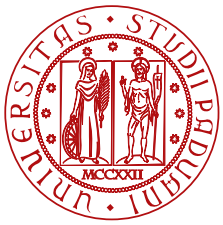


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Sviluppo di una libreria per la rappresentazione di sequenze genomiche in immagini

Relatore
Prof.ssa Pizzi Cinzia

Laureando
Tiberio Filippo

ANNO ACCADEMICO 2023-2024

Data di laurea 27/09/2024

Abstract

Questa tesi sviluppa una libreria per la rappresentazione visiva di sequenze genomiche tramite immagini, utilizzando le tecniche *Chaos Game Representation* (CGR) e *Frequency Chaos Game Representation* (FCGR). Dopo aver introdotto CGR e FCGR, vengono descritti tre metodi per calcolare la *Frequency Chaos Game Representation*: tramite discretizzazione della CGR, tramite mappatura diretta delle coordinate CGR e tramite l'utilizzo di un *k-mer counter*. L'implementazione, fatta in Python, è basata sul calcolo diretto e permette di generare rapidamente immagini FCGR da sequenze genomiche in formato FASTA.
I risultati dello sviluppo sono stati confrontati con *ComplexCGR*, uno strumento che utilizza il metodo basato su *k-mer counter*. Test eseguiti su dataset di coronavirus e cromosomi umani mostrano che la nostra libreria offre un vantaggio in termini di velocità ed efficienza in diverse casistiche.

Indice

Capitolo 1

Introduzione

La rappresentazione delle sequenze genomiche è un campo di ricerca in rapida crescita grazie alla sua capacità di fornire una visione visiva dei vari dati biologici. Le sequenze di DNA e RNA contengono informazioni genetiche fondamentali, ma la loro analisi tradizionale può risultare limitata e complessa. Per affrontare questo problema, sono state sviluppate tecniche come la *Chaos Game Representation* (CGR) e la *Frequency Chaos Game Representation* (FCGR), che mappano sequenze genomiche in immagini.

La *Chaos Game Representation* (CGR) è una tecnica introdotta originariamente per la rappresentazione visiva di sequenze simboliche, successivamente applicata alle sequenze biologiche. La CGR converte una sequenza di DNA o RNA in una rappresentazione bidimensionale basata su frattali, generando una figura che codifica informazioni sulla struttura della sequenza stessa. Questa tecnica è particolarmente apprezzata in bioinformatica per la sua capacità di evidenziare patterns ed eventuali motivi assenti nelle sequenze analizzate.

Frequency Chaos Game Representation (FCGR) si è affermata come una soluzione complementare a CGR. La FCGR suddivide l'immagine CGR in una griglia e conta la frequenza con cui i punti della rappresentazione cadono in ciascuna cella, permettendo di ottenere una visione globale più grossolana, ma più gestibile e meno rumorosa, delle sequenze genomiche. La FCGR è stata utilizzata con successo per identificare patterns sovra e sottorappresentati, oltre che per effettuare confronti tra sequenze con approcci *alignment-free*[1].

Negli ultimi anni, queste tecniche sono state utilizzate in una vasta gamma di applicazioni, grazie alla loro capacità di mappare sequenze di diversa lunghezza nello stesso spazio, il che le rende strumenti preziosi per generare dati genomici uniformi, utilizzabili in modelli di *Machine Learning*. Questi metodi, seppur meno precisi rispetto ai tradizionali approcci basati sull'allineamento, risultano molto più veloci e adatti a esplorazioni preliminari su larga scala.

Questa tesi si pone l'obiettivo di sviluppare una libreria per la rappresentazione in im-

magini di sequenze genomiche, implementando le tecniche di CGR e FCGR. Il focus principale è sull'ottimizzazione del processo di generazione delle rappresentazioni FCGR, con particolare attenzione alla velocità di calcolo. L'approccio seguito prevede l'uso del metodo geometrico a mappatura diretta per il calcolo della matrice FCGR, una soluzione che garantisce una migliore gestione delle risorse rispetto ai metodi basati sul conteggio diretto dei *k-mer* o sulla generazione dell'intera lista di coordinate CGR.

Nel capitolo 2 vengono illustrati i concetti di base, con una panoramica sulle tecniche CGR e FCGR e una discussione delle loro proprietà in bioinformatica. Il capitolo 3 si concentra sui metodi di implementazione del calcolo FCGR, presentando i tre approcci principali: da lista di coordinate CGR, a mappatura CGR diretta, e tramite *k-mer counter*. Viene descritto in dettaglio il metodo geometrico a mappatura diretta utilizzato nella libreria sviluppata. Nel capitolo 4 vengono presentati i risultati dei test eseguiti su due dataset: uno composto da 7 classi di coronavirus e uno contenente i 24 cromosomi umani. I risultati ottenuti con la libreria sviluppata vengono confrontati con quelli ottenuti tramite *ComplexCGR*, un tool basato sul metodo del conteggio dei *k-mer*. Infine, il capitolo 5 conclude con una discussione sui risultati ottenuti e suggerisce possibili sviluppi futuri.

Capitolo 2

Concetti di Base

2.1 Chaos Game Representation

La *Chaos Game Representation* (CGR) è una tecnica introdotta in [2] e successivamente estesa al DNA in [3] per la rappresentazione visuale di una sequenza di caratteri con un'immagine. Nel corso degli anni sono stati sviluppati diversi algoritmi per la costruzione dell'immagine, e si sono scoperte nuove applicazioni, tanto da renderlo uno strumento chiave della bioinformatica moderna.
Recentemente, CGR è stata applicata anche ad altri dati biologici [1], come RNA o sequenze di amminoacidi, mantenendo lo stesso concetto di base: mappare una sequenza di caratteri unidimensionale in uno spazio di dimensione maggiore.

2.1.1 Costruzione di CGR genomica

È possibile mappare una sequenza di DNA su un quadrato con centro in $(0,0)$ assegnando ad ogni vertice un nucleotide e una direzione:

$$A = (-1,1), \quad C = (-1,-1), \quad G = (1,-1), \quad T = (1,1)$$

Partendo dal centro, come in Figura 2.1, ad ogni carattere della stringa, ci si sposta in direzione del vertice corrispondente con spostamenti frattali basati su un coefficiente denominato *scaling factor* $0 < sf < 1$. Per il DNA $sf = 0.5$.

Il metodo geometrico è quello più utilizzato negli ultimi anni per la creazione della rappresentazione CGR, e grazie a questo, si è giunti alla seguente rappresentazione compatta:

$$P_i^j = P_{i-1}^j + sf(V_{i-1}^j - P_{i-1}^j) \tag{2.1}$$

dove:

- j: l'indice della dimensione considerata;
- i: l'indice del carattere considerato nella sequenza di partenza S;
- P_0^j: la coordinata di partenza per la dimensione j;
- P_i^j: la coordinata i-esima per la dimensione j;
- sf: il valore di *scaling factor*. 0.5 per DNA/RNA;
- V_i^j: la direzione assegnata al carattere S_i per la dimensione j;

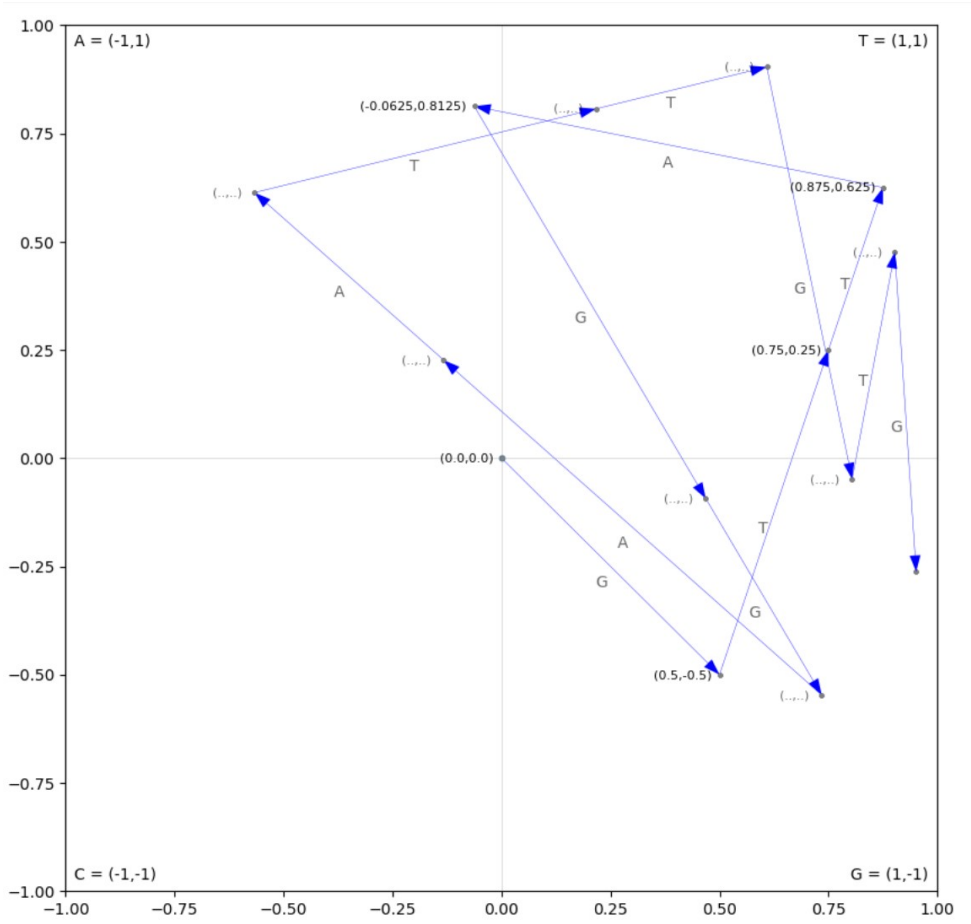


Figura 2.1: Esempio di CGR applicata alla sequenza di DNA: *GTTAGGAATTGTG*

Generalizzando, è possibile rappresentare sequenze con un numero qualsiasi di basi semplicemente aumentando la dimensione dello spazio di destinazione (ad esempio passando da $\mathbb{R}^2$ a $\mathbb{R}^3$) oppure aumentando il numero di vertici come in Figura 2.2.

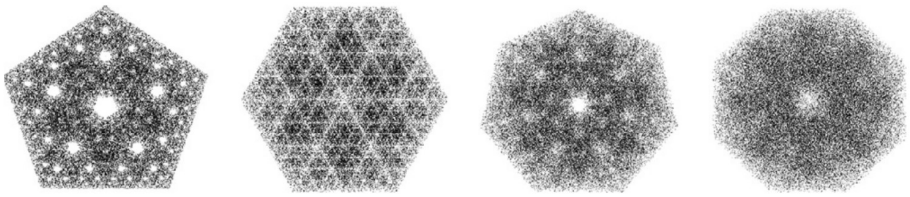


Figura 2.2: Esempio di CGR generalizzata con un numero variabile di vertici, da 5 a 8, su sequenze randomiche e *scaling factor* di 0.5 [1]

2.1.2 Proprietà di CGR

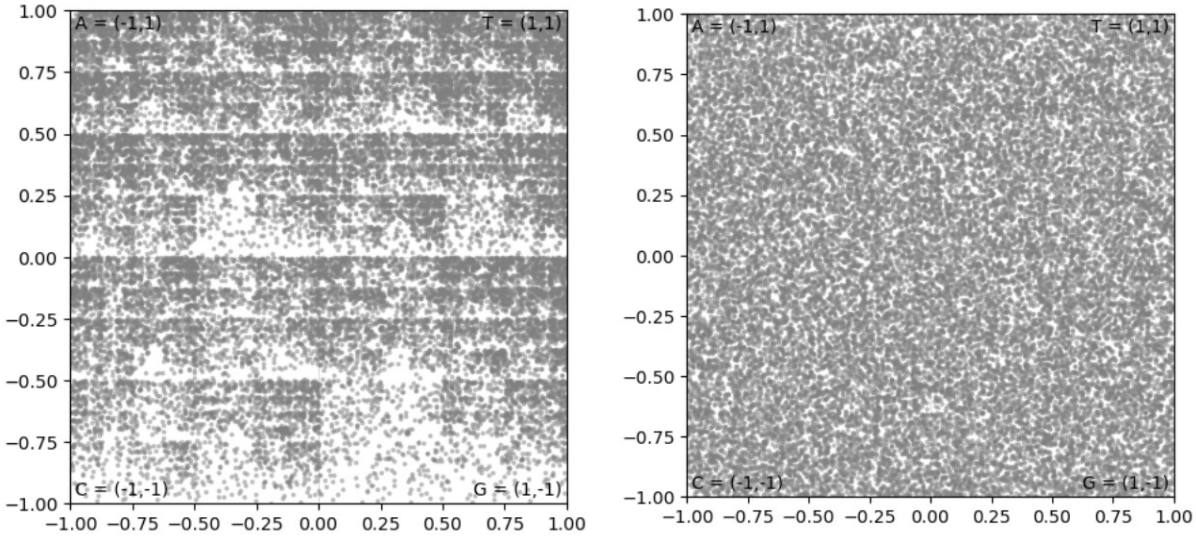
CGR possiede delle interessanti proprietà che hanno contribuito alla popolarità della tecnica[1]:

- Una sequenza è rappresentata da un unico pattern;
- La relazione tra sequenza e coordinata finale è univoca;
- La tecnica della CGR mappa qualsiasi sequenza nello stesso spazio di dimensione maggiore e il punto di partenza influenza solo in minima parte il risultato. Questo ne facilita il confronto.

Da queste proprietà ne deriva un ulteriore fondamentale corollario, ovvero che la coordinata di destinazione codifica interamente la sequenza sorgente. Oltre a quelle precedentemente definite, Burma et al. [4] ha riconosciuto ulteriori proprietà legate all'applicazione di CGR alle sequenze di DNA:

- Usando CGR è possibile identificare sequenze sovrarappresentate, sottorappresentate o assenti.
- Le similitudini nei patterns CGR indicano l'omologia delle due sequenze di partenza.
- Applicando CGR a sottosequenze si ottengono gli stessi patterns caratteristici che si otterrebbero applicandola al genoma di partenza.

Applicando questa tecnica si è osservato inoltre che nella rappresentazione finale emergono dei patterns caratteristici per ogni sequenza ma solo se questa non è randomica come ben visibile in Figura 2.3. L'assegnamento delle basi è arbitrario e si è osservato che a configurazioni diverse corrispondono patterns diversi.



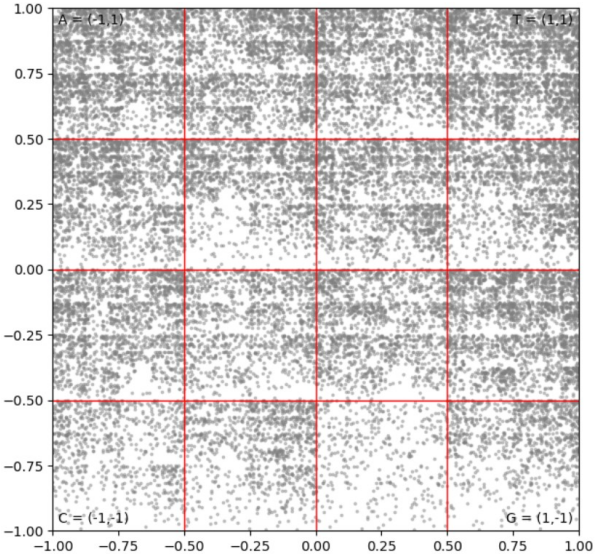
(a) (b)

Figura 2.3: A sinistra (a): esempio di CGR con *scaling factor* 0.5 sulla sequenza di Sars-Cov-2 (GenBank:MW166052.1) lunga 29796 caratteri. A destra (b): esempio di CGR con parametri identici ma su una sequenza generata randomicamente lunga sempre 29796 caratteri.

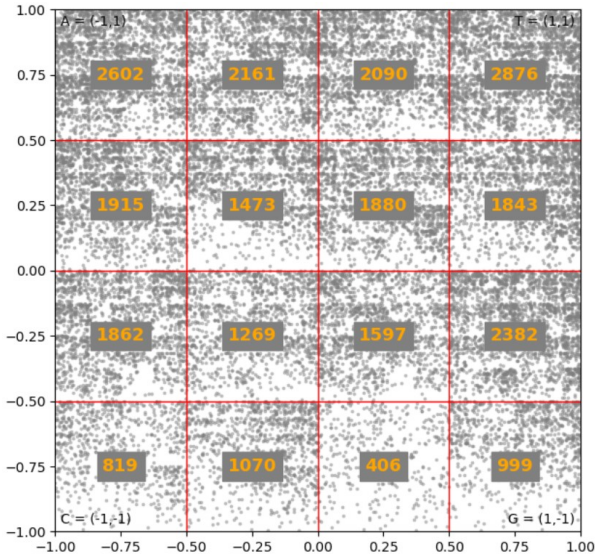
2.2 Frequency Chaos Game Representation

Per ottenere una rappresentazione più grossolana ma anche meno rumorosa è possibile discretizzare CGR sovrapponendo al risultato una griglia quadrata di dimensione $n \times n$ e contando quanti punti della rappresentazione sono contenuti in ogni cella.

Se $n = 2^k$, allora il risultato di questa procedura, chiamata *Frequency Chaos Game Representation* (FCGR), consiste in una matrice contenenete il conteggio dei vari *k-mer* della sequenza di partenza ed eventualmente un'immagine se questi vengono convertiti in valori di una scala di grigi come in Figura 2.6. In [4] è stato inoltre dimostrato che, come per CGR, FCGR permette di identificare l'omologia tra due sequenze e di riconoscere *k-mer* sotto o sovra rappresentati.



(a) Applicazione della griglia $2^2 \times 2^2$ alla rappresentazione CGR.



(b) Conteggio dei punti contenuti nelle celle della griglia.

Figura 2.4: Discretizzazione della rappresentazione CGR ($sf = 0.5$) di una sequenza di Sars-Cov-2 (GenBank:MW166052.1)"

.

Per il DNA, ad esempio, FCGR associa i *k-mer* alle celle della matrice secondo la seguente relazione:

$$\overbrace{\begin{bmatrix} A & T \\ C & G \end{bmatrix} \otimes \ \ldots \ \otimes \begin{bmatrix} A & T \\ C & G \end{bmatrix}}^{k-1 \text{ times}}$$

dove l'operatore non commutativo $\otimes$ corrisponde alla seguente operazione matriciale:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \otimes \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot E & \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot F \\ \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot G & \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot H \end{bmatrix} = \begin{bmatrix} AE & BE & AF & BF \\ CE & DE & CF & DF \\ AG & BG & AH & BH \\ CG & DG & CH & DH \end{bmatrix}$$

ottenendo quindi una divisione come in Figura 2.5.

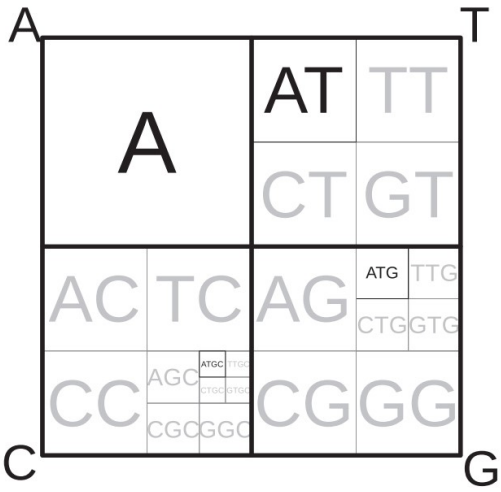
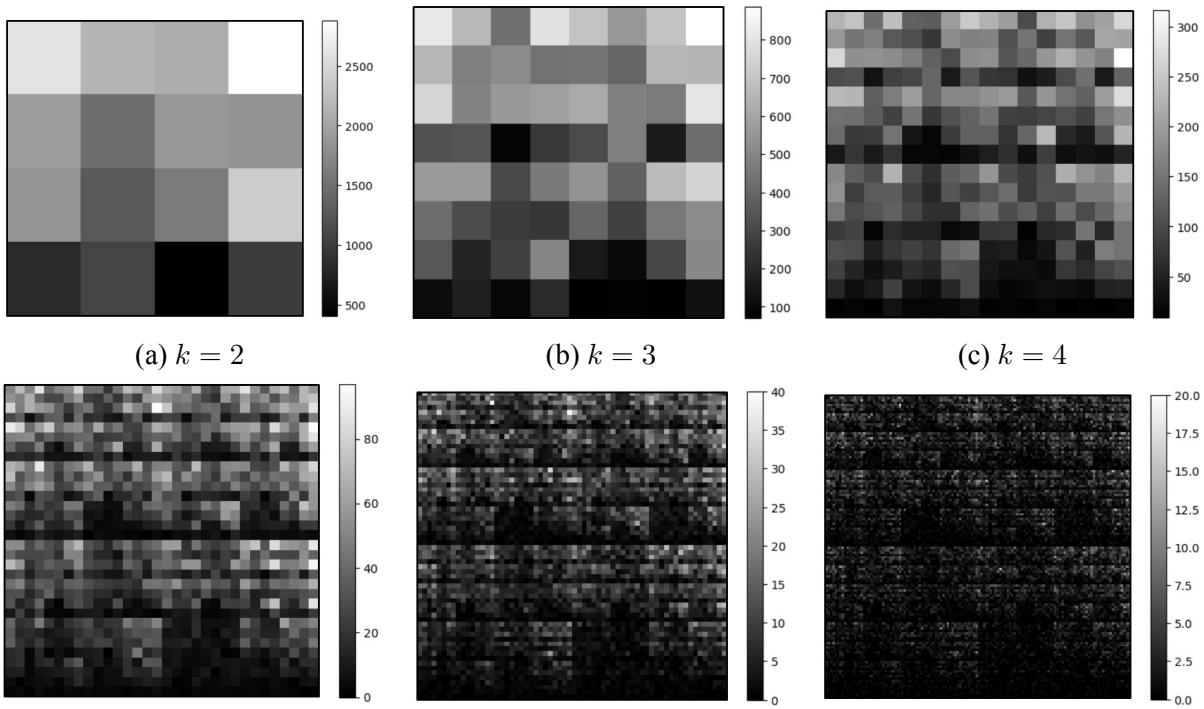


Figura 2.5: Esempio di suddivisione dei *k-mer* ottenuta con FCGR [1]



(a) $k = 2$ (b) $k = 3$ (c) $k = 4$

(d) $k = 5$ (e) $k = 6$ (f) $k = 7$

Figura 2.6: Esempi di FCGR sulla sequenza di Sars-Cov-2 (GenBank:MW166052.1) al variare di k.

In alternativa, considerando il significato intrinseco della rappresentazione, ovvero il conteggio dei *k-mer* della sequenza, è possibile costruire FCGR anche senza l'utilizzo di CGR ma sfruttando un normale *k-mer counter* e popolando successivamente la matrice con i risultati della scansione.

Capitolo 3

Metodi e Implementazione

In questo capitolo verranno analizzate le modalità di analisi e di implementazione utilizzate durante lo sviluppo del tool.

3.1 Metodi

Per la generazione della rappresentazione FCGR emergono 3 metodi principali che possono essere riassunti in:

1. Metodo 1 - Calcolo CGR e discretizzazione in FCGR
2. Metodo 2 - Mappatura diretta di CGR in FCGR
3. Metodo 3 - Utilizzo di un *k-mer counter*

In Tabella 3.1 sono riassunte le informazioni descritte nei successivi paragrafi.

3.1.1 Metodo 1 - Calcolo CGR e discretizzazione in FCGR

Questo approccio consiste nello scorrere la sequenza di n caratteri, calcolare tutte le coordinate dei punti CGR e solo successivamente produrre la rappresentazione FCGR contando quanti punti ricadono in una determinata cella dato un determinato valore di k.

Il vantaggio principale di questa metodica è che, in caso di analisi con valori multipli di k, la sequenza iniziale viene analizzata solamente la prima volta. Un ulteriore pregio di questo metodo è la possibilità di ottenere un doppio output: CGR e FCGR. Lo svantaggio principale, invece, è quello di dover tenere in memoria tutte le n coordinate e di doverle scansionare anche per analisi con un singolo valore di k.

Possiamo quindi ricondurre la complessità del calcolo di FCGR alla seguente espressione:

$$O\Big(NM \cdot \underbrace{n}_{\substack{\text{Scansione sequenza +} \\ \text{scansione coordinate}}}\Big)$$

dove M indica il numero di diversi valori di k utilizzati e N è il numero di sequenze analizzate. Nell'espressione non compare il termine k in quanto, dato che è possibile ricondursi alla cella FCGR direttamente dalla coordinata CGR, non è mai necessario scansionare l'intera matrice.

3.1.2 Metodo 2 - Mappatura diretta di CGR in FCGR

Questo approccio, scelto come metodo di riferimento in questa tesi, consiste nello scorrere la sequenza di n caratteri e per ognuno di questi calcolare la coordinata CGR e mapparla direttamente nella matrice.

Il vantaggio principale di questa metodica è che non risulta necessaria la scansione iniziale della sequenza e, al contrario del metodo precedente, non è necessario tenere in memoria la lista delle coordinate CGR. Lo svantaggio principale è quello di dover scansionare la sequenza e analizzarne i caratteri più volte in caso di analisi con più valori di k

Possiamo quindi ricondurre la complessità del calcolo di FCGR alla seguente espressione:

$$O\Big(NM \cdot \underbrace{n}_{\text{Scansione sequenza}}\Big)$$

A differenza del metodo precedente, in fase di mappatura, è necessario anche calcolare la coordinata. Questo se dal punto di vista temporale risulta essere uno svantaggio, dal punto di vista spaziale permette di scansionare progressivamente la sequenza e iniziare direttamente con la creazione di FCGR senza dover necessariamente attendere di avere l'intera sequenza, o l'intera lista di coordinate, in memoria. In caso di sequenze particolarmente lunghe risulta vantaggioso non dover mantenere l'intero set di dati.

3.1.3 Metodo 3 - Utilizzo di un k-mer counter

Un approccio alternativo, significativamente diverso rispetto a quelli precedentemente descritti, si basa sul significato della rappresentazione FCGR e costruisce la matrice a partire dalle informazioni che questa racchiude al termine dell'analisi, ossia il conteggio dei *k-mer* della se-

quenza. Il metodo procede dunque associando direttamente, tramite l'uso di un *k-mer counter*, il conteggio di tutti i *k-mer* della sequenza alle celle delle matrice.

Il vantaggio principale di questo approccio sono la possibilità di utilizzare un *k-mer counter* esterno e, quindi poter scegliere lo strumento più efficiente caso per caso, e la possibilità di poter eseguire l'analisi FCGR anche su *k-mer canonici*. Un altro vantaggio di questa tecnica è che, al contrario degli approcci precedenti, l'eventuale informazione del conteggio di kappameri è già disponibile e non è necessaria quindi un'ulteriore scansione della matrice. Lo svantaggio principale invece è dato dal fatto che, al contrario delle coordinate CGR, non è possibile mappare direttamente il conteggio del *k-mer* a una cella: è infatti necessario costruire un indice che permetta di associare i 4^k *k-mers* alle celle della matrice. Questo svantaggio risulta meno incisivo se, per un valore di k fissato, vengono analizzate più sequenze: questo perché è possibile riutilizzare l'indice creato durante la prima analisi senza doverlo ricostruire ad ogni sequenza.

Supponendo l'inserimento dei conteggi nel *k-mer counter* $O(1)$ possiamo dunque ricondurre la complessità del calcolo di FCGR alla seguente espressione:

$$\sum_{k \in K} \Bigg(\underbrace{O(4^k)}_{\substack{\text{Creazione indice delle coordinate}}} \Bigg) + O\Bigg(NM \cdot \underbrace{(n+D)}_{\substack{\text{Scansione sequenza e calcolo dei conteggi +} \\ \text{Mappatura dei conteggi}}} \Bigg)$$

dove D indica il numero di kappameri unici presenti nella sequenza analizzata e $K = \{k_1, ..., k_M\}$ l'insime di valori di k utilizzati per l'analisi.

	Vantaggi	Svantaggi	Complessità
Metodo 1	Più efficiente per analisi con più valori di k. Doppia informazione CGR+ FCGR	Scansione iniziale della sequenza ; Maggior utilizzo di memoria	$O(NM \cdot n)$
Metodo 2	Ottime performance spaziali; Unica scansione, non necessita scansione iniziale	Solo FCGR in output	$O(NM \cdot n)$
Metodo 3	Possibilità di analisi su *k-mer* canonici; *K-mer counter* esterni; Doppia informazione: FCGR + conteggio dei kmer Efficiente per analisi di più sequenze con k fissato	Importante overhead per mappare i conteggi; Non si dispone delle coordinate CGR	$\sum_{k \in K} \Big(O(4^k) \Big) + O\Big(MN \cdot (n+D) \Big)$

Tabella 3.1: Tabella riassuntiva della comparazione dei tre metodi per la generazione della rappresentazione FCGR

3.2 Implementazione

Per lo sviluppo di un tool efficiente per la rappresentazione FCGR di sequenze genomiche, con un focus principale su DNA e RNA, si è deciso di basare lo strumento sulla tecnica di generazione geometrica a mappatura diretta *(Metodo 2)* e di scartare a priori le altre due metodologie. Inizialmente si era scelto di implementare il progetto in linguaggio C++ per sfruttare i vantaggi offerti da questo linguaggio in termini di gestione della memoria e velocità di esecuzione. Tuttavia, durante lo sviluppo, si è deciso di migrare verso Python3 per garantire una maggiore

compatibilità con gli strumenti esistenti e per agevolare il confronto delle prestazioni con altri tools simili, anch'essi basati su Python.
Il tool sviluppato consiste in un pacchetto Python3 che permette di generare in modo efficiente l'immagine della rappresentazione FCGR a partire da sequenze di DNA e RNA in formato FASTA sfruttando il metodo geometrico.

3.2.1 Il formato FASTA

Il FASTA è un formato di file testuale largamente utilizzato in bioinformatica per la rappresentazione di sequenze genomiche in cui ciascun elemento viene rappresentato con una lettera. Nel file possono essere presenti più sequenze e ognuna di queste è composte da:

- Un preambolo, le cui righe iniziano con il simbolo '>', contenente un identificatore di sequenza ed eventuali altre informazioni come il nome dell'organismo di provenienza o annotazioni varie.
- La sequenza vera e propria, su una o più righe.

Nella tabella 3.2 sono indicati i caratteri utilizzati per rappresentare DNA e RNA e quali di questi sono stati implementati o meno nel tool sviluppato, mentre l'Esempio 3.2.1 mostra una possibile rappresentazione di una sequenza.

Lettera	Nucleotide	Lettera	Nucleotide
A	A - Adenina	**M**	A o C
C	C - Citosina	**S**	C o G
G	G - Guanina	**W**	A, T o U
T	T - Tiamina	**B**	non A (ovvero C, G, T or U)
U	U - Uracile	**D**	non C (ovvero A, G, T or U)
(i)	i - Inosina	**H**	non G (ovvero, A, C, T or U)
R	A o G (I)	**V**	né T né U (ovvero A, C or G)
Y	C, T o U	**N**	A C G T U
K	G, T o U		

Tabella 3.2: Elenco caratteri FASTA per DNA e RNA e relativo supporto del tool sviluppato. I simboli non supportati vengono ignorati durante la generazione delle coordinate CGR e relativa rappresentazione FCGR.

Esempio 3.2.1

```
>CM027413.1 Hydrochoerus hydrochaeris isolate Pop chromosome 1,
>whole genome shotgun sequence
GTTAGGAATTGTGATAGTGTTTCTCATTGCTTTAAATAATTTTCACTATTTTCCAAAATGCCTAC
AGCATTAACACGCACCACTTTCTTATGACTTCATGTTATTTCAGTTAATTCTGTCAGTGTAAATA
```

GAAAATTCACAACTAATGGCTTCTGTTTTTTCCTTCTGCTTATTTGTGTGTTTGAGCAGTTAAAT
GAACCAAAAGAAATCCCTAAGATATGAACCAATATTCTCCAAAAACAGTTCTTATGCATTAAAGT
ATAGTAGAAATTCCAGTTTT...

3.2.2 Modalità di lavoro e gestione del codice

L'intero sviluppo è stato basato sulla necessità di ottenere uno strumento che fosse il più performante possibile mantenendo, però, la massima usabilità e tutte le features chiave che questo tool dovrebbe avere. Partendo da un'implementazione basilare, ad incrementi successivi, si sono analizzate e ridisegnate componenti del sistema in modo da renderle più veloci ed efficienti. Ad ogni incremento seguiva una fase di test per verificare che questo comportasse un miglioramento effettivo e che non andasse a deteriorare la qualità dell'output o del tool stesso.

Per il mantenimento del codice e la gestione del versionamento si è optato per l'utilizzo di *Git*, ospitando una repository su *Github*, e di distribuire il tool finale come un pacchetto Python caricandolo su *PyPI*.

Il codice sorgente è visionabile al seguente indirizzo https://github.com/FilippoTib/FastFCGR, mentre il tool è disponibile al link https://pypi.org/project/fastfcgr/.

3.2.3 Librerie utilizzate

Per la realizzazione si sono sfruttate tre librerie ausiliarie che hanno contribuito a rendere più leggibile e veloce il codice:

- **Math:** Modulo contenente le funzioni matematiche elementari definite dallo standard C[5]. È stata utilizzato per gestire arrotondamenti e logaritmi.
- **NumPy:** Libreria utilizzata per la gestione efficiente di array in Python[6]. È stata utilizzata per tenere in memoria e manipolare la matrice generata da FCGR.
- **Pillow:** Libreria utilizzata per l'elaborazione di immagini in Python[7]. È stata utilizzata per l'esportazione efficiente in formato immagine della matrice FCGR.

3.2.4 Codice

Il codice è raggruppato in una classe Python che consiste principalmente in 5 sezioni:

- Costruttore, dichiarazione attributi e *properties (getters)*.
- Metodi di input.
- Metodi principali.

- Metodi di output.
- Metodi ausiliari.

Costruttore, dichiarazione attributi e properties

All'interno della classe è presente un unico costruttore che inizializza gli attributi. Per ciascuno di essi viene inoltre implementata una *property* con funzionalità di *getter*:

```
def __init__(self):
    self.__sequence = []
    self.__k = 0
    self.__matrix = None
    self.__isRNA = False
    self.__maxValue = 0
    self.__currMatrixSize = 0
```

Metodi di input

Sono presenti due metodi di input, uno da file in formato FASTA e uno da stringa, i quali caricano in memoria la sequenza da analizzare. Sono entrambi programmati per lanciare un'eccezione se la stringa è stata già caricata in precedenza ma questo comportamento è bypassabile utilizzando il flag '`force: bool`'

Core

Il core del tool è composto da due metodi: `initialize` per inizializzare la matrice e `calculate` per popolarla con la rappresentazione FCGR.

Si è deciso di creare un metodo dedicato per l'*init* della matrice, e di non fonderne il comportamento all'interno del costruttore, per permettere l'analisi con più valori di k della stessa sequenza senza dover obbligatoriamente ricreare l'oggetto ad ogni iterazione.

```
def initialize(self, k, isRNA:bool=False):
    matrixSize = int(2 ** k)
    self.__currMatrixSize = matrixSize
    self.__matrix = np.zeros((matrixSize, matrixSize),
                    dtype=np.uint16 if k > 8 else np.uint32)
    self.__maxValue = 0
    self.__k = k
    self.__isRNA = isRNA
```

Il metodo calculate ricalca perfettamente il paradigma del metodo geometrico. Di default, lavora con uno *scaling factor* di 0.5 e in caso di carattere non supportato lo ignora e procede passando direttamente a quello successivo. È importante notare come i primi $k-1$ caratteri vengano considerati solo per il calcolo delle coordinate CGR ma non contribuiscano nella matrice in quanto non consistono in veri e propri *k-mer* ma solamente prefissi del primo kappamero.

```
def calculate(self, scalingFactor:float=0.5):
    lastX, lastY = 0.0, 0.0
    self.__maxValue = 0
    halfMatrixSize = self.__currMatrixSize / 2
    valid_bases = {'A', 'C', 'G'} | ({'U'} if self.__isRNA else {'T'})

    temp_matrix = {}
    for i in range(1, len(self.__sequence) + 1):
        base = self.__sequence[i - 1]
        if base not in valid_bases: #skip invalid characters
            continue

        #calculate directions
        dirX : float = 1 if base in {'T','G','U'} else -1
        dirY = 1 if base in {'A','T','U'} else -1

        #calculate new coordinates
        lastX += scalingFactor * (dirX - lastX)
        lastY += scalingFactor * (dirY - lastY)

        if(i < self.__k): #skip first k-1 iterations
            continue

        #calculate matrix coordinates
        x = int(math.floor((lastX + 1.0) * halfMatrixSize))
        y = int(math.floor((1.0 - lastY) * halfMatrixSize))

        #check for round errors
        x = x if x < self.__currMatrixSize else self.__currMatrixSize - 1
        y = y if y < self.__currMatrixSize else self.__currMatrixSize - 1
```

```
            if (y, x) in temp_matrix:
                temp_matrix[(y, x)] += 1
            else:
                temp_matrix[(y, x)] = 1

            tmp_val = temp_matrix[(y, x)] #update max value
            if tmp_val > self.__maxValue:
                self.__maxValue = tmp_val

        for (y, x), value in temp_matrix.items():
            self.__matrix[y, x] = value

        return self.__maxValue
```

Per massimizzare la velocità del calcolo si è deciso di tenere traccia solamente del massimo valore presente nella matrice e di assumere quindi che il minimo sia sempre lo 0. Questa approssimazione può ritenersi valida in quanto, per le proprietà di distribuzione dei *k-mer*, è ragionevole ipotizzare che per valori di k sufficientemente grandi ci siano dei kappameri assenti nella sequenza iniziale[8].

Metodi di output

Sono presenti due metodi di output:

- Uno per la semplice stampa della matrice in formato numerico
- Uno per il salvataggio in formato immagine della matrice

```
    def save_image(self, path:str, d_max:int=255):
        # rescale matrix values to fit d_max value
        normalized_matrix =
            FastFCGR.__rescale_interval(self.__matrix, self.__maxValue, d_max)

        # convert to PIL image and save
        image = Image.fromarray(
            grayscale_image,
            mode = FastFCGR.__pillow_mode_from_bits(
                FastFCGR.__num_bits_needed(d_max)
            )
```

```
        )
        image.save(path)
```

Quest'ultimo prende in input il *path* in cui salvare il file e il valore massimo desiderato di un pixel e salva su disco un'immagine in scala di grigi calcolando, grazie ai metodi ausiliari, la profondità in bit per $pixel$ necessaria.

Metodi ausiliari

Sono presenti dei metodi ausiliari che rendono più semplice e chiara la funzione di salvataggio dell'immagine su file:

- **`__num_bits_needed`**: calcola quanti bit per pixel sono necessari per rappresentare correttamente i valori da 0 a d_max.

```
@staticmethod
def __num_bits_needed(n:int):
    return 1 if n == 0 else math.ceil(math.log2(n + 1))
```

- **`__numpy_type_from_bits`**: ritorna il corretto tipo *numpy* necessario a rappresentare dati espressi in n bits.

- **`__pillow_mode_from_bits`**: ritorna la corretta modalità di conversione di *pillow* necessaria a rappresentare dati espressi in n bits per pixel.

- **`__rescale_interval`**: ritorna una copia della matrice con i valori riportati nell'intervallo $[0, d\_max]$.

```
@staticmethod
def __rescale_interval(value, s_max:int, d_max:int):
    mat = d_max - ((value / s_max) * d_max)
    return mat.astype(
        FastFCGR.__numpy_type_from_bits(
            FastFCGR.__num_bits_needed(d_max)
        )
    )
```

3.3 Considerazioni sui tempi di esecuzione

Il tool costruito in questo modo, quindi aggiornando il massimo ad ogni iterazione e approssimando il valore di minimo, permette di generare la matrice eseguendo una sola scansione della sequenza iniziale. Risulta agevole suddividere l'elaborazione in tre fasi distinte:

1. **Inizializzazione**: allocazione della matrice che ospiterà i valori FCGR e caricamento in memoria della sequenza genomica;
2. **Elaborazione**: calcolo della FCGR partendo dalla sequenza caricata in memoria;
3. **Esportazione**: salvataggio dell'immagine finale.

In Figura 3.1 è possibile notare come, al crescere del valore di k, la fase di elaborazione diventi superflua mentre la fase di esportazione risulti predominante. Per valori di k sufficientemente grandi inizia ad essere apprezzabile anche la fase di inizializzazione della matrice.

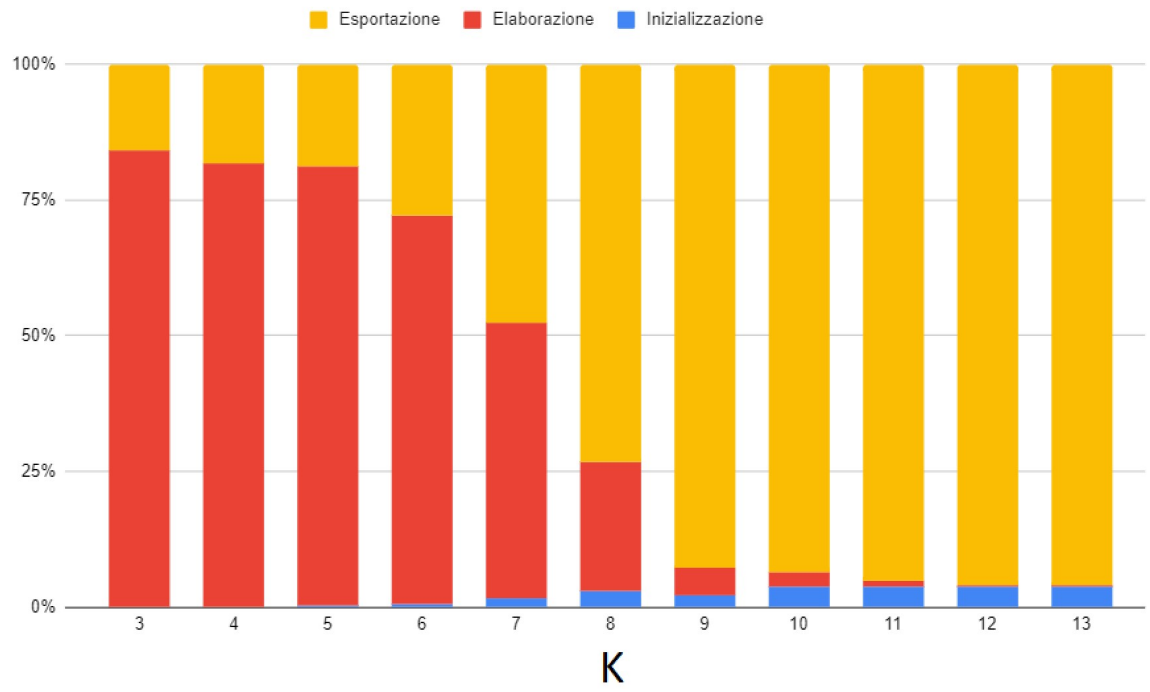


Figura 3.1: Ripartizione del tempo utilizzato per ciascuna fase (Inizializzazione, Elaborazione ed Esportazione) al variare di k. Test eseguito su un dataset di 847 sequenze di lunghezza media 29598 caratteri.

Capitolo 4

Risultati

In questo capitolo verranno introdotti i dataset utilizzati per confrontare il tool con altri strumenti e verranno descritti i risultati ottenuti.

4.1 Ambiente e Tool di riferimento

Si è deciso di confrontare il tool sviluppato in questa analisi con il modulo FCGR dello strumento *ComplexCGR*, un tool basato sulla tecnica del *k-mer counting*. Nonostante la possibilità di utilizzare un *k-mer counter* esterno, per tutti i test è stato utilizzato quello nativo, integrato nello strumento. Inoltre, si è scelto di comparare solamente i *metodi* 2 *e* 3, poiché, per analisi con singoli valori di k, il *metodo* 1 risulterebbe sicuramente più lento, anche se di poco, rispetto al metodo a mappatura diretta utilizzato dal tool sviluppato in questo studio. Questo perché, come discusso nei capitoli precedenti, il *metodo* 1 condivide la maggior parte delle caratteristiche con il *metodo* 2, ma richiede anche una pre-scansione e l'uso di memoria aggiuntiva.

I test sono tutti stati eseguiti in una macchina (*Personal Computer*) con le seguenti caratteristiche:

- **SO**: Win10
- **CPU**: Intel I5-10400F - 6C12T@4.30GHz
- **RAM**: 32GB DDR4 - 3600MHz
- **Disco**: NVME - Dedicato all'analisi e non utilizzato da altri processi

4.2 Dataset

Per il confronto del tool si sono utilizzati due dataset: uno composto da 7 classi di coronavirus e uno composto dall'insieme dei cromosomi umani.

4.2.1 7 classes coronavirus

Il dataset è costituito da 874 sequenze, di lunghezza media 29 598 caratteri, suddivise in 7 classi:

- HCoV-229E
- HCoV-HKU1
- HCoV-NL63
- HCoV-OC43
- MERS-CoV
- SARS-COV-1
- SARS-COV-2

Di queste 874 sequenze, data la presenza di caratteri non supportati da entrambi gli strumenti, 175 sono state scartate e non rientrano in alcun modo nelle successive statistiche temporali.
In Figura 4.1 è riportata la ripartizione del dataset nelle 7 classi. La lunghezza minima di ogni sequenza è di 20 520 caratteri, mentre la massima è di 30 818.
Si è scelto questo dataset in quanto rappresenta il caso critico del metodo geometrico; infatti, sequenze corte e analizzate con valori di k bassi ($k < 8$) rientrano nel tipo di analisi in cui le criticità del metodo basato su *k-mer counter* risultano meno incisive.

È possibile visionare e scaricare il dataset al link: `https://github.com/SAkbari93/WalkIm`.

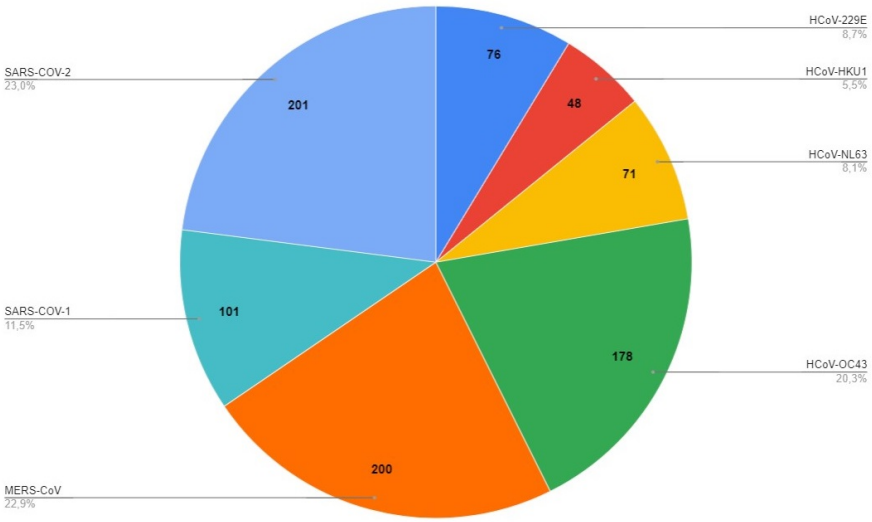
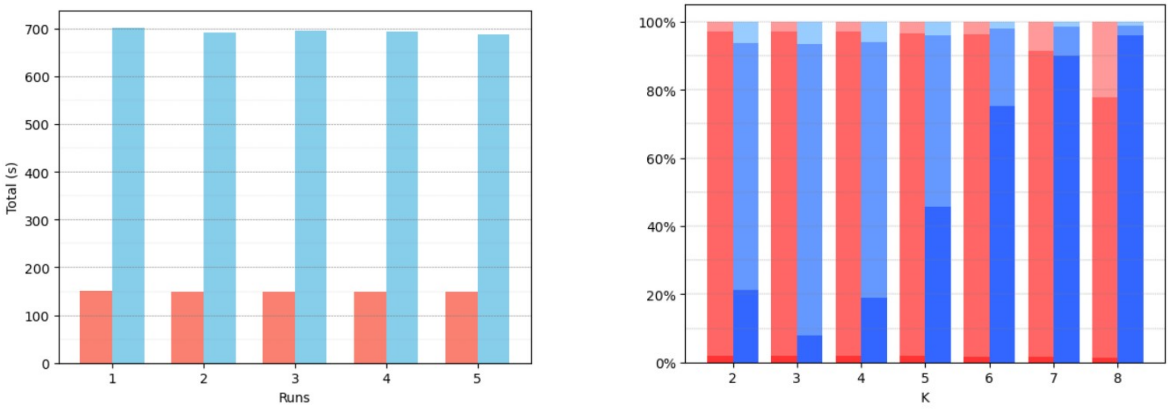


Figura 4.1: Ripartizione del dataset '7 classes coronavirus' nelle 7 classi.

Esperimento e risultati

Per questo dataset, composto principalmente da sequenze corte, si è optato per l'utilizzo di valori relativamente bassi di k e in particolare sono stati utilizzati valori interi nell'intervallo $[2,8]$. Per ogni sequenza del dataset è stata eseguito in successione il test per tutti i valori di k ripetendo, ad ogni iterazione, l'intera analisi, senza sfruttare quindi la proprietà di analisi multipla di una singola istanza di questo strumento. In questo modo, è possibile rendere equilibrato il confronto tra gli strumenti e valutare, quindi, principalmente le differenze tra le due metodiche. Per mediare eventuali oscillazioni dovute all'ambiente di test, l'analisi è stata ripetuta 5 volte e i risultati sono stati mediati. Le immagini sono state salvate in formato PNG in scala di grigi con profondità di $8bpp$.

In Figura 4.2 sono riportati il tempo totale di elaborazione (sinistra) e la ripartizione nelle varie fasi di calcolo (destra), mentre in Figura 4.3 viene riportato il tempo medio di elaborazione per i vari valori di k.



(a) (b)

Figura 4.2: In rosso il tool di questa tesi, in blu *ComplexCGR*.
A sinistra (a): tempo impiegato in secondi per ognuna delle 5 runs da ciascuno dei due tools. A destra (b): ripartizione del tempo di calcolo tra inizializzazione, elaborazione ed esportazione (rispettivamente dal più scuro al più chiaro), per i vari valori di k

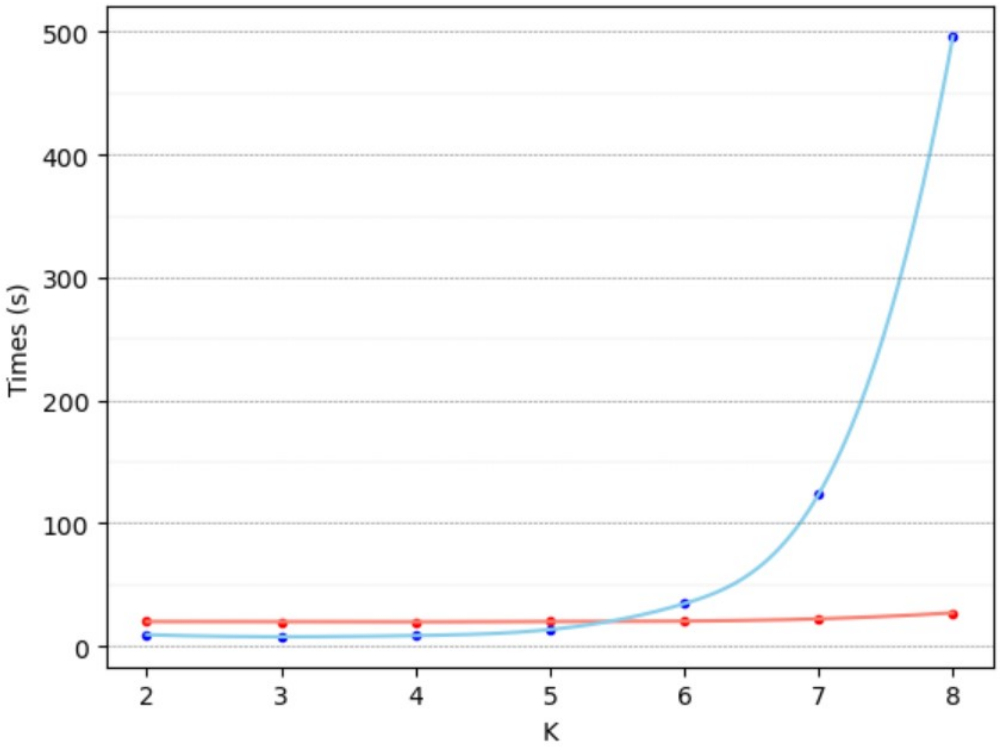


Figura 4.3: Tempo medio impiegato da entrambi i tools, quello di questa tesi in rosso mentre *ComplexCGR* in blu, per produrre la rappresentazione FCGR per i vari valori di k.

Si può intuire dai grafici come il tool di questo studio risulti essere più lineare al crescere di k anche se, per valori di k sufficientemente bassi ($k < 6$), il tool basato su *k-mer counter* risulta essere più veloce.

4.2.2 Human Chromosomes

Il dataset è costituito da 24 sequenze, di lunghezza media $128\,677\,909$ caratteri, minima $46\,709\,983$ e massima $248\,956\,422$, comprendente i 24 cromosomi umani.

È possibile visionare e scaricare le sequenze del dataset al link: `https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.40`.

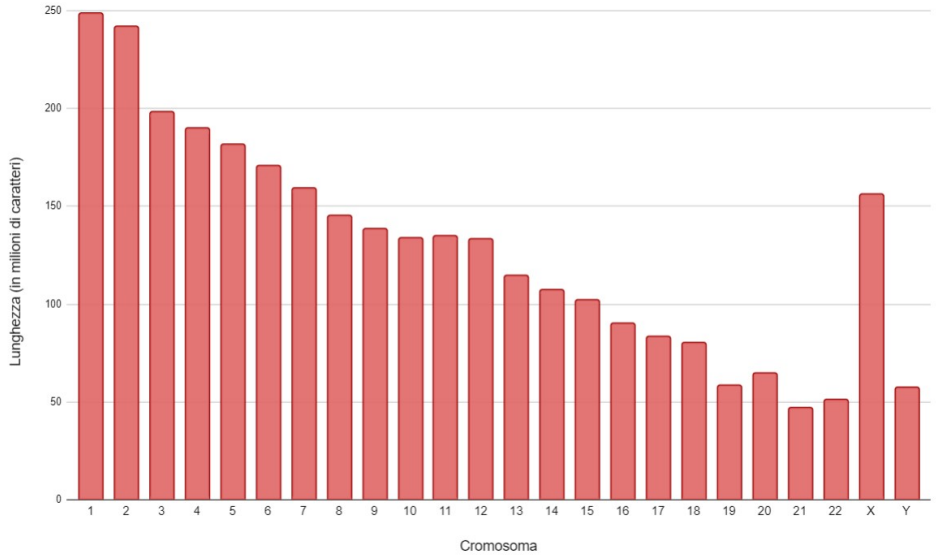


Figura 4.4: Lunghezza (in milioni di caratteri) delle sequenze dei 24 cromosomi umani.

Esperimento e risultati

Per questo dataset, composto da poche sequenze particolarmente lunghe, si è scelto un range di valori di k maggiori di quelle del test precedente e in particolare sono stati utilizzati i valori $\{8{,}10{,}12{,}14\}$. Come per il test precedente, per ogni sequenza del dataset è stata eseguito in successione il test per tutti i valori di k ripetendo ad ogni iterazione l'intera analisi.
In Figura 4.5 viene riportato il tempo di elaborazione per i vari valori di k. Per lo strumento *ComplexCGR*, con il valore di $k = 14$, il test è stato interrotto dopo 8 ore di calcolo sulla prima sequenza. Per verificare che il problema non fosse legato al *k-mer counter* integrato nello strumento *ComplexCGR*, è stato eseguito un test, per il solo valore di $k = 14$, utilizzando anche KMC[9], un contatore esterno, ottenendo però il medesimo risultato.

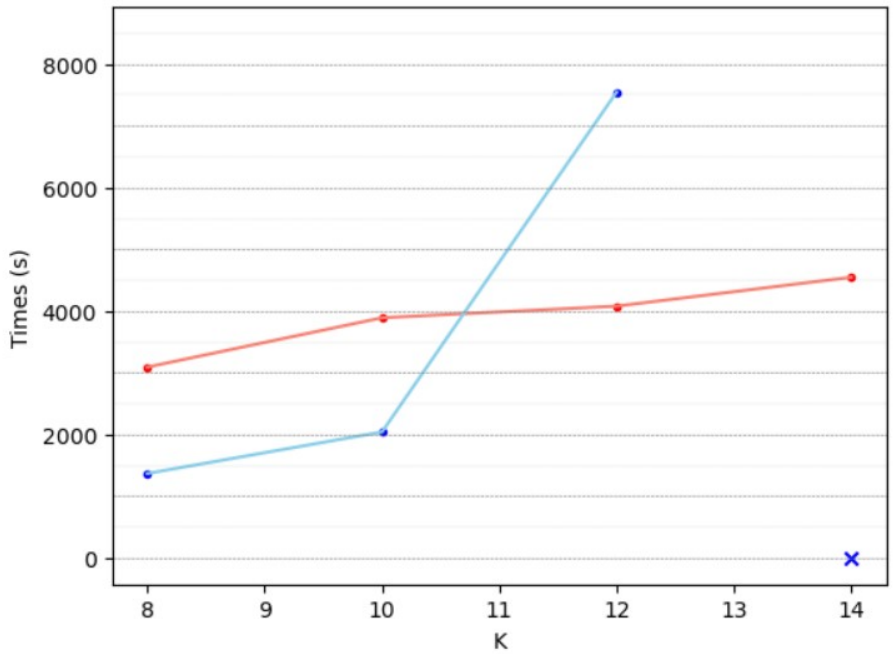


Figura 4.5: In rosso il tool di questa tesi, in blu *ComplexCGR*.
Tempo medio impiegato da entrambi i tools per produrre la rappresentazione FCGR delle sequenze del dataset *'Human Chromosomes'* per i vari valori di k.

Esattamente come per il test precedente, anche su sequenze più lunghe si può notare nei grafici come il tool di questo studio risulti essere più efficiente al crescere di k anche se, per valori di k sufficientemente piccoli ($k < 11$), risulta ancora meno performante del tool basato su *k-mer counter*.

Capitolo 5

Conclusione e lavori futuri

La libreria sviluppata per la rappresentazione in immagini delle sequenze genomiche si è dimostrata efficace nel calcolo e nella visualizzazione delle rappresentazioni FCGR, con notevoli miglioramenti in termini di velocità rispetto ad altri strumenti come *ComplexCGR*. L'approccio basato sulla mappatura geometrica diretta ha permesso di ridurre significativamente i tempi di esecuzione, specialmente su sequenze di grandi dimensioni e con valori di k elevati, dimostrando la sua efficienza in contesti dove la gestione della memoria è cruciale. I risultati ottenuti dai tests su dataset di coronavirus e cromosomi umani confermano la validità del metodo e mostrano come la libreria sia particolarmente adatta a lavorare su dataset genomici di varia complessità.

Tuttavia, sono emerse alcune limitazioni che aprono interessanti prospettive per i lavori futuri. In particolare, la libreria mostra prestazioni inferiori per analisi con valori di k bassi e per sequenze più brevi, dove strumenti basati su *k-mer counter* risultano ancora più rapidi. Inoltre, la libreria è al momento limitata al solo calcolo delle immagini FCGR per DNA e RNA, senza supportare direttamente altre applicazioni bioinformatiche.

Per estendere ulteriormente le funzionalità della libreria, è possibile implementare le seguenti migliorie:

1. **Aggiunta dell'opzione per la scelta tra valore di minimo esatto e approssimato**

   L'attuale implementazione presuppone che il valore minimo nella matrice FCGR sia sempre zero. Tuttavia, sarebbe utile fornire agli utenti la possibilità di scegliere se calcolare esattamente il minimo della matrice, per avere un maggiore controllo sulla rappresentazione grafica.

2. **Aggiunta del calcolo CGR come output**

   Oltre a FCGR, sarebbe interessante aggiungere la possibilità di fornire la scelta per calcolare e visualizzare anche la *Chaos Game Representation* (CGR) pura.

3. **Opzioni per il rescaling dell'immagine finale**

   Attualmente, l'immagine generata dalla matrice FCGR dipende esclusivamente dal valore di k scelto, ma sarebbe utile aggiungere delle opzioni per permetterne il ridimensionamento con interpolazione, in modo da garantire una maggiore flessibilità nella visualizzazione. Questa modifica risulta cruciale se le immagini prodotte dal tool vengono usate come input per un modello di *Machine Learning*.

4. **Estensione dell'alfabeto per l'analisi di altre sequenze biologiche**

   Al momento, la libreria supporta solo DNA e RNA, ma l'estensione del metodo per includere sequenze proteiche (con un alfabeto più ampio) sarebbe un miglioramento significativo.

Con questi miglioramenti, la libreria potrebbe diventare uno strumento ancora più versatile, ampliando le sue potenziali applicazioni nel campo della bioinformatica e offrendo nuove opportunità di analisi.

Bibliografia

[1] H. F. Löchel e D. Heider, «Chaos game representation and its applications in bioinformatics,» *Computational and Structural Biotechnology Journal*, vol. 19, pp. 6263–6271, 2021, issn: 2001-0370. doi: https://doi.org/10.1016/j.csbj.2021.11.008. indirizzo: https://www.sciencedirect.com/science/article/pii/S2001037021004736.

[2] Barnsley, Michael F., *Fractals Everywhere: New Edition.* Dover Publications, 2012.

[3] H. Jeffrey, «Chaos game representation of gene structure,» *Nucleic Acids Research*, vol. 18, n. 8, pp. 2163–2170, apr. 1990, issn: 0305-1048. doi: 10.1093/nar/18.8.2163. eprint: https://academic.oup.com/nar/article-pdf/18/8/2163/7059915/18-8-2163.pdf. indirizzo: https://doi.org/10.1093/nar/18.8.2163.

[4] P. K. Burma, A. Raj, J. K. Deb e S. K. Brahmachari, «Genome analysis: A new approach for visualization of sequence organization in genomes,» *Journal of Biosciences*, vol. 17, n. 4, pp. 395–411, 1992, issn: 0973-7138. doi: 10.1007/BF02720095. indirizzo: https://doi.org/10.1007/BF02720095.

[5] Python, *Math*, https://docs.python.org/3/library/math.html, 2024.

[6] N. Team, *Numpy*, https://numpy.org/, 2024.

[7] P. Team, *Pillow*, https://pypi.org/project/pillow/, 2024.

[8] B. Chor, D. Horn, N. Goldman, Y. Levy e T. Massingham, «Genomic DNA k-mer spectra: models and modalities,» *Genome Biology*, vol. 10, n. 10, R108, 2009, issn: 1474-760X. doi: 10.1186/gb-2009-10-10-r108. indirizzo: https://doi.org/10.1186/gb-2009-10-10-r108.

[9] M. Kokot, M. Długosz e S. Deorowicz, «KMC 3: counting and manipulating k-mer statistics,» *Bioinformatics*, vol. 33, n. 17, pp. 2759–2761, mag. 2017, issn: 1367-4803. doi: 10.1093/bioinformatics/btx304. eprint: https://academic.oup.com/bioinformatics/article-pdf/33/17/2759/49040995/bioinformatics_33_17_2759.pdf. indirizzo: https://doi.org/10.1093/bioinformatics/btx304.