UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Analisi comparativa degli algoritmi di compressione spaziale di k-mer sets

Relatore: Prof. Comin Matteo

Laureando: Trettenero Enrico

ANNO ACCADEMICO 2023-2024

Data di laurea 23/09/2024

Abstract

Un problema noto nell'ambito della bioinformatica consiste nell'efficiente memorizzazione dei dati ottenuti dalla fase di sequenziamento di un particolare genoma, infatti la lunghezza della sequenza grezza rappresenta un importante ostacolo sia alla memorizzazione che all'elaborazione di essa.

Una prima fase di compressione deriva dall'utilizzo dei noti *k-mers* ossia sottosequenze di lunghezza k decisa a priori, derivate da una lettura di dimensione maggiore.

L'obiettivo di questa tesi consiste nel presentare e confrontare nuove tecniche utilizzabili per comprimere al meglio codesti *k-mer sets*, in correlazione all'occupazione spaziale, il costo computazionale di compressione ed il costo computazionale di utilizzo ed analisi del dato compreso in relazione alle finalità su di esso, ricercando l'algoritmo più adatto e analizzando come le prestazioni mutano in funzione della variazione del parametro k all'interno nelle sottosequenze.

Indice

Capitolo 1

Introduzione

1.1 DNA

Il *DNA* come generalmente si conosce è la struttura fondamentale alla base degli organismi viventi, la sua successione infatti specifica tutte le istruzioni genetiche necessarie per lo sviluppo ed il mantenimento della vita, motivo per il quale è oggetto di estremo interesse studiarne e decifrarne i segreti.

La sua scoperta e complessa natura è ciò che ha principalmente portato alla creazione di una recente branchia di studio nell'ambito tecnologico: la *bioinformatica*.

1.1.1 Struttura

La struttura del *DNA* consiste in un filamento a doppia elica il cui scheletro è costituito da zucchero ed un gruppo fosfato, i quali fanno da collante per le basi azotate; codeste basi sono quattro e sono conosciute come **Adenina, Timina, Citosina e Guanina [1]** che da ora in avanti verranno indicate come $\{A, T, C, G\} \in D$ le quali saranno di vitale importanza all'interno dell'analisi. Infatti la sola combinazione lineare di esse permette la formazione della sequenza genetica di un qualsiasi organismo.

Il principale problema di questa struttura consiste nella sua lunghezza, infatti il solo genoma[1] umano conta una numerosità di oltre 3 miliardi di *bp*[2] motivo per il quale è necessario sequenziarlo, immagazzinarlo ed analizzarlo in modo efficiente.

[1] Un **genoma** è un insieme di geni che rappresentano la sua impronta genetica, nel caso umano un gene è formato da una molecola di *DNA* mentre in un virus la molecola è di *RNA*.

[2] Il termine *bp* è utilizzato come acronimo di **una coppia di basi**, nota appunto la struttura a doppia elica del filamento di *DNA* le basi si rilevano solamente a coppie, le quali uniche ammesse sono $A - T$ e $G - C$.

1.1.2 Acquisizione

A partire dagli anni 2000 una nuova tecnologia ha permesso di rivoluzionare la complessa e costosa fase di sequenziamento, il **Next Generation Sequencing**[2] implementato nelle varie piattaforme quali: *Illumina, Ion Torrent, Oxoford Nanopore* e molte altre. Esso permette un sequenziamento parallelo il quale risulta essere più preciso e veloce, consentendo di restringere i tempi e ridurre gli errori, risultando inoltre relativamente economico date le minori risorse utilizzate.

Procedimento

Il processo di sequenziamento può essere suddiviso in **quattro** fasi principali:

1. **Preparazione**: Data la sua lunghezza e fragilità, oltre al fatto che non è possibile definire con precisione un punto di inizio, la lettura, al fine di sequenziare l'intero genoma deve essere totale, di conseguenza si procede ad una prima fase di **frammentazione**, ossia si suddivide la sequenza primaria in frammenti di lunghezza prefissata con valore espresso in *bp* (tendenzialmente fra i 300 e 500).

2. **Replicazione**: Successivamente si procede ad **amplificare**[3] i vari frammenti al fine di creare molteplici copie identiche, permettendo l'esecuzione di una massiva lettura parallela.

3. **Sequenziamento**: Ogni singolo frammento viene sequenziato[4] per determinare le basi dell'insieme D 1.1.1 che lo compongono, è importante notare che la fase di lettura di ogni frammento è sequenziale ma con inizio differente (vedi figura 1.1), il che crea sovrapposizioni tra le letture, le quali intersecandosi permettono tra i vari vantaggi una individuazione e correzione degli errori dovuti ad un sequenziamento errato, ma non assicurano la lettura completa fino alla fase successiva.

   Si introduce quindi il *quality score* associato ad ogni frammento letto, nella forma $MAPQ = -10 * log_{10}(P_{map\_wrong})$ dove P_{map_wrong} indica la possibilità di errore.

4. **Assemblaggio**: I singoli frammenti vengono quindi combinati utilizzando sottosequenze corrispondenti al loro interno, venendo uniti dove presenti sovrapposizioni, successivamente vengono assemblati i **Contigs** 1.2.1 ed eseguita una fase di correzione errori e validazione del risultato col fine di ricreare, se possibile, la sequenza originale.

[3] Con amplificare si intende clonare in varie copie i frammenti di *DNA*.
[4] Con il termine sequenziato si intende leggere e comprendere le basi che compongono un frammento.

Figura 1.1: Esempio di sequenziamento tramite *NGS*, in rosso le letture eseguite

Risultato

Come è possibile notare il numero di dati ricavati prima della fase di assemblaggio non soltanto è svariate volte più elevato della lunghezza iniziale del genoma, arrivando ad occupare uno spazio di svariati Terabyte, ma deve essere ulteriormente elaborato al fine di ricreare la sequenza desiderata, il quale processo può richiede molteplici ore o giorni nel peggior dei casi.

Risulta quindi fondamentale utilizzare algoritmi che permettano un elevato rapporto di compressione sfruttando la struttura topologica dell'insieme D 1.1.1 e nello stesso momento permettendo una rapida ricostruzione della sequenza (quindi una rapida lettura ed un rapido inserimento all'interno dell'insieme già compresso) oltre che consentire una veloce elaborazione su i dati ottenuti.

1.2 Introduzione alla compressione

In questa sezione si introducono i concetti ed i costrutti fondamentali sui quali sono basati i successivi algoritmi di compressione.

1.2.1 K-Mers

I **K-mers**[3] sono sottostringhe continue di lunghezza **k** derivate da una sequenza o frammento, il quale insieme crea un **k-mer set**, una stringa può contenere più volte lo stesso *k-mer*.

Il procedimento di estrazione è il seguente:

Data una stringa S vengono estratti i primi k caratteri, i quali formeranno il *k-mer*, dalla stringa S viene quindi rimosso il carattere in prima posizione ottenendo $S^* = \text{suffix}_{k-1}(S)$[5], il processo si ripete in maniera ciclica fino alla condizione critica di $|S| = k$ dopo la quale, una volta formato l'ultimo *k-mer* l'algoritmo termina (vedi fig 1.2).

Mentre se la sequenza in ingresso ha come dimensione $|S| < k$ la sequenza viene scartata.

Inoltre da analisi sperimentali si è notato come in un genoma la numerosità dei *k-mers* tende ad essere simile, ossia il conteggio di *k-mers* uguali all'interno di una sequenza tende ad essere un valore $x - \sigma$ dove σ è la varianza.

Il che permette di ipotizzare, per valori inferiori di una determinata soglia di rilevanza, ottenuta da informazioni precedenti sul presente genoma, che un *k-mer* con molteplicità ridotta non appartenga alla sequenza in questione e di conseguenza si tratti di un errore di sequenziamento.

Interessante notare come l'introduzione della rappresentazione in *k-mers* senza un opportuna tecnica di compressione peggiora notevolmente le prestazioni spaziali, infatti per immagazzinare su file una sequenza S scomposta con *k-mers* di lunghezza k viene occupato uno spazio di $(|S| - 2) * k$ caratteri a differenza degli $|S|$ della stringa originale.

Varianza del *k-mer*

Un chiaro quesito consiste nella scelta di un valore ragionevole di k, per valutare come questo parametro vari il risultato della rappresentazione si procede considerando i cambiamenti con valori di k portati agli estremi.

- **k minimo**: La seguente formula indica il numero di *k-mers* massimi distinti che un *k-mer sets* può contenere in funzione di k: $|K| = 4^k$, dove K è il *k-mer set* in questione.

[5] Con $suffix_x$ si intende una nuova stringa di lunghezza x ottenuta partendo dal carattere di indice $|S| - x$ compreso fino a terminazione.

Figura 1.2: Esempio di estrazione di *k-mers* da una stringa S con $k = 3$

Si ottiene quindi utilizzando il valore minimo pari ad 1 un massimale di 4 *k-mers* distinti che rappresentano chiaramente l'insieme D 1.1.1, i quali avranno un conteggio dei singoli estremamente elevato non fornendo alcuna informazione utile sulle combinazioni intrinseche del genoma oltre a rendere estremamente difficile la fase di compressione data la mancanza di costrutti similari da poter semplificare.

- **k elevato**: Come è facile intuire utilizzare un valore k superiore al numero di *bp* scelto per il sequenziamento dei frammenti non sarebbe accettabile, come presentato nell'algoritmo di estrazione dei *k-mers*, di conseguenza si considera come valore elevato un k tendente a *bp*.

  Nel quale caso si otterrà un elevata maggioranza di *k-mers* univoci i quali rendono l'insieme K particolarmente difficile da comprimere poiché la struttura che contiene l'indice dei *k-mers* risulterà molto voluminosa, facendo risultare la conversione in *k-mers* un passaggio poco utile rispetto a comprimere direttamente le singole letture.

Come descritto il valore k permette di alterare sostanzialmente le prestazioni di un determinato algoritmo, motivo per il quale è importante studiarne i valori più adatti al fine di garantire la miglior performance possibile in base alla struttura che si è deciso di utilizzare.

Contig

Abbreviazione di *contiguos sequence*[4], i **contigs** rappresentano una sequenza continua di *DNA* assemblata da vari frammenti, i quali realizzano una porzione del genoma ma non l'intera lunghezza.

Sono quindi un prodotto intermedio che fornisce informazioni aggiuntive e consente ulteriori fasi di compressione, la loro qualità si indica con Nx con $0 < x < 100$, infatti non sono esenti da possibili errori dovuti ad un assemblaggio errato.

1.2.2 Utilizzo dei K-Mers

Nonostante l'ulteriore complessità che viene introdotta, il corretto uso dei *k-mers* fornisce importanti vantaggi in molteplici applicazioni, il che li rende di estremo valore nel contesto della *bioinformatica*, nel dettaglio sono elencati i diversi utilizzi più diffusi:

- **Conteggio dei *k-mers***: Come suggerisce la nomenclatura consiste nel conteggio complessivo della numerosità dei *k-mers* all'interno di una sequenza, si inizia col dire che si tratta normalmente di una fase preliminare nell'analisi poiché il conteggio dei *k-mers* può fornire importanti informazioni, le quali sono alla base del processo di correzione preventiva degli errori dovuti alle successive letture approfondite, inoltre forniscono informazioni per un assemblaggio parziale del genoma insieme ad una stima delle sue dimensioni, provvedono ad un supporto all'alienamento di sequenze ed un aiuto alla classificazione della lettura in un *dataset* più ampio. Infatti il conteggio dei *k-mers* facilità l'identificazione di *pattern*[6] tipici che possono essere ritrovati in altri genomi, poiché genomi comuni tendono a condividere uguali *k-mers* con simile conteggio. Alcuni algoritmi che implementano queste funzioni sono: *KMC3* e *Sequeakr*, inoltre nel dettaglio verranno presentati due nuovi algoritmi adatti a questo scopo.

- **Analisi delle frequenze**: La distribuzione di conteggi dei *k-mers* fornisce importanti informazioni, permettendo di caratterizzare una lettura tramite l'analisi dello **spettrogramma**, con il quale si intende un grafico con ordinata il conteggio dei *k-mers* e ascissa i *k-mers* univoci in ordine di lettura.

  In particolare grazie all'analisi dello *spettrogramma* è possibile, in base alla distribuzione dei valori massimi del grafico, differenziare la lettura ottenuta in due categorie:

  - **Unimodale** ossia il grafico presenta la maggioranza dei massimali in prossimità fra loro, tipico di specie batteriche.
  - **Multimodale** che invece presenta massimali sparsi, che sono tipici di altri *metagenomi*.

  Inoltre analizzando il posizionamento della distribuzione all'interno del grafico è possibile rilevare e filtrare contaminazioni avvenute sulla sequenza, comparandola con un database di riferimento, oltre alla possibilità di individuare eventuali mutazioni genetiche. Il che inoltre risulta utile per sia per ottimizzare il processo di assemblaggio che per valutare la qualità di un genoma assemblato. Alcuni algoritmi che supportano l'analisi spettrografica sono GenomeScope e KHMer.

[6] Gruppi o strutture similari che si ripetono in diverse sequenze

- **Allineamento delle sequenze**: Un altro dei principali utilizzi dei *k-mers* è identificare all'interno delle varie sequenze componenti sovrapponibili tra loro al fine di consentire l'allineamento ottimale fra le varie letture, all'interno dei quali algoritmi i *k-mers* trovano utilità nell'obiettivo di ridurre sia la memoria che la potenza richiesta per la computazione. Supportato ulteriormente dal fatto che spesso è sufficiente analizzare solamente un ridotto sottogruppo al fine di ottenere dei risultati di massima, ed i *k-mers* supportano perfettamente questa casistica.

- **Assemblaggio del genoma** Si tratta del processo finale di ricostruzione della sequenza iniziale di *DNA* e di conseguenza rappresenta la fase più delicata, la quale può essere soggetta a errori dovuti a sequenze incoerenti, regioni ripetute e variazioni strutturali, in particolare i *k-mers* trovano applicazione in entrambe le due grandi metodologie di assemblaggio utilizzate, le quali sono: metodologia **reference-based** che ricalca un genoma di riferimento per la ricostruzione e metodologia **de novo** che ricostruisce la sequenza da zero, normalmente tramite un *grafo di De Bruijn*.

- **Classificazione della tassonomia e metagenoma**: Vengono utilizzati in combinazione i precedenti metodi basati sui *k-mers* al fine di identificare le differenze tassonomiche rispetto a diversi ambienti in associazione alla diversità, complessità e qualità dei dati, nei quali i *k-mers* vengono utilizzati per la comparazione dei *dataset* di riferimento come avviene in Kraken2 oppure con strumenti quali VirFinder.

Inoltre i *k-mers* trovano ampio utilizzo anche nell'analisi della struttura proteica di un campione.

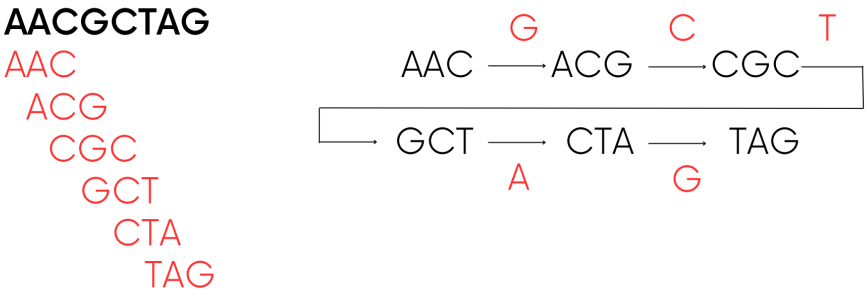


Figura 1.3: Esempio di creazione di un grafo di De Bruijn, con a destra la successione dei nodi con i relativi archi di collegamento

1.2.3 Grafo di De Bruijn

Si tratta di un grafo orientato[5] rappresentante una sequenza genomica in cui ogni *k-mer* è associato ad un nodo del grafo.

Si indica con V l'insieme di nodi e con E l'insieme degli archi che li interconnettono, un arco fra due nodi v_1, v_2 si crea solo se $\text{suffix}_{k-1}(v_1) = \text{prefix}_{k-1}(v_2)$, ossia solo se i due nodi sono sovrapponibili a meno in un carattere, in questo caso si costruisce l'arco e etichettandolo con il carattere aggiunto per rappresentare v_2, come è facile intuire un nodo può essere collegato a più nodi, portando in alcuni casi alla creazione di un ciclo, in particolare per valori di k ridotti.

Il presente grafo rappresenta una delle strutture base per l'elaborazione e memorizzazione dei *k-mers*, infatti conosciuta la sua configurazione è possibile ricostruire interamente la sequenza di partenza.

Unitigs

Un **Unitig** è una sequenza estratta da un *grafo di De Bruijn* che rappresenta un percorso all'interno di esso, è quindi formato da vari archi e nodi connessi fra loro.

Essendo un cammino ogni nodo, fatta eccezione per il nodo di inizio e di fine, ha un solo nodo connesso in entrata ed un solo nodo connesso in uscita, di conseguenza non sono presenti diramazioni, a differenza dei *conting* (infatti sono normalmente più corti) il che assicura che gli *unitigs* non siano ambigui, motivo per il quale è possibile ricostruire i *k-mers* di partenza conoscendo il relativo *unitig*.

Grafo di De Bruijn Compatto

Il *grafo di De Bruij compatto* è una variante nella quale i nodi non sono più formati da *k-mers* ma da *unitigs*, i quali sono, come da definizione, sono formati da diversi *k-mers* permettendo per conseguenza di ridurre il numero di nodi all'interno del grafo, diminuendo la memoria necessaria

e migliorando le prestazioni, inoltre date le proprietà degli *unitigs* è possibile ricostruire i *k-mers* di partenza senza perdere in generalità.

Grafo Hamiltoniano

A differenza del normale *grafo di De Bruijn* la sua variante **Hamiltoniana** ha come obiettivo la costruzione di un singolo ciclo nei quali i nodi sono attraversati una sola volta, ossia un *k-mer* appare unicamente all'interno del percorso fatta eccezione se il *k-mer* si trova contemporaneamente all'inizio e fine che per definizione da origine ad un ciclo.

1.2.4 Tecnologie e Prodotti utilizzati ad oggi

Si introducono gli strumenti ad oggi di uso più comune, i quali costituiscono sia la base di confronto per i nuovi metodi che verranno presentati nei successivi capitoli, sia in alcuni casi componenti fondamentali di essi.

BCalm 2

Sviluppato da *Rayan Chikhi, Antoine Limasset e Paul Medvedev*[6] arrivato alla seconda versione, produce un *grafo di De Bruijn compatto* 1.2.3, il quale utilizza *unitgs* al posto dei *k-mers*, implementando nativamente un filtro per ridurre *k-mers* di bassa cardinalità ed il suo prodotto è appunto un file di tipo *Fastqa* con all'interno il grafo compatto. Si tratta di un algoritmo molto diffuso, semplice e leggero che opera nella fase preliminare di elaborazione dei dati, il quale è anche di utilizzo come fondamenta di alcuni algoritmi successivamente presentati.

CuttleFish 2

Diretto successore di *BCalm*, *CuttleFish*[7] fornisce prestazioni superiori e migliore scalabilità in particolare su sistemi ad alte prestazioni, aumenta la qualità dell'assemblaggio tramite un approccio ibrido, oltre a lavorare molto meglio con genomi più complessi rispetto a *BCalm*, a discapito di un maggior consumo di memoria temporanea, il prodotto in uscita è sempre un grafo di *De Bruijn Compatto*.

UST

Unitig STich sviluppato da *Rahman e Medvedv*[8] ha come obiettivo la compressione di un *grafo di De Bruijn compatto* effettuando una compressione di tipo *lossless*[7] al fine di ridurre ulteriormente lo spazio occupato. Il funzionamento generale è di semplificare la struttura del grafo in ingresso generando nuovi *unitigs* di lunghezza superiore rispetto a coloro già contenuti nel grafo, cercando contemporaneamente di mantenerne basso il numero, la complessità deriva dall'estendere il più possibile ogni *unitig* affinché rimangano univoci, il costo chiaramente è in tempi di esecuzione più elevati ma permette un considerevole risparmio in termini di archiviazione.

BiFrost

Sviluppato da *Guillaume Holley Páll Melsted* [9], *BiFrost* opera su *grafi di de Bruijn colorati* 2.1.1 attraverso una costruzione incrementale, utile per *datasets*[8] in continua evoluzione, inol-

[7] Come suggerisce il nome, senza perdita di qualità a differenza della compressione di tipo *lossy*.
[8] insieme di genomi

tre implementa nativamente strutture come *Bloom filter* 2.4.1 ed assicura rapide prestazioni di accesso con ottima scalabilità.

Algoritmi General Porpouse

A differenza degli algoritmi precedentemente presentati, codesti sono generalmente usati per la compressione di file di testo generici, si basano su procedimenti come la codifica di *Huffman*[10] oppure tecniche a dizionario, i quali permettono di aggiungere un ulteriore strato di compressione al risultato un algoritmo specifico precedentemente utilizzato. Alcuni di questi sono: *LZ4, Deflate, Snappy, LZO, LZMA*[11], la principale controindicazione consiste nell'accesso dei dati poiché devono essere decompressi completamente o in parte a seconda dell'algoritmo per essere poi analizzati.

Capitolo 2

Presentazione Nuove Tecniche

Di seguito verranno illustrate nel dettaglio nuove metodologie che possono essere adottate per la compressione di un *k-mer set*, importante specificare che le varie tecniche hanno diversi obiettivi e finalità, nel dettaglio verranno trattati due tipologie di approcci:

- Tecniche basate sul *grafo di De Bruijn*.
- Tecniche basate sul conteggio dei *k-mers* tramite un dizionario.

Le quali sono estremamente correlate e possono lavorare in sinergia.

2.1 GGCAT

2.1.1 Introduzione

Sviluppato da *Andrea Cracco e Alexandru I. Tomescu*[12] il seguente algoritmo si pone l'obiettivo di costruire un *grafo di Bruijn compatto* 1.2.3 partendo da un'opportuna lettura, il quale è richiesto come componente fondamentale di svariati algoritmi successivi, in particolare è in grado di generare anche un grafico di **De Bruijn colorato** 2.1.1, inoltre l'algoritmo rimane esatto (ossia è possibile ricreare con precisione l'intera lettura di partenza) fino a $k < 65$, successivamente possono verificarsi collisioni causate dalla funzione di hash *Rabin-Karp* a 128bit utilizzata, il che lo rende comunque adatto ad un ampio spettro di utilizzi, infatti è progettato per ambienti ad elevata capacità di calcolo e parallelizzazione, in aggiunta per pieno supporto le *API*[1] di interfacciamento sono scritte sia in *Rust* che *C++*[2].

[1] Interfaccia che mette a disposizione delle funzioni per accedere ad una funzionalità più complessa in maniera semplice.

[2] Linguaggi di programmazione a basso livello estremamente performanti.

Grafo di De Bruijn Colorato

Variante del grafo di *De Bruijn*, è costruito a partire da una collezione di *data sets* (varie sequenze genomiche), per ogni *k-mer* il grafo salva un identificatore comunemente detto **colore** derivato dal *data set* in cui appare, al fine di identificare se il *k-mer* in questione appartiene al genoma. Un *k-mer* all'interno del grafo può quindi appartenere a più *data sets* e di conseguenza avere più *colori* contemporaneamente.

2.1.2 L'algoritmo

Introduzione

Come classica implementazione del *GDBC*[3] l'idea alla base consiste nel comprimere il grafo sostituendo i *k-mers* con *Unitigs* 1.2.3 equivalenti, in questo caso fino al raggiungimento di un **unitig massimale**, ossia un *unitig* che non può essere esteso di un singolo nodo senza perdere le sue proprietà. L'obiettivo è quindi ottenere tutti gli *unitig massimali* di un grafo R.

Definizioni

Canonical Maximal Unitigs Un insieme U è definito come l'insieme degli **unitigs massimali canonici** per un insieme R di stringhe, con dimensione k per i *k-mers* e con soglia di abbondanza a, il quale deve rispettare le seguenti proprietà:

1. Ogni stringa di U deve essere lunga almeno k e non ci possono essere stringhe duplicate o palindrome.
2. Ogni *k-mer* di R con molteplicità di almeno a appartiene ad U.
3. Se un *k-mer* appare più di una volta in U esso appare solo come prefisso o suffisso delle stringhe in U.
4. Gli *unitigs* in U sono massimali e quindi non possono essere ulteriormente estesi.

Procedimento

Gli *unitigs* vengono costruiti in modo incrementale partendo dai *k-mers* e tentando l'espansione in entrambi le direzioni (nodi in ingresso e nodi in uscita) affinché rimangano tali, il primo passaggio è il seguente.

[3] Grafo di De Bruijn Compatto

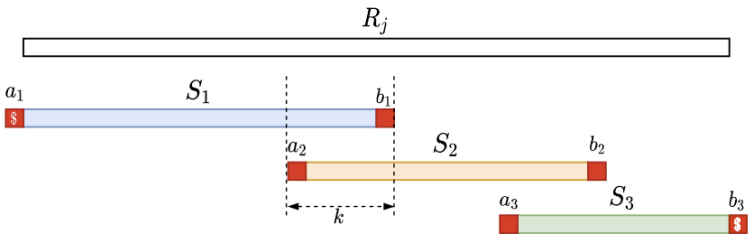


Figura 2.1: Esempio di sottostringhe sovrapposte con caratteri di collegamento associati

Read splitting Ogni lettura R proveniente da *NGS* viene frammentata in sottostringhe, le quali sovrapposte hanno in comune $k-2$ caratteri, in modo che tutti i *(k-1)-mers* abbiano in comune lo stesso *minimizer* 2.4.1, in particolare la funzione di *hash* usata per il *minimizer* è *ntHash*[13]. Vengono quindi individuati due caratteri a, b che rispettivamente precedevano e seguivano la sottostringa nella lettura R, i quali sono utilizzati per collegare fra loro le varie sottostringhe, infatti sono denominati come **caratteri di collegamento**, stringhe collegate fra loro sono identificate come parte di un gruppo G il quale ha le seguenti proprietà:

- Un *k-mer* appare in un solo gruppo rispetto alla lettura R.
- Il numero di occorrenze di un *k-mer* nella lettura corrispondono alle occorrenze all'interno del gruppo.
- Se un *minimizer* senza caratteri di collegamento ha una corrispondenza di $k-1$ caratteri con qualche *k-mer*, allora quel *k-mer* appare nel gruppo.

Costruzione degli unitigs intermezzi Si procede come secondo passaggio alla costruzione dell'*intermezzo unitig*, in questo momento si conosce che ogni *k-mer* può essere ottenuto semplicemente interrogando il gruppo G di appartenenza, su ogni gruppo si procede contando i *k-mers* con una tabella *hash* contrassegnando coloro che presentano caratteri di collegamento, aumentandone il conteggio all'interno della *hashmap*. Se i *k-mers* soddisfano il conteggio minimo richiesto dalla soglia di abbondanza a si cerca l'estensione sia a destra che sinistra (per estensione si intende aggiungere un suffisso dell'insieme D e valutare se è presente all'interno del gruppo) fino alla formazione di un *unitig*, se l'estensione viene interrotta da un carattere di collegamento lo *unitig* viene salvato all'interno della lista. Per ogni *unitig* inserito all'interno della lista vengono memorizzati in un'altra lista L i *k-mers* che iniziano e terminano lo *unitig* in questione.

Unitgs Merging Si procede come ultimo passaggio ad unire *unitigs di intermezzo* al fine di renderli **massimali**, si tratta del procedimento più delicato data la complessità nella parallelizzazione di questo passaggio, la soluzione individuata è adottare un approccio randomizzato.

Gli *unitigs* vengono prelevati tramite associazione dalla struttura L, dai quali viene creata una nuova lista di coppie denominata P, la quale all'interno delle coppie contiene gli identificatori derivati dalla lista L degli *unitigs* che hanno uguali *k-mer* di inizio o fine.

Si procede estraendo in maniera randomica gli elementi da P i quali vengono trasferiti all'interno di vari *buckets*, nei quali gli elementi vengono ordinati e si procede ad unire gli *unitigs di intermezzo* finché non è più possibile rilevare elementi sovrapponibili fra loro.

Se il *k-mer* terminale dello *unitigs* generato non è presente in altri *bucket* allora il *bucket* viene definito come *sigillato* ed il processo è completo.

Colorare il grafo

Si procede, opzionalmente, a computare i colori per ogni *k-mer* del grafo, i principali problemi sono tenere traccia di tutti i **colori** che appartengono ad ogni *k-mer* e salvare codesti colori con prestazione.

L'idea principale dell'algoritmo consiste nell'unire le informazioni dei colori per ogni *k-mer* che condivide lo stesso *set* di colori, la lista di essi viene tracciata osservando la sorgente si genera quindi un *hash* a *128bit* che contiene la lista di colori a cui un *k-mer* appartiene e viene inserito all'interno di un dizionario, se un nuovo *k-mer* con un nuovo *set* di colori non è presente all'interno del dizionario si aggiunge alla *mappa colori*, se presente invece lo si contrassegna con il valore *hash* corrispondente al determinato *set* di colori all'interno del dizionario.

Questo procedimento assicura una compatta rappresentazioni dei colori correlati al *k-mer*, inoltre la *mappa colori* essendo un file esterno viene ulteriormente compressa con algoritmo *LZ4* 1.2.4.

Query

Al fine di ottimizzare al meglio la ricerca all'interno del *GDBC* le interrogazioni sono eseguite dividendo il grafo in *unitigs* che insieme alle *query*[4] sono inseriti in *buckets*, similmente all'idea base dell'algoritmo, per ogni *bucket* viene eseguito il conteggio dei *k-mers* che soddisfano le condizioni della query, successivamente tutti i risultati dei vari conteggi per ogni *bucket* vengono sommati col fine di ottenere come risultato finale l'esatto numero di *k-mers* che rispettano le specifiche dell'interrogazione fornita in ingresso.

[4]Definizione di interrogazione, richiesta di un determinato tipo di dato ad una struttura.

2.2 USTAR

2.2.1 Introduzione

Acronimo di *Unitig STitch Advanced constRuction*, sviluppato da *Enrico Rossignolo e Matteo Comin*[14] si tratta di una tecnica avanzata di esplorazione di un *grafo di De Bruijn compatto*, che promette prestazioni migliori di **UST** 1.2.4 e tools similari riducendo le dimensioni del prodotto finale.

2.2.2 Idea

Il costrutto di partenza è un *grafo di De Bruijn compatto*, ottenibile tramite vari strumenti, tra i quali *BCALM2* 1.2.4 il quale utilizza *unitigs* al posto dei *k-mers*, il primo passaggio di *USTAR* consiste nel semplificare ulteriormente i nodi ed archi del grafo calcolando i *Simplitigs* con una funzione euristica, successivamente esattamente come *UST* 1.2.4 viene selezionato un **seed node** dal quale si cerca un percorso di nodi adiacenti che possono essere espansi affinché tutti i nodi non sono raggiunti, tuttavia al fine di selezionare il miglior *seed node*, data la natura non uniforme della distribuzione dei conteggi (la quale rende i valori ad alto conteggio difficili da comprimere) *USTAR* seleziona come *seed* il nodo che ha il conteggio medio più elevato utilizzando la connettività del grafo, il quale porta un miglioramento nella compressione.

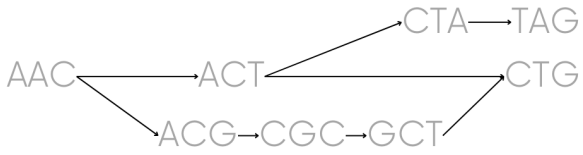
2.2.3 Definizioni

Lunghezza Cumulativa Comprimere un set K di *k-mers* può essere visualizzato come il problema di rappresentare $S \in M$ formato da stringhe di lunghezza qualsiasi, tali che il set M di sottostringhe di lunghezza k sia equivalente a K.

È inoltre dimostrato che la **lunghezza cumulativa**[5] quando non sono presenti *k-mers* duplicati equivale a $|K| + (k-1) * |S|$, da il quale si nota che l'obiettivo è minimizzare il numero di stringhe dell'insieme S, ossia trovare il numero minimo di percorsi disgiunti che coprono l'intero grafo; tuttavia si tratta di un problema estremamente complesso di categoria *NP-hard*[15], gli algoritmi attuali infatti scelgono un nodo arbitrario da cui partire.

L'obiettivo primario di *USTAR* è quindi ridurre al minimo la lunghezza cumulativa dei percorsi all'interno del *GDBC*.

[5] con lunghezza cumulativa si intende la somma della cardinalità di ogni stringa



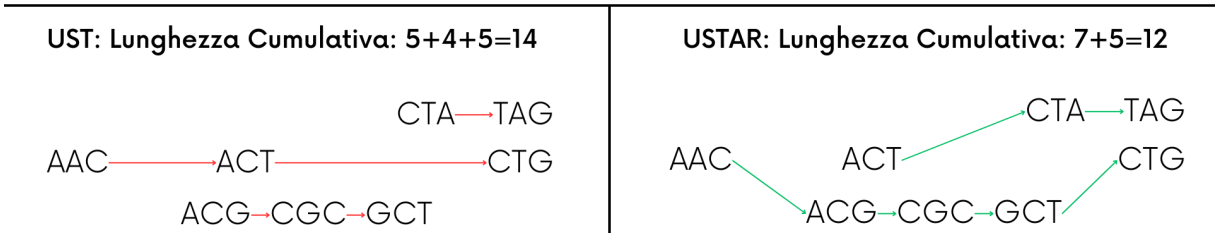


Figura 2.2: Confronto nel calcolo della lunghezza cumulativa tra *UST* in rosso ed *USTAR* in verde su un grafo comune, il valore indicato nella somma corrisponde alla lunghezza del percorso

2.2.4 Algoritmo

USTAR a differenza di *UST* utilizza la connettività del grafo implementando una **funzione euristica** per calcolare i *simplitigs*[6], (i quali è stato dimostrato avere prestazioni migliori rispetto ai soli *unitigs*) avendo come principali difficoltà la selezione di un nodo iniziale valido e la costruzione di un percorso partendo da esso collegando nodi adiacenti fino alla massima espansione.

Selezione nodo di partenza

Si sfruttano le caratteristiche topologiche dell'insieme note a priori, il fatto che la distribuzione dei conteggi sia alquanto asimmetrica e non uniforme comporta, cercando la facilità di compressione, a preferire i nodi con i conteggi medi più elevati poiché nodi a conteggio più elevato sono più difficili da comprimere quindi di conseguenza vengono inseriti all'inizio poiché è probabile che siano vicini ad altri nodi con conteggio simile, aiutando la compressione.

Costruzione di percorsi

Si procede a costruire il percorso di copertura totale del grafo, in particolare *USTAR* per migliorare le prestazioni, cerca di evitare in partenza **nodi altamente connessi** (a differenza di *UST*) che seppur possano portare un vantaggio nelle fasi iniziali della compressione risultano più utili nelle fasi finali per terminare l'interconnessione dei nodi isolati rimanenti, di conseguenza i

[6] si tratta di una sequenza ottenuta collegando fra loro unitigs e k-mer sovrapposti durante l'esplorazione di un grafo

percorsi vengono estesi partendo dai nodi meno connessi, perseguendo l'obiettivo generale di ottenere una *lunghezza cumulativa* inferiore, al fine di migliorare la compressione.

2.3 W-SSHash

Sviluppato da *Giulio Ermanno Pibiri*[16] si tratta di un dizionario esatto, pesato, associativo, veloce e compatto, il quale rappresenta un'elaborazione ad alto livello non agendo direttamente sui dati grezzi. Infatti l'obiettivo è rappresentare un gruppo di *k-mer* con il loro relativo **conteggio** in modo da accedere velocemente ad essi, in particolare il seguente algoritmo deriva direttamente da *SSHash Dictionary*[17].

Vengono quindi introdotte le seguenti notazioni:

- **S**: sequenza di *DNA*.
- **K**: set di n distinte coppie $< g, w(g) >$ dove g è un *k-mer* e $w(g)$ il suo peso[7].
- **Runs**: con la presente nomenclatura ci si riferisce ad una sottosequenza massimale di simboli uguali, in particolare con simboli si intendono i pesi correlati ai *k-mers*, il quale approccio permette di memorizzare in posizioni ravvicinate della memoria *k-mers* con conteggi uguali conteggiando appunto il peso una singola volta.

2.3.1 Metodo di compressione

Obiettivo

La principale sfida da risolvere consiste nel trovare una rappresentazione compressa di K che permetta di verificare velocemente l'appartenenza di un g qualsiasi e nel caso ritornare $w(g)$.

Compressione dei pesi

L'algoritmo di *SSHash Dictionary* implementa una funzione di *hash* che si comporta similmente ad una *MPHF*[8] con l'eccezione che rifiuta nativamente i *k-mers* non appartenenti all'insieme K, si denota con *i* il valore *hash* equivalente ad un *k-mer* g ottenuto come $i = h(g)$ al quale *hash* può essere legato il peso, l'idea alla base consiste nell'elaborare l'input K utilizzando la proprietà del *spectrum-preserving string set* ossia: una collezione di stringhe dove un *k-mer* appare solo una volta, la quale collezione è possibile generarla utilizzando algoritmi specifici per minimizzare il totale di simboli in S come *UST* 1.2.4 oppure *USTAR* 2.2.

Importante notare che la funzione h preserva l'ordine relativo dei *k-mers* in ingresso, il che aiuta la compressione poiché *k-mers* consecutivi hanno normalmente lo stesso peso; si procede successivamente a codificare la sequenza ottenuta con *RLW*[9].

[7] Con peso di *k-mer* si intende la sua molteplicità all'interno della sequenza

[8] Minimal Perfect Hash Function nel dettaglio 2.4.1

[9] Tecnica di compressione general porpouse Lossless derivata da RLE il quale sostituisce valori consecutivi con una coppia valore-ripetizioni

Riduzione del numero di Runs

Al fine di comprimere ulteriormente i pesi di K mantenendo valido il contenuto, è possibile agire sull'ordine e sull'orientamento dell'insieme S, di fatto riordinando i *k-mers*, la quale procedura non ha influenza sulle proprietà della funzione h anzi ne agevola eventualmente le prestazioni, è quindi possibile permutare l'ordine di S per ridurre i **Runs** all'interno di W ossia le sottosequenze di peso uguale.

La permutazione viene elencata come $[x, y, z, ...]$ dove l'indice all'interno dell'elenco indica l'indice della stringa da traslare mentre il parametro indica l'indice della nuova posizione, nel caso fosse presente un segno $-$ indica l'inverso della stringa come si nota in fig 2.3, questa tecnica permette di avvicinare stringhe simili per facilitare la compressione.

2.3.2 Grafo

Al fine di trovare la permutazione migliore da applicare all'insieme S è possibile trasformare la scelta in un problema di copertura totale di un grafo pesato agli estremi, ossia creare un percorso il quale attraversa ogni nodo una sola volta, il grafo in questione ha le seguenti caratteristiche:

- È presente un nodo u per ogni sequenza di S ed u possiede un *front* e *back* denominati con il primo ed ultimo peso della sequenza, un nodo è identificabile univocamente tramite una 4-tupla formata da *(id,front,back,sign)* dove id è l'identificatore correlato ad una sequenza $s \in S$ e $sign$ indica se la sequenza è invertita.

- È presente un arco attraverso ogni due nodi u e v che hanno un estremo con lo stesso peso, di conseguenza si ha che $u_1.back = u_2.front$.

- Un percorso orientato di lunghezza l in G, con l che corrisponde al numero di nodi attraversati, può essere indicato in maniera univoca come una sequenza di *id* dei corrispettivi nodi.

Si tratta di un problema di difficoltà **NP-hard**[15] il quale tuttavia, utilizzando le caratteristiche dell'insieme è possibile risolvere in tempo lineare.

Permutazione ottima

Si introduce quindi la metodologia utilizzata per computare il percorso minimo che ricopra tutti i nodi (al quale per conseguenza corrisponde il numero minimo di *runs*), vengono quindi definite le seguenti proprietà:

- Per ogni nodo vale che $u.front < u.back$, fatta eccezione se l'orientamento è stato modificato, allora in quel caso $u.front > u.back$.

- L'insieme I_w di nodi dove w compare come estremo è chiamato **incidenza di** w mentre **frequenza di** w è definito come il numero di volte in cui w compare nei nodi di I_w indicata come $n(I_w)$.
- Un **cammino** $u_1 - > \cdots - > u_l$ può essere sostituito dal singolo nodo (u_1.front, u_l.back), la semplificazione viene eseguita solamente a livello logico al fine di non perdere le informazioni associate.
- Sia $E_{x,y}$ il sottoinsieme di nodi uguali (x, y), si definisce $d = |E|$, se d pari i nodi possono essere orientati per formare un cammino massimo di estremi (x, x) o (y, y), se d dispari allora il cammino ha estremi (x, y).
- Un estremo con frequenza dispari è un nodo w tale che $|I_w|$ è dispari.
- Se $|I_w|$ è dispari allora w appare come estremo di qualche cammino in C.
- $|W_{odd}|$ è pari.
- Se $|W_{even}| = 0$ allora $|C| = \frac{|W_{odd}|}{2}$ e quindi l'algoritmo *greedy-cover* è ottimale.
- Sia c una componente connessa di G, al termine dell'algoritmo *merge-even* c è semplificata in un grafo in cui o c'è solo un nodo oppure $I_w = 0$.
- **Teorema** sia C_{even} l'insieme delle componenti connesse di G in cui i nodi hanno solo estremi di frequenza pari, allora $|C^*| = |C_{even}| + \frac{|W_{odd}|}{2}$ e l'algoritmo *min-cover* è ottimale.

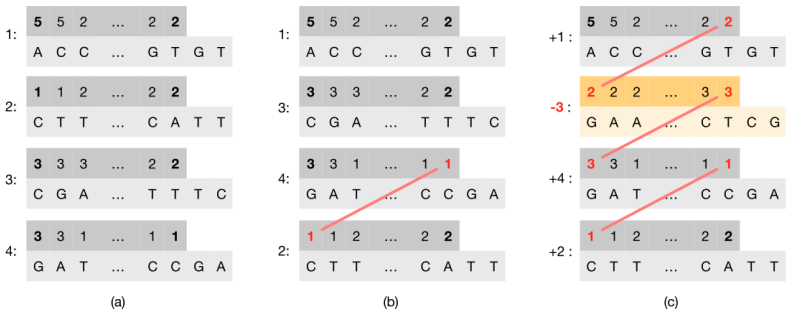


Figura 2.3: Esempio di applicazione di una permutazione [1,3,4,2] dalla *fig.a* alla *fig.b* e della permutazione [1, −3,4,2] in *fig.c*, notando con particolare attenzione come i pesi al termine della sequenza vanno a combinarsi

Utilizzando le seguenti proprietà è possibile semplificare il grafo ottenendo che:

- Tutti i nodi di $E_{x,x}$ sono orientati per formare un unico cammino con estremi (x, x). Tutti questi nodi vengono rimossi da G e sostituiti con un nuovo nodo (x, x) che viene successivamente unito ad un nodo (x, y).

- Tutti i nodi degli insiemi $E_{x,y}$ sono orientati per formare un unico cammino con estremi x, y se $|E_{x,y}|$ dispari oppure due cammini entrambi con estremi (x, y), se $E_{x,y}$ è pari in questo caso due nodi uguali x, y vengono aggiunti a G. Dopo questa prima fase di filtraggio si ottiene G^* nel quale non ci sono nodi della forma (x, x) ed i nodi della forma (x, y) compaiono al massimo 2 volte.

Successivamente vengono costruiti due insiemi I ed U nel seguito manipolati tramite *merge-even* e *greedy-cover* dove I è un insieme di raccolta di tutti gli insiemi di incidenza I_w mentre U rappresenta l'insieme di tutti i nodi che devono ancora essere raggiunti da un cammino di C.

Min-Cover

L'algoritmo **min-cover** ha come scopo l'ottenere una copertura ottimale dei nodi minimizzando come da nome il numero di percorsi, partendo da un grafo con nodi pesati si procede applicando prima l'algoritmo **merge-even** che si occupa di unire i nodi con peso di frequenza pari al fine di ridurre le dimensioni del grafo, successivamente si procede con l'algoritmo **greedy-cover** il quale costruisce una copertura del grafo minimizzando i percorsi utilizzando, come il nome suggerisce, un tecnica greedy[10].

2.3.3 Prestazioni

Utilizzando delle tabelle *hash*[11] per implementare gli insiemi I ed U le operazioni di inserimento, cancellazione e lettura sono $O(1)$ in media, il costo di inserimento è sempre costante utilizzando una coda doppia[12] per il percorso P.

Quindi l'algoritmo *min-cover* viene eseguito in $\theta(m)$ con $m = |G|$ ossia il numero di nodi del grafo di input G. Mentre *greedy-cover* richiede $O(m)$, *merge-even* può essere ottimizzato ad $O(m)$, inoltre l'algoritmo presenta un'occupazione spaziale di $\theta(m)$ al massimo.

[10] Tenica che prevede di effettuare solo scelte locali in base alla conoscenza attuale dello stato del problema
[11] Struttura dati particolare
[12] Struttura dati a nodi doppiamente puntati che permettere un rapido accesso dei valori all'inizio ed alla fine

2.4 Space-efficient representation of genomic K-mer count tables

Come già presentato l'utilizzo di un dizionario al fine di memorizzare *k-mers* ed il relativo conteggio hanno come principale svantaggio lo spazio occupato e la velocità di accesso ai dati.

Sviluppato da *Yoshihiro Shibuya, Djamal Belazzougui e Gregory Kucherov*[18] il seguente algoritmo con relative varianti propone un'efficiente soluzione ai problemi sopra citati.

L'idea alla base consiste nel rappresentare il dizionario senza memorizzare direttamente i *k-mers* che lo compongono, tramite utilizzo di costrutti quali *CSF* e *Bloom Filter* per poter disporre la creazione di due strutture dati indipendenti al fine di applicare ottimizzazioni più massive.

2.4.1 Introduzione

Vengono di seguito introdotti alcuni metodi e tecniche utilizzate all'interno delle strutture presentate.

Funzioni Hash Perfette Minime

Costrutto alla base dell'algoritmo, si tratta di di una *funzione biettiva* che mappa ogni elemento di un insieme S in un intervallo $[0, |S| - 1]$, mentre eventuali informazioni aggiuntive sono salvate all'interno di un *array*[13] con indicizzazione correlata, tuttavia non risulta ottimale quando i conteggi non sono una distribuzione abbastanza uniforme, ossia con un *entropia empirica bassa*; situazione ancora peggiore con k elevati nella quale si ha un conteggio del singolo *k-mer* tendente ad uno.

La natura della funzione fornisce spazio per ulteriori miglioramenti i quali però possono peggiorare la velocità di accesso casuale ai dati, inoltre lo spazio occupato diventa non trascurabile con dimensioni delle chiavi minime ben al di sopra del limite minimo teorico.

Funzioni Statiche Costanti CSF

Una **funzione statica** ha come compito, definito un sottoinsieme S di ritornare il valore associato ad un elemento s se $s \in S$ altrimenti produce un risultato arbitrario (tendenzialmente 0) se l'ingresso non appartiene all'insieme.

È stato dimostrato che è possibile restituire il valore correlato senza memorizzare direttamente l'insieme S, rappresentando ottime soluzioni per sostituire costrutti *MPHF*, infatti le funzioni *CSF* sono in grado di fornire un ulteriore livello di compressione, tuttavia per natura non performano al meglio con distribuzioni che hanno bassa *entropia empirica*.

[13] Struttura dati sequenziale dove gli elementi sono salvati ed individuabili tramite un indice

Minimizers

Dato un *k-mer* il suo minimizer associato è la più piccola sottostringa di lunghezza m i cui parametri sono definiti tramite una **funzione di hash non crittografica**, chiaramente si ottiene $m < k$, motivo per il quale sono anche definiti *m-mers*, l'idea di base consiste nell'utilizzare il *minimizer* come impronta o *hash* di un *k-mer* correlato, in modo che *k-mers* simili abbiano lo stesso minimizer, tecnica utilizzata per esempio da *Kraken*[19].

Bloom Filter

Un **Bloom Filter** è una struttura dati probabilistica che permette di individuare l'appartenenza di un set S in un universo U secondo determinati parametri che è possibile selezionare, in questo caso particolare si conta l'abbondanza di un *k-mer* al fine di scartare *k-mers* con molteplicità ridotta che potrebbero essere generati da errore di lettura.

Di fatto è possibile definire il *Bloom Filter* come un filtro basato sul peso dei *k-mers* permettendo l'accesso solo a *k-mers* col peso minimo richiesto.

2.4.2 BCSF

Si tratta di un'ottimizzazione della funzione *CSF* a cui viene applicato un **filtro Bloom** al fine di gestire *k-mer sets* con bassa entropia minimizzando lo spazio occupato, l'applicazione avviene tramite una prima richiesta alla struttura del *Bloom Filter* e se il dato ritorna con esito positivo si procede al recupero dalla funzione *CSF*.

Procedimento

Si indica con l'insieme K_0 i *k-mers* più frequenti mentre con K tutti i *k-mers*, i *k-mers* meno numerosi (quindi $K - K_0$) vengono memorizzati tramite un *Bloom Filter*, importante considerare i falsi positivi che possono essere generati dal *Bloom Filter* che vengono indicati con $FPB(K_0)$ e con ϵ il tasso di falsi positivi, per valutare l'utilità del *Bloom Filter* si utilizza la formula $CBF \cdot \frac{1-a}{a} \cdot log(\frac{1}{\epsilon}) < 1$, dove *CBF* indica i bit per chiave chiesti dal *Bloom Filter* ed a rappresenta $|K_0| = a|K|$.

Importante quindi valutare il parametro ϵ arrivando alla formula finale: $a > \frac{CBF \cdot log(e)}{CCSF + CBF \cdot log(e)}$ dove *CCSF* indica il numero di bit per chiave richiesti dalla funzione *CSF*, con calcoli sperimentali si ottiene che il limite teorico del *Bloom filter* è $CBF = 1.44$ quindi il **BCSF** è utile per valori di a sufficientemente elevati.

2.4.3 Minimizer Bucket

L'idea come si intuisce dal nome consiste nell'utilizzare i *minimizer* per migliorare le performace, in particolare vengono utilizzati i *minimizer* come chiave *hash* di un singolo *bucket*[14] che contiene tutti i *k-mers* che condividono lo stesso *minimizer*, il *bucket* appunto come in una classica *hashmap* viene utilizzato come *overflow*[15], che se presente fa definire il *bucket* come **ambiguo**, si nota inoltre che lo stesso *minimizer* tende ad avere lo stesso conteggio dei *k-mers* associati, di fatto i bucket sostituiscono i *k-mers* anche nel conteggio, i quali possono essere recuperati tramite una *query* che li ritorna esatti se presenti.

AMB

Primo tipo di implementazione dell'algoritmo, si contrassegnano i *minimizer* non ambigui con u quindi $g(u)$ è l'unico valore del bucket mentre per *minimizer* ambigui v si imposta $g(v) = 0$ dove 0 è un valore sentinella.

Il fatto di identificare *bucket* ambigui e non, consiste nell'evitare di far procedere al prossimo livello dell'algoritmo *bucket* che sono già non ambigui e di conseguenza non necessitano di ulteriori operazioni, mentre per i *bucket* ambigui si necessita di una seconda struttura simile a g chiamata f la quale funziona all'esatto modo di g ma l'insieme di partenza è il risultato di g stessa, il numero di strutture e quindi livelli può essere scelto dall'utente. Il vantaggio è che bisogna solamente memorizzare la mappatura f dei *bucket* ambigui, tutte le mappature di *AMB* sono salvate usando *BCSF*.

FIL

La seconda implementazione è un estensione di *AMB*, $g(s)$ viene definito come il valore più grande tra tutti i valori all'interno del bucket ossia $g(s)$ è il peso associato più frequente tra *k-mers* con lo stesso *minimizer*, in particolare se s è un *minimizer* non ambiguo allora $g(s)$ viene impostato come unico valore del bucket, altrimenti la scelta è arbitraria.

Viene quindi creata una mappa di correzione h dove $h(q)$ rappresenta la differenza tra il conteggio iniziale del *k-mer* ed il valore rappresentato dal suo *minimizer*, il tutto viene memorizzato tramite *BCSF*.

Cascading Un ulteriore strato tra *minimizer* e *k-mers* può essere definito come un filtro per alcuni *k-mers* mentre ne propaga altri allo strato successivo, funzionando similmente a vari *Bloom Filters* in sequenza, in questo caso quindi dall'ingresso al risultato finale ogni strato

[14] Struttura che può contenere vari elementi, l'ordine non è importante
[15] Eccedenza

riduce l'insieme di competenza partendo dai *k-mers* per arrivare ai *minimizers*. Inoltre è possibile implementare se le specifiche lo consentono, un algoritmo di approssimazione, il quale può ulteriormente ridurre gli elementi nella fase di collisione di un *bucket* tramite con costrutto probabilistico. Per esempio se due valori hanno una differenza al di sotto di una determinata soglia minima uno di essi viene scartato, generando di conseguenza un errore che potrebbe essere accettabile.

Figura 2.4: Confronto tra le strutture utilizzate a supporto dei bucket per *AMB* e *FIL*

Conclusioni

BCSF rappresenta la migliore compressione per distribuzioni sbilanciate oltre ad avere ottime performance in accesso casuale.

AMF e *FIL* invece permettono di rompere i limiti entropici di compressione a patto che la distribuzione dei pesi sia sufficientemente adatta, inoltre *AMB* risulta sempre più veloce di *FIL* quando i *minimizer* rappresentano effettivamente un vantaggio nella compressione.

Capitolo 3

Analisi e Comparazione tecnica

Nel seguente capitolo si procede all'analisi dei vari metodi descritti nel capitolo precedente, da sottolineare come, data la differente tipologia non sarà un confronto fra di essi, bensì verrà valutato il miglioramento rispetto alle attuali tecnologie ed il loro ambito di applicazione.

3.1 Costruzione Grafo di De Bruijn

3.1.1 GGCAT

Per il presente algoritmo, che si ricorda genera sia grafi di *De Bruijn Compatti Colorati* che non, sono stati selezionati per il confronto Cuttlefish2 1.2.4 il quale si è dimostrato avere prestazioni generali migliori rispetto a BCALM2 1.2.4 per la computazione di grafi non colorati.

Mentre la scelta per l'algoritmo di confronto per la computazione di grafi colorati è ricaduta sul noto BiFrost 1.2.4.

Dataset Per l'analisi del *GDBC* non colorato sono stati selezionati i seguenti datasets:

- Una lettura del genoma umano tramite tecnologia Illumina.
- Una lettura del microbioma dello stomaco umano.
- Un genoma di salmonella.
- Un genoma batterico.

Mentre per il confronto del *GDBC* colorato sono stati utilizzati:

- Il genoma umano.
- Un genoma di salmonella.

- Un genoma batterico ad ampio spettro.

La scelta è ricaduta sui presenti *datasets* al fine di avere un ampio insieme di valutazione con parametri dei singoli molto diversi, inoltre ogni *datasets* viene testato con diversi valori di lunghezza dei *k-mers* come illustrato in Fig 3.1.

Nome	k	Numero di k-mers in milioni	Numero di k-mers unici in milioni	Numero di Unitigs massimali in milioni	Lunghezza media	Lunghezza massima
Sequenziamento Umano	27	193386	3143	271.71	37.57	4257
	63	162119	3442	85.85	102.09	6428
Microbioma dello stomaco	27	7199	2583	210.92	38.24	5633
	63	51096	3100	154.78	82.03	3730
	119	18580	1692	60.74	145.86	3196
Genoma Umano	27	288091	2531	76.52	59.07	7397
	63	288091	3251	41.09	141.13	12638
Genoma Salmonella	27	480109	372	25.82	40.41	98331
	63	479815	689	24.02	90.67	329835
Genoma batterico	27	2527666	36035	1356.47	52.57	68201
	63	2523967	46791	632.47	135.98	798154
	119	2518215	56390	448.62	243.70	2185866
	255	2504523	71553	318.53	478.64	3427805

Figura 3.1: Nella presente tabella sono illustrate le specifiche delle varie sequenze utilizzate per l'analisi.

Analisi grafi non colorati Come si nota dai grafici 3.2, salvo ridotte eccezioni l'algoritmo *GGCAT* ha prestazioni nettamente superiori a *Cuttlefish2* in particolar modo sul tempo di esecuzione, mostrando ottimi miglioranti anche nella dimensione finale su disco del *GDBC* in particolare con sequenze molto lunghe, pagando solamente in alcuni casi un consumo maggiore di memoria RAM[1] con miglioramenti fino al $4{,}3$x per $k < 64$ e fino al $20{,}8$x per k superiori.

		Cuttlefish 2			GGCAT		
Data Set	k	Tempo	Ram GB	Dimensione GB	Tempo	Ram GB	Dimensione GB
Sequenziamento Umano	27	1h:15min	3,95	209	1h:16min	4,54	220
	63	2h:07min	4,23	140	1h:03min	7,11	156
Microbioma dello stomaco	27	0h:30min	3,35	78	0h:22min	6,09	78
	63	1h:08min	3,86	107	0h:19min	5,42	51
	119	1h:04min	3,13	97	0h:12min	5,33	32
Genoma Salmonella	27	6h:59min	4,38	1515	3h:38min	3,46	378
	63	12h:02min	3,88	1145	3h:31min	3,96	274
	119	17h:07min	3,95	1088	3h:39min	4,12	279
	255	77h:58min	4,82	1056	3h:44min	4,33	325
Genoma batterico	27	13h:24min	42,6	2604	7h:7min	8,49	1186
	63	25h:29min	53,84	1962	5h:56min	10,06	810
	119	Crash	Crash	Crash	6h:14min	9,46	793
	255	Crash	Crash	Crash	6h:30min	9,42	849

Figura 3.2: Vengono elencate le performance di Cuttlefish 2 e GGCAT nel caso di grafo non colorato

Analisi grafi colorati Dalla figura 3.3 si nota un estremo miglioramento sui tempi di esecuzione persino in situazioni nel quale *BiFrost* non è in grado di portare a termine l'esecuzione.

[1] Memoria temporanea volatile diversa dalla memoria dove vengono immagazzinati i dati

Si misurano miglioramenti con incrementi medi del $4{,}8$x per sequenze più corte, in linea con la variante non colorata, ed incrementi medi del 35x per genomi di dimensioni più elevate.

A discapito delle grandi prestazioni, il principale difetto risiede nello spazio di archiviazione occupato, il quale risulta essere di un ordine di grandezza superiore rispetto a *BiFrost*, tuttavia l'indice dei colori prodotto da *GGCAT* risulta essere tra il 2x ed il 20x più compatto, inoltre l'utilizzo di memoria Ram è nettamente minore rispetto al caso precedente di grafo non colorato, la quale, dati i valori in gioco, può rappresentare un problema di portata maggiore in gran parte dei sistemi computazionali attuali rispetto all'occupazione di memoria di massa.

		BiFrost Colorato			GGCAT Colorato		
Data Set	k	Tempo	Ram GB	Dimensione GB	Tempo	Ram GB	Dimensione GB
Genoma Umano	27	6h:39min	30,68	13	1h:17min	8,29	290
	63	5h:16min	38,65	9	1h:08min	8,8	224
Genoma Salmonella	27	50h:29min	79,32	54	1h:31min	6,54	201
	63	46h:34min	85,21	61	1h:11min	6,18	128
Genoma batterico	27	Crash	Crash		13h:28min	21,64	1296
	63	>10d			11h:33min	21,48	838

Figura 3.3: Vengono presentate le prestazioni di GGCAT comparato a BiFrost nella specifica di grafo colorato

Query I risultati di interrogazioni specifiche sul $GDBC$ colorato riportano risultati eccezionali con misurazioni sperimentali che portano un miglioramento del $83{,}6$x per $k = 63$ e del 480x rispetto a *BiFrost* per $k = 27$, con un confronto assolutamente impari di *33h:14min* per *BiFrost* contro soli *4min* di tempo di esecuzione per *GGCAT*.

Risultati L'algoritmo *GGCAT* ha riportato risultati estremamente positivi con prestazioni di ordini di grandezza superiori in determinati massimi, tuttavia si dimostra da altri esperimenti non essere particolarmente ottimizzato per valori di $k < 15$ mantenendo comunque risultati in linea con altri algoritmi, inoltre è importante ricordare che per $k > 64$ possono sussistere leggere imprecisioni dovute alle collisioni della funzione di *hashing* interna utilizzata.

GGCAT quindi in situazioni generali risulta estremamente più veloce di **CuttleFish2** e **BiFrost**, risultando ottimale nella maggior parte dei casi, fatta eccezione se l'obiettivo è ottenere un *GDBC colorato* di dimensioni il più compatte possibili senza considerare il tempo richiesto.

3.2 Compressione Grafo di De Bruijn

3.2.1 USTAR

Come già descritto *USTAR* 2.2 si occupa di comprimere ulteriormente un *GDBC*, al fine di preparare il grafo per l'elaborazione con successivi algoritmi.

Per definizione il diretto confronto va effettuato con *UST* 1.2.4 dal quale eredita l'idea progettuale dell'algoritmo, nella comparativa si è scelto di includere anche *BCalm* al fine di valutare il miglioramento in termini di spazio rispetto alla mancata applicazione dell'algoritmo *USTAR*. Infatti è importante ricordare che *USTAR* opera su *GDBC*, il quale viene generato da algoritmi come appunto *BCalm*, inoltre il principale obiettivo di *USTAR* è la riduzione delle dimensioni finali del grafo, motivo per il quale non vengono valutate le performance in termini di tempo di esecuzione ma solamente in termini di occupazione spaziale.

3.2.2 Dataset e confronto

Dataset	Lunghezza Read	Numero di Reads	Dimensione GB
Escherichia coli	36	20.816.448	9,304
Microbioma Umano	101	53.588.068	3,007
RNA di Soia	125	83.594.116	3,565
DNA di pollo	92	14.763.228	0,230
Drosophila ananassae	75	18.365.926	0,683

Figura 3.4: Dataset di rifermento per l'analisi di USTAR

Dataset Il *dataset* utilizzato è il seguente in tabella 3.4 composto da varie sequenze con letture di differente lunghezza e dimensione in *GB* molto varia, al fine di testare al meglio la controparte di compressione.

Dataset	BCalm	UST	USTAR
Escherichia coli	43.100.358	12.641.658	12.332.551
Microbioma Umano	792.616.145	194.173.905	185.905.825
RNA di Soia	278.247.157	64.694.925	61.236.404
DNA di pollo	110.324.321	25.833.347	24.546.244
Drosophila ananassae	21.192.576	5.737.778	5.599.034

Figura 3.5: Confronto tra BCalm, UST e USTAR in termini di spazio occupato per la compressione, il valore più basso è il migliore

Analisi Prestazione Come si evidenzia in tabella 3.5 dalle comparazioni eseguite con $k = 21$ si nota come *BCALM* abbia chiaramente il peggiore rapporto di compressione mentre *USTAR* ha prestazioni generali paragonabili a *UST* tuttavia risultando sempre in miglioramento con una media globale del 4.2%, si procede quindi con un confronto in profondità fra *USTAR* ed *UST* definendo i seguenti parametri al fine di valutarlo al meglio.

- **CL**: Lunghezza cumulativa.
- **Counts**: Dimensioni del file contenenti i pesi.
- **Overall**: Somma dei file compressi di *k-mers* e conteggi relativi.

k	ΔCL	ΔCounts	ΔOverall
15	33,64	12,07	26,4
17	13,61	15,85	13,92
21	2,1	14,17	4,2
31	0,97	12,7	2,3

Figura 3.6: Vengono riportati in percentuale i miglioramenti di *USTAR* rispetto ad *UST*

Come si può notare dalla tabella 3.6 si ottiene un netto miglioramento di performance con k compresi tra 15 e 21 mentre le prestazioni tendono a ridursi fino a raggiungere un livello molto simile per $k > 31$, potendo quindi stimare un miglioramento medio rispetto a *UST* del $13{,}50\%$ nel caso di valore k favorevole.

3.2.3 Risultati

I risultati elencati sono un miglioramento massivo del 76% di media su *BCALM* come ci si aspettava, ed una media del 4.2% su *UST* con una tendenza di maggiori prestazioni per valori di k contenuti nell'intorno $[15 - 21]$ fino ad un incremento massimo di 26.4% rispetto *UST*.

Invece per k maggiori di 31 le prestazioni tendono a ridursi mantenendo comunque miglioramenti di vari punti percentuali.

USTAR si dimostra quindi come un ottimo aggiornamento di *UST* senza individuabili controindicazioni.

3.3 K-mers Dictionary

Di seguito vengono confrontati svariati algoritmi che operano sulla costruzione di un dizionario con chiave i *k-mers* e valore il loro *peso*, con differenti finalità in termini di spazio occupato e prestazioni di lettura.

3.3.1 w-SSHash

Come descritto nel capitolo 2.3 si tratta di un dizionario avanzato derivato dal noto *SSHash[17]* al quale sono stati implementati svariati miglioramenti.

Dataset Utilizzati

Viene utilizzata la collezione genomica formata da *[E-Coli, C-Elegans, S-Enterica, Human-Chr-13]* con pesi collezionati tramite algoritmo *BCALM2* senza operazioni di filtro, di conseguenza si ottiene in media un *entropia empirica* relativamente bassa, situazione data dal fatto che la maggior parte dei *k-mers* appare una sola volta, fatta eccezione per *S-Enterica* che possiede un *entropia empirica* molto più elevata avendo vari *k-mers* con conteggio simile come si nota dalla tabella 3.7.

La scelta è stata effettuata al fine di valutare l'efficacia su dati sia a bassa che ad alta *entropia empirica*.

Dataset	K-mers	Pesi Distinti	Peso Massimo	Entropia Empirica
E-Coli	5.235.781	22	27	0,206
S-Enterica	12.408.741	630	7.956	4,155
Human-Chr	90.911.778	806	6.354	0,160
C-elegans	94.006.897	398	3.478	0,223

Figura 3.7: Nel dettaglio i *dataset* utilizzati per l'analisi di **w-SSHash**

Prestazioni

In prima analisi viene valutata l'efficienza dell'algoritmo per la riduzione del numero di *runs* infatti come si nota dalla tabella 3.8 l'entropia empirica in *bits/k-mer* migliora ulteriormente in tutti i casi rispetto alla sola compressione con *SSHash* ottenendo valori ben al di sotto del limite teorico indicato nella seconda colonna.

Dataset	Entropia empirica	SSHash	SSHash+Algoritmo
E-Coli	0,206	0,017	0,014
S-Enterica	4,155	0,464	0,328
Human-Chr	0,160	0,135	0,108
C-elegans	0,223	0,069	0,055

Figura 3.8: Valori di entropia empirica descritta in *bits/k-mer* ottenuta dalla formula teorica e dai presenti algoritmi

Si procede quindi al confronto complessivo mostrato in tabella 3.9, ai fini della comparazioni sono stati selezionati i seguenti algoritmi:

- **dBG-FM**: Basato su *FM-index*[20] è un dizionario di *k-mers* ponderati che restituisce il conteggio tramite una *query*, la presente implementazione ha un parametro selezionabile di *trade-off*[2] per calibrare le prestazioni dell'accesso casuale, al fine del test è stato scelto un valore intermedio di 64.
- **cw-dBG**[21] Dizionario basato sull'architettura *BOSS*[22], similmente a *dBG-FM* è stato testato con parametro di *trade-off* pari a 64.
- **SSHash+BCSF/AMB**: Dizionario *SSHash* non ponderato unito a varie funzioni di ottimizzazione, le quali sono state precedentemente descritte rispettivamente nel capitolo 2.4.2 e 2.4.3.
- **SSHash**: Si è scelto di comparare anche una normale interazione di *SSHash* senza strutture a supporto, importante ricordare che il presente costrutto non possiede informazioni sui pesi dei *k-mers* ma viene incluso solo per valutare quanto gli algoritmi che si basano su esso peggiorino le prestazioni.

Risultati Si ottiene dalle analisi sperimentali di *w-SSHash* un rapporto di compressione ottimale, con solamente *cw-dBG* in grado di migliorarlo saltuariamente ma pagando tuttavia un elevata complessità in termini di struttura, comportando un tempo di risposta alle interrogazioni molto più elevato rispetto ad altri algoritmi, mentre *w-SSHash* oltre alle dimensioni spaziali mantiene anche un'ottima velocità di interrogazione della struttura dati.

In generale quindi *w-SSHash* ottiene prestazioni medie migliori rispetto ad algoritmi quali *dBG-FM* e *cw-dBG*, mentre ha prestazioni in linea ma sempre leggermente migliori per altri algoritmi basati su *SSHash* utilizzando come insieme di confronto distribuzioni a bassa entropia.

Mentre presenta un ottimo miglioramento utilizzando distribuzioni ad alta entropia come si nota in fig 3.9. Si nota inoltre che lo spazio richiesto per la memorizzazione non aumenta in

[2]parametro di ottimizzazione

maniera considerevole rispetto a *SSHash*, il quale non memorizza i pesi. Inoltre dai presenti test si nota come abbia prestazioni migliori rispetto agli algoritmi che verranno comparati di seguito.

Dataset	E-Coli			S-Enterica			Human-Chr			C-Elegans		
Algoritmo	bits/k-mer	MB	qtm	bits/k-mer	MB	qtm	bits/k-mer	MB	qtm	bits/k-mer	MB	qtm
dBG-FM	4,02	2,51	6,57	147,84	218,70	9,43	4,07	44,07	9,98	4,01	44,89	9,60
cw-dBG	2,86	1,87	62,97	5,58	8,66	76,74	2,86	32,55	67,63	2,84	33,34	77,72
SSHash+BCSF	5,02	3,29	0,48	11,43	17,73	0,52	6,12	69,55	0,88	5,94	69,80	0,90
SSHash+AMB	4,85	3,17	0,57	9,15	14,19	0,68	6,05	68,75	1,06	5,82	68,39	1,07
w-SSHash	4,80	3,14	0,35	5,97	9,26	0,46	6,04	68,66	0,82	5,75	67,52	0,85
SSHash	4,79	3,14	0,32	5,63	8,73	0,39	5,93	67,39	0,73	5,69	66,86	0,77

Figura 3.9: Confronto fra dizionari di *k-mers*, con **MB** corrispondente a peso in Megabyte e **qtm** corrispondente a tempo medio di risposta di una query espresso in *us/k-mer*

3.3.2 Rappresentazioni efficienti di k-mers in dizionari

Dataset e Confronto

Al fine dell'analisi è stata utilizzata la seguente collezione di genomi: *[Escherichia coli, Escherichia Coli Sakai, Caenorhabditis ed una lettura tramite tecnologia Illumina codice:SRR10211353]* analizzando le performance con svariati valori di k. Gli algoritmi confrontati sono i seguenti:

- **CSF**: Implementazione standard basata su *Sux4J*[23]
- **BSCF**: vedi sezione 2.4.2
- **AMB**: vedi sezione 2.4.3
- **FIL**: vedi sezione 2.4.3
- **FIL 3 layer**: Implementazione di *FIL* ma con un livello aggiunto alla struttura.

Algoritmo	k=13	k=15	k=18	k=21
Entropia empirica	0,61	0,24	0,19	0,18
CSF	1,34	1,23	1,22	1,22
BCSF	0,87	0,34	0,26	0,25
AMB 2 layer	0,87	0,26	0,11	0,08
AMB 3 layer	0,87	0,26	0,10	0,07
FIL 2 layer	0,90	0,29	0,13	0,10
FIL 3 layer	0,91	0,29	0,13	0,10

Figura 3.10: Comparazione su genoma a bassa entropia, le prestazioni sono indicate in *bits/k-mer*

Prestazioni

Dati a bassa entropia Come si nota dalla tabella 3.10 nel caso di dati sbilanciati a bassa *entropia empirica*, fatta eccezione per *CSF* che paga oltre un bit in più per singolo *k-mers* rispetto agli altri algoritmi, con valori di k ridotti le prestazioni spaziali di *AMB* e *FIL* sono molto simili a *BCSF*, mentre per k superiori a 18 l'incremento è evidente, dimostrando il miglioramento di performance derivate dall'utilizzo dei *minimizer*.

Infatti per k elevati la maggior parte dei *bucket* non è ambigua quindi l'algoritmo ne ha poca necessità, riducendo di conseguenza la propagazione ai livelli inferiori nel caso di *AMB*.

Mentre per *FIL* il miglioramento è simile ma occupando leggermente più spazio, dimostrando che per k piccoli risulta difficoltoso migliorare l'entropia empirica poiché i *minimizer* per definizione non sono efficienti nel contesto di *k-mers* ridotti, nella particolare situazione invece si nota un miglioramento di prestazioni con *BSCF* dovuto appunto all'utilizzo del *Bloom Filter*.

Algoritmo	k=10	k=11	k=12	k=13	k=15	k=18	k=21	k=24
Entropia empirica	2,68	4,46	4,40	3,95	3,58	3,48	3,41	3,36
CSF	3,07	5,01	4,94	4,43	4,07	3,97	3,89	3,82
BCSF	3,37	5,01	4,94	4,43	4,13	4,10	4,06	4,00
AMB 2 layer	3,28	5,01	4,94	4,43	3,23	2,25	1,94	1,84
AMB 3 layer	3,28	5,01	4,94	4,43	3,15	2,11	1,80	1,71
FIL 2 layer	3,46	5,01	4,94	4,43	3,63	2,59	2,29	2,18
FIL 3 layer	3,41	5,16	4,94	4,43	4,13	2,61	2,29	2,20

Figura 3.11: Comparazione su genoma ad alta entropia con prestazioni elencate in *bits/k-mer*

Dati ad alta entropia Con dati ad elevata *entropia empirica* corrispondono frequenti collisioni all'interno dei *bucket* utilizzati per il conteggio dei *k-mers* poiché potrebbe esserci un valore

preponderante fra di essi, tuttavia sia *AMB* che *FIL* come visualizzato in tabella 3.11, compensano ugualmente bene con miglioramenti evidenti rispetto a *CSF* per $k > 13$, mentre per k di valore inferiore *BCSF* rimane la soluzione migliore anche se in questa particolare situazione non riesce a migliorare le performance rispetto alla singola funzione *CSF*.

Algoritmo	k=10	k=11	k=12	k=13
Entropia empirica	3,88	2,27	1,20	0,61
CSF	4,38	2,61	1,62	1,34
BSCF	4,38	2,68	1,61	0,87
AMB 2 layer	4,38	2,67	1,61	0,87
AMB 2 layer e=1	4,42	2,35	1,04	0,45
AMB 2 layer e=3	3,83	1,46	0,52	0,31
AMB 2 layer e=5	3,08	0,94	0,40	0,29
AMB 3 layer	4,38	2,67	1,61	0,87
AMB 3 layer e=1	4,45	2,38	1,01	0,37
AMB 3 layer e=3	3,83	1,35	0,32	0,14
AMB 3 layer e=5	3,02	0,74	0,14	0,11

Figura 3.12: Comparazione tra algoritmi esatti (e=0) ed algoritmi approssimati ($e\neq0$) in *bits/k-mer*

Conteggi approssimativi Se è accettabile un errore all'interno dei conteggi allora è possibile ottenere migliori rapporti di compressione come si evince dalla tabella 3.12, infatti per valori k compresi tra 9 e 14 i *minimizer* non consentono di utilizzare al meglio le tecniche sopra elencate, di conseguenza l'applicazione di un algoritmo approssimativo permette di ottiene in maniera quasi scontata un miglioramento ulteriore delle prestazioni all'aumentare della soglia e di errore accettabile.

Interessante notare che per k ridotti ($k < 10$) non sono presenti particolari vantaggi ad utilizzare strutture a 3 livelli con anzi dei lievi peggioramenti dovuti alla complessità aggiuntiva, mentre per $k > 12$ si ottiene un miglioramento al crescere di k sempre più considerevole.

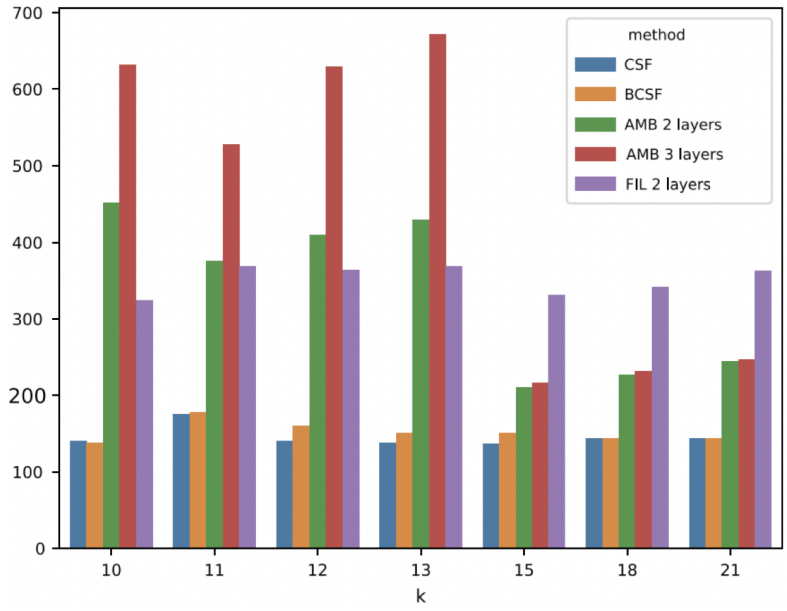


Figura 3.13: Comparazioni tempistiche di esecuzione query in *ns/k-mer*

Velocità query Particolare importanza nella struttura di questi algoritmi sono le prestazioni relative alla velocità di risposta delle interrogazioni, nella tabella 3.13 sono elencate le prestazioni medie in *ns/k-mer* di varie query precedentemente eseguite.

Come facile sospettare una semplice *CSF* ottiene le prestazioni migliori non avendo strutture accessorie ed in maniera molto simile *BCSF* data la semplicità di accesso della struttura del *Bloom Filter*. Mentre la struttura a *bucket* ha chiaramente un impatto sulle performance che peggiorano all'aumentare dei livelli in particolare per k ridotti i tempi di ricerca aumentano considerevolmente.

Contrariamente per k elevati si ottengono prestazioni di *AMB* paragonabili a *CSF* grazie al funzionamento a strati che impedisce la propagazione di valori già ottenuti, mentre interessante notare che *FIL* ha tempo di esecuzione delle *query* quasi costante, essendo che è necessario accedere comunque a tutti gli strati del costrutto a prescindere, di conseguenza si omette *FIL* con *3 layer* il quale sarebbe stato di sicuro più lento della variante a *2 layer*.

Risultati Come possibile notare dalle verifiche sperimentali effettuate, a seconda dell'entropia dell'insieme e del valore di k selezionato la scelta degli algoritmi può variare, in particolare: per sequenze a bassa entropia e con $k < 16$ gli algoritmi si equivalgono, fatta eccezione per *CSF*, quindi la preferenza ricade su *BCSF* data la miglior velocità di esecuzione delle *query*, per valori di $k > 18$ invece *AMB* risulta il migliore pagando tuttavia una velocità di accesso considerevole.

Mentre per sequenze ad alta entropia gli algoritmi si equivalgono fino a $k < 14$ quindi in questo caso si può preferire *BCSF* o *CSF,* invece per k superiore *AMB* risulta il più efficiente

sotto molteplici punti di vista, anche se *BCSF* mantiene una velocità di esecuzioni delle *query* leggermente superiore.

Capitolo 4

Conclusione e Considerazioni Finali

In conclusione nella presente tesi sono stati esposti nuovi algoritmi al fine di migliorare gli attuali standard in termini di compressione di una sequenza genomica ottenuta con *NGS*, in particolare nei metodi presentati ed analizzati si identificano miglioramenti generali nella maggior parte delle situazioni, con alcuni come *USTAR* che rappresentato un ottimo aggiornamento rispetto al suo predecessore ed altri come *GGCAT* che forniscono prestazioni di ordini di grandezza superiore in svariate aree, ampliando la capacità di scelta nel momento di conseguire una finalità specifica, migliorando sia le prestazioni in termini computazionali che soprattutto in termini di spazio occupato.

L'importante finalità di questi sviluppi, oltre alle prestazioni, è di ottenere anche una considerevole diminuzione dei costi e delle specifiche dei sistemi di elaborazione, riducendo i requisiti di memoria e potenza computazionale richiesti, al fine di rendere la tecnologia di sequenziamento più accessibile a diversi enti, offrendo un importante contributo nel continuo sviluppo della *bioinformatica*.

Sviluppi Futuri Sicuramente il tema della compressione dei *k-mers* è ancora ampio e molte nuove soluzioni possono essere esplorate, partendo per esempio dal combinare vari algoritmi qui citati come *GGCAT* che può essere utilizzato come generatore del *GDBC* per *USTAR* il quale può essere applicato come costrutto preliminare all'ingresso di *w-SSHash*, valutandone l'eventuale miglioramento complessivo nei vari ambiti.

Come del resto si stanno facendo strada nuove tecnologie che potrebbero rivoluzionare l'approccio alla compressione. Tra le quali si trovano nuove ottimizzazioni per la gestione di rappresentazioni sparse come *Succint data structure* oppure le potenti *reti neurali*, le quali, fra le varie idee, potrebbero essere utilizzate a partire da una fase preliminare di supporto alla selezione del miglior algoritmo con i relativi parametri per una particolare sequenza, previa un'analisi sommaria e veloce, oppure si potrebbe valutarne l'efficienza nella compensazione dell'errore

di algoritmi qualitativi allenando la rete neurale sul *metagenoma* di interesse. Oppure ancora sperimentare una rappresentazione parziale di un *hash* ed attraverso un'opportuna rete neurale specifica per il contesto, ricostruire i *k-mers* da informazioni incomplete.

In conclusione le possibilità sono molteplici e grazie ai recenti sviluppi tecnologici molte altre stanno diventando facilmente accessibili.

Bibliografia

[1] Wikipedia, *DNA — Wikipedia, L'enciclopedia libera*, 2024. indirizzo: http://it.wikipedia.org/w/index.php?title=DNA&oldid=140430342.

[2] Wikipedia, *Next Generation Sequencing — Wikipedia, L'enciclopedia libera*, [Online; in data 28-agosto-2024], 2024. indirizzo: http://it.wikipedia.org/w/index.php?title=Next_Generation_Sequencing&oldid=140536228.

[3] Wikipedia contributors, *K-mer — Wikipedia, The Free Encyclopedia*, 2024. indirizzo: https://en.wikipedia.org/w/index.php?title=K-mer&oldid=1233945106.

[4] Wikipedia contributors, *Contig — Wikipedia, The Free Encyclopedia*, [Online; accessed 20-August-2024], 2022. indirizzo: https://en.wikipedia.org/w/index.php?title=Contig&oldid=1080344663.

[5] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson e P. Medvedev, «On the representation of de Bruijn graphs,» in *Research in Computational Molecular Biology: 18th Annual International Conference, RECOMB 2014, Pittsburgh, PA, USA, April 2-5, 2014, Proceedings 18*, Springer, 2014, pp. 35–55.

[6] P. Wahl, D.-S. Ly-Gagnon, C. Debaes, D. A. Miller e H. Thienpont, «B-CALM: An open-source GPU-based 3D-FDTD with multi-pole dispersion for plasmonics,» *Optical and Quantum Electronics*, vol. 44, pp. 285–290, 2012.

[7] J. Khan, M. Kokot, S. Deorowicz e R. Patro, «Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2,» *Genome biology*, vol. 23, n. 1, p. 190, 2022.

[8] A. Rahman e P. Medevedev, «Representation of k-mer sets using spectrum-preserving string sets,» *Journal of Computational Biology*, vol. 28, n. 4, pp. 381–394, 2021.

[9] Holley, G., Melsted, P. Bifrost, *highly parallel construction and indexing of colored and compacted de Bruijn graphs*, 2020. indirizzo: https://doi.org/10.1186/s13059-020-02135-8.

[10] A. Moffat, «Huffman coding,» *ACM Computing Surveys (CSUR)*, vol. 52, n. 4, pp. 1–35, 2019.

[11] M. Bartík, S. Ubik e P. Kubalik, «LZ4 compression algorithm on FPGA,» in *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, IEEE, 2015, pp. 179–182.

[12] A. Cracco e A. I. Tomescu, «Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT,» *Genome Research*, vol. 33, n. 7, pp. 1198–1207, 2023.

[13] H. Mohamadi, J. Chu, B. P. Vandervalk e I. Birol, «ntHash: recursive nucleotide hashing,» *Bioinformatics*, vol. 32, n. 22, pp. 3492–3494, 2016.

[14] E. Rossignolo e M. Comin, *USTAR: Improved Compression of k-mer Sets with Counters Using de Bruijn Graphs*, X. Guo, S. Mangul, M. Patterson e A. Zelikovsky, cur. Singapore: Springer Nature Singapore, 2023, pp. 202–213.

[15] D. E. Knuth, «Postscript about NP-hard problems,» *ACM SIGACT News*, vol. 6, n. 2, pp. 15–16, 1974.

[16] G. E. Pibiri, «On weighted k-mer dictionaries,» *Algorithms for Molecular Biology*, 2023. indirizzo: `https://doi.org/10.1186/s13015-023-00226-2`.

[17] «SSH Dictionary,» in *Pibiri in Bioinformatics 38*, 2022, pp. 185–194.

[18] B. D. Shibuya Yoshihiro, «Space-efficient representation of genomic k-mer count tables,» *Algorithms for Molecular Biology*, 2022. indirizzo: `https://doi.org/10.1186/s13015-022-00212-0`.

[19] D. E. Wood, J. Lu e B. Langmead, «Improved metagenomic analysis with Kraken 2,» *Genome biology*, vol. 20, pp. 1–13, 2019.

[20] L. Depuydt, L. Renders, T. Abeel e J. Fostier, «Pan-genome de Bruijn graph using the bidirectional FM-index,» *BMC bioinformatics*, vol. 24, n. 1, p. 400, 2023.

[21] G. F. Italiano, N. Prezza, B. Sinaimeri e R. Venturini, «Compressed weighted de Bruijn graphs,» in *CPM 2021-32nd Annual Symposium on Combinatorial Pattern Matching*, vol. 191, 2021, pp. 1–16.

[22] B. A. O. T. S. K. S. T., «Succinct de Bruijn graphs.,» *international workshop on algorithms in bioinformatics (WABI). Berlin: Springer; p. 225–35.*, 2012.

[23] M. Genuzio, G. Ottaviano e S. Vigna, «Fast scalable construction of ([compressed] static | minimal perfect hash) functions,» *Information and Computation*, vol. 273, p. 104 517, 2020, DCC (Data Compression Conference) 2018, issn: 0890-5401. doi: https://doi.org/10.1016/j.ic.2020.104517. indirizzo: https://www.sciencedirect.com/science/article/pii/S0890540120300043.