{-1}$. Pertanto, il numero minimo di bit richiesti per memorizzare il Bloom Filter ed il set `cFP` è approssimativamente:

$$n \cdot \left(1.44 \log_2 \left(\frac{16k}{2.08}\right) + 2.08\right) \tag{3}$$



(a) (b)

Figure 2: (a) Dimensione della struttura (Bloom Filters, cFP) in funzione del numero di bit per k-mer allocati nel Bloom Filter (detto anche rapporto r) per k = 32. Il compromesso che ottimizza la dimensione totale è mostrato con linee tratteggiate. (b) Dimensione ottimale della struttura per diversi valori di k.

Un' altra importante caratteristica dell'architettura di Minia è una particolare struttura dati utilizzata per gestire il traversing del grafo.
Ciò è stato necessario in quanto nè il Bloom Filter nè il grafo stesso permettono di tener traccia del traversing in quanto il primo è immutabile mentre il secondo non consente l'inserimento di un secondo bit per tale scopo.
Questa particolare struttura dati consente di capire quali porzioni del grafo sono già state visitate.
L'idea alla base di questo meccanismo è che non tutti i nodi devono essere marcati. In particolare, i nodi che si trovano all'interno di percorsi semplici (cioè nodi con un grado entrante pari a 1 e un grado uscente pari a 1) saranno o tutti marcati o tutti non marcati. I nodi con un grado entrante o uscente diverso da 1 saranno denominati **nodi complessi**. La proposta è di registrare le informazioni di marcatura dei nodi complessi, memorizzando esplicitamente questi nodi in una tabella hash separata. Nei grafi di de Bruijn dei genomi, l'insieme completo dei nodi supera di gran lunga l'insieme dei nodi complessi, anche se il rapporto dipende dalla complessità del genoma.

L'uso della memoria della struttura di marcatura è ncC, dove nc è il numero di nodi complessi nel grafo e C è l'uso della memoria di ciascuna voce nella tabella hash ($C \approx 2k + 8$).

Il workflow di Minia prevede un passaggio preliminare nel quale viene recuperato l'elenco dei k-mer distinti che compaiono nelle sequenze, ovvero i nodi veri del grafo. Per eliminare gli errori di sequenziamento probabili, vengono mantenuti solo i k-mer che compaiono almeno d volte (k-mer solidi).

I metodi classici per recuperare i k-mer solidi si basano su tabelle hash, ed il loro utilizzo di memoria aumenta linearmente con il numero di k-mer distinti. Per evitare di utilizzare più memoria di quella disponibile, è stata implementata una procedura di conteggio dei k-mer in memoria costante.
In Minia è presente un algoritmo di attraversamento del grafo che costruisce un insieme di *contig* (sequenze senza spazi).
Il Bloom Filter e la struttura `cFP` vengono utilizzati per determinare i vicini di ciascun nodo. La struttura di marcatura registra i nodi già attraversati. Viene eseguito un algoritmo `BFS` (breadth-first search) con profondità e ampiezza limitate per attraversare regioni brevi e localmente complesse. In particolare, l'attraversamento ignora i "tips" (percorsi senza uscita) più corti di $2k + 1$ nodi.
L'algoritmo sceglie un singolo percorso (in modo coerente ma arbitrario) tra tutti i percorsi possibili che attraversano regioni del grafo con una larghezza massima di 20, a condizione che queste regioni terminino con un singolo nodo di profondità massima di 500. Si presume che queste regioni siano errori di sequenziamento, varianti brevi o ripetizioni corte di lunghezza $\leq 500bp$. Il limite di larghezza impedisce una crescita esponenziale. Nota che le informazioni sui read paired-end non vengono considerate in questa procedura. In un tipico flusso di lavoro di assemblaggio, un programma separato (scaffolder) può essere utilizzato per collegare i contig utilizzando le informazioni sui pairing.

4 Abyss

Abyss, rilasciato per la prima volta nel 2009 con ABySS 1.0, è stato il primo tool di assemblaggio de novo scalabile in grado di assemblare il genoma umano, utilizzando delle short reads. Per riuscire nell'impresa necessitava però di moltissima memoria distribuita su più compute nodes che comunicavano utilizzando il protocollo MPI.
Questo requisito lo rendeva difficile da usare in piccoli centri di ricerca, costituendo di fatto un bottleneck. La maggior parte degli strumenti concorrenti condivideva lo stesso identico problema, e nel tempo sono state sviluppate diverse alternative, fra cui il sopraccitato Minia.
Oggi ABySS 2.0 si presenta come un assemblatore de novo in grado di ridurre i requisiti di memoria di un ordine di magnitudo, con risultati che lo rendono paragonabile agli strumenti già esistenti.
La sua struttura di base segue il modello di Minia, utilizzando un Bloom Filter per l'embedding del grafo di De Bruijn, con alcune aggiunte, fra cui la più significativa è l'uso di una particolare funzione hash, nTHash2.
Ora analizzeremo molto brevemente la struttura dello strumento di assemblaggio, nella sua versione 1.0 e nella versione 2.0. Descriveremo i problemi della versione 1.0 che sono stati poi rifiniti nella 2.0, e ci soffermeremo sulle novità introdotte e sui punti di forza, mostrando anche alcuni snippet di codice.

4.1 Caratteristiche

ABySS 1.0 era organizzato come una pipeline multistage, in cui le fasi principali erano quelle di unitig, contig e scaffold [5]. Di queste fasi, quella unitig effettua l'assemblaggio utilizzando il grafo di de Bruijn, come visto in precedenza. Quella di contig e di scaffolding si concentrano sull'allineare le paired-end reads, e capire come orientare e fondere le unitig che si sovrappongono, inserendo eventualmente sequenze di caratteri laddove sono presenti dei gap nella coverage. La parte più onerosa computazionalmente e su cui si basa il lavoro è quella di unitig, ed è qui che ABySS 1.0 richiedeva più memoria.
Infatti, per portare a compimento questa fase bisogna caricare tutti i k-mer dalle read di input, caricarle in una hash table, conservare valori aggiuntivi per ogni k-mer che possono essere usati per le fasi successive, come ripetizioni o possibili vicini nel grafo di de Bruijn.
Per questa fase veniva utilizzata la memoria distribuita, su cluster di hardware commodity. Questo però richiedeva quantità smodate di memoria, nell'ordine delle decine di TB [5].
ABySS 2.0 utilizza invece una rappresentazione basata sul Bloom Filter, in cui, come visto in precedenza, il grafo di de Bruijn viene rappresentato all'interno del filtro.
Il funzionamento a grandi linee è questo:

- Vengono estratti i k-mers dalle reads e vengono caricati nel Bloom Filter. Essi costituiranno i nodi del grafo di de Bruijn.
  Una cosa da notare è che in questa fase vengono scartati tutti i k-mer che hanno un numero di occorrenze sotto una certa soglia, per discriminare dei k-mer che potrebbero essere frutto di errori di sequenziamento.
  Questa operazione viene effettuata utilizzando un Bloom Filter speciale, detto Cascading Bloom Filter. Un Cascading Bloom Filter è un array incatenato di Bloom Filter dove ogni filtro conserva i k-mer associati a un conteggio. Ogni bloom filter ha una capacità massima di occorrenze di uno superiore a quello precedente nella catena. Di questi Bloom Filter ne si instanziano un numero pari alla soglia desiderata. Per inserire un k-mer in un Cascading Bloom Filter bisogna effettuare una query a ogni filtro, in successione, e aggiungere il k-mer al primo filtro dove non è presente. Dopo che tutti i k-mer delle reads sono stati inseriti, l'ultimo BF della catena sarà quello che conterrà i k-mer che hanno un

certo numero di occorrenze c, dove c è la soglia scelta.
Questi k-mer saranno quelli che non verranno scartati, chiamati *solid k-mers*.

- Dopo di che vengono identificate delle reads che consistono interamente di solid k-mers, chiamate *solid reads*. Queste sono le read che poi verranno usate durante l'assemblaggio per individuare i branching points, perché sono quelle di cui ci si può fidare di più.
  Mentre Minia individuava e salvava in memoria i branching points del grafo di de Bruijn, ABySS estende le solid reads ottenute finché non arriva a un branching point, evitando quindi altre strutture aggiuntive. Le estensioni sono quelle a singola base ottenute estendendo il k-mer a destra aggiungendo una delle 4 basi.
  Questa estensione è molto semplice da visualizzare, ma può portare alla creazione di contig ridondanti, dovuti al fatto che più solid reads potrebbero essere localamente vicine nel grafo di de Bruijn. Per evitare questo problema viene usato un altro Bloom Filter di tracciamento, che registra i k-mer che sono stati già inclusi all'interno di un contig. Se ci si accorge durante l'estensione di star aggiungendo dei k-mer già presenti, non si estende ma si skippa l'estensione attuale.

- Dopo di che vengono gestiti i falsi positivi.
  Per gestirli non si usano i cFP, ma un meccanismo di look-ahead. Il meccanismo di look-ahead interviene durante l'attraversamento del grafo di de Bruijn, e ci consente di eliminare gli short branch dovuti appunto ai FP e ad eventuali errori di sequenziamento. Notare come arrivati a questo punto già i cascading bloom filter dei passi precedenti abbiano rimosso gran parte degli errori di sequenziamento, eliminando i k-mer che sono sotto la soglia di tolleranza.
  In breve l'algoritmo di look-ahead viene invocato a ogni branching point del grafo di de Bruijn, fino a una distanza di k nodi. Se questo step rivela che un branch ha una lunghezza minore o uguale di k, viene considerato un false branch e ignorato. Se il branch point ha due o più branch più lunghi di k nodi, l'estensione viene fermata.

4.2 ntHash

Una delle differenze più significative di Abyss rispetto a Minia è l'utilizzo di una funzione hash specializzata, pensata appositamente per l'uso con il Bloom Filter: ntHash.
ntHash è stata inizialmente pensata per essere una funzione ricorsiva per l'hashing di k-mers in sequenze nucleotidiche.
Si tratta di una funzione molto importante, perché l'analisi su larga scala delle sequenze nucleotidiche ha bisogno di catalogare o contare k-mer consecutivi in sequenze di DNA per i più svariati motivi, alcuni dei quali citati in questa tesina.
Il Bloom Filter stesso è una struttura dati che si basa sulle funzioni hash sottostanti, e quindi avere una funzione hash con delle buone performance può essere utile per alleggerire, in questo caso, il costo computazionale dell'assemblatore. Vedremo ora brevemente la primissima versione e andremo ad analizzare la seconda più nel dettaglio. Nel prossimo capitolo la vedremo all'opera.

4.2.1 ntHash 1

L'algoritmo proposto per ntHash prevedeva inizialmente una semplice funzione ricorsiva. Supponendo di avere k-mer per un certo k, e di avere una sequenza r di lunghezza $l > k$, la funzione f definita dagli autori di ntHash definisce il valore hash H dell'i-esimo k-mer in questo modo:

$$H(k-mer_i) = f(H(k-mer_{i-1}), r[i+k-1], r[i-1])$$

Ovvero, il valore hash dell'i-esimo k-mer dipende dal valore hash del k-mer precedente nella sequenza, dal suo primo carattere ($r[-1]$) e dal suo ultimo carattere ($r[i+k-1]$). Questa funzione viene definita *"rolling hash function"* e, dipendendo soltanto da pochissime informazioni per computare il risultato, è veramente veloce, come vedremo più avanti. Il calcolo effettivo è una variante di una funzione hash polinomiale ciclica che è in grado di calcolare i valori hash normali o canonici per i k-mer di una sequenza di DNA, utilizzando degli shift bit a bit invece che delle moltiplicazioni. Trattandosi di una funzione ricorsiva si definisce la base, cioé l'hash del primo k-mer, come:

$$H(k-mer_0) = rol^{k-1}h(r[0]) \oplus rol^{k-2}h(r[1]) \oplus ... \oplus h(r[k-1)]$$

Dove *rol* è una rotazione ciclica binaria a sinistra dove l'esponente ci dice di quanto dobbiamo ruotare, $\oplus$ è l'operatore di XOR bit a bit e $h()$ è una seed table, dove alle lettere dell'alfabeto del DNA, $\Sigma = A, C, G, T$ si fanno corispondere degli interi a 64 bit casuali. Gli hash values per i k-mer successivi vengono derivati a partire da quello di partenza in questo modo:

$$H(k-mer_i) = rol^1 H(k-mer_{i-1}) \oplus rol^k h(r[i-1]) \oplus h(r[i+k-1])$$

Ovvero, si shifta a sinistra di una posizione l'hash del k-mer precedente, ne si fa lo XOR con lo shift a sinistra di k posizioni dell'entry della seed table associata al primo carattere del k-mer corrente e ne si fa nuovamente lo XOR con l'entry della seed table associata all'ultimo carattere del k-mer corrente.

Questa funzione hash ha una complessità temporale di $O(k+l)$ [9], perché di fatto il primo step costa $O(k)$ passi, ogni step successivo viene effettuato in tempo costante perché devo effettuare solo 2 operazioni: accesso al valore di una tabella, presupposto costante e shift a sx, presupposto costante.
Gli step successivi da effettuare sono 1 per ogni successivo k-mer della sequenza, quindi si scorrerà l'intera sequenza (di lunghezza l). Per cui riepilogando: primo passo $O(k)$ e poi $O(l)$ è la somma dei restanti passi.

Dunque $O(k+l)$, che è un grande miglioramento rispetto al classico $O(kl)$ di molte hash function regolari. Non verrà approfondito in questa tesina, ma è possibile usare questa funzione anche per calcolare il valore hash per le sequenze forward e reverse and complement, attraverso una combinazione di shift e xor leggermente diversa che sfrutta una rotazione a destra invece che a sinistra.

4.2.2 ntHash 2

La versione più moderna di ntHash, che è stata oggetto di test [8], presenta delle aggiunte significative per la risoluzione di alcuni problemi. Innanzitutto, con la diffusione del sequencing high-throughput e del sequencing short-read proposto da Illumina, molte soluzioni che analizzano i k-mer hanno cominciato a popolare lo scenario globale.
Anche se risultano essere soluzioni molto versatili non riescono a distinguere fra sequenze simili, che potrebbero essere la stessa sequenza, e la cui somiglianza è dovuta al polimorfismo o a errori di sequenziamento. Per tollerare errori dovuti a situazioni di questo tipo i k-mers possono essere sostituiti dai cosiddetti spaced seeds, cioé dei pattern che definiscono quali posizioni di un k-mer debbano essere prese in considerazione e quali ignorate.
ntHash2 si propone l'obiettivo di essere uno dei primi algoritmi ottimizzati per l'hashing di spaced seeds. Infatti, se tipicamente le posizioni da ignorare vengono rimpiazzate con caratteri da ignorare e poi si usa sempre un algoritmo basato su k-mer, ntHash usa un approccio alternativo che sfrutta l'hashing ricorsivo. Non vedremo in dettaglio la variante spaced seeds perché non è l'oggetto del nostro studio, ma utilizzerà gran parte delle modifiche che vedremo tra poco. Per approfondire è presente una spiegazione dettagliata nel materiale supplementare di [6]
Rispetto alla prima versione, una prima modifica è la sosituzione dell'operazione di *rol* (shift a sx ciclico) con l'operazione di *srol*. Ciò è dovuto al fatto che $rol^{64}(x) = x$ quando trattiamo parole a 64 bit e questo crea una certa periodicità che aumenta i rate di collisioni per valori molto alti di k [6]. L'operazione utilizzata questa volta è una split rotation. La split rotation funziona in questo modo: Prima si divide x in n sottoparole di $d_1, ..., d_n$ bit tali che sommando le loro lunghezza si riottiene la lunghezza della parola di partenza $\Sigma d_i = 64$ e che il massimo comune divisore fra ogni (d_i, d_j) sia 1 per ogni i, j (coprimi), poi si ruotano le sottoparole separatamente e infine si uniscono i risultati. Questo garantisce una periodicità più lunga, che da meno problemi con valori di k molto elevati.
L'implementazione di questa funzione è descritta da questo codice:

```
1  inline uint64_t srol(const uint64_t x) {
2      uint64_t m = ((x & 0x8000000000000000ULL) >> 30) |
3          ((x & 0x100000000ULL) >> 32);
4      return ((x << 1) & 0xFFFFFFFDFFFFFFFFULL) | m;
5  }
```

Come si può osservare, gli autori utilizzano soltanto uno split in due sottoparole da ruotare, una di lunghezza 31 e l'altra di lunghezza 33, visto che il minimo comune multiplo di entrambi è 1023. Questa operazione può essere ulteriormente generalizzata aumentando il numero di sottoparole le cui lunghezze sono numeri coprimi. La periodicità massima della srol può arrivare a 2,042,040 se utilizziamo un pattern di 7 sottoparole, come visto in questa tabella riassuntiva.

Anche in questo caso abbiamo la possibilità di effettuare l'hash della normale e della reverse and complement, utilizzando l'analoga della *srol*, ovvero la *sror*.

Un'altra cosa che ntHash2 è in grado di fare, che è di vitale importanza per l'uso nel Bloom Filter, è la generazione di diversi valori hash extra. Ricordiamo che in un Bloom Filter posso avere diverse funzioni

#Splits	Period	Split Bits
-	64	64
2	1,023	31 33
3	9,660	20 21 23
4	62,985	13 15 17 19
5	306,306	9 11 13 14 17
6	855,855	5 7 9 11 13 19
7	2,042,040	3 5 7 8 11 13 17

Table 1: A ogni possibile suddivisione corrisponde un periodo sempre maggiore.

hash a guardia del filtro, e la possibilità di avere tanti valori hash diversi per lo stesso elemento usando lo stesso strumento è molto comoda e rende l'implementazione immediata. La generazione dei valori hash extra utilizza la seguente funzione.

```
inline uint64_t nte64(const uint64_t h_val, const unsigned k, const unsigned i) {
    uint64_t t_val = h_val;
    t_val *= (i ^ k * MULTISEED);
    t_val ^= t_val >> MULTISHIFT;
    return t_val;
}
```

Ulteriori ottimizzazioni riguardano il modo in cui viene velocizzato l'hashing del primo k-mer precalcolando il valore hash di ogni possibile $m - mer$ di lunghezza $m \leq 4$ in quattro 4^m array. Per formare il primo hash quindi si itera sui 4-mer del primo k-mer e si uniscono le precalcolazioni utilizzando srol e XOR. Se k non è divisibile per 4, si utilizza l'operatore di modulo.

Tutte le ottimizzazioni adottate, di cui alcune non citate in questa tesina, consentono di sopperire alla complessità aggiuntiva della split rotation, migliorando di molto l'efficienza dell'algoritmo, come visto nella prossima sezione.

Infine, la funzione è stata scritta in linguaggio C++. Da un'analisi del codice risulta che moltissime delle funzionalità core della funzione potrebbero essere implementate anche in C. Infatti non vengono utilizzate molto spesso strutture dati dinamiche aggiuntive o template. La scelta del linguaggio C++ è comunque importante. Infatti, presenta una serie di vantaggi, dalla semplicità nella gestione delle stringhe, e in generale della memoria, al controllo degli errori semplificato, e alla possibilità di encapsulare tutte le varianti della funzione in diverse classi in un approccio object oriented.
Per esempio, le stringhe di C++ mantengono informazioni sulla lunghezza e sul contenuto in un oggetto gestito dalla memoria dinamica, e questo risulta essere molto utile in alcune sezioni del codice per effettuare dei controlli a costo zero, come per esempio nel codice di inizializzazione dell'oggetto ntHash, dove viene usato per assicurarsi che la stringa su cui si sta costruendo l'oggetto abbia la giusta dimensione (file kmer.cpp della repository del progetto):

```
NtHash::NtHash(const char* seq,
               size_t seq_len,
               typedefs::NUM_HASHES_TYPE num_hashes,
               typedefs::K_TYPE k,
               size_t pos)
  : seq(seq, seq_len)
```

```
  , num_hashes(num_hashes)
  , k(k)
  , pos(pos)
  , initialized(false)
  , hash_arr(new uint64_t[num_hashes])
{
  if (k == 0) {
    raise_error("NtHash", "k must be greater than 0");
  }
  if (this->seq.size() < k) {
    raise_error("NtHash",
                "sequence length (" + std::to_string(this->seq.size()) +
                  ") is smaller than k (" + std::to_string(k) + ")");
  }
  if (pos > this->seq.size() - k) {
    raise_error("NtHash",
                "passed position (" + std::to_string(pos) +
                  ") is larger than sequence length (" +
                  std::to_string(this->seq.size()) + ")");
  }
}
```

Inoltre, C++ fornisce metodi per fare controlli relativi alla piattaforma, utili per esempio in questo caso:

```

static_assert(std::numeric_limits<uint64_t>::max() + 1 == 0,
              "Integers don't overflow on this platform which is necessary for "
              "ntHash canonical hash computation.");
```

Anche in questo è presente un'alternativa C, solo un pelo più laboriosa. Infatti se questo approccio fa parte della standard library di C++, l'equivalente C richiede l'import di "limits.h".

Infine, l'estrema velocità della funzione può essere imputata, oltre che a un uso perfetto del codice e delle risorse e a una progettazione del codice molto ordinata, a un linguaggio di per sé fulmineo ulteriormente alleggerito da una funzione che di fatto computa pochissimi valori alla volta, anche al fatto che moltissimi hash value sono precalcolati e intervengono nel calcolo dei successivi.

```
const uint64_t DIMER_TAB[4 * 4] = {
  5015898201438948509U, 5225361804584821669U, 6423762225589857229U,
  5783394398799547583U, 6894017875502584557U, 5959461383092338133U,
  4833978511655400893U, 5364573296520205007U, 9002561594443973180U,
  8212239310050454788U, 6941810030513055084U, 7579897184553533982U,
  7935738758488558809U, 7149836515649299425U, 8257540373175577481U,
  8935100007508790523U
};

const uint64_t TRIMER_TAB[4 * 4 * 4] = {
```

```
 //Valori omessi per evitare confusione
};

const uint64_t TETRAMER_TAB[4 * 4 * 4 * 4] = {
  6047278271377325800U,  6842100033257738704U,  5716751207778949560U,
  5058261232784932554U,  5322212292231585944U,  4955210659836481440U,
  6153481158060361672U,  6630136099103187130U,  7683058811908681801U,
  7460089081761259377U,  8513615477720831769U,  9169618076073996395U,
  .... // Valori omessi per evitare confusione, ma questa è la tabella più grande
};
```

Questi valori intervengono nel calcolo del base hash, che avviene per quanto riguarda il caso semplice qui:

```
inline uint64_t
base_forward_hash(const char* seq, unsigned k)
{
  uint64_t h_val = 0;
  for (unsigned i = 0; i < k - 3; i += 4) {
    h_val = srol(h_val, 4);
    uint8_t loc = 0;
    loc += 64 * CONVERT_TAB[(unsigned char)seq[i]];     // NOLINT
    loc += 16 * CONVERT_TAB[(unsigned char)seq[i + 1]]; // NOLINT
    loc += 4 * CONVERT_TAB[(unsigned char)seq[i + 2]];
    loc += CONVERT_TAB[(unsigned char)seq[i + 3]];
    h_val ^= TETRAMER_TAB[loc];
  }
  const unsigned remainder = k % 4;
  h_val = srol(h_val, remainder);
  if (remainder == 3) {
    uint8_t trimer_loc = 0;
    trimer_loc += 16 * CONVERT_TAB[(unsigned char)seq[k - 3]]; // NOLINT
    trimer_loc += 4 * CONVERT_TAB[(unsigned char)seq[k - 2]];
    trimer_loc += CONVERT_TAB[(unsigned char)seq[k - 1]];
    h_val ^= TRIMER_TAB[trimer_loc];
  } else if (remainder == 2) {
    uint8_t dimer_loc = 0;
    dimer_loc += 4 * CONVERT_TAB[(unsigned char)seq[k - 2]];
    dimer_loc += CONVERT_TAB[(unsigned char)seq[k - 1]];
    h_val ^= DIMER_TAB[dimer_loc];
  } else if (remainder == 1) {
    h_val ^= SEED_TAB[(unsigned char)seq[k - 1]];
  }
  return h_val;
}
```

5 Benchmark NtHash2, MurmurHash3, CityHash e test di Kolmogorov-Smirnov

Basandoci su [6], abbiamo deciso di imbastire una serie di test sia per vedere da vicino il funzionamento delle strutture dati osservate in entrambi gli assemblatori, sia per testarne l'efficacia. Abbiamo deciso di testare ntHash2 assieme ad alcune delle funzioni hash citate da [6] e [9]. Per i nostri scopi sono stati scelti MurmurHash3 [1], che è una funzione hash general purpose utilizzata in alcuni tool come per esempio SMASHER e Mash e CityHash [11], una funzione hash che usa un'approccio spaced seed simile a quello di ntHash2. Fin dall'inizio c'é stata una certa complessità nell'equiparare le prestazioni delle funzioni dovuta al fatto che ntHash lavora direttamente con i k-mer, mentre le altre due con una stringa generica. Il motivo di queste differenze è che come visto sopra ntHash2 adotta una serie di ottimizzazioni che però hanno senso e funzionano nel contesto di un Bloom Filter applicato alla bioinformatica, ma non al di fuori. Inoltre, abbiamo scelto il linguaggio C++ perché l'implementazione di ntHash2 è in C++ così come Murmurhash3 e Cityhash. Tutto il codice mostrato da questo punto in poi è reperibile presso [7]. La macchina utilizzata per svolgere i test è un Macbook Pro 14", con 32 GB di RAM e montante una CPU M1 Pro, le cui specifiche sono reperibili facilmente online. I primi test che abbiamo fatto riguardano la distribuzione dei valori e la performance. Per la distribuzione dei valori, si fa riferimento al file DistributionHash.cpp. Quello che facciamo è creare delle sequenze sintetiche, per il test di lunghezza pari a 250bp, randomiche, assicurandoci di non aver alcun duplicato, attraverso l'uso di questa funzione.

```
std::string generateRandomDNASequence(int length) {
    const std::string bases = "ACGT";
    std::string sequence;

    for (int i = 0; i < length; ++i) {
        int randomIndex = rand() % 4;
        sequence += bases[randomIndex];
    }

    return sequence;
}
```

In questo caso non ci interessava prelevare delle stringhe da file o avere delle stringhe particolari, perché ci interessa solo capire come sono distribuiti i valori di output della funzione all'interno dello spazio delle soluzioni, come fatto in [6].

```
for(int i = 0; i < iterations; i++){
    std::string current_sequence = generateRandomDNASequence(seqLen);
    nthash::NtHash nth(current_sequence, num_hashes, k);
    while (nth.roll()) {
        for(int i = 0; i < num_hashes; i++){
            ntHashDisOut << nth.hashes()[i] << std::endl;
        }
    }
}
```

Semplicemente a ogni step invochiamo la funzione su una stringa generata casualmente, poi la scriviamo

all'interno di un file.
Il procedimento adottato è lo stesso per ogni funzione hash. Non ci interessava che le stringhe utilizzate fossero le stesse per tutte e 3 le funzioni, perché questo test non cerca di trovare una dipendenza fra le 3. Siamo interessati solo a capire se i rispettivi output sono ben distribuiti all'interno dello spazio delle soluzioni. Qui possiamo osservare come nthash abbia un funzionamento particolare: infatti, dopo aver scelto il numero di volte in cui fare l'hash del k-mer corrente e avergli passato la sequenza, possiamo ottenere i risultati un k-mer alla volta utilizzando la funzione *roll()* che è parte dell'interfaccia della classe nthhash, fornita dal pacchetto del progetto reperibile su GitHub.

5.1 Test di Kolmogorov-Smirnov

Una volta provate le tre funzioni hash in locale si è deciso di effettuare il test di Kolmogorov-Smirnov sui valori hash generati dalle funzioni. Tale test è stato eseguito per appurare il corretto funzionamento delle funzioni hash verificando l'uniformità della distribuzione dei valori generati [9]. Il test di Kolmogorov-Smirnov (KS) è un test non parametrico utilizzato per confrontare una distribuzione campionaria con una distribuzione di riferimento, nel nostro caso una distribuzione uniforme. L'output del test più importante è il p-value. Attraverso questo valore è possibile stabilire se l'ipotesi iniziale di similarità tra le distribuzione decada o meno. Nello specifico:

- $p - value \leq 0.05$ È molto probabile che le distribuzioni siano diverse e quindi l'ipotesi iniziale decade.
- $p - value > 0.05$ L'ipotesi iniziale non viene respinta in quanto è plausibile che le due distribuzioni siano simili.

Il KS test è stato eseguito in ambiente R su tutte e tre le funzioni hash con output generati da nostre esecuzioni riportati in file `.csv`. Di seguito è riportato lo script `R` per eseguite i KS Tests

```r
#Script to evaluate uniform distribuition of the data
options(scipen = 999)

#NtHash2 ---------

#Reading and adjusting data
dataNt <- read.csv("nthashDis.csv")
colnames(dataNt)[1] <- "Values"
dataNt$Values <- as.numeric(as.character(dataNt$Values))

# Kolmogorov-Smirnov test on the data
ks_resultNt <- ks.test(dataNt$Values, "punif", min(dataNt$Values),
                       max(dataNt$Values))
print(ks_resultNt)

#---------------------------
#MURMURHASH3 TEST

#Reading and adjusting data
```

```r
dataMur <- read.csv("murmurDis.csv")
colnames(dataMur)[1] <- "Values"
dataMur$Values <- as.numeric(as.character(dataMur$Values))

# Kolmogorov-Smirnov test on the data
ks_resultM <- ks.test(dataMur$Values, "punif", min(dataMur$Values),
                      max(dataMur$Values))
print(ks_resultM)

#------------------------
#CITYHASH TEST

#Reading and adjusting data
dataCity <- read.csv("cityDis.csv")
colnames(dataCity)[1] <- "Values"
dataCity$Values <- as.numeric(as.character(dataCity$Values))

# Kolmogorov-Smirnov test on the data
ks_resultCity <- ks.test(dataCity$Values, "punif", min(dataCity$Values),
                         max(dataCity$Values))
print(ks_resultCity)

```

L'output di NTHash2 è stato il seguente:

```
    Asymptotic one-sample Kolmogorov-Smirnov test

data:  dataNt$Values
D = 0.0056338, p-value = 0.9909
alternative hypothesis: two-sided
```

L'output di Murmurhash3 è stato il seguente:

```
  Asymptotic one-sample Kolmogorov-Smirnov test

data:  dataMur$Values
D = 0.022608, p-value = 0.256
alternative hypothesis: two-sided
```

L'output di CityHash è stato il seguente:

```
   Asymptotic one-sample Kolmogorov-Smirnov test

data:  dataCity$Values
D = 0.013531, p-value = 0.8556
alternative hypothesis: two-sided
```

Da tutti e tre i test possiamo notate un $p - value > 0.05$. I valori generati dalle funzioni seguono quindi una distribuzione uniforme.

5.2 Benchmark Nthash2, MurmurHash3, CityHash

Una volta studiata e compresa la struttura di NtHash2 abbiamo deciso di effettuare alcuni Benchmark per valutarne le prestazioni all'aumentare del numero di sequenze in input e all'aumentare del numero di hash da generare per ogni K-mer. Le lunghezze delle sequenze è fissato a 250 caratteri.
Abbiamo condotto tre test per ogni funzione hash, nel dettaglio:

- Test 1: 1 Hash per K-mer — Numero di sequenze: 500k, 1M, 2M, 4M, 8M.
- Test 2: 3 Hash per K-mer — Numero di sequenze: 500k, 1M, 2M, 4M, 8M.
- Test 3: 5 Hash per K-mer — Numero di sequenze: 500k, 1M, 2M, 4M, 8M.

Ricordiamo che MurmurHash3 è una funzione Hash generica rispetto ad nTHash2 che è invece più specializzata. Questa caratteristica di MurmurHash3 ha reso necessario adattare la funzione perchè potesse essere testata in un ambiente analogo ad NtHash2. Una differenza sostanziale è che, come visto prima, nTHash2 lavora direttamente con i K-mers mentre MurmurHash3 lavora con l'intera sequenza genomica. Ciò ha reso necessario adattarne l'esecuzione affinchè lavorasse anch'essa con i k-mers estratti dalla sequenza originale. Questo vale anche per CityHash, trattandosi di un'altra hash function general purpose.

L'idea è rappresentata in questo codice:

```
for(int i = 0; i < iterations; i++) {
        // Hash all k-mers, num_hashes times
        for (int i = 0; i <= seqLen - k; ++i) {
            // Get pointer to string buffer
            std::string current_kmer = randomDNASequence.substr(i, k);
            const void* keyPtr = static_cast<const void*>(current_kmer.c_str());
                for(int j = 0; j < num_hashes; j++){
                    MurmurHash3_x64_128(keyPtr, k, 0, &hashValue);
                }
        }
}
```

Questo ciclo scorre la sequenza ed estrae in ordine i k-mer, applicando Murmurhash su un k-mer alla volta. Ovviamente scrivere il codice in questo modo rallenta molto l'esecuzione della funzione, ma questo è normale, perché è ciò che ntHash2 fa di default, ed è stato fatto per equipararne il funzionamento.
Alla linea 7 il cast viene fatto per ovviare a un problema relativo al fatto che la stringa corrente viene creata usando le stringhe di C++, ma Murmurhash3 è stato scritto per C strings.
Ognuno di questi test viene cronometrato utilizzando un cronometro ad alta risoluzione fornito dalla standard library del linguaggio C++.

```
auto start_mur = std::chrono::high_resolution_clock::now();
// codice di test
auto stop_mur = std::chrono::high_resolution_clock::now();
auto duration_mur = std::chrono::duration_cast<std::chrono::microseconds>
(stop_mur - start_mur);
```

Come si evince dal codice i test avranno i risultati espressi in microsecondi.

Di seguito sono riportati i risultati dei test prima in forma tabellare, come nella tabella 5.2 e successivamente con il grafico, come in 3, che ne mostra la curva dei tempi di esecuzione all'aumentare del numero di sequenze.

Sequenze	ntHash2	Murmur3	CityHash128
500000	590041	10478165	6837160
1000000	1172714	21411632	13783864
2000000	2354687	42532279	27640788
4000000	4726203	85038815	55412640
8000000	9453482	169883089	110815079

Table 2: num_hashes = 1, tempo di esecuzione in μs

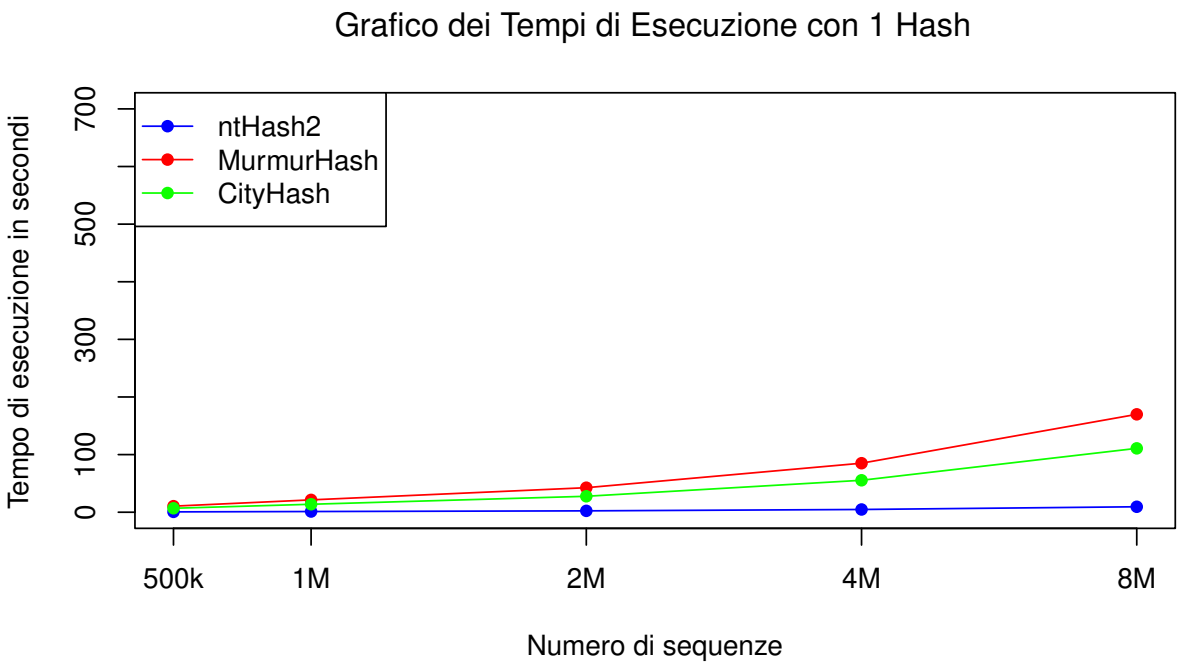


Figure 3: Grafico prestazioni con 1 Hash Function al crescere del numero di sequenze

Sequenze	ntHash2	Murmur3	CityHash128
500000	644370	25398397	13236472
1000000	1290662	50502687	26143531
2000000	2569664	100071046	52202050
4000000	5094342	201278539	105429622
8000000	10321009	405966511	211265220

Table 3: num_hashes = 3, tempo di esecuzione in μs

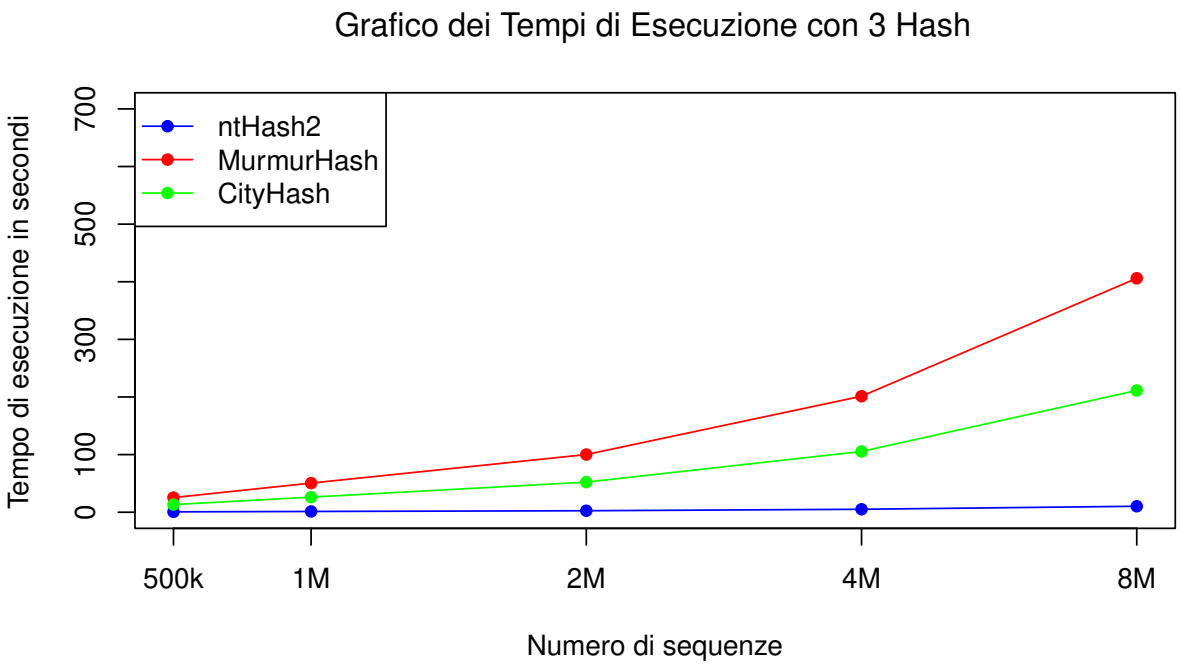


Figure 4: Grafico prestazioni con 3 Hash Function al crescere del numero di sequenze

Sequenze	ntHash2	Murmur3	CityHash128
500000	714879	38171580	20029548
1000000	1420439	76092219	40057794
2000000	2835241	152616776	80593656
4000000	5750207	308050725	162227078
8000000	11508723	613389670	323457974

Table 4: num_hashes = 5, tempo di esecuzione in μs

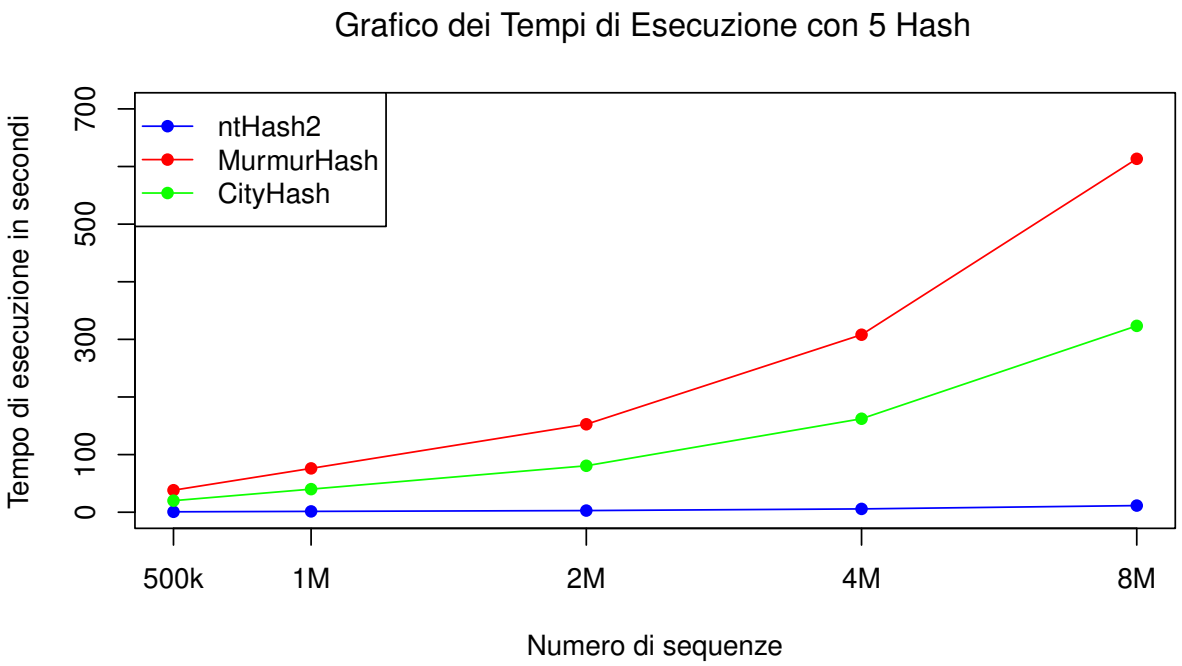


Figure 5: Grafico prestazioni con 5 Hash Function al crescere del numero di sequenze

Come è ben visibile dai grafici e con ancora più evidenza nel grafico relativo al test effettuato con cinque funzioni hash, NTHash2 è molto più prestante in termini di tempi di esecuzione delle concorrenti, risentendo molto poco dell'aumento del numero di sequenze in input così come del numero di hash da generare.
La funzione hash con peggiori prestazioni in termini di tempi di esecuzione risulta essere MurmurHash3. Ciò è dovuto con molta probabilità all'essenza della funzione stessa previamente descritta.

5.3 Implementazione e test di un Bloom Filter

Visto che ntHash2 era stata pensata per l'uso per un bloom filter, abbiamo pensato di provare a introdurne uno per capire la sua efficacia. Utilizzando sempre il linguaggio C++, abbiamo scritto un semplice filtro che però utilizza alcune particolarità del linguaggio, ovvero i Template, per poter essere istanziato con la funzione hash desiderata. Il motivo di questa scelta è la facilità di confronto tra le 3 funzioni, visto che per lo scopo del nostro test siamo interessati a capire quale sia la funzione hash col minore false positive rate fra

le 3, nel caso in cui ci sia differenza.
Per farlo definiamo genericamente cosa sia una funzione hash con un template:

```
template<typename T>
class HashFunction {
public:
    virtual size_t operator()(const T& data, size_t length) const = 0;
};
```

In questo template ridefiniamo l'operatore (), che potremmo dire sia l'operatore di invocazione, in modo da poter definire cosa bisogna passare a una funzione hash quando viene invocata. Per inostri scopi ci interessa definire il dato, espresso in modo generico perché non tutte le funzioni hash utilizzano la stessa tipologia di stringa (Stringhe C++ vs Stringhe C), e ne specifichiamo la lunghezza nel caso in cui siano stringhe C. Poi, per ognuna delle funzioni hash effettive, definiamo un nuovo template che dettaglia come debba comportarsi la funzione hash. Qui, per esempio, riportiamo quello di nTHash2:

```
template<typename T>
class NtHashFunction : public HashFunction<T> {
public:
    NtHashFunction(size_t hashes, size_t k) :
        num_hashes(hashes),
        k(k)
    {}

    size_t operator()(const T& data, size_t length) const override {
        nthash::NtHash nth(data, num_hashes, k);
        while(nth.roll()){}
        return nth.get_forward_hash();
    }

private:
    size_t num_hashes;
    size_t k;
};
```

In questo caso visto che l'interfacciamento con la funzione avviene attraverso una classe definita nella libreria fornita dagli autori stiamo di fatto creando un wrapper che verrà istanziato con le variabili necessarie a istanziare l'oggetto della classe nthash.
Per tutte le altre funzioni invece il discorso è più semplice e non servono nemmeno delle variabili per istanziare nessun oggetto interno che non sia il wrapper.

A questo punto il filtro vero e proprio:

```
template <typename HashFunc>
class Bloomfilter
{
public:

```

```
    void insert(const std::string& object)
    {
        size_t hash = hashFunc(object, object.size()) % bloomfilter_store.size();
        bloomfilter_store[hash] = true;
        ++object_count_;
    }

    void clear()
    {
        bloomfilter_store.clear();
        object_count_ = 0; // Reset the object count
    }

    bool contains(const std::string& object) const
    {
        size_t hash = hashFunc(object, object.size()) % bloomfilter_store.size();
        return bloomfilter_store[hash];
    }

    size_t object_count() const
    {
        return object_count_;
    }

    bool empty() const
    {
        return 0 == object_count();
    }

    size_t size() const
    {
        return bloomfilter_store.size();
    }

private:

    std::vector<bool> bloomfilter_store;

    size_t object_count_;

    HashFunc hashFunc;
};
```

Implementiamo l'interfaccia senza la delete, come suggerito da [10]. L'insert a questo punto è molto semplice perché non gestiamo direttamente nessuna logica specifica di nessuna hash function ma ci limitiamo

a dire: otteniamo l'hash invocando la funzione sul dato passato, mettiamo a true la casella segnalata dal filtro. Ovviamente questa implementazione molto semplice assume che l'hash si riferisca a un solo indice modulo dimensione del filtro, e non prevede più funzioni hash simultanee. Come vedremo più avanti però non dovrebbe essere molto difficile migliorarlo, e sopratuttto le prestazioni sono comunque molto buone anche su taglie molto piccole.
Questo filtro può essere facilmente invocato da un utilizzatore in un programma separato con questa sintassi:

```
1    Bloomfilter<CityHash>* cityFilter = new Bloomfilter<CityHash>(filter_size);
2    Bloomfilter<MurmurHash3>* murmurFilter = new Bloomfilter<MurmurHash3>(filter_size);
3    Bloomfilter<NtHashFunction>* ntFilter = new Bloomfilter<NtHashFunction>(filter_size);
```

Il codice di sopra non esplicita alcune cose per semplicità, reperibili in [7].
Per esempio il fatto che il filtro che usa nthash debba essere istanziato anche passando il numero di volte in cui si vuole fare l'hash dell'elemento e la dimensione del k-mer, che sono parametri necessari per inizializzare e usare nthash ma non le altre due. Ovviamente questo già fa notare come nthash sia pensato per essere usato in filtri più complessi, in cui si utilizzano più hash diversi e ci si concentra solo sulla bioinformatica. Come si può osservare dalla tabella 5, abbiamo usato dimensioni molto piccole appositamente per mostrare l'andamento e l'efficacia del filtro, che con 125 KB ha un FPR del 50%, ma già con poco meno di 1MB comincia a scendere al 20%. Test con dimensioni superiori riportano un FPR inferiore al 5%, con un risultato che è sostanzialmente identico per tutte le funzioni hash provate al netto di piccolissime differenze. Ciò non rappresenta un problema perché ntHash2 non millanta una distribuzione dei valori migliore rispetto ad altre funzioni, solo delle prestazioni molto più elevate nel contesto del dominio applicativo.

Dimensione Filtro (KB)	**CityHash**	**MurmurHash3**	**NtHash2**
125	0.489858	0.490161	0.489921
250	0.428847	0.429241	0.429109
375	0.363858	0.364049	0.363989
500	0.311659	0.311690	0.311794
625	0.271105	0.271123	0.271084
750	0.239204	0.239201	0.239259
875	0.213800	0.213971	0.213873
1000	0.193125	0.193314	0.193229

Table 5: False Positive Rate

6 Conclusioni

Dalle nostre analisi appare evidente come l'uso di una funzione hash pensata esclusivamente per l'uso bioinformatico e che possa mettere in atto delle ottimizzazioni pensate specificatamente per il dominio del problema risulta in delle performance di gran lunga superiori. Per quanto le nostre prove siano state da un certo punto di vista molto semplici, hanno mostrato immediatamente la velocità e l'efficienza di ntHash2 e dell'assemblatore ABySS rispetto alle soluzioni precedenti. Ulteriori sviluppi e ricerche potrebbero concentrarsi nell'ambito dello spaced seeds hashing e utilizzando questa base per test più approfonditi, magari su macchine diverse o utilizzato dataset reali invece che dati sintetici. La codebase può essere utile anche per poter creare dei piccoli strumenti giocattolo, da mostrare durante un corso o come esercizi per casa.

References

[1] Austin Appleby. *MurmurHash3*. Tech. rep. LiveJournal. URL: https://github.com/aappleby/smhasher.

[2] Alex Bowe et al. "Succinct de Bruijn Graphs". In: Sept. 2012, pp. 225–235. ISBN: 978-3-642-33121-3. DOI: 10.1007/978-3-642-33122-0_18.

[3] Rayan Chikhi and Guillaume Rizk. "Space-efficient and exact de Bruijn graph representation based on a Bloom filter". en. In: *Algorithms Mol. Biol.* 8.1 (Sept. 2013), p. 22.

[4] Phillip E C Compeau, Pavel A Pevzner, and Glenn Tesler. "How to apply de Bruijn graphs to genome assembly". en. In: *Nat. Biotechnol.* 29.11 (Nov. 2011), pp. 987–991.

[5] Shaun D Jackman et al. "ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter". en. In: *Genome Res.* 27.5 (May 2017), pp. 768–777.

[6] Parham Kazemi et al. "ntHash2: recursive spaced seed hashing for nucleotide sequences". In: *Bioinformatics* 38.20 (Aug. 2022), pp. 4812–4813. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btac564. eprint: https://academic.oup.com/bioinformatics/article-pdf/38/20/4812/46535020/btac564.pdf. URL: https://doi.org/10.1093/bioinformatics/btac564.

[7] Francesco Maddaloni and Pietro Negri. *abyss-e-bloom-filter*. Tech. rep. Università degli Studi di Salerno, 2024. URL: https://github.com/blazQ/abyss-e-bloom-filter.

[8] Francesco Maddaloni and Pietro Negri. *Il fogliettino*. 2024. URL: https://drive.google.com/file/d/1qFm1wNjbU1bdE6DjadeBaqxFWhqGhaAj/view?usp=sharing.

[9] Hamid Mohamadi et al. "ntHash: recursive nucleotide hashing". In: *Bioinformatics* 32.22 (July 2016), pp. 3492–3494. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw397. eprint: https://academic.oup.com/bioinformatics/article-pdf/32/22/3492/49027120/bioinformatics_32_22_3492.pdf. URL: https://doi.org/10.1093/bioinformatics/btw397.

[10] Sabuzima Nayak and Ripon Patgiri. "A Review on Role of Bloom Filter on DNA Assembly". In: *IEEE Access* 7 (2019), pp. 66939–66954. DOI: 10.1109/ACCESS.2019.2910180.

[11] Geoff Pike and Jyrki Alakuijala. *Introducing CityHash*. en. Tech. rep. Google, 2011. URL: https://opensource.googleblog.com/2011/04/introducing-cityhash.html.