Corso di Laurea Magistrale in Informatica

Bioinformatic - A.A. 2022/2023

Project: **Alignment-Free Sequence to Graph**

Author: ***Davide Grandesso 85278***

Repository: https://github.com/dadegrande99/AlignmentFreeSequence-to-Graph

---

Table of contents

1 Introduction

In the realm of bioinformatics, the study of genetic sequences holds paramount importance in understanding the fundamental principles governing life. At the core of this discipline lies the analysis of nucleotide sequences, the building blocks of DNA and RNA molecules. The advent of high-throughput sequencing technologies has ushered in an era of unprecedented data generation, facilitating the exploration of genomic landscapes with unparalleled depth and breadth.

In this context, the analysis of genetic sequences often involves deciphering intricate patterns and relationships encoded within the nucleotide strings. Traditional approaches to sequence analysis have relied heavily on alignment-based methods, which compare sequences to identify similarities and differences. However, these methods are often computationally intensive and may struggle to handle the vast quantities of data generated by modern sequencing technologies.

To address these challenges, novel approaches have emerged, leveraging graph-based representations to model genetic sequences and their interactions. Graph-based methods offer a versatile framework for capturing complex relationships between sequences, facilitating more efficient analysis and interpretation of genomic data. By representing sequences as nodes and their interactions as edges, graph-based approaches enable the exploration of structural and functional properties encoded within the genome.

In this project, I delve into the realm of graph-based sequence analysis, developing an application that harnesses the power of graph data structures to unravel the intricacies of genetic sequences. By combining principles from bioinformatics, graph theory, and computer science, I aim to provide researchers and practitioners with a powerful tool for exploring and interpreting genomic data in a more efficient and intuitive manner.

2 Technologies

To implement this project has been selected modern and highly compatible tools to ensure efficient project execution. The approach was geared toward adopting state-of-the-art technologies to maximize synergy between components and optimize overall system performance.

2.1 Neo4J

Considering the necessity to handle iterations with a graph, Neo4J was selected due to its optimization for executing queries on graphs. Moreover, it offers an ecosystem of tools and libraries for developers in various programming languages, enabling easy and efficient integration of the database with applications.

Neo4J is an open-source graph database known for its high performance and flexibility. Unlike traditional relational databases, which store data in tables, Neo4j stores data as graphs, composed of nodes (entities) and relationships (connections between entities). This model naturally represents complex relationships, making it ideal for domains such as bioinformatics.

2.1.1 Docker

Docker is an open-source platform increasingly used in project development since it offers an efficient and flexible way to deploy and manage applications. Using containerization technology, Docker encapsulates applications and dependencies into portable units called containers, thereby facilitating seamless deployment across various computing environments, ensuring consistency, scalability, and resource efficiency throughout the development lifecycle.

In order to simplify the installation and deployment of the Neo4J database, a docker-compose.yml file has been developed which allows an instance of Neo4J to be easily started within a Docker environment. This approach guarantees the portability and reproducibility of our system by isolating the execution environment, thereby enhancing maintainability and facilitating consistent deployment across various setups.

2.2 Python

The language used to for the project is Python, this choice was determined by several factors including:

- **Syntax**: Python is written with a very intuitive syntax, and in the field of bioinformatics this is very important since it allows it to be understood even by biologists who do not have much programming experience.
- **Open Source**: the source code of Python is accessible to everyone for free, which means that no licenses are needed to be purchased to use it, promoting access and collaboration among researchers.
- **Wide range of third-party libraries**: like Neo4j, Python also has a large community of developers who often work in different fields, this has resulted in the creation of numerous libraries that make programming easier depending on the context of the application to be implemented.

2.2.1 Libraries

The libraries used for this project are:

- json → make the reading of JSON file easier by avoiding the need to handle data parsing.
- Py2neo → Allows the project to handle the connection with the Neo4J database.
- NetworkX → Used for constructing the graph when it needs to be displayed.
- Gfapy → To import and read GFA files easily.
- Matplotlib → Used for displaying the graph.
- CustomTkinter → is an open-source library that extends the functionality of Tkinter, the standard library for creating GUIs in Python. CustomTkinter has several advantages over Tkinter such as better aesthetics and more intuitive APIs.

2.3 Data Structures

2.3.1 Graph Design

To manage the color of the sequences, using Neo4J, it was decided to color the arcs, the color is set as the label of the link they have no properties. Instead, the nodes were designed so that each had `:base` as the label and two properties are defined:

- `name`: is the character of the nitrogen base which can then be one of {`'A'`, 'C', 'G', 'T'}
- `id`: is an integer that uniquely represents the node

An example of the link structure between two nodes is given below:

```
1.  [:base {id:1, name:"A"}]-[:blue]->[:base {id:2, name:"C"}]
```

In this way, it is possible to manage a large graph in which there are several different colored sequences passing through the same nodes without having to create duplicates and simplifying querying on the graph.

2.3.2 HashTable

Given that Python lacks a native HashTable data structure, a dictionary in Python has been chosen as the surrogate data structure for this purpose. A python dictionary is a data structure that stores data by associating a value with a specific key.

The dictionary designated to accommodate the HashTable comprises unique k-mers as keys and tuples of two elements as values. The first element denotes the ID of the initiating k-mer node, while the second element represents an array containing the colors attributed to that k-mer.

Example representation:

```
1.  {"ACG": (1, ["red", "blue"]),
2.  "CGT": (2, ["blue", "green"])}
```

In this representation:

- Key `"ACG"` corresponds to a tuple with the ID `1` for the starting k-mer node, along with an array of colors `["red", "blue"]`.
- Key `"CGT"` maps to a tuple with the ID `2` for the starting k-mer node, accompanied by an array of colors `["blue", "green"]`.

2.4 Data Import

The data importation component of the application serves as a critical foundation for genetic sequence analysis. Its importance lies in enabling researchers to efficiently access, manage, and analyze genetic data.

Data can be imported according to two different methods depending on the format that is passed to the application (JSON or GFA). As with the database connection phase, when importing new data, the application continuously checks to see if the edges make the graph cyclic in the database, in which case the system removes the last relation and continues with the next data entry.

2.4.1 Import from JSON

It is indeed feasible to import graph data in JSON format using a specialized formatting tailored for this application. In this format:

- **Nodes**: Each node entry must include the ID and character representation of the nitrogenous base. Optionally, a label `":base"` may be assigned.
- **Relationships**: Each relationship entry should specify the starting node (denoted by the `"from"` key), the ending node (specified by the `"to"` key), and the label representing the relationship (indicated by the `"label"` key).

For a clearer understanding of the adopted standard, please refer to the data.json file provided.

2.4.2 Import from GFA

The `upload_from_gfa` function serves to import graph data from a GFA (Graphical Fragment Assembly) file into the application. This method employs the following steps:

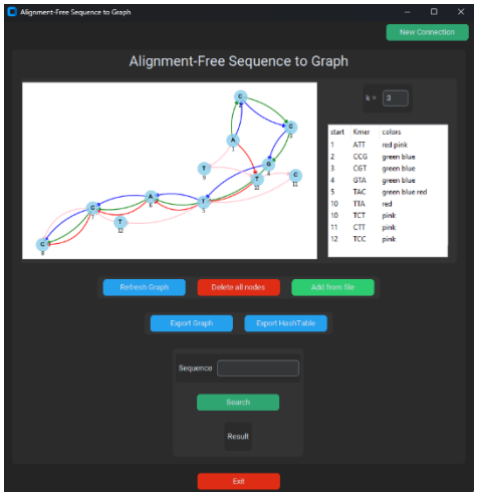
1) **File Parsing**: The GFA file is parsed to extract information regarding nodes and relationships.
2) **Node Creation**: Nodes are created based on the segments defined in the GFA file. Each segment represents a node, and its sequence is used to populate the node's properties.
3) **Relationship Establishment**: Relationships between nodes are established based on the paths defined in the GFA file. These paths dictate the connectivity between nodes, considering the sequence orientation and order.

This method provides a seamless mechanism for incorporating graph data from GFA files, ensuring accurate representation and utilization within the application.

The example GFA files were taken from the HLA-zoo project.

2.5 User Interface

The main goal of creating the graphical user interface (interface.py) is to make the functionality of this project accessible and intuitive to use even for those who do not know how to program, from data import to sequence analysis.



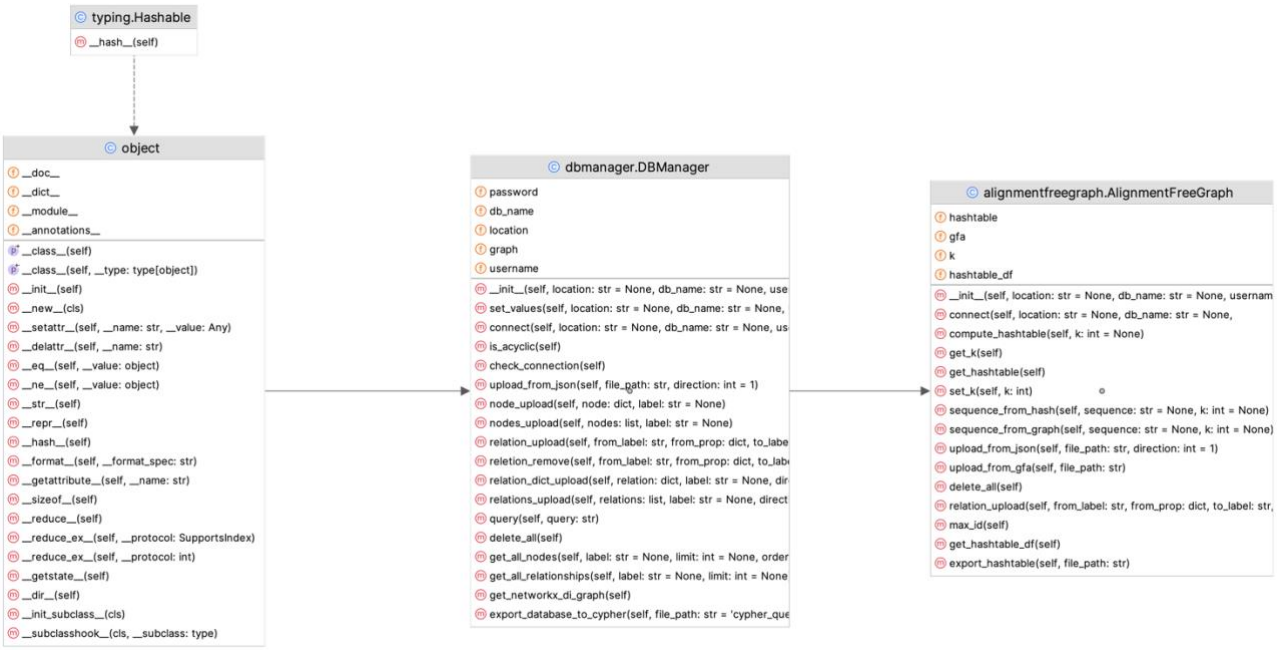
- **Database connection**: This interface allows automatically accessing the server when conditions permit or otherwise facilitate connection to it.
- **Data import**: Importing data is done through a few clicks and by taking advantage of the resource explorer of the device on which the interface is running.
- **Graph and HashTable**: It allows the graph to be visualized and the k-value to be easily changed for the HashTable calculation, the problem arises as the dimensionality of the graph increases, which makes the visualization of this data more complex.
- **Sequence analysis**: it is possible to do analysis on sequences by filling in only a text field.

Through this interface, numerous experiments have been easily conducted, allowing for the verification of code execution correctness, facilitated by comprehensive visualization of all data used for this purpose.

3 Algorithmic solutions

3.1 Class design

For the realization of this project, an approach was chosen where there is a class responsible for managing operations with the database and another class that inherits these operations to implement analyses on the graph.



The class design prioritizes principles of modularity and extensibility, affording avenues for seamless integration of additional features and enhancements in alignment with evolving requirements.

3.1.1 DBManager Class `dbmanager.py`

Entrusted with the management of interactions and queries directed towards the Neo4j graph database, the DBManager class assumes the mantle of responsibility in establishing, maintaining, and executing operations within the database realm. Its functionalities span the spectrum of database connectivity, query execution, and seamless integration with external data sources.

3.1.2 AlignmentFreeGraph Class `alignmentfreegraph.py`

Building on the foundation laid by the DBManager class, the AlignmentFreeGraph class is the heart of this project. This class contains the operations that handle graph import, Hash table construction, and sequence analysis. Whenever a direct operation is done with the database, it is checked that the graph used is a DAG (Direct Acyclic Graph) before performing the requested operation.

3.2 HashTable building

The construction of the HashTable emerges as a pivotal endeavor in ensuring the efficient storage and retrieval of k-mers and their associated nodes within the graph. The construction of it is done with the `compute_hashtable` function; key aspects of this function include:

1) **Initialization**: The HashTable undergoes initialization, assuming the form of a dictionary with k-mers serving as keys and corresponding nodes as values.
2) **Construction from Graph**: a query is made on the database to extract all k-character sequences.
3) **Handling Collisions**: In scenarios where multiple nodes share identical k-mers, the algorithm handles these collisions by keeping only the unique k-mers in the graph even if they are shared by multiple sequences.

3.2.1 Time complexity

The time complexity associated with hash table construction depends on the number of nodes present in the graph and the number of k-mers found in the graph. The graph is observed starting with each node present for k neighboring nodes and then checking all possible k-mers found, the time complexity is then:

$$O(NK) + O(NK) = O(NK)$$

3.3 Computation of vertexes from the sequence

The computation of vertices from sequences is performed through a series of steps orchestrated, by the `sequence_from_hash` function

1) **Sequence Parsing**: Parsing sequences into k-mers entails traversing the sequence and extracting k-mers.
2) **HashTable Lookup**: Lookup operations within the HashTable encompass dictionary lookups, characterized by an average time complexity of $O(1)$ per operation in the HashTable.
3) **Vertex Identification**: This operation is used to identify the starting node of the k-mers to be analyzed; it is also a straightforward operation due to the HashTable structure.
4) **Handling Ambiguity**: It is finally checked if the k-mers analyzed do not share any common sequence between them, to do this an intersection between the sets of sequences is made.

3.3.1 Time complexity

Thanks to the structure of the HashTable this analysis has a very low computational complexity which depends on the number of k-mers analyzed and is therefore $O(L)$ where L is precisely the number of k-mers to analyze.

4 Conclusions

In conclusion, this project has tackled the challenge of developing an application for sequence analysis based on graphs, using an object-oriented programming approach and an intuitive graphical interface. Through the implementation of a class for database management and another for graph analysis, I have created a versatile and functional system.

During development, I focused on several crucial aspects, including data import, HashTable construction, sequence analysis, and result visualization. Thanks to the graphical interface, users can easily import their data, perform custom analyses, and visualize the results clearly and intuitively.

A key element of the project was the implementation of efficient algorithms for HashTable construction and graph analysis, ensuring optimal performance even on large datasets. In particular, leveraging Neo4J as the database allowed us to harness the power of graph data structures to execute complex queries efficiently.

Looking to the future, there are several directions in which this project could evolve. One could consider implementing new sequence analysis features or integrating with other bioinformatics tools for a wider range of applications. Additionally, further optimizations could be made to improve the performance and usability of the application.

Ultimately, this project represents a significant contribution to the field of sequence analysis, providing users with a powerful and flexible tool to explore and understand the complex relationships within genomic data.