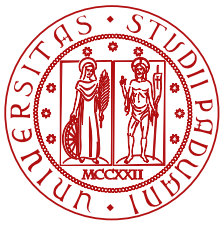

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Compressione efficiente di k-mers sets: presentazione dei metodi e analisi comparativa

Relatore
Prof. Comin Matteo
Università degli studi di Padova

Laureanda
Bellesso Beatrice
Matricola: 2032453

ANNO ACCADEMICO 2023-2024

Data di laurea 23/09/2024

Abstract

La quantità di dati prodotta dai laboratori bioinformatici cresce sempre più ogni anno, stimando per il 2025 ordini dei zettabytes. Risulta quindi cruciale per la ricerca biologica e bioinformatica una gestione efficiente di tali output.
Questa tesi esplora e confronta due approcci distinti per la compressione di dati bioinformatici: la compressione singola e la compressione simultanea. In particolare verranno studiati e analizzati due software di compressione: Gzip, un programma di compressione general purpose, e GGCAT, un nuovo programma di compressione altamente specializzato per dati genomici e biologici. Se Gzip è altamente utilizzato per la sua velocità e scalabilità, GGCAT riesce a sfruttare le particolaritò dei dati bioinformatici come la ripetizione di sequenze genomiche e il formato dei file FASTA.
L'obiettivo di questo studio è paragonare i due approcci in termini di tasso di compressione, spazio salvato, memoria richiesta e tempo di esecuzione, evidenziandone i cambiamenti a seconda del software scelto.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 DNA

L'acido desossiribonucleico, o DNA, è una struttura molecolare presente nelle cellule sia procariote che eucariote e in molti virus. Il compito del DNA è di codificare l'informazione genetica per la sua trasmissione e per la generazione di proteine.
Questa molecola ha una struttura tridimensionale rappresentata da una doppia elica arrotata. Ciascun filamento è costituito da una catena di nucleotidi, a loro volta composti da uno zucchero desossiribosio (da qui il nome acido desossiribonucleico) a cui è legato un gruppo fosfato e una base azotata fra adenina, guanina, citosina e timina (ovvero rispettivamente A, C, T, G).

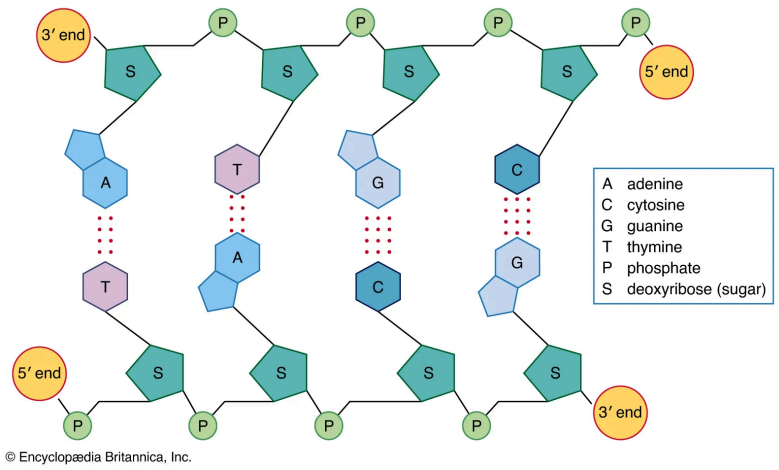

Figura 1.1: Rappresentazione dei due filamenti antiparalleli e dei legami esistenti tra le varie basi azotate.[1]

1.2 Sequenziamento del DNA

In generale con sequenziamento del DNA intendiamo il processo attraverso cui riusciamo a determinare l'ordine dei nucleotidi che compongono il genoma, ossia il DNA contenuto nella cellula.
Naturalmente la grandezza del genoma, ovvero il numero di basi azotate da cui è costituito, varia da organismo a organismo: se le basi del batterio *E. coli* sono all'incirca 5 milioni, questo numero aumenta vertiginosamente fino a 3 miliardi considerando il genoma umano. Il costo di questa tecnologia è proporzionale alla taglia del genoma da sequenziare.
È esemplificativo il *The Human Genome Project*, ovvero il progetto con obiettivo il sequenziamento del genoma umano lanciato nel 1990 e terminato nel 2003. All'epoca il costo del solo sequenziamento, sebbene difficile da determinare con esattezza, variava tra i 500 milioni e 1 miliardo di dollari. [2]
Nel corso dei decenni la spesa è notevolmente diminuita e la facilità di riperimento di dati genomici aumentata, grazie all'avvento di nuove tecnologie di sequenziamento, tra cui il *Next Generation Sequencing*.

1.2.1 Next Generation Sequencing

Come già accennato il Next Generation Sequencing, abbreviato in NGS, ha rivoluzionato il mondo della biologia rendendo possibile studiare le strutture genomiche, le variazioni genomiche e le attività dei geni in maniera estremamente efficiente dal punto di vista di costi, velocità e scalabilità. Tutto questo può essere applicato sia a DNA che a RNA, l'acido ribonucleico, sequenziando in modo altamente parallelo milioni di frammenti.
Sebbene oggi siamo nell'era del sequenziamento di Terza Generazione, l'avvento della tecnologia NGS ha avuto un impatto significativo, determinando la più drastica riduzione dei costi e lasciando una *legacy* di grande valore simbolico.

Fasi del sequenziamento

L'NGS comprende un insieme di tecnologie che però sono pasate su due fasi principali:

- Amplificazione PCR [1]
- Sequenziamento

Prendiamo come caso di studio la tecnologia di sequenziamento di Illumina che si fa di tre punti:

[1] dall'inglese *Polymerase Chain reaction*

- Library Preparation
- Amplificazione
- Sequenziamento

Library Preparation

In seguito alla frammentazione del DNA, con lunghezza del frammento variabile a seconda della tecnologia utilizzata, c'è la fase di library preparation.
In questo step il campione di DNA viene arricchito con primers, indici e adattatori.

Amplificazione

I frammenti risultanti dalla fase precedente vengono denaturati e in seguito rilasciati sulla piastra. Sulla piastra sono presenti degli oligonucleotidi, ovvero sequenze di pochi nucleotidi, che grazie alla complementarietà fra questi fanno sì che i frammenti di DNA aderiscano alla piastra. In seguito i frammenti si ripiegheranno a ponte (da cui il nome della fase *bridge amplification*). Un primer si occuperà di sintetizzare il filamento complementare al frammento in esame. A lungo andare si formano vari cluster di frammenti di DNA e dei loro complementari.

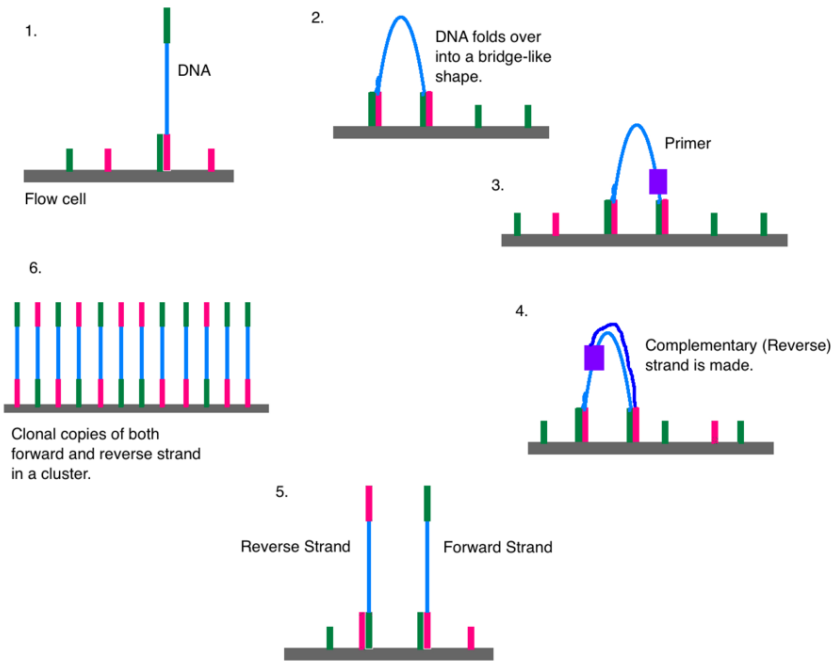

Figura 1.2: Schema degli steps dell'amplificazione[3]

Sequenziamento

A questo punto i campioni di genoma vengono letti. Per far ciò, Illumina utilizza dei nucleotidi modificati e marcati con un colorante fluorescente. Ad ogni ciclo lo stesso nucleotide viene incorporato da tutte le sequenze nello stesso cluster, questo processo avviene in parallelo per tutti i cluster di frammenti. Ogni cluster produce come output una *read*.

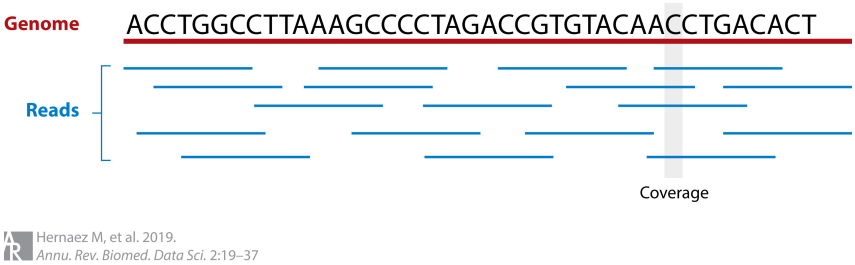

Figura 1.3: Le reads rappresentano parti del genoma sequenziato. La copertura invece indica quante reads ha generato un punto random nel genoma. [4]

Formato dei dati prodotti dal sequenziamento

Nello specifico i dati prodotti dal sequenziamento del genoma vengono salvati come un file di testo in vari formati, come ad esempio il formato *fasta*.

```
>SRR001665.2 071112_SLXA-EAS1_s_4:1:1:657:649 length=36
GCAGAAAATGGGAGTGAAAATCTCCGATGAGCAGCT
>SRR001665.3 071112_SLXA-EAS1_s_4:1:1:708:653 length=36
GAGAGAGCAGTGGGCGAGGTTGGGACATGTCATGAT
>SRR001665.4 071112_SLXA-EAS1_s_4:1:1:675:644 length=36
GAACATTTATTATAATCCTATTCAATTATAATAATC
>SRR001665.5 071112_SLXA-EAS1_s_4:1:1:721:668 length=36
GCTGTAGATCTGGAAATCGCAACGGAGGAAGAAAGA
>SRR001665.6 071112_SLXA-EAS1_s_4:1:1:748:638 length=36
GACACTGTTCATGCTGGTGTCGCTGTCGGGCATTAT
>SRR001665.7 071112_SLXA-EAS1_s_4:1:1:747:721 length=36
GGTCAATGTTGCAATATTTGAGCGCTGCGCGTGCAG
>SRR001665.8 071112_SLXA-EAS1_s_4:1:1:543:387 length=36
GCAATGTAAATCGAAATCATGTTCACTTTGTATCAT
>SRR001665.9 071112_SLXA-EAS1_s_4:1:1:483:751 length=36
GGAATTTGTAGGCCTGATAAGACGCGCAAGCGTCGC
>SRR001665.10 071112_SLXA-EAS1_s_4:1:1:712:602 length=36
GTAAGTTTGATGGCCCCTCGAATAGTTCAATTTTTT
>SRR001665.11 071112_SLXA-EAS1_s_4:1:1:725:648 length=36
GCCTGCGTGGAAAGCTTCTTTCATTCGCTGAAAGTG
>SRR001665.12 071112_SLXA-EAS1_s_4:1:1:534:122 length=36
GATTAAAACCATAATTGGTTTCATGTTGTTGCAGGC
>SRR001665.13 071112_SLXA-EAS1_s_4:1:1:700:599 length=36
GCACATGCGATGGTGGGGACTTTCCTCGGCATTTTA
>SRR001665.14 071112_SLXA-EAS1_s_4:1:1:697:622 length=36
GGCTATCTTGAAGCCAATGAGTTGTTAACTGGCAAG
>SRR001665.15 071112_SLXA-EAS1_s_4:1:1:714:738 length=36
```

Figura 1.4: Esempio di un file in formato *fasta*. Ciascuna read inizia con una linea header indicata dal carattere >, seguita dall'ID e dalla lunghezza della sequenza. La linea successiva esprime la sequenza biologica.

Capitolo 2

Compressione di dati

2.1 Big Data nella bioinformatica

La diminuzione del costo del sequenziamento e contemporaneamente la velocità di computazione dell'output ha fatto in modo che gli studi sul genoma, e di conseguenza anche la mole di dati prodotta, aumentassero esponenzialmente.
Secondo lo studio [4] condotto da Hernaez, Pavlichin, Weissman e Ochoa, l'andamento di crescita segue una legge per cui i dati di sequenziamento si raddoppiano ogni sette mesi: il risultato è più di un exabyte[1] di dati provenienti da sequenziamento stimato all'anno, raggiungendo grandezze dell'ordine di zettabytes [2] nel 2025.
È evidente la difficoltà di gestione di questa mole di dati, soprattutto per quanto ne riguarda l'archiviazione. La sola compressione di questi file non è più necessaria, questa dev'essere anche efficiente dal punto di vista di spazio risparmiato, velocità e memoria impiegata.

2.2 Compressione di dati

La compressione è un concetto noto, almeno a grandi linee, a quasi tutti coloro che fanno utilizzo di una macchina di elaborazione.
Con la compressione dei dati riusciamo, per mezzo di algoritmi di elaborazione dati, a ridurre il numero di bit necessari per rappresentare un'informazione in formato digitale. Nella bioinformatica, come in altre discipline, abbiamo lo scopo di comprimere dati per ridurre la dimensione di un dato file, in modo da diminuirne lo spazio di archiviazione, senza però compromettere la qualità dell'informazione. In base al tipo di file su cui operare distinguiamo principalmente due tecniche di compressione:

[1] Il prefisso exa indica il fattore 10^{19}, ovvero un exabyte corrispende a un miliardo di gigabytes
[2] Il prefisso zetta indica il fattore 10^{21}, dunque un zettabyte corrisponde a un trilione di gigabytes.

- Compressione di dati senza perdita
- Compressione di dati con perdita

Compressione di dati senza perdita

La compressione di dati senza perdita[3] consiste nel rappresentare l'informazione in modo perfettamente invertibile, e dunque senza perdita di informazione.
È utilizzata in caso di compressione di file di testo, programmi, database ed esempi di algoritmi che rientrano in questa categoria sono le codifiche entropiche o le codifiche a dizionario.

Compressione di dati con perdita

Per i file di dimensioni molto grandi come i file multimediali, si utilizza spesso la tecnica di compressione con perdita[4] che permette di ottenere un file più leggero, compromettendo in piccola parte però la qualità.
Un esempio di algoritmi che ne fanno uso sono gli algoritmi basati sulla trasformata discreta del coseno (DCT).

2.2.1 Compressione general purpose e compressione specifica

Un fattore determinante nel risultato della compressione è la tipologia di file da comprimere: si possono sfruttare i pattern e le peculiarità di uno specifico tipo di dato per raggiungere livelli di compressione maggiori. Questo approccio è sfruttato da programmi di compressione specifica. Esistono anche programmi di compressione generici applicabili a qualsiasi tipologia di dati, proprio perché non fanno affidamento sulle strutture specifiche del dato.
Se da una parte la compressione general-purpose è estremamente versatile e utilizzabile da chiunque, non raggiunge la stessa efficienza di programmi di compressione specifica per il dato in esame.
In seguito verranno studiate anche le differenze di compressione tra un tool general-purpose come Gzip e GGCAT, un programma di compressione specifica per i dati bioinformatici.

2.2.2 Parametri utilizzati per l'analisi della bontà di compressione

Nello specifico di questo studio per effettuare i confronti sulle compressioni effettuate sono stati considerati i seguenti parametri:

[3] Anche detta compressione *lossless*.
[4] In Inglese codifica *lossy*.

- **Compression ratio**

$$CR = \frac{\textit{Dimensione dei dati non compressi}}{\textit{Dimensione dei dati compressi}} \tag{2.1}$$

  In questo caso un dato compresso avrà una *CR* maggiore o uguale a 1. Maggiore il valore, migliore la compressione.

- **Risparmio di spazio**

$$\textit{Space saving} = \left(1 - \frac{\textit{Dimensione dei dati compressi}}{\textit{Dimensione dei dati non compressi}}\right) \times 100\% \tag{2.2}$$

  Ovvero la percentuale di spazio salvato.

- **Memoria RAM**
  Ovvero la memoria RAM utilizzata dal processo.

- **Tempo**
  Ovvero il tempo impiegato dal processo per terminare.

Un buon programma di compressione avrà una *compression ratio* e un risparmio di spazio elevati, mentre la quantità di memoria utilizzata e il tempo impiegato saranno relativamente bassi.

Capitolo 3

Aspetti preliminari

Questo capitolo avrà come obiettivo approfondire alcuni concetti teorici su cui si basano gli studi e vari programmi bioinformatici, tra cui anche il software GGCAT presentato nel capitolo successivo.

3.1 K-mers

Definiamo un k-mer come una stringa di lunghezza k su un alfabeto, che nel nostro caso sarà $\Sigma = \{A, C, G, T\}$.
Nella pratica i k-mers sono le sottosequenze in cui dividiamo le read genomiche per poterle studiare meglio. L'insieme dei k-mers ottenuti da una data sequenza genomica è definito come k-mers set.

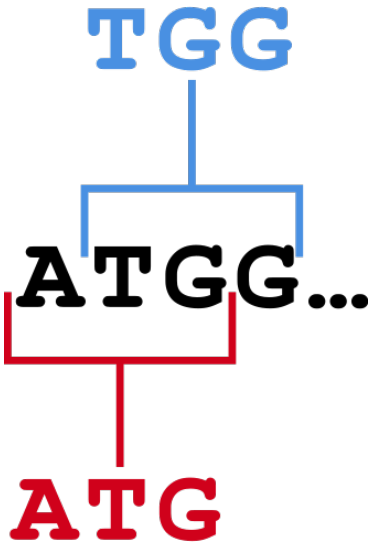

Figura 3.1: Considerando la sequenza biologica ATGG e un k=3, otteniamo i k-mers ATG e TGG. L'insieme {ATG, TGG} rappresenta il k-mers set della data sequenza[6].

3.1.1 Utilizzo dei k-mers nella bioinformatica

Il concetto di k-mer è popolare nel campo della bioinformatica: hanno reso possibile elaborare grandi quantità di dati genomici complessi in tempo e complessità ragionevoli.
Utilizzando i k-mers introduciamo un'ulteriore frammentazione nelle reads ottenute dal sequenziamento. Questo consente di trovare delle parole di riferimento nel genoma: ricordiamo che questo si presenta come una sola sequenza di caratteri su un alfabeto, senza definire un inizio e una fine. I k-mers riescono a fornire al genoma una struttura ottenendo diversi benefici, tra cui il miglioramento del rapporto segnale/rumore: un errore nel sequenziamento provoca un mismatch con il template, tuttavia un solo sottoinsieme dei suoi k-mers conterrà l'errore (nello specifico una read di lunghezza L contiene $L\text{-}k\text{+}1$ k-mers, tra questi solo k saranno sfalsati mentre il resto rappresenterà correttamente la sequenza genomica).
In generale i k-mers sono uno strumento versatile che ha permesso l'affinamento di varie tecnologie utilizzate in molti tools per diversi approcci:

- *K-mer counting*[7]

- Analisi delle frequenze[8]
  I k-mers possono essere utilizzati per stimare la taglia e la complessità del genoma, in modo da ottimizzare i processi successivi.

- Indicizzazione delle sequenze[9]
  I k-mers possono essere utilizzati per la ricerca di sequenze, esatte o approssimate, in datasets: i metodi attuali consentono di determinare la presenza o meno di qualsiasi k-mer all'interno di collezioni fino a circa 2500 datasets.

- *Genome assembly*[10]
  Come abbiamo visto precedentemente il sequenziamento produce un output frammentato: frammentandolo ulteriormente possiamo sfruttare gli overlap tra i k-mers per assemblare sequenze contigue.

- Allineamento di sequenze[11]

- Classificazione tassonomica[12]
  Studiando le frequenze di determinati k-mers possiamo stabilire l'appartenenza o meno tra vari ranks tassonomici.

- Ricostruzione filogenetica[13]

- Proteomica[14]

- Databases[15]
  Esistono databases di k-mer che forniscono sequenze presenti o meno in ciascuna specie.

3.1.2 Proprietà

La scelta del valore *k* influisce su due proprietà essenziali dei k-mers: la complessità del k-mer set e la copertura del k-mer. Consideriamo la complessità come il numero di diversi k-mers distinti dato un valore di *k*. Nel caso genomico con un alfabeto dato dalle quattro basi azotate, la complessità cresce esponenzialmente all'aumentare di *k* secondo la legge 4^k. Nella pratica però le sequenze complementari *reverse* e le sequenze *forward* vengono contate insieme, dunque consideriamo solo i k-mers canonici.
Un valore di *k* troppo piccolo (ovvero $k \leq 11$) non consente di rappresentare appieno lunghe sequenze. Valori alti da contenere quantità di k-mers unici a quella posizione nel genoma invece permettono di stimare la copertura C_k, ovvero quante reads in media contengono un k-mer. Definiamo la copertura C_k come

$$C_k = \frac{C_g \times (L - k + 1)}{L} \tag{3.1}$$

dove C_g è la profondità media di lettura per base in un genoma e viene definita come

$$C_g = \frac{N \times L}{G} \tag{3.2}$$

dove N è il numero di reads sequenziate, L è la lunghezza media della read e G è la taglia totale del genoma.
Dunque valori di *k* più bassi garantiranno una copertura più alta.
Abbiamo già parlato precedentemente di come i k-mers permettano di migliorare il tasso segnale/rumore. Gli errori nel sequenziamento si propagano dalle reads ai k-mers a seconda del valore di k: assumendo errori uniformemente distribuiti, la probabilità che un k-mer rappresenti la reale sequenza genomica è

$$P\,(\textit{Error free k-mer}) = (1 - e)^k \tag{3.3}$$

dove *e* è il tasso di errore di sequenziamento per base. Con un *k* elevato dunque otteniamo una probabilità di avere un k-mer error free più alta.
La scelta di *k* è un trade-off tra la complessità, copertura e tasso di errori: dunque il valore giusto dipende dall'applicazione. Per datasets di read corte tipicamente si sceglie un *k* appartenente all'intervallo $[21, 31]$. Inoltre scegliere $k \leq 31$ permette alla macchina di rappresentarlo in meno di 64 bits, rendendo il tempo di confronto minore.

3.2 Rappresentazione di set di k-mers

Vista l'elevata quantità di dati raccolti dai sequenziamenti, il numero di k-mers sets aumenta di conseguenza. È quindi necessario minimizzare lo spazio di archiviazione e il tempo di elaborazione.
Possiamo suddividere le strutture dati in:

- strutture dati non specializzate (come le *hash tables*) e applicate ai k-mer sets senza essere modificate
- strutture dati non specializzate adattate all'uso su k-mer sets
- strutture dati appositamente progettate per i k-mer sets

La rappresentazione più semplice è una lista ordinata di k-mers. Al tempo stesso però è inefficiente: il tempo di costruzione è $O(nk)$ utilizzando un qualsiasi algoritmo di ordinamento di stringhe lineare, mentre lo spazio occupato è $\Theta(nk)$.
Altre scelte comuni sono gli alberi di ricerca binaria e le hash tables. In particolare quest'ultime richiedono in generale un tempo di *query* di $O(k)$, utilizzando invece funzioni di hash *rolling*[1] questo tempo si riduce a $O(1)$ per le query su k-mer forward e backward. Il problema è lo spazio richiesto, ovvero $\Theta(nk)$, proibitivo per applicazioni intensive vista la dipendenza lineare da *k*.
Spesso si ricorre quindi all'utilizzo del grafo di de Bruijn, un grafo costruito su un set di k-mers S. Un grafo di de Bruijn può essere implementato in vari modi, come utilizzando i Bloom filters oppure basandosi su hash tables. A prescindere dalla sua implementazione, questo tipo di grafo rimane spesso una scelta valida.

3.2.1 Grafo di de Bruijn

I Grafi di de Bruijn sono tra le strutture dati fondamentali della bioinformatica. Le loro applicazioni sono molteplici come ad esempio l'analisi e la fase di assemblaggio dei dati provenienti dal sequenziamento di DNA e RNA.
Un grafo può essere *node-centric* quando l'insieme di nodi è dato dall'insieme di k-mers S e c'è un arco da *x* a *y* se gli ultimi $k-1$ caratteri di *x* sono uguali ai primi $k-1$ caratteri di *y*. Un grafo è *edge-centric* se invece l'insieme di nodi è dato dall'insieme dei (k-1)-mers presenti in S e per ciascun $x \in S$ c'è un arco da $x[1, k-1]$ a $x[2, k]$. Sostanzialmente quindi i k-mers appartenenti a S sono nodi nel grafo node-centric e archi nel grafo edge-centric. Entrambi i grafi rappresentano la stessa informazione.

[1] In una funzione di hash rolling conosciamo il valore per un k-mer *x* e possiamo calcolare il valore hash per qualsiasi estensione forward e backward in $O(1)$

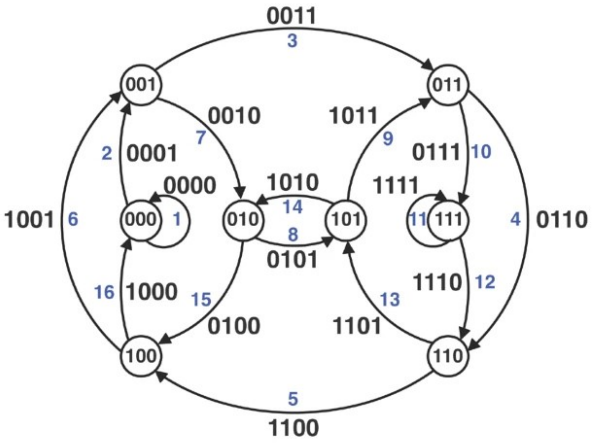

Figura 3.2: Un grafo di de Bruijn per stringhe definite su un alfabeto $\Sigma = \{0, 1\}$ con $k = 4$. Ciascun arco collega vertici con prefisso-suffisso comuni.[16]

Una variante di questa struttura sono i grafi di de Bruijn colorati[2]. In origine questi particolari grafi sono stati introdotti per il *de novo assembly*[3] e la genotipizzazione delle varianti, ma in seguito estesi a moltre altre applicazioni. I grafi di de Bruijn colorati sono costruiti su una collezione di sequenze e associano ad ogni k-mer un ID (il colore) che specifica la sequenza a cui il k-mer appartiene.
In generale i grafi di de Bruijn sono strutture molto utili per vari motivi. Innanzitutti si possono associare ai grafi delle *abundance tresholds*, ovvero delle soglie che fungono da filtri, eliminando k-mers con frequenze troppo basse che derivano da errori di sequenziamento. Dopodiché la struttura a grafo permette di risparmiare spazio di rappresentazione poiché le sottostringhe uguali, provenienti da regioni ripetute, vengono rappresentate una sola volta. Infine dai percorsi del grafo si può stabilire la presenza di regioni *variation-free*.

Unitigs massimali

Come già accennato i percorsi del grafo sono di interesse gli studi sulla presenza di regioni senza mutazioni. I paths di interesse sono i percorsi *non-branching* massimali, ovvero i più

[2] *Colored de Bruijn graphs.*
[3] Ovvero l'assemblaggio di sequenze corte in reads più lunghe senza l'aiuto di un genoma di riferimento.

lunghi paths i cui nodi interni hanno un grado in entrata e un grado in uscita entrambi uguali a uno: questi percorsi vengono chiamati unitigs massimali.
In un grafo di de Bruijn possiamo pensare di sostituire ciascun unitig con un arco contrassegnato con un'etichetta rappresentativa dell'unitig, ottenuta identificando i (k-2)-mers sovrapposti. In questo modo otteniamo un grafo con le stesse informazioni, ma di una taglia ben minore: i grafi risultanti vengono chiamati grafi di de Bruijn compatti.
Quindi la lista di unitigs massimali è essa stessa una rappresentazione di un k-mer set che occupa meno spazio in quando un set di *x* k-mers viene rappresentato utilizzando $k - 1 + x$ caratteri, mentre il set originario di k-mers ne utilizza $k \cdot l$. Dunque il calcolo degli unitigs massimali di un k-mer set costituisce uno step di compressione lossless.

Rappresentazione plain text

Siccome gli unitigs non presentano diramazioni nei nodi interni, preservano la topologia del grafo.
Possiamo pensare di utilizzare una rappresentazione ancora più leggera dei grafi ignorandone la topologia, dunque una rappresentazione plain text[4]. Più formalmente una rappresentazione plain text consiste in un insieme di stringhe che contiene solo ciascun k-mer presente nella sequenza di input. Gli autori dell'articolo [17] definiscono tale insieme come un *spectrum preserving string set*(SPSS).
Il vantaggio della rappresentazione plain text consiste nell'abbassamento di RAM utilizzata e anche in prestazioni più veloci dei tools di analisi, questo però se tali tools riescono ad utilizzarla senza modifiche. Nel caso ci fosse la necessità di decomprimere la rappresentazione prima dell'utilizzo probabilmente si eliminerebbero i risparmi di RAM e potrebbe verificarsi overhead di runtime.

[4]Ovvero una rappresentazione dei caratteri leggibili, dunque non include rappresentazione grafica o altri oggetti.

Capitolo 4

Presentazione dei tools

In questo capitolo verranno presentati i tools utilizzati per questo studio: GZip, un software di compressione general-purpose, e GGCAT, un software di compressione specifica per dati bioinformatici.

4.1 Gzip

Gzip è un programma di compressione general-purpose utilizzato in tutto il mondo da milioni di utenti. GNU Gzip è stato scritto da Jean-Ioup Gailly per il progetto GNU, mentre la parte di decompressione è stata scritta da Mark Adler.
Permette di ridurre la dimensione di un file mantenendo *mode*, *ownership* e *timestamp* del file originale. È inoltre l'algoritmo utilizzato dai web server per comprimere le pagine prima di mandarle al browser client.
Per gli scopi di questa tesi non approfondiremo gli aspetti implementativi di questo software.

4.2 GGCAT

GGCAT è un programma di costruzione sia di grafi di de Bruijn compatti sia di grafi di de Bruijn compatti e colorati, e in questo caso permette di associare ciascun k-mer alle sequenze in cui questo appare.

4.2.1 Stato dell'arte

Per quanto riguarda il calcolo degli unitigs massimali lo stato dell'arte è rappresentato da Cuttlefish 2[23]. Questo tool parte con una fase di k-mer counting utilizzando l'algoritmo KMC3[24] che impiega una funzione di hash perfetta, ovvero una funzione di hash che mappa elementi

distinti appartenenti ad un insieme generico in un insieme di *m* interi, non causando collisioni. In particolare usa un approccio *automaton-based* per calcolare lo stato di diramazioni di ciascun (k-1)-mer usando meno informazioni possibili (zero, uno o più nodi vicini a destra/sinistra). Poi costruisce il grafo basandosi sull'automata di ciascun (k-1)-mer, allungando l'unitig se il (k-1)-mer corrente non ha un branch forward e se il successivo (k-1)-mer non ha un branch backward. Cuttlefish 2 fa un grande uso della memoria su disco, il che influisce sulle sue prestazioni.
Per quanto riguarda invece il calcolo degli unitigs massimali con associate informazioni riguardo al colore, lo stato dell'arte è BiFrost[10]. Viene utilizzato un approccio totalmente *in-memory*, con uso di blocked Bloom filters parzialmente indicizzati da *minimizers* che approssimano i k-mers presenti nel grafo finale. I falsi archi creati dai filtri poi vengono eliminati ripassando l'input originale. I k-mers poi vengono salvati internamente raggruppati dai minimizers, in questo modo gli inserimenti e cancellamenti sono relativamente veloci. Tuttavia i filtri non sono cache efficient e il tempo di costruzione del grafo è elevato, inoltre vengono salvati insiemi di colori ridondanti (k-mers che condividono lo stesso set di colori vengono codificati come due set separati).
In entrambi i casi di applicazione GGCAT supera le prestazioni dei tools precedenti.

4.2.2 Panoramica

GGCAT, a differenza di Cuttlefish 2 e altri tools, unisce le fasi di k-mer counting e costruzione degli unitigs. Dopo aver suddiviso l'input in buckets e averne calcolato gli unitigs globali, questi vengono compressi con l'algoritmo lz4 prima di essere scritti su disco riducendone lo spazio di archiviazione.
Viene anche evitata una struttura dati *union-find*[1] utilizzando piuttosto uno step di unioni tra buckets che garantisce risultati esatti stimando un tempo di esecuzione basso. Infine l'algoritmo è stato diviso in unità di esecuzioni più piccole, eseguite in parallelo.

4.2.3 Implementazione

Definizioni

Riprendiamo il concetto di unitigs mostrato nel capitolo precedente, ma consideriamolo in un grafo edge-centric. Gli autori di [19] definiscono un unitig come un path del grafo di de Bruijn di un un multiset R di stringhe contenente almeno un arco tale che tutti i nodi interni del path (ad esclusione del primo e dell'ultimo) abbiano un grado in entrata e in uscita uguale ad uno. Inoltre per lo studio condotto dagli stessi autori, non vengono considerati paths che ripetono nodi ad

[1] Sono chiamate anche strutture dati *disjoint-set*. Sono strutture dati che riescono a tener traccia di una collezione di insiemi digiunti *pairwise* contenenti un totale di n elementi.

eccezione del primo o dell'ultimo: in tal caso il path è un ciclo.
Seguiranno alcune definizioni di concetti preliminari per questo software.
Dato un multiset di stringhe R e una stringa x, definiamo come $occ(x, R)$ il numero di occorrenze di x nelle stringhe di R, contando indivualmente ciascuna occorrenza nella stessa stringa in R. Data una stringa $x \in \Sigma$, indichiamo come $x^{-1} \in \Sigma$ il reverse di x. Se $x \neq x^{-1}$ definiamo

$$occ_{cn} = occ(x, R) + occ(x^{-1}, R) \tag{4.1}$$

altrimenti $occ_{cn} = occ(x, R)$. Definiamo ulteriormente $app(x, R) = min(1, occ(x, R))$ e $app_{cn} = min(1, occ_{cn}(x, R))$.
Dati R e U, due multiset di stringhe, diciamo che R e U hanno lo stesso *k-mer set* se ciascun k-mer che appare in uno dei sets appare anche nell'altro set. Allo stesso modo R e U hanno lo stesso *canonical k-mer set* se ciascun k-mer q che appare in uno dei sets, q oppure q^{-1} appare nell'altro set. Più formalmente esprimiamo il concetto dello stesso k-mer set non canonico tra R e U come

$$\forall q \in \Sigma^k \, occ(q, R) \geq 1 \text{ iff } occ(q, U) \geq 1 \tag{4.2}$$

Allo stesso modo R e U hanno lo stesso canonical k-mer set se

$$\forall q \in \Sigma^k \, occ_{cn}(q, R) \geq 1 \text{ iff } occ_{cn}(q, U) \geq 1 \tag{4.3}$$

Vogliamo ora dare una definizione alternativa di unitig massimale che non faccia esplicitamente riferimento al grafo di de Bruijn. Per far ciò partiamo dal caso di k-mers sets non canonici, dunque senza la presenza del reverse complementare.
Come ipotesi, dato l'insieme U di unitigs massimali del multiset di stringhe R, richiediamo che tutte le stringhe in U abbiano lunghezza almeno k, quindi gli unitigs che contengono almeno un arco. Inoltre R e U devono avere lo stesso k-mer set. Infine, se un (k-1)-mer appare almeno due volte in U, allora non è un nodo interno in nessun unitig: ciò vuol dire che non possono essere uniti due unitigs separati al (k-1)-mer dato che questo (k-1)-mer deve apparire in almeno un'altra stringa in U e dunque

$$\forall q \in \Sigma^{k-1} \, if \, occ(q, U) > 1 \text{ allora } q \text{ appare sono come prefisso o suffisso delle stringhe in } U. \tag{4.4}$$

Per 4.4 sappiamo non esserci due unitigs ciclici equivalenti in U. Ora dobbiamo imporre la massimalità e per far ciò dichiariamo che un (k-1)-mer è un prefisso o suffisso di un unitig se e solo se è un nodo branching, source o sink:

$$\forall q \in ends_{k-1}(U), \sum_{c \in \Sigma} app(q \cdot c, U) \neq 1 \text{ or } \sum_{c \in \Sigma} app(c \cdot q, U) \neq 1. \tag{4.5}$$

A questo punto forniamo la definizione incentrata sulle stringhe di unitig massimale considerando anche i reverse complementari, in modo da avere una definizione più generale su cui verrà basato l'algoritmo del software.

Definizione (Unitigs massimali canonici). *Dato un multiset R di stringhe e interi con k ≥2 e a ≥1, diciamo che un insieme U di stringhe è un insieme di unitigs massimali canonico di R con taglia dei k-mer k e abundance treshold a se valgono le seguenti condizioni:*

0. $\forall x \in U,\ |x| \geq k,\ \forall x, y \in U,\ x \neq y^{-1}$
1. $\forall q \in \Sigma^k\ occ_{cn}(q, R) \geq a$ iff $occ_{cn}(q, U) \geq 1$
2. $\forall q \in \Sigma^{k-1}$, if $occ_{cn}(q, U) > 1$ *allora* q *e* q^{-1} *appaiono solo come prefisso o suffisso delle stringhe in* U
3. $\forall q \in ends_{k-1}(U),\ \sum_{c\in\Sigma} app_{cn}(q \cdot c, U) \neq 1$ or $\sum_{c\in\Sigma} app_{cn}(c \cdot q, U) \neq 1$

Dunque stiamo richiedendo che i due insiemi abbiamo lo stesso k-mer set canonico, che gli unitigs non attraversino (k-1)-mers ramificati e stiamo infine richiedendo la massimalità.
Introduciamo poi il concetto di *minimizer* di un k-mer. Dato un intero $m \leq k$ e una rolling function hash: $\Sigma^m \to \mathbb{Z}$, definiamo il minimizer di un k-mer x come

$$mini(x) = \min_{y\in\Sigma^m \wedge y\in x} \text{hash}(y) \tag{4.6}$$

Quindi consideriamo il minimizer come un valore hash, non tenendo conto della particolare posizione dell' m-mer che ha il minimo valore di hash. Inoltre il valore di m è scelto di default da GGCAT, basando questa scelta sul valore di k. Questa scelta può essere modificata dall'utente. Infine con buckets ci riferiamo a partizioni di dati salvati come un insieme compatto. Partizionare i dati in più buckets permette di avere un approccio parallelizzato elaborando ciascun bucket indipendentemente. Solo i buckets in uso vengono conservati nella memoria principale, diminuendo il consumo di memoria dell'algoritmo.

Read splitting

Partendo dalla read R_j vogliamo ottenere delle sue sottostringhe $S_1, ..., S_l$ che abbiano un overlap di $k-2$ caratteri in modo che tutti i (k-1)-mers di S_i abbiano lo stesso minimizer per tutti gli $i \in \{1, ..., l_j\}$. L'algoritmo riesce a calcolare $S_1, ..., S_l$ in tempo lineare secondo la taglia della read di partenza. Innanzitutto per ciascun m-mer x di R_j viene calcolato $hash(x)$. In seguito viene calcolato il minimo di ciascuna finestra di $k-m$ m-mers consecutivi, che corrispondono ad un (k-1)-mer. Infine i (k-1)-mers che condividono lo stesso minimo vengono raggruppati. Questi tre step avvengono in un pass singolo su R_j.

Per ciascuna sottostringa S_i ottenuta, siano a e b i caratteri di R_j rispettivamente immediatamente precedenti e successivi a S_i in R_j, \$ se questi non esistono. Allora a e b sono chiamati *linking characters*. Considerando la stringa $S'_i := a \cdot S_i \cdot b$[2], osserviamo che S'_{i-1} e S'_i hanno un overlap di k caratteri.

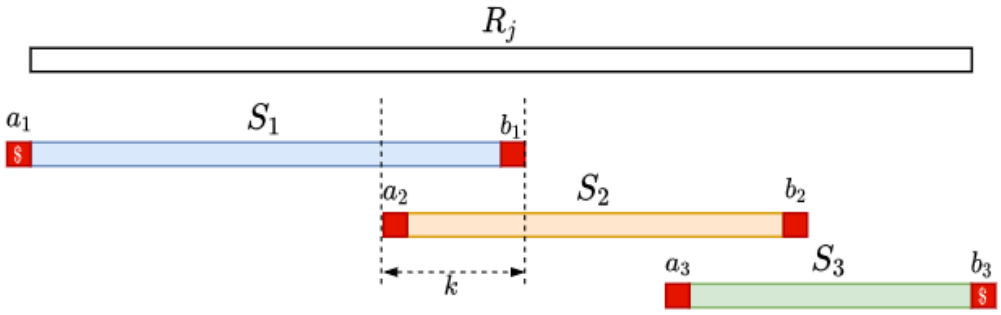

Figura 4.1: La read R_j è stata suddivisa in sottostringhe S_1, S_2 e S_3 in modo che tutti i k-mers di ciascuna sottostringa abbiano lo stesso minimizer. Vengono aggiunti i linking characters, facendo si che l'overlap tra due sottostringhe consecutive sia di k caratteri.[19]

Costruzione degli unitigs intermedi

Gli autori di [19] hanno provato che estendere qualsiasi k-mer x può essere fatto correttamente anche solo facendo una query nel raggruppamento di x. Per ciascun raggruppamento si esegue:

- Un k-mer counting step delle stringhe presenti all'interno del ragruppamento utilizzando una hashmap, questo tenendo conto della presenza o meno di linking characters in un k-mer.
- Guardando all'hashmap, si costruisce una lista di k-mers unici del gruppo che hanno l'*abundance* richiesta.
- Si attraversa la lista di k-mers e per ciascun k-mer x inutilizzato, viene inizializzata una stringa $z := x$ che sarà estesa a destra e a sinistra fin tanto che mantiene le proprietà di un unitig.
  Per estendere z a destra, facciamo una query dell'hashmap per $suf_{k-1}(z) \cdot c$, per tutti i $c \in \{A, C, G, T\}$. Se c'è un'unica estensione y tale che $suf_{k-1}(z) = pre_{k-1}(y)$, allora ricerchiamo nella hashmap $c \cdot pre_{k-1}(y)$ per tutti i $c \in \{A, C, G, T\}$. Se viene trovato esattamete un match allora sostituiamo z con l'unione di z e y, ovvero $z \odot^{k-1} y$, marcando y come usato. Se y non ha linking characters, si ripete l'estenzione con la nuova stringa z. Dopo l'estensione a destra, avviene una estensione a sinistra simmestrica di z. Dopo

[2]Con questa notazione indichiamo la concatenazione fra stringhe

la fase di estenzione, all'untig z viene dato un indice unico id_z. Se l'estensione di z era stata fermata a causa di un linking character nel primo o ultimo k-mer y di z, aggiungiamo (y, id_z) ad una lista L.

Dopo l'elaborazione di tutti i gruppi, per ciascun (y, id_z) in L esiste esattamente un'altra (y, id'_z) in L, questo poiché c'è al più un altro gruppo in cui x appare, inoltre x appare in due gruppi diversi se e solo se $mini(pre_{k-1}(x) \neq mini(suf_{k-1}(x))$.[19]

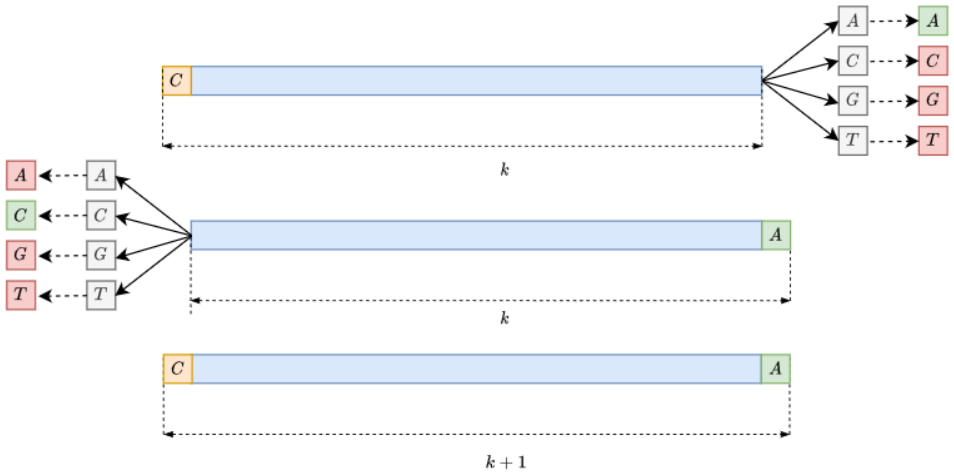

Figura 4.2: In figura viene illustrato lo step di estensione dell'unitigs intermedio all'interno di ciascun gruppo. Per ciascun k-mer cerchiamo una possibile estensione verificando tutti i 4 caratteri possibili. Nell'ultimo k-mer vediamo come abbiamo esteso solo se c'è un unico match sia forwards che backwards. È un processo continuo finché non possono essere effettuate altre estensioni.[19]

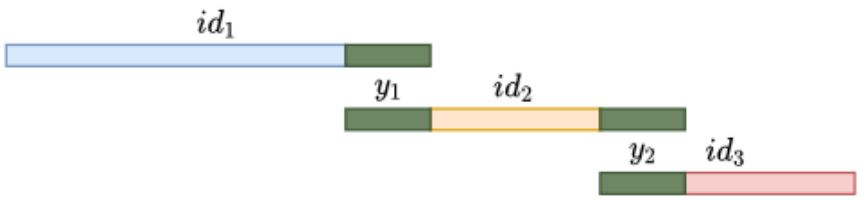

Figura 4.3: La figura mostra il risultato della costruzione intermedia degli unitigs. Tutti gli unitig che hanno una possobile estensione condividono la fine con un altro unitig intermedio.[19]

Unione degli unitigs

Le tuple (y, id_z) in L sono ordinate secondo y in modo tale che le entrate (y, id_z) e (y, id'_z) appaiano conseguentemente. Da qui creiamo una lista (id_z, id'_z), inserita a sua volta in un'ulteriore lista P di $pairs$ di unitigs da unire per ottenere gli unitigs massimali. In questa fase non possono

essere fatti partizionamenti anticipati che inseriscano tutti gli unitigs che costituiranno l'unitig massimale nella stessa partizione. Se altri tools usano strutture dati union-find, GGCAT utilizza un approccio randomizzato per inserire nella stessa partizione gli unitigs che dovrebbero essere uniti, ripetendo il processo fino a che tutti questi sono uniti negli unitigs massimali finali.
La procedura è la seguente: viene allocato un numero fisso di buckets e all'inizio per ciascuna lista in P, entrambe le estremità vengono marcate come *unsealed*. Si ripete fintanto che P non è vuota:

- Per ciascuna lista in P scegliamo casualmente una delle sue estremità unsealed, chiamandola l. Inseriamo la lista nel bucket corrispondente a l, mentre nel bucket corrispondente all'altra estremità r inseriamo un placeholder.

- All'interno di ciascun bucket, ordiniamo le liste secondo l'estremità per cui sono state inserite in quel bucket. Poi uniamo tutte le estremità uguali, formando liste più lunghe. Se un'estremità in un bucket rimane spaiata, e non ha un placeholder corrispondente appartenente ad un'altra lista nel bucket, viene marcata come *sealed*.

- Infine rimuoviamo da P ciascuna lista con entrambe le estremità sealed.

Date due liste in P da unire, c'è una probabilità di almeno $\frac{1}{4}$ che queste siano assegnate allo stesso bucket. Dunque ci si aspetta che il risultato desiderato avvenga dopo 4 tentativi.
Sia L che P vengono salvate in buckets per poter una miglior concorrenza nell'elaborare i dati.

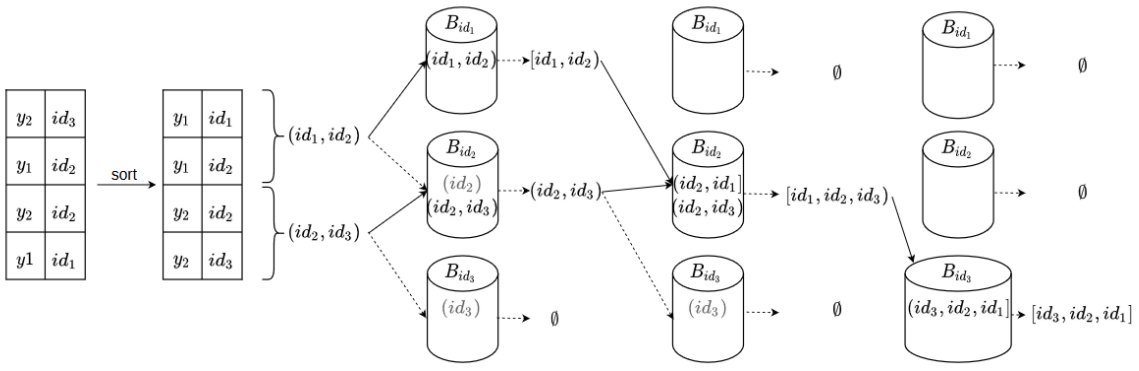

Figura 4.4: La figura rappresenta lo step di unione. Le coppie vengono ordinate, e gli indici di unitigs che hanno la stessa estremità vengono raggruppati in una tupla. Ciascuna tupla viene assegnata ad un bucket corrispondente ad una delle sue estremità unsealed. Per l'indice dell'altra estremità inseriamo un placeholder nel bucket corrispondente. All'interno dello stesso bucket poi le coppie con lo stesso indice di unitig vengono unite formando tuple più grandi.
Come esempio, nella prima fase in B_id1 la coppia (id_1, id_2) è sealed a id_1 poiché per questo non c'è un placeholder. In B_id2 invece la coppia (id_2, id_3) non è sealed a id_2 perché c'è un placeholder.[19]

Considerata la complessità della prova di correttezza dell'algoritmo, allo scopo di questa tesi si è scelto di non includerla. La prova formale è disponibile nello studio [19].

Coloring

Nel calcolo dei colori per ciascun k-mer di un grafo di de Bruijn incontriamo due difficoltà principali:

- Trovare tutti i colori appartenenti a ciascun k-mer
- Salvare i colori in maniera efficiente sia dal punto di vista dello spazio che del tempo

L'idea di GGCAT è unire le informazioni riguardo al colore per i k-mers che condividono lo stesso insieme di colori, evitando confronti dispendiosi dei set interi per ciascun k-mer. Da ciascun k-mer si ottiene una lista C normalizzata di colori. Vengono salvati tutti i colori, anche in modo rindonante, dopodiché vengono ordinati e deduplicati. Viene poi generato un hash h e confrontato con una hashmap globale che associa h ad un indice di sottoinsieme di colori. Se non viene trovato un match allora la lista L viene scritta nella colormap e viene creato un nuovo indice di sottoinsieme di colori per L. Altrimenti l'insieme di colori dev'essere già apparso in un k-mer precedentemente elaborato, dunque viene ritornato l'indice a quell'insieme di colori. Infine, il k-mer nel grafo viene etichettato con il suo corrispettivo indice di sottoinsieme che indica univocamente un sottoinsieme di colori. Avendo codificato ciascun sottoinsieme solo una volta, questo permette una miglior compressione. La codifica viene effettuata con una compressione *run-length* sulle differenze dei colori ordinati del sottoinsieme, successivamente la colormap viene divisa in chunks rendendo l'accesso più veloce e compressa nuovamente con l'algoritmo lz4. Inoltre, scrivendo gli unitigs sul disco, i colori di ciascun header della sequenza di unitigs nel file FASTA vengono marcati codificando run-length anche l'insieme di indici dei colori di tutti i k-mers degli unitigs. Essendo gli unitigs zone variation-free, tendono ad avere solo un piccolo numero di sottoinsiemi di colori possibili associati ai suoi k-mers.

Querying delle sequenze

Le queries vengono eseguite dividendo gli unitigs del grafo in input e le queries nei buckets, con un approccio simile a quello della fase di reads splitting. Indipendentemente per ogni bucket poi, si fa un k-mer counting per trovare il numero di k-mers che soddisfano la query. I risultati intemedi vengono sommati poi per trovare il numero globale di k-mer che offrono un match per la query. Nel caso uncolored, l'output viene presentato in sottoforma di file *.csv* con una linea per ciascuna query di input, contenente numero e percentuale di matching k-mers. Per il caso colored invece, l'output è un file JSON con una linea per ciascuna query contenente, se positivo, il numero di match di k-mers per ciascun colore del grafo.

Capitolo 5

Analisi comparativa e risultati

In questo capitolo verranno riportati i risultati ottenuti comprimendo i datasets e analizzati i fattori di compress ratio, tempo e memoria impiegati dai processi per quanto riguarda la compressione singola di ciascun dataset e la loro compressione simultanea, utilizzando i software GGCAT e Gzip.

5.1 Ambiente di elaborazione

Prima di riportare i dati effettivi, segue una breve descrizione dell'ambiente di elaborazione in modo da contestualizzare i risultati.

5.1.1 Hardware

I processi sono stati eseguiti sul cluster blade del dipartimento di Ingegneria dell'Informazione (DEI) dell'Università degli studi di Padova.
Le richieste vengono gestite secondo uno scheduler Slurm ed eseguite dalla risorsa di calcolo più adeguata in quel momento. In particolare la piattaforma è composta da vari nodi con diversi hardware:

- **runner-[01-03]**: tre nodi con 48 CPUs (4x Intel Xeon Gold 5118 CPU @ 2.30/3.20GHz) e 1.5 TB di RAM.
- **runner-[04-06]**: tre nodi con 72 CPUs (4x Intel Xeon Gold 5220 CPU @ 2.20/3.90GHz), 2 TB di RAM e una GPU Nvidia Quadro P2000.
- **runner[07-10]**: quattro nodi con 96 CPUs (4x Intel Xeon Gold 6252N CPU @ 2.30/3.60GHz) e 3 TB di RAM.

- **runner-11**: 192 CPUs (8x Intel Xeon Platinum 8260 CPU @ 2.40/3.90GHz) e 6 TB di RAM.

- **gpu1**: 24 CPUs (2x Intel Xeon Gold 5118 CPU @ 2.30/3.20GHz), 1TB di RAM e 8 GPUs Nvidia A40.

- **gpu[2-3]**: due nodi con 32 CPUs (2x Intel Xeon Gold 5218 CPU @ 2.30/3.90GHz), 1.5 TB di RAM e 8 GPUs Nvidia RTS 3090.

- **gpu4**: 32 CPUs (2x Intel Xeon Gold 5218 CPU @ 2.30/3.90GHz), 1.5 TB di RAM, 5 GPUs Nvidia RTX 3090 e 3 GPUs Nvidia A40.

- **gpu[5-6]**: due nodi con 64 CPUs (2x Intel Xeon Platinum 8358 CPU @ 2.60/3.40GHz), 1 TB di RAM e 10 GPUs Nvidia A40.

5.1.2 Datasets

Nella tabella 5.1 vengono riportati i dati riguardo ai vari datasets utilizzati, ricavati dalla banca dati NCBI[26].
Come si può ben vedere, i dati sono eterogenei e provenienti da diverse specie e da diverse tecnologie di sequenziamento utilizzando sia il DNA che l'RNA.
Il totale di spazio occupato dai datasets corrisponde a circa 61 GB.

Dataset	Denominazione SRA	Dimensione per dataset in GB
Escherichia coli	SRR001665	2,0856
Human microbiome	SRR062379	10,9532
Soybean; RNA-seq	SRR11458718	13,9174
Escherichia coli	SRR14005143	0,489701
Human gut microbiota	SRR341725	3,8514
Homo sapiens; RNA-seq	SRR957915	8,7066
Human microbiome	SRR061958	1,956381
Musa balbisiana; RNA-seq	SRR10260779	7,872313
Broiler chicken	SRR13605073	0,97234
Drosophila ananassae; RNA-seq	SRR332538	2,443997
Danio rerio; RNA-seq	SRR5853087	7,786706
Totale		61,04

Tabella 5.1: Datasets usati nello studio. L'ultima riga rappresenta il totale di GB occupati dall'insieme dei datasets.

5.1.3 Parametri

La compressione dei datasets è avvenuta secondo due modalità principali: compressione di ciascun dataset singolarmente e compressione simultanea di tali datasets. Sono stati utilizzati i tools GGCAT e Gzip e messi a confronto in termini di compression ratio, tempo e memoria impiegati. Sottolineiamo che il tempo riportato è il *CPU-time*, dunque solo il tempo in cui il processore elabora le istruzioni.

Lo scopo principale di questo studio è stabilire quale sia il metodo più efficiente, tra compressione singola e simultanea, a seconda del software utilizzato e confrontando i risultati secondo le misure già citate.

Per quanto riguarda GGCAT abbiamo studiato la compressione sia con il calcolo degli unitigs massimali sia con il calcolo dei greedy matchtigs, ovvero il calcolo greedy della minima rappresentazione plain text. Le analisi sono state fatte al variare del valore di k e con un numero di threads impostato di default a 16. Trattandosi di un sistema multithread il tempo riportato sarà il tempo complessivo per elaborare il processo, ovvero la somma dei tempi richiesti dalle varie CPUs.

I file output prodotti da GGCAT sono poi stati compressi ulteriormente con Gzip per studiarne il margine di miglioramento. Infine per ciascun file output prodotto da GGCAT sono stati cal-

colati il numero di run di colori (ovvero di ciascuna collezione di dataset) e la dimensione della sola sequenza senza header.

5.2 Dati ottenuti

5.2.1 Compressione singola di insiemi di k-mers

I dati presentati in questa sottosezione indicano i parametri riscontrati comprimendo ciascun dataset singolarmente con i software Gzip e GGCAT.

Risultati di Gzip

- **CR e spazio risparmiato**

Dataset	File compresso in Bytes	CR	Spazio risparmiato
SRR001665	452141560	4,61	78,32%
SRR062379	1732080946	6,32	84,19%
SRR11458718	3477262499	4,00	75,02%
SRR14005143	146466431	3,34	70,09%
SRR341725	1039206703	3,71	73,02%
SRR957915	2179004885	4,00	74,97%
SRR061958	449221077	4,36	77,04%
SRR10260779	1979768827	3,98	74,85%
SRR13605073	226316493	4,3	72,73%
SRR332538	451516054	5,41	85,53%
SRR5853087	2111006192	3,69	72,89%

Tabella 5.2: I dati ottenuti dalla compressione singola dei datasets con Gzip. La prima colonna riporta la dimensione in bytes del file in output, la seconda colonna riporta il calcolo della compression ratio e la terza colonna la percentuale di spazio salvati rispetto al dataset non compresso.

- **Memoria RAM utilizzata e tempo impiegato**

Dataset	GB di RAM impiegata	Tempo richiesto
SRR001665	3	0h:01m:44s
SRR062379	3,44	0h:13m:09s
SRR11458718	3	0h:31m:49s
SRR14005143	3	0h:01m:28s
SRR341725	3,22	0h:08m:45s
SRR957915	3,22	0h:18m:28s
SRR061958	3	0h:03m:54s
SRR10260779	3	0h:16m:41s
SRR13605073	3,22	0h:01m:57s
SRR332538	3	0h:02m:53s
SRR5853087	3,48	0h:20m:45s

Tabella 5.3: Memoria RAM richiesta da Gzip per i processi e il tempo impiegato per portarli a termine.

Osservazioni

Notiamo che già una compressione con Gzip porta tutto sommato dei buoni risultati: lo spazio salvato parte da un minimo di $70{,}09\%$ fino ad arrivare ad un massimo di $85{,}53\%$. La memoria richiesta non supera mai i $3{,}5GB$ e il tempo rimane ragionevolmente basso.

Risultati di GGCAT

UNITIGS MASSIMALI

- **CR e spazio risparmiato**

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
SRR001665	12	238713289	8,74	88,54%
	22	216947770	9,61	89,6%
	32	212416092	9,82	89,82%
SRR062379	12	300835326	36,41	96,25%
	22	1142130136	9,59	89,57%
	32	1184426002	9,25	89,19%
SRR11458718	12	299782401	46,43	97,85%
	22	849099984	16,39	93,9%
	32	910678483	15,28	93,46%
SRR14005143	12	247681603	1,98	49,42%
	22	146261286	3,35	70,13%
	32	146845183	3,33	70,01%
SRR341725	12	299794186	12,85	92,22%
	22	735847113	5,23	80,89%
	32	726321728	5,30	81,14%
SRR957915	12	300843714	28,94	96,54%
	22	2506544919	3,47	71,21%
	32	2576835356	3,38	70,40%

Tabella 5.4: Dati ottenuti comprimendo i primi 6 datasets con GGCAT. La seconda colonna indica i valori di *k*, la terza colonna indica la dimensione dei file in output, la quarta colonna indica il tasso di compressione e infine la quinta colonna indica lo spazio risparmiato.

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
SRR061958	12	300928602	6,5	84,61%
	22	828698212	2,36	57,64%
	32	800441490	2,44	59,09%
SRR10260779	12	299212016	26,31	96,2%
	22	869156401	9,06	88,96%
	32	960618177	8,2	87,8%
SRR13605073	12	257967652	3,77	73,47%
	22	202713921	4,8	79,15%
	32	221302269	4,39	77,24%
SRR332538	12	242036437	10,1	90,1%
	22	188063555	13	92,31%
	32	187666271	13,02	92,32%
SRR5853087	12	300437395	25,92	96,14%
	22	1887589867	4,13	75,76%
	32	1771783431	4,39	77,25%

Tabella 5.5: Dati ottenuti comprimendo gli ultimi 5 datasets con GGCAT.La seconda colonna indica i valori di *k*, la terza colonna indica la dimensione dei file in output, la quarta colonna indica il tasso di compressione e infine la quinta colonna indica lo spazio risparmiato.

- **Memoria RAM utilizzata e tempo impiegato**

Dataset	K	Memoria RAM	Tempo
SRR001665	12	1,12	0h:02m:23s
	22	1,03	0h:01m:34s
	32	0,94	0h:0m:43s
SRR062379	12	1,27	0h:10m:43s
	22	1,67	0h:11m:31s
	32	1,53	0h:07m:40s
SRR11458718	12	1,64	0h:32m:30s
	22	2,13	0h:25m:32s
	32	1,93	0h:21m:51s
SRR14005143	12	1,09	0h:01m:40s
	22	1,09	0h:01m:33s
	32	0,99	0h:01m:17s
SRR341725	12	1,24	0h:07m:49s
	22	1,62	0h:08m:52s
	32	1,44	0h:05m:49s
SRR957915	12	1,31	0h:15m:40s
	22	2,08	0h:19m:15s
	32	2,1	0h:14:08s

Tabella 5.6: Dati ottenuti comprimendo i primi 6 datasets con GGCAT. La seconda colonna indica i valori di *k*, la terza colonna indica la memoria RAM in GB impiegata da GGCAT, la quarta colonna indica invece il tempo richiesto dal software.

Dataset	K	Memoria RAM	Tempo
SRR061958	12	1,2	0h:04m:15s
	22	1,48	0h:04m:56s
	32	1,38	0h:03m:36s
SRR10260779	12	1,32	0h:18m:49s
	22	1,81	0h:12m:30s
	32	1,71	0h:09m:04s
SRR13605073	12	1,14	0h:02m:56s
	22	1,21	0h:02m:38s
	32	1,14	0h:01:38s
SRR332538	12	1,11	0h:05m:02s
	22	1,19	0h:03m:34s
	32	1,16	0h:02m:36s
SRR5853087	12	1,4	0h:21m:21s
	22	2,05	0h:19m:36s
	32	1,96	0h:13m:31s

Tabella 5.7: Dati ottenuti comprimendo gli ultimi 5 datasets con GGCAT.La seconda colonna indica i valori di *k*, la terza colonna indica la memoria RAM in GB impiegata da GGCAT, la quarta colonna indica invece il tempo richiesto dal software.

- **Numero di run di colori e dimensione della sequenza**

Dataset	K	Run di colori	Sequenza
SRR001665	12	6661768	79992781
	22	4434305	114158625
	32	3748553	125854375
SRR062379	12	8387401	100648812
	22	20504262	652390650
	32	17558107	769907811
SRR11458718	12	8358153	100297857
	22	14773583	498907920
	32	12965215	607009936
SRR14005143	12	6909029	82979221
	22	2446848	87751573
	32	1924568	100892759
SRR341725	12	8358476	100301962
	22	11678623	477836588
	32	9938188	515138714
SRR957915	12	8387634	100651608
	22	45507031	1393215833
	32	37960754	1653156177

Tabella 5.8: Dati ottenuti analizzando i primi sei files *.fasta* prodotti da GGCAT. La prima colonna indica il numero di run di colori mentre la seconda colonna indica la dimensione della sequenza.

Dataset	K	Run di colori	Sequenza
SRR061958	12	8389992	100679904
	22	13809159	500064528
	32	10972245	544300397
SRR10260779	12	8342309	100107782
	22	16353689	488746416
	32	15103860	614997782
SRR13605073	12	7194926	86400826
	22	3281133	123299509
	32	3012222	148689447
SRR332538	12	6753047	81090817
	22	3609034	104537222
	32	2927048	120638500
SRR5853087	12	8173742	98087258
	22	9063532	312819265
	32	6436464	331331779

Tabella 5.9: Dati ottenuti analizzando gli ultimi cinque files *.fasta* prodotti da GGCAT. La prima colonna indica il numero di run di colori mentre la seconda colonna indica la dimensione della sequenza.

Osservazioni

Rispetto a Gzip, GGCAT si mostra notevolemente più efficiente. In termini di memoria e tempo richiesto arriviamo rispettivamente a dei massimi di $2{,}13GB$ e di circa 25 minuti. Come CR arriviamo a sfiorare 47 di tasso. Dunque la scelta di GGCAT con calcolo di unitigs massimali si rivela già una buona scelta.

GREEDY MATCHTIGS

- **CR e spazio risparmiato**

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
SRR001665	12	19750074	105,6	99,05%
	22	110609057	18,86	94,7%
	32	1423492622	14,65	93,18%
SRR062379	12	8847312	1238,03	99,92%
	22	489511234	22,38	95,53%
	32	560577206	19,54	94,88%
SRR11458718	12	10146113	1371,7	99,93%
	22	338274223	41,14	97,57%
	32	402282757	34,6	97,11%
SRR14005143	12	20254653	24,18	95,86%
	22	55575183	8,81	88,65%
	32	62292109	7,86	87,28%
SRR341725	12	9153499	420,76	99,76%
	22	443189836	8,69	88,49%
	32	481983273	7,99	87,49%
SRR957915	12	709278	12275,3	99,99%
	22	961754233	9,05	88,95%
	32	1137634687	7,65	86,93%

Tabella 5.10: Dati ottenuti comprimendo i primi 6 datasets con GGCAT con l'opzione greedy matchtigs. La seconda colonna indica i valori di *k*, la terza colonna indica la dimensione dei file in output, la quarta colonna indica il tasso di compressione e infine la quinta colonna indica lo spazio risparmiato.

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
SRR061958	12	8748213	223,63	99,55%
	22	410418900	4,77	79,02%
	32	448667016	4,36	77,07%
SRR10260779	12	10009484	786,49	99,87%
	22	365434704	21,54	95,36%
	32	429556337	19,33	94,54%
SRR13605073	12	6984043	139,22	99,28%
	22	110882792	8,77	88,6%
	32	130549553	7,45	86,57%
SRR332538	12	20229425	120,81	99,17%
	22	69183423	35,33	97,17%
	32	90930011	26,88	96,28%
SRR5853087	12	8370151	930,29	99,89%
	22	213509892	36,47	97,26%
	32	255251496	30,51	96,72%

Tabella 5.11: Dati ottenuti comprimendo gli ultimi 5 datasets con GGCAT con l'opzione greedy matchings.La seconda colonna indica i valori di *k*, la terza colonna indica la dimensione dei file in output, la quarta colonna indica il tasso di compressione e infine la quinta colonna indica lo spazio risparmiato.

- **Memoria RAM utilizzata e tempo impiegato**

Dataset	K	Memoria RAM	Tempo
SRR001665	12	5,79	01h:14m:02s
	22	3,18	0h:01m:52s
	32	0,96	0h:01m:14s
SRR062379	12	5,42	0h:17m:53s
	22	10,63	0h:46m:27s
	32	9,12	0h:10m:35s
SRR11458718	12	7,54	01h:04m:04s
	22	8,08	0h:33m:15s
	32	6,84	0h:23m:31s
SRR14005143	12	5,93	01h:28m:12s
	22	1,07	0h:02m:04s
	32	0,996	0h:01m:41s
SRR341725	12	7,55	01h:34m:41s
	22	6,26	0h:11m:43s
	32	3,85	0h:07m:00s
SRR957915	12	6,78	0h:33m:43s
	22	22,16	02h:03m:00s
	32	19,46	0h:25m:33s

Tabella 5.12: Dati ottenuti comprimendo i primi 6 datasets con GGCAT opzione greedy matchtigs. La seconda colonna indica i valori di *k*, la terza colonna indica la memoria RAM in GB impiegata da GGCAT, la quarta colonna indica invece il tempo richiesto dal software.

Dataset	K	Memoria RAM	Tempo
SRR061958	12	5,63	0h:06m:53s
	22	7,48	0h:17m:03s
	32	6,6	0h:06m:49s
SRR10260779	12	7,64	0h:56m:10s
	22	8,55	0h:19m:31s
	32	7,89	0h:11m:52s
SRR13605073	12	7,04	03h:02m:19s
	22	2,49	0h:04m:54s
	32	2,51	0h:06m:07s
SRR332538	12	5,89	0h:42m:42s
	22	2,63	0h:10m:19s
	32	2,42	0h:03m:51s
SRR5853087	12	7,63	0h:51m:13s
	22	6,13	06h:39m:15s
	32	4,28	04h:12m:45s

Tabella 5.13: Dati ottenuti comprimendo gli ultimi 5 datasets con GGCAT con l'opzione greedy matchtigs.La seconda colonna indica i valori di *k*, la terza colonna indica la memoria RAM in GB impiegata da GGCAT, la quarta colonna indica invece il tempo richiesto dal software.

- **Numero di run di colori e dimensione della sequenza**

Dataset	K	Run di colori	Sequenza
SRR001665	12	2192572	6390955
	22	2888554	60836978
	32	2959193	84864573
SRR062379	12	68710	8382680
	22	11265146	328334051
	32	10346789	399327373
SRR11458718	12	280953	8332158
	22	7062753	243945439
	32	6649448	304457568
SRR14005143	12	2246649	6618399
	22	1083927	41500438
	32	877293	49679304
SRR341725	12	123549	8338435
	22	7505519	330361255
	32	7190066	368562741
SRR957915	12	211944	8383238
	22	24270960	629376275
	32	21879992	803257159

Tabella 5.14: Dati ottenuti analizzando i primi sei files *.fasta* prodotti da GGCAT con opzione greedy matchtigs. La prima colonna indica il numero di run di colori mentre la seconda colonna indica la dimensione della sequenza.

Dataset	K	Run di colori	Sequenza
SRR061958	12	52960	8387346
	22	7873895	289864809
	32	6864196	332815049
SRR10260779	12	263057	8307084
	22	9188835	237297638
	32	8913446	299012931
SRR13605073	12	979	6953397
	22	1288442	79796591
	32	1388091	97135763
SRR332538	12	2287298	6382582
	22	1315161	46950672
	32	1429380	64540323
SRR5853087	12	1701	8070585
	22	2165469	160142247
	32	2343445	197029467

Tabella 5.15: Dati ottenuti analizzando gli ultimi cinque files *.fasta* prodotti da GGCAT con opzione greedy matchtigs. La prima colonna indica il numero di run di colori mentre la seconda colonna indica la dimensione della sequenza.

Osservazioni

Per GGCAT con opzione greedy matchtigs la compressione, soprattutto con un valore di $k = 12$ risulta ben più elevate anche rispetto all'esecuzione di GGCAT di default. I risultati di CR e spazio salvato rimangono comunque migliori di Gzip e generalmente di GGCAT con calcolo di unitigs massimali anche per i restanti valori di k. Aumenta però in linea di massima la memoria richiesta, fino ad arrivare a ben $22GB$, e anche il tempo di esecuzione con un massimo di ben oltre 6 ore.

Per quanto rigurada il confronto sulla dimensione delle sequenze in output tra GGCAT e GGCAT con opzione greedy matchtigs, generalmente la dimensione delle sequenze prodotte dal calcolo con greedy matchtigs è inferiore così come il numero di run dei colori.

Risultati di una prima compressione con GGCAT e una seconda compressione con Gzip

Per capire il margine di miglioramento di GGCAT, i file *.fasta* ottenuti sono stati nuovamente compressi con Gzip: in seguito i risultati.

UNITIGS MASSIMALI

- **CR e spazio risparmiato**

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
SRR001665	12	43195466	48,28	97,93%
	22	53614792	38,9	97,43%
	32	52046807	40,07	97,5%
SRR062379	12	50499695	216,9	99,54%
	22	282315736	38,8	97,42%
	32	295919012	37,01	97,3%
SRR11458718	12	48605026	286,34	99,65%
	22	212772951	65,41	98,47%
	32	230372350	60,41	98,34%
SRR14005143	12	44722672	10,95	90,87%
	22	37883828	12,93	92,26%
	32	38540620	12,71	92,13%
SRR341725	12	50933118	75,62	98,68%
	22	202086846	19,06	94,75%
	32	204418053	18,84	94,69%
SRR957915	12	49716520	175,12	99,43%
	22	587311153	14,82	93,25%
	32	608201826	14,32	93,01%

Tabella 5.16: Dati ottenuti comprimendo i primi 6 datasets con GGCAT e successivamente con Gzip. La seconda colonna indica i valori di *k*, la terza colonna indica la dimensione dei file in output dalla seconda compressione, la quarta colonna indica il tasso di compressione e infine la quinta colonna indica lo spazio risparmiato.

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
SRR061958	12	52624581	37,18	97,31%
	22	211878207	9,23	89,17%
	32	209638116	9,33	89,28%
SRR10260779	12	49669164	158,49	99,37%
	22	212264273	37,09	97,3%
	32	236798763	33,24	96,99%
SRR13605073	12	45353928	21,44	95,34%
	22	49591015	19,61	94,9%
	32	53528707	18,16	94,49%
SRR332538	12	43404827	56,31	98,22%
	22	41849074	58,4	98,29%
	32	40981083	59,64	98,32%
SRR5853087	12	50950851	152,83	99,35%
	22	124170622	62,71	98,41%
	32	116367603	66,91	98,51%

Tabella 5.17: Dati ottenuti comprimendo gli ultimi 5 datasets con GGCAT e successivamente con Gzip.La seconda colonna indica i valori di *k*, la terza colonna indica la dimensione dei file in output da Gzip, la quarta colonna indica il tasso di compressione e infine la quinta colonna indica lo spazio risparmiato.

- **Memoria RAM utilizzata e tempo impiegato**

Dataset	K	Memoria RAM	Tempo
SRR061958	12	3,48	0h:0m:18s
	22	3,44	0h:22m:0s
	32	4,5	0h:02m:11s
SRR10260779	12	3,22	0h:0m:14s
	22	4,5	0h:01m:57s
	32	3,44	0h:02m:12s
SRR13605073	12	2,85	0h:0m:14s
	22	3,22	0h:0m:26s
	32	3,44	0h:0m:31s
SRR332538	12	2,47	0h:0m:14s
	22	3,48	0h:0m:21s
	32	3,41	0h:0m:19s
SRR5853087	12	3,04	0h:0m:18s
	22	4,5	0h:01m:10s
	32	4,6	0h:01m:11s

Tabella 5.19: Dati ottenuti comprimendo gli ultimi 5 datasets con GGCAT e successivamente con Gzip. La seconda colonna indica i valori di *k*, la terza colonna indica la memoria RAM in GB impiegata da Gzip nella seconda compressione, la quarta colonna indica invece il tempo richiesto dal software.

Dataset	K	Memoria RAM	Tempo
SRR001665	12	3,22	0h:0m:14s
	22	4,6	0h:0m:30s
	32	2,85	0h:0m:25s
SRR062379	12	3,22	0h:0m:15s
	22	3,94	0h:02m:12s
	32	3,22	0h:02m:38s
SRR11458718	12	3,48	0h:0m:15s
	22	3,94	0h:01m:38s
	32	3,22	0h:02m:05s
SRR14005143	12	3,48	0h:0m:17s
	22	3,56	0h:0m:23s
	32	3,56	0h:0m:24s
SRR341725	12	2,47	0h:0m:15s
	22	3,48	0h:02m:01s
	32	3,48	0h:02m:08s
SRR957915	12	3,03	0h:0m:15s
	22	3,22	0h:04m:40s
	32	3,48	0h:05m:30s

Tabella 5.18: Dati ottenuti comprimendo i primi 6 datasets con GGCAT e successivamente con Gzip. La seconda colonna indica i valori di *k*, la terza colonna indica la memoria RAM in GB impiegata da Gzip nella seconda compressione, la quarta colonna indica invece il tempo richiesto dal software.

GREEDY MATCHTIGS

- **CR e spazio risparmiato**

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
SRR001665	12	4046037	2707,14	99,96%
	22	31343493	349,46	99,71%
	32	38343627	285,66	99,65%
SRR062379	12	2540821	5477,52	99,98%
	22	148821412	93,52	98,93%
	32	169755696	81,98	98,78%
SRR11458718	12	2727831	179,52	99,44%
	22	102705933	4,77	79,03%
	32	122333062	4,0	75,01%
SRR14005143	12	4048157	951,4	99,89%
	22	17023339	226,24	99,56%
	32	19139516	201,23	99,50%
SRR341725	12	2603793	3343,81	99,97%
	22	134388237	64,79	998,46%
	32	145382842	59,89	98,33%
SRR957915	12	2695831	725,71	99,86%
	22	288263968	6,79	85,27%
	32	340058931	5,75	82,62%

Tabella 5.20: Dati ottenuti comprimendo i primi 6 datasets con GGCAT con l'opzione greedy matchtigs e successivamente con Gzip.La seconda colonna indica i valori di *k*, la terza colonna indica la dimensione dei file in output di GZip, la quarta colonna indica il tasso di compressione e infine la quinta colonna indica lo spazio risparmiato.

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
SRR061958	12	2506614	3140,62	99,97%
	22	122888773	64,06	98,44%
	32	132702724	59,32	98,31%
SRR10260779	12	2700010	360,12	99,72%
	22	109198207	8,90	88,77%
	32	128786416	7,55	86,76%
SRR13605073	12	2050907	1191,67	99,92%
	22	30888351	79,12	98,74%
	32	35329618	69,18	98,55%
SRR332538	12	4009485	1942,07	99,95%
	22	20282822	383,91	99,74%
	32	25698980	302,99	99,67%
SRR5853087	12	2378715	25659,08	99,996%
	22	61695816	989,3	97,26%
	32	71735396	850,84	99,88%

Tabella 5.21: Dati ottenuti comprimendo gli ultimi 5 datasets con GGCAT con l'opzione greedy matchings e successivamente con Gzip.La seconda colonna indica i valori di *k*, la terza colonna indica la dimensione dei file in output di GZip, la quarta colonna indica il tasso di compressione e infine la quinta colonna indica lo spazio risparmiato.

- **Memoria RAM utilizzata e tempo impiegato**

Dataset	K	Memoria RAM	Tempo
SRR001665	12	3,56	0h:0m:02s
	22	3,38	0h:0m:16s
	32	3,36	0h:0m:23s
SRR062379	12	2,47	0h:0m:02s
	22	3	0h:01m:25s
	32	3,22	0h:01m:40s
SRR11458718	12	3,48	0h:0m:02s
	22	3,48	0h:01m:06s
	32	3,48	0h:01m:22s
SRR14005143	12	2,98	0h:0m:02s
	22	3,22	0h:0m:11s
	32	3,04	0h:0m:13s
SRR341725	12	3,48	0h:0m:02s
	22	3,48	0h:01m:28s
	32	3,22	0h:01m:30s
SRR957915	12	2,47	0h:0m:02s
	22	3,06	0h:02m:56s
	32	3,13	0h:03m:27s

Tabella 5.22: Dati ottenuti comprimendo i primi 6 datasets con GGCAT con opzione greedy matchtigs e successivamente con Gzip. La seconda colonna indica i valori di *k*, la terza colonna indica la memoria RAM in GB impiegata da Gzip nella seconda compressione, la quarta colonna indica invece il tempo richiesto dal software.

Dataset	K	Memoria RAM	Tempo
SRR061958	12	2,47	0h:0m:02s
	22	4,5	0h:01m:21s
	32	3,46	0h:01m:26s
SRR10260779	12	3,32	0h:0m:02s
	22	3	0h:01m:02s
	32	4,5	0h:01m:23s
SRR13605073	12	3,44	0h:0m:02s
	22	3,41	0h:0m:17s
	32	3,44	0h:0m:23s
SRR332538	12	2,47	0h:0m:02s
	22	3,48	0h:0m:12s
	32	3,41	0h:0m:14s
SRR5853087	12	3,46	0h:0m:02s
	22	4,5	0h:0m:42s
	32	4,6	0h:0m:48s

Tabella 5.23: Dati ottenuti comprimendo gli ultimi 5 datasets con GGCAT con l'opzione greedy matchtigs e successivamente con Gzip. La seconda colonna indica i valori di *k*, la terza colonna indica la memoria RAM in GB impiegata da Gzip nella seconda compressione, la quarta colonna indica invece il tempo richiesto dal software.

Osservazioni

Sia per quanto riguarda GGCAT con opzione greedy matchtigs sia per GGCAT con calcolo degli unitigs massimali, la seconda compressione con Gzip ha portato a risultati migliori in termini di compressione. Per quanto riguarda tempi e memoria, Gzip si è mostrato in linea con i risultati ottenuti dalla prima compressione con lo stesso Gzip.

5.2.2 Compressione simultanea di insiemi di k-mers

In questa sottosezione vengono riportati i dati ottenuti comprimendo insieme i vari datasets.

Risultati di Gzip

- **CR e spazio risparmiato**

Dataset	File compresso in Bytes	CR	Spazio risparmiato
Lista di datasets	12589408700	4,85	79,37%

Tabella 5.24: Dati ottenuti comprimendo la lista di datasets con Gzip. La seconda colonna riporta la dimensione del file in output, la terza colonna riporta la compression ratio e la quarta colonna riporta la percentuale di spazio salvato.

- **Memoria RAM utilizzata e tempo impiegato**

Dataset	Memoria RAM	Tempo
Lista di datasets	4,5	02h:01m:50s

Tabella 5.25: Dati ottenuti comprimendo la lista di datasets con Gzip. La seconda colonna riporta la memoria RAM richiesta dal processo mentre la terza colonna indica il tempo richiesto per la compressione.

Osservazioni

Rispetto alla compressione singola, Gzip si dimostra essere più efficiente nella compressione simultanea: il tasso di compressione rimane in linea con i valori precedenti, ma il tempo e la memoria sono notevolmente inferiori rispetto alla somma di tempi e GB richiesti dalle compressioni singole.

Risultati di GGCAT

UNITIGS MASSIMALI

- **CR e spazio risparmiato**

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
Lista di datasets	12	307423481	198,54	99,5%
	22	8676500132	7,03	85,78%
	32	8536646615	7,15	86,01%

Tabella 5.26: Dati ottenuti comprimendo la lista di datasets con GGCAT. La terza colonna riporta la dimensione del file in output, la quarta colonna riporta la compression ratio e la quinta colonna riporta la percentuale di spazio salvato.

- **Memoria RAM utilizzata e tempo impiegato**

Dataset	K	Memoria RAM	Tempo
Lista di datasets	12	2,29	01h:56m:30s
	22	3,64	01h:55m:01s
	32	3,34	01h:24m:23s

Tabella 5.27: Dati ottenuti comprimendo la lista di datasets con GGCAT. La terza colonna riporta la memoria RAM richiesta e la quarta colonna riporta il tempo impiegato dal processo.

- **Numero di run di colori e dimensione della sequenza**

Dataset	K	Run di colori	Sequenza
Lista di datasets	12	6915460	8388622
	22	86025360	2400475952
	32	73207626	2971941356

Tabella 5.28: Dati ottenuti analizzando i file *.fasta* prodotti da GGCAT. La prima colonna indica il numero di run di colori mentre la seconda colonna indica la dimensione della sequenza.

Osservazioni

Anche questa volta GGCAT si mostra più efficiente nella compressione, in questo caso simultanea, rispetto a Gzip: il tasso di compressione è sempre maggiore, arrivando fino a quasi 200, mantenendo un tempo e una memoria dedicati all'esecuzione sempre minori di Gzip.

GREEDY MATCHTIGS

- **CR e spazio risparmiato**

Dataset	K	File compresso in Bytes	CR	Spazio risparmiato
Lista di datasets	12	55180267	1106,11	99,91%
	22	3634320606	16,79	94,04%
	32	4174837668	14,62	93,16%

Tabella 5.29: Dati ottenuti comprimendo la lista di datasets con GGCAT con l'opzione greedy matchtigs. La terza colonna riporta la dimensione del file in output, la quarta colonna riporta la compression ratio e la quinta colonna riporta la percentuale di spazio salvato.

- **Memoria RAM utilizzata e tempo impiegato**

Dataset	K	Memoria RAM	Tempo
Lista di datasets	12	58,76	1-0h:05m:09s
	22	70,86	1-17h:17m:20s
	32	60,39	15h:36m:14s

Tabella 5.30: Dati ottenuti comprimendo la lista di datasets con GGCAT con l'opzione greedy matchtigs. La terza colonna riporta la memoria RAM richiesta e la quarta colonna riporta il tempo impiegato dal processo.

- **Numero di run di colori e dimensione della sequenza**

Dataset	K	Run di colori	Sequenza
Lista di datasets	12	6914816	8388622
	22	86008778	2400452815
	32	73249742	2972034623

Tabella 5.31: Dati ottenuti analizzando i file *.fasta* prodotti da GGCAT con l'opzione greedy matchtigs. La prima colonna indica il numero di run di colori mentre la seconda colonna indica la dimensione della sequenza.

Osservazioni

GGCAT con opzione greedy matchtigs conferma quanto abbiamo detto in precedenza: il tasso di compressione e lo spazio risparmiato sono i più alti, ma il tempo e la memoria richiesti sono altrettanto elevat, arrivando a sforare i $70GB$ di memoria e impiegando oltre un giorno di calcolo.

5.3 Confronto e guadagni ottenuti

A seconda del software e delle opzioni utilizzate abbiamo visto come i valori dei parametri analizzati varino molto. In questa sezione compareremo i vari risultati.

5.3.1 Memoria

Riportiamo la memoria totale richiesta dai vari processi.

Memoria totale della prima compressione con GGCAT

Compressioni	k	memory-utilized	memory-utilized (-g)[1]
Compressioni singole	12	13,84	72,84
	22	17,36	78,66
	32	16,28	64,92
Compressione simultanea	12	2,29	58,76
	22	3,64	70,86
	32	3,34	60,39

Tabella 5.32

Memoria totale della prima compressione con Gzip

Compressioni	Memory-utilized
Compressioni singole	34,58
Compressione simultanea	4,5

Tabella 5.33

Memoria totale della prima e seconda compressione

Compressioni	k	memory-utilized	memory-utilized (-g)
Compressioni singole	12	13,87	72,87
	22	17,4	78,7
	32	16,32	64,96
Compressione simultanea	12	2,32	58,79
	22	3,68	70,9
	32	3,38	60,43

Tabella 5.34

Osservazioni

Possiamo vedere più facilmente in queste tabelle come l'opzione *-g* sia la più dispendiosa in termini di memoria. Possiamo anche notare però come la richiesta di memoria dimuisca sempre passando da compressioni singole a compressioni simultanee. Considerando GGCAT, l'esecuzione con $k = 12$ richiede meno risorse, ad eccezione delle compressioni singole con opzione *-g*, dove si tiene nel mezzo.

5.3.2 Tempo

Tempo totale della prima compressione con GGCAT

Compressioni	k	CPU-utilized time	CPU-utilized time (-g)
Compressioni singole	12	02h:03m:08s	14h:54m:11s
	22	01h:51m:31s	11h:09m:23s
	32	01h:22m:07s	05h:50m:58s
Compressione simultanea	12	01h:56m:30s	1-00h:05m:09s
	22	02h:06m:44s	1-17h:27m:17s
	32	01h:36m:11s	15h:49m:26s

Tabella 5.35

Tempo totale della prima compressione con Gzip

Riportiamo il tempo totale impiegato dai vari processi.

Compressioni	CPU-utilized time
Compressioni singole	02h:01m:33s
Compressione simultanea	02h:01m:50s

Tabella 5.36

Tempo totale della prima e seconda compressione

Compressioni	k	CPU-utilized time	CPU-utilized time (-g)
Compressioni singole	12	02h:05m:57s	11h:52m:14s
	22	02h:08m:49s	11h:20m:19s
	32	01h:41m:41s	06h:03m:47s
Compressione simultanea	12	01h:56m:36s	1-00h:05m:14s
	22	02h:06m:44s	1-17h:27m:17s
	32	01h:36m:11s	15h:49m:26s

Tabella 5.37

Osservazioni

Il tempo richiesto dalle compressioni singole è paragonabile al tempo richiesto dalle compressioni simultanee, fatta eccezione per GGCAT *-g* che sfora le 24 ore in due casi su tre. Generalmente la compressione con $k = 32$ richiede meno tempo.

5.3.3 Efficacia della compressione

Per analizzare questi dati sono state paragonate le somme delle dimensioni dei files prodotti dalle compressioni singole e simultanee rispetto alla somma delle dimensioni dei datasets non compressi.

- **Guadagno della compressione singola rispetto a dati non compressi**

GGCAT	12	+94,94%
	22	+84,32%
	32	+84,11%
GGCAT (-g)	12	99,8%
	22	+94,15%
	32	+93,21%
Gzip	76,66%	

Tabella 5.38

- **Guadagno della compressione simultanea rispetto a dati non compressi**

GGCAT	12	+99,5%
	22	+85,78%
	32	+86,01%
GGCAT (-g)	12	99,91%
	22	+94,05%
	32	+93,16%
Gzip	+79,37%	

Tabella 5.39

- **Guadagno della compressione simultanea rispetto alla compressione singola**

GGCAT	12	+99,1%
	22	+9,37%
	32	+11,99%
GGCAT (-g)	12	55,21%
	22	-1,85%
	32	-0,79%
Gzip	+79,37%	

Tabella 5.40

Osservazioni

Da questi dati è evidente la superiorità di GGCAT per quanto concerne la compressione rispetto a Gzip. Inoltre la compressione simultanea inoltre si dimostra più efficiente, tranne considerando l'opzione -*g* di GGCAT con valore $k = 22$ e $k = 32$. In questi casi la compressione simultanea si rivela leggermente meno conveniente rispetto alla compressione singola: questo è dovuto alla capacità di compressione già molto elevata dell'opzione greedy matchtigs. Il peggioramento, essendo molto lieve, risulta trascurabile.

5.3.4 Efficacia della seconda compressione rispetto alla prima compressione

Per comprendere i margini di miglioramento della compressione con GGCAT riportiamo i guadagni ottenuti comprimendo una seconda volta con Gzip i file *.fasta* ottenuti dalla prima compressione con GGCAT.

- **Guadagno della seconda compressione rispetto alla prima compressione singola**

GGCAT	12	+82,85%
	22	+78,94%
	32	+78,49%
GGCAT (-g)	12	+73,77%
	22	+70,08%
	32	+70,32%

Tabella 5.41

- **Guadagno della seconda compressione rispetto alla prima compressione simultanea**

GGCAT	12	+95,98%
	22	+97,18%
	32	+99,12%
GGCAT (-g)	12	+77,78%
	22	+68,39%
	32	+68,81%

Tabella 5.42

- **Guadagno della seconda compressione seguita alla compressione simultanea rispetto alla seconda compressione seguita alla compressione singola.**

GGCAT	12	+97,67%
	22	+87,84%
	32	+96,42%
GGCAT (-g)	12	+62,05%
	22	-7,61%
	32	-5,93%

Tabella 5.43

Osservazioni

Le precedenti tabelle mostrano i guadagni ottenuti dalla seconda compressione con Gzip. Rispetto alla prima compressione, vediamo dei margini di miglioramento di GGCAT, ottenendo un guadagno fino al 99,12% considerando il confronto con la prima compressione simultanea. Anche in questo caso confrontando le seconde compressioni seguite rispettivamente alle compressioni singole e alle compressioni simultanee, notiamo che abbiamo un peggioramento, questa volta leggermente più alto ma comunque ragionevolmente trascurabile, ottenuto dalla compressione simultanea con GGCAT *-g*.

Capitolo 6

Conclusioni

In questa tesi è stato studiato il tema dell'archiviazione di dati, in particolare di dati di origine genetica. Sono stati introdotti i concetti di k-mers, del loro utilizzo e della loro rappresentazioni. Concetti essenziali per il principale software di questo studio, GGCAT: un programma di costruzione di grafi di de Bruijn, di cui abbiamo analizzato la capacità di compressione. Abbiamo confrontato secondo i parametri di compression ratio, spazio risparmiato, tempo e memoria richiesti dai processi di compressione di GGCAT, dunque una compressione specifica, e dai processi di compressione di Gzip, il famoso tool di compressione general purpose.
Gli studi si sono divisi, concentrandosi da una parte sulla compressione di singoli k-mers sets e da un'altra su insiemi di k-mers sets. In una prima compressione, come riportato dalla tabella 5.40, possiamo in generale dire che per entrambi i software la compressione simultanea risulta più efficiente rispetto alla compressione singola. Lo stesso dicasi in seguito alla seconda compressione dei file *.fasta* con Gzip come si evince dalla tabella 5.42. La compressione simultanea si è mostrata più conveniente anche sia in termini di memoria che di tempo richiesti dai softwares.
Per quanto riguarda il confronto tra i programmi GGCAT e Gzip, in termini di memoria GGCAT si è mostrato sempre più efficiente di Gzip mentre in termini di tempo si mantengono entrambi sullo stesso ordine. Fa eccezione però l'esecuzione di GGCAT con opzione greedy matchtigs che, sebbene in termini di tasso di compressione e spazio salvato si mostri quasi sempre come la scelta migliore (5.38, 5.39, 5.40), richiede spesso più tempo e più memoria rispetto sia a GGCAT senza opzione *-g* sia rispetto a Gzip.
GGCAT è un software molto potente ed essendo un programma di costruzione di grafi disk-based, come sottolineato per altro dagli autori[19], le sue prestazioni vengono influenzate molto dall'hardware sottostante: è preferibile dunque utilizzare una RAM veloce e un disco NVME. I lavori futuri su GGCAT prevedono già il supporto di aggiornamenti online, ovvero aggiornare i grafi di de Bruijn compatti e colorati con vari set di reads o genomi mantenedo la possibilità

di effettuare queries al grafo tra gli aggiornamenti. Inoltre, dai risultati sperimentali di questo studio (5.3.4) notiamo che GGCAT può avere buoni margini di miglioramento in termini di compressioni. Un'altra direzione di lavoro possibile potrebbe concentrarsi sull'ottimizzazione dell'opzione greedy matchtigs di GGCAT: la diminuzione dei tempi e della memoria richiesti, mantendendo un tale tasso di compressione, sarebbe estremamente efficiente per l'archiviazione efficiente dei dati di origine bioinformatica.

Avendo dunque considerato principalmente l'aspetto della compressione di GGCAT, possiamo affermare che risulta essere generalmente più efficiente di un programma di compressione general-purpose come GZip e quindi si conferma un ottimo tool di archiviazione di dati genetici.

Possiamo anche affermare che la compressione di insiemi di k-mers sets risulta più efficiente in termini di spazio, tempo e memoria rispetto alla compressione di k-mers sets singoli ottimizzando la compressione sulla base di rindondanze e sovrapposizioni di k-mers.

Bibliografia

[1] T. E. of Encyclopaedia Britannica, «DNA,» *Encyclopedia Britannica*, 2024. indirizzo: https://www.britannica.com/science/DNA.

[2] National Institute of Health, *The Cost of Sequencing a Human Genome*, Accesso: 20 Agosto 2024, 2021. indirizzo: https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost.

[3] W. Contributors, *Next Generation Sequencing - Wikipedia, L'enciclopedia libera*, Accesso: 21 Agosto 2024, 2024. indirizzo: https://it.wikipedia.org/wiki/Next_Generation_Sequencing.

[4] M. Hernaez, «Genomic Data Compression,» *Annual review of Biomedical Data Science*, vol. 2, pp. 19–37, 2019. doi: 10.1146/annurev-biodatasci-072018-021229.

[5] B. D. Camillo e F. Vandin, *Introduction to Sequencing Technologies*.

[6] W. Contributors, *k-mer*, Accesso: 25 Agosto 2024, 2024. indirizzo: https://en.wikipedia.org/wiki/K-mer.

[7] S. C. manekar e S. R. Sathe, «A benchmark study of k-mer counting methods for high-throughput sequencing,» *GigaScience*, 2018. doi: https://doi.org/10.1093/gigascience/giy125.

[8] G. Vurture, F. Sedlazeck, M. Nattestad et al., «GenomeScope: fast reference-free genome profiling from short reads,» *Bioinformatics*, 2017.

[9] M. Karasikov, H. Mustafa, D. Danciu et al., «Metagraph: Indexing and Analysing nucleotide Archives at Petabase-scale,» *bioRxiv*, 2020. doi: 10.1101/2020.10.01.322164.

[10] G. Holley e P. Melsted, «Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs,» *Genome Biology*, 2020. indirizzo: https://genomebiology.biomedcentral.com/articles/10.1186/s13059-020-02135-8.

[11] B. Langmead, «Aligning short sequencing reads with Bowtie,» *Curr. Protoc. Bioinformatics*, 2010.

[12] D. Wood, J. Lu e B.Langmead, «Improved metagenomic analysis with Kraken 2,» *Genome Biology*, 2019.

[13] H. Fan, A. Ives e Y. Surget-Groba, «Reconstructing phylogeny from reduced-representation genome sequencing data without assembly or alignment,» *Mol. Ecol. Resour.*, 2018.

[14] S. Altschul, W. Gish, W. Miller, E. Myers e D. Lipman, «Basic local alignment search tool,» *Journal of Molecular Biology*, 1990.

[15] I. Mouratidis, F. Baltoumas, N. Chantzi et al., «kmerDB: a database encompassing the set of genomic and proteomic sequence information for each species,» *bioRxiv*, 2023. doi: 10.1101/2023.11.13.566926.

[16] P. Compeau, P. Tesle G., «How to apply de Bruijn graphs to genome assembly,» *Nat Biotechnol 29*, pp. 987–991, 2011. doi: https://doi.org/10.1038/nbt.2023.

[17] S. et al., «Matchtigs: minimum plain text representation of k-mer sets,» *Genome Biology*, vol. 24, 2023. doi: https://doi.org/10.1186/s13059-023-02968-z.

[18] R. Rizzi, S. Beretta, M. Patterson, Y. Pirola, M. Previtali, G. Della Vedova, P. Bonizzoni, «Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era,» *Quantitative Biology*, vol. 7, pp. 278–292, 2019. doi: https://doi.org/10.1007/s40484-019-0181-x.

[19] Andrea Cracco, Alexandru I. Tomescu, «Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT,» *Genome Res*, vol. 33, n. 7, pp. 1198–1207, 2023. doi: 10.1101/gr.277615.122.

[20] K. M. Jenike, L. Campos-Domíniguez, M. Boddé et al., «Guide to k-mer approaches for genomics across the tree of life,» *Quantitative Biology*, 2024. doi: https://doi.org/10.48550/arXiv.2404.01519.

[21] C. Moeckel, M. Mareboina, M. A.Konnaris et al., «A survey of k-mer methods and applications in bioinformatics,» *Computational and structural biotechnology journal*, 2024. doi: 10.1016/j.csbj.2024.05.025.

[22] R. Chikhi, J. Holub e P. Medvedev, «Data structures to Represent a Set of k-long DNA Sequences,» *ACM Comput. Surv*, vol. 54, 2021. doi: https://doi.org/10.1145/3445967.

[23] 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*, 2022. indirizzo: https://genomebiology.biomedcentral.com/articles/10.1186/s13059-022-02743-6.

[24] M. Kokot, M. Długosz e S. Deorowicz, «KMC3: counting and manipulating k-mer statistics,» *Bioinformatics*, 2017. doi: https://doi.org/10.1093/bioinformatics/btx304.

[25] Community, *GNU Gzip*. indirizzo: https://www.gnu.org/software/gzip/.

[26] U.S. government Service and Information, *National Library of Medicine*. indirizzo: https://www.ncbi.nlm.nih.gov/.

[27] Università degli studi di Padova, *DEI docs*. indirizzo: https://docs.dei.unipd.it/en/CLUSTER/Overview.