Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

PyGFA

Progettazione ed implementazione di una libreria Python per la gestione di file GFA

Relatore: *Gianluca Della Vedova*

Co-relatore: *Marco Previtali*

Relazione della prova finale di:

Diego Lobba

Matricola 795702

Anno Accademico 2016-2017

Indice

Elenco delle figure

Capitolo 1

Panoramica su *PyGFA*

Il DNA racchiude le informazioni che contraddistinguono gli organismi viventi. Esso si può descrivere per mezzo di una stringa composta da quattro lettere: A, C, G, T che denotano le quattro basi azotate (adenina, citosina, guanina e timina rispettivamente) che costituiscono gli elementi fondamentali del DNA stesso. Esso è un identificatore dello stato corrente di un organismo, per questo lo studio delle sue variazioni è di particolare interesse sia per l'identificazione di malattie, che si possono manifestare come variazione di un particolare gene, sia nel contesto evolutivo di una specie. A seconda delle necessità è possibile sequenziare l'intero genoma di un organismo oppure estrapolare le sole sequenze che ne codificano una parte, per esempio una proteina particolare.

In principio la tecnica di sequenziamento del DNA più frequente era la tecnica Sanger, in grado di restituire sequenze piuttosto lunghe (superiori al migliaio di coppie di basi, indicate con il termine *bp*) che permettono una ricostruzione meno complessa dell'intero genoma di un organismo. Questa metodologia permette di avere un'alta affidabilità sui dati ottenuti, ma ha degli alti costi d'utilizzo; per questo motivo il metodo Sanger viene oggi solitamente utilizzato per il sequenziamento completo di un DNA in casi particolarmente rari, solitamente per specie dalle quali non si ha ancora un genoma di riferimento o per la convalida di DNA ottenuti da metodi meno affidabili.

Negli ultimi anni si stanno diffondendo le cosiddette NGS (Next Generation Sequencing), tecniche che permettono di ottenere una grande mole di dati a prezzi molto ridotti, se confrontati con il metodo Sanger. Di contro questi metodi producono sequenze di lunghezza inferiore, solitamente intorno alle centinaia di coppie di basi, per questo motivo l'utilizzo di queste tecniche non permette il sequenziamento efficace di DNA molto lunghi o che presentano numerose ripetizioni, poiché il loro assemblaggio risulterebbe poco affidabile.

Le sequenze ottenute da entrambi i metodi non è garantito che siano le

une contigue alle altre, di conseguenza è richiesto l'uso di algoritmi di allineamento per individuare le parti sovrapposte che denotano il proseguimento della stringa di DNA. Le informazioni di questi allineamenti vengono salvate su file, seguendo una formattazione ben definita, per poi essere processate da programmi per l'assemblaggio del genoma.

I file GFA[8] (Graphical Fragment Assembly) sono file che descrivono questo tipo di informazioni. Ogni sequenza viene vista come un nodo sul quale possono esserci collegate altre sequenze per mezzo di collegamenti. Tali collegamenti possono coinvolgere la sequenza secondo due direzioni:

- la prima, indicata dal simbolo +, considera la sequenza così come appare nella sua definizione all'interno del file,
- la seconda, indicata dal simbolo -, considera la sequenza dopo che su di essa viene effettuata un'operazione di *reverse and complement*.

I file GFA possono arrivare a contenere milioni tra sequenze e collegamenti presenti tra di loro, per questo motivo si è voluto sviluppare una libreria, *PyGFA*, in grado di gestire tali file permettendo non solo di andare ad effettuare operazioni di filtraggio e selezione sulle informazioni contenute, ma anche in grado di fornire una serie di strumenti per la loro manipolazione ed ulteriore analisi.

*PyGFA*è una libreria Python che permette di gestire le informazioni contenute in file GFA rappresentandole mediante una struttura a grafo. La gestione delle operazioni sul grafo avviene sfruttando una libreria preesistente: *networkx*[2], per la quale vengono fornite le interfacce ai metodi. *PyGFA*inoltre permette l'attraversamento del grafo mediante iteratori personalizzati, che considerano solamente archi rappresentanti un determinato tipo di connessione (*dovetail overlap*) fra i nodi del grafo.

1.1 RGFA e GfaPy

*PyGFA*non è la sola libreria che gestisce file GFA mediante una struttura a grafo. RGFA è una libreria scritta in Ruby creata appositamente per questo scopo e GfaPy è l'equivalente riscritta in Python. GfaPy non solo implementa le funzionalità di RGFA, ma estende il supporto alla specifica GFA2 (che verrà successivamente illustrata).

Nonostante le somiglianze fra GfaPy e *PyGFA*, le due librerie non solo differiscono a livello implementativo, ma hanno una gestione dei dati completamente diversa. Le informazioni in GfaPy continuano a tenere informazioni sulla loro origine (il tipo di linea dalla quale provengono). In *PyGFA*invece le informazioni vengono ricondotte ad un unico livello di astrazione; tutti i collegamenti possiedono lo stesso numero di campi, i quali successivamente potranno averli definiti o meno a seconda del tipo di linea/collegamento.

Questo approccio di *PyGFA*è stato preferito per tenere una coerenza generale nei dati usati dal grafo networkx sottostante e per avere un'informazione che l'utente della libreria può interpretare in senso più ampio; slegato dal concetto che la specifica GFA le assegna, ma rappresentante il concetto biologico che tale informazione vuole significare.

Capitolo 2

Ecosistema e strumenti

In questo capitolo verranno descritti i linguaggi utilizzati e gli strumenti impiegati nell'analisi, nello sviluppo e nei test della libreria. Di ogni elemento verrà fornita una breve panoramica, ponendo maggior enfasi sugli aspetti (alle volte piuttosto tecnici) che è stato necessario tenere in considerazione durante lo sviluppo di *PyGFA*.

2.1 Il linguaggio Python

Python è un linguaggio di programmazione ideato da Guido van Rossum all'inizio degli anni '90. Python è un linguaggio che si appresta a molteplici stili di programmazione, sfruttando caratteristiche del paradigma object oriented, funzionale e della programmazione strutturata. Tali proprietà permettono l'uso del linguaggio in una grande varietà di attività: nella creazione di script di automazione di sistema, nella scrittura di sistemi web di backend fino allo sviluppo di complesse librerie di analisi numerica e machine learning.

La minima struttura necessaria a produrre un programma Python, l'inferenza del tipo delle variabili a runtime (*dynamic typing*), la gestione automatica della memoria e la sua espressività lo rendono un candidato ideale nella prototipazione di sistemi nuovi, coerente con processi di sviluppo agili e con la pratica di extreme programming.

Figura 2.1: Logo del linguaggio di programmazione Python.

Inoltre Python è un linguaggio interpretato, di conseguenza è possibile effettuare, direttamente all'interno dell'interprete, verifiche in tempo reale delle funzionalità attualmente in sviluppo. Il fatto che il linguaggio venga interpretato da un interprete permette una facile integrazione del codice nativo

con il quale l'interprete stesso è stato implementato. Per questo motivo -e grazie al livello di astrazione sul quale si colloca- esistono diverse implementazioni del linguaggio Python che sfruttano la JVM, la piattaforma .NET, il linguaggio C o che si concentrano maggiormente su alcuni aspetti specifici come la velocità, il multithreading e la minimalità per l'uso in ambienti embedded.

Listato 2.1: Un esempio dell'espressività del linguaggio applicata a *PyGFA*.

```
>>> import pygfa
>>> gfa = pygfa.gfa.GFA.from_file("data/sample1.gfa")
>>> gfa.nodes()
['1', '3', '2', '5', '13', '6', '11', '12', '4']
>>> pygfa.nodes_connected_components(gfa) #calcola i nodi
...     di ciascuna componente connessa
...
[{'2', '1', '3', '6', '5'}, {'11', '13', '12'}, {'4'}]
>>> # estrai tutte le componenti connesse
        con numero di nodi maggiore di 3
...
>>> [component for component in pygfa.nodes_connected_components(
    gfa)
        if len(component) > 3]
[{'2', '1', '3', '6', '5'}]
```

2.1.1 Classi, convenzioni e duck typing

In Python per creare una classe è necessario definirla mediante la dicitura:

```
class MiaClasse(Antenata1, Antenata2, ...):
        ...
```

Come è possibile notare, è ammissibile ereditare da più classi. Tale caratteristica non è molto frequente negli altri linguaggi e viene spesso considerata una cattiva pratica, visto che non permette di accorgersi di una errata definizione della gerarchia delle classi di progetto. Tuttavia, in Python questa proprietà permette l'aggiunta di funzionalità alla classe, in modo analogo alla modalità `implements` di Java, rendendo possibile una definizione di quelle che in realtà sono le interfacce e le loro implementazioni. In *PyGFA* tale funzionalità è stata applicata per aggiungere le funzionalità degli iteratori personalizzati alla classe `GFA`.

Python è un linguaggio fortemente influenzato dai movimenti open source; infatti ogni programma, libreria e sistema scritto in Python presuppone che l'utilizzatore abbia libero accesso al sorgente e che possa capire le modalità di utilizzo di ogni modulo di cui è composto. Per questo motivo i

programmi tendono ad essere ricchi di documentazione, sia essa incorporata nel codice che allegata nel manuale. Come conseguenza il linguaggio non offre un meccanismo per definire i metodi di una classe come privati in virtù del fatto che l'utilizzatore, avendo la piena possibilità di capire il funzionamento del singolo modulo di sistema, ha la piena responsabilità delle sue azioni. Per aiutare a distinguere elementi del programma che l'autore vorrebbe fossero non utilizzati (o utilizzati con particolare consapevolezza) si è soliti nominare tali elementi facendo precedere i loro nomi da un singolo trattino basso (*weak internal use*). E' possibile imprimere maggior enfasi nell'oscuramento di un elemento precedendo il suo nome da due trattini bassi, in tal modo l'accesso all'elemento avviene precedendo il nome della classe al nome dell'elemento.

Listato 2.2: Convenzioni per l'uso interno.

```
>>> class A:
...     def __init__(self):
...         self.normal_use = 5
...         self._internal_use = 10
...         self.___strong_internal = 15
...
>>> a = A()
>>> a.normal_use
5
>>> a._internal_use
10
>>> a._A___strong_internal
15
```

Una ulteriore convenzione comune al Python e ampiamente considerata nello sviluppo di *PyGFA* è il cosiddetto *duck typing*. Visto che il Python usa il binding dinamico, le funzioni e i metodi non possono verificare il tipo dei parametri passati tramite signature. Una possibile soluzione è verificare il tipo del parametro mediante la funzione `isinstance`, come avviene per esempio quando si cerca di effettuare il *downcast* dalla classe antenata ad una sottoclasse nei linguaggi in cui il binding avviene staticamente. Per esempio tale pratica si verifica in Java quando si va ridefinire il metodo `equal` di una classe, avente come signature un tipo Object rappresentante l'istanza da confrontare ed effettuando il downcasting del parametro alla classe attuale prima di effettuare i confronti fra le due istanze.

In Python invece viene considerata un'altra via. L'oggetto passato come parametro viene confrontato direttamente con l'istanza e in caso di errori (per esempio l'oggetto potrebbe non avere un parametro al quale si sta accedendo), viene lanciata un'eccezione dalla quale si ritorna l'ineguaglianza fra i due oggetti. Questo è ciò che si intende per duck typing.

Questo comportamento permette di avere delle classi molto flessibili, visto che la compatibilità fra queste viene garantita dall'uguaglianza delle interfacce (se due oggetti sono diversi, ma in un determinato contesto hanno lo stesso comportamento, allora possono essere considerati simili); ma potenzialmente annulla la simmetria dell'operatore d'uguaglianza. Di conseguenza, dati due oggetti A e B di classi diverse, si può avere la situazione in cui $A = B$, ma $B \neq A$.

2.1.2 Perchè è stato scelto Python

Oltre alle caratteristiche spiegate precedentemente che rendono Python un linguaggio flessibile, potente e anche divertente da usare, altri fattori che motivano la sua scelta sono:

- l'elevato numero di librerie disponibili, di strumenti che accompagnano e velocizzano lo sviluppo e la manutenzione di un progetto: analizzatori di sintassi, strumenti di test e ambienti per la generazione automatica della documentazione;
- la sua diffusione in ogni campo di applicazione, compreso quello bioinformatico, che è una delle principali motivazioni dello sviluppo di *PyGFA*;
- l'ampia documentazione sia della libreria standard che delle librerie esterne, con soluzioni a buona parte dei problemi di programmazione comuni, comportando un abbassamento dei costi e dei tempi di manutenzione e sviluppo dei sistemi.

2.2 NetworkX

NetworkX è una libreria Python che permette di creare e gestire grafi. Essa fornisce inoltre un'ampia collezione di algoritmi applicabili a grafi e alberi: algoritmi di attraversamento, di cammini minimi, di analisi del flusso; sui quali si basa per fornire informazioni topologiche del grafo, come la presenza di cicli, di componenti connesse e di punti di articolazione.

La libreria usa le liste di adiacenza per rappresentare i nodi e gli archi ad essi collegati mediante tre dizionari Python nidificati. Il primo dizionario contiene gli identificativi univoci di ogni nodo e come valore ha un secondo dizionario con chiavi i nodi collegati ad esso mediante un arco. Il valore di questi è a loro volta un dizionario che rappresenta gli archi che collegano il nodo del primo dizionario con quello definito nel secondo.

Questa scelta, oltre a rappresentare il modo più corretto per implementare tale struttura[6], permette un rapido accesso ai nodi e agli archi che li mettono in relazione.

Oltre a specificare un identificativo univoco da assegnare a nodi e archi, è possibile definire delle proprietà che possono essere dei tipi primitivi o degli oggetti.

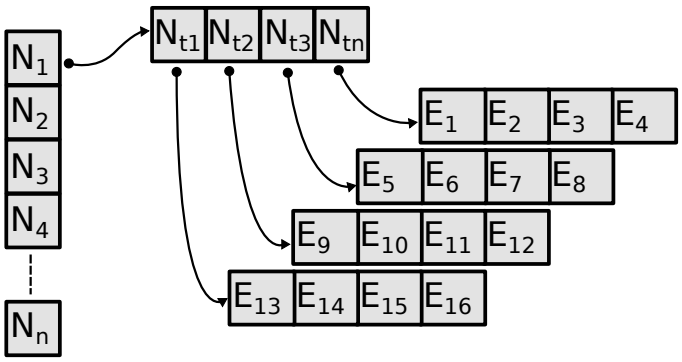

Figura 2.2: Rappresentazione grafica dei nodi e degli archi descritti in networkx.

2.2.1 Tipi di grafo

Vi sono quattro differenti tipologie di grafo a disposizione:

- grafo non diretto,
- grafo diretto,
- multigrafo non diretto,
- multigrafo diretto

Nel grafo non diretto, all'aggiunta di un arco (u, v) tra i nodi u e v viene aggiunto automaticamente un arco (v, u).

I multigrafi permettono di definire molteplici archi delimitati dalla stessa coppia di nodi. In tal caso la definizione dell'identificativo di un arco è di maggiore importanza, visto che sarà necessario discriminare l'arco desiderato da un insieme di collegamenti tra la stessa coppia di nodi. Per i multigrafi (diretti e non) NetworkX permette, all'aggiunta di un arco, di specificare la chiave (identificativo) associato all'arco che si sta inserendo; nel caso in cui l'utente non definisce un identificativo, l'oggetto rappresentante il grafo fornisce automaticamente un numero intero da assegnarli non ancora utilizzato.

2.2.2 Perchè è stata scelta e limiti

Il contenuto di un file GFA è direttamente rappresentabile mediante un grafo, per questo sfruttare una libreria già esistente ha permesso di velocizzare i tempi di sviluppo.

NetworkX offre la miglior implementazione dei grafi scritta in Python, sfruttando dove richiesto librerie matematiche anche di basso livello per garantire prestazioni molto alte in termini di velocità. La vasta gamma di algoritmi a disposizione, uno sviluppo costantemente attivo[3] e un'ampia documentazione rendono questa libreria un'ovvia candidata per lo sviluppo di *PyGFA*.

Uno svantaggio, dovuto all'implementazione direttamente in Python di Networkx, è il consumo piuttosto elevato di memoria richiesto per contenere il grafo. Tale peso è dovuto principalmente all'uso dei dizionari come struttura di rappresentazione di nodi ed archi. In *PyGFA* tale inconveniente

è possibile notarlo specialmente nei dizionari degli archi, che arrivano ad occupare più di 2 kB.

Un altro aspetto negativo della libreria è che non permette l'impiego degli algoritmi considerando le proprietà di archi e nodi. Ciò significa che, supponendo di avere degli archi colorati (blu, giallo, rosso), non è possibile applicare gli algoritmi solo agli archi di uno specifico colore. Per questo in *PyGFA* si è reso necessario andare a ridefinire (sfruttando il sorgente) gli algoritmi di interesse andando a considerare nell'algoritmo i nodi definiti da un iteratore, che li seleziona in base ad una proprietà dell'arco, evitando di considerare l'intera lista di adiacenza del singolo nodo.

2.3 unittest e Coverage.py

Python è un linguaggio dinamico, per questo ogni riga di un programma viene analizzata solo nel momento in cui lo script viene lanciato. Ciò vuole significare che un errore (esclusi quelli di sintassi) non può essere individuato prima della sua esecuzione.

Per questo motivo nello sviluppo di *PyGFA* si è scelto di scrivere i casi di test con la libreria *unittest*, fornita direttamente con l'interprete, e successivamente si sono andati ad analizzare gli script dei casi di test con *Coverage.py*.

Coverage.py è un programma Python in grado di misurare la copertura del listato scritto, annotando le parti del codice che sono state eseguite e creando un report (su file o su interfaccia web) indicando le righe che necessitano ulteriore copertura.

Entrambi gli strumenti sono stati presi in considerazione per via della loro facilità di integrazione nello sviluppo, per la loro chiarezza nell'esposizione degli errori e delle statistiche e per la loro diffusione tra gli sviluppatori.

2.3.1 Funzionamento

Lo strumento di analisi per la copertura prende in input un programma Python il quale viene eseguito linea per linea. Tutti i file coinvolti nel programma (poiché contenenti funzioni, classi o variabili usate presenti nello script) vengono considerati nel calcolo della copertura. La misura della copertura avviene sia per singolo file che sull'intero insieme di file coinvolti dal programma. La copertura sul singolo file viene calcolata come:

$$Cov(f) = \frac{\#linee_{coperte}}{\#linee_{totali}} \tag{2.1}$$

unittest invece permette di confrontare il risultato di un'operazione rispetto ad un valore atteso, fermando l'esecuzione nel caso in cui uno di questi confronti fallisce. E' possibile inoltre confrontare il comportamento

atteso da una certa operazione, come l'invocazione di un'eccezione in caso di operazioni non valide.

2.4 Pylint

Pylint è uno strumento di analisi del codice, curato dalla Python Code Quality Authority[11], in grado di applicare una serie di regole atte a verificare:

- la compatibilità del programma rispetto le convenzioni stabilite dal linguaggio[4];
- problemi di importazione delle librerie;
- la presenza di variabili usate in un contesto in cui il loro valore sarebbe indefinito;
- il verificarsi di codice duplicato, sia all'interno dello stesso file sia fra sorgenti diverse.

Lo strumento quindi non solo fornisce dei meccanismi di standardizzazione del codice, ma effettua quelle operazioni complementari a unittest e coverage.py per la verifica della correttezza, da un punto di vista sintattico e simbolico, del programma.

2.4.1 Come è stato usato

Pylint è uno strumento di *controllo*, che fornisce suggerimenti riguardanti molti aspetti del codice. Osservare tutti gli avvertimenti e risolvere tutti i problemi rilevati avrebbe richiesto un prolungamento dei termini di consegna, oltre che costituire un lavoro non prioritario. Il suo impiego in *PyGFA* è voluto per cercare di uniformare una nuova libreria Python con le convenzioni e le pratiche più comuni che la maggior parte degli utilizzatori di questo linguaggio si aspettano, fornendo loro un ambiente familiare nel quale la complessità non sia data dalla costante ricerca nella documentazione di nomi e comportamenti insoliti circa elementi che compongono la libreria. Generalmente si sono cercati di risolvere tutti quei problemi relativi gli standard di nomenclatura oltre che problemi di analisi simbolica che avrebbero compromesso il corretto funzionamento di *PyGFA*.

2.5 Sphinx e Read the Docs

Sphinx è uno strumento per la generazione automatizzata di documentazione del codice. Grazie a questo strumento è possibile ricavare un manuale ben formatto direttamente dal sorgente, il quale deve essere scritto secondo un

linguaggio ben specifico per indicare elementi di rilievo del codice, come parametri, valori di ritorno, eccezioni, note o link.

Sphinx è stato creato in origine per la documentazione ufficiale del linguaggio Python[5] e fin da subito si è sparso come strumento di aiuto nella documentazione di programmi scritti non solo in questo linguaggio, ma anche in molti altri linguaggi supportati, vista la sua flessibilità e i risultati ottimi che produce.

Esso usa il *reStructuredText* come linguaggio di markup, permettendo un'ampia espressività e una vasta gamma di notazioni come link (interni ed esterni), definizione di tag personalizzati e liste (ordinate e non) oltre la possibilità di scrivere formule matematiche in LaTeX.

Sphinx permette di generare la documentazione finale nei formati più comuni: HTML (con supporto mobile nativo), PDF e EPUB. Solitamente la scelta più diffusa (che *PyGFA* segue) è quella di produrre la documentazione in HTML e di renderla disponibile su *Read the Docs*.

Read the Docs è una piattaforma online, supportata dalla community, appositamente pensata per salvare, catalogare e rendere disponibile la documentazione scritta con Sphinx. Il sito fornisce un ambiente Python e permette di collegare la documentazione direttamente ad un repository di progetto su Github, evitando così di dover mantenere aggiornate due copie di un singolo progetto separate. Il sito usa l'ambiente Python per generare la documentazione di progetto, permettendo di aggiungere le dipendenze del codice mediante file di testo. La procedura è automatizzata (vedere figura 2.3) e il codice HTML risultante viene pubblicato sulla piattaforma al termine del processo.

Grazie a Sphinx e Read the Docs è stato possibile documentare *PyGFA* in modo facile e veloce, evitando di costruire soluzioni *ad-hoc* per la sua distribuzione e manutenzione e, allo tempo stesso, si è riusciti ad ottenere un risultato più che accettabile utilizzando i temi predisposti della piattaforma, garantendo all'utilizzatore una facile consultazione sia da desktop che da mobile, una visione del sorgente direttamente dalla documentazione e una funzione di ricerca efficace.

2.6 git e GitHub

git è un sistema di controllo versione, ideato da Linus Torvalds per gestire lo sviluppo del kernel Linux[13], tra i più utilizzati al mondo. Esso permette di gestire progetti di ogni dimensione, garantendo velocità, coerenza e ripristino fra le diverse tappe di sviluppo di un progetto.

Il programma permette, una volta inizializzato all'interno di una cartella, di gestire i file in base ai cambiamenti ad essi effettuati. Per integrare un cambiamento apportato ad un file è necessario aggiungerlo a quella che viene chiama *staging area*, contenente l'insieme di file modificati a partire

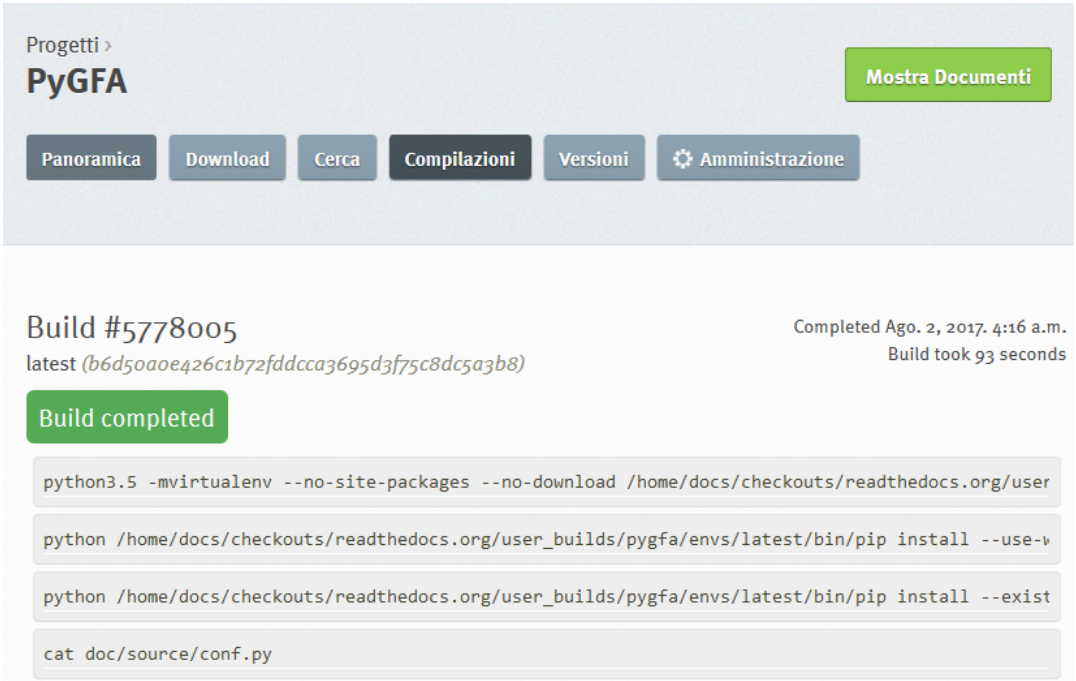

Figura 2.3: Interfaccia per la generazione della documentazione in Read the Docs.

dall'ultimo stato aggiornato del sistema. Generalmente i file nella staging area sono accomunati da un contesto comune (una modifica che coinvolge per lo stesso motivo quello specifico insieme di file), ma ai fini del sistema ciò non è un requisito indispensabile. Terminate le modifiche e aggiunte alla staging area è necessario integrare tali modifiche nel sistema mediante una *commit*, un'operazione che crea un identificativo univoco dello stato del sistema nel momento esatto in cui i cambiamenti vengono integrati con esso. Grazie alla creazione di questo identificativo è possibile ritornare nell'esatto stato del sistema indicato, nel caso fosse necessario.

Git permette non solo di lavorare ad un progetto procedendo in un'unica sequenza di sviluppo, ma permette la creazione di più diramazioni parallele (*branch*), indipendenti dalle future modifiche apportate al sistema, che possono procedere nello sviluppo. Tali diramazioni garantiscono un ambiente di lavoro isolato e stabile nel quale un singolo sviluppatore può concentrare il suo sviluppo, senza preoccuparsi delle modifiche che altri sviluppatori potrebbero apportare al sistema, per poi far ricongiungere il componente nella principale sequenza di sviluppo (solitamente indicato dal branch *master*).

La flessibilità di sviluppo che questo strumento offre permette di strutturare al meglio le diverse fasi di implementazione di un progetto, in piena coerenza con un processo agile o di extreme programming. In *PyGFA* ta-

le caratteristica si è rivelata di fondamentale importanza, permettendo di suddividere fasi di refactoring o di ridefinizione della struttura di progetto (operazioni molto delicate da un punto di vista della stabilità della libreria) in un ambiente controllato e reversibile in caso di problemi.

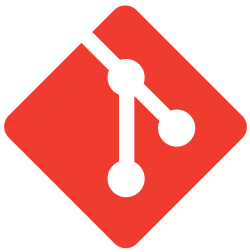

Figura 2.4: Logo dello strumento di controllo versione git.

Completate le modifiche ed effettuata la commit, è possibile sincronizzare i cambiamenti locali con una repository remota, un luogo decentralizzato sul quale il lavoro viene salvato e grazie al quale è possibile condividere il progetto con i propri collaboratori. Una tra le più famose piattaforme che permettono di ospitare un repository git remoto è *GitHub*.

GitHub è un sito che offre funzionalità paragonabili a quelle che Read the Docs (vedi sezione 2.5 a pagina 18) fornisce per Sphinx. Permette di ospitare repository remote, catalogarle per una maggiore accessibilità (se pubbliche) offrendo un'interfaccia intuitiva per:

- la creazione di *tag* per il rilascio delle versioni stabili del sistema,
- la creazione di aree di discussione relative a bachi, miglioramenti e problematiche generali, fornendo agli sviluppatori un luogo centralizzato e coerente per la discussione di queste tematiche,
- la creazione di punti di branching e analisi delle diverse ramificazioni del sistema e dell'attuale sequenza di sviluppo del progetto,
- la navigazione dell'intero progetto, con la possibilità di ispezionarlo ricreandone lo stato dopo una specifica commit,
- la visione di statistiche relative le commit effettuate da chiunque abbia contribuito al sistema, con la possibilità di analizzare le modifiche apportate a livello di singola commit.

Il ruolo che questi due strumenti hanno avuto e avranno nello lo sviluppo di *PyGFA*è *incalcolabile*. Non solo ai fini del salvataggio del progetto, della sua gestione e della sua reperibilità, ma anche per la possibilità che esso offre di permettere ad un qualsiasi sviluppatore di effettuare facilmente una diramazione del codice ai fini di poterlo riscrivere in base alle proprie necessità, senza doversi cimentare nello sviluppo di un sistema nuovo (*fork*).

2.7 Bandage

Bandage[12] è un programma per la visualizzazione grafica di grafi di assemblaggio. Questo strumento è in grado di visualizzare grafi descritti da

file GFA1, per questo il suo impiego è stato di vitale importanza nell'analisi dei collegamenti presenti fra le sequenze, specialmente nei casi di dovetail overlap, che descrivono un continuum nel genoma e che per tanto costituiscono un'informazione di rilievo per una libreria che si occupa della gestione di questi file.

2.8 Conclusioni

In questo capitolo sono stati presentati gli strumenti impiegati nello sviluppo di *PyGFA*, nella fase di test e di documentazione indicando i dettagli che si sono rivelati di particolare importanza ai fini implementativi del sistema.

Capitolo 3

Le specifiche GFA

In questo capitolo verrà fornita un'ampia panoramica sulle specifiche GFA, sulle linee presenti e sulle diverse circostanze di assemblaggio di sequenze che possono essere rappresentate. Verrà inoltre fornita una breve introduzione ad alcuni concetti biologici riguardanti gli elementi che compongono DNA e RNA e la loro struttura, necessarie a comprendere le diverse situazioni descritte nelle specifiche.

3.1 Introduzione a GFA, motivazioni e struttura

GFA è l'acronimo per Graph Assembly Format, è un formato per la rappresentazione dei legami presenti fra le sequenze di un genoma al fine di riuscire a ricostruirne la struttura. Le motivazioni che risiedono alla base della proposta per un nuovo formato consistono nell'uniformare le notazioni che programmi di visualizzazione, di assemblaggio e di manipolazione potessero utilizzare.

La prima versione della specifica GFA viene indicata col termine GFA1. Questa prima versione, come vedremo successivamente, limita la descrizione delle possibili situazioni in cui due sequenze possono trovarsi in relazione. Per questo motivo, e per estendere maggiormente l'insieme delle informazioni utili da descrivere, è stata sviluppata una seconda specifica, indicata con GFA2. Questa specifica generalizza, usando un'unica notazione, i collegamenti fra sequenze descritti da GFA1 e permette inoltre di descrivere relazioni di ogni tipo fra due sequenze. GFA2 è un *superset* di GFA1 e come tale permette (con un minimo numero di operazioni) di trasformare un file GFA2 nell'analogo (rappresentabile) in GFA1. Questa seconda specifica è stata appositamente pensata per permettere la descrizione di sequenze e collegamenti imponendo un minimo numero di vincoli, permettendo all'utilizzatore di impiegarla per la descrizione di dati indipendentemente dai dettagli che questi forniscono.

Entrambe le specifiche adoperano la stessa formattazione delle linee. Una linea descrive un'informazione di assemblaggio, sia essa una sequenza, un collegamento o un insieme di elementi. In ogni riga, il primo carattere indica l'identità della linea stessa alla quale seguono, separati esclusivamente da tabulazioni, gli elementi che costituiscono l'informazione che la linea descrive e che prendono il nome di *campi*. I campi possono essere definiti o meno, nel qual caso l'assenza dell'informazione viene indicata con un asterisco `*`.

In ogni linea di entrambe le specifiche è possibile descrivere campi opzionali (che possono essere predefiniti per una linea o introdotti direttamente dall'utente), descritti nel formato `TAG:TIPO:CONTENUTO` dove `TAG` è una sequenza di due caratteri alfanumerici (in maiuscolo se il campo è predefinito dalla linea, in minuscolo altrimenti) che identifica l'informazione che esso indica. Il `TIPO` di un campo viene anch'esso descritto da un identificatore, ciascuno indicante il seguente contenuto:

Tipo	Descrizione
A	Singolo carattere stampabile(escluso lo spazio)
i	Intero con segno
f	Decimale con precisione singola
Z	Stringa stampabile (incluso lo spazio)
J	Stringa JSON, escludendo caratteri di newline e di tabulazione
H	Array di Byte in formato esadecimale
B	Array di interi o di decimali

Tabella 3.1: Tabella dei tipi che è possibile usare per specificare campi opzionali.

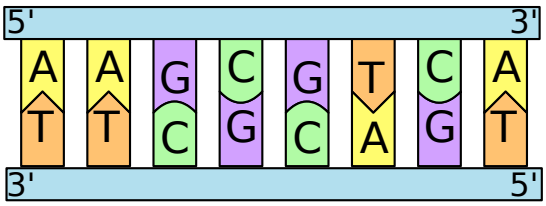

Figura 3.1: Rappresentazione grafica degli strand che compongono il DNA.

Mentre verranno analizzate le linee delle due specifiche, è essenziale avere un'idea di cosa sia una sequenza e di come questa può essere in relazione con le altre. Con il termine sequenza viene indicata una *sequenza nucleotidica*, un susseguirsi di lettere che denotano le unità molecolari che compongono gli acidi nucleici di RNA e DNA (*nucleotide*). Una sequenza è priva di un ordine specifico, ma è possibile attribuirgliene uno osservando la composizione del tipo di legame che collegano gli elementi costitutivi il nucleotide, in base all'orientamento del legame presente tra le unità di carbonio 3' di un un'unità e la stessa unità 5' della

successiva. Grazie a tale osservazione è possibile individuare un ordinamento che verrà definito come 5'3'.

Oltre questa considerazione, bisogna tenere conto che l'informazione presente nel DNA è la stessa a parità di estremità, ma in ordine inverso e complementato (vedi figura 3.1) (sostituendo la citosina con la guanina e l'adenina con la timina). Nel caso di RNA alla timina si sostituisce l'uracile, ma il processo di formazione del RNA prevede anch'esso questa operazione di complementazione della sequenza.

Ergo, quando si considera una sequenza (nel caso dell'assemblaggio del DNA), è necessario tenere presente che un collegamento fra due sequenze potrebbe considerare una sequenza posta sullo strand (una delle estremità che compone l'elica del DNA) opposto e di conseguenza una loro sovrapposizione potrebbe richiedere un preprocessamento della stringa che la porti ad essere coerente con l'altra, operazione che prende il nome di *reverse and complement*.

3.2 Linee GFA1

GFA1 è la prima versione della specifica, essa si concentra nella descrizione delle sequenze, collegate tra loro da una relazione di *contenimento* o di *successione*.

Le linee previste dalla specifica sono header, segment, link, containment e path.

L'header è una riga il cui scopo è quello di indicare la versione della specifica in uso, può ripresentarsi più volte all'interno del file per indicare parametri opzionali validi per tutti gli elementi. Tale linea viene indicata con il simbolo `H`.

3.2.1 Segment

La linea di Segment (indicata con il simbolo `S`) descrive in termini generici una sequenza, la quale può essere definita o no. All'informazione viene attribuito un identificativo che deve essere unico in tutto in file. A queste proprietà se ne possono aggiungere altre, descritte da campi opzionali, tra i quali la lunghezza (`LN`) e il conto dei *k-meri* (l'insieme di tutte le possibili sottostringhe di lunghezza k contenute nella stringa).

Listato 3.1: Una possibile Segment line.

```
S       5       CCCGGGGTAA              LN:i:10
```

3.2.2 Link

I Link (indicati dal simbolo `L`) sono il principale tipo di relazione fra due sequenze. Essi indicano una sovrapposizione fra le sequenze indicate da due

Segment. Nello specifico, un Link fra due Segment indica che la parte terminale della prima sequenza è coinvolta in una sovrapposizione (*overlap*) con l'inizio della seconda; il termine che descrive esattamente questa situazione è *dovetail overlap* tradotta in sovrapposizione a coda di rondine. Tale tipologia di collegamento costituisce un'informazione di rilievo nell'analisi dell'assemblaggio poiché descrive un susseguirsi fra due sequenze.

Il link non solo descrive questa situazione, ma indica anche quale estremità della sequenza è coinvolta nel overlap. Ricordando che l'informazione contenuta nella struttura elicoidale del DNA è la stessa a parità di estremi (ma in senso inverso e complementata), è possibile che due sequenze siano contigue (in una situazione di dovetail overlap) considerando la loro provenienza da due estremità diverse dell'elica (strand). Il link permette di esprimere il collegamento considerando anche questa particolarità; per farlo esso utilizza un segno "`+`" per indicare che la sequenza non necessita di alcun processamento nel suo coinvolgimento nella sovrapposizione, mentre utilizza un segno "`-`" per esplicare la necessità di effettuare un' operazione di reverse and complement sulla sequenza prima di poterla considerare nel overlap.

Visto che, come si diceva poc'anzi, un Link descrive una sovrapposizione tra la fine della prima sequenza e l'inizio della seconda (indipendentemente dai segni associati alle due sequenze coinvolte), ciò da luogo a quattro possibili situazioni:

- la parte destra della prima sequenza si sovrappone con la parte sinistra della seconda (vedi figura 3.2a);
- la parte destra della prima sequenza si sovrappone con la parte destra della seconda (vedi figura 3.2b);
- la parte sinistra della prima sequenza si sovrappone con la parte sinistra della seconda (vedi figura 3.2c);
- la parte sinistra della prima sequenza si sovrappone con la parte destra della seconda (vedi figura 3.2d).

Oltre a questa considerazione sulle sequenze, il Link fornisce una descrizione dell'allineamento, dato da una stringa CIGAR. Una stringa CIGAR è una serie di lettere e numeri che descrivono lo stato di somiglianza fra le due sequenze. Tra i campi opzionali che il Link predispone si trova il campo `ID`, mediante il quale è possibile riferirsi a tale linea.

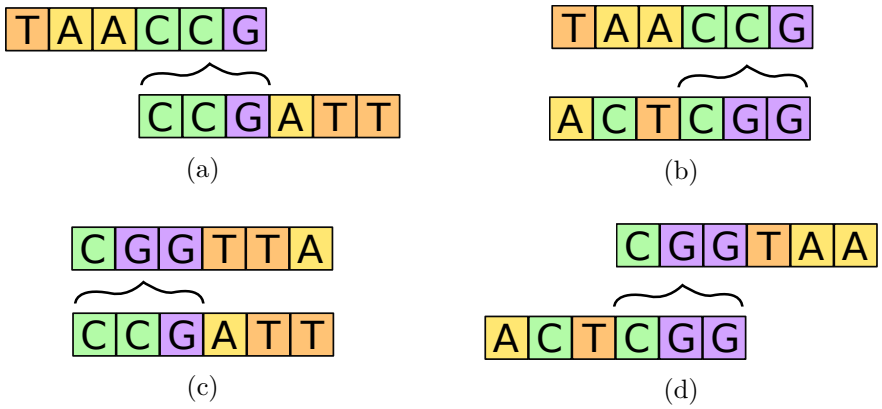

Figura 3.2: *(a)* Dovetail overlap senza bisogno di alterare le sequenze. *(b)* Dovetail overlap dove la seconda sequenza necessita di un'operazione di reverse and complement per sovrapporsi alla prima. *(c)* Un'operazione di reverse and complement deve essere eseguita sulla prima sequenza, affinché ci sia un overlap. *(d)* Entrambe le sequenze richiedono operazioni di reverse and complement.

3.2.3 Containment

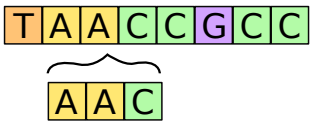

Figura 3.3: Una rappresentazione grafica della situazione di contenimento fra due sequenze.

Le linee di Containment (indicate con il simbolo `C`) descrivono sovrapposizioni fra sequenze nelle quali una stringa intera è contenuta nell'altra. I campi descrivono le stesse informazioni dei Link, ma è bene notare i dettagli circa le posizioni delle sequenze che una sovrapposizione di questo tipo comporta. Date due sequenze $s1$ ed $s2$, un Containment tra la sequenza $s1$ e la sequenza $s2$ indica che la sequenza $s1$ *contiene* la sequenza $s2$. In tale situazione vuole significare che la sovrapposizione comincia dal primo carattere della sequenza di $s2$ e continua fino all'ultimo (vedi figura 3.3).

Oltre ai classici campi che descrivono i nodi indicanti le sequenze coinvolte, il loro orientamento nella sovrapposizione e l'allineamento; queste linee hanno un campo che indica la posizione di inizio della sequenza contenuta nella sequenza contenitrice.

3.2.4 Path

Un Path (indicato dal simbolo `P`) descrive un susseguirsi di sequenze collegate esclusivamente da Link. Indica pertanto un percorso di sequenze contigue all'interno del grafo. Queste linee indicano esclusivamente gli identificativi e l'orientamento delle sequenze coinvolte nel percorso cui seguono l'insieme delle stringhe CIGAR relative l'allineamento delle sequenze prese a due a due.

Listato 3.2: Un esempio di file GFA 1.

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

3.3 GFA2

GFA2 come accennato in precedenza è un'estensione di GFA1, pensata per fornire più libertà all'utente circa le informazioni che è possibile descrivere. Le linee appartenenti a questa specifica non comprendono campi opzionali predefiniti, l'utente è libero di definire i campi aggiuntivi che più ritiene opportuni per la sua applicazione.

3.3.1 Segment

Queste linee sono analoghe ai Segment in GFA1, ai campi viene aggiunto un numero intero per descrivere la lunghezza della sequenza. La lunghezza non vuole essere l'esatta lunghezza della sequenza, ma vuole indicare la grandezza che tale sequenza assume quando rappresentata da un programma di disegno (come Bandage, descritto a pagina 20). Nell'indicare le sequenze non viene più richiesto l'uso di caratteri IUPAC[14], la sequenza può essere descritta con un qualsiasi carattere stampabile, nello specifico dal simbolo "`!`" al simbolo "`~`" della tabella ASCII.

3.3.2 Edge

La linea di Edge (indicata con la lettera `E`), indica un qualsiasi tipo di sovrapposizione. Essa quindi generalizza le linee di Link e Containment ed aggiunge la situazione in cui una generica parte di una sequenza è sovrapposta ad una qualsiasi parte (non solo agli estremi) di un'altra (come rappresentato in figura 3.4).

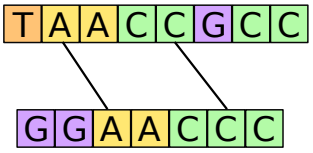

Figura 3.4: Una rappresentazione grafica di una generica di sovrapposizione fra sequenze.

Questa linea, come Link e Containment, fornisce gli identificatori delle sequenze coinvolte nel overlap e i rispettivi orientamenti, cui si aggiungono le *posizioni* di inizio e fine delle parti

delle rispettive sequenze sulle quali si svolge la sovrapposizione. La posizione è un intero che parte da 0 (descrivendo il primo carattere della sequenza) e termina in posizione pari alla lunghezza stessa della sequenza, all'ultima posizione della sequenza si pone il simbolo "`$`", non farlo costituirebbe un errore.

In questo modo si avrà che una situazione di dovetail overlap verrà indicata da un edge in cui le posizioni delle due sequenze sono $inizio1 = 0$ e $fine2 = y\$$ o $fine1 = x\$$ e $inizio2 = 0$; mentre una situazione di contenimento viene descritta da un edge in cui le posizioni delle sequenze sono $inizio1 = 0$ e $fine1 = x\$$ o $inizio2 = 0$ e $fine2 = y\$$. Si osservi che mentre un contenimento in GFA2 non impone alcun ordine circa la sequenza contenuta e quella contenitrice, un Containment in GFA1 prevede che la prima sequenza sia la contenitrice e la seconda sia la contenuta; inoltre in GFA2 non vi è alcun campo obbligatorio che indica l'inizio della sequenza contenuta, diversamente da GFA1.

Come in GFA1 è possibile specificare l'allineamento, non solo mediante stringa CIGAR, ma indicando una traccia DAZZLER (un indicatore per eseguire l'allineamento fra sequenze in un tempo quasi lineare). Quindi non solo GFA2 permette di descrivere la natura dell'allineamento tramite CIGAR string, ma anche di descrivere un modo veloce per calcolarlo usando le tracce DAZZLER. Come nelle altre situazioni delle specifiche, in caso di mancata informazione viene posto un asterisco in tale campo.

Questa generalizzazione delle possibili sovrapposizioni tra due sequenze permette di usare la specifica non solo per la descrizione di grafi di assemblaggio, come nel caso di GFA1; ma anche di rappresentare, in un unico formato, i risultati provenienti da diversi stadi del processo di assemblaggio.

3.3.3 Fragment

Le linee di Fragment (indicate con la lettera `F`) indicano un collegamento fra una sequenza indicata nel file e una sequenza presente in un file esterno. Il collegamento esprime un allineamento fra le due sequenze, in modo analogo ad un Edge.

3.3.4 Gap

Le linee di Gap (indicate con la lettera `G`) indicano uno spazio presente fra due sequenze, indicando la distanza che le separa e la varianza di tale supposizione.

3.3.5 Group

I gruppi in GFA possono essere di due tipi, gli OGroup (indicati con la lettera `O`) e gli UGroup (indicati con la lettera `U`). I primi indicano una sequenza ordinata di elementi GFA2 (escludendo gli UGroup) che individuano un

percorso all'interno del grafo, mentre gli UGroup indicano un insieme di elementi del grafo privi di ordine. Entrambi i gruppi descrivono un sottografo che è possibile ricavare dal grafo descritto dal file GFA.

Listato 3.3: Un esempio di file GFA 2.

```
S       1       122     *
S       3       29      TGCTAGCTGACTGTCGATGCTGTGTG
E       1_to_2  1+      2+      110     122$    0       12      12M
S       5       130     *
S       13      150     *
O       14      11+ 12+
S       11      140     *       xx:i:11
F       1       read1+  0       42      12      55      *       id:Z:read1_in_1
U       16      1 3 1_to_3
U       16sub   5 16
S       12      150     *
E       1_to_3  1+      3+      112     122$    0       12      10M
G       1_to_11 1+      11-     120     *
E       11_to_13        11+     13+     20      140$    0       120     120M
```

3.4 Conclusioni

In questo capitolo sono state esaminate le due versioni che costituiscono la specifica GFA, indicando lo scopo del quale ciascuna versione intende occuparsi e descrivendo i concetti che ciascuna linea vuole rappresentare nel contesto dell'assemblaggio del genoma. Nel prossimo capitolo si procederà nella descrizione del lavoro svolto nello sviluppo di *PyGFA* e di come si è dovuto procedere nella rappresentazione delle informazioni descritte nelle specifiche.

Capitolo 4

Sviluppo

In questo capitolo verrà descritto il processo di sviluppo seguito per l'implementazione di *PyGFA*, analizzandone le fasi principali, descrivendo i problemi incontrati e come sono stati affrontati. Verrà infine fornito un caso di esempio che mostra le funzionalità della libreria.

4.1 Processo

Lo scopo di *PyGFA* è quello di fornire un ambiente per lo sviluppo di applicativi in grado di analizzare e manipolare file GFA. Non è pertanto un prodotto finito, con casi d'uso definiti e risultati attesi con cui è possibile confrontare i risultati. Non si è ritenuto appropriato, di conseguenza, seguire un processo di sviluppo con fasi di analisi e pianificazione profonde, che con il mutare dei requisti (per la maggior parte non definiti fin dall'inizio) avrebbero potuto compromettere la struttura del sistema. Un altro fattore da tenere in considerazione è che le mie conoscenze sul significato che i dati contenuti nei file GFA e le considerazioni che si potevano dedurre da esse sono cresciute con lo sviluppo del sistema stesso. Perciò un'analisi, anche se non approfondita, non poteva essere svolta a priori poiché avrebbe potenzialmente comportato una serie di ritardi nello sviluppo del progetto dovute allo studio dei concetti biologici che avrebbe richiesto diverso tempo.

Per questi motivi il processo di sviluppo che si è deciso di utilizzare è di *extreme programming*; con fasi di analisi e pianificazione molto veloci, dando priorità all'implementazione delle parti essenziali del sistema aventi maggior priorità per poi ripetere il procedimento con l'evolversi dei requisiti e delle funzionalità richieste, cercando di avere un riscontro costante con i clienti finali (in questo caso i referenti della tesi).

Affiancata alla parte di implementazione si è svolta la parte di testing, che purtroppo non si è riusciti a condurre esattamente nella forma di Test Driven Development, ma alla quale è stata data comunque una priorità molto alta sia per la verifica delle funzionalità dei metodi che per la verifica della

presenza di errori nel codice, che nel contesto di un linguaggio non compilato, quale è Python, risulta una pratica molto importante per garantire il corretto funzionamento del programma.

Nel complesso si è riusciti a seguire abbastanza rigorosamente questi cicli di pianificazione veloce, implementazione e test; procedendo al termine di ogni ciclo con una fase di refactoring del codice e di miglioramento della documentazione presente in esso, avvalendosi di Pylint per individuare quelle porzioni di codice che era possibile migliorare.

4.1.1 Fasi di sviluppo

La rappresentazione delle informazioni contenute nei file GFA subisce diverse trasformazioni prima di giungere come dato di un nodo, arco o sottografo presente in un oggetto grafo GFA. L'iniziale rappresentazione testuale di ogni linea viene rappresentata da una classe che ne indica il tipo e i valori dei campi che contiene. Successivamente le linee vengono convertite in archi, nodi o sottografi e infine nodi e archi vengono rappresentati mediante dizionari Python, una volta inseriti effettivamente nel grafo. Quest'ultima trasformazione è stata adottata per uniformarsi al trattamento dei dati di un grafo in modo analogo a quello in cui NetworkX li gestisce, garantendo una facilità di accesso alle informazioni ed evitando che l'utente finale debba adattarsi ad un nuovo modo di operare.

É possibile suddividere lo sviluppo di *PyGFA* in tre fasi principali:

- sviluppo del parser;
- progettazione delle classi di astrazione dei dati GFA;
- sviluppo della classe del grafo GFA e delle operazioni che è possibile eseguire su di esso.

Il parser si occupa di leggere le linee di un file GFA, di verificare la correttezza sintattica dei suoi campi e di rappresentarne le informazioni mediante una classe specifica per ogni tipo di linea.

Nella seconda fase si sono analizzate le linee delle due specifiche e si è stabilito come attribuire ad ogni linea un ruolo che potesse essere di nodo, arco o sottografo del grafo GFA finale. Gli attributi delle classi del grafo astraggono gli attributi delle linee delle specifiche, di conseguenza è stata necessaria una pianificazione dell'assegnamento degli attributi delle linee ad attributi degli elementi del grafo.

Nella fase finale si è sviluppata la classe del grafo GFA fornendo metodi di inserimento, accesso ed eliminazione sui singoli elementi che lo compongono ed aggiungendo le interfacce agli algoritmi forniti da Networkx per eseguire le operazioni sui dati GFA.

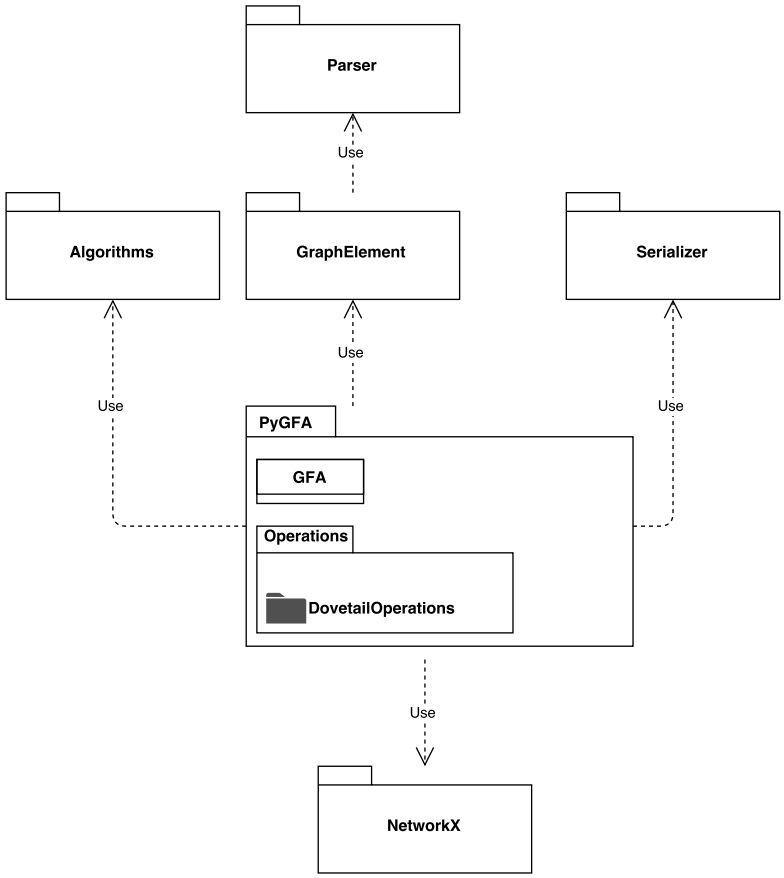

Figura 4.1: Diagramma dei package di *PyGFA*.

4.2 Fase1: sviluppo del parser

Per la scrittura del parser si è seguito un approccio *bottom-up*, sviluppando dapprima le classi rappresentanti i campi, che sono presenti in ogni linea, e successivamente descrivendo con una classe ciascun tipo di linea presente nelle specifiche.

Il parser effettua solo un controllo sintattico sulle informazioni dei file al fine di garantire una corretta gestione delle informazioni da parte della libreria; non verifica eventuali incongruenze tra le informazioni presenti. Per questo motivo si suppone che il file GFA che viene fornito sia stato già validato da un punto di vista di namespace degli elementi e di coerenza delle informazioni (per esempio il riutilizzo di un identificativo già utilizzato da un altro elemento o riferimenti a elementi che non vengono definiti).

Ogni campo di ogni linea, in entrambe le specifiche, può essere descritto da un *espressione regolare*. Per questo motivo è stato implementato un modulo Python per la validazione di tutti i campi definiti dalle specifiche,

associando un nome ad ogni espressione e creando un metodo `is_valid` che, data una stringa e il nome del tipo di un campo, verifica che la stringa rispetti l'espressione regolare indicata dal nome del campo fornito. Per rappresentare i campi delle linee sono state create due classi, `Field` e `OptField`; la prima descrive i campi obbligatori, per i quali non viene specificato esplicitamente il tipo di dato che contengono; la seconda descrive i campi opzionali per i quali sono forniti nome (il tag), tipo e valore. Mentre nei campi opzionali è possibile, fin dall'instanziazione dell'oggetto, effettuare una validazione sul contenuto, sui campi obbligatori non è possibile, in quanto la tipologia del loro contenuto assume valore solo nel contesto della linea cui appartengono.

Successivamente si è modellata la classe `Line` dalla quale derivano le classi rappresentanti le altre linee. Questa classe racchiude due campi: `PREDEFINED_OPTFIELDS` e `REQUIRED_FIELDS` che racchiudono i campi opzionali che ogni linea può contenere e i campi obbligatori che necessita, rispettivamente. Inoltre questa classe possiede i metodi di aggiunta e rimozione dei campi, assicurandosi di validare il contenuto dei campi obbligatori nel contesto di ciascuna linea. Le altre classi, derivando da questa, devono ridefinire i propri campi opzionali predefiniti e i campi obbligatori oltre a indicare un metodo per convertire una stringa nel corrispettivo oggetto che la rappresenta, condividendo la stessa logica di manipolazione e validazione dei campi che viene riutilizzata grazie al *polimorfismo*.

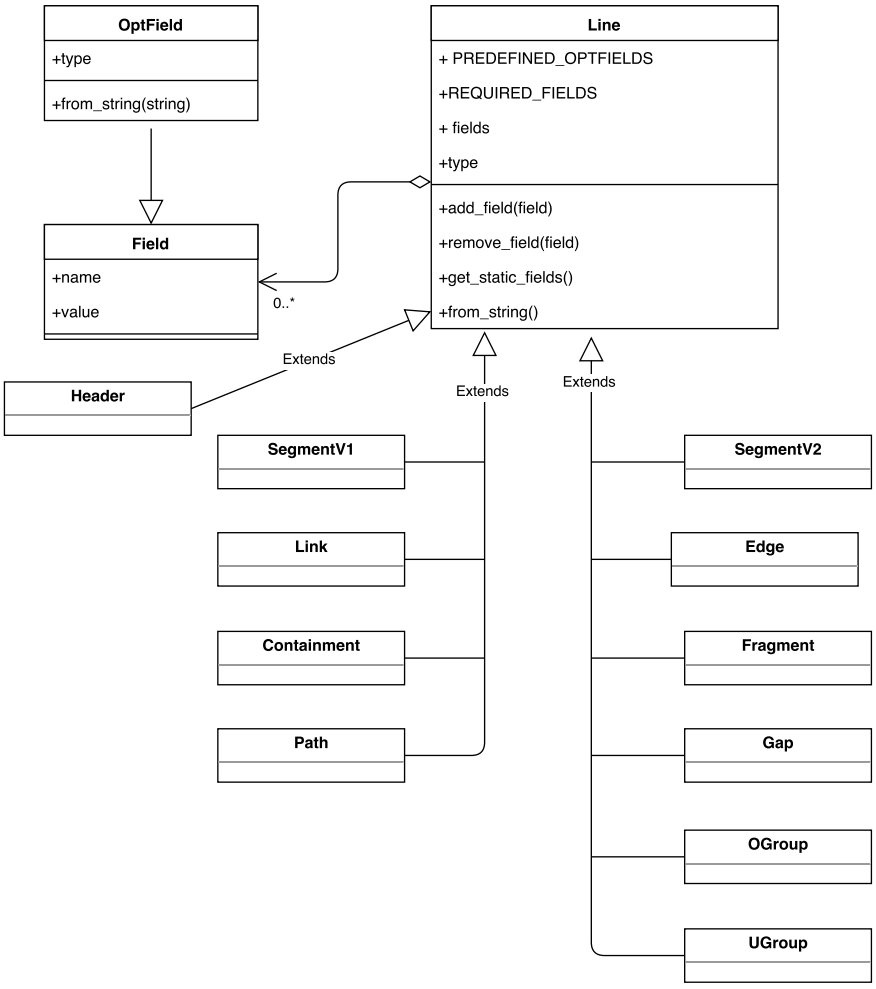

Figura 4.2: Diagramma delle classi usate dal parser.

4.3 Fase2: astrazione dei dati

In questa fase sono state analizzate le singole informazioni presenti nelle linee di entrambe le specifiche, evidenziandone gli aspetti simili al fine di giungere ad una loro rappresentazione generalizzata per mezzo di tre entità: *Nodo*, *Arco*, *Sottografo*.

Ciascuna linea GFA può essere facilmente associata ad uno di questi tre elementi. Nonostante sia possibile estendere GFA2 con nuovi tipi di linee, è bene notare che questo lavoro di assegnazione di un ruolo ad ogni linea limita questa funzionalità della specifica, in quanto sarebbe necessario capire il ruolo che queste nuove linee possono ricoprire nei grafi di assemblaggio e ridefinire i meccanismi con cui *PyGFA* riesce a manipolare queste nuove informazioni. É comunque possibile, vista la mancanza dei concetti di visibilità privata del linguaggio, andare ad inserire queste informazioni direttamente al grafo NetworkX che sta alla base dell'oggetto grafo GFA, ma non è possibile garantire la consistenza delle operazioni che è possibile effettuare usando la libreria.

Una sequenza costituisce il principale tipo di informazione cui si è interessati, più sequenze sono in relazione da una serie di collegamenti descritti dalle specifiche. Perciò possiamo attribuire alle sequenze (quindi alle linee Segment) il ruolo di nodi. La linea di header non contiene di per se informazioni che è possibile rappresentare su di un grafo. Al momento *PyGFA* non modella le informazioni che possono essere descritte dalle linee di header.

Tutte quelle linee che vedono come protagoniste due sequenze possono essere considerate come degli archi che collegano due nodi u e v. Appartengono a tale insieme le linee di link, containment, edge, fragment e gap.

Le linee rimanenti: path, ogroup e ugroup, rappresentano tutte un insieme (ordinato o meno) di nodi e archi che descrivono un sottografo composto dagli elementi del file GFA; di conseguenza tali linee vengono considerate come dei sottografi.

Visto che GFA2 è un superset di GFA1, si è analizzato come sarebbe stato possibile rappresentare ciascuna linea di GFA1 nel corrispettivo elemento di GFA2, capendo in questo modo gli attributi che ciascuna delle classi (Nodo, Arco e Sottografo) avrebbe dovuto contenere per modellare quelle informazioni. Terminato il confronto delle linee di GFA1 si sono aggiunte quelle informazioni delle linee di GFA2 che non erano state prese in considerazione (poiché non rappresentate) e le si sono andate ad aggiungere come attributi aggiuntivi delle classi rappresentanti gli elementi del grafo. Le informazioni che diventeranno attributi espliciti delle classi sono date esclusivamente dai campi obbligatori di ciascuna linea, per entrambe le classi i dati provenienti da campi opzionali verranno memorizzate all'interno di un dizionario Python e indicate da un unico campo `opt_fields`; si noti che nonostante il nome, i campi contenuti non sono (solamente) riferimenti ai campi opzionali delle

specifiche, ma si riferiscono ad una qualsiasi informazione non obbligatoria che l'utente vuole aggiungere all'elemento.

Di seguito verrà elencato per ciascun campo di ogni linea la relativa rappresentazione in GFA2 e la scelta di come si è deciso di modellare tale campo nell'elemento del grafo corrispondente. Per i nomi dei campi si farà riferimento alla nomenclatura utilizzata dalla specifica per indicare ciascun campo, si noti che alcuni nomi potrebbero subire variazioni in quanto la specifica è attivamente in sviluppo e subisce continui cambiamenti. In questa tabella si fa riferimento al nome dei campi così come si sono presentati al momento della fase dello sviluppo.

4.3.1 Attributi della classe Nodo

La classe nodo rispecchia senza aggiungere altre complicazioni i campi descritti dalla linea GFA2 Segment. Il campo `slen` che descrive la lunghezza della linea, nel caso di GFA1, viene recuperato dal campo opzionale `LN`; nel caso non fosse specificato si cerca di ricavare tale valore dalla sequenza stessa, calcolandone la lunghezza. Se la sequenza non è specificata il dato assume valore `None`, per indicare la mancanza di tale informazione.

Campo GFA1	**Campo GFA2**	**Attributo nodo**
Name	sid	nid (node id)
Sequence	sequence	sequence
Campo opzionale LN, lunghezza linea o **None**	slen	slen

Tabella 4.1: Tabella di analisi degli attributi del nodo.

4.3.2 Attributi della classe Arco

Per determinare gli attributi della classe arco si sono dovuti analizzare i campi di tutte le linee che potessero essere riconducibili ad un arco del grafo.

La prima linea da analizzare è stata la linea Link della specifica GFA1, la quale è possibile ricondurla alla linea Edge di GFA2. Si nota dalla tabella 4.2 l'esistenza di una corrispondenza uno a uno tra i campi della linea Link e quelli della linea Edge. I campi della linea Edge contengono anche le posizioni delle sequenze nelle quali si verifica l'overlap, ma tale informazione non è presente nel Link di conseguenza questi quattro attributi dell'arco rappresentante un link (beg1, end1, beg2 ed end2) saranno impostati a None.

Lo stesso comportamento è stato usato con la linea di Containment; in questo caso però il campo relativo la posizione di inizio della sequenza con-

Campo GFA1	**Campo GFA2**	**Attributo arco**
Campo opzionale ID o **None**	eid	eid (edge id)
From	sid1 (escludendo il segno)	from_node
From Orientation	segno di sid1	from_orn
To	sid2 (escludendo il segno)	to_node
To Orientation	segno di sid2	to_orn
Alignment	alignment	alignment

Tabella 4.2: Tabella di analisi degli attributi della linea Link.

tenuta non è indicata nell'equivalente linea GFA2 Edge, e tale informazione si è rivelata deducibile dalle posizioni di inizio e fine dell'overlap indicata dal Edge stesso, di conseguenza a questa informazione è stata attribuita una priorità minore e si è scelto di inserirlo come ulteriore campo opzionale dell'arco (chiamandolo `pos`), in modo da non perdere l'informazione nella fase di serializzazione del grafo GFA in formato testuale.

Analizzando le altre linee rimanenti in GFA2 (Edge, Gap e Fragment), oltre agli attributi necessari a rappresentare i campi di GFA1, si sono aggiunti gli attributi per rappresentare l'inizio e la fine delle posizioni che coinvolgono l'overlap rispettivamente per la sequenza di partenza e di arrivo, inoltre l'analisi sulle linee di Gap ha richiesto l'aggiunta di attributi relativi la varianza e la distanza tra due gruppi di sequenze che questa linea descrive. L'unica particolarità da evidenziare è la mancanza di un campo analogo ad `eid` per la linea Fragment.

Si noti che le informazioni, con questo livello di astrazione, rendono difficile distinguere le linee di Link, da quelle di Edge e di Fragment (le linee di Containment si distinguono per la presenza del campo `pos` all'interno dei campi opzionali dell'arco mentre quelle di Gap hanno sempre definiti gli attributi di varianza e distanza che le altre linee hanno impostate a **None**). In questo caso la distinzione fra queste linee avviene come segue: gli Edge e i Link avranno il simbolo di asterisco come identificativo nel caso l'informazione sia mancante, mentre i Fragment non hanno alcun campo che descrive un identificatore che li referenzia, perciò il loro campo `eid` sarà None. Fatta questa distinzione, le linee di Edge se necessario possono essere distinte da quelle di Link per la mancanza delle informazioni circa le posizioni del overlap fra le sequenze che i Link descrivono.

In aggiunta agli attributi dei campi, si è deciso di aggiungere altre tre informazioni per ogni arco. Queste informazioni riguardano nello specifico i Link e gli Edge che rappresentano un dovetail overlap (cioè quegli Edge che descrivono un Link). Per questo tipo di archi viene impostato un attributo

booleano `is_dovetail` che viene posto a vero e successivamente con i valori degli orientamenti delle sequenze e delle posizioni dell'overlap (per gli Edge) vengono impostati due campi `from_segment_end` e `to_segment_end` che indicano l'estremità delle sequenze ("from" e "to", rispettivamente) che vengono prese in considerazione dall'overlap. Nel caso degli archi che non presentano questa situazione, l'attributo `is_dovetail` è impostato a falso e gli attributi riguardanti le estremità sono posti a None.

Questa scelta ha permesso di andare ad effettuare tutta una serie di operazioni sul grafo di notevole importanza ai fini dell'assemblaggio del genoma, visto che le sovrapposizioni di dovetail rappresentano un continuum fra sequenze. Se non si fosse adoperata questa soluzione il livello di astrazione introdotto da *PyGFA* nella rappresentazione dei dati sarebbe stato troppo elevato e non avrebbe permesso di usare efficacemente tutta una serie di operazioni che avrebbero avuto senso solo per collegamenti di questo tipo.

4.3.3 Attributi della classe Sottografo

Le linee che sono rappresentabili dalla classe Sottografo sono i Path, gli OGroup e gli UGroup.

Gli OGroup sono l'equivalente GFA2 dei Path, indicando ogni elemento e il suo orientamento, a differenza degli UGroup i quali non indicano il segno degli elementi che lo compongono.

Visto che i due gruppi in GFA2 indicano un eventuale overlap direttamente negli elementi che lo compongono, il campo `overlaps` del Path è stato inserito (in modo analogo al campo `pos` del containment) nei campi opzionali della classe, in modo che sia possibile senza ulteriori operazioni ricondurre il dato alla sua descrizione testuale in formato GFA1.

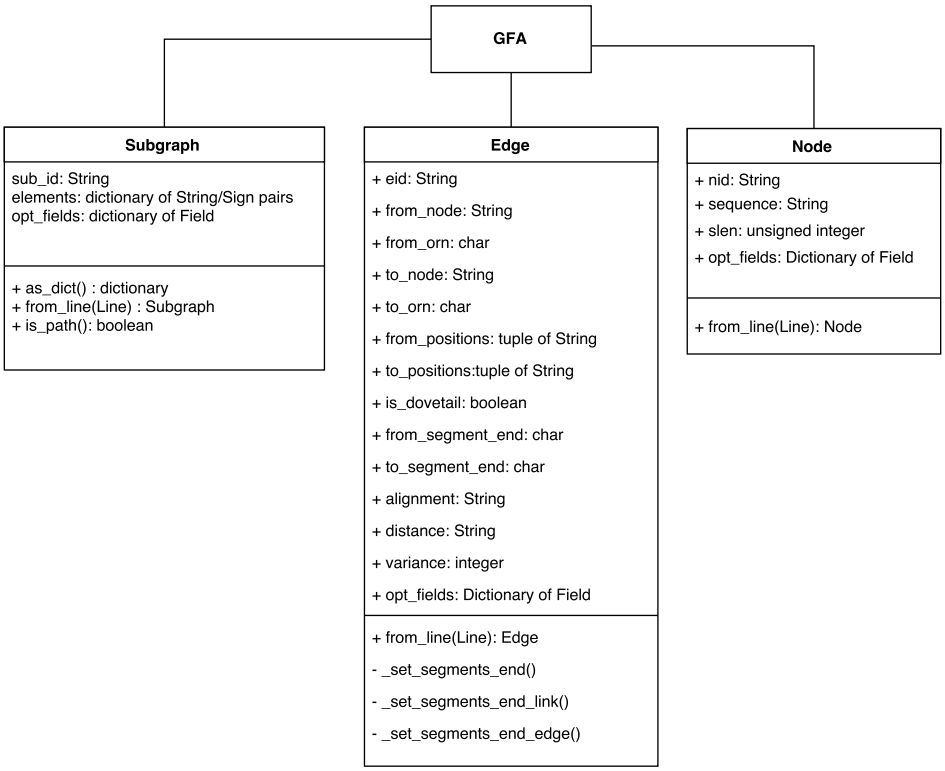

Figura 4.3: Diagramma delle classi degli elementi del grafo.

4.4 Fase3: rappresentazione del grafo GFA

La classe che modella il grafo GFA sfrutta la classe Multigraph offerta dalla libreria NetworkX come struttura ospitante gli archi e i nodi individuati dalla fase precedente. Il multigrafo è stato scelto in quanto permette di inserire più archi tra due nodi u e v, inoltre le relazioni presenti fra le sequenze non esprimono un senso di direzionalità degli archi, perciò si è usato un grafo non diretto. A nodi e archi è possibile inserire un qualsiasi tipo di riferimento ad un oggetto specificando in fase di inserimento l'attributo del nodo al quale collegare il valore definito; la libreria salverà l'informazione in un dizionario, di conseguenza il reperimento del valore riferito al campo del nodo avverrà in modo analogo all'accesso ad un dizionario Python.

Per uniformarsi a tale comportamento *PyGFA* in fase creazione di nodi ed archi estrapola gli attributi del elemento del grafo da inserire (in caso di nodo o arco) e li inserisce sotto forma di coppia indice-valore di un dizionario. In questo modo è possibile accedere facilmente ai diversi attributi di nodi e archi; inoltre tale scelta è stata preferita in quanto questi elementi non hanno un comportamento, ma rappresentano solamente un insieme di informazioni con relativi metodi di accesso. di conseguenza sarebbe stato inappropriato descriverli mediante una classe. É bene notare che ciò non vuole significare che le classi che modellano gli elementi del grafo ricoprano un ruolo minore, esse sono servite ad astrarre informazioni comuni ad elementi diversi tra loro riducendo quella complessità che altrimenti si avrebbe avuto al momento di inserire nodi e archi nel grafo GFA. I sottografi sono invece contenuti in un dizionario a parte, separato dalla struttura a triplice dizionario rappresentata in figura 2.2 a pagina 15; il dizionario in questo caso contiene direttamente riferimenti ad oggetti Subgraph i quali vengono convertiti in dizionari in base alle necessità dei metodi o dell'utente grazie al metodo `as_dict()`.

Per l'implementazione del grafo GFA si è scelto di usare la composizione al posto di sfruttare l'ereditarietà della classe multigrafo. Tale scelta si è preferita per un semplice concetto logico: la classe GFA *sfrutta* un multigrafo senza volerne emulare le funzionalità. Di conseguenza alla classe sono stati forniti i metodi di interfaccia alla classe multigrafo oltre ad un metodo di accesso agli archi fornendo solamente l'identificativo di un arco.

4.4.1 Iteratore sugli archi di dovetail

Come accennato a pagina 25 le sovrapposizioni che si sviluppano tra la fine di una sequenza e l'inizio di un'altra sono informazioni di rilievo nel contesto dell'assemblaggio del genoma, di conseguenza si è cercato di inserire una serie di meccanismi in *PyGFA* che permettessero di distinguere gli archi che possiedono il valore `is_dovetail` a true, visto che NetworkX non fornisce una soluzione a questa necessità (come descritto a pagina 15).

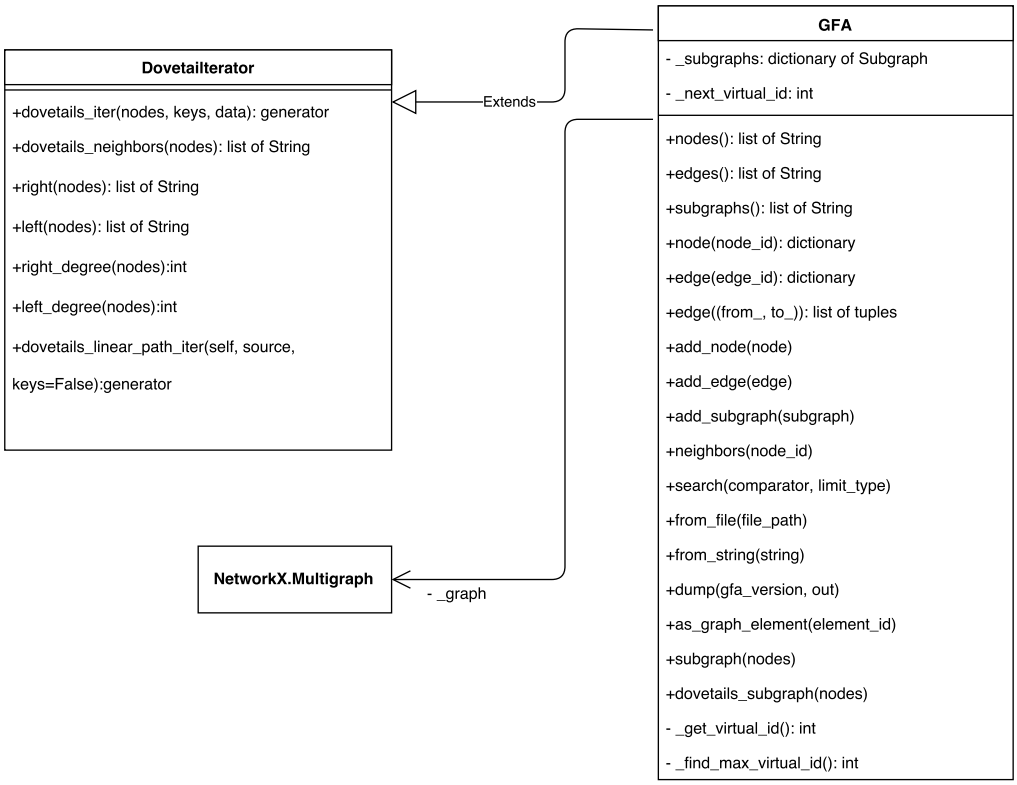

Figura 4.4: Diagramma delle classi del grafo GFA.

Questa funzionalità è stata implementata dalla classe `DovetailIterator` che fornisce una serie di iteratori per scorrere i soli nodi del grafo collegati da archi di dovetail overlap.

4.5 Operazioni sul grafo

PyGFA offre una serie di operazioni che è possibile applicare sul grafo composto da sequenze (nodi) e relazioni fra due sequenze (archi), dividendole in due gruppi applicativi che si distinguono per le modalità in cui gli archi vengono considerati nell'esecuzione delle operazioni.

Il primo insieme considera tutti gli archi del grafo, indistintamente dall'essere archi di edge, fragment, gap o altro tipo. Queste operazioni sono di utilità generale e servono per avere la massima flessibilità nella selezione delle informazioni che il grafo contiene.

In questo gruppo troviamo il metodo della classe GFA `search` che restituisce l'insieme di tutti gli elementi del grafo per i quali la funzione fornita tra i parametri assume valore true. In questo modo l'utente definendo un comparatore può effettuare una serie di operazioni di ricerca e filtraggio sul grafo.

Inoltre, grazie all'uso della libreria NetworkX, è stato possibile con estrema facilità creare delle interfacce ai metodi per l'individuazione delle componenti connesse, sia per quanto riguarda la ricerca della componente connessa contente un nodo dato, che la ricerca di tutte le componenti connesse presenti nel grafo.

Considerando l'alto livello di astrazione delle informazioni contenute nel grafo, per l'utente che intende utilizzare *PyGFA* come strumento di aiuto nella ricostruzione del genoma, è sconveniente andare ad eseguire operazioni considerando archi che non sono di dovetail overlap, visto che la relazione che essi descrivono è quella di maggior rilievo ai fini dell'assemblaggio. Di conseguenza *PyGFA* come gfapy offre una serie di operazioni che considerano esclusivamente archi di dovetail overlap.

4.5.1 Operazioni sugli archi di dovetail overlap

Gli algoritmi disponibili con NetworkX non permettono l'attraversamento del grafo considerando le proprietà degli archi, non è quindi possibile direttamente usare i metodi forniti come invece fatto per le operazioni che considerano l'intero grafo (come già discusso a pagina 15, quando sono stati presentati i limiti della libreria). Nonostante questo inconveniente, vista la natura open source di NetworkX e la licenza favorevole alla sua modifica e distribuzione senza vincoli, è stato possibile modificare gli algoritmi necessari per l'implementazione delle operazioni richieste. In tutti i casi l'unica modifica richiesta per ottenere il risultato atteso consisteva nel modificare l'elenco dei nodi considerati in fase di iterazione dell'algoritmo, non più considerando l'intera lista di adiacenza del nodo selezionato al "livello" corrente, ma prendendo in considerazione i nodi adiacenti collegati da un arco di dovetail overlap. Visto che l'iteratore personalizzato per l'attraversamento di questi archi è stato già sviluppato nella classe DovetailIterator ed esteso

dalla classe GFA, i metodi di iterazione sono già forniti direttamente con il grafo GFA.

Listato 4.1: BFS in networkx.

```
1  def _plain_bfs(G, source):
2    G_adj = G.adj
3    seen = set()
4    nextlevel = {source}
5      while nextlevel:
6        thislevel = nextlevel
7        nextlevel = set()
8        for v in thislevel :
9          if v not in seen:
10           yield v
11           seen.add(v)
12           nextlevel.update(G_adj[v])
```

Listato 4.2: BFS in *PyGFA*.

```
1  def _plain_bfs_dovetails(gfa_, source):
2    if source not in gfa_:
3      return ()
4    seen = set()
5    nextlevel = {source}
6    while nextlevel:
7      thislevel = nextlevel
8      nextlevel = set()
9      for v in thislevel :
10       if v not in seen:
11         yield v
12         seen.add(v)
13         nextlevel.update(gfa_.right(v))
14         nextlevel.update(gfa_.left(v))
```

I due listati rappresentano l'implementazione dell'algoritmo BFS (Breadth First Search) presente nella libreria NetworkX (listato 4.1) e l'equivalente implementato in *PyGFA* (listato 4.2). É possibile notare come l'unica differenza fra i due listati è presente nella selezione dei nodi da considerare per la prossima iterazione, nelle ultime righe di entrambi i listati; mentre in NetworkX viene considerata l'intera lista di adiacenza, in *PyGFA* si sfruttano gli iteratori definiti in DovetailIterator per selezionare gli archi di dovetail overlap presenti alle estremità del nodo. Questo rappresenta una semplice modifica che si è dovuta effettuare per implementare le operazioni di dovetail overlap, in alcuni casi le modifiche hanno richiesto più tempo e si è dovuta effettuare un'analisi completa dell'operato dell'algoritmo (come nel caso

dell'algoritmo per la ricerca dei percorsi semplice fra due nodi), ma nella maggior parte dei casi le modifiche richieste sono state semplici ed intuitive.

Effettuando le modifiche sopra descritte è stato possibile sviluppare le seguenti operazioni sugli archi di dovetail overlap:

- ricerca delle componenti connesse;
- rimozione delle componenti connesse con lunghezza totale delle sequenze al di sotto di un determinato valore di soglia;
- rimozione delle estremità "morte", cioè tutti i nodi con grado di ingresso o uscita minore o uguale a 1 e la cui rimozione non causa la divisione di una componente connessa;
- ricerca degli insiemi di percorsi formati da unitig, cioè sequenze le cui estremità sono collegate ad una sola altra sequenza;
- ricerca di tutti i percorsi praticabili per arrivare da un nodo u ad un nodo v.

4.6 Esempio

Si andrà ora ad illustrare le funzionalità di *PyGFA* considerando il file GFA2 contenente le informazioni nel listato 4.3, che visualizzato con la libreria Matplotlib (strumento usato per la visualizzazione dell'intero grafo) produce il risultato in figura 4.5; si sottolinea che tale libreria non considera il contesto biologico associato alle informazioni, ma visualizza solamente gli elementi del grafo e la loro disposizione. Per visualizzare il grafo con Bandage, in grado di visualizzare solo archi Link di GFA1, il file GFA1 equivalente è stato ottenuto usando la funzione `dump` di *PyGFA*, la quale è in grado di convertire le informazioni del grafo GFA in una delle due specifiche. Il risultato finale è in figura 4.6, si può notare come i Gap (per esempio quello tra s8 ed s11) non sono visualizzati da questo strumento.

Listato 4.3: Il file GFA2 usato per l'esempio.

```
S       s1      10      *
S       s2      10      *
S       s3      10      *
S       s4      10      *
S       s5      10      *
S       s6      10      *
S       s7      10      *
S       s8      10      *
S       s9      10      *
S       s10     10      *
S       s11     10      *
S       s12     10      *
S       s13     10      *
S       s14     10      *
S       s15     10      *
S       s16     10      *
E       ls1s2   s1+     s2+     7       9$      0       2       *
E       ls2s4   s2+     s4+     7       9$      0       2       *
E       ls1s3   s1+     s3+     7       9$      0       2       *
E       ls3s5   s3+     s5+     7       9$      0       2       *
E       ls4s6   s4+     s6+     7       9$      0       2       *
E       ls5s6   s5+     s6+     7       9$      0       2       *
E       ls6s7   s6+     s7+     7       9$      0       2       *
E       ls7s8   s7+     s8+     7       9$      0       2       *
E       ls9s10  s9+     s10+    7       9$      0       2       *
E       ls11s12 s11+    s12+    7       9$      0       2       *
E       ls12s13 s12+    s13+    7       9$      0       2       *
E       ls14s15 s14+    s15+    7       9$      0       2       *
E       ls15s16 s15+    s16+    7       9$      0       2       *
E       ls16s14 s16+    s14+    7       9$      0       2       *
G       gs1s9   s1+     s9+     15      *
G       gs8s11  s8+     s11+    15      *
```

Caricato il grafo estraendo le informazioni dal file `gfa-es` si andrà a presentare il modo in cui queste vengono rappresentate mediante dizionari, si procederà successivamente con la ricerca delle componenti connesse. Si andrà ad illustrare l'operato degli iteratori personalizzati cercando i segmenti collegati alle estremità di un nodo specifico mediante l'uso i metodi `left` e `right`. Si procederà quindi con le operazioni sugli archi di dovetail

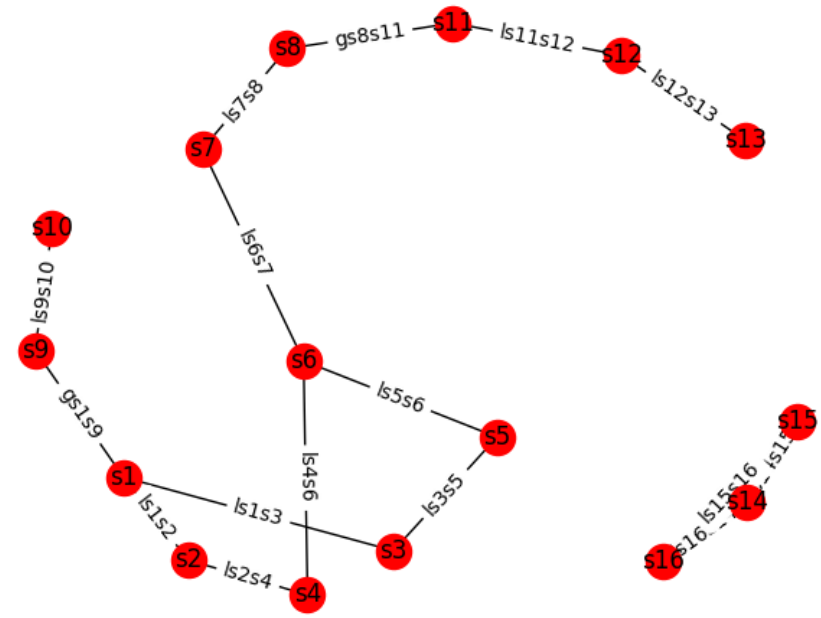

Figura 4.5: Visualizzazione del grafo con matplotlib.

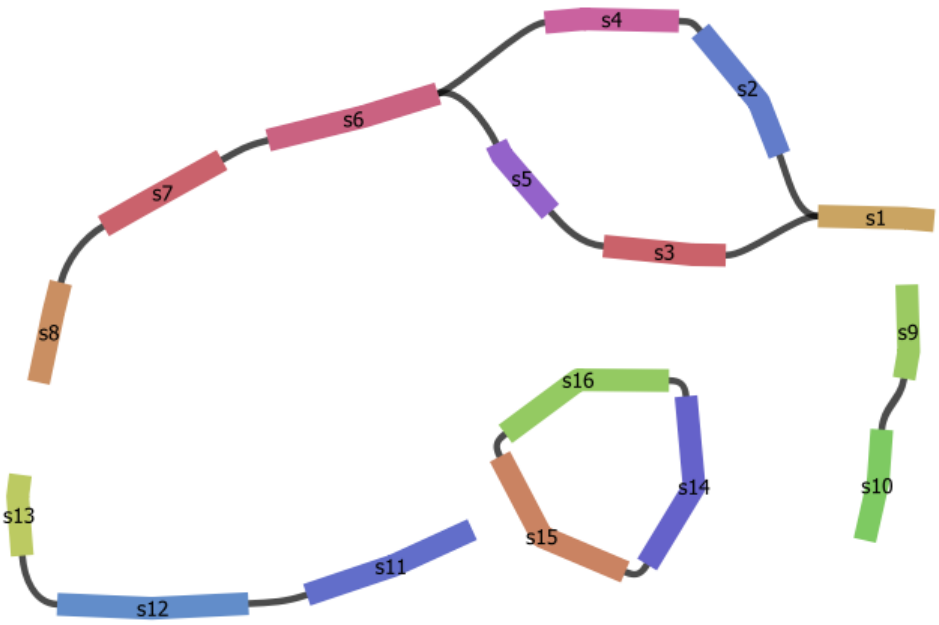

Figura 4.6: Visualizzazione del grafo con Bandage.

individuando le componenti connesse e comparandole con il risultato precedentemente ottenuto, per poi concludere con i metodi per la ricerca delle sequenze di unitig e dei cammini semplici che si diramano tra due nodi u e v.

```
>>> import pygfa
>>> g = pygfa.gfa.GFA.from_file("..\..\gfa_es.gfa")
>>> g.nodes()
['s1', 's2', 's3', 's4', 's5', 's6', ..., 's15', 's16']
>>> g.node("s1")
{'nid': 's1', 'sequence': '*', 'slen': 10}
>>> g.edges(keys=True)
[('s1', 's2', 'ls1s2'), ('s1', 's3', 'ls1s3'), ..., ('s15', 's16', 'ls15s16')]
>>> g.edge(("s1", "s2"))
{'ls1s2': {'eid': 'ls1s2',
       'from_node': 's1', 'from_orn': '+', 'to_node': 's2', 'to_orn': '+',
       'from_positions': ('7', '9$'), 'to_positions': ('0', '2'),
       'alignment': '*', 'distance': None, 'variance': None,
       'is_dovetail': True, 'from_segment_end': 'R', 'to_segment_end': 'L'}}
>>> g.edge("ls1s2")
{'eid': 'ls1s2', 'from_node': 's1', 'from_orn': '+', 'to_node': 's2', 'to_orn': '+',
       'from_positions': ('7', '9$'), 'to_positions': ('0', '2'),
       'alignment': '*', 'distance': None, 'variance': None,
       'is_dovetail': True, 'from_segment_end': 'R', 'to_segment_end': 'L'}
>>> g.edge("ls1s2")["is_dovetail"]
True
```

La conversione degli elementi del grafo in dizionari ha permesso una migliore integrazione nella libreria NetworkX facilitando l'accesso alle informazioni del grafo. Di contro tale comportamento non permette di ricalcolare automaticamente le informazioni della linea rappresentata in caso di modifica, cioè non avendo una classe non è possibile attribuire un comportamento nella gestione delle informazioni. Questo vuole significare che l'utente è libero di modificare le informazioni, ma è suo compito assumersi la responsabilità che le modifiche siano coerenti con il resto dei dati.

```
>>> pygfa.nodes_connected_components(g)
<generator object connected_components at ...>
>>> le componenti connesse vengono restituite
... # come generatore python
... # una componenente alla volta.
... # Per poterle visualizzare
... # tutte in una unica soluzione e'
... # necessario inserirle in una lista Python
...
>>> list(pygfa.nodes_connected_components(g))
[
       {'s4', 's13', 's10', 's1', 's2',
              's3', 's9', 's5', 's8', 's12',
              's11', 's7', 's6'},
       {'s14', 's15', 's16'}
]
```

I due insiemi presenti all'interno della lista individuano le due componenti connesse come mostra la figura 4.5. Tutti i tipi di archi che collegano due nodi vengono presi in considerazione in tale operazione.

```
>>> g.node("s1")
{'nid': 's1', 'sequence': '*', 'slen': 10}
>>> g.neighbors("s1")
['s2', 's3', 's9']
>>> g.right("s1")
['s2', 's3']
>>> g.left("s1")
[]
>>> g.dovetails_neighbors("s1")
['s2', 's3']
```

Dato il nodo $s1$, i nodi adiacenti considerando tutti i tipi di archi sono dati dal metodo `neighbors`, mentre le operazioni successive operano considerando solo archi di dovetail. Il metodo `left` restituisce tutte le seguenze in cui è presente un dovetail overlap con la parte sinistra della sequenza selezionata ($s1$ in questo caso), mentre il metodo `right` restituisce le sequenze in cui la sovrapposizione avviene con la parte destra della sequenza. I nodi adiacenti ad $s1$, considerando solo questo tipo di collegamenti fra nodi, sono un sottoinsieme dei nodi individuati dal metodo `neighbors`.

```
>>> list(pygfa.dovetails_nodes_connected_components(g))
[
       {'s4', 's2', 's1', 's3', 's5', 's6', 's8', 's7'},
       {'s9', 's10'},
       {'s13', 's12', 's11'},
       {'s16', 's14', 's15'}
]
```

Le componenti connesse considerando gli archi di dovetail sono più numerose rispetto quelle trovate senza fare distinzione fra i legami dei nodi. Infatti si può notare che gli archi di gap $(s9, s1)$ e $(s8, s13)$ non sono presi in considerazione, facendo così aumentare il numero totale delle componenti connesse.

```
>>> g.dovetails_linear_path_iter("s3")
<generator object DovetailIterator.dovetails_linear_path_traverse_edges_iter at ...>
>>> list(g.dovetails_linear_path_iter("s3"))
[('s3', 's5')]
>>> list(pygfa.dovetails_linear_paths(g))
[
       [('s4', 's2')],
       [('s15', 's14'), ('s14', 's16')],
       [('s13', 's12'), ('s12', 's11')],
       [('s9', 's10')],
       [('s5', 's3')],
       [('s7', 's8')]
]
>>> list(pygfa.dovetails_linear_paths(g, keys=True))
[
       [('s4', 's2', 'ls2s4')],
       [('s15', 's14', 'ls14s15'), ('s14', 's16', 'ls16s14')],
       [('s13', 's12', 'ls12s13'), ('s12', 's11', 'ls11s12')],
       [('s9', 's10', 'ls9s10')],
       [('s5', 's3', 'ls3s5')],
       [('s7', 's8', 'ls7s8')]
]
```

L'iteratore sulle sequenze di unitig è un metodo della classe GFA che restituisce gli archi che costituiscono il percorso lineare contenente il nodo indicato

come parametro. La funzione per ricercare tutti i percorsi lineari sfrutta gli iteratori sugli unitig per cercare l'insieme di tutte le sequenze di unitig presenti nel grafo; alle tuple di nodi che compongono gli archi che denotano i percorsi lineari è possibile associare anche il nome specifico degli archi, in modo da distinguerli in presenza di più archi tra due nodi.

```
>>> list(pygfa.dovetails_all_simple_paths(g, "s2", "s7"))
[
       ['s2', 's1', 's3', 's5', 's6', 's7'],
       ['s2', 's4', 's6', 's7']
]
>>> list(pygfa.dovetails_all_simple_paths(g, "s2", "s7", edges=True, keys=True))
[
       [('s2', 's1', 'ls1s2'), ('s1', 's3', 'ls1s3'),
              ('s3', 's5', 'ls3s5'), ('s5', 's6', 'ls5s6'),
              ('s6', 's7', 'ls6s7')],
       [('s2', 's4', 'ls2s4'), ('s4', 's6', 'ls4s6'),
              ('s6', 's7', 'ls6s7')]
]
>>> list(pygfa.dovetails_all_simple_paths(g, "s2", "s11"))
[]
```

Nell'esempio si vogliono trovare tutti i percorsi che congiungono il nodo $s2$ al nodo $s7$, la funzione ritorna le liste dei nodi che individuano i percorsi di congiunzione. Impostando l'attributo `edges` a True la funzione ritorna le sequenze di archi dei percorsi. Nel caso non sia presente un percorso fra i due nodi specificati, la funzione ritorna un generatore privo di elementi, che viene convertito dalla funzione `list` in una lista vuota.

Capitolo 5

Benchmark

In questo breve capitolo verranno mostrati alcuni risultati di *PyGFA* evidenziando il tempo di calcolo richiesto per alcune delle operazioni che la libreria fornisce e misurando lo spazio di memoria richiesto per l'immagazzinamento del grafo.

Tutte le operazioni sono state effettuate su una macchina virtuale Linux 64bit con le seguenti caratteristiche:

- 8 GB di ram;
- processore host da 3.7 GHz quad-core, il sistema guest sfrutta due soli core;
- hard disk magnetico.

Le misure sono state effettuate su grafi generati casualmente da uno script usato nello sviluppo di gfapy; il programma crea una serie di nodi collegati tra loro da sovrapposizioni a coda di rondine, il risultato viene salvato in un file GFA1. Visto che i grafi sono generati casualmente è improbabile che si riesca a ripresentare la stessa configurazione di componenti connesse e sequenze di unitig (a meno di salvare i file generati), mentre il numero di nodi e archi rimane prestabilito al ripetersi dei test. Per *elementi del grafo*, nel contesto dei grafici presentati, si intendono l'insieme di nodi e archi del grafo. Per automatizzare la misurazione delle performance è stato scritto un programma che si occupa di generare automaticamente i grafi, effettuare le misurazioni, salvare i risultati su un file di testo e rimuovere i grafi creati al termine.

Le operazioni che si sono andate a misurare sono:

- il tempo impiegato per effettuare il parsing del file contenente il grafo;
- il tempo richiesto per ottenere i dizionari contenenti i nodi e gli archi;

- il tempo per calcolare le componenti connesse (sia considerando solo archi di dovetail sia considerando tutti i tipi di archi) [1];
- il tempo richiesto per la ricerca delle sequenze di unitig.

Oltre alla misura di questi tempi si è andato a misurare il consumo di memoria necessario per contenere il grafo GFA.

[1] In questo caso, visto che lo script genera solo archi a coda di rondine, le componenti connesse individuate saranno le medesime. Si tenga comunque a mente che il tempo misurato riguarda il calcolo di entrambi i tipi di componenti connesse.

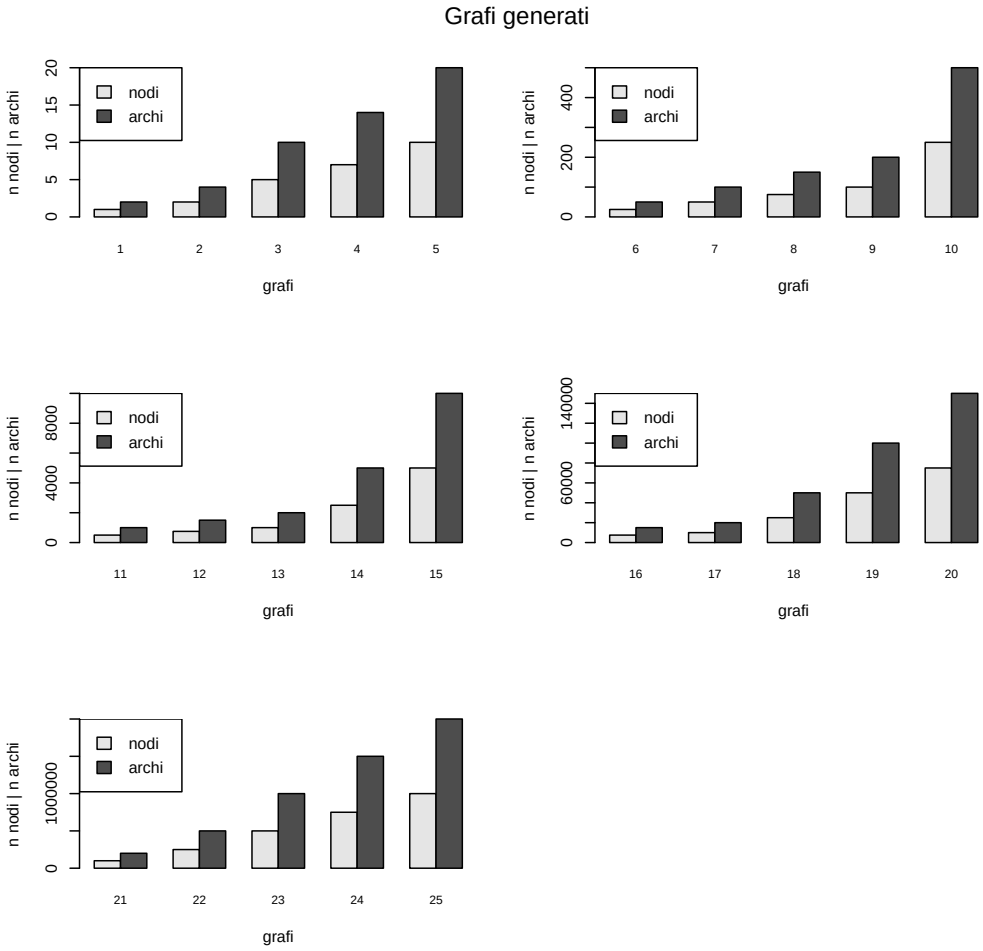

Figura 5.1: Grafici a barre relativi i numeri di nodi e archi dei grafi generati.

Per il numero dei nodi dei grafi generati si sono prese in considerazione le potenze di dieci generando, all'aumentare dell'esponente, grafi il cui numero di nodi ripercorreva i fattori 1, 2.5, 5, 7.5 e 10. Si sono creati quindi 25 grafi, raggiungendo la soglia del milione di nodi, collegati tra loro da due milioni di archi.

I tempi (figura 5.2) indicano un andamento lineare per tutte le operazioni eseguite. Il caso peggiore si ha nel caso della funzione di ricerca delle sequenze di unitig, la quale impiega un tempo doppio rispetto la ricerca delle componenti connesse. I tempi per la lettura del file sono lineari e strettamente vincolati dal tempo di accesso del supporto magnetico, si può pensare di ottenere un risultato decisamente migliore usando un SSD, visto che ha un tempo accesso più veloce.

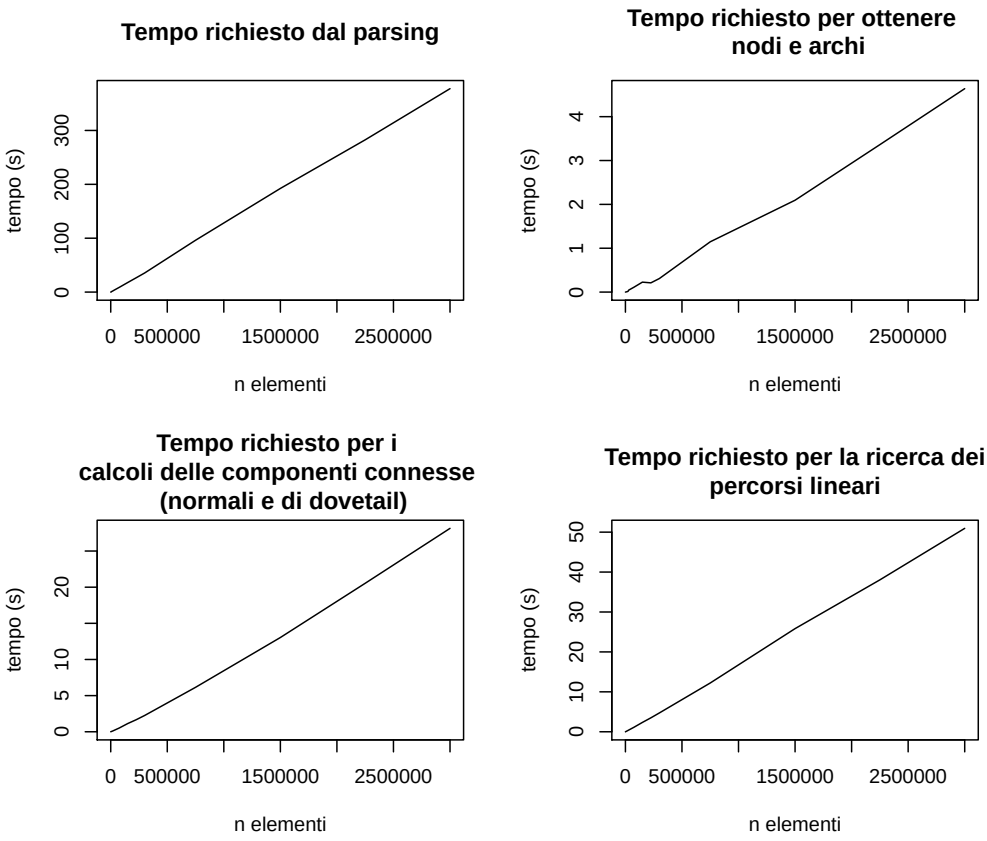

Figura 5.2: Misurazioni dei tempi di calcolo.

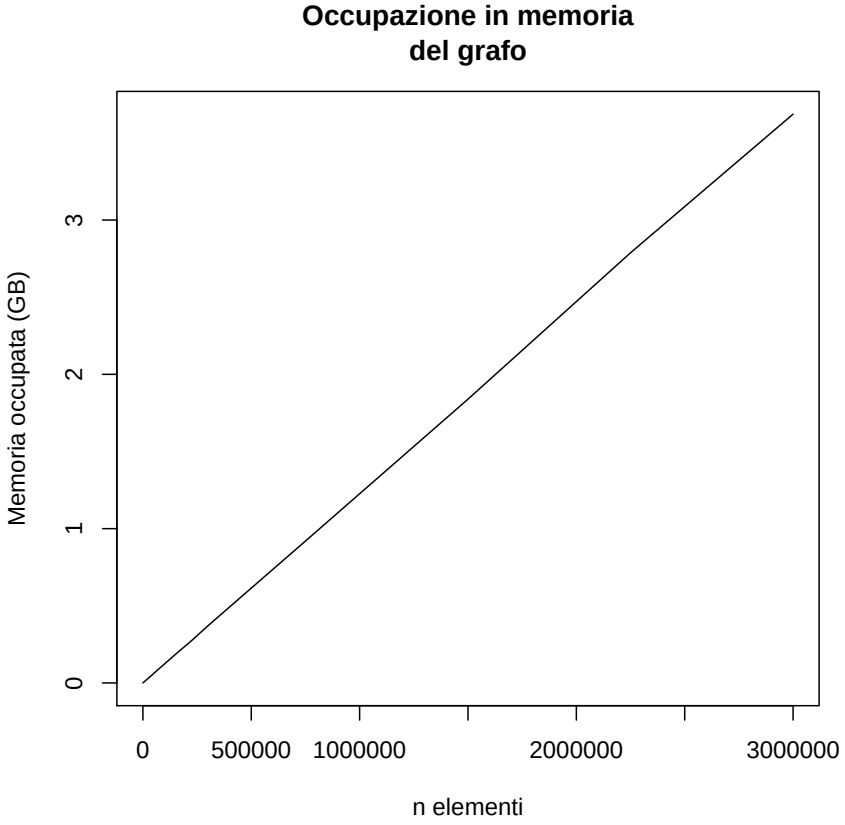

Figura 5.3: Misurazione della memoria occupata dal grafo.

L'uso della libreria NetworkX, come già accennato a pagina 15, comporta una grossa occupazione della memoria. I dati riportati nel grafico in figura 5.3 riguardano il peso del grafo GFA subito dopo la sua creazione da file, senza prendere in considerazione la memoria di "contesto" necessaria per avviare il processo Python e la libreria standard. In totale la memoria occupata raggiungeva, nell'ultimo grafo, circa i 4.2 GB raggiungendo il limite di 5.1 GB durante la ricerca dei percorsi lineari, operazione che si è rivelata essere la più dispendiosa fra quelle misurate.

5.1 Conclusioni

PyGFA ha gli stessi punti di forza e di debolezza di NetworkX: le operazioni sono molto veloci, ma a costo della memoria occupata. Nonostante ciò, la rapidità di sviluppo e l'ottima efficienza di questa libreria rendono *PyGFA* un sistema performante molto facile da ampliare e modificare.

Capitolo 6

Conclusioni

In questo capitolo si riassumerà lo stato di sviluppo di *PyGFA*, i problemi che ancora si riscontrano e di come sarebbe possibile risolverli. Verranno infine presentate le conclusioni personali sull'attività di stage presentando ciò che si è imparato nel suo sviluppo.

6.1 Prima release di *PyGFA*

PyGFA è alla sua prima release stabile, la copertura dei test scritti si appresta sopra il 95% e il numero di issue individuate da PyLint si aggira sulle 280 ottenendo un voto dallo strumento di 8,5/10. Molta parte del lavoro è stata dedicata nel fornire un ambiente che ricordasse NetworkX, sia per facilitare l'utilizzatore sia per aiutare il futuro programmatore ad integrare nuovi moduli della libreria nel sistema. Il processo di sviluppo adottato ha aiutato a favorire i cambiamenti, ad adottare strategie migliori man mano che le conoscenze aumentavano; per questo la struttura della libreria risulta essere divisa in due metodologie di integrazione delle funzioni di NetworkX. I primi algoritmi, riscritti affinché venissero considerati solo archi di dovetail, sono delle copie degli equivalenti offerti dalla libreria, modificati dei soli nodi considerati durante le iterazioni; gli ultimi sono stati invece pensati per essere riutilizzati da un qualsiasi iteratore personalizzato, sfruttando la capacità del linguaggio di passare funzioni (l'iteratore stesso) come parametri. Per questo motivo

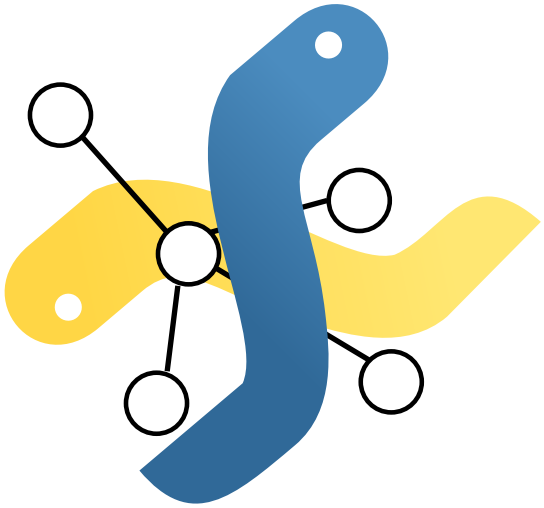

Figura 6.1: Logo di *PyGFA*.

PyGFA necessita di una fase di refactoring più profonda, che stabilisca una struttura unica da rispettare.

L'astrazione dei dati rappresentati nel grafo è un altro problema degno di nota, visto che in alcune situazioni genera ambiguità riguardo le linee delle specifiche dalle quali le informazioni provengono. Si può pensare di aggiungere un ulteriore campo agli elementi del grafo specificando la provenienza dei dati, evitando così nelle fasi successive di fare affidamento alle particolari configurazioni dei dati per risalire alla linea di origine.

6.2 Conoscenze acquisite

Elencare le conoscenze che questo stage ha apportato al mio curriculum informatico richiederebbe troppo spazio, cercherò di riassumere le competenze acquisite che ritengo più importanti. In primo luogo il processo di sviluppo usato: lo stage mi ha permesso di lavorare con grande libertà per questo un processo di sviluppo agile come extreme programming, che pone lo sviluppo al primo posto, mi ha senza dubbio insegnato a gestire al meglio il tempo a disposizione e ad impostare fin dal principio un progetto con delle fondamenta solide, ma allo stesso tempo flessibili a futuri cambiamenti.

Imparare e usare Python durante lo sviluppo si è rivelato più difficile del previsto; se da una parte permette una rapida prototipazione ed un riscontro immediato delle funzionalità, dall'altra riuscire a capire come strutturare un progetto corposo con un linguaggio così flessibile richiede molta pratica. Nonostante questi dettagli, imparare ad usare questo linguaggio e gli strumenti di test di correttezza, della qualità del codice e della documentazione di progetto che gli sono di corredo ha accresciuto notevolmente non solo la mia abilità nell'implementazione di un sistema nuovo, ma anche l'approccio col quale se ne conduce lo sviluppo.

Concluderei questo capitolo e conseguentemente questa relazione con ciò che questo stage mi ha permesso di apprezzare più di ogni altra cosa: lo sviluppo di sistemi performanti in grado di lavorare con una vasta mole di informazioni quali sono le informazioni contenute dai grafi di assemblaggio. Nonostante *PyGFA* non pone molta enfasi sulle prestazioni è stato possibile esaminare i contesti e le situazioni nei quali un linguaggio di alto livello come Python risulta meno indicato rispetto un linguaggio di basso livello come il C e viceversa.

Bibliografia

[1] Wikipedia contributors. *Python (programming language)*. URL: https://en.wikipedia.org/w/index.php%20title=Python_(programming_language)&oldid=792718397 (visitato il 31/07/2017).

[2] NetworkX developers. *NetworkX*. URL: http://networkx.github.io (visitato il 28/07/2017).

[3] NetworkX developers. *Networkx - master*. URL: https://github.com/networkx/networkx (visitato il 01/08/2017).

[4] The Python Software Foundation. *PEP 8 – Style Guide for Python Code*. URL: https://www.python.org/dev/peps/pep-0008/ (visitato il 02/08/2017).

[5] The Python Software Foundation. *Python Documentation*. URL: https://docs.python.org/3/ (visitato il 02/08/2017).

[6] The Python Software Foundation. *Python Patterns - Implementing Graphs*. URL: https://www.python.org/doc/essays/graphs/ (visitato il 01/08/2017).

[7] *git*. URL: https://git-scm.com/about (visitato il 02/08/2017).

[8] The GFA Format Specification Working Group. *GFA*. URL: http://github.com/GFA-spec/GFA-spec (visitato il 28/07/2017).

[9] Vivien Marx. "Next-generation sequencing: The genome jigsaw". In: *Nature* 501.7466 (nov. 2013), pp. 263–268. DOI: 10.1038/501261a.

[10] *Pylint*. URL: https://www.pylint.org/ (visitato il 02/08/2017).

[11] *Python Code Quality Authority*. URL: http://meta.pycqa.org/en/latest/introduction.html (visitato il 02/08/2017).

[12] Ryan R. Wick et al. "Bandage: interactive visualization of de novo genome assemblies". In: *Bioinformatics* 31.20 (2015), pp. 3350–3352. DOI: 10.1093/bioinformatics/btv383. URL: +%20http://dx.doi.org/10.1093/bioinformatics/btv383.

[13] Wikipedia. *Git — Wikipedia, The Free Encyclopedia*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Git&oldid=796151355 (visitato il 21/08/2017).

[14] Wikipedia. *Nucleic acid notation — Wikipedia, The Free Encyclopedia*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Nucleic_acid_notation&oldid=790638814 (visitato il 08/08/2017).