



Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Sviluppo di una libreria Python per la gestione di grafi del pangenoma

Relatore: Prof. Gianluca Della Vedova

Co-relatore: Dott. Simone Ciccolella

Relazione della prova finale di:

Davide Pizzo

Matricola 869254

Anno Accademico 2022/2023

*A me stesso,
sperando che questo sia il primo di tanti traguardi*

Indice

Introduzione	1
1 Introduzione teorica	3
1.1 Teoria dei grafi	3
1.2 Assemblaggio Genomico e Grafi Associati	5
1.3 Il formato GFA	7
2 Studio e Implementazione	11
2.1 Test con networkX	11
2.2 Analisi codice preesistente	12
2.3 Implementazione funzionalità	14
2.4 Gfatk	26
3 Analisi sperimentale	27
3.1 Caricamento grafo	27
3.2 Funzionalità implementate	29
4 Possibili sviluppi futuri	37
5 Conclusioni	39
Bibliografia	40
Ringraziamenti	41

Introduzione

Con la crescente avanzata nello sviluppo degli algoritmi nel campo pangenomico, il formato GFA sta rapidamente acquisendo importanza. In questa prospettiva, il lavoro che ho svolto si focalizza sull'espansione e modifica della libreria Python **pygfa**, mirando a migliorarne la funzionalità ed implementarne di nuove.

Inoltre, ho sviluppato un software di utility in parallelo all'estensione della libreria. Questo tool, denominato **gfatk**, è progettato per fornire funzionalità della libreria, direttamente da linea di comando. Questo software si propone di semplificare l'utilizzo della libreria e di fornire agli utenti uno strumento pratico e flessibile. Il suo funzionamento sarà spiegato più avanti.

Per raggiungere gli obiettivi prefissati, il lavoro è stato suddiviso in diverse fasi chiave. All'inizio ho condotto una fase di test mirata a valutare la capacità, in termini di memoria, di networkX (una libreria Python ampiamente utilizzata per la manipolazione di grafi) nello gestire grafi di dimensioni rilevanti per il contesto trattato, ovvero quello pangenomico.

Questi test li ho disegnati in autonomia, ovviamente con l'appoggio del Relatore e del Co-Relatore, con i quali ho discusso la progettazione dei test.

Tutto ciò è stato fatto per garantire che la libreria scelta potesse gestire grafi complessi e di grandi dimensioni.

Successivamente ho proceduto con una fase di reverse engineering, finalizzata a comprendere a fondo il codice già esistente della libreria **pygfa**. Ho adottato questo approccio per acquisire una conoscenza approfondita del come funzionasse la libreria nel dettaglio, preparando così il terreno per le nuove funzioni che da lì a poco tempo avrei dovuto implementare.

Questa fase in particolare è stata molto importante non solo per ciò che ho fatto in questo contesto, ma anche a livello di crescita di competenze personali.

La fase di implementazione è stata il cuore del lavoro che ho svolto, durante la quale ho aggiunto le nuove funzionalità alla libreria **pygfa**. Questa fase ha richiesto una combinazione di competenze di programmazione, comprensione approfondita delle specifiche del formato GFA e un'attenta integrazione con le funzionalità esistenti della libreria.

Infine ho svolto una fase di testing sulle nuove funzioni che ho implementato. Questi test, che sono stati disegnati volta per volta con Relatore e Co-Relatore, sono progettati per monitorare il tempo di esecuzione di ogni funzione implementata e il consumo di memoria RAM. Il motivo di questa fase di testing è il garantire che le nuove funzionalità che mi sono impegnato a implementare non solo funzionassero, ma che fossero anche efficienti e stabili se usate in grafi di grandi dimensioni, come appunto i grafi del pangenoma.

Capitolo 1

Introduzione teorica

1.1 Teoria dei grafi

La teoria dei grafi rappresenta un ramo fondamentale dell'informatica, offrendo un approccio astratto per la modellazione e l'analisi di relazioni tra entità. Questo sottocapitolo si propone di esplorare i concetti chiave della teoria dei grafi [4], evidenziando le definizioni di base, i tipi di grafi e le loro applicazioni.

1.1.1 Definizioni di base

- **Grafo:** Un grafo è una struttura composta da un insieme di nodi (o vertici) e un insieme di archi che collegano coppie di nodi.
Formalmente è definito come $G=(V, E)$, dove 'V' è l'insieme dei nodi e 'E' è l'insieme degli archi
- **Nodo:** Rappresenta un'entità all'interno del grafo, spesso associato a oggetti o concetti. I nodi sono gli elementi fondamentali che costituiscono la struttura di un grafo.
Due nodi connessi da un arco sono detti adiacenti.
- **Arco:** Un arco è una connessione tra due nodi e può essere diretto (se ha una direzione) o non diretto. Gli archi rappresentano le relazioni tra le entità del grafo.

1.1.2 Altri concetti importanti

Vediamo ora altri concetti e definizioni che risulteranno importanti per comprendere il lavoro svolto.

- **Cammino:** Un cammino in un grafo è una sequenza ordinata di nodi in cui ciascuna coppia di nodi consecutivi è collegata da un arco. In altre parole, è un percorso che attraversa i nodi del grafo, seguendo gli archi da un nodo all'altro in modo sequenziale.

Rappresenta un ottimo modo di navigare all'interno del grafo ed evidenziare le connessioni tra i nodi.

- Vicinato di un nodo: Rappresenta l'insieme dei nodi direttamente collegati a uno specifico nodo attraverso gli archi; è quindi la lista dei nodi adiacenti del nodo specificato.
- Densità di un grafo: E' una misura relativa al rapporto tra numero di nodi e archi, compresa tra 0 e 1.
Questa misura consente di capire quanto il grafo sia "riempito" di connessioni; quindi se la densità vale 1 vuol dire che il grafo ha tutte le connessioni possibili(ogni coppia di nodi è connessa), mentre se vale 0 vuol dire che il grafo è completamente disperso(nessuna coppia di nodi connessa)
- Intorno di un nodo: L'intorno di un nodo in un grafo rappresenta il sottoinsieme dei nodi del grafo che sono raggiungibili dal nodo specificato attraverso 'x' transizioni, con 'x' lunghezza dell'intorno.
Ciò vuol dire che un intorno di lunghezza 1 corrisponde al vicinato di un nodo.
- Sottografo: Un sottografo è un sottoinsieme di nodi e archi di un grafo più grande da cui eredita la sua struttura.
Può essere definito formalmente:
 $H=(V', E')$ con H sottografo di $G=(V, E)$, V' sottoinsieme di V , E' sottoinsieme di E

1.1.3 Tipologie di grafi

Ci sono 2 principali tipologie di grafi:

- Grafo Orientato: Un grafo orientato è un grafo dove ogni arco ha una direzione specifica.
Ogni arco è indicato con una coppia specifica di nodi che rappresentano la sorgente e la destinazione dell'arco.
L'insieme 'E' sarà dunque composto da tuple (u, v) con 'u' nodo di partenza e 'v' nodo di arrivo.
- Grafo Non Orientato: Un grafo non orientato è un grafo dove gli archi collegano i nodi senza una direzione specifica. Ogni connessione tra 2 nodi è quindi simmetrica.

1.2 Assemblaggio Genomico e Grafi Associati

1.2.1 Assemblaggio di genomi

L'assemblaggio di genomi costituisce un processo cruciale nella bioinformatica, mirando alla ricostruzione della sequenza originale del DNA mediante l'utilizzo esclusivo delle "reads" [9]. Le reads rappresentano frammenti di genoma di lunghezza variabile tra le 50 e le 1000 basi nucleotidiche. Sebbene suddividere un genoma in reads renderebbe impossibile un assemblaggio diretto, si sfrutta l'osservazione che sequenze di DNA simili compaiono ripetutamente all'interno di un campione genetico. [11]

Il principio chiave nell'assemblaggio risiede nell'identificazione degli "overlap" tra le diverse reads. Un'overlap si verifica quando una sequenza di DNA presente in una read è condivisa da un'altra read, creando così una sovrapposizione. Questi tratti in comune forniscono le connessioni necessarie per ricostruire le sequenze originali.

Il cuore del problema nell'assemblaggio di genomi è la progettazione di algoritmi efficienti per condurre questo riassemblaggio. L'analisi degli overlap diventa essenziale per comprendere le relazioni tra le reads e ricostruire accuratamente il genoma originale.

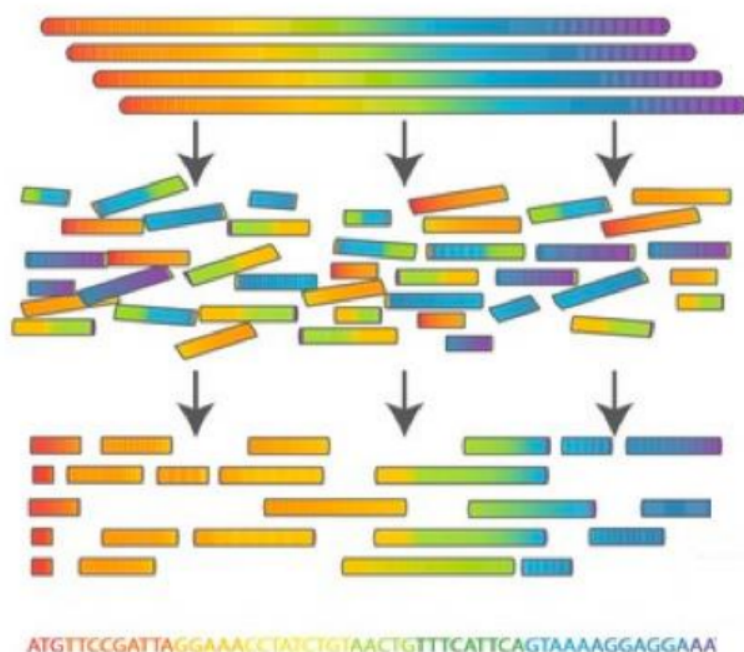


Figura 1.1: Come funziona nella pratica il processo di assemblaggio

L'assemblaggio genomico può avere dei punti critici:

- Errori di sequenziamento: Si possono verificare nella fase di sequenziamento del DNA e possono influire negativamente sulla qualità delle basi sequenziate.
- Coverage non omogeneo: La copertura (coverage) è la misura del numero di sequenze che sovrappongono una data regione del genoma. Una copertura non omogenea può portare a regioni del genoma che sono sottorappresentate o sovrarappresentate nell'assemblaggio.
- Elementi ripetuti: Le sequenze ripetute nel genoma possono creare problemi durante l'assemblaggio come duplicazioni o omissioni di queste regioni.

1.2.2 Grafi di assemblaggio

I grafi di assemblaggio sono uno strumento concettuale fondamentale nel campo della ricostruzione dei genomi [7]. Forniscono una rappresentazione strutturata delle relazioni tra le reads genomiche, in modo che sia facilitata la comprensione degli overlap e delle connessioni necessarie per ricostruire la sequenza completa del genoma.

Vediamo ora 2 tipi di grafi di assemblaggio[12]:

- Grafi di Overlap: Sono grafi che nascono dalla sovrapposizione diretta tra le reads. Un nodo rappresenta una read e un arco, invece, un'overlap tra due reads di una lunghezza maggiore di una misura prefissata. [13] Queste sovrapposizioni sono sfruttate da algoritmi come OLC(Overlap-Layout-Consensus) per gestire le reads e ricostruire il genoma. Infatti una volta costruito l'intero grafo, è possibile trovare il genoma originario individuando un percorso che passi per tutti i nodi una e una sola volta, ovvero un cammino hamiltoniano. [3]
- Grafi di De Bruijn: Questa tipologia di grafo rappresenta un'alternativa, dove le reads sono divise in k-meri, delle sequenze brevi di lunghezza fissa. Ogni vertice è un (k-1)-mero; ogni vertice p un k-mero. Il grafo incorpora tutte le combinazioni di k-meri, offrendo un'indicazione delle possibili connessioni tra le sequenze. Generalizzando il concetto dei grafi di De Bruijn: dato un certo alfabeto, si identifica una sequenza ciclica di lettere per cui ogni possibile parola di lunghezza k compaia come combinazione di caratteri una volta soltanto.[2]

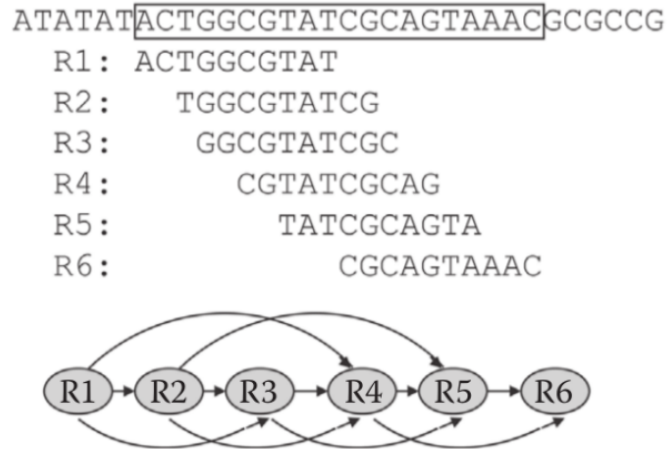


Figura 1.2: Esempio di grafo di Overlap, basato sulle sovrapposizioni tra reads

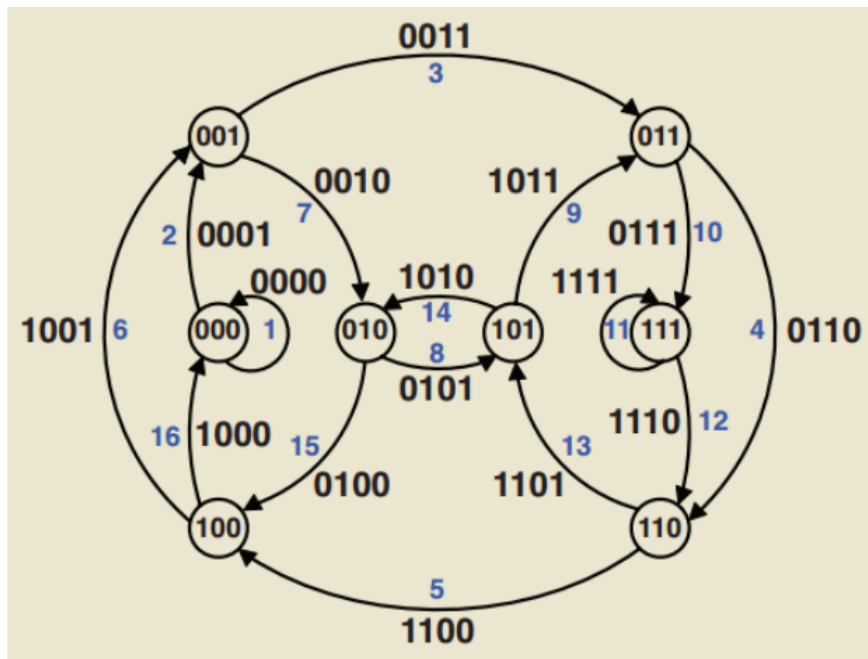


Figura 1.3: Esempio di grafo di De Bruijn con $k=4$ e alfabeto= $\{0,1\}$

1.3 Il formato GFA

Il formato GFA (Graphical Fragment Assembly) rappresenta uno standard di file utilizzato per descrivere grafi di assemblaggio genomico. Questa sottosezione si propone di esplorare nel dettaglio la struttura e le caratteristiche fondamentali del formato GFA, focalizzandosi sulla sua importanza nel contesto dell'assemblaggio genomico e sulla gestione di grafi del pangenoma. In particolare verrà affrontato il GFA versione 1. [5]

1.3.1 Terminologia

Vediamo ora un veloce elenco di termini la cui assimilazione risulta fondamentale per la comprensione del formato GFA. [1]

- Segment: Una sequenza o sottosequenza continua. Nei nostri grafi saranno i nodi.
- Link: Un overlap tra due segments. Ogni collegamento va dalla fine di un segmento all'inizio di un altro segmento. Ogni collegamento memorizza l'orientamento di ogni segment e il numero di basi sovrapposte. Nei grafi che vedremo saranno gli archi.
- Containment: Un overlap tra due segmenti dove uno è contenuto in un altro. Anche questi saranno degli archi.
- Path: Un elenco di segmenti orientati, dove ogni coppia consecutiva di essi è collegata da un link o un containment.

Nota n.1 Rispettivamente dalla versione 1.1 e 1.2 ci sono anche i Walk e i Jump, ma in questo caso non ci interessa affrontarli

1.3.2 Struttura delle linee

Ogni riga in GFA ha i campi delimitati da tabulazioni(/t) e il primo campo definisce il tipo di linea. Avremo quindi: le H-linee per l'header a inizio file; le S-linee per rappresentare i segment; le L-linee per rappresentare i link; le C-linee per rappresentare i containment; le P-linee per rappresentare i path e # per commentare. Ogni tipo di linea avrà dei campi obbligatori e dei campi facoltativi.

- H-linee: L'unico campo obbligatorio è il 'RecordType'(cioè il 1° campo che dice la tipologia di riga), mentre è invece opzionale il numero della versione(VN)
- S-linee: I campi obbligatori sono il 'RecordType'(S) e il 'Name' del segment. Il campo della sequenza(sequence) è opzionale, infatti può valere *, cioè non specificata.
Tra alcuni dei campi opzionali figurano la lunghezza del segment(LN), il conteggio delle reads(RC) e altri meno importanti.
- L-linee: I campi obbligatori e più importanti sono il 'RecordType'(L), 'From'(indica il nome del segment da qui parte il link), 'FromOrient'

(+|-), con + si intende la sequenza normale, con - si intende il reverse complement della sequenza), 'To' (il nome del segment a cui arriva il link) 'ToOrient'(+|-). Il campo 'Overlap' è facoltativo e indica la lunghezza della sovrapposizione.

- C-linee: La struttura è simile alle L-linee, dove il primo segment è il 'Container' (quindi chi contiene), mentre il secondo segment è il 'Contained' (il contenuto). In aggiunta ha il campo 'pos' che indica da che posizione il 'Contained' è contenuto nel 'Container'.
- P-linee: I campi da citare sono il 'RecordType'(P), il 'PathName' che è il nome univoco del path, il 'SegmentNames' che è un elenco di nomi e orientamenti di segments nel path separati da una virgola, e infine il campo 'Overlap' anche qua opzionale (se presente deve avere un valore in meno rispetto al numero di segmenti).

Listing 1.1: Esempio di un file GFA

```
H VN:Z:1.0
S 11 ACCTT
S 12 TCAAGG
S 13 CTTGATT
L 11 + 12 - 4M
L 12 - 13 + 3M
L 11 + 13 + 3M
P 14 11+,12-,13+ 4M,5M
```

Listing 1.2: Path 14 risultante dall'esempio precedente

```
11 ACCTT
12 CCTTGA
13 CTTGATT
14 ACCTTGATT
```

In 1.1 soprastante abbiamo un esempio di file GFA (come detto all'inizio della sezione, gli spazi bianchi sono tabulazioni \t).

La prima riga indica l'intestazione del file e la sua versione (1.0) Dalla linea 2 alla linea 4 vengono definiti 3 segments con identificatori , rispettivamente, 11,12 e 13 e a tutti e tre viene assegnata una sequenza nucleotidica.

Le linee dalla 5 alla 7 rappresentano degli archi (più precisamente dei link). Esaminiamo nel dettaglio una di queste righe, per esempio la riga 5.

In questa riga vediamo che il link è tra il segment con id=11 e con id=12, per quanto riguarda gli orientamenti il segment con id=11 ha un '+', l'altro invece ha un '-'. Ciò vuol dire che l'overlap sarà tra la sequenza del segment 11 e il reverse complement della sequenza del segment 12 (cioè il reverse della sequenza sostituendo le A con le T e viceversa, le C con le G e viceversa). Infine abbiamo anche un'informazione sull'overlap, ovvero 4 basi. L'overlap calcolato quindi è tra ACCTT e CCTTGA. Come si può notare le ultime 4 basi della prima sequenza coincidono con le prime 4 basi della seconda sequenza, generando così overlap. L'ultima linea invece è una linea rappresentante un Path con id=14. I segments che fanno parte di questo path sono i segment con id=11,12 e 13 con gli orientamenti mostrati in figura. Infine ci sono le quantità di basi sovrapposte per ogni coppia di segment.

In 2.1 è possibile visualizzare il risultato del path. Quindi come descritto dalla linea abbiamo che tra 11+ e 12- c'è un'overlap di 4 basi, invece tra 12- e 13+ un'overlap di 5 basi.

Capitolo 2

Studio e Implementazione

Il presente capitolo rappresenta il fulcro del lavoro svolto. La sua struttura si articolerà in diverse sezioni, ciascuna focalizzata su un aspetto cruciale del percorso di ricerca e implementazione.

Il primo sottocapitolo si concentrerà sui test iniziali mirati a valutare le dimensioni di grafi gestibili attraverso l'uso di NetworkX; successivamente, verrà esplorato il processo di reverse engineering effettuato in fase iniziale di lavoro. Il sottocapitolo più importante sarà quello dedicato alle funzioni implementate, in particolare qua verranno mostrati codici scritti, le motivazioni che hanno portato all'implementazione di una determinata funzione ecc...

La sezione finale di questo capitolo sarà dedicata a una breve panoramica sul software di utility **gfatk**.

2.1 Test con networkX

Prima di parlare dei test effettuati, vediamo prima un'introduzione generale di networkX.

NetworkX è una libreria Python specializzata nella manipolazione e nell'analisi di strutture di rete e grafi. Offre un'ampia gamma di funzionalità per la creazione, l'analisi e la visualizzazione di grafi, rendendola una scelta ideale per il nostro lavoro.

Per comprendere le prestazioni di NetworkX nelle condizioni specifiche del nostro progetto, abbiamo condotto una serie di test utilizzando istanze di MultiGraph [8] (i grafi GFA che useremo, saranno appunto dei MultiGraph di networkX). Questi test sono stati progettati per valutare il comportamento della libreria in termini di consumo di memoria al variare delle dimensioni dei grafi (in particolare il numero dei nodi).

Per quanto riguarda i test sono stati generati grafi di dimensioni crescenti, aumentando progressivamente il numero di nodi e archi, riflettendo così le sfide potenziali associate alla gestione di grafi pangenomici sempre più complessi. Ogni test è stato ripetuto più volte per garantire la coerenza dei risultati.

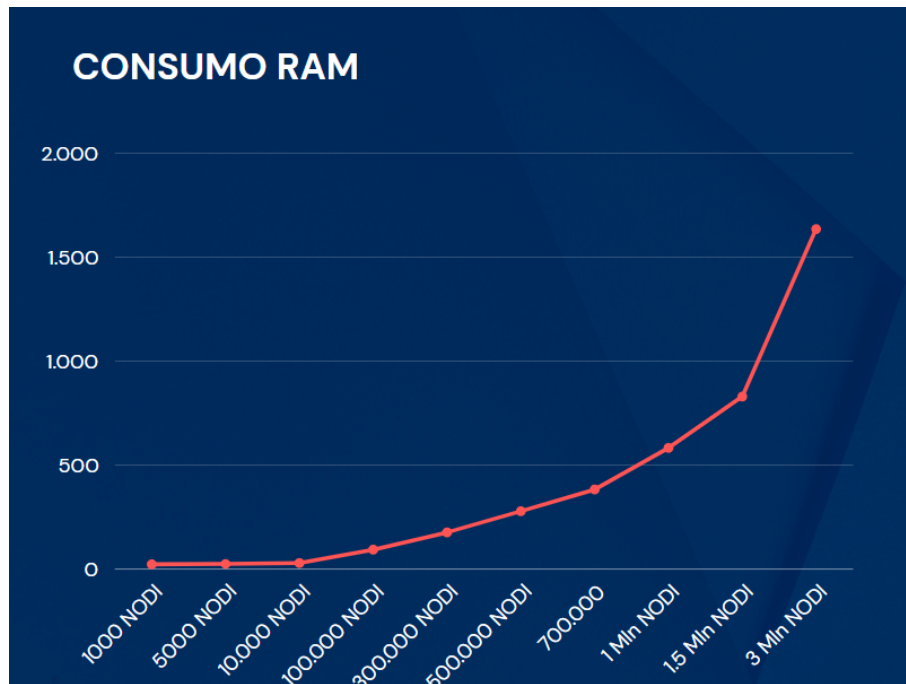


Figura 2.1: Grafico relativo al consumo di memoria

Il grafico 2.1, che ha come asse delle ordinate il consumo in MB e come asse delle ascisse il numero di nodi, illustra chiaramente la tendenza di crescita all'aumentare del numero di nodi (e quindi anche all'aumentare del numero di archi), come era prevedibile all'inizio della fase di testing.

2.2 Analisi codice preesistente

La fase di 'analisi del codice preesistente', comunemente conosciuta come debugging, assume un ruolo cruciale nell'intero percorso di sviluppo del lavoro, contribuendo in modo significativo anche al percorso personale dell'autore. Questa tappa si è rivelata fondamentale per garantire una comprensione completa e accurata della libreria **pygfa** in ogni suo dettaglio. L'approfondita analisi del codice esistente è stata il punto di partenza, consentendo di gettare le basi per l'implementazione successiva di nuove funzionalità in modo coeso e coerente con il funzionamento preesistente della libreria.

Dopo le iniziali osservazioni del codice sorgente, si è avviata una fase di debugging mirata. Questo processo è stato essenziale per acquisire una comprensione approfondita di come ogni componente della libreria interagisse e funzionasse. Un esempio concreto di questa fase è emerso durante il tentativo di creare un nuovo 'Segment' (S-linea), dove l'obiettivo principale era comprendere i passaggi necessari per aggiungere efficacemente il Segment al file .gfa. L'approccio al debugging è stato metodico, coinvolgendo test appro-

fonditi e variazioni controllate per esplorare il comportamento della libreria in diversi scenari.

La fase di analisi e debugging ha fornito non solo un'opportunità per migliorare la comprensione tecnica della libreria, ma ha anche svolto un ruolo cruciale nella crescita personale dell'autore, contribuendo a sviluppare competenze nel troubleshooting e nella risoluzione di problemi complessi. L'esperienza acquisita durante questa fase è stata preziosa nel plasmare il percorso di sviluppo successivo, influenzando le scelte di progettazione e le implementazioni delle nuove funzionalità.

2.3 Implementazione funzionalità

La parte focale del lavoro svolto risiede nell'implementazione delle funzionalità all'interno della libreria **pygfa**. Questo sottocapitolo si propone di condurre il lettore attraverso le diverse fasi di sviluppo, mettendo in luce le funzionalità chiave implementate, le modifiche apportate e la logica che ha guidato ogni passo del processo.

2.3.1 Gestione path (P-linee)

Uno dei cambiamenti fondamentali apportati alla libreria riguarda la gestione dei path (P-linee), precedentemente trattati come sottografi GFA e ora implementati come liste di nodi, in particolare ogni grafo ha una sua 'PATHLIST' che è una lista di sottoliste dei nodi di un path.

Ovviamente i path vengono inseriti sempre con la stessa tipologia di stringa come in precedenza.

La decisione di cambiare la gestione dei path è stata guidata da due considerazioni principali: il consumo di memoria, che risulta minore (anche se la differenza non è enorme) e soprattutto l'usabilità dal lato utente e la semplicità dal lato sviluppatore.

Come si può osservare dal codice 2.1 alla pagina successiva, sono stati implementati anche 3 metodi molto semplici che servono a restituire all'utente i paths. In particolare **paths** restituisce le sottoliste dei paths; **id_paths** serve per poter accedere agli id dei cammini e infine **get_paths_with_id** restituisce all'utente tutti i cammini affiancati dal corrispettivo id.

Si è affrontata per prima questa modifica, perchè tutte le implementazioni successive fanno riferimento a questa implementazione della gestione dei path.

Listing 2.1: Metodi di "accesso ai path"

```
def paths(self):
    if len(self.PATHLIST)==0:
        raise ValueError("Graph_has_no_paths")
    segmentsPath = [[tupla[0] for tupla in path] for
        path in self.PATHLIST]
    return segmentsPath

def id_paths(self):
    if len(self.idPaths)==0:
        raise ValueError("Graph_has_no_paths")
    return self.idPaths

def get_paths_with_id(self):
    dict={}
    for id_path, path in zip(self.idPaths, [(tupla[0],
        tupla[1]) for tupla in path] for path in self.
        PATHLIST):
        dict[id_path] = path
    return dict
```

2.3.2 Concat_path_sequences

La funzione `concat_path_sequences` rappresenta il punto di partenza nell'implementazione della libreria, essendo la prima ad essere sviluppata. Nonostante la sua semplicità rispetto alle funzioni successive, il suo ruolo è cruciale nel contesto dell'analisi e manipolazione delle sequenze di Segment su specifici path all'interno del grafo del pangenoma.

L'obiettivo principale della funzione è la restituzione della concatenazione delle sequenze dei Segment presenti su un path identificato da un `path_id` specifico passato come parametro. La funzione gestisce, inoltre, l'orientamento dei Segment. Nel caso in cui l'orientamento di un Segment sia negativo, la sequenza viene sottoposta a un processo di reverse e complement grazie a una funzione di supporto (`reverse_complement`) dedicata, come si può notare dal codice 2.2.

Più nel dettaglio, la funzione accede alle informazioni del grafo per ottenere le sequenze dei Segment associati ai nodi presenti sul path specificato. Se l'orientamento di un Segment è negativo, viene applicato il reverse and complement della sequenza attraverso la funzione ausiliaria `reverse_complement`. La concatenazione delle sequenze dei Segment restituisce la sequenza completa del path.

Listing 2.2: L'implementazione delle 2 funzioni citate sopra

```
def reverse_complement(self , sequence):
    complement = { 'A' : 'T', 'T' : 'A', 'C' : 'G', 'G'
                  : 'C'}
    reverse_comp_seq = ''.join(complement[nucleotide]
                               for nucleotide in reversed(sequence))
    return reverse_comp_seq

def concat_path_sequences(self , path_id):
    path=self.get_paths_with_id()[path_id]
    sequence=""
    for segment in path:
        segmentId = segment[0]
        segmentOrn = segment[1]
        node=self.get(segmentId)
        if segmentOrn == '+':
            sequence = sequence + node['sequence']
        elif segmentOrn == '-':
            segmentSequence = node['sequence']
            sequence = sequence + self.
                reverse_complement(segmentSequence)
    return sequence
```

2.3.3 Remove_node_and_merge

Si osserva ora la funzione `Remove_node_and_merge`, la quale è progettata per eliminare un determinato nodo, se presente nel grafo, dato il suo id e successivamente creare uno o più archi di collegamento tra i nodi adiacenti, come mostrato in figura 2.2.

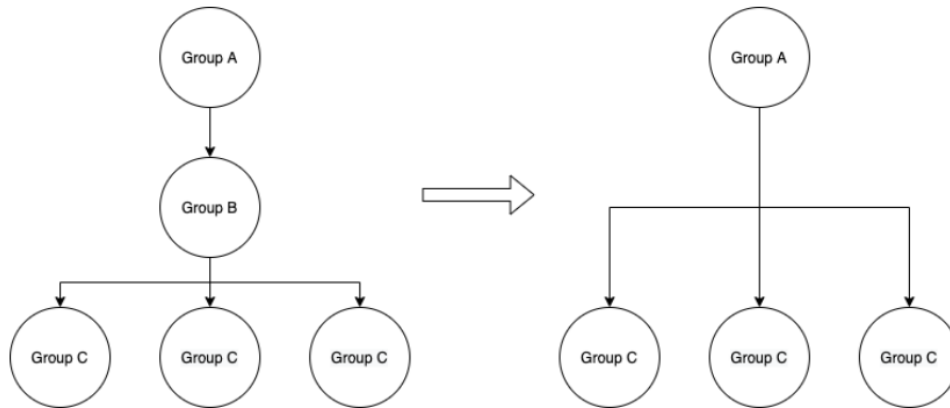


Figura 2.2: La logica su cui si basa la funzione

La funzione si appoggia sulla `remove_node` già presente all'interno della libreria, la quale elimina un nodo dato il suo id e anche gli archi a lui adiacenti; come si può ben intuire questa funzione verrà richiamata alla fine del procedimento, quando saranno stati aggiunti tutti gli archi di collegamento.

Dopo una prima grezza versione e dopo un paio di ragionamenti fatti col gruppo di lavoro, si è arrivati alla conclusione di implementare la funzione con due modalità diverse. Nella modalità 1(2.3) vengono creati gli archi di collegamento tra tutti i predecessori e tutti i successori. La modalità 2(2.4) si basa invece sui path (P-linee) e sulla PATHLIST del grafo; in pratica viene aggiunto un arco di collegamento tra il nodo A e il nodo B se questi sono nello stesso path divisi dal nodo che si sta eliminando.

Listing 2.3: Codice della funzione (modalità 1)

```
def remove_node_and_merge(self, nid, mod):
    if mod == 1 and mod != 2:
        raise ValueError("mod_must_be_1_or_2")
    if nid not in self.nodes():
        raise ValueError(f"Node_with_ID_{nid}_not_found_in_the_graph")
    if len(self.neighbors(nid)) == 0:
        print(f"Node_with_ID_{nid}_doesn't_have_adjacent_edges")
    else:
        edges = self.edges()
        if mod == 1:
            nidEdgesList = []
            for e in edges:
                n1 = e[0]
                n2 = e[1]
                if n1 == nid or n2 == nid:
                    nidEdgesList.append(e)
            for i in range(len(nidEdgesList)):
                edge = nidEdgesList[i]
                if edge[0] != nid: fromPos = edge[0]
                else: fromPos = edge[1]
                for j in range(len(nidEdgesList)):
                    if i != j:
                        edge1 = nidEdgesList[j]
                        if edge1[0] != nid: toPos = edge1[0]
                        else: toPos = edge1[1]
                        if (fromPos, toPos) not in edges and (toPos, fromPos) not in edges:
                            edges.append((fromPos, toPos))
            self.from_string("L" +
                             "\t" + fromPos + "\t" +
                             "\t" + toPos + "\t+\t0M")
```

Listing 2.4: Codice della funzione (modalità 2)

```
elif mod == 2:
    for path in self.PATHLIST:
        cont=0
        for tuple in path:
            if nid == tuple[0]:
                nid_index = cont
                if nid_index > 0:
                    predecessor = path[cont-1]
                    nid_predecessor = predecessor[0]
                else:
                    nid_predecessor = None
            if nid_index < len(path)-1:
                successor = path[cont+1]
                nid_successor = successor[0]
            else:
                nid_successo = None
            if nid_predecessor != None and
               nid_successor != None:
                if (nid_predecessor , nid_successor
                   ) not in edges and (
                   nid_successor , nid_predecessor)
                   not in edges:
                    self.from_string("L\t" +
                                       nid_predecessor + "\t+\t" +
                                       nid_succesor + "\t+\t0M")
        cont = cont + 1
self.remove_node(nid)
```

2.3.4 `Get_Subgraph_from_neighborhood`

La funzione `get_Subgraph_from_neighborhood` si propone di creare un sottografo in base all'intorno (vedi 1.1.2) di un nodo passando il suo id come parametro della funzione. Anche la lunghezza dell'intorno viene passata come parametro.

La funzione restituisce in output un oggetto GFA rappresentante il sottografo, ciò per 2 motivi principali:

- Coerenza con le funzioni già implementate: infatti nella libreria ci sono funzioni pre-implementate come `get_subgraph` che dovrebbero restituire un sottografo; ciò che viene restituito in realtà è un oggetto GFA che rappresenta il sottografo.
- Migliore usabilità lato utente: infatti restituendo un oggetto GFA l'utente potrà compiere "azioni" su di esso o anche banalmente poter stampare i nodi e gli archi.

Come si vedrà nel capitolo 3, le dimensioni del sottografo saranno molto sensibili alla lunghezza dell'intorno.

Listing 2.5: Implementazione della funzione descritta

```
def get_Subgraph_from_neighborhood(self , nid , leng):
    sg = GFA()
    if leng < 1:
        raise ValueError("len_value_must_be_at_least_1")
    nodes = [nid]
    edges = []
    nodesTemp = []
    for i in range(0 , leng):
        nodes.extend(nodesTemp)
        nodesTemp = []
        for node in nodes:
            mainNode = self.get(node)
            neighborList = self.get(node)
            for neighbor in neighborList:
                if not(neighbor in nodes):
                    currentNode = self.get(neighbor)
                    sg.add_node("S\t" + currentNode[ '
                        nid ' ] + "\t" + currentNode[ '
                            sequence ' ])
                    nodesTemp.append(currentNode[ 'nid '
                        ])
            if not(((node , neighbor) in edges)or(
                neighbor , node) in edges)and (not(
                    node == neighbor)):
                sg.add_edge("L\t" + mainNode[ 'nid ' ]
                    + "\t+\t" + currentNode[ 'nid ' ]
                    + "\t-\t0M")
                edges.append((node , neighbor))
    return sg
```

2.3.5 `Get_subgraph_between_nodes`

La funzione implementata più recentemente è denominata `get_subgraph_between_nodes`. Questa funzione richiede tre parametri in ingresso: un nodo sorgente, un nodo destinazione e un flag booleano denominato 'orientation'. Il flag 'orientation' determina se gli archi del grafo devono essere considerati con l'orientamento oppure no.

Lo scopo fondamentale di questa funzione è generare e restituire un oggetto GFA che rappresenta il sottografo ottenuto attraverso tutti i cammini dal nodo 'Source' al nodo 'Destination' (o viceversa nel caso in cui il flag 'orientation' sia impostato su False). È importante notare che i cammini su cui si basa la costruzione del sottografo sono quelli definiti dalle P-linee.

Inizialmente, l'approccio previsto era leggermente più complesso: il sottografo doveva basarsi sui cammini effettivi del grafo, dal nodo sorgente al nodo destinazione. Tuttavia, questa idea è stata accantonata principalmente a causa della sua inefficacia. La funzione funzionava correttamente su grafi di piccole dimensioni, ma risultava eccessivamente lenta quando applicata a grafi del pangenoma. Nel tentativo di ottimizzarla, data la scadenza imminente del termine del lavoro, si è preferito semplificarla come descritto sopra.

Listing 2.6: Codice di `get_Subgraph_between_nodes` (senza orientamento)

```
def get_Subgraph_between_nodes(self , source ,
destination , orientation):
    sg = GFA()
    paths = [[tupla[0] for tupla in path] for path in
        self.PATHLIST]
    if orientation == False:
        for path in paths:
            if ((path[0] == source and path[len(path)
-1] == destination))or((path[0] ==
destination and path[len(path)-1] ==
source)):
                for i in range(0 , len(path)):
                    nid = path[i]
                    node = self.get(str(nid))
                    if nid not in sg.nodes():
                        sg.add_node("S\t" + node['nid']
+ "\t" + node['sequence'])
                    if i+1 < len(path):
                        nid1 = path[i + 1]
                        node1 = self.get(str(nid1))
                        if(nid1 not in sg.nodes()):
                            sg.add_node("S\t" + node1['
nid'] + "\t" + node1['
sequence'])
                    if ((nid , nid1) not in sg.edges()
or (nid1 , nid) not in sg.edges
()):
                        if not(nid == nid1):
                            sg.add_edge("L\t" + node['
nid'] + "\t+\t" + node1[
'nid'] + "\t-\t0M")
```

Listing 2.7: Codice di `get_Subgraph_between_nodes` (con orientamento)

```
elif orientation == True:
    for path in paths:
        if ((path[0] == source and path[len(path) -
            1] == destination)):
            for i in range(0 , len(path)):
                nid = path[i]
                node = self.get(str(nid))
                if nid not in sg.nodes():
                    sg.add_node("S\t" + node['nid']
                        + "\t" + node['sequence'])
                if i + 1 < len(path):
                    nid1 = path[i + 1]
                    node1 = self.get(str(nid1))
                    if (nid1 not in sg.nodes()):
                        sg.add_node("S\t" + node1['
                            nid'] + "\t" + node1['
                                sequence'])
                if ((nid , nid1) not in sg.edges()
                    or (nid1 , nid) not in sg.edges
                        ()):
                    if not(nid == nid1):
                        sg.add_edge("L\t" +
                            node['nid'] + "\t+\t
                                " + node1['nid'] + "
                                    \t-\t0M")

return sg
```

2.4 Gfatk

In parallelo allo sviluppo della libreria **pygfa**, è stata creata un'applicazione di utility denominata **gfatk**. Questo software fornisce un'interfaccia da linea di comando per sfruttare le funzionalità di base della libreria in modo pratico e immediato. La sua implementazione rappresenta un passo significativo nell'obiettivo di rendere accessibili le potenzialità della libreria anche a utenti non esperti di programmazione.

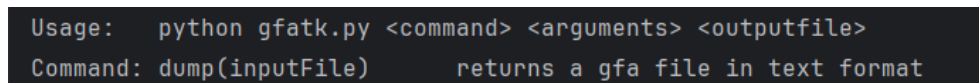
Gfatk.py è un programma "nuovo" rispetto alla libreria **pygfa** che aveva già del codice preesistente. Attualmente, il software supporta solo il comando "dump", il quale consente di visualizzare o salvare su file.txt il contenuto di un grafo in formato GFA.

Il suo design e il suo funzionamento trae ispirazione dal software **seqtk** [6] sviluppato però in linguaggio C.

Vediamo ora il funzionamento del software. In generale il funzionamento è molto semplice e intuitivo, inoltre vengono forniti degli output all'utente per indirizzarlo al corretto utilizzo.

Infatti se si prova a lanciare il programma (sia da console, sia da linea di comando) il risultato sarà quello mostrato in figura 2.3.

In particolare verrà stampato il modo di utilizzare lo strumento da linea di comando e i comandi disponibili con relativa descrizione.



```
Usage:  python gfatk.py <command> <arguments> <outputfile>
Command: dump(inputFile)      returns a gfa file in text format
```

Figura 2.3

Per quanto riguarda l'uso:

- **<command>**: è il nome del comando da usare.
- **<arguments>**: gli argomenti da passare (nella lista di comandi disponibili, per ogni comando, sono indicati i parametri).
- **<outputfile>**: nome del file di output, opzionale.

Ovviamente questo strumento è all'albore del suo sviluppo e in futuro sarà arricchito da nuovi comandi e nuove modifiche, ma questo discorso sarà affrontato nel capitolo apposito.

Capitolo 3

Analisi sperimentale

Questo capitolo si propone di mostrare al lettore le performance delle funzioni implementate alla libreria **pygfa**, attraverso una serie di test accuratamente progettati e svolti. In particolare si vogliono evidenziare le variazioni in termini di tempo [10] e di consumo della memoria.

L'obiettivo principale è condurre un'indagine approfondita sul comportamento delle varie implementazioni in relazione a varie proprietà.

In generale tutti i test sono stati costruiti tenendo conto delle proprietà che più potevano influire su una determinata funzione; queste proprietà a turno sono rimaste a un valore fisso per vedere l'incidenza che avevano.

3.1 Caricamento grafo

Prima di osservare i risultati ottenuti testando le nuove funzionalità di **pygfa**, è necessario ritagliare uno spazio al caricamento del grafo GFA da file. L'obiettivo è fornire una panoramica dettagliata delle variazioni temporali e dei requisiti di memoria in base alle caratteristiche dei grafi considerati.

Risultati attesi

In questo primo caso, quello che ci si aspetta è abbastanza banale, ovvero che il tempo di esecuzione e il consumo di memoria siano strettamente legati alle dimensioni del grafo (numero di nodi e numero di archi). Ci si attende, quindi, che grafi con un elevato numero di nodi e archi possano richiedere risorse computazionali più consistenti, rispetto a un grafo di dimensioni ridotte, comportando un aumento del tempo di caricamento e del consumo di memoria.

Risultati finali

Alla luce dei risultati di test ,mostrati nelle figure 3.1 e 3.2, si evidenzia il chiaro impatto del numero di nodi e archi sulla prestazione generale del caricamento. Infatti il tempo di esecuzione e il consumo di memoria RAM

mostrano una correlazione diretta con l'aumentare delle dimensioni del grafo.

In particolare, nodi e archi hanno all'incirca la stessa influenza, in quanto entrambi sono aggiunti con il medesimo metodo `from_string`

Un ultimo particolare da sottolineare è che il caricamento del grafo (sia in termini di tempo sia in termini di memoria) non viene influenzato dalla densità del grafo (come altri casi che vedremo tra qualche pagina). Ad esempio un grafo con 100 nodi 5000 archi (molto denso) impiegherà meno risorse rispetto a un grafo con 200.000 nodi e 5000 archi (poco denso).

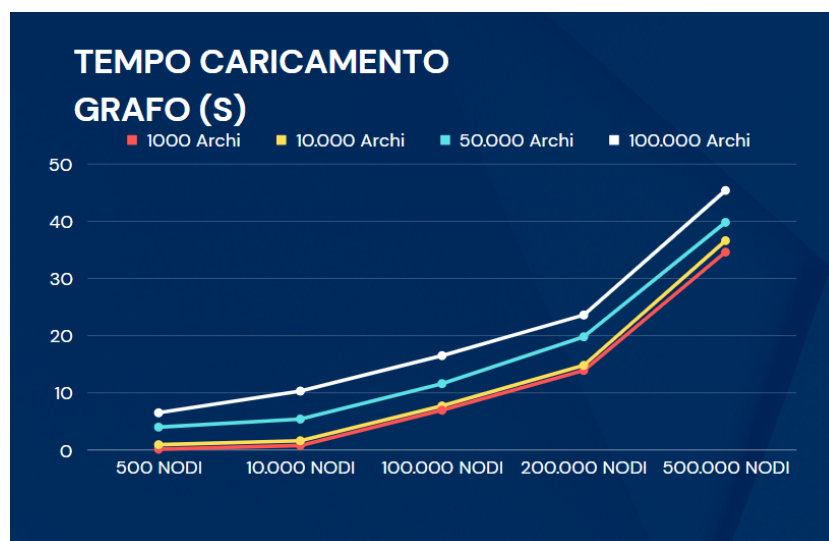


Figura 3.1

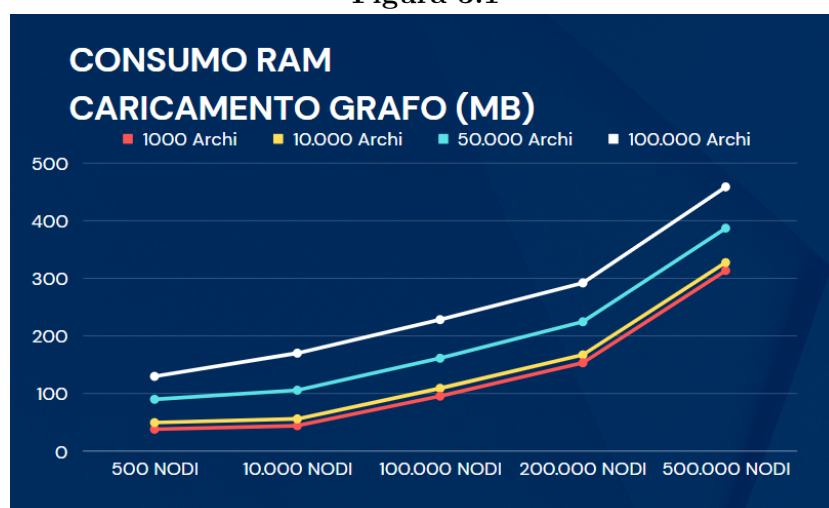


Figura 3.2

3.2 Funzionalità implementate

Vediamo ora per ogni funzione implementata i risultati di test e le varie considerazioni su essi.

3.2.1 Funzione: `concat_path_sequences`

La prima funzione implementata di cui si vuole osservare i risultati di testing è la funzione `concat_path_sequences`.

Risultati attesi

In relazione alla funzione `concat_path_sequences`, la cui logica risulta abbastanza semplice, ci si aspetta che le dimensioni del grafo, ovvero il numero di nodi e il numero di archi, non abbiano un'influenza sulle prestazioni della funzione stessa, al contrario di quanto abbiamo visto nel sotto-capitolo precedente.

Tuttavia, dato il codice della funzione ci si potrebbero aspettare altri parametri che influenzano il tempo di esecuzione, questi parametri sono il numero di path(P-linee) e la lunghezza delle sequenze associate a ogni nodo.

Risultati finali

Per quanto riguarda il tempo di esecuzione risulta essere sempre nullo, poichè il lavoro principale della funzione si limita a una semplice concatenazione di stringhe. L'assenza di variazioni temporali evidenzia la semplicità in termini computazionali della funzione.

Invece il grafico in figura 3.3 relativo al consumo di memoria rispecchia fedelmente quanto osservato durante il caricamento del grafo, poichè la funzione usa la memoria per gestire il grafo. Di conseguenza il consumo di memoria segue un andamento proporzionale al numero di nodi e archi.

L'analisi sul numero di path ha rivelato un impatto trascurabile sulle prestazioni della funzione. Infatti anche con aumenti significativi del numero di path(da 1 fino a 100 Milioni) ci sono state variazioni di tempo impercettibili. Il risultato sottolinea la robustezza della funzione al variare di questo parametro. Un discorso analogo può essere fatto per la lunghezza delle sequenze, oltre al fatto che come detto nel capitolo 1.2 di solito questa è compresa tra le 50 e le 1000 basi nei grafi considerati. Di conseguenza anche questo parametro ha un'influenza nulla.

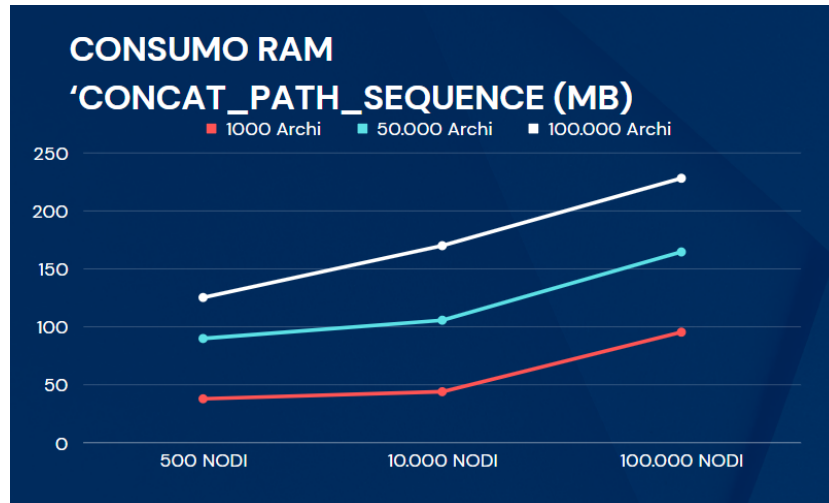


Figura 3.3

3.2.2 Funzione: `remove_node_and_merge`

Si osservano ora i risultati di test della funzione `remove_node_and_merge`. I test di cui si osserveranno i risultati sono stati costruiti in maniera simile ai precedenti, quindi la creazione di un grafo casuale con numero di nodi e archi ben preciso, invece per quanto riguarda il nodo da eliminare passato come parametro, si è tenuto sempre lo stesso nodo (ovvero sempre lo stesso id) per ogni singolo test.

Si evidenzia che l'analisi si concentra esclusivamente sulla modalità 1, poiché computazionalmente è considerata la più complessa. Questa modalità genera archi di collegamento tra tutti i predecessori e i successori del nodo da rimuovere. Per la modalità 2, l'andamento della curva temporale è simile, ma dipende anche dal numero di path (P-linee) presenti nel grafo.

Risultati attesi

Ci si aspetta che, con l'aumentare delle dimensioni del grafo e del numero di path (P-linee), aumentino sia il tempo di esecuzione che il consumo di memoria. La modalità 1, con la creazione di numerosi archi di collegamento, dovrebbe richiedere risorse computazionali proporzionali alle dimensioni del grafo. Per la modalità 2, l'andamento della curva temporale è previsto simile, ma con una dipendenza anche dal numero di path, riflettendo l'interazione tra la struttura del grafo e il numero di percorsi presenti.

Risultati finali

I risultati finali, come evidenziati dal grafico 3.4, mostrano un andamento del consumo di memoria simile a quello osservato durante il caricamento del grafo, aumentando con l'aumentare del numero di nodi e archi.

Per quanto riguarda il tempo di esecuzione emerge un andamento più interessante. Infatti non è la dimensione del grafo a influire in modo predominante, bensì la densità del grafo stesso. Come si può osservare dal grafico 3.5 il caso che ha impiegato il maggior tempo ed essere eseguito è quello con 500 nodi e 100.000 archi (densità=0,16) mentre quello che ci mette meno è il caso con 200.000 nodi e 100.000 archi (densità=0,0000005).

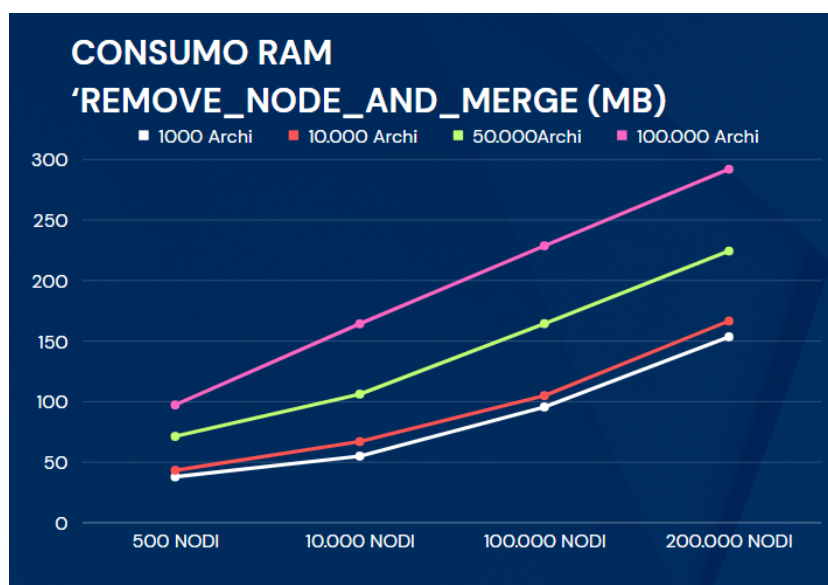


Figura 3.4

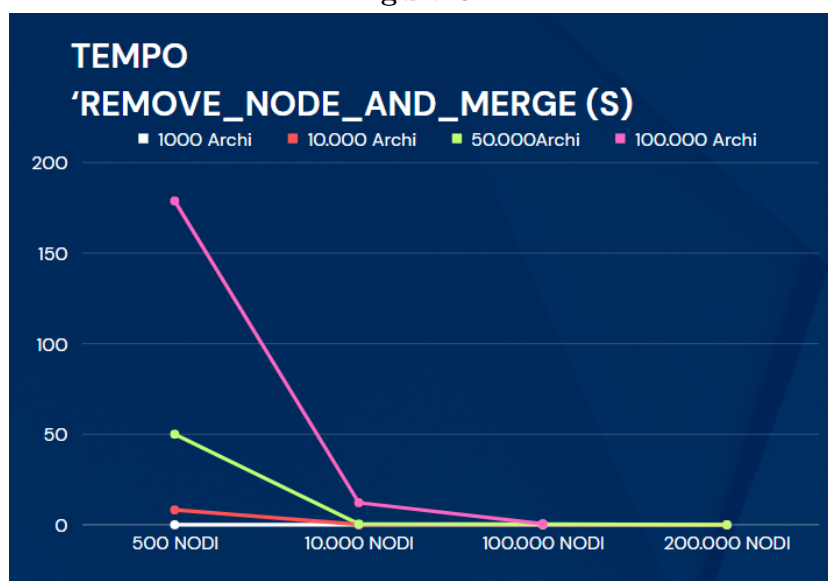


Figura 3.5

Questo trend è attribuibile al fatto che, nel primo caso, il nodo da eliminare ha generalmente più vicini, aumentando il numero di archi da creare. Nel secondo caso, è più probabile che il nodo da eliminare non abbia vicini, semplificando l'operazione di rimozione senza la necessità di generare numerosi collegamenti.

Quindi il motivo ancora più intrinseco che fa aumentare esponenzialmente la curva tempo, e che è una conseguenza della densità, e il numero di vicini. Come si può osservare nel grafico 3.6 infatti, tenendo un grafico con numero di nodi fisso (in questo caso relativamente basso, così da facilitare il caricamento da file) e aumentando il numero di vicini del nodo da eliminare (aumentando gli archi ovviamente) la curva relativa al tempo di esecuzione aumenta esponenzialmente.

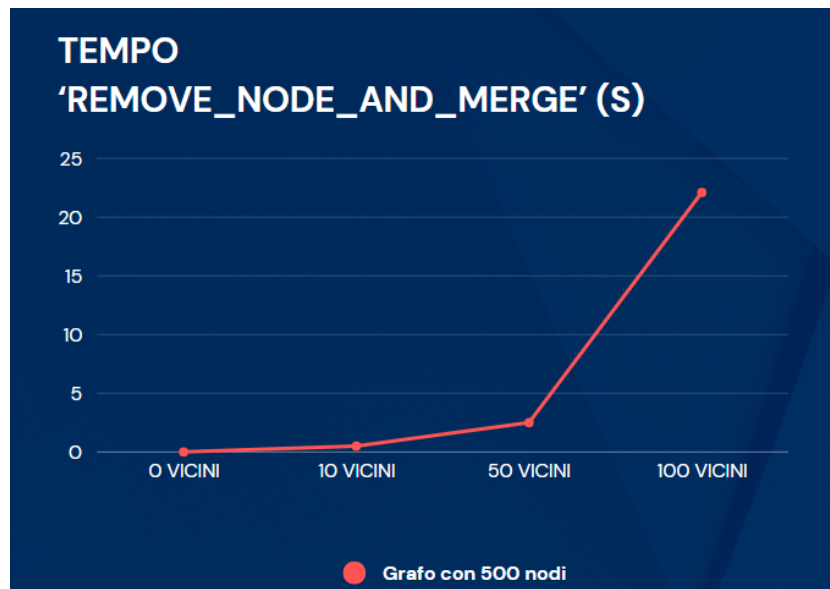


Figura 3.6

Si sottolinea che, per la modalità 2, non è presente un grafico temporale in quanto l'andamento risulta simile alla modalità 1 e il numero di path non risulta essere troppo influente sulle prestazioni.

3.2.3 Funzione: `get_Subgraph_from_neighborhood`

Per quanto riguarda la `get_Subgraph_from_neighborhood`, i test sono stati strutturati in linea con le metodologie adottate nelle sezioni precedenti. In particolare, per ogni dimensione di grafo, sono stati svolti test con tre diverse lunghezze di intorno (1, 2 e 3). Sebbene possano sembrare valori modesti, si prevede che l'impatto di questa proprietà risulterà significativo, come verrà evidenziato nei successivi risultati finali.

Risultati attesi

Si prevede che la lunghezza dell'intorno del nodo (vedi 1.1.2) influisca significativamente sulle prestazioni della funzione, soprattutto in termini di tempo di esecuzione. L'analisi si concentra sulla comprensione di come la densità del grafo, correlata al numero di vicini di un nodo, possa influenzare il tempo richiesto per ottenere il sottografo desiderato.

Risultati finali

Come evidenziato dai grafici 3.7, 3.8, 3.9, la densità del grafo emerge come un fattore preponderante nella curva temporale, influenzando principalmente il tempo di esecuzione della funzione. In aggiunta a ciò, si osserva che la lunghezza dell'intorno del nodo contribuisce al valore temporale, rendendo il tempo di esecuzione direttamente proporzionale alla dimensione dell'intorno considerato. In altre parole, pur mantenendo una forma di curva simile, l'aumento della lunghezza dell'intorno si traduce in valori temporali più elevati. Questo fenomeno suggerisce che, oltre alla densità del grafo, la configurazione specifica dell'intorno del nodo ha un impatto significativo sulle prestazioni della funzione.

È da notare che, a differenza degli altri sottocapitoli, non è presente un grafico relativo al consumo di RAM. Questo è dovuto al fatto che il consumo di memoria risulta essere analogo a quanto osservato durante il caricamento del grafo, e per evitare la ripetizione di informazioni simili, si preferisce concentrarsi sull'analisi del fattore tempo.

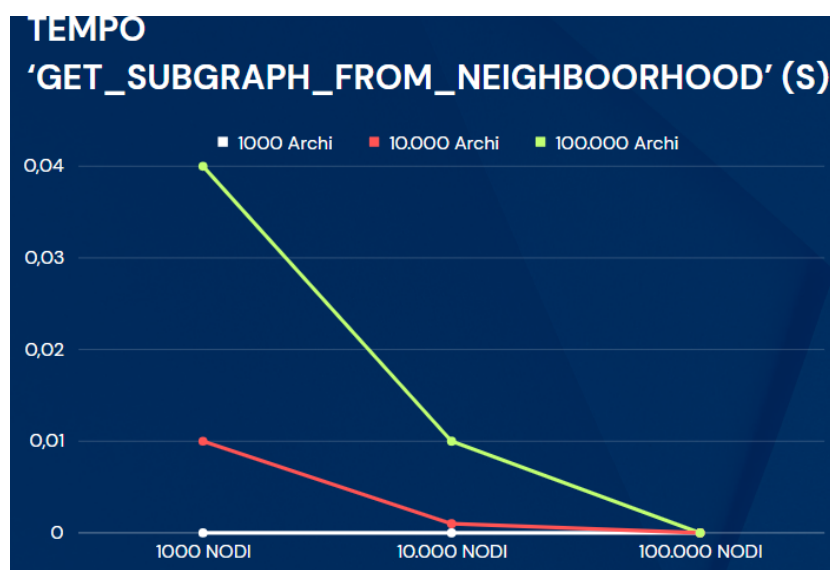


Figura 3.7: Intorno di lunghezza 1

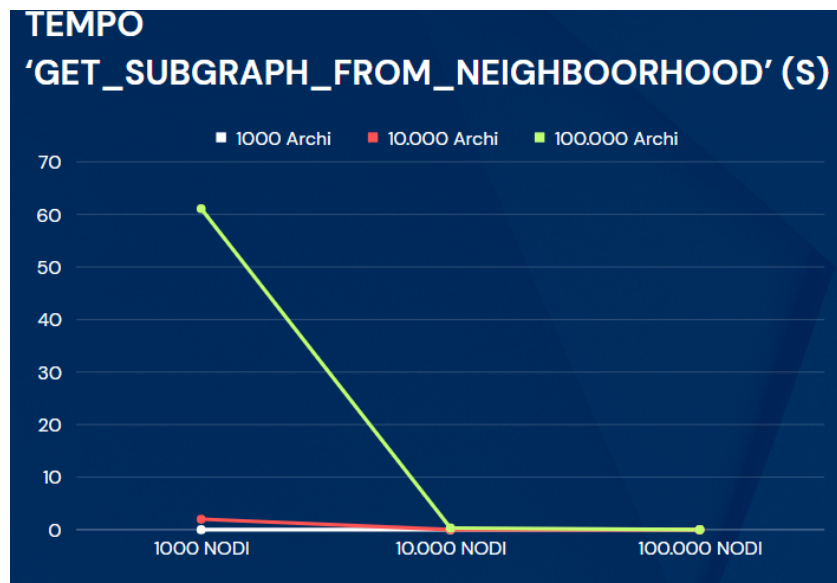


Figura 3.8: Intorno di lunghezza 2

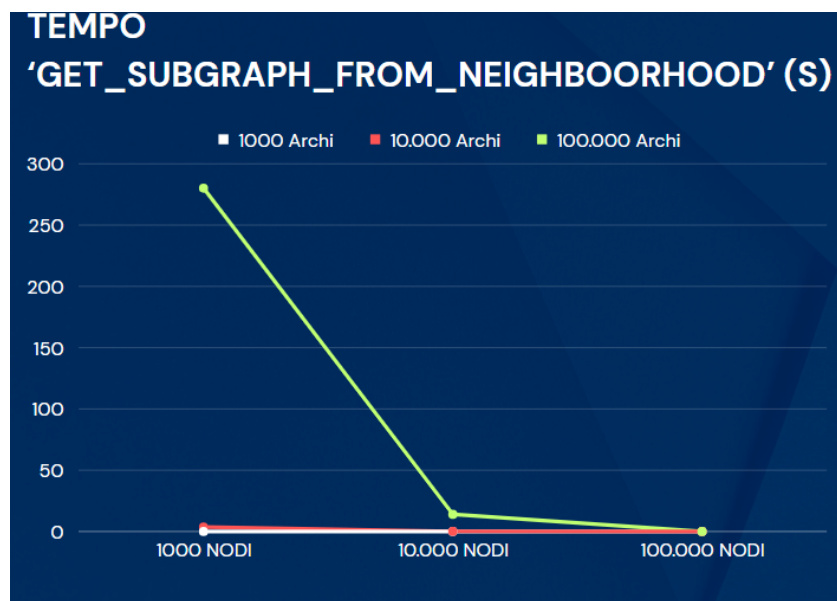


Figura 3.9: Intorno di lunghezza 3

3.2.4 Funzione: `get_Subgraph_between_nodes`

Ci si focalizza ora sull'analisi delle prestazioni della funzione `get_subgraph_between_nodes`, la quale restituisce il sottografo compreso tra un nodo di origine e un nodo di destinazione. Va notato che, in fase di testing, è stata valutata solo una delle due modalità, ovvero quella che tiene conto degli orientamenti degli archi. Entrambe le modalità sono state considerate equivalenti in quanto, in entrambi i casi, il fattore determinante è il numero di percorsi (P-linee).

Risultati attesi

Prima della fase di testing, ci si aspetta che il fattore determinante nelle prestazioni della funzione sia principalmente il numero di percorsi (P-linee). La previsione è che, all'aumentare del numero di percorsi, il tempo di esecuzione della funzione cresca in modo significativo.

Per quanto riguarda la memoria ci si aspetta che cresca in modo proporzionale con la dimensione del grafo (numero di nodi e archi).

Risultati finali

I risultati finali, come evidenziato dai grafici 3.10, 3.11, confermano in parte le ipotesi iniziali. L'andamento della curva temporale mostra una crescita esponenziale al crescere del numero di percorsi. Tuttavia, va notato che se il grafo contiene un numero significativo di percorsi, ma nessuno di essi collega direttamente il nodo di origine con quello di destinazione, l'operazione può risultare notevolmente più efficiente, eseguendo un semplice ciclo di controllo che richiederà poco tempo.

Per quanto riguarda il consumo di RAM, il grafico riflette l'andamento osservato durante il caricamento del grafo, crescendo in proporzione alle dimensioni del grafo stesso. In questo contesto, il numero di percorsi non sembra incidere significativamente sul consumo di memoria (o comunque ha un'influenza impercettibile).

In sintesi, quindi, la funzione mostra una sensibilità significativa al numero di percorsi, nel caso questi vadano effettivamente dal nodo sorgente al nodo destinazione.

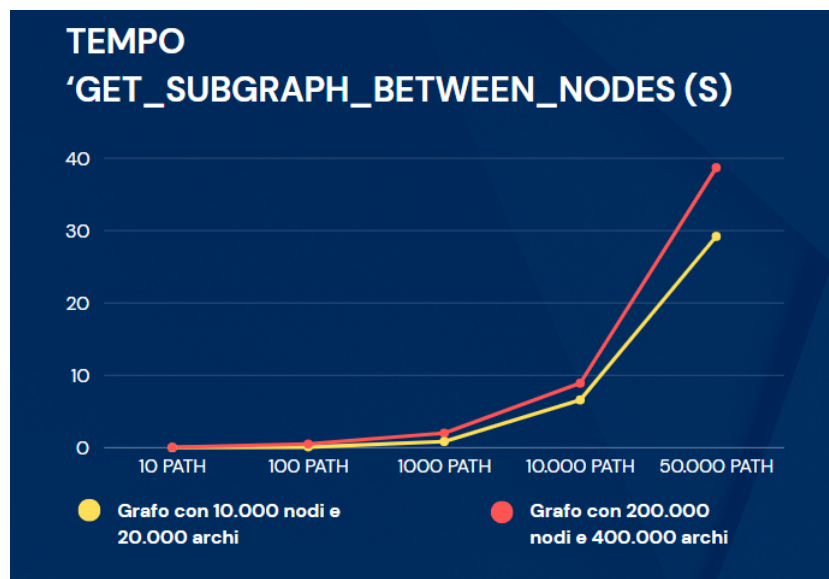


Figura 3.10

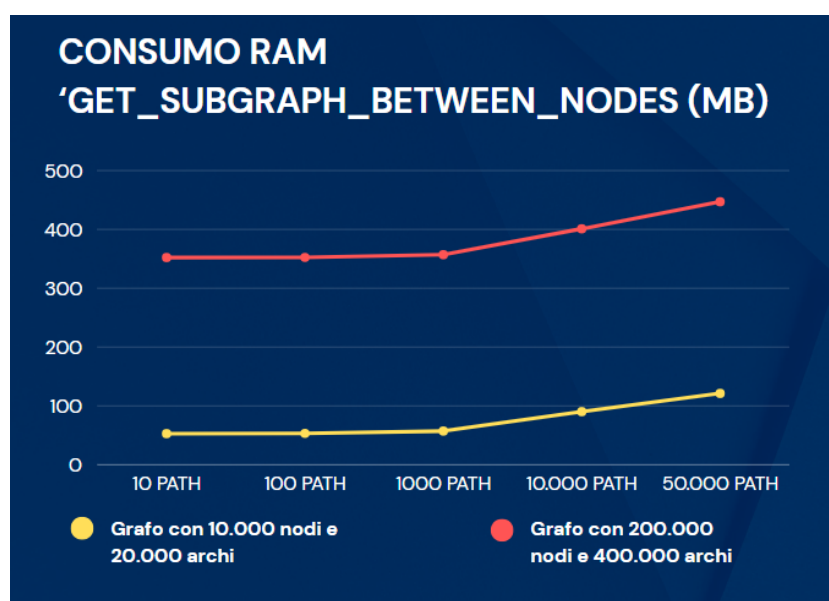


Figura 3.11

Capitolo 4

Possibili sviluppi futuri

Il lavoro svolto fino a questo punto ha rappresentato una fase importante nell'evoluzione della libreria **pygfa** e nella progettazione del tool **gfatk**.

Tuttavia, ci sono molte opportunità per ulteriori sviluppi e miglioramenti, sia nell'ambito delle operazioni implementate all'interno della libreria che nella crescita del software 'gfatk'.

Di seguito i possibili sviluppi da avere in futuro:

1. Ottimizzazione **get_subgraph_between_nodes**

Un potenziale sviluppo futuro riguarda l'ottimizzazione della funzione **get_subgraph_between_nodes**. Attualmente, la funzione si basa sulle P-linee per costruire il sottografo. Tuttavia, in futuro si potrebbe esplorare la possibilità di basare il sottografo sui cammini veri e propri del grafo. Questo richiederebbe un'analisi approfondita delle prestazioni per garantire tempi accettabili anche su grafi di dimensioni molto grandi.

2. Espansione della libreria

La strategia di sviluppo per la libreria **pygfa** si propone di mantenere un obiettivo chiave: la continua espansione e l'arricchimento delle sue funzionalità. Questa evoluzione costante sarà fondamentale per assicurare che la libreria rimanga al passo con le crescenti esigenze dell'analisi pangenomica e possa fornire soluzioni avanzate e versatili per una vasta gamma di scenari. Attraverso l'aggiunta di nuove funzionalità, operazioni più sofisticate e l'introduzione di nuove strutture dati, la libreria sarà in grado di coprire una maggiore varietà di compiti, supportando così una più ampia gamma di applicazioni pangenomiche.

Le possibili aree di sviluppo includono operazioni di editing del grafo più avanzate, consentendo agli utenti di effettuare modifiche complesse e mirate alla struttura del grafo pangenomico. Inoltre, l'espansione delle capacità di manipolazione delle sequenze rappresenterà un passo significativo, consentendo agli utenti di eseguire analisi più dettagliate e sofisticate sulle informazioni genomiche contenute nel grafo.

Un altro punto cruciale sarà l'incremento del supporto per formati di file aggiuntivi, ampliando così l'interoperabilità della libreria con altri strumenti e consentendo agli utenti di integrare agevolmente **pygfa** nelle proprie pipeline di analisi.

3. Sviluppo di **gfatk**

Il tool **gfatk** ha il potenziale per diventare uno strumento estremamente potente e flessibile per la manipolazione di grafi GFA. In futuro, sarà vantaggioso arricchire il tool con nuovi comandi che amplino la gamma di operazioni applicabili. Alcune possibili estensioni includono la capacità di convertire un path (concatenazione delle sequenze dei segments presenti in un path, costruito partendo da una P-linea) in un formato diverso (ad esempio, da GFA a FASTA) oppure estrarre sottosequenze specifiche da una sequenza più lunga oppure rinominare segmenti o path all'interno del grafo.

In sintesi arricchire il tool con nuovi comandi. L'obiettivo sarà quello di rendere **gfatk** uno strumento completo e versatile che soddisfi una varietà di esigenze analitiche all'interno del contesto pangenomico.

Capitolo 5

Conclusioni

Il percorso di sviluppo della libreria **pygfa** e del tool **gfatk** rappresenta un contributo significativo al campo dell'analisi pangenomica, fornendo strumenti avanzati per la manipolazione dei grafi GFA.

Nel corso di questo lavoro, mi sono concentrato nell'espandere la libreria e progettare un tool che consenta di utilizzare la libreria anche da linea di comando, favorendo una semplicità nell'utilizzo.

Con la libreria che si è evoluta in una risorsa più ricca e versatile e con l'implementazione di **gfatk** che offre un'interfaccia di utilità per le operazioni di gestione dei grafi GFA, l'obiettivo iniziale che era stato definito con il gruppo con cui ho lavorato può definirsi raggiunto.

Bibliografia

- [1] *A proposal of the Grapical Fragment Assembly format*. URL: <https://lh3.github.io/2014/07/19/a-proposal-of-the-grapical-fragment-assembly-format>.
- [2] Phillip E C Compeau, Pavel A Pevzner e Glenn Tesler. «How to apply de Bruijn graphs to genome assembly». In: *Nature Biotechnology* 29.11 (1 nov. 2011), pp. 987–991. ISSN: 1546-1696. DOI: 10.1038/nbt.2023. URL: <https://doi.org/10.1038/nbt.2023>.
- [3] Theodoros P. Gevezes e Leonidas S. Pitsoulis. «Recognition of overlap graphs». In: *Journal of Combinatorial Optimization* 28.1 (1 lug. 2014), pp. 25–37. ISSN: 1573-2886. DOI: 10.1007/s10878-013-9663-3. URL: <https://doi.org/10.1007/s10878-013-9663-3>.
- [4] *Graph Theory*. URL: <https://link.springer.com/book/9781846289699>.
- [5] The GFA Format Specification Working Group. *Graphical Fragment Assembly (GFA) Format Specification*. GFA-spec. 7 Giu. 2022. URL: <http://gfa-spec.github.io/GFA-spec/GFA1.html>.
- [6] Heng Li. *lh3/seqtk*. original-date: 2012-03-23T23:24:13Z. 4 Feb. 2024. URL: <https://github.com/lh3/seqtk>.
- [7] Vijini Mallawaarachchi. *Visualising Assembly Graphs*. Medium. 25 Mar. 2020. URL: <https://towardsdatascience.com/visualising-assembly-graphs-fb631f46bbd1>.
- [8] *MultiGraph—Undirected graphs with self loops and parallel edges — NetworkX 3.2.1 documentation*. URL: <https://networkx.org/documentation/stable/reference/classes/multigraph.html>.
- [9] Eugene W. Myers. «The fragment assembly string graph». In: *Bioinformatics* 21 (suppl_2 1 set. 2005), pp. ii79–ii85. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bti1114. URL: <https://doi.org/10.1093/bioinformatics/bti1114>.
- [10] Kelvin Salton do Prado. *Understanding time complexity with Python examples*. Medium. 15 Feb. 2020. URL: <https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7>.
- [11] Alberto Rizzardi e Matteo Comin. «Costruzione efficiente di overlap graph mediante fm-index per il metagenomic binning». In: ().
- [12] Raffaella Rizzi et al. «Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era». In: *Quantitative Biology* 7.4 (1 dic. 2019), pp. 278–292. ISSN: 2095-4697. DOI: 10.1007/s40484-019-0181-x. URL: <https://doi.org/10.1007/s40484-019-0181-x>.
- [13] Ghent University. *Overlap graph*. URL: <https://dodona.be/en/activities/649786026/>.

Ringraziamenti

Desidero esprimere la mia sincera gratitudine al Professore Gianluca Della Vedova per avermi introdotto a questo interessante campo dell'informatica e, soprattutto, per la sua costante disponibilità. La sua guida è stata fondamentale nel percorso di questo lavoro.

Un ringraziamento speciale anche ai Dottori Simone Ciccollella e Luca Denti che mi hanno accompagnato in questo percorso stimolando la mia riflessione e accrescendo il mio bagaglio di competenze.

Ringrazio i miei amici e tutte le persone che ho conosciuto in questi 3 anni che, anche se per poco tempo, hanno contribuito a distrarmi nei momenti più difficili del mio percorso universitario.

Desidero esprimere, infine, la mia profonda gratitudine alla mia famiglia, mia madre, mio fratello, mio padre e tutti i miei parenti, per il costante sostegno e per aver sempre creduto nelle mie potenzialità, anche quando io stesso potevo dubitarne. La loro fiducia incondizionata è stata la luce guida durante il mio percorso, e sono grato per avere una famiglia così solidale al mio fianco. Quello che sono oggi lo devo soprattutto a loro.