UNIVERSITY OF GRANADA

MAJOR IN COMPUTER SCIENCE

GeneSys

A BIOINFORMATIC TOOL FOR GENOMIC DATA ANALYSIS

Author: Bruno Otero Galadí

Supervisor: Dr. Fernando Berzal Galiano

August, 2024. Granada, Spain.

GeneSys: A bioinformatic tool for genomic data analysis

Bruno Otero Galadí

Keywords: reverse transcriptase, amino acid, molecular biology, nucleotide, protein, DNA, RNA, fasta, PATRIC, Biopython, BV BRC, presence-absence matrix

Abstract:

Recent decades' advancements in biological research have brought numerous benefits to our understanding of nature and society's progress. However, these advancements have also generated a vast amount of biological data that must be processed in an affordable amount of time to remain valuable for researchers. If not, this processing could become a bottleneck, potentially slowing the current rate of scientific discoveries.

The majority of the problems are related to the preprocessing of big amounts of biological data stored in public databases, which are continuously updated to locate more and more examples of genomic information coming from all kind of sources. So, if researchers without advanced programming knowledge want to dive into these databases in search for a specific kind of genomes, they must be able to manipulate the data in a way that allows them to repeat the process as many times as needed, as well as separating the process into tasks that will be executed sequentially. This is where GeneSys comes into play.

GeneSys is a modular and scalable software tool with an user-friendly interface that allows researchers to define tasks within a workflow that can be executed and redefined freely in order to satisfy their researching needs, regardless of complexity.

This work tries to offer a toll that solves a real life issue involving reverse transcryptases, also known as RTs, an unique kind of proteins with significant research potential, many aspects of which remain unexplored. Such proteins are currently being studied by Dr. Francisco Martínez-Abarca Pastor at La Estación Experimental del Zaidín (EEZ) in Granada, Spain. GeneSys will help Martínez-Abarca to efficiently face his investigations involving RTs.

GeneSys: una herramienta bioinformática para el análisis de datos genómicos

Bruno Otero Galadí

Palabras clave: reverso transcriptasa, aminoácido, biología molecular, nucleótido, proteína, ADN, ARN, fasta, PATRIC, Biopython, BV BRC, matriz de presencia-ausencia,

Resumen:

Muchos avances se han dado en las últimas décadas en la investigación biológica. No obstante, estos avances implican la necesidad de procesar cada vez más datos biológicos a un ritmo que debe permanecer constante para ser rentable. Si dicha eficiencia no se alcanza, existe el riesgo de que se convierta en un cuello de botella que, llegado el momento, reduja el ritmo con el que se han producido avances en esta materia hasta ahora.

La mayoría de los problemas van de la mano al preprocesamiento de información genética contenida en diversas bases de datos, que además se incrementa en volumen con el paso del tiempo. Cualquier persona investigadora que carezca de un nivel alto de programación y desee emplear información de una base de datos para acometer una tarea va a necesitar herramientas que le permitan repetir el proceso aplicado a los datos tantas veces como desee, así como subdividir el trabajo en tareas y ejecutarlas una a una. Es aquí donde entra GeneSys.

GeneSys es una aplicación modular y escalable con una interfaz de usuario fácil de usar, enfocada en ayudar en las tareas de investigación de datos biológicos. Permite a un usuario general definir tareas dentro de un flujo que podrá ejecutar y modificar según sus necesidades.

Este trabajo trata de resolver un problema de preprocesado de datos relativo a las reverso transcriptasas, un tipo de proteínas con un gran potencial investigador. El doctor Francisco Martínez-Abarca Pastor de la Estación Experimental del Zaidín (EEZ) de Granada, España, se encarga en la actualidad de estudiar dichas proteínas. GeneSys le servirá para progresar en sus investigaciones.

I, **Bruno Otero Galadí**, scholar of the **computer science** university degree at the “**Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**”, with a Spaniard national identification number of **75574203K**, authorize the placement of the present work at my school’s library so it can be consulted by anyone who wishes to.

Signed: Bruno Otero Galadí

[illegible]

Granada, on September the 1st of 2024.

Mr. **Fernando Berzal Galiano**, teacher of the Computing Science and Artificial Intelligence Department of the University of Granada.

Informs:

That the present work entitled as **Genesys: A bioinformatic tool for genomic data manipulation**, has been realized under his guidance by Bruno Otero Galadí, and authorizes the defense of the aforementioned work under the collegiate tribunal that might correspond.

And so that it is stated, he issues and signs the present invoice in Granada on <month> the <day> of 2024.

Supervisor:

Fernando Berzal Galiano

Acknowledgements

This work would have never existed without my supervisor, Fernando, whose suggestion to focus on bioinformatics was crucial in shaping the direction of my research. Additionally, I would not have been able to discover the significance of reverse transcryptases (RTs) and the reasons for their study without the assistance of Francisco Martínez-Abarca Pastor, a former researcher at the Estación Experimental del Zaidín (EEZ) in Granada, Spain. Francisco asked me to help him facing the RTs issue involving the preprocessing of amino acid data from the Bacterial and Viral Bioinformatics Resource Center (BV-BRC) online database. His role as a client in this work is the very reason it came into existence.

Furthermore, I want to give sincere thanks to Antonio Quesada Ramos, my former high school biology teacher, who facilitated my connection with Francisco. He is also the reason why I am so interested in biology as a field of research.

Finally, all of the time and resources I have dedicated to this matter are direct merit of my family —Dulcinea, David and Leonardo— whose support and understanding provided me with all the space I needed in order to achieve the main goals of this work. Their encouragement has been indispensable. So, thank you.

MAIN INDEX

IMAGE INDEX

Note: all the employed images have been taken/created by the author.

TABLE INDEX

1. INTRODUCTION

1.1. A brief overview of molecular biology.

If we assume selecting breeding as a form of molecular biology researching, we can affirm that genetics has been taking part in humanity's history since, at least, the Neolithic period. But it was not until nineteenth century and the appearance of Gregor Johann Mendel's works that a first theoretical basis for the principles of heredity was set. Since then, molecular biology has become one of the most developed researching fields, being continually adapted to answer new questions and to face new challenges. As a result, molecular biology went from studies about peas to relatively recent works that suggest the existence of life beyond planet Earth, always being strongly correlated to chemistry and incorporating key discoveries like the DNA structure, which also paved the way for other numerous applications such as the Polymerase Chain Reaction (PCR) with a major relevance in the understanding and diagnosing of several diseases, or the Human Genome Project in 1990[4]. All of this improvements have provided invaluable benefits for society. Biology and specially molecular biology are not just fields with history. They are fields with future.

Before we get deeper into biological concepts, we should remember what DNA, RNA and proteins are and how they are related to each other. DNA[5] and RNA[6] sequences are identified as sequences made up of repetitions of up to four nucleotides, represented as A, C, T, G for DNA strings and A, C, U, G for RNA ones. Apart from the composition, the main differences between both structures involve their spacial distributions (with a double-helix polymer structure for DNA and a single-stranded biopolymer for RNA) and their biological functions, with DNA serving as a codification of the genetic information and RNA using DNA to synthesize the proteins that are stored in cells[7]. Proteins are chemical components made of elements called amino acids. The amino acids that might be found in proteins' cells differ between species, but there are no more than twenty different amino acids that occur naturally in any living being's proteins[8].

To sum up, DNA defines the genetic composition of a living being, and RNA replicates that composition in order to define the structure of proteins. But, what is the mechanism that translates RNA into proteins? The nucleotides contained in a RNA sequence (and therefore in its equivalent DNA sequence) are read in intervals of three each, and they can be read starting from anyone of the first three nucleotides that compound the aforementioned RNA (or DNA) sequence, onward and backward, which gives us up to six different ways of getting a protein from a same RNA string. A protein is properly identified when, using one of those lecture ways, a specific set of three nucleotides that marks the end of the lecture is found. This sets of nucleotides are called stop codons, and correspond to a certain amino acid that serves as a delimiter of the protein[9].

1.2. Reverse transcryptases.

Reverse transcription refers to the process of turning specific RNA sequences into DNA. Not all RNA strings are valid for this issue, and those that indeed are are formally called “RNA-dependent DNA polymerases”, “reverse transcryptases” or “RTs”[10]. As not all RNA strings serve as RTs, they need to be specifically recognized before researchers start experimenting with them. As we have stated before, RNA sequences are translated into proteins. That process can be done backwards, too, which means that it is possible to find a specific protein that is configured by a RNA sequence that in fact is a RT. In other words, we can identify RTs by observing proteins.

Reverse transcryptases have remarkable biotechnological applications, such as molecular cloning strategies or in the field of synthetic biology. But the most important use they have provided to humanity, or at least the most widespread one, might be the detection of viral RNA in SARS-CoV-2 testings[1], as they serve as a key element in the propagation of genetic elements across specific DNA structures.

Since 2020 COVID-19 pandemic, the lock-down and the infection waves, the interest in molecular biology seems to have gained so much popularity, being mentioned in the news, in social networks or even at the dinning room with our families. However, and despite the crucial role they have played throughout all these years, reverse transcryptases have not become that popular. And as researchers and diverse studies point out that pandemics would be more common in the future, it is quite clear that RTs will keep being at the spotlight of scientific investigations. The main arguments that are exposed to support the assumption of pandemics becoming more likely to happen concern topics such as climate change[2], the destruction of the environment or the increasing contact between humans and disease-harboring animals[3].

In a post-COVID world, it is crucial to be prepared for upcoming similar events. RTs take part in that process by playing a key function in PCRs, which play a potentially high disease detection role.

1.3. When more is less. Current problems involving genomic databases.

Nowadays, biologists tend to work obtaining their genetic data from enormous public domain databases whose volume of biological information is increasing at a relatively faster rhythm than the stored datasets of other scientific disciplines, with the amount of raw data corresponding to genome sequencing experiencing the biggest growth along with exome sequencing data[11]. This has led to an overwhelming amount of genomic data that needs to be correctly preprocessed in order to start searching for valuable knowledge. And RTs are taking part in that problem, too, as new examples of them are being included in those databases month by month, increasing the difficulty to distinguish which are recent discoveries from those which are not, as well as requiring more complexity in the computing of all the existing samples in order to identify common patterns between them.

Francisco Martinez-Abarca Pastor is a researcher from the Estación Experimental del Zaidín (EEZ) in Granada, Spain. Among his current issues there is the exploration of RTs' datasets in search of undiscovered correlations between reverse transcriptase samples that are separated in evolutive terms. In the year 2019, he supervised a work involving RTs' written by the former postgraduate degree student Mario Rodríguez Mestre. The aforementioned work was entitled "Analysis of Novel and unexplored groups of prokaryotic Reverse Transcriptases" and consisted of the extraction of all the available datasets of RT's stored in certain databases, its preprocessing, its classification through clustering algorithms and the seeking of undiscovered common behavior patterns between the RT's contained in those clusters[12].

The results of the study were considered successful, and Martínez-Abarca decided he would repeat the experiment once the databases were updated with new samples. The problem is that in order to repeat the process, all the steps of preprocessing the raw data and applying the clustering had to be done manually again, which required so much effort in terms of time to be worth, so even though Martínez-Abarca wished to repeat the study, he was not able to do so.

But if he had a tool that at least could automatize the preprocessing of the raw data downloaded from the databases just as Rodríguez Mestre did back in time, he would be capable of make the same experiment whenever he wanted, so in the long term he could exploit all the advantages RTs have. GeneSys serves as a software tool that solves Martínez-

Abarca's issue.

1.4. Memory structure.

In order to facilitate the reader's task, here are mentioned the main parts of this memory and what each exposes:

- **Introduction**: this chapter exposes briefly the context in which GeneSys is made. It provides a generic view of the problem as and helps those readers with basic biology knowledge to understand what reverse transcriptases are and what they are used for.
- **Objectives**: includes the objectives to accomplish with GeneSys development. Such accomplishments will be analyzed in the "Conclusions and future work" section.
- **Planning**: organization, estimated developing time and hypothetical budget it might require.
- **Problem analysis**: statement of the preprocessing tasks that GeneSys must accomplish and the biological reasons that justify why they must be done that way.
- **Architecture and design**: it provides various diagrams that explain the architecture of the application. Also, justifies why that architecture has been chosen and what requirements are crucial to satisfy.
- **Implementation**: abstract of GeneSys' coding process and the development stages that occurred while implementing the tool.
- **GeneSys user's guide**: A friendly-user manual that explains how to use the app. It is aimed to be understood by any researcher how wishes to employ GeneSys in their investigations.
- **Conclusions and future work**: it compares the accomplished objectives against those exposed in the "Objectives" section. It also proposes improvements that can be made to the application in the future.

2. OBJECTIVES

The main objective is to provide a functional application that can be intuitively employed by an user experienced with biological terminology but with a basic programming knowledge, which accomplishes properly the issue of RTs preprocessing. In order to achieve the proposed goal, we can distinguish the following objectives in the development of the application:

1. To develop a bug free logic that works exactly as the user needs it to work.
2. To provide an intuitive yet attractive friendly-user interface.
3. To properly recognize which tasks must be automated by the application, so that the user receives a window to modify certain parts of the preprocessing of the RTs in order to adapt their experiments freely while not losing the automatization benefits that GeneSys provides.
4. To provide a software framework that unifies all the preprocessing tasks in an unique execution context. In other words, we do not want the user to open any other application than GeneSys to achieve the proposed preprocessing goals.
5. To make an application that does not freeze or crash when it is executing a long task.
6. To give the user the capacity to apply changes to the task that it is going to be executed, such as the pathnames where to save the results. We assume that it would be always better to give the user as much freedom as possible when defining parameters related to biological terms.
7. To properly inform users about what is happening in the preprocessing of the RTs, so that they can study the results returned at all the steps of the process and draw their own conclusions for their research.

3. PLANNING

3.1. Development steps.

Here are the phases that the application's development process had and how much time was required for each.

3.1.1. Researching phase (April 2024).

As molecular biology and reverse transcriptases were unexplored fields for the student, it was crucial for him to properly understand what they are so that he could have an approximated idea of how to bring up the interface design and the formats with which manipulate the data. Some meetings at Dr. Martínez-Abarca's office took place during the month of April, along which the student received detailed answers to all his inquiries. Among other issues, he was informed about the existence of fasta files, a special kind of format employed for representing genetic data that is used by almost every bioinformatic tool in the world. He also was introduced to Rodríguez Mestre's work[12] and received context about the way in which the experiment was done back in 2019. A lot of notes were taken, specially those concerning the first steps of the development. From the beginning, the application was meant to be deployed as a desktop application, as Martínez-Abarca was not interested in paying the use of some cloud services for completing the preprocessing tasks.

Finally, there was a virtual meeting between the student, his supervisor Fernando Berzal Galiano, and Martínez-Abarca, which ended with the supervisor approving the proposed project and giving his own advises and opinions about the issue.

3.1.2. Requirement analysis phase (April 2024).

The student sought for potential tools to develop GeneSys. Finally, he opted for VS code as his main coding environment, Python as the programming language with which develop the app and Kivy for the user interface.

As GeneSys was going to be a tool designed for working with genetic data, it was necessary to decide from which database the data should be taken. The selected one was PATRIC, the official public database of the Bacterial and Viral Bioinformatics Resource

Center, also known as BV BRC[21]. All the application would be designed in order to manipulate data downloaded from PATRIC, even though the resulting information would be returned in an universal format easily readable by any other tool.

For managing backups, a GitHub repository was created. At first, it was private, but by the end of the development it was made public[20].

3.1.3. Design, implementation and testing phase (May-July 2024).

At first, some trial genetic data were downloaded from PATRIC database and deeply studied in order to understand the format they had. After that, the student resolved to divide the preprocessing task that GeneSys would apply to the data into smaller tasks which would be executed in a workflow.

Next, the student focused in learning Kivy in order to understand how to use it to make interfaces in Python. A prototype of interface was made, and the tasks that would be applied to the workflow were implemented by receiving their parameters from that interface. For each new task of the preprocessing that was implemented, a new Kivy screen was designed for getting its parameters.

Throughout the process, there were cyclical testings of GeneSys' made features so far. As a result, some modifications were applied to the structure of the code that leaded to obtain the final design that can be found in "architecture and design" section.

Finally, once the final task to accomplish worked properly, The efforts were putted into giving a prettier design to the user interface. Some colored images were created employing different tones of green, and all the buttons and boxes of the application were decorated with those colors.

3.1.4. Deployment and evaluation phase (August 2024).

The ending of the project began with the redaction of the memory, and also with the final testings of the application. All the available information about the development process was gathered and included in this work. Also, the student notified to Dr. Martínez-Abarca that the application was done and that he would be enchanted to teach him to use it.

Image 3.1.4.1, Gantt diagram of GeneSys' development phases.

3.2. Estimated budget.

Now, it is time to estimate how much money might a project like this cost. There are some important matters to look at when making a budget. All of them have been studied one by one in this section.

3.2.1. Human resources.

The next scenario will be considered:

→ The development of the application takes place in **Andalusia, Spain, in the year 2024**.

→ The raw monthly wage for each developer will be of **2000€** on a full time contract of forty hours per week.

→ There is only **one developer**, Bruno Otero Galadí.

→ The development of the application involves **five hours a day**, from Monday to Saturday, which is equivalent to thirty hours a week.

→ The development will extend **from April to August**, five months in total.

Considering the above, human resources' cost will be of **1500€ of raw wage** per month, during a total of five months, 7500€.

Applying the corresponding Andalusian taxes, we get a **net wage of 1247€** per month, from which have been discounted 157€ corresponding to Andalusian "IRPF" and 96€ relative to social security. For the company that has hired the developer, a raw wage of 1500€ means to pay 471€ for each payed month corresponding to company imposing taxes, which leads to a total budget of **1971€ per month for the company**.

1971€ per month during five months corresponds to **9855€**, from which 7500€ will be the worker's raw wage, from which 6235€ will be the worker's net wage[13].

3.2.2. Hardware resources.

The hardware tools that are going to be employed in the application's development are specified in the next table.

Table 3.2.2.1

Hardware tool	Total cost	Average lifespan	Cost for five

			months
ASUS VivoBook 14/15 laptop	850€	4 years[14]	88.54€
HP monitor	100€	15 years[15]	2.78€
Xiaomi Redmi Note 12 smartphone	150€	3 years[16]	20.83€

In total, hardware's budget rises up to **112.15 €**.

Here is another table that specifies the laptop's characteristics:

Table 3.2.2.2

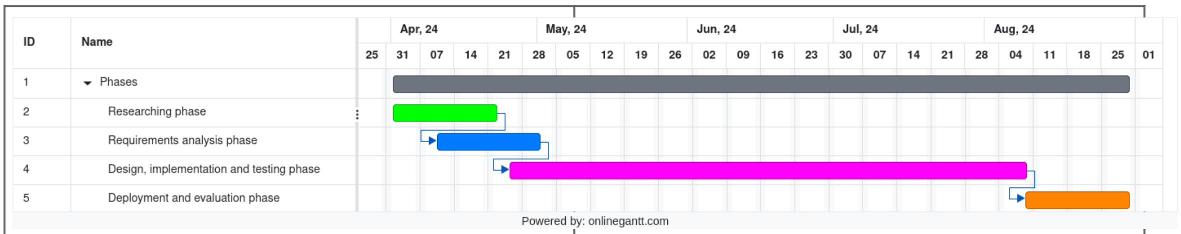
Component	Characteristics
Laptop model	ASUSTeK COMPUTER INC. VivoBook_ASUSLaptop X421JAY_X413JA
CPU	Intel Core i7-1065G7 CPU 1.30GHz × 8
RAM memory	16.0 GiB DRAM
Disk memory	1.0 TB SSD
GPU	Mesa Intel Iris(R) Plus Graphics (ICL GT2)
Operative System	Ubuntu 22.04.4 LTS 64 bits
Dimensions	229 mm x 18 mm x 360 mm[17]

3.2.3. Software resources.

The next software tools will be employed, all of them at a free cost as they are open-source software. All of the licenses have been selected as open-source because they are tools to which the developer is used and that help making the budget cheaper.

Table 3.2.3.1

Software tool	License
Visual Studio Code text editor	Microsoft Software License
Ubuntu 22.04 operative system	GNU General Public License version 2 (GPLv2 for the Kernel)
GitHub online repository	As it is an online tool, the user is subscribed to GitHub's Terms of Service instead of a license.
Git tool for coding backups	GNU General Public License version 2 (GPLv2 for the Kernel)
Kivy 2.3	MIT License
Python 3.10	Python Software Foundation License



ID	Name
1	Phases
2	Researching phase
3	Requirements analysis phase
4	Design, implementation and testing phase
5	Deployment and evaluation phase

(PSFL)

3.2.4. Indirect costs and materials.

The average electricity costs in Spain from April to August of 2024 are[18]:

→ April: 85.58 €/MWh

→ May: 99.67 €/MWh

→ June: 117.02 €/MWh

→ July: 132.58 €/MWh

→ August: 144.06 €/MWh

These prices result in an average cost of **115.72 €/MWh** per month. Let's assume **our laptop cosumes 100 W per hour, our monitor consumes 30 W and our smartphone's consumption is too insginificant to be worth counting**. Keeping in mind that we will be using this tools for thirty hours a week, per four weeks a month has, that equals to 120 hours a month.

100 W of laptop's consumption · 120 hours of usage per month = **12000 W of laptop's consumption in a month** = 12 kW = 0.012 mW.

30 W of monitor's consumption · 120 hours of usage per month = **3600 W of monitor's consumption in a month** = 3.6 kW = 0.0036 mW.

Both tools have a consumption of 0.0156 mW per month. If electricity costs 115.72 €/Mwh, then the electricity cost per month equals to 1.81€ per month.

Also, we will consider that **our office consumes a total of 50 kW per month**, which adds 0.05 mW · 115.72 €/Mwh = 5.79 € to the final invoice. In total, electricity costs rise up to 7.60 € per month, which results in **38€ across five months**.

For the Internet connection, our reference will be Digi's plan, which costs 15€ per month[19], which results in **75€ across five months**.

Considering there might be also requirements to satisfy concerning office materials such as pens or notebooks, an extra of **30€ will be added to the budget**.

In total, this section has a cost of 30 + 75 + 38 = **143 €**.

3.2.5. Total budget.

Here is a final table of the budget that this project might require.

Table 3.2.5.1

Resource	**Cost**
Human resources	9855 €
Software resources	0 €
Hardware resources	112.15 €
Indirect costs and materials	143 €
Total	10,110.15 €
Spanish “IVA” of 21%	2,123.13 €
Final budget	12,233.28 €

4. PROBLEM ANALYSIS

In this section, we will dive into the reasons that justify the existence of this work and explain step by step the specific problem that it must solve.

4.1. Biological context: the prediction of unknown RTs' behavior patterns experiment.

Mario Rodríguez Mestre's post degree work is not the only one that focuses on the RTs' experiment. Indeed, a similar article signed by Rodríguez Mestre himself along with Dr. Martínez-Abarca and others[22] was published back on December 2020.

The key of the study resides in functional association. As it has been mentioned in the introduction, the research was aimed to find common behavior patterns between RTs that were separated in evolutive terms. In other words, the value of the study resides in discovering remarkable unknown correlations between RTs that despite of being far away from each other in terms of genetic evolution, present similar amino acid patterns and therefore common researching applications.

In nature, proteins are never found isolated. Instead, they are found as parts of a very large sequence of amino acids in which they are presented among many other proteins. One of the main researching tasks to accomplish with this matter is to recognize all the valid RTs contained in these chains, which usually implies to search for stop codons in the converted RNA/DNA sequence.

But, how can those long nucleotides sequences be found in the database? Researchers never have the complete string they are seeking. Instead, they search for a far more brief protein that they already know is contained in the sequence they are looking for (usually, each protein of a database is identified by an ID, so researchers use that ID in the search bar instead of the amino acid sequence that forms the protein. It is less prone to errors and more comfortable). This brief protein is known as a bait. Once the nucleotides sequences related to that bait are returned, researchers start looking for all the valid RTs that are contained in the nucleotides surrounding the bait.

In the study, researchers had an initial set of 198,760 annotated RTs, from which 9141 were finally selected as those that where representative of each evolutive branch. It is important to remember that the main objective was to discover unknown patterns between non-related samples, because it was assumed that related samples would show a similar behavior, so their matches would have considered noise and consequently they needed to avoid. The filter applied consisted in comparing each protein with the rest of the dataset, and if an equivalent protein at 85% sequence identity was found, the selected protein was removed from the set. In the end, there were only samples corresponding to proteins having a minimum difference among each other of 25%. In the process of filtering the samples, a filo-genetic tree was created.

Each of the selected 9141 samples was employed as a bait for which were searched up to 30,000 nucleotides positioned before and after it (up to 60,000 nucleotides for each bait). Once all the nucleotides were isolated, the efforts were putted into finding all the proteins contained in each one of them. Finally, the clustering was applied to the final obtained set.

A comparison between two RTs can be done just by looking for common amino acid sequences spotted in both proteins. That is why a clustering algorithm is the perfect approach for this task, as a good implementation can help researchers discovering new patterns by just organizing the clustering results. As more than 60,000 clusters were returned by the clustering algorithm, in order to optimize the computational time required to process all of them, it was established a cutoff of the minimum samples that a cluster had to contain in order to consider that cluster to have a significant amount of proteins. After that, the number of clusters was reduced to 5413.

The final results were showed in a presence-absence matrix (PAM), which consists of a matrix where cells only get binary values. Usually, rows define a certain characteristic, columns another one, and each cell indicates if there is a value in the dataset that matches both characteristics at the same time. In this experiment, the PAM had the rows corresponding to the RTs ordered by the position on the previously constructed filo-genetic tree, and the columns corresponded to the various neighboring protein clusters, ordered by size (the first column represented the cluster that contained the major amount of samples, the second column, the second largest cluster, and so on). This matrix was then analyzed in order to get the results of the experiment.

4.2. The current problematic.

It has been explained how the experiment was made. But, how was the process of obtaining the samples? And how were they preprocessed before applying the clustering algorithm? In

Rodríguez Mestre's post degree work[12], the preprocessing of the samples implied using multiple tools, formats and manual operations that would be impossible to repeat in an affordable amount of time if the experiment wanted to be done again. It has been mentioned previously how Martínez-Abarca wished to analyze the new RT samples that were being included periodically in the databases, but that idea could not prosper as he had not the appropriate tool to automatize the preprocessing.

Long story short, there were no existing implementations of this issue, as it was a very specific and relatively new challenge. As a result, there were no similar tools that accomplish what the application had to do yet. In that way, GeneSys started from scratch, so at first the student's efforts were put into understanding what exactly the preprocessing to accomplish was. The tasks that compose the preprocessing are stated down below.

4.2.1. Tasks to accomplish.

- **Isolate useful information from RTs downloaded from databases**. Usually, genetic databases incorporate a search bar and a friendly-using online interface to download the data recovered after searching for a specific term. It is possible to access to a given set of data and download it in some easily manipulable format.

  Image 4.2.1.1 Interface of the National Center for Biotechnology Information's (NCBI) main page[23].

  Image 4.2.1.2. Results available for download in NCBI's website.

  In many cases, the downloaded samples are not given as protein strings, but as its corresponding IDs, among other characteristics. In that case, it is mandatory to isolate proteins IDs' and remove the rest of the downloaded information, as we would need only the IDs' in order to get the amino acid strings corresponding to each ID.

- **Obtain the proteins from its corresponding IDs**. In case we had downloaded proteins' IDs, we would have to find a tool that returns each protein's full amino acid sequence given its ID, and integrate that tool into our application. In case a protein is repeated in the final dataset, it should be removed.

- **Isolate one sample from each evolutive branch**. Once all the proteins are stored without repetitions, it is time to apply the previously mentioned filtering to discard those samples that are closer to others in terms of evolution, so that we end up with a final dataset that stores one sample from each evolutive branch. The default way to decide wether a protein is in the same evolutive branch as another would be to compare each one with the rest and discarding those which are equivalent to at least another one by 85% of coincidence. However, we should let users to change that percentage in case they wanted.

  Note that no filo-genetic tree is being created in this step as it is not a necessary action in order to accomplish the task. However, Martínez-Abarca stated that it could be useful, but unfortunately there was no available time to include such functionality in the application.

- **Get 30,000 nucleotides to the right and to the left from each sample**. The next phase consists of employing each selected protein as a bait from which obtain up to 30,000 bases of nucleotides. The associated nucleotide sequence to which each bait is associated can be found in databases such as PATRIC[21] and can be manually downloaded from it. This phase must automatize the extraction of such large nucleotides sequences for each bait, so that researchers do not have to do it by themselves.

  Image 4.2.1.3. Screenshot of PATRIC's online tool for manually downloading nucleotide bases of a certain sequence given its bait. The bait can be recognized in the image with an emphasized yellow color.

- **Recognize all the available RTs in the nucleotides sequence associated to each bait**. Finally, before giving the preprocessed data to researchers so they can apply clustering, each obtained sequence of nucleotides must be read in order to get all the proteins it stores. Fortunately, it has been previously mentioned that the classification of potential proteins in a large sequence of nucleotides is already done by the databases that contain them, and when a large amount of RNA/DNA nucleotides is downloaded from them, some fragments of such nucleotides sequences that might correspond to proteins are already tagged. We would only have to observe those specific fragments and confirm if they have a valid stop codon or not.

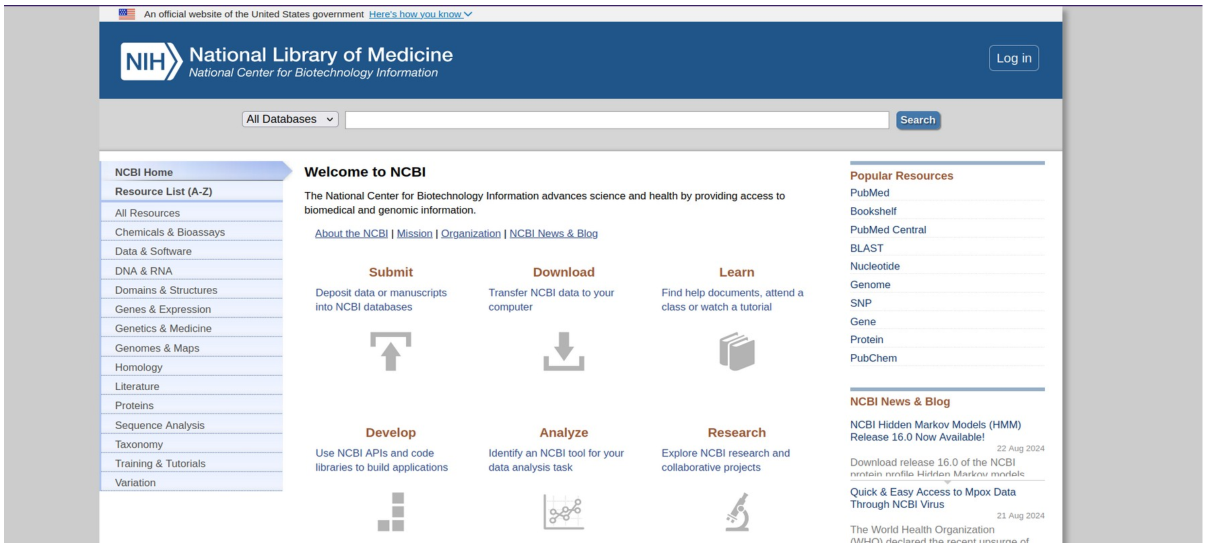
The key of the process lies, indeed, in recognizing stop codons. The available combinations of bases that correspond to that case are: “UAA”, “UAG” and “UGA” for RNA sequences and “TAA”, “TAG” and “TGA” for DNA sequences. If a stop codon is not found in a candidate protein, it must be reversed and the process must be repeated. If still there is no valid stop codon at the end of the protein, the protein is discarded. A protein would also be discarded if it stores a stop codon in any part of the sequence but the end, and if it has not a length equivalent to a multiple of three, too (remember that nucleotides were read three by three in order to convert a string of bases into its corresponding string of amino acids).

Image 4.2.1.4. An example of how PATRIC shows information about a certain fragment of a nucleotides sequence that might correspond to a valid protein.

4.3. Selected languages and tools.

Once all the required task to achieve are defined, it is time to select which tools to employ to face the development. The main issue is to select the database to work with. There are multiple available options for downloading genetic data: from the already mentioned PATRIC[21] and NCBI[23] to others such as PDB[24] (Protein Data Bank), CD Genomics[25], the European Nucleotide Archive[26] (ENA) and Mgnify[27].

When deciding which database should be employed, the major characteristic to look for is


An official website of the United States government Here's how you know

NIH National Library of Medicine
National Center for Biotechnology Information

Log in

All Databases | Search

NCBI Home
Resource List (A-Z)
All Resources
Chemicals & Bioassays
Data & Software
DNA & RNA
Domains & Structures
Genes & Expression
Genetics & Medicine
Genomes & Maps
Homology
Literature
Proteins
Sequence Analysis
Taxonomy
Training & Tutorials
Variation

Welcome to NCBI

The National Center for Biotechnology Information advances science and health by providing access to biomedical and genomic information.

About the NCBI | Mission | Organization | NCBI News & Blog

Submit
Deposit data or manuscripts into NCBI databases

Download
Transfer NCBI data to your computer

Learn
Find help documents, attend a class or watch a tutorial

Develop
Use NCBI APIs and code libraries to build applications

Analyze
Identify an NCBI tool for your data analysis task

Research
Explore NCBI research and collaborative projects

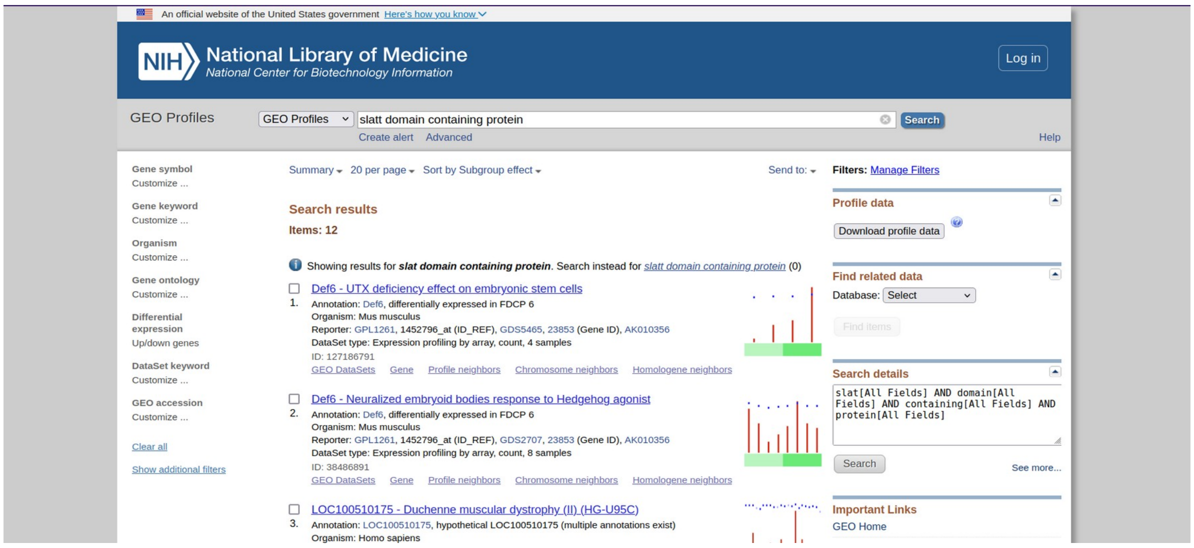
Popular Resources
PubMed
Bookshelf
PubMed Central
BLAST
Nucleotide
Genome
SNP
Gene
Protein
PubChem

NCBI News & Blog

NCBI Hidden Markov Models (HMM) Release 16.0 Now Available!
22 Aug 2024
Download release 16.0 of the NCBI

Quick & Easy Access to Mpox Data Through NCBI Virus
21 Aug 2024
The World Health Organization

the disposition of a tool that allows to consult the database from a local script, so it can be called by the main GeneSys desktop application. It turns out that PATRIC has a command line set of tools[28] compatible with Linux Debian systems which can be employed to consult information from the database when needed. So, as PATRIC gives the option to design simple bash scripts that can be launched from our main application with the specific parameters to consult given by the user, that database is the chosen one for our work.



An official website of the United States government Here's how you know

National Library of Medicine
National Center for Biotechnology Information

Log in

GEO Profiles | GEO Profiles | slatt domain containing protein | Search
Create alert Advanced Help

Gene symbol
Customize ...
Gene keyword
Customize ...
Organism
Customize ...
Gene ontology
Customize ...
Differential expression
Up/down genes
DataSet keyword
Customize ...
GEO accession
Customize ...
Clear all
Show additional filters

Summary 20 per page Sort by Subgroup effect Send to:

Search results

Items: 12

Showing results for ***slat domain containing protein***. Search instead for *slatt domain containing protein* (0)

1. Def6 - UTX deficiency effect on embryonic stem cells
Annotation: Def6, differentially expressed in FDCP 6
Organism: Mus musculus
Reporter: GPL1261, 1452796_at (ID_REF), GDS5465, 23853 (Gene ID), AK010356
DataSet type: Expression profiling by array, count, 4 samples
ID: 127186791
GEO DataSets Gene Profile neighbors Chromosome neighbors Homologene neighbors

2. Def6 - Neuralized embryoid bodies response to Hedgehog agonist
Annotation: Def6, differentially expressed in FDCP 6
Organism: Mus musculus
Reporter: GPL1261, 1452796_at (ID_REF), GDS2707, 23853 (Gene ID), AK010356
DataSet type: Expression profiling by array, count, 8 samples
ID: 38486891
GEO DataSets Gene Profile neighbors Chromosome neighbors Homologene neighbors

3. LOC100510175 - Duchenne muscular dystrophy (II) (HG-U95C)
Annotation: LOC100510175, hypothetical LOC100510175 (multiple annotations exist)
Organism: Homo sapiens

Filters: Manage Filters

Profile data
Download profile data

Find related data
Database: Select
Find items

Search details
```
slat[All Fields] AND domain[All
Fields] AND containing[All Fields] AND
protein[All Fields]
```
Search
See more...

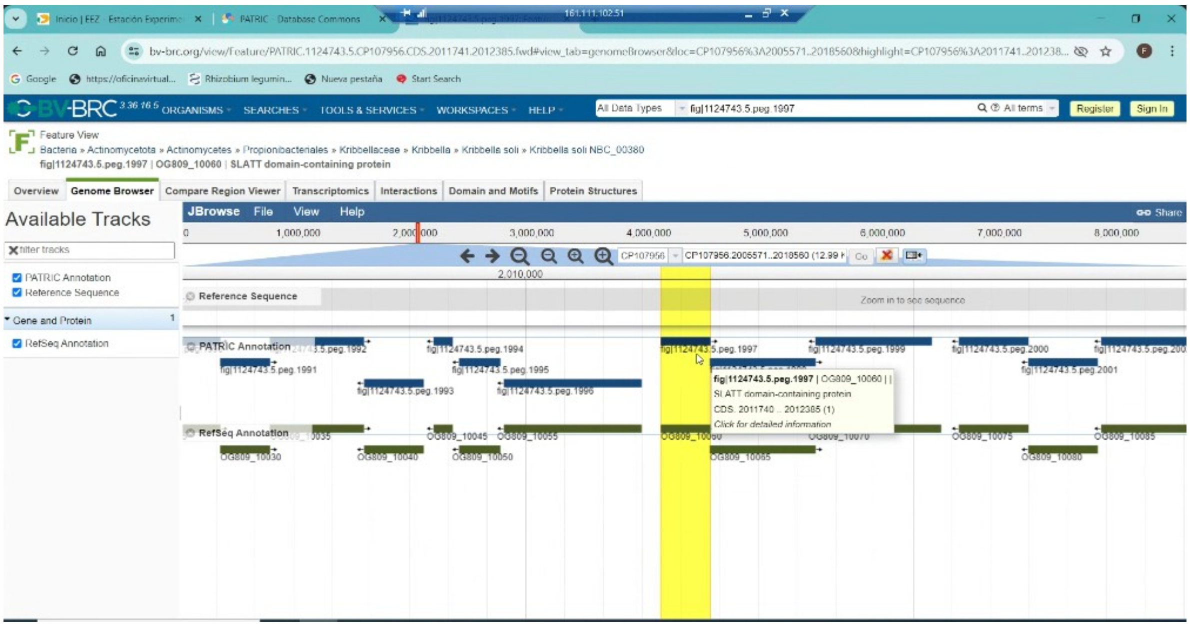
Important Links
GEO Home

As the main programming languages, Python and Kivy were selected for the development. Kivy[29] is a very comfortable language that allows the creation of user interfaces in a very customizable and intuitive way. On the other hand, Python was chosen for two major reasons. The first one is the large amount of libraries that it provides to easily manipulate bioinformatic data at a high abstraction level without requiring to write complex code, specifically, the Biopython package includes a lot of classes aimed to manipulate DNA, RNA and amino acid strings. The second reason is that Python also provides a full package to develop Kivy applications directly from Python code, which will be so helpful in the developing process, making it easier to integrate the components of the user interface with the logic that is spotted below them. Using Kivy package for Python, Kivy interface objects can be created by just calling to Python constructors. The final design of the interface can be detailed with extra Kivy files, too.

5. ARCHITECTURE AND DESIGN

Now it is going to be exposed how the application is actually designed.

It would be inefficient to implement an app exclusively aimed to solve RTs issue. Indeed, it would be much better to approach the problem from a more complete point of view. It is quite obvious that biology is a field in constant expansion that continually evolves and faces new challenges. We have stated previously that the volume of data related to bioinformatics



BV-BRC
ORGANISMS
SEARCHES
TOOLS & SERVICES
WORKSPACES
HELP
Feature View
Bacteria » Actinomycetota » Actinomycetes » Propionibacteriales » Kribbellaceae » Kribbella » Kribbella soli » Kribbella soli NBC_00380
fig|1124743.5.peg.1997 | OG809_10060 | SLATT domain-containing protein
Overview
Genome Browser
Compare Region Viewer
Transcriptomics
Interactions
Domain and Motifs
Protein Structures
Available Tracks
JBrowse
File
View
Help
PATRIC Annotation
Reference Sequence
Gene and Protein
RefSeq Annotation
Zoom in to see sequence
fig|1124743.5.peg.1997 | OG809_10060 | |
SLATT domain-containing protein
CDS: 2011740 .. 2012385 (1)
Click for detailed information

is increasing through time. As a result, it is not strange to suggest the development of a tool that can be easily expanded and is aimed to incorporate new features. Even though at first the app will only solve RTs' problem, if it is designed in a very extensible way, it will be possible to employ its framework as a starting point for future bioinformatic data manipulation tools, so that in the end GeneSys can incorporate a lot of functionalities aimed to solve many bioinformatic challenges.

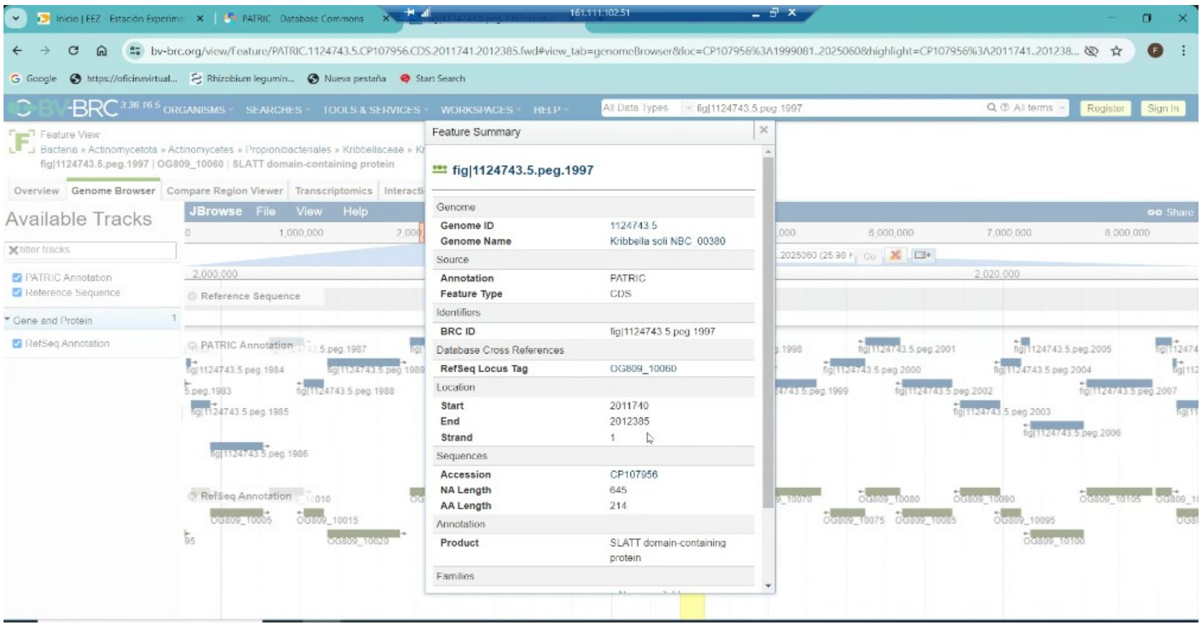
In order to accomplish that, it is crucial to design GeneSys attending to two major requirements.

- **Maintainability**. The software must be correctly designed so it is easy to update and modify through time, as well as having an easy-readable code and proper comments that explain how it works.
- **Extensibility**. It is fundamental to develop GeneSys as a software that allows the incorporation of future functionality aimed to solve new problems along with the RTs' experiment. All the incorporated new features must not change (or change as little as

possible) the existing code and architecture of GeneSys.

Also, other requirements must be satisfied. Among them, all those related with making the application desirable for a generic user are crucial: usability, consistency and accessibility. Security, scalability, interoperability, reliability, availability, portability efficiency are important as well, and deserve to be mentioned.

To sum up, the project seeks to develop a flexible, scalable software framework catering to the needs of bioinformatics experts, ultimately enhancing their research endeavors for current and future challenges they might face.



Feature Summary

fig|1124743.5.peg.1997

Genome	
Genome ID	1124743.5
Genome Name	Kribbella soli NBC_00380
Source	
Annotation	PATRIC
Feature Type	CDS
Identifiers	
BRC ID	fig|1124743.5.peg.1997
Database Cross References	
RefSeq Locus Tag	OG809_10060
Location	
Start	2011740
End	2012385
Strand	1
Sequences	
Accession	CP107956
NA Length	645
AA Length	214
Annotation	
Product	SLATT domain-containing protein
Families	

5.1. Class diagrams.

Image 5.1.1. GeneSys' class diagram.

Apart from the initial GenesysApp class, which starts the Kivy application itself, there are two major sections in which the app is divided: Screens and Modules.

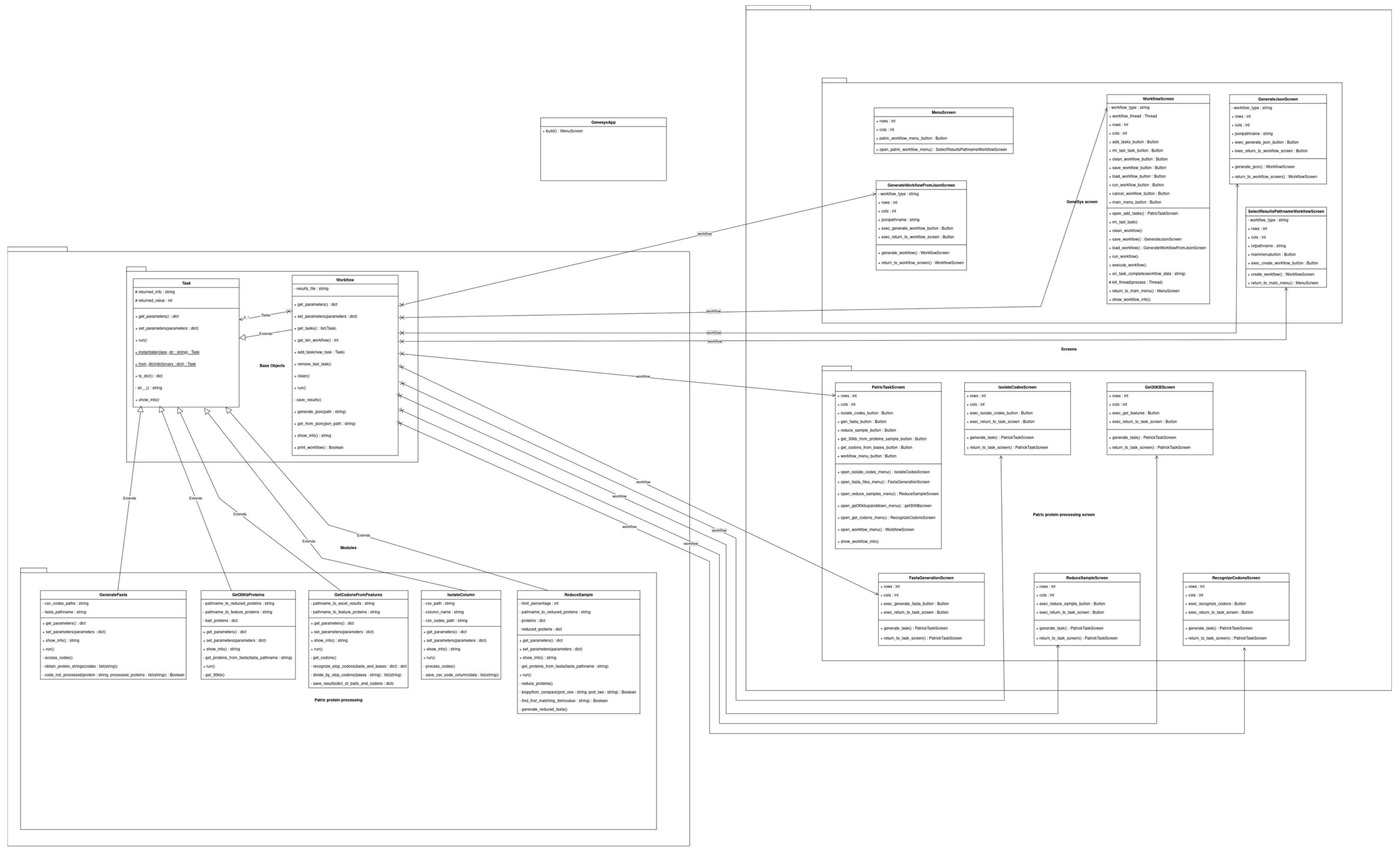
Screens contains all the Kivy interfaces defined in Python that show different windows, each one with a different function. Also, it is divided into two other sections. Firstly, GeneSys screen, corresponding to all the windows that are common to any module included in GeneSys aimed to solve any task.

- **MenuScreen** is the window that shows the user all the available issues for which a workflow can be defined. For now, it only has a button corresponding to the definition of a workflow to solve the RTs issue. It has been designed this way so the architecture does not change in case new modules are added in the future.

- **SelectResultsPathnameWorkflowScreen**. Once a problem has been selected in the previous window, this one opens and asks the user to introduce where a txt file with the execution results of the workflow should be created. All the executed workflows will create a file in the specified path that provides important information about all the stages of the workflow execution.
- **WorkflowScreen** is the window where the user finds all the possible functions to apply to the workflow. Here, new tasks can be added to the workflow, all workflow's tasks or just the last one added can be removed, the workflow can be executed, the execution can be canceled and the workflow can be completely destroyed by returning to the main menu. Also, it allows the user to save the workflow in JSON format and to load a workflow from a previously defined JSON file.
- **GenerateWorkflowFromJsonScreen** is the window that opens when the option to load a workflow from a JSON file is selected in the WorkflowScreen window. It asks the user the pathname from which load the workflow.
- **GenerateJsonScreen** is the window that opens when the option to save a workflow into a JSON file is selected in the WorkflowScreen window. It asks the user the pathname to which save the workflow.

Secondly, Patric protein processing screen, which defines all the windows related to the tasks that must be accomplished by the GeneSys module that solves the RTs issue. In the future, each new module that would be added to GeneSys will have a new section like this one, including all the screens corresponding to the definition of the tasks that the module implements.

- **PatricTaskScreen** is the window that opens when the user selects the option to add a new task to a workflow aimed to solve the RTs issue. It has five buttons, each one corresponding to a new window that allows the user to define the task to be added to the workflow.
- **IsolateCodesScreen** is the window that allows the definition of the task that isolates a specific column corresponding to the IDs of proteins downloaded from PATRIC.
- **FastaGenerationScreen** is the window that allows the definition of the task that obtains the amino acid sequences of a set of proteins given a csv file with an unique column that contains proteins' IDs.


GenesysApp
+ build() : MenuScreen
Base Objects
Task
# returned_info : string
# returned_value : int
+ get_parameters() : dict
+ set_parameters(parameters : dict)
+ run()
+ instantiate(class_str : string) : Task
+ from_dict(dictionary : dict) : Task
+ to_dict() : dict
- str__() : string
+ show_info()
Workflow
- results_file : string
+ get_parameters() : dict
+ set_parameters(parameters : dict)
+ get_tasks() : list(Task)
+ get_len_workflow() : int
+ add_task(new_task : Task)
+ remove_last_task()
+ clean()
+ run()
- save_results()
+ generate_json(path : string)
+ get_from_json(json_path : string)
+ show_info() : string
+ print_workflow() : Boolean
Tasks
0..*
Extends
Modules
Patric protein processing
GenerateFasta
- csv_codes_paths : string
- fasta_pathname : string
+ get_parameters() : dict
+ set_parameters(parameters : dict)
+ show_info() : string
+ run()
- access_codes()
- obtain_protein_strings(codes : list(string))
- code_not_processed(protein : string, processed_proteins : list(string)) : Boolean
Get30KbProteins
- pathname_to_reduced_proteins : string
- pathname_to_feature_proteins : string
- bait_proteins : dict
+ get_parameters() : dict
+ set_parameters(parameters : dict)
+ show_info() : string
- get_proteins_from_fasta(fasta_pathname : string)
+ run()
- get_30kb()
GetCodonsFromFeatures
- pathname_to_excel_results : string
- pathname_to_feature_proteins : string
+ get_parameters() : dict
+ set_parameters(parameters : dict)
+ show_info() : string
+ run()
- get_codons()
- recognize_stop_codons(baits_and_bases : dict) : dict
- divide_by_stop_codons(bases : string) : list(string)
- save_results(dict_of_baits_and_codons : dict)
IsolateColumn
- csv_path : string
- column_name : string
- csv_codes_path : string
+ get_parameters() : dict
+ set_parameters(parameters : dict)
+ show_info() : string
+ run()
- process_codes()
- save_csv_code_column(data : list(string))
ReduceSample
- limit_percentage : int
- pathname_to_reduced_proteins : string
- proteins : dict
- reduced_proteins : dict
+ get_parameters() : dict
+ set_parameters(parameters : dict)
+ show_info() : string
- get_proteins_from_fasta(fasta_pathname : string)
+ run()
- reduce_proteins()
- biopython_compare(prot_one : string, prot_two : string) : Boolean
- find_first_matching_item(value : string) : Boolean
- generate_reduced_fasta()
Screens
GeneSys screen
MenuScreen
+ rows : int
+ cols : int
+ patric_workflow_menu_button : Button
+ open_patric_workflow_menu() : SelectResultsPathnameWorkflowScreen
GenerateWorkflowFromJsonScreen
- workflow_type : string
+ rows : int
+ cols : int
+ jsonpathname : string
+ exec_generate_workflow_button : Button
+ exec_return_to_workflow_screen : Button
+ generate_workflow() : WorkflowScreen
+ return_to_workflow_screen() : WorkflowScreen
WorkflowScreen
- workflow_type : string
+ workflow_thread : Thread
+ rows : int
+ cols : int
+ add_tasks_button : Button
+ rm_last_task_button : Button
+ clean_workflow_button : Button
+ save_workflow_button : Button
+ load_workflow_button : Button
+ run_workflow_button : Button
+ cancel_workflow_button : Button
+ main_menu_button : Button
+ open_add_tasks() : PatricTaskScreen
+ rm_last_task()
+ clean_workflow()
+ save_workflow() : GenerateJsonScreen
+ load_workflow() : GenerateWorkflowFromJsonScreen
+ run_workflow()
+ execute_workflow()
+ on_task_complete(workflow_data : string)
# kill_thread(process : Thread)
+ return_to_main_menu() : MenuScreen
+ show_workflow_info()
GenerateJsonScreen
- workflow_type : string
+ rows : int
+ cols : int
+ jsonpathname : string
+ exec_generate_json_button : Button
+ exec_return_to_workflow_screen : Button
+ generate_json() : WorkflowScreen
+ return_to_workflow_screen() : WorkflowScreen
SelectResultsPathnameWorkflowScreen
- workflow_type : string
+ rows : int
+ cols : int
+ txtpathname : string
+ mainmenubutton : Button
+ exec_create_workflow_button : Button
+ create_workflow() : WorkflowScreen
+ return_to_main_menu() : MenuScreen
Patric protein processing screen
PatricTaskScreen
+ rows : int
+ cols : int
+ isolate_codes_button : Button
+ gen_fasta_button : Button
+ reduce_sample_button : Button
+ get_30kb_from_proteins_sample_button : Button
+ get_codons_from_bases_button : Button
+ workflow_menu_button : Button
+ open_isolate_codes_menu() : IsolateCodesScreen
+ open_fasta_files_menu() : FastaGenerationScreen
+ open_reduce_samples_menu() : ReduceSampleScreen
+ open_get30kbupanddown_menu() : get30KBscreen
+ open_get_codons_menu() : RecognizeCodonsScreen
+ open_workflow_menu() : WorkflowScreen
+ show_workflow_info()
IsolateCodesScreen
+ rows : int
+ cols : int
+ exec_isolate_codes_button : Button
+ exec_return_to_task_screen : Button
+ generate_task() : PatrickTaskScreen
+ return_to_task_screen() : PatrickTaskScreen
Get30KBScreen
+ rows : int
+ cols : int
+ exec_get_features : Button
+ exec_return_to_task_screen : Button
+ generate_task() : PatrickTaskScreen
+ return_to_task_screen() : PatrickTaskScreen
FastaGenerationScreen
+ rows : int
+ cols : int
+ exec_generate_fasta_button : Button
+ exec_return_to_task_screen : Button
+ generate_task() : PatrickTaskScreen
+ return_to_task_screen() : PatrickTaskScreen
ReduceSampleScreen
+ rows : int
+ cols : int
+ exec_reduce_sample_button : Button
+ exec_return_to_task_screen : Button
+ generate_task() : PatrickTaskScreen
+ return_to_task_screen() : PatrickTaskScreen
RecognizeCodonsScreen
+ rows : int
+ cols : int
+ exec_recognize_codons : Button
+ exec_return_to_task_screen : Button
+ generate_task() : PatrickTaskScreen
+ return_to_task_screen() : PatrickTaskScreen
workflow

- **ReduceSampleScreen** is the window that allows the definition of the task that reduces the sample of proteins so in the end the dataset contains only one sample of each evolutive branch.
- **Get30KBScreen** is the window that allows the definition of the task that obtains 30,000 nucleotide bases to right and to the left from a set of proteins that work as baits each.
- **RecognizeCodonsScreen** is the window that allows the definition of the task that searches for valid stop codons in a large set of nucleotides, recognizes the valid proteins associated with each bait and organizes the resulting matches in a final xlsx file that can be readable by researchers.

The Modules section includes the logic to solve the tasks that the user defines in the interface provided by the Screens section. For now, it includes two major parts.

- The **Task** class is the key of GeneSys' logic, from which all the classes in the Modules section inherit. It includes some abstract methods that are specifically implemented when the class is inherited.
- The **Workflow** class inherits from class, and defines how a workflow can be defined by the user and how it can be manipulated. A workflow is essentially defined as a list of tasks, which may also correspond to a workflow, which opens the window for implementing new modules in the future that might contain other workflows as tasks to be executed.

In essence, Task and Workflow are very flexible classes oriented to provide a first logic layer from which all the logic of any GeneSys module can be defined according to the challenge it tries to overcome.

The second part of the Modules section corresponds to the specific tasks defined in order to solve the RTs issue. Whenever a new module is implemented, a new section like this will be added to the application.

- **IsolateColumn** implements the logic given by the user in the IsolateCodesScreen associated window.
- **GenerateFasta** implements the logic given by the user in the FastaGenerationScreen associated window.
- **ReduceSample** implements the logic given by the user in the ReduceSampleScreen associated window.
- **Get30KbProteins** implements the logic given by the user in the Get30KBScreen associated window.

- **GetCodonsFromFeatures** implements the logic given by the user in the RecognizeCodonsScreen associated window.

The given permissions to all class' attributes and methods is always the most restrictive possible, for security reasons.

There are no sequence diagrams nor communication diagrams as the methods of all models tend to interact exclusively with their own classes or with just one object of another class. The cases where an object of a class calls to an object of another class are exposed verbally down below.

- The build method in GenesysApp calls to MenuScreen cosntructor.
- The open_patric_workflow_menu method in MenuScreen calls to SelectResultsPathname-WorkflowScreen constructor.
- The create_workflow and return_to_main_menu methods in SelectResultsPathname-WorkflowScreen call respectively to WorkflowScreen and MenuScreen constructors.
- The open_add_tasks, save_workflow, load_workflow and return_to_main_menu methods in WorkflowScreen call respectively to PatricTaskScreen, GenerateJsonScreen, GenerateWorkflowFromJsonScreen and MenuScreen constructors.
- The generate_workflow and return_to_workflow_screen methods in GenerateWorkflow-FromJsonScreen call both to WorkflowScreen constructors.
- The generate_json and return_to_workflow_screen methods in GenerateJsonScreen call both to WorkflowScreen constructors.
- The open_isolate_codes_menu, open_fasta_files_menu, open_reduce_sample_menu, open_get30KBupanddown_menu, open_get_codons_menu and open_workflow_menu methods in PatricTaskScreen call respectively to IsolateCodesScreen, FastaGenerationScreen, ReduceSampleScreen, Get30KBScreen, RecognizeCodonsScreen and MenuScreen constructors.
- The generate_task and return_to_task_screen methods in IsolateCodesScreen call both to PatricTaskScreen constructor.
- The generate_task and return_to_task_screen methods in FastaGenerationScreen call both to PatricTaskScreen constructor.
- The generate_task and return_to_task_screen methods in ReduceSampleScreen call both to PatricTaskScreen constructor.

- The generate_task and return_to_task_screen methods in Get30KBScreen call both to PatricTaskScreen constructor.
- The generate_task and return_to_task_screen methods in RecognizeCodonsScreen call both to PatricTaskScreen constructor.
- The process_codes method in IsolateColumn calls to csv.DictReader constructor.
- The access_codes method in GenerateFasta calls to csv.DictReader constructor.
- The save_results method in GetCodonsFromFeatures calls to pd.DataFrame constructor.

5.2. Folder organization.

Image 5.2.1. GeneSys’ folder organization.

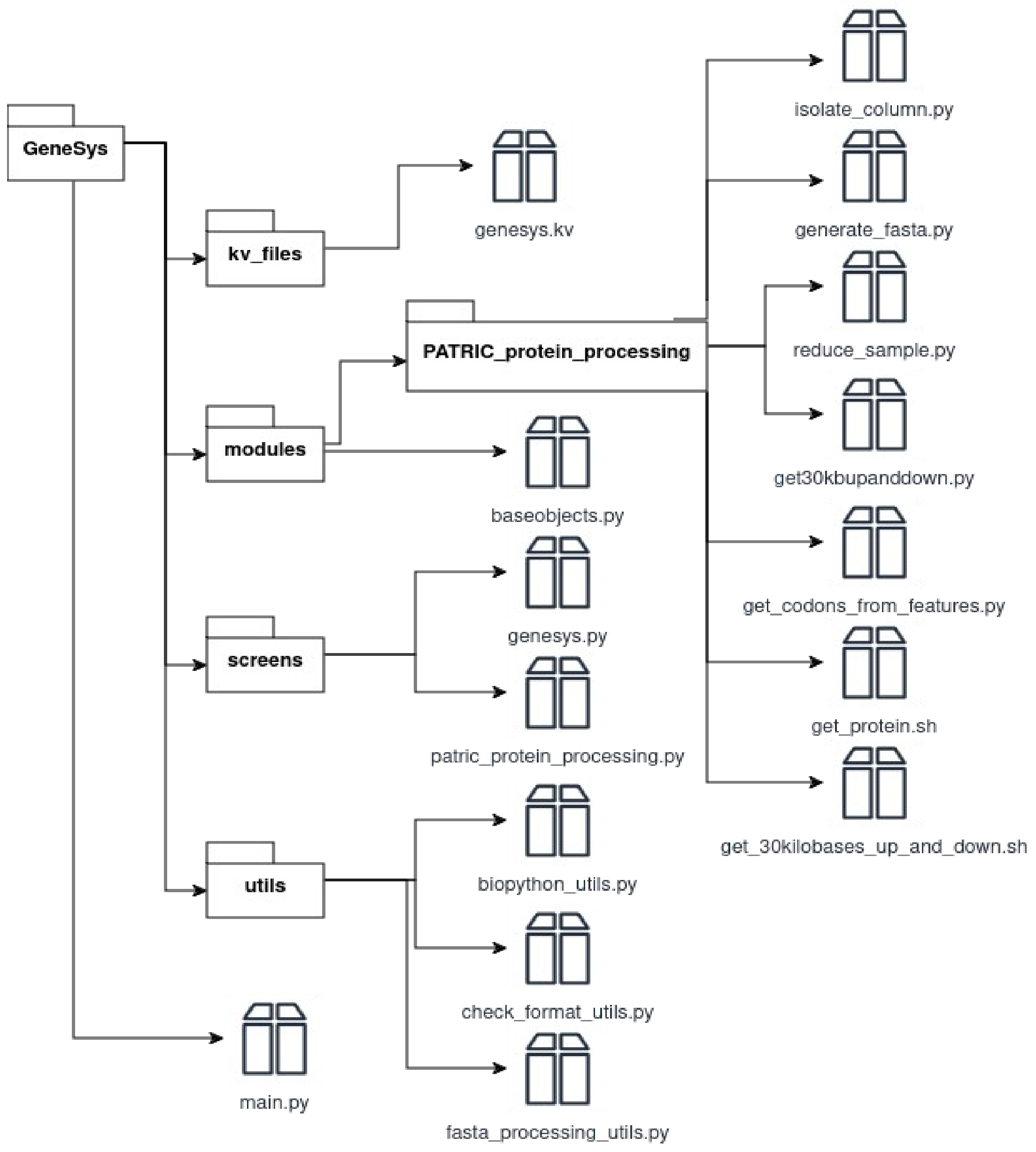
Concerning the folder disposition of the application, there is a main .py file at the root path that contains GenesysApp class. As a consequence, all the relative paths specified at any other file in the application will be given from the route at which the main.py file is located.

The kv_files folder contains a .kv file that specifies extra design specifications that affect to the user interface.

The modules folder includes all the implemented logic of the application. Inside, there are the baseobjects.py file, which contains the definition of the Task and Workflow base classes, and the folder associated with the tasks that are needed to solve RTs issue. In the future, when a new module is added to GeneSys, all its tasks will be defined inside a new folder in this path.

The PATRIC_protein_processing folder contains a .py file for each task that must be solved in the RTs issue. Also, it includes two basic bash scripts that are called while executing the tasks defined in generate_fasta.py and get30kbupanddown.py files. get_protein.sh receives a protein PATRIC ID as a parameter and calls to the specific command line BV BRC tool that returns the amino acid string corresponding to that protein ID, while get_30kilobases_up_and_down.sh receives a path to a fasta file and an undefined number of protein PATRIC IDs, calls to the command line tool that gets 30,000 nucleotides to the right and to the left of each of those IDs and stores the result in the specified fasta file.

The screens folder contains two Python files corresponding the first one to all the classes that define the screens common to all GeneSys modules (such as the workflow screen or the screen to save the workflow in a json file) and the second one to all the screens exclusive to the RTs issue (all the windows that allow users to manage the tasks associated with this



matter).

Finally, the utils folder does not contain any Python classes. Instead, it stores some generic functions that might be employed by any task in any module. It includes three files, each one corresponding to functions oriented to solve similar problems. biopython_utils.py has functions that employ classes from the Biopython package in order to cover specific necessities. check_format_utils.py is exclusively aimed to contain function that check the format of a path given by the user, such as csv, fasta, json… And fasta_processing_utils.py is oriented to provide generic functions that unify the way in which fasta files are read and written by all modules, apart from avoiding to rewrite the same code every time a task needs to fill or read a fasta file.

6. IMPLEMENTATION

6.1. GeneSys’ Kivy interface.

In Python, Kivy interfaces are managed through kivy Python library. This library disposes a set of classes that represent screen interaction objects. In order to add a specific object to the interface, it is necessary to inherit the Kivy class that represents the type of object that is going to be added. By defining the __init__ method of this classes, the aspect with which the elements will be displayed inside the selected object can be defined, as well as the elements the object itself will contain.

The only exception is the App class, which launches the main application using build() method, that returns the object that represents the main menu screen. Also, extra kivy files (which contain Kivy code that modifies the final look of the interface) can be added in the build method of the App class. In GeneSys, the final look of the interface is given by genesys.kv file contained in kv_files folder. This Kivy file sets the default aspect of buttons, text input boxes, pop-up windows and labels, as well as managing the padding and spacing between the elements in the implemented classes of the application.

The App class is executed by calling to its run() predefined method. In GeneSys, the App class that launches the application is contained in main.py, located in the root of the folder hierarchy of the application.

Apart from the App class, the rest of the classes that define GeneSys’ interface inherit from GridLayout Kivy class, which allows to organize Kivy objects in the screen by dividing the available space into rows and columns. The rows and cols attributes of GridLayout Kivy class can be set in the __init__ method in order to distribute the structure of the elements included in the screen.

Whenever a pathname to a file is collected from a text input box, its format is properly checked, and if it is not correct, an error message is displayed in the text input box instead. Also, all the text boxes implement an inner logic that establishes a pathname value by default in case the text input is given empty.

Inside the screens folder are located the genesys.py and patric_protein_processing.py. Both files contain GridLayout classes that define screens related to two different architectural aspects of the application.

6.1.1. Screens related to all implemented modules.

It has been mentioned in the previous section that GeneSys disposes a software layer that works as a base for the implementation of many different modules aimed to accomplish different bioinformatic tasks. As a result, the code that represents the interface related to the functions that the user can employ regardless of which module is employing are defined separately from those related to the tasks the user wants to accomplish. That module-shared interface is defined in the genesys.py file. Its content is described down below.

- **MenuScreen class.** This screen includes a button that allows users to select which kind of workflow they want to work with. In other words, it makes them select which module they want to use. As right now there is only one implemented module, it only has a button which allows the user to select to implement a workflow for managing RTs issue.

  Is **__init__** method calls to a function that cleans all the elements in the screen and adds a new widget corresponding to a SelectResultsPathnameWorkflowScreen object whose constructor requires a string parameter called “type”. This argument specifies for which module a workflow is going to be created, and consequently which tasks the user will be able to add to the workflow, corresponding uniquely to the tasks implemented in the module that is being used.

  For each new implemented module, a new button should be added, as well as an specific function that calls to the constructor of the next screen but changing the value of the “type” argument to the one that corresponds to the new module.

- **SelectResultsPathnameWorkflowScreen class.** This screen is called whenever a new workflow is going to be created from the main menu. Essentially, this screen asks users where they want to save the txt file that will contain information about the workflow’s execution. As a result, it contains two buttons, one that allows the user to return to the main menu and another one that creates a new workflow object with the specified pathname for the results file.

- **WorkflowScreen class.** This class receives a workflow and the type of the workflow as parameters in its constructor, and it offers all the available functions for manipulating workflows to users. It includes a scroll view that shows information about the workflow that is being currently processed, task by task. Each function that modifies the workflow can be accessed using a specific button. The available actions to use are the following.

  → Add tasks to the workflow: this button calls to a class’ method that checks the class attribute “type”, and returns the screen with the available tasks addable to the workflow depending on the module that is being employed.

→ Remove last task from the workflow. Removes the last task added to the workflow by calling to a method of the workflow object, and updates the scroll view to show the implemented changes.

→ Clean workflow. Removes all the tasks that are currently in the workflow, but does not remove the workflow object itself. It updates the scroll view to show the changes as well.

→ Save workflow in .json format. Returns a GenerateJsonScreen screen to save the current workflow in a json file.

→ Load workflow from a .json file. Returns a GenerateWorkflowFromJsonScreen to load a new workflow by reading a json file.

→ Run workflow. Calls to the run method of the workflow object and stores its execution in a Thread object that is an attribute of the class. Doing so, the workflow starts its execution, but the application does not freeze as its execution thread is not occupied by the workflow’s execution. As users can access to all GeneSys functionality while the workflow is executing, it would be an error to allow them to relaunch the workflow while its running, as both threads may access to the same data without the proper defined multi-threading policy access. That is why the run button is disabled during the execution. But if a new screen is loaded and then users return to the workflow manipulation screen, the disabled buttons are enabled again by default. As a consequence, all the buttons that load a different screen are disabled during the execution of a workflow as well. Only the buttons for removing tasks from the workflow and the button to cancel workflow’s execution are enabled. When the workflow is completed, a new method is triggered using the Clock library, showing a pop-up box with an ending message, and all the disabled buttons are enabled again.

→ Cancel workflow. Only enabled while the workflow is executing. It kills the Thread object of the class, triggering the forced ending of the execution.

→ Return to main menu. Returns to the menu that allows the selection of the module to employ. By doing so, the current workflow object is removed.

- **GenerateWorkflowFromJsonScreen class.** It stores the current workflow as an attribute. Asks users to introduce a pathname to a json file. It includes a button to return to the workflow manipulation screen, as well as another one to load the corresponding json file and setting the workflow tasks from it before returning to the workflow manipulation screen by passing the modified workflow.

- **GenerateJsonScreen class.** It stores the current workflow as an attribute. Asks users to introduce a pathname to a json file. It includes a button to return to the workflow manipulation screen, as well as another one to save the current workflow in the corresponding json file before returning to the workflow manipulation screen.

6.1.2. Screens related to the module to process PATRIC proteins.

On the other hand, there is the patric_protein_processing.py file which implements the classes related to the interface that allows users to define the tasks to be executed concerning the manipulation of RTs, which is the main purpose to which GeneSys is aimed. All the future modules that may be implemented for GeneSys should respect this organization and have their interfaces defined in new Python files.

The classes defined in the file are listed down below.

- **PatricTaskScreen class.** This class receives a workflow in its constructor and allows users to select which tasks of the PATRIC protein processing module they want to add to the workflow using buttons. It includes a scroll view to visualize the current workflow. The available buttons to press in this screen are the following.

  → Isolate PATRIC codes. Opens the screen to add a new task that isolates a given column from a csv file and saves it in a new csv file.

  → Generate ".fasta" files. Opens the screen to add a new task that given a csv full of BV BRC protein IDs and obtain all the associated protein amino acid strings of those IDs in a new fasta file.

  → Reduce sample. Opens the screen to add a new task that takes a percentage and employs it to reduce a given sample of proteins in a fasta file to a shorter one.

  → Get 30 kilobases up and down from given proteins. Opens the screen to add a new task that takes a fasta file full of proteins and returns a new fasta file with the nucleotides corresponding to each one of those proteins plus another 30,000 nucleotides to the right and to the left.

  → Find protein codons from a set of genome bases. Opens the screen to add a new task that receives a fasta file full of protein baits with its associated nucleotides among which there are marked some candidates that might be valid surrounding proteins. It recognizes the surrounding proteins and store the results in an excel file.

→ Return to workflow menu. Open again the screen to manipulate the workflow, so it can be executed, tasks can be removed from it, etc.

- **IsolateCodesScreen class.** This class generates a screen that asks for a csv file and a column name from that csv, and generates an object of a task that will isolate that column in a new csv when executed. In the PATRIC database, proteins can be downloaded in csv format. It is mandatory that users download the data they want to work with in that format so that this task can be correctly done. Starting from that csv, the idea is to isolate the column that contains the proteins' IDs so they can be processed later. In PATRIC, protein codes are usually stored in a column called "BRC ID". As a result, that is the default value that is selected for the column, in case the user did not specify anyone.

  This screen and all the others that follow include two buttons. The first one generates the task object with the given parameters in the text boxes and adds it to the current workflow before returning to the task selection menu. The second one allows users to return to the task selection menu without modifying the current workflow.

- **FastaGenerationScreen class.** This class generates a screen that has two text boxes. One asks for a csv file with only one column corresponding to a set of PATRIC protein IDs. The other one asks for a pathname to a fasta file. This screen allows users to generate an object of a task that will read the csv and will save all the proteins associated to the given IDs in the fasta pathname. This screen will tend to employ as the csv file to read the one returned by the previous task, so if a IsolateColumn task is defined in the workflow right before this one, its parameter to the returned csv file will be automatically assigned as the csv file of this task, regardless of what users specified in the text box.

- **ReduceSampleScreen class.** This class generates a screen that has three text boxes. One asks for a fasta file with a set of BV BRC proteins. The second one asks for a fasta pathname where to save the results of the task. And the third one asks for a number between zero and one hundred, which specifies a similarity percentage. The screen allows users to generate an object of a task that will read the first fasta file, compare all the proteins contained in it between them and removing from the set those that are at least as similar as the specified percentage, and will save the non removed proteins in the second fasta pathname. This screen will tend to employ as the first fasta file to read that one returned by the previous task, so if a GenerateFasta task is defined in the workflow right before this one, its parameter to the returned fasta file will be automatically assigned as the first fasta file of this task, regardless of what users specified in the text box.

- **Get30KBScreen class.** This class generates a screen that has two text boxes. One asks for a fasta file with a reduced set of BV BRC proteins. The second one asks for a fasta pathname where to save the results of the task. The screen allows users to generate an object of a task that will read the first fasta file, obtain the surrounding 30,000 nucleotides of each protein contained in it to the right and to the left (up to 60,000 obtained nucleotides in total) and save the results in the second given fasta file. This screen will tend to employ as the first fasta file to read that one returned by the previous task, so if a ReduceSample task is defined in the workflow right before this one, its parameter to the returned fasta file will be automatically assigned as the first fasta file of this task, regardless of what users specified in the text box.
- **RecognizeCodonsScreen class.** This class generates a screen that has two text boxes. One asks for a fasta file with a set of proteins that work as baits and its corresponding surrounding nucleotides (in other words, a very large string of nucleotide bases that potentially contains RTs within it and also stores a known protein that is employed to recognize the aforementioned long amino acid sequence as a whole). The second one asks for a xlsx pathname where to save the results of the task. The screen allows users to generate an object of a task that will read the fasta file. If that fasta file has been generated using the previous task, then each nucleotides string will contain sequences marked in capital letters that are potential valid proteins that surround the bait. The task will check those sequences one by one, and select those that actually are a valid protein. Once all the samples have been processed, the results will be stored in the given xlsx file. This screen will tend to employ as the fasta file to read that one returned by the previous task, so if a Get30KbProteins task is defined in the workflow right before this one, its parameter to the returned fasta file will be automatically assigned as the fasta file of this task, regardless of what users specified in the text box.

6.2. Inner logic implemented in GeneSys.

Now we will dive into the inner logic that allows GeneSys to effectively accomplish the tasks selected by users in the interface. All of this logic is implemented in the modules folder.

6.2.1. Task and Workflow, the key classes that define GeneSys' logic.

There are two main classes relative to all modules implemented in GeneSys, Task and

Workflow. Both are the base that defines how the application logic is implemented and are defined in the baseobjects.py file.

- **Task class**. It is an abstract class that initially stores two protected attributes: returned_info and returned_value. In the abstract Task class, both attributes store a default value, but they will be filled once the task is executed and will provided useful information about the execution context of the task. The implemented methods of the class are listed down below.

  → __init__. A constructor that for now does nothing.

  → get_parameters. Abstract method that returns a dictionary of the attributes of the class with their current values.

  → set_parameters. Abstract method that receives a dictionary with the attributes of the class paired with values. The method fills the values of the attributes of the current task object with their respective values contained in the dictionary.

  → run. Abstract method that for now does nothing, but that in further inheritances of the Task class it will execute the task object under it is applied, providing an unique functionality depending of the task that is being executed.

  → instantiate. Class method that gets a string parameter that corresponds to the name of a certain task class. The method will try to instantiate dynamically an object of the specified class, and if the instantiation is successful, the created object will be returned.

  → to_dict. Public method that transforms the object under it is applied in a readable dictionary of values from which it can be instantiated, and returns it. For now, it will store a key called type whose value will be the name of the current class (the name of the class will be stored in the same format that is employed by the instantiate method) and the current parameters of the class (which are obtained by calling to get_parameters method).

  → from_dict. Class method that receives a valid dictionary of values and returns an instance of the task defined in that dictionary by calling sequentially to the methods instantiate and set_parameters.

  → str__. Private method that returns a string cast of a call to to_dict method. In practice, it returns information about the current task.

  → show_info. Abstract method that for now does nothing, but in the future will have the same function as str__ method but removing some attributes from the dictionary before casting it to string. The function of this method is to provide information about the current task specifically filtered to be readable in the user's interface scroll view. For example, if a task stores an attribute that corresponds to a very large list of proteins, it would not be a good decision to show that attribute to users in the scroll view, as it would obstruct the rest of the attributes values that might be more useful to check. This method can be then inherited to remove that attribute from the dictionary of values before showing it in the scroll view.

- **Workflow class**. It has a list of task objects that can be executed sequentially and a pathname to a results file that will contain information about the execution context. It is important to note that this class inherits from Task, so a workflow object will be always a task object, and as a result a workflow will store both returned_info and returned_value attributes and can potentially store other workflows in the list of tasks to execute. Even though the situation of having workflows as the tasks to execute in a workflow never happens in the current version of GeneSys, it is a valid scenario for future module implementations. The methods of the class are listed down below.

→ __init__. Receives a list of tasks and fills the task list of the current Workflow object.

→ get_parameters. Calls to the same method of the superclass and adds specific sentences to incorporate the new attributes of the Workflow class to the dictionary.

→ set_parameters. Calls to the same method of the superclass and adds specific sentences to read the new attributes of the Workflow class from the dictionary.

→ get_tasks. Returns the list of Task objects attribute of the workflow that represents a pipeline to execute.

→ get_len_workflow. Returns the length of the list of task of the workflow.

→ add_task. Receives a new Task object as a parameter and incorporates it at the end of the tasks list of the workflow.

→ remove_last_task. Removes the last task that was added to the workflow.

→ clean. Removes all the tasks that are in the list of tasks of the workflow.

→ run. Tries to execute all the tasks of the workflow one by one. For each executed task, takes its returned_info attribute and adds it to the returned_info attribute of the workflow, as well as the name of the executed task and its returned value. Finally, calls to save_results function.

→ save_results. Private function that opens the results file attribute of the workflow and fills it with the returned info and returned value attributes of the workflow.

→ generate_json. Takes a json pathname as a parameter. Goes through each stored task of the workflow and calls to the to_dict function, adding each task dictionary to a list of task dictionaries. Finally, opens the json path and fills the json file with the list of dictionaries by calling to json.dump fucntion of the json Python library.

→ get_from_json. Receives a json pathname as a parameter. If that path exists in the system, opens it and reads its content using json.load function. The obtained data should correspond to a list of tasks dictionaries. For each obtained task, a new Task object is instantiated using from_dict method and then it is added to the list of tasks of the workflow.

→ str__. Goes through each task object of the workflow and calls to its str__ method. Appends each str__ call to a same string variable that is finally returned as context information about the workflow itself.

→ show_info. Works exactly as the previous method but calling to the show_info methods of each task. This is the method that is employed to show information about the current workflow in the scroll views available in the user interface.

→ print_workflow. Calls to print Python function using the str__ method of the current workflow object as the given information to print.

6.2.2. Specific tasks related to the module to process PATRIC proteins.

Now that it has been explained how Task and Workflow work, it is time to focus in the specific inheritances of the Task class that define the tasks to accomplish by the RTs manipulation module. They are listed down below. Note that, even though it is not mentioned, both returned_info and returned_value parameters are continually modified in this classes while they are being executed in order to provide as many context information as possible to researchers, including the proper explanation of any problem that might happen during the execution.

- **IsolateColumn task**. This task is exclusively aimed to work with a csv file downloaded from PATRIC database website[21], which always contains a column corresponding to protein IDs. It defines a task that stores three extra private attributes: csv_path (str), column_name (str) and csv_codes_path (str). When executed, it will look for the column of the csv file that is equal to the column in its attributes, and stores it in the system in a new csv file. The available methods of the class are listed down below. The idea is to isolate the column of the csv that specifically corresponds to the protein codes with which researchers are going to work. This task exists separately from the others as it is assumed that researchers are interested in keeping a csv file in their systems that exclusively contains protein codes.

→ __init__. Calls to the same method of the superclass (which in practice does nothing as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). It also receives a csv path and a column name as parameters, and fills with them the csv_path and column_name attributes. The csv_codes_oath attribute takes the value of the given csv_path but adding the characters "_new" to its name.

→ get_parameters. Calls to the same method of the superclass, which returns a dictionary of values with the returned_info and returned_value attributes filled. Then it adds current object's attribute values to that same dictionary and returns it.

→ set_parameters. Calls to the same method of the superclass, which receives a dictionary of values as a parameter and fills the returned_info and returned_value attributes of the current object. Then, the rest of the attributes of the current object are filled by reading the rest of the elements contained in the dictionary.

→ show_info. Calls to the same method of the superclass (which in practice does nothing as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). Then it calls to the to_dict method of the current object and removes the returned_info and returned_value attributes from the dictionary right before returning it as a string value, giving users useful information about the task.

→ run. Puts the returned_value and returned_info attributes to -1 and empty string respectively. Then calls to process_codes method.

→ process_codes. Private method that opens the csv_path attribute and isolates the column_name attribute with all its values using csv.DictReader method from csv Python library. Then calls to save_csv_code_column, passing the isolated column as a parameter.

→ save_csv_code_column. Private method that receives a column of data to save, opens the csv_codes_path file and writes the received data in the it.

- **GenerateFasta task**. This task is exclusively aimed to work with a csv file generated by a previous IsolateColumn task. It defines a task that stores two extra private attributes: csv_codes_path (str) and fasta_pathname (str). When executed, it will fill fasta_pathname with the amino acid strings of the proteins read from the csv file. The available methods of the class are listed down below.

→ __init__. Calls to the same method of the superclass (which in practice does nothing as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). It also receives a path to a csv file and a path to a fasta file as parameters, and fills class' attributes with them.

→ get_parameters. Calls to the same method of the superclass, which returns a dictionary of values with the returned_info and returned_value attributes filled. Then it adds current object's attribute values to that same dictionary and returns it.

→ set_parameters. Calls to the same method of the superclass, which receives a dictionary of values as a parameter and fills the returned_info and returned_value attributes of the current object. Then, the rest of the attributes of the current object are filled by reading the rest of the elements contained in the dictionary.

→ show_info. Calls to the same method of the superclass (which in practice does nothing as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). Then it calls to the to_dict method of the current object and removes the returned_info and returned_value attributes from the dictionary right before returning it as a string value, giving users useful information about the task.

→ run. Puts the returned_value and returned_info attributes to -1 and empty string respectively. Then calls to access_codes method.

→ access_codes. Private method that reads csv_codes_path with csv.DictReader and gets the values of the first column of that csv (technically, the csv file should contain only one column corresponding to BV BRC protein IDs). Then calls to obtain_protein_strings method, passing the column values as a parameter.

→ obtain_protein_strings. Private method that receives a list of BV BRC protein IDs. It calls to subprocess Python library to create the fasta file given by fasta_pathname attribute using touch command. Opens the fasta file, declares an empty list of processed proteins and goes through all proteins IDs. For each ID, employs subprocess to execute a bash script that calls to a specific BV BRC command line tool[28] that gets the corresponding amino acid string of a protein given its ID. Calls to code_not_processed method, passing the obtained amino acid string and the processed protein list as parameters, and if the protein corresponds to a non processed sample, it is added to processed proteins list and saved in the fasta file using save_fasta_string function from the fasta_processing_utils GeneSys file. If the execution is correct, the fasta file is closed and the results attributes are properly updated.

→ code_not_processed. Private method that receives an amino acid string and a list of amino acid strings as parameters. It returns false if the amino acid sample is already in the list. If not, returns true.

- **ReduceSample task**. This task is exclusively aimed to work with a fasta file of proteins returned by a previous GenerateFasta task. It defines a task that stores five extra private attributes: limit_percentage (int), pathname_to_reduced_proteins (str), fasta_pathname (str), proteins (dict) and reduced_proteins (dict). When executed, it will reduce the proteins stored in fasta_pathname according to limit_percentage, and store the final sample of proteins in pathname_to_reduced_proteins file. Both proteins and reduced proteins are stored as attributes because they are accessed by more than one method of the class. The available methods of the class are listed down below.

  → __init__. Calls to the same method of the superclass (which in practice does nothing as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). It also receives as parameters the values corresponding to limit_percentage, fasta_pathname and pathname_to_reduced_proteins attributes.

  → get_parameters. Calls to the same method of the superclass, which returns a dictionary of values with the returned_info and returned_value attributes filled. Then it adds current object’s attribute values to that same dictionary and returns it.

  → set_parameters. Calls to the same method of the superclass, which receives a dictionary of values as a parameter and fills the returned_info and returned_value attributes of the current object. Then, the rest of the attributes of the current object are filled by reading the rest of the elements contained in the dictionary.

  → show_info. Calls to the same method of the superclass (which in practice does nothing as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). Then it calls to the to_dict method of the current object and removes the returned_info, returned_value, proteins and reduced_proteins attributes from the dictionary right before returning it as a string value, giving users useful information about the task.

  → get_proteins_from_fasta. Private method that calls to get_fasta_content function from fasta_processing_utils.py GeneSys file in order to assign the proteins contained in fasta_pathname attribute to proteins attribute.

  → run. Puts the returned_value and returned_info attributes to -1 and empty string respectively. Then calls to reduce_proteins method.

→ reduce_proteins. Private method that copies the proteins attribute values in a temporary list and goes through that list. For every protein in the list, goes through the temporary list again but starting to read from the protein that follows the current one that is being analyzed. Each protein obtained in the second loop is compared to the protein of the first loop using byopython_compare method. If the proteins are to similar to each other, then the protein from the second loop is deleted from the sample. Once all the similar proteins have been removed from the sample, the protein that is being analyzed in the first loop is obtained with its corresponding key in the proteins dictionary attribute of the class by calling to find_first_matching_item method, and then it is stored the reduced_proteins attribute. Finally, there is a call to the generate_reduced_fasta method.

→ biopython_compare. Gets two amino acid strings as parameters and compares both using get_coincidence_percentage function from biopython_utils.py GeneSys file. If the given similarity percentage returned by the function is bigger than limit_percentage attribute, then returns true. Otherwise, returns false. If it returns true, then both proteins are considered to be closed in terms of evolution, and consequently one of them must be removed from the sample in order to end up isolating just one sample of each evolutive branch.

→ find_first_matching_item. Private method that gets an amino acid string as a parameter, and returns the tuple (key, value) corresponding to the first element in proteins dictionary attribute that has that parameter as a value associated with a key.

→ generate_reduced_fasta. Creates the pathname_to_reduced_proteins fasta file in the system and fills it with the values in reduced_proteins attribute by repeatedly calling to save_fasta_string function of fasta_processing_utils.py GeneSys file.

- **Get30KbProteins task**. This task is exclusively aimed to work with fasta file of proteins returned by a previous ReduceSample task. It defines a task that stores two extra private attributes: pathname_to_reduced_proteins (str) and pathname_to_feature_proteins (str). When executed, it will transform each protein from the fasta file into its corresponding nucleotide string with up to 30,000 extra nucleotide bases to the right and to the left of the protein, which will now work as a bait to recognize all those nucleotides. The available methods of the class are listed down below.

  → **__init__**. Calls to the same method of the superclass (which in practice does nothing, as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). It also receives the corresponding parameters to fill the attributes of the class.

→ get_parameters. Calls to the same method of the superclass, which returns a dictionary of values with the returned_info and returned_value attributes filled. Then it adds current object's attribute values to that same dictionary and returns it.

→ set_parameters. Calls to the same method of the superclass, which receives a dictionary of values as a parameter and fills the returned_info and returned_value attributes of the current object. Then, the rest of the attributes of the current object are filled by reading the rest of the elements contained in the dictionary.

→ show_info. Calls to the same method of the superclass (which in practice does nothing as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). Then it calls to the to_dict method of the current object and removes the returned_info and returned_value attributes from the dictionary right before returning it as a string value, giving users useful information about the task.

→ get_proteins_from_fasta. Private method that receives a fasta_pathname as a parameter, reads it by calling to get_fasta_content function from fasta_processing_utils.py GeneSys file and returns the obtained dictionary of proteins.

→ run. Puts the returned_value and returned_info attributes to -1 and empty string respectively. Then calls to get_30kb method.

→ get_30kb. Private method that uses subprocess.run to call to a bash script that employs a BV BRC command line tool to generate a new fasta file with the nucleotides sequences obtained from the protein baits. It executes that bash script with a bunch of arguments composed by the name of the fasta file where to save the results, corresponding to pathname_to_feature_proteins attribute, plus all the IDs of the proteins that work as baits, obtained by isolating the keys of the dictionary returned by get_proteins_from_fasta method.

- **GetCodonsFromFeatures task**. This task is exclusively aimed to work with a fasta file of nucleotide bases returned by a previous Get30KbProteins task. That task should have generated a fasta file where the header of each entrance corresponds to the ID of a protein bait, and the body of each entrance is a very long chain of nucleotides where all the candidates to be valid proteins are nucleotides marked in capital letters, while the rest are not in capital letters. It defines a task that stores two extra private attributes: pathname_to_feature_proteins (str) and pathname_to_excel_results (str). When executed, it will recognize all the proteins that surround each bait in the pathname_to_feature_proteins fasta file and will store the results in the pathname_to_excel_results excel file. Excel is employed because it is a format that can be easily manipulated by researchers. The available methods of the class are listed down below.

→ __init__. Calls to the same method of the superclass (which in practice does nothing as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). It also receives the corresponding parameters to fill the class' attributes.

→ get_parameters. Calls to the same method of the superclass, which returns a dictionary of values with the returned_info and returned_value attributes filled. Then it adds current object's attribute values to that same dictionary and returns it.

→ set_parameters. Calls to the same method of the superclass, which receives a dictionary of values as a parameter and fills the returned_info and returned_value attributes of the current object. Then, the rest of the attributes of the current object are filled by reading the rest of the elements contained in the dictionary.

→ show_info. Calls to the same method of the superclass (which in practice does nothing as the corresponding abstract method of Task does nothing, but it is still called in order to keep the programming good practices). Then it calls to the to_dict method of the current object and removes the returned_info and returned_value attributes from the dictionary right before returning it as a string value, giving users useful information about the task.

→ run. Puts the returned_value and returned_info attributes to -1 and empty string respectively. Then calls to get_codons method.

→ get_codons. Private method that reads pathname_to_feature_proteins fasta file by calling to get_fasta_content function from fasta_processing_utils.py GeneSys file, which returns a dictionary of the fasta file. It then passes the dictionary to recognize_stop_codons method as a parameter, which returns a new dictionary where the keys are the baits IDs and the values are lists of recognized proteins that surround each bait. Finally, it passes that dictionary as a parameter to the save_results method.

→ recognize_stop_codons. Private method that receives a dictionary of baits and all the bases associated to each. Goes through the keys and values of the dictionary and, for each tuple calls to divide_by_stop_codons method passing the value (corresponding to the nucleotide string) as a parameter. The method returns a list with all the recognized valid proteins contained in the string of nucleotides already converted into amino acid strings. That list is added with its corresponding key in a new dictionary. Finally, that new dictionary is returned.

→ divide_by_stop_codons. Private method that receives a long string of nucleotide bases as a parameter. Reads the string sequentially and every time it finishes reading a sequence of capital letters, calls to the has_valid_stop_codon function of the biopython_utils.py GeneSys file. If no a valid stop codon is found, the isolated nucleotide sequence of capital letters is transformed into its equivalent amino acid sequence by calling to from_bases_to_aminoacid function also included in biopython_utils.py, and includes the returned amino acid string in a list. Finally, the list of valid amino acid sequences is returned.

→ save_results. Private method that takes a dictionary whose keys are protein baits and its values are lists of valid proteins that surround each bait as a parameter. It generates a data frame from the dictionary by calling to the DataFrame function of pandas Python library. Finally, saves that dataframe object into pathname_to_excel_results attribute by calling to to_excel Python method.

Two bash scripts have been already mentioned in this section. Both are employed to call to specific options from the BV BRC available command line tools[28]. The bash scripts are detailed down below.

- **getprotein.sh**. It receives an unique argument corresponding to a BV BRC protein ID. Executes the "p3-echo $1 | p3-get-feature-data --attr aa_sequence" command inside an echo sentence so that the the result of applying the tool p3-get-feature-data to the protein ID is printed. p3-get-feature-data tool returns a matrix with information about the protein. It is asked to obtain only the column corresponding to the protein amino acid sequence by specifying the option --attr --aa_sequence.
- **get_30kilobases_up_and_down.sh**. A little more complex script than the other one. It receives a fasta file as the first argument, ensures that the file exists in the system employing touch command and saves it in a variable called $path. Then goes through the rest of the given arguments, each one referring to a protein BV BRC ID, and for each one given as $code, executes an echo sentence of the command "p3-echo $code | p3-get-feature-regions --distance 30000 >> $path". p3-get-feature-regions tool returns all the surrounding nucleotides of the specified protein, which works as a bait, with all the nucleotide subsequences that might correspond to valid proteins already marked in capital

letters. It is asked to obtain up to 30,000 nucleotides to the right and to the left of the bait by specifying the option --distance 30000.

6.3. Utils folder.

The utils folder, as it has been mentioned in previous sections, contains some Python files with generic functions that might be employed by more than one GeneSys module. For now, the folder contains three Python files, each one containing functions concerning a same topic or purpose.

6.3.1. Biopython related utils.

This file includes functions concerning the Biopython library. The idea is that each task that needed to employ Biopython functionality to accomplish a specific task would eventually call to a function contained in this file. The available functions are listed down below.

- **get_coincidence_percentage**. Given two proteins, constructs an Align.PairwiseAligner object from Biopython library and to its align method. The it accesses to the returned matches and returns the percentage of the length of the longest of the two proteins that corresponds to the total length of the matches. In other words, the function returns the percentage of bases that are coincident between two given proteins.
- **has_valid_stop_codon**. Takes a string of DNA bases and checks if they correspond to a valid protein or not. First, it transforms the string into a Seq object from Biopython library, and then checks if the sequence ends by a valid stop codon. If not, it calls to the reverse_complement Seq class method to obtain the reversed sequence, and checks again if it ends by a valid stop codon. If it ends by a valid stop codon, the method proceeds to check if the bases string length is a multiple of three. If so, then looks for stop codons that are not at the end of the sequence. If no stop codon is found in the middle of the sequence, then it is a valid string. If any of the already mentioned conditions is not true, then the sequence is not a valid string.
- **is_stop_codon_dna**. Checks if a given sequence of DNA bases corresponds to a stop codon (TAA, TAG or TGA).
- **from_bases_to_aminoacid**. Receives a sequence of bases, transforms the string into a Seq object from Biopython library, and then calls to the translate Seq class method to obtain

the equivalent amino acid sequence, and returns it.

6.3.2. Format checking utils.

This functions are employed in the user interface and are exclusively aimed to properly check all the formats given by the user in the text boxes. The available functions are listed down below.

- **check_fasta_format**. Checks if a pathname given as a string parameter ends with ".fasta" characters.
- **check_json_format**. Checks if a pathname given as a string parameter ends with ".json" characters.
- **check_txt_format**. Checks if a pathname given as a string parameter ends with ".txt" characters.
- **check_csv_format**. Checks if a pathname given as a string parameter ends with ".csv" characters.
- **check_excel_format**. Checks if a pathname given as a string parameter ends with ".xlsx" characters.

6.3.3. Fasta processing utils.

This file includes proper functionality to manipulate fasta files. The available functions are listed down below.

- **save_fasta_string**. Takes a identifier, a string of values and an opened fasta file. Writes the received information in the fasta file following fasta format. Writes the identifier preceded by a ">" character, jumps into a new line and writes the string of values divided into lines of eighty characters each by calling to split_fasta_sequence function. The lines are then written in the fasta file.
- **split_fasta_sequence**. Receives a string of characters and divides it into a list of subsequences of up to eighty length characters each.
- **get_fasta_content.** Receives a fasta file path as a parameter, declares an empty dictionary, opens the fasta file and reads it line by line. Each time a ">" character is found, a new entrance from the fasta file is added to the dictionary, where the key is the identifier that follows the ">" character and the value is all the lines that follow the identifier, ignoring

the jumps between those lines, until a new identifier is found or the end of the fasta file is reached. Finally, returns the filled dictionary.

7. GENESYS USER’S GUIDE

8. CONCLUSIONS AND FUTURE WORK

Quizás renta explicar que el tipo de workflow debe estar especificado en la clase workflow, no en la interfaz, pero que no se cambia ahora porque solo con un módulo implementado no da lugar a ningún fallo.

9. BIBLIOGRAPHY

1. Reverse Transcriptases: From Discovery and Applications to Xenobiology

2. Factors that may predict next pandemic

3. Statistics Say Large Pandemics Are More Likely Than We Thought

4. Methods in molecular biology and genetics: looking to the future

5. DNA chemical compound

6. RNA chemical compound

7. What Is the Difference Between DNA and RNA?

8. Protein. Biochemistry

9. Translation of RNA to Protein

10. Reverse Transcription—A Brief Introduction

11. Data volume growth in genomics versus other disciplines

12. Mestre MR (2020) "Analysis of novel and unexplored groups of prokaryotic Reverse Transcriptases". Trabajo Fin de Master Universitario en Biotecnología (Universidad de Granada, curso 2019-20). University of Granada repository.

13. Calculadora de IRPF Andalucía

14. What is the average lifespan of a laptop?

15. How Long Does a Computer Monitor Last: Lifespan Expectations Explained

16. How Long Can A Smartphone Last? (With 6 Real Examples)

17. Asus VivoBook S15 OLED K5504

18. Precio medio de la electricidad por meses: 2024

19. Qué fibra es mejor en España en relación calidad precio

20. GitHub repository: GeneSys.

21. BACTERIAL AND VIRAL BIOINFORMATICS RESOURCE CENTER, BV BRC.

22. Systematic prediction of genes functionally associated with bacterial retrons and classification of the encoded tripartite systems

23. National Center for Biotechnology Information main page.

24. Protein Data Bank.

25. CD Genomics.

26. European Nucleotide Archive, ENA.

27. Mgnify.

28. PATRIC command line set of tools.

29. Kivy language documentation.