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, NCBI, PATRIC, Biopython, BV BRC, presence-absence matrix, slatt domain containing proteins.

Abstract:

Recent decades' advancements in biological research have brought numerous benefits to our understanding of nature and society's progress. However, these improvements have also generated a vast amount of biological data that must be processed in an affordable amount of time to remain valuable for researchers.

The majority of the problems are related to the processing 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.

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 tool that solves a real life issue involving the recognition of neighbor proteins. As an example of a problem about this matter, there is the work involving reverse transcriptases, also known as RTs, a family of proteins with significant research potential, addressed 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 research.

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, NCBI, PATRIC, Biopython, BV BRC, matriz de presencia-ausencia, proteínas con un dominio slatt.

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.

La mayoría de los problemas van de la mano al procesamiento 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 de ese estilo va a necesitar herramientas que le permitan repetir el proceso aplicado a los datos tantas veces como desee.

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 al reconocimiento de proteínas vecinas. El doctor Francisco Martínez-Abarca Pastor de la Estación Experimental del Zaidín (EEZ) de Granada, España, ya acometió en su momento un trabajo de este tipo relacionado con las reverso transcriptasas, una familia de proteínas con un gran potencial. 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

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 analysis**, 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 September the 6th 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 seeking for neighbor proteins 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 this issue involving the preprocessing of proteic 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.

Main 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 the end of nineteenth century and the appearance of Gregor Johann Mendel's works that a first theoretical basis for the principles of heredity was set. The birth of molecular biology can be considered with the discovery of the DNA structure as the basis of the genetic heredity. Since then, this field has continually adapted to answer new questions and to face new challenges.

As a result, molecular biology has gone 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. Good examples are: the uses of 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] among others. All of these improvements have provided invaluable benefits for society. Biology and, in particular, molecular biology are not just fields with history. They are fields with present and future.

Before getting deeper into biological concepts, we should know 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 be translated to 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 by the translation machinery (Ribosomes, transfer RNA, etc…) in intervals of three each called codons, and they can be read starting generally by AUG (start codon), that compound the aforementioned RNA (or DNA) sequence, onward and backward, which gives us up to six different ways (Open Reading Frames, ORFs) of getting a protein from a same DNA 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 (UAA, UAG or UGA), and serve to delimit the end of a protein[9].

1.2. Reverse transcriptases.

Reverse transcription refers to the process of turning RNA sequences into DNA. This process was discovered in the 70's by studying the so called retroviruses, which encode specific enzymes: the reverse transcriptases. Lately, these enzymes have been found in all forms of life. They are annotated as "RNA-dependent DNA polymerases", "reverse transcriptases" or "RTs"[10]. They are characterized by specific protein domains (0 to 7) that allow to identify all a family of related genes.

Reverse transcriptases have remarkable biotechnological applications, such as molecular cloning strategies or in the field of synthetic biology. Thus, their involvement in the detection of viral RNA in SARS-CoV-2 testings[1], has contributed to a final control of the world wide pandemic.

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 transcriptases 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 next generation sequencing (NGS) 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. Increasing volume of databases month by month, derives 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]. In the process of recognizing which proteins were RTs, Rodríguez Mestre needed to seek for neighbor proteins from the initial set of samples.

The results of the study were considered successful, and Martínez-Abarca decided to repeat the experiment once the databases were updated with new samples and with new types of enzymes (not only RTs). The problem is that in order to repeat the process, all the steps of preprocessing the raw data and recognizing neighbor proteins had to be done manually again, which required so much effort in terms of time to be worth.

GeneSys serves as a software tool that automatizes the preprocessing of the raw data downloaded from the databases and recognizes neighbor proteins, solving 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 does each expose:

- **Introduction**: this chapter explains briefly the context in which GeneSys is made. It provides a generic view of the problem and helps those readers with basic biology knowledge to understand the full context in which GeneSys is developed.
- **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 reasons that justify why they must be done that way.
- **Architecture and design**: it provides 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 researchers who may wish to employ GeneSys in their

investigations.

- **Conclusions and future work**: it evaluates the accomplished objectives and proposes improvements that can be made to the application in the future.

1.5. GitHub’s repository.

Access to GeneSys’ GitHub repository here.

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. An application that accomplishes properly the issue of studying the neighborhood of any family of genes.

In order to achieve the proposed goal, we can distinguish the following objectives in the development of the application:

- To develop a bug free logic that works exactly as the user needs it to work.
- To provide an intuitive yet attractive friendly-user interface.
- To properly recognize which tasks must be automated by the application, so that the user receives a window to modify certain parts of the proteins preprocessing in order to adapt their experiments freely while not losing the automatization benefits that GeneSys provides.
- To provide a software framework that unifies all the preprocessing tasks in an unique execution context. In other words, we do not want users to open any other application than GeneSys to achieve their goals.
- To make an application that does not freeze or crash when it is executing a long task.
- To properly inform users about what is happening in the pipeline, 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 the student was not familiarized with molecular biology concepts, it was crucial to properly understand what it consists of and 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 and recognized 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 RTs' seeking of neighbor proteins 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 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 Visual Studio 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[20]. 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.

The dataset employed for testing GeneSys' functionality consisted of a group of samples downloaded from PATRIC under the search of the term "slatt domain containing proteins", a family of proteins with a neighboring behavior similar to RTs.

For managing backups, a GitHub repository was created. At first, it was private, but by the end of the development it was made public[19].

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

The student decided to divide the preprocessing task that GeneSys would apply to the data into smaller tasks which would be executed in a workflow.

Also, he 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 that was implemented, a new Kivy screen was designed for getting its parameters.

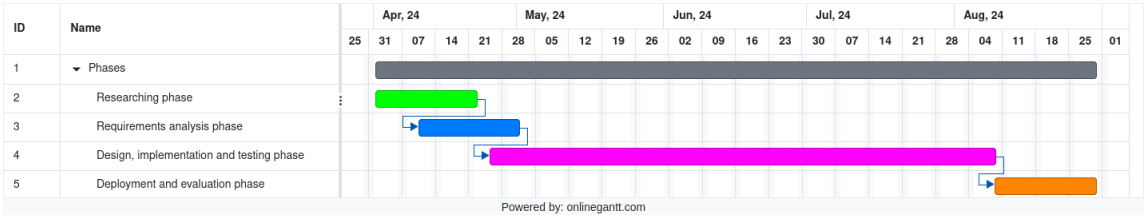
Throughout the process, there were cyclical testings of GeneSys' features so far. As a result, some modifications were applied to the structure of the code that leaded to the final design.

Finally, once the final task to accomplish worked properly, the efforts were putted into giving a prettier design to the user interface. Some 1x1 pixels colored images were created employing different tones of green. 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 budget.

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. Laptop characteristics.

Component	Characteristics
Laptop model	ASUSTeK COMPUTER INC. Vivo-Book_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

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.

Table 3.2.3.1. Software licenses.

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 (PSFL)

3.2.4. Indirect costs and materials.

The average electricity costs in Spain from April to August of 2024 are[17]:

→ 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 consumes 100 W per hour, our monitor consumes 30 W and our smartphone's consumption is too insignificant 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[18], 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. Final project budget.

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 involving RTs' is not the only one that focuses on a neighbor proteins experiment. Indeed, a similar article signed by Rodríguez Mestre himself along with Dr. Martínez-Abarca and others[21] was published back on December 2020.

This research was also aimed to find common behavior patterns between RTs that were classified in evolutive terms (phylogenetic relationships). In other words, the value of the study resided in discovering remarkable unknown correlations between RTs that despite of being far away from each other in terms of genetic evolution, presented similar amino acid patterns and therefore common biochemical roles.

In nature, proteins are encoded by genes within a DNA sequence. These DNA sequences in bacteria generally present a conserved gene order as indication of gene interaction for a particular biological function. As a result, the role of any specific gene should be related with the context where is present. One of the main researching tasks to accomplish with this matter is to recognize all the valid proteins contained in these chains, which usually implies to search for stop codons in the converted RNA/DNA sequence.

Any bacterial genome consists in a DNA information (well an unique replication or several contigs assembled). These DNA information also contain annotated all the genes (ORFs) present. Thus, every protein is identified by an ID. There are proteins known as "baits" that help researchers to recognize the nucleotide sequences that surround them. So, once the nucleotide sequences surrounding a bait are returned, researchers start looking for all the valid neighbors that are contained in the nucleotides surrounding the bait. And the baits can be found by seeking for their IDs.

In the study of Mestre et al 2020[12], 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 been considered noise and consequently they had to be avoided. The filter applied consisted in comparing each protein with the rest of the dataset, and if an equivalent protein at 85% sequence identity or more was found, then the selected protein was removed from the set. 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 in total for each

bait). Once all the nucleotides were isolated, the efforts were putted into finding all the neighbor proteins of each one of them. Finally, a clustering algorithm was applied to the final obtained set.

A comparison between two proteins can be done just by looking for common amino acid sequences spotted in both sequences. 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 looking at the clustering results. In this case, 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 five 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 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.

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 with different data.

Long story short, it is mandatory to automatize the study of neighbor proteins preprocessing. The tasks that compose such issue are stated down below.

4.2.1. Tasks to accomplish.

- **Isolate useful information from proteins downloaded from databases**. Usually, genetic databases incorporate a search bar and an online interface to download the data recovered after searching for a specific term in some easily manipulable format.

  Image 4.2.1.1. Interface of the National Center for Biotechnology Information's (NCBI) main page[22].

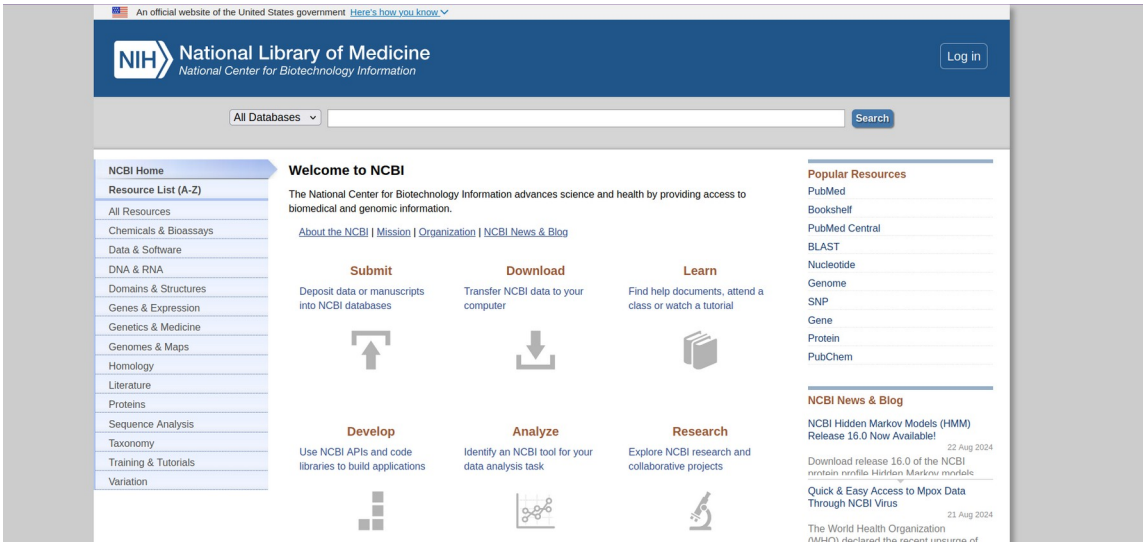


Image 4.2.1.2. Results available for download in NCBI's website.

In many cases, the samples are not downloaded 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' to get the amino acid strings that form the dataset.

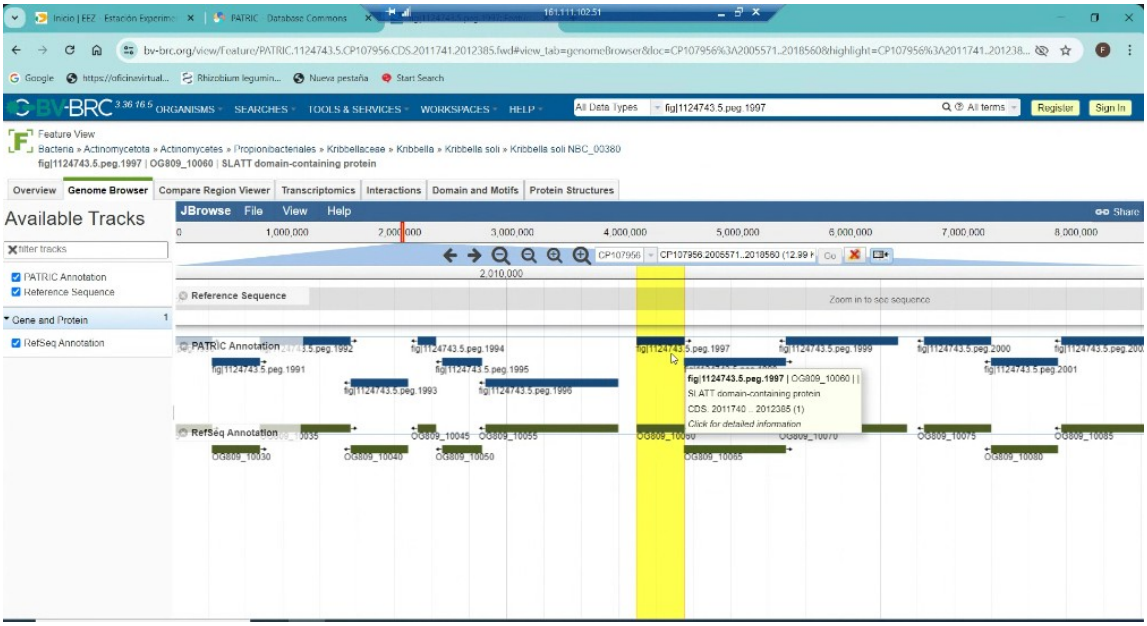
- **Obtain the proteins from its corresponding IDs**. In case we had downloaded protein IDs, we would have to find a tool that returns each protein's full amino acid sequence given its ID, and integrate that tool into our application. In case a protein is repeated in the final dataset, it should be removed.

- **Isolate one sample from each evolutive branch**. Once all the proteins are stored a amino acid sequences with no 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 a given percentage of coincidence specified by users. 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.

- **Get 30,000 nucleotides to the right and to the left from each sample**. The next phase consists of employing each protein as a bait from which obtain up to 30,000 bases of nucleotides to the right and to the left. 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, but phase should automatize the extraction of such nucleotides sequences and prevent researchers from doing it by themselves. In PATRIC's downloaded sequences, there are marked in capital letters those nucleotide subsequences that might correspond to a neighbor protein. That would be useful for the next step.

  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.

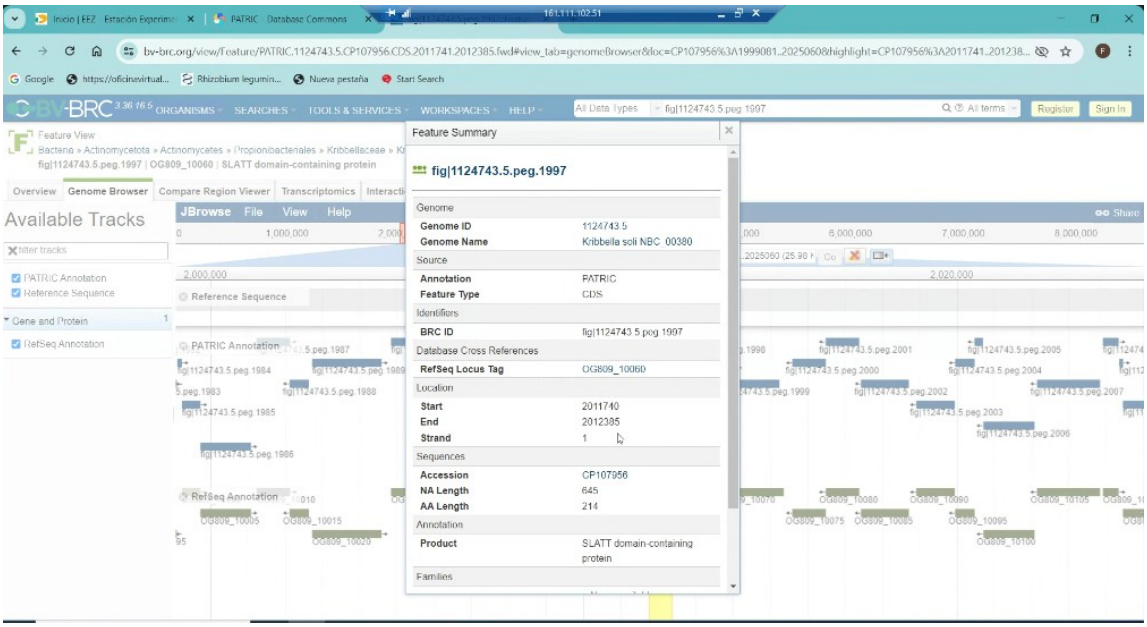


- **Recognize all the existing neighbor proteins in the nucleotides sequences associated to each bait**. Finally, before giving the preprocessed data to researchers so they can apply clustering, each obtained sequence of nucleotides must be analyzed in order to get all the neighbor proteins it stores. Fortunately, it has been previously mentioned that the classification of potential proteins in a large sequence of nucleotides is already done by PATRIC database, so all the bases downloaded in the previous step already have some fragments tagged as potential neighbor proteins. 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 not any 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 or if it has not a length equivalent to a multiple of three.

Image 4.2.1.4. An example of how PATRIC shows information about a certain fragment of a nucleotide 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 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[22] to others such as PDB[23] (Protein Data Bank), CD Genomics[24], the European Nucleotide Archive[25] (ENA) and Mgnify[26].

When deciding which database should be employed, the major characteristic to look for is 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[27] 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. A set of slatt domain containing proteins from that database was selected as testing data.

As the main programming languages, Python and Kivy were selected for the development. Kivy[28] 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 calling to Python constructors. The final appearance 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 proteins neighboring issue. Indeed, it would be much better to approach to 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 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 proteins neighboring problem, if it is designed in a modular 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.

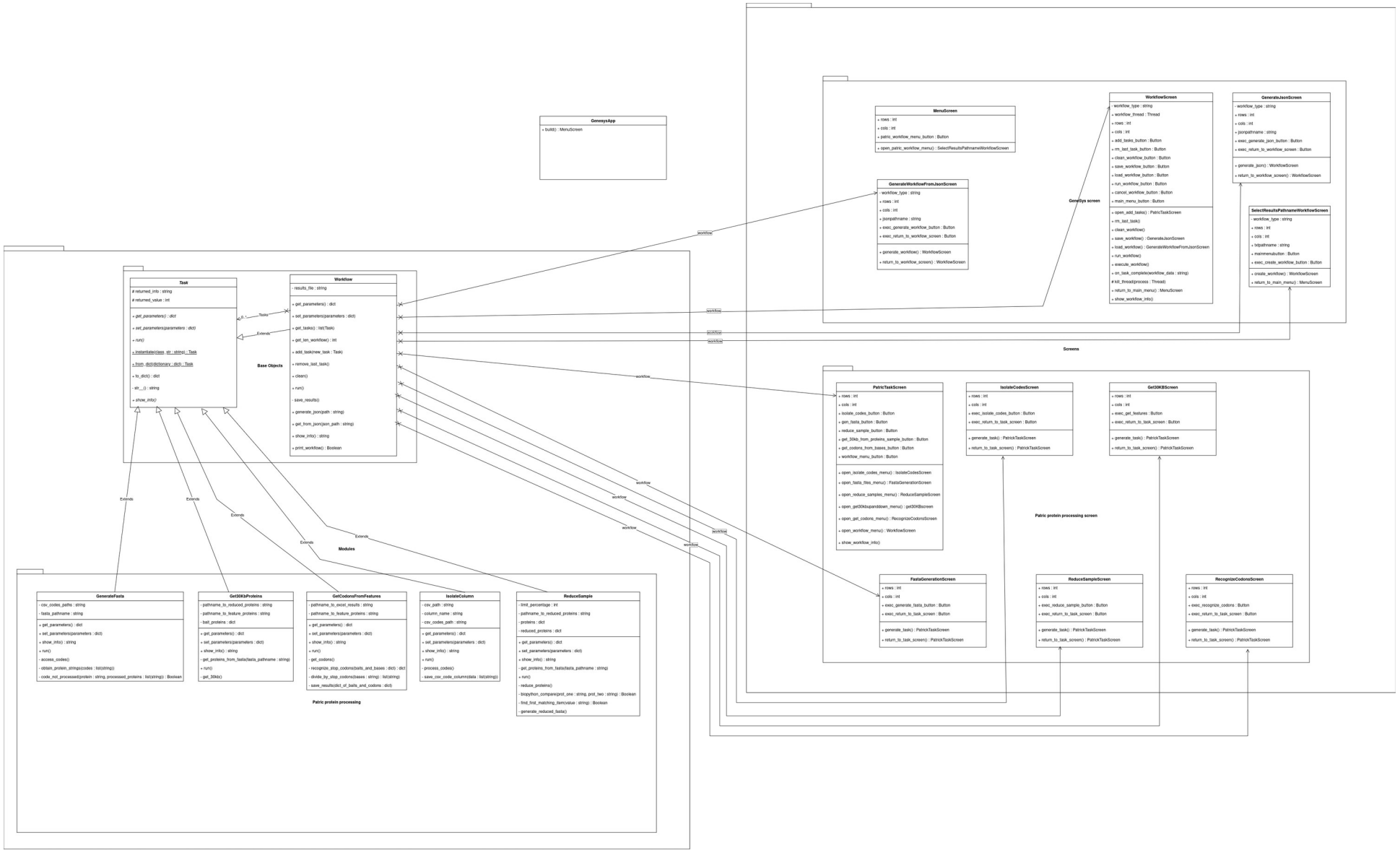
- **Maintainability**. The software must be correctly designed so it is easy to update and modify through time, as well as having an easy-readable code and proper comments that explain how it works.
- **Extensibility**. It is fundamental to develop GeneSys as a software that allows the incorporation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

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 and 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.

5.1. Class diagrams.

Image 5.1.1. GeneSys’ class diagram.


GeneysApp
+ build() : MenuScreen
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
GeneSys screen
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
Screens
workflow
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()
0..*
Tasks
Extends
Base Objects
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
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
Patric protein processing screen
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
Extends
Modules
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()
Patric protein processing

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 screens, 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 proteins neighboring issue. It has been designed this way so that 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 screen opens and asks the user to introduce a pathname to a txt file where the execution results of the workflow will be saved, providing 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 up 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 up 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, there is Patric protein processing screens section, which defines all the windows related to the tasks that must be accomplished by the GeneSys module that is selected. For now, there is only a module that solves the proteins neighboring issue. In the future, each new module that is added to GeneSys will add a new section like this one to the architecture, including all the screens corresponding to the definition of the tasks that the module implements. For now, these are the screens contained in the proteins neighboring module.

- **PatricTaskScreen** is the window that opens up when the user selects the option to add a new task to a workflow aimed to solve the proteins neighboring 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 a task that isolates a specific column corresponding to the IDs of a set of proteins downloaded from PATRIC.
- **FastaGenerationScreen** is the window that allows the definition of a task that obtains the amino acid sequences of a set of proteins given a csv file with an unique column that contains their proteins' IDs.

- **ReduceSampleScreen** is the window that allows the definition of a task that reduces the sample of proteins so in the end the dataset contains only one protein sample of each evolutive branch.
- **Get30KBScreen** is the window that allows the definition of a task that obtains 30,000 nucleotide bases to the right and to the left from a set of proteins that work as baits.
- **RecognizeCodonsScreen** is the window that allows the definition of a 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 excel file that can be then analyzed 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 workflows. This aspect makes possible to implement modules in the future that might contain other workflows as tasks to be executed, even though the proteins neighboring module does not employ this aspect.

In essence, Task and Workflow are flexible classes oriented to provide a first logic layer from which the behavior of any GeneSys module will 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 proteins neighboring issue. Whenever a new module is implemented, a new section like this will be added to the application's architecture.

- **IsolateColumn** implements the logic of the task defined in the IsolateCodesScreen associated window.
- **GenerateFasta** implements the logic of the task defined in the FastaGenerationScreen associated window.
- **ReduceSample** implements the logic of the task defined in the ReduceSampleScreen associated window.
- **Get30KbProteins** implements the logic of the task defined in the Get30KBScreen associated window.
- **GetCodonsFromFeatures** the logic of the task defined in the RecognizeCodonsScreen associated window.

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

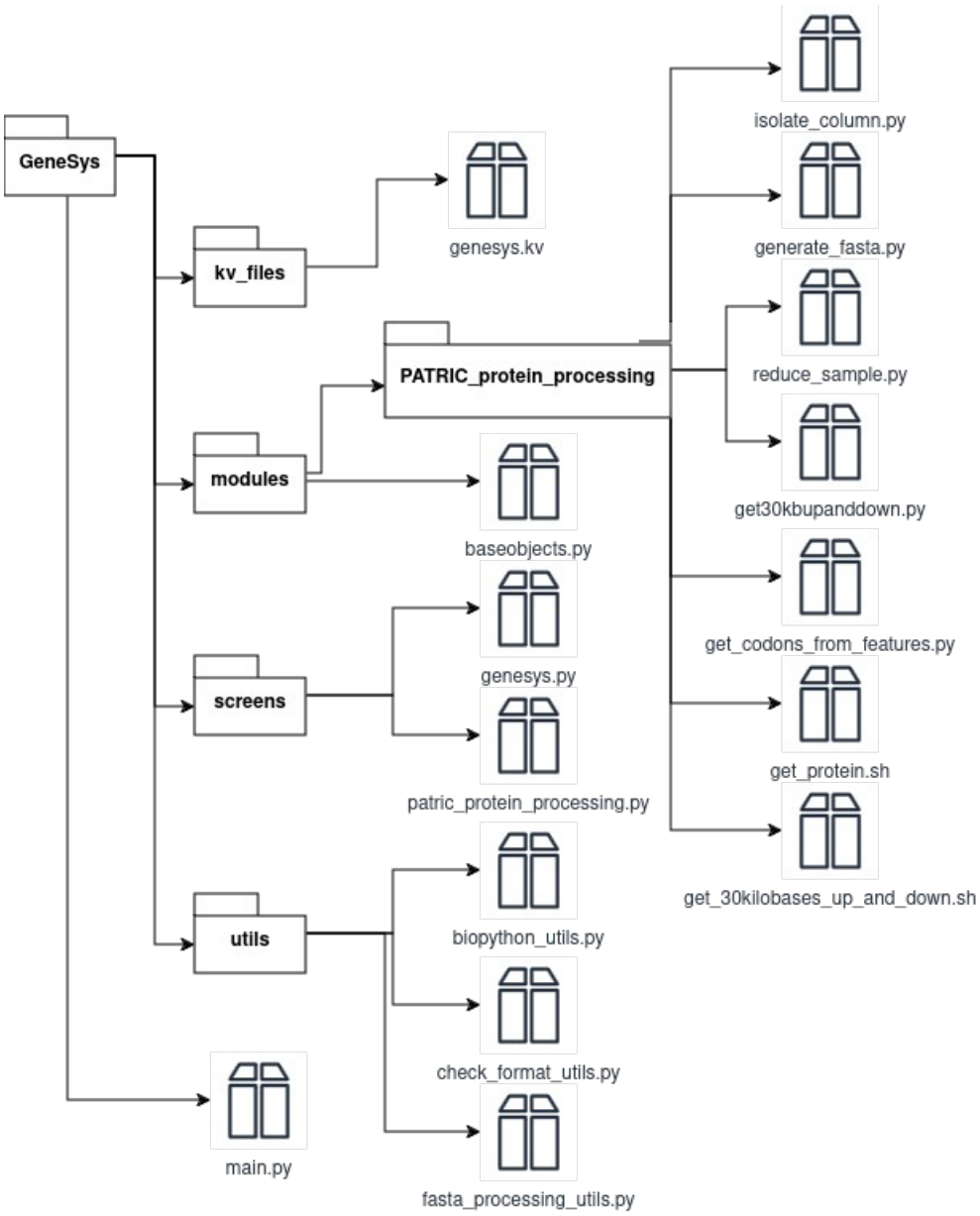
There are no sequence diagrams nor communication diagrams as the methods of all models tend to interact exclusively with their own classes or with just one object of another class. The cases where an object of a class calls to an object of another class are exposed verbally

down below.

- The build method in GenesysApp calls to MenuScreen constructor.
- 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 calls 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 appearance specifications 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 proteins neighboring 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 proteins neighboring issue. Also, it includes two basic bash scripts that are called while executing the tasks defined in generate_fasta.py and get30kbupanddown.py.

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 WorkflowScreen or the screen to save the workflow in a json file) and the second one to all the screens related to the proteins neighboring issue.

Finally, the utils folder does not contain any 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 functions 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, which avoids 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 these classes, the aspect with which the elements will be displayed inside the object can be defined, as well as the elements it 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 appearance 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 in 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 default value in case their text inputs are given empty.

Inside the screens folder are located the genesys.py and patric_protein_processing.py files. Both contain GridLayout classes that define the 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 basis for the implementation of many different modules aimed to accomplish different bioinformatic tasks. Thus, 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, there is only a button related to the implementation of a workflow for managing the PATRIC proteins neighboring issue.

  Its __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. This screen asks users where they want to save the txt file that will contain information about the workflow's execution. It contains two buttons, one that allows the user to return to the main menu and other that creates a new workflow object with the specified pathname to 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 such workflow. 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, so that whenever the workflow starts executing the application does not freeze. 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, then 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. As a result, only the buttons for removing tasks from the workflow and the button to cancel workflow's execution are enabled while it is running. 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 specified json file and setting the workflow tasks from its content before returning to the workflow manipulation screen.
- **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 neighboring PATRIC proteins. 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, obtains all the associated protein amino acid strings 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.

  → 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 of each.

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

→ Return to workflow menu. Opens 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 protein IDs to be processed later. In PATRIC downloaded csv files, protein IDs 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 and adds it to the current workflow before returning to the task selection menu. The second one returns 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 its contained IDs in the fasta pathname. If a IsolateColumn task is defined in the workflow right before this one, this screen will take the csv file to be read from the one returned by that previous 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 and remove 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 employ as the first fasta file to read that one returned by the previous task whenever a GenerateFasta task is defined in the workflow right before this one, 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 employ as the first fasta file to read that one returned by the previous task whenever a

ReduceSample task is defined in the workflow right before this one, 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 neighbor proteins of a known protein that is employed to recognize the aforementioned long amino acid sequence as a whole). The second one asks for an excel 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 nucleotide string will contain subsequences marked in capital letters that are potentially neighbor proteins surrounding the bait. The task will check those subsequences one by one, and will select those that actually are a valid neighbor protein. Once all the samples have been processed, the results will be stored in the given excel file. This screen will employ as the fasta file to read that one returned by the previous task whenever a Get30KbProteins task is defined in the workflow right before this one, 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 form the basis 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 may receive a different one after the task is done, depending on the execution context. The implemented methods of the Task 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 their values. Fills the attributes of the current Task object with the values contained in the dictionary.

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

  → instantiate. Class method that gets a string parameter that corresponds to the name of a

certain Task inherited 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 needs to be employed in 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 read by users through the graphic 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 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 occurs 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 sequentially. 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 task 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 the 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 proteins neighboring 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 good programming 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_path 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 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 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 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, providing 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 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 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, providing 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 (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[27] 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 the 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. 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 good programming 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 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 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, providing 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 too 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, 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 the 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 a 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 good programming 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 the 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 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, providing 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 neighbor proteins are nucleotides marked 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 format is employed because it 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 good programming 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 adds the 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 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, providing 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 the sequence is a valid neighbor protein, it 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 neighbor proteins 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 the 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[27]. 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 result of applying the tool p3-get-feature-data to the protein ID is printed. p3-get-feature-data tool returns a table 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 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 needs to employ Biopython functionality to address a specific issue 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 calls to its align method. Then accesses to the 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 checks if the bases string length is a multiple of three. If so, then seeks 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 protein. If any of the already mentioned conditions is not true, then the sequence is not a valid protein.

- **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 calls to the translate method to obtain its 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 users 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 an 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 characters length 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 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

This section serves as a generic user's manual for anyone who wishes to start employing GeneSys.

7.1. Main menu.

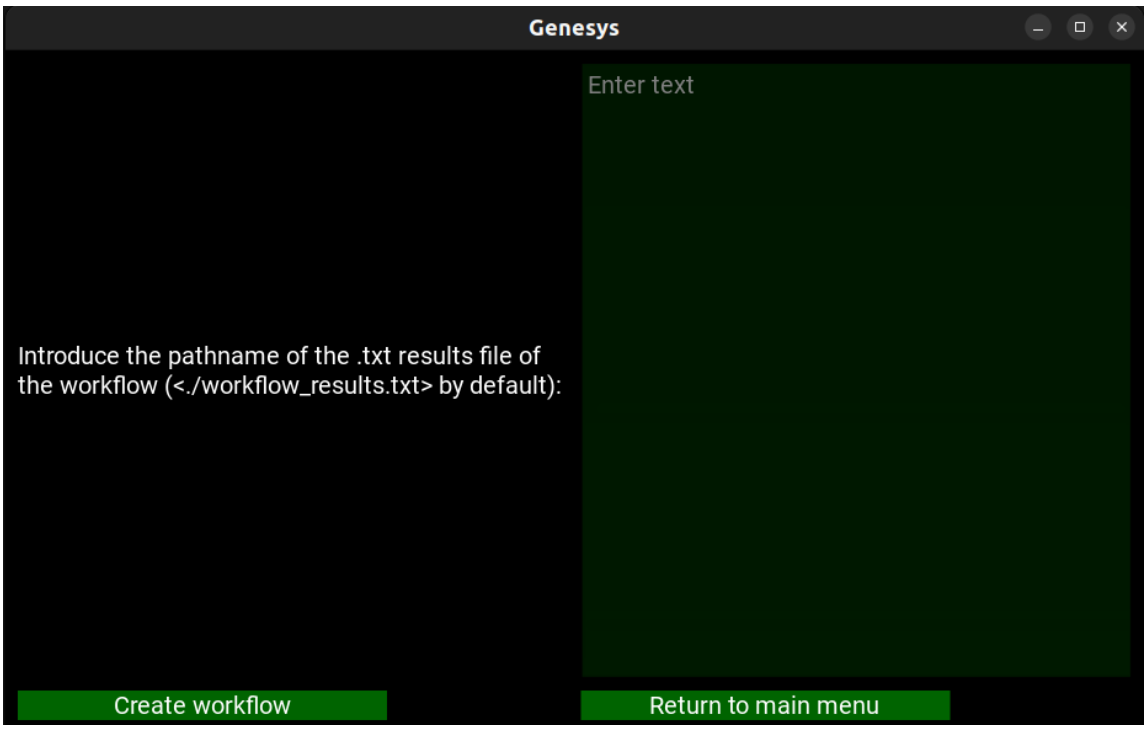
This is the screen that is showed when GeneSys is launched. It provides a button to select which kind of workflow users want to create. For now, it only stores one button corresponding to a PATRIC protein manipulation workflow.

Image 7.1.1. GeneSys' main menu.

7.2. Results file selection menu.

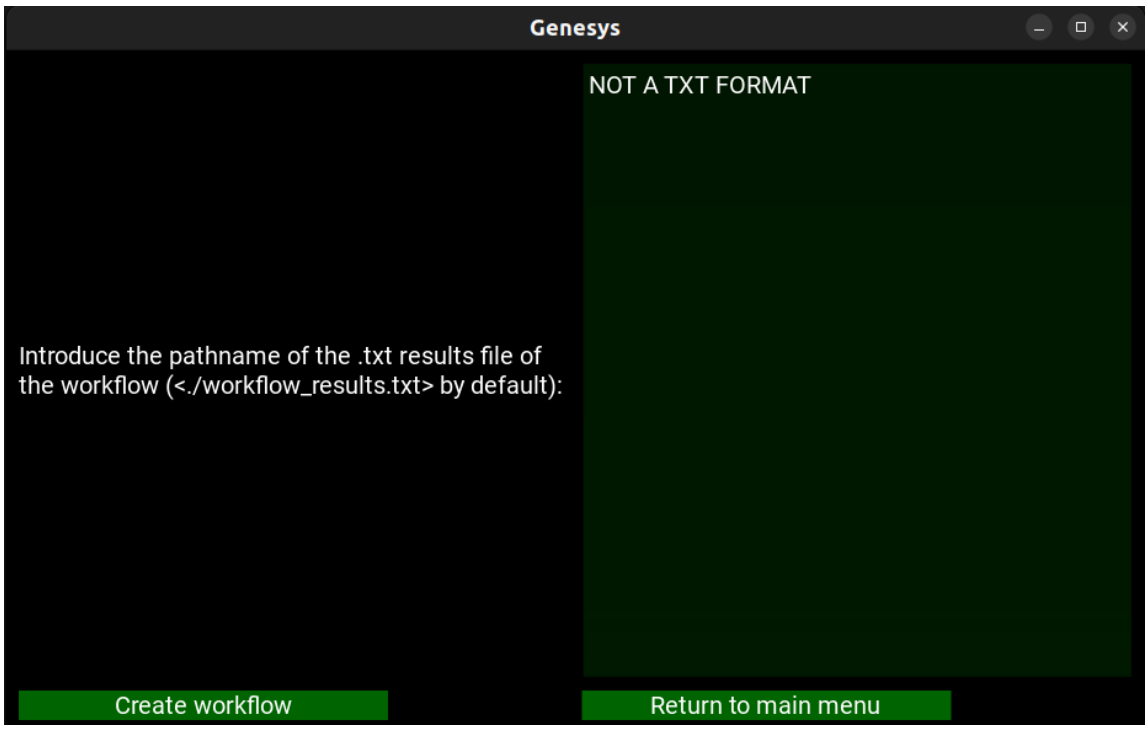
When any button from the main menu is clicked, this screen will appear, asking users to introduce the pathname to the results file where they want to save information about the workflow's execution. It includes a button to return to the main menu and a button to generate the workflow once a proper pathname has been given for the results file.

Image 7.2.1. Pathname to the workflow's results file menu.



The format of the results file must be txt. If an incorrect format is given and the button to generate the workflow is pressed, the content of the text box will change to show an error message. This will happen in any text box of GeneSys that asks for a certain format in a pathname.

Image 7.2.2. Incorrect format in the user interface text boxes.

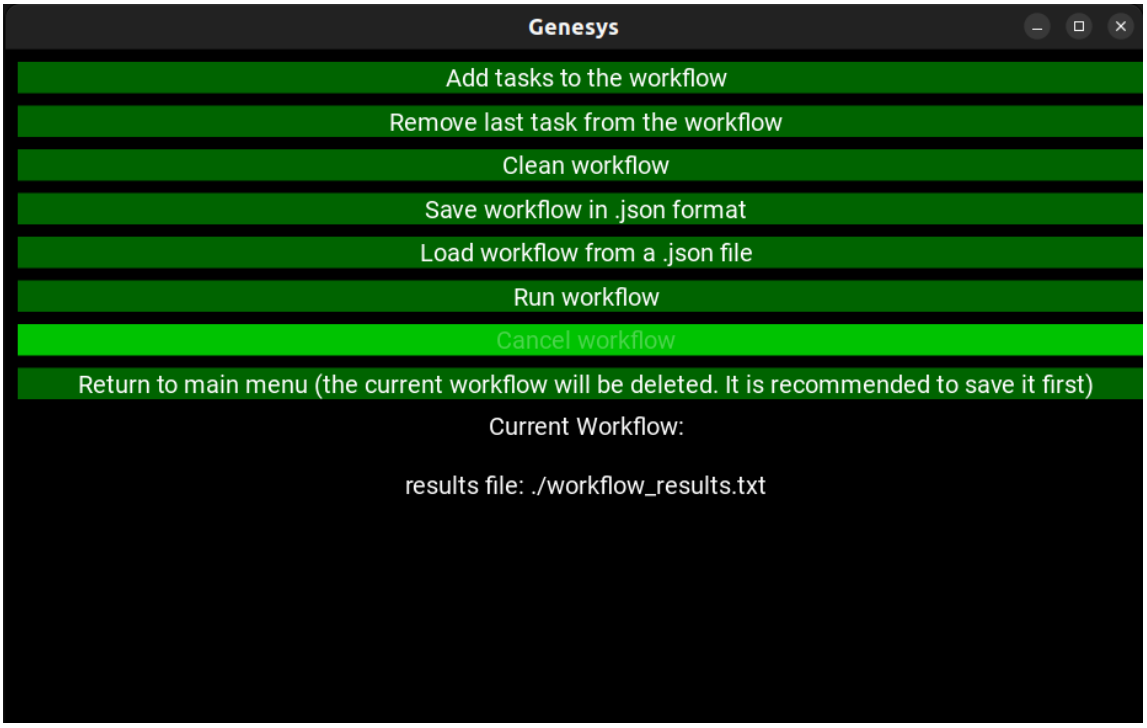


If the text box is leaved empty, the default pathname will be set as the relative route "./workflow_results.txt". In this example, this final option is the one selected.

7.3. Workflow manipulation menu.

Once the results file route has been correctly given, when clicking the “create workflow” button, the screen for manipulating the workflow will appear as follows.

Image 7.3.1. Workflow manipulation menu.

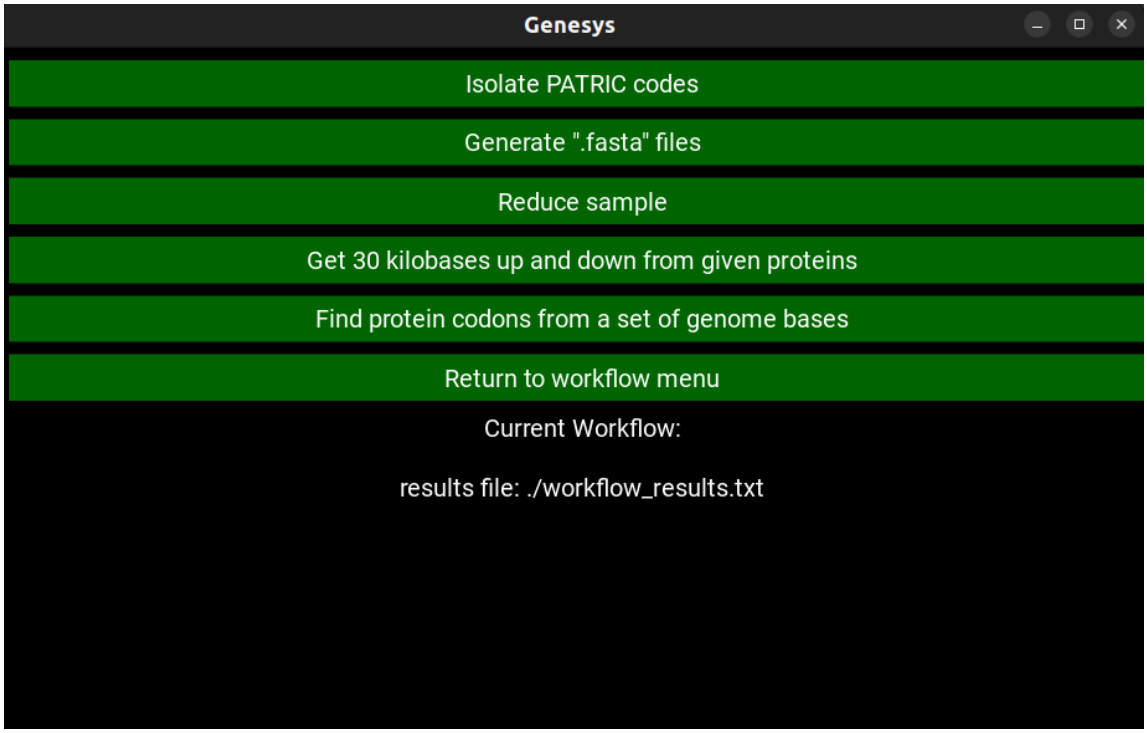


The screen contains a black scroll view underneath that shows information about the current workflow. For now, it only shows the selected results file for the workflow’s execution, as there are no defined tasks yet. Also, all the available actions to execute to a workflow are available through buttons. All the buttons will be explained later. Now it is time to dive into the “Add tasks to the workflow” option.

7.4. Add tasks menu.

Once that button is pressed, this screen will appear.

Image 7.4.1. Tasks selection menu.

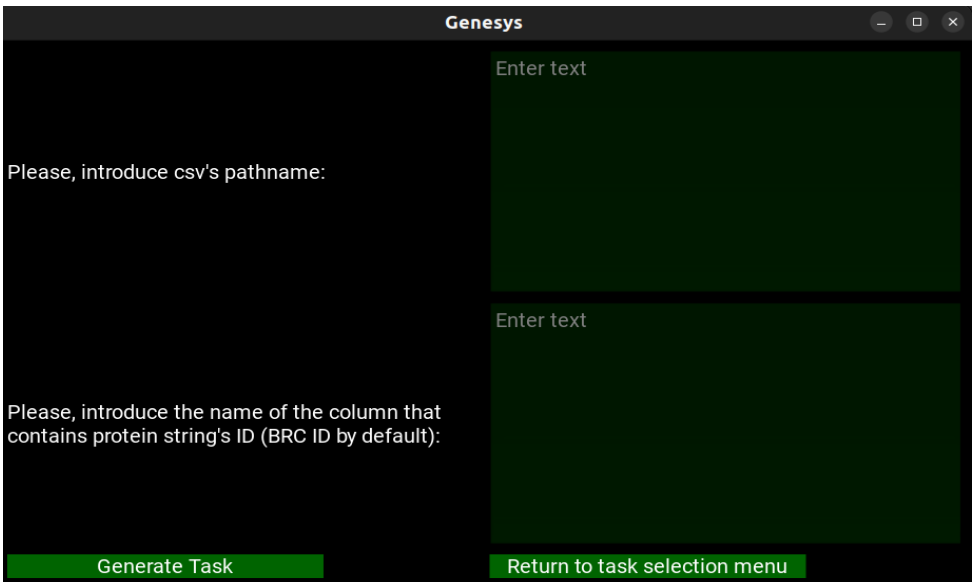


It also includes a scroll view of the current workflow underneath, as well as multiple buttons that correspond to the available tasks that can be added to the workflow. The tasks available to add to the workflow will change depending on the module that is being used. Right now, it includes all the tasks that compound the proteins neighboring issue initially requested by Dr. Martínez-Abarca. Any of the tasks can be selected. There is no duty in the order. However, in this tutorial the tasks will be added sequentially, so a full workflow is created and executed in order to provide the best possible overview of GeneSys functionality.

7.5. Isolate PATRIC codes task.

This screen appears when the first task is clicked. It asks for a csv file downloaded from PATRIC website and the name of the column to isolate from that file into a new csv file.

Image 7.5.1. Isolate column screen.



For this case, there is a thirty slatt domain containing proteins example file downloaded from PATRIC in the root folder of the application. We will employ that dataset as the entry data for this tutorial.

Image 7.5.2. Tutorial data folder.

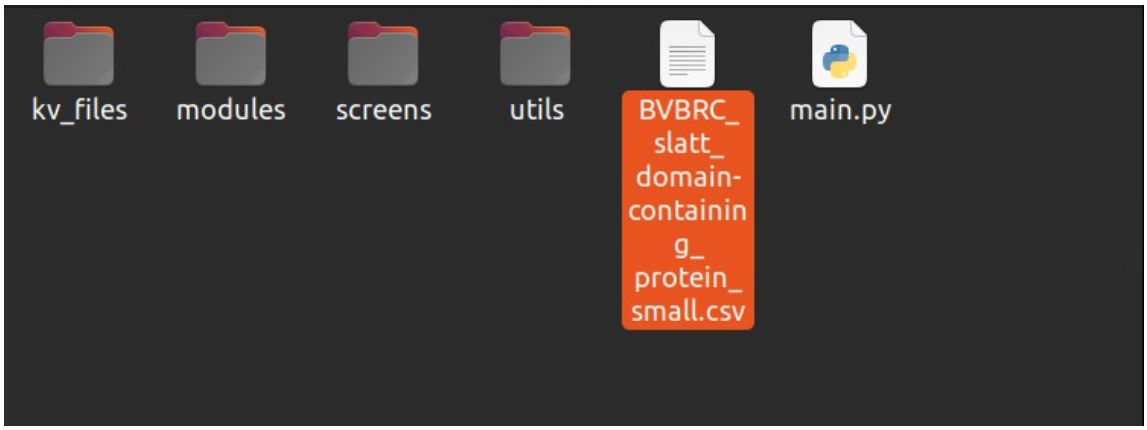
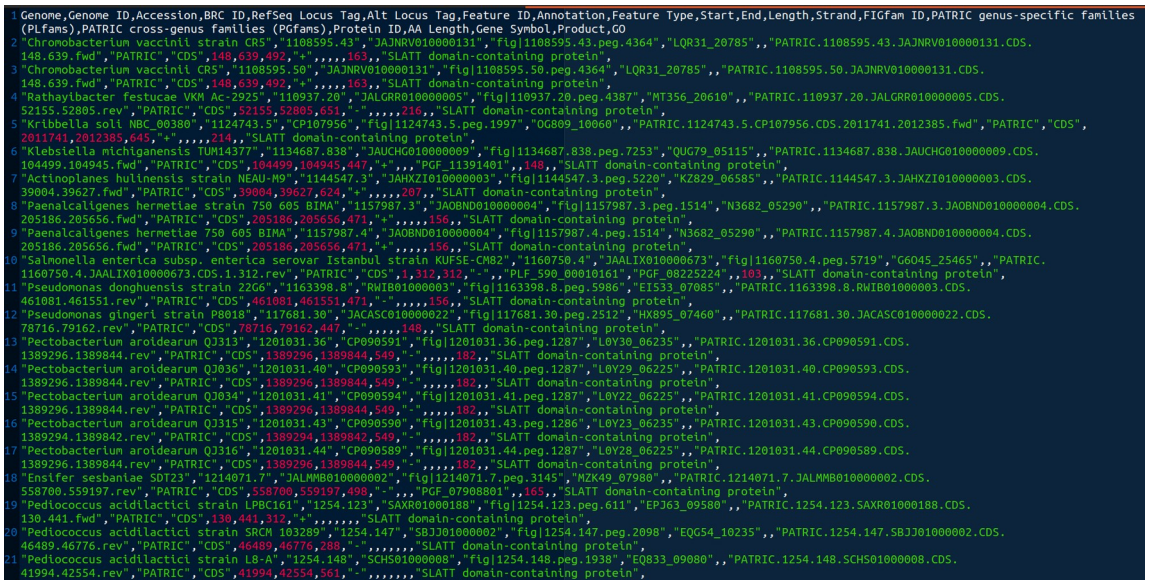
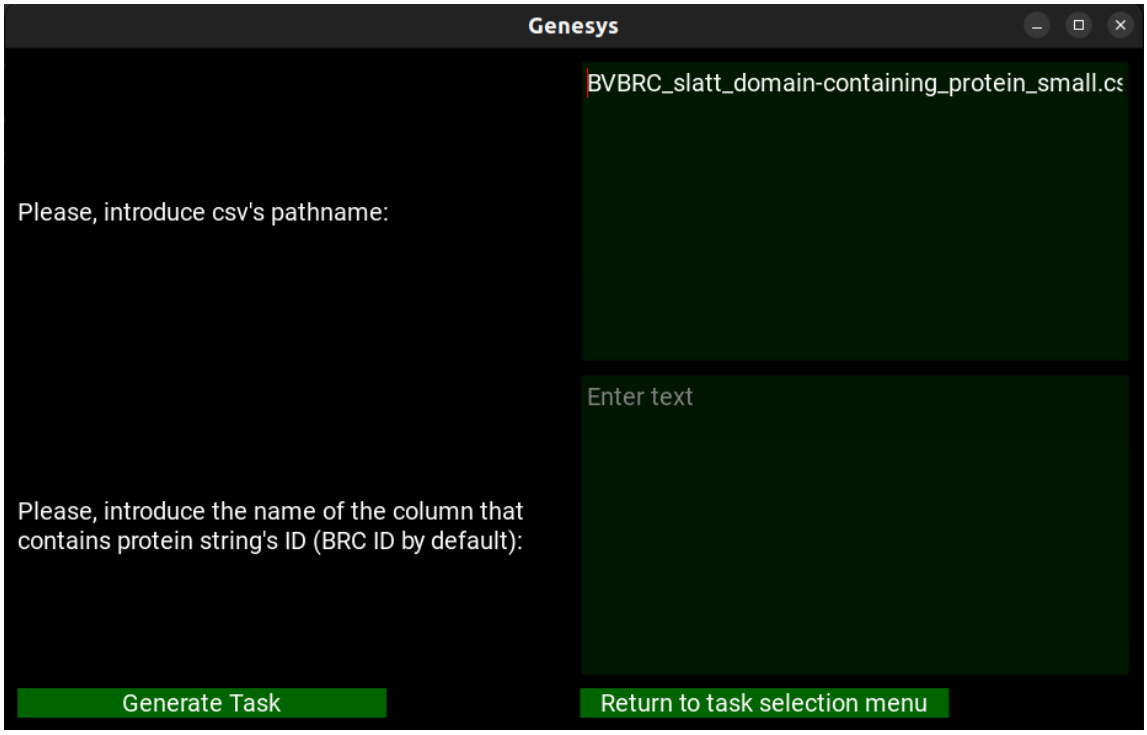


Image 7.5.3. Tutorial dataset.



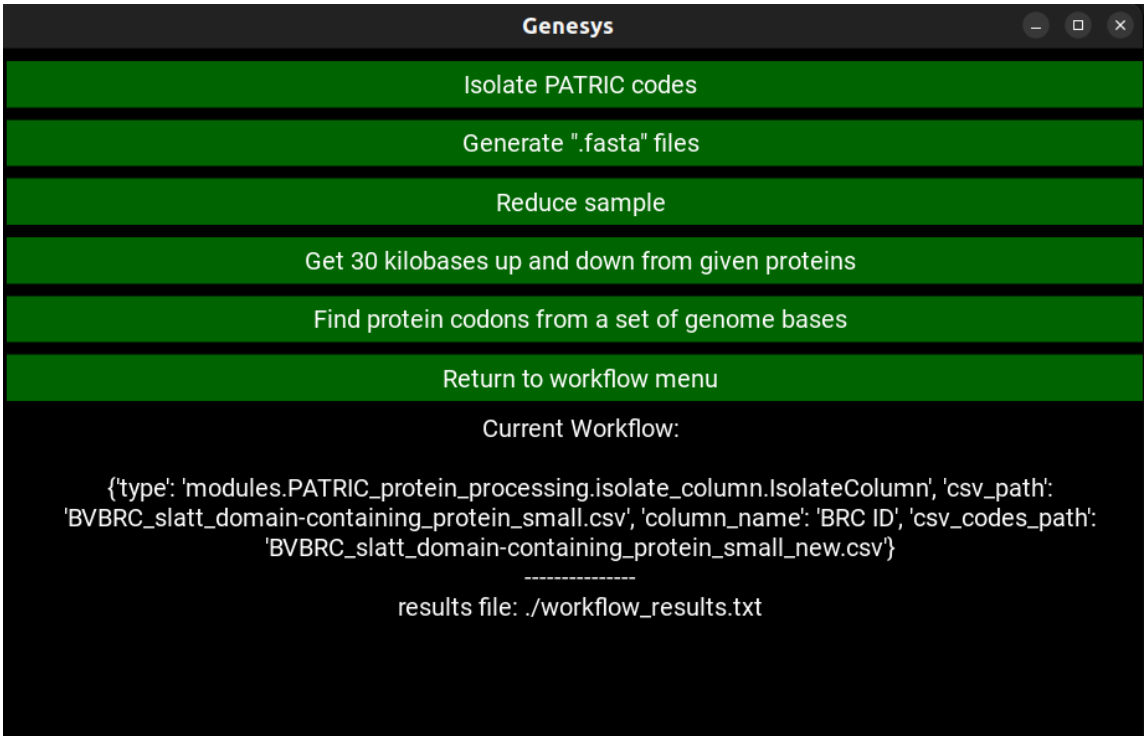
The task screen also states that if no column is given in its corresponding text box, the string “BRC ID” will be employed by default. As that is exactly the column that must be isolated from the dataset, that box will be leaved empty.

Image 7.5.4. Isolate column task filled.



The “Generate task” button will create an object of the task, add it to the workflow and automatically return to the task selection screen. On the other hand, the “Return to task selection menu” button will also move back to that screen but without adding the current task to the workflow. This is the new look of the task selection screen once the task has been generated.

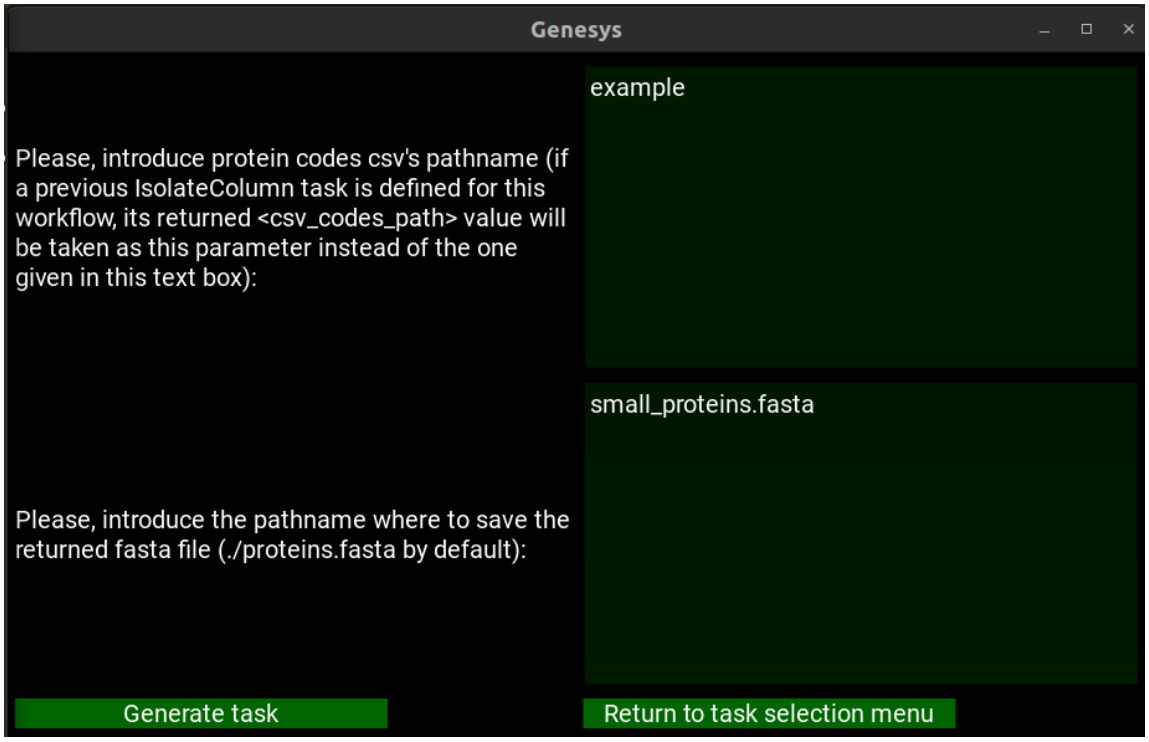
Image 7.5.5. Tasks menu with one task.



7.6. Generate “.fasta” files task.

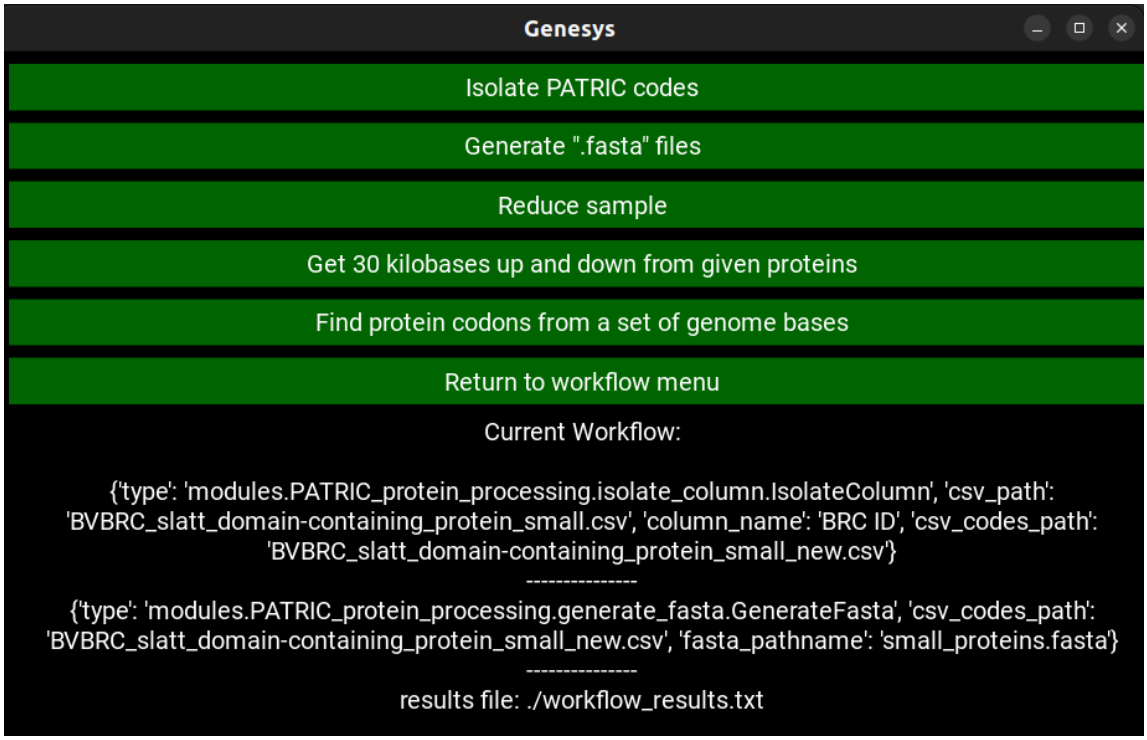
Moving on to the next screen, it asks for a csv file with one column from which take the entry protein IDs for the task, as well as a pathname to a fasta file that will contain their respective protein sequences. As this task will tend to be used right after an isolate column task, there is a special implementation that automatically fills the entry data csv pathname every time an isolate column task is previously defined, so that the returned csv file of the previous task will fill the entry csv pathname required by the generate fasta task, regardless of what users introduce in the text box.

Image 7.6.1. Generate fasta task filled.



In this case, the csv pathname entry has been filled with the word “example”, but looking at the task once it has been defined in the task selection menu, it can be seen that the actual pathname that has been taken for the entry data “csv_codes_path” actually is “BVBRC_slatt_domain-containing_protein_small_new.csv”, which corresponds to the parameter “csv_codes_path” specified for the previous task.

Image 7.6.2. Tasks menu with two tasks.



7.7. Reduce sample task.

It will automatically fill the first text input if a previous generate fasta task is defined. It also asks for the fasta file where to save the reduced sample of proteins and the similarity percentage to employ for determine wether two proteins are considered to belong to a same evolutive branch.

Image 7.7.1. Reduce sample task filled.

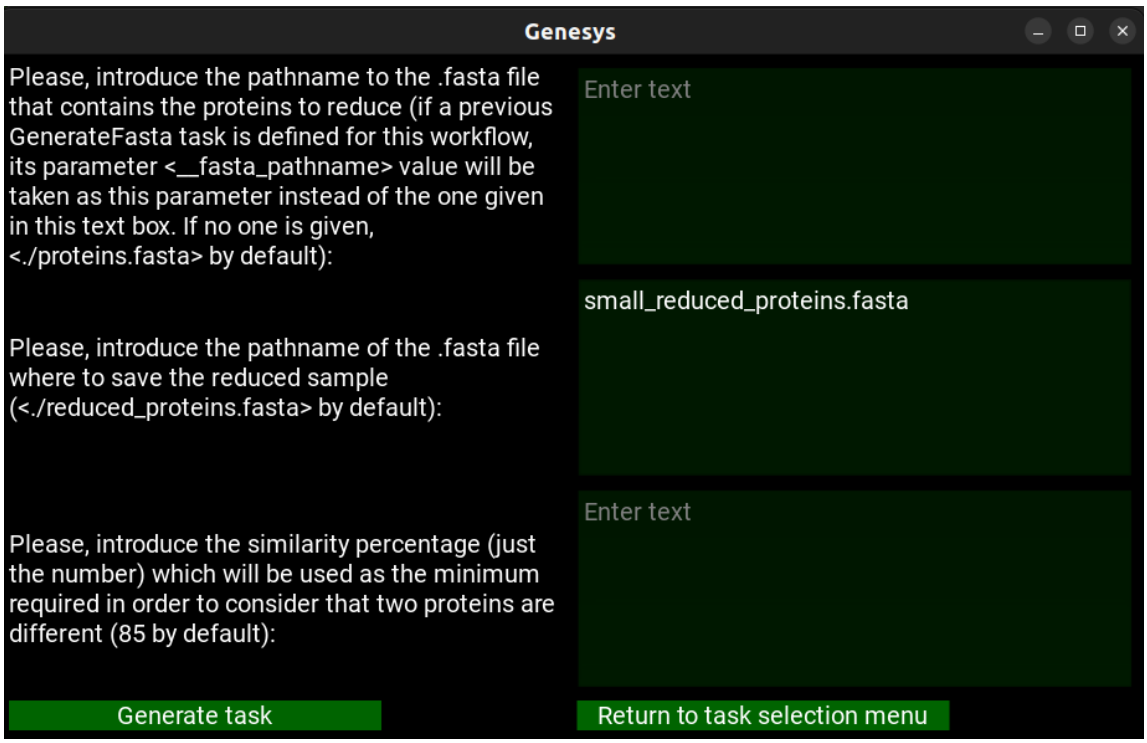
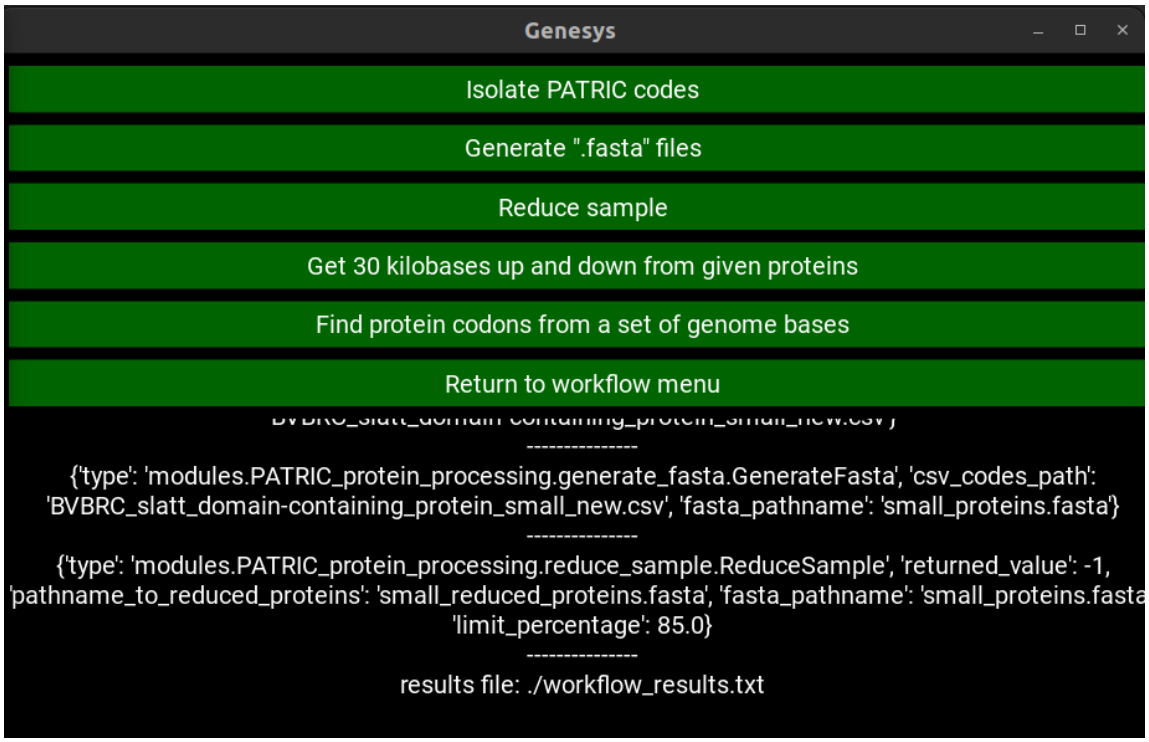


Image 7.7.2. Tasks menu with three tasks.



7.8. Get 30 kilobases up and down from given proteins task.

It will get the fasta file with the protein baits samples entry automatically if a previous reduce sample task is defined. The other text input corresponds to the fasta file where to save the nucleotides sequences.

Image 7.8.1. Get 30kb task filled.

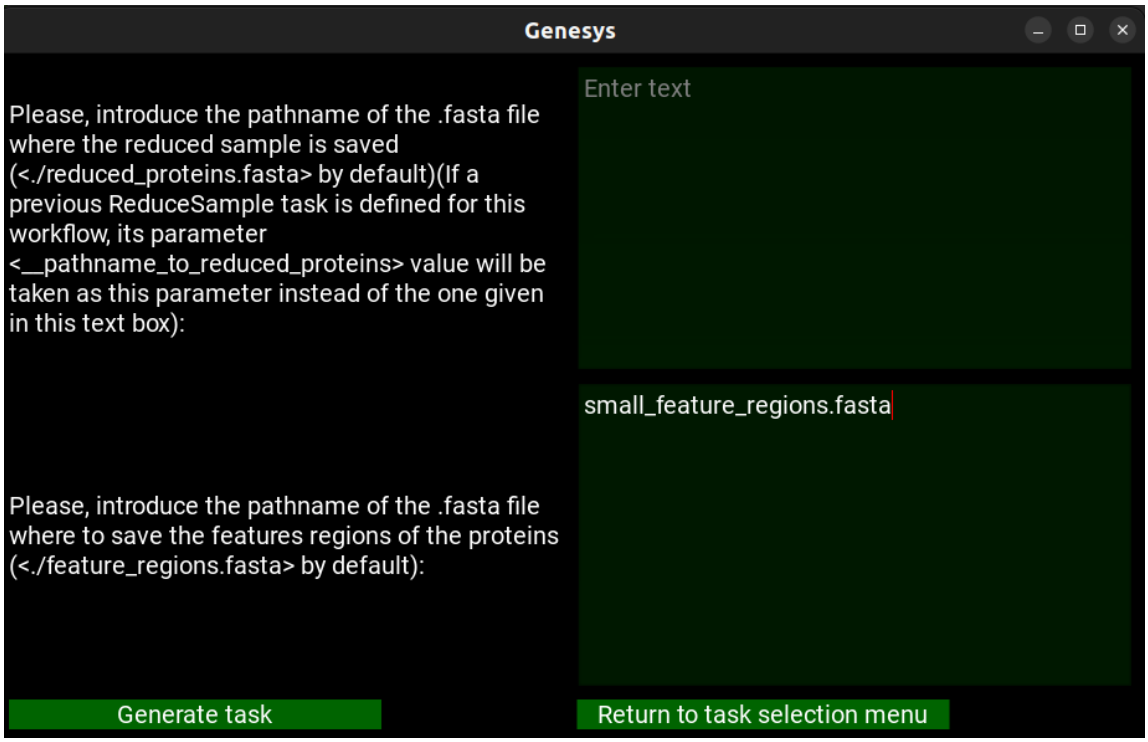
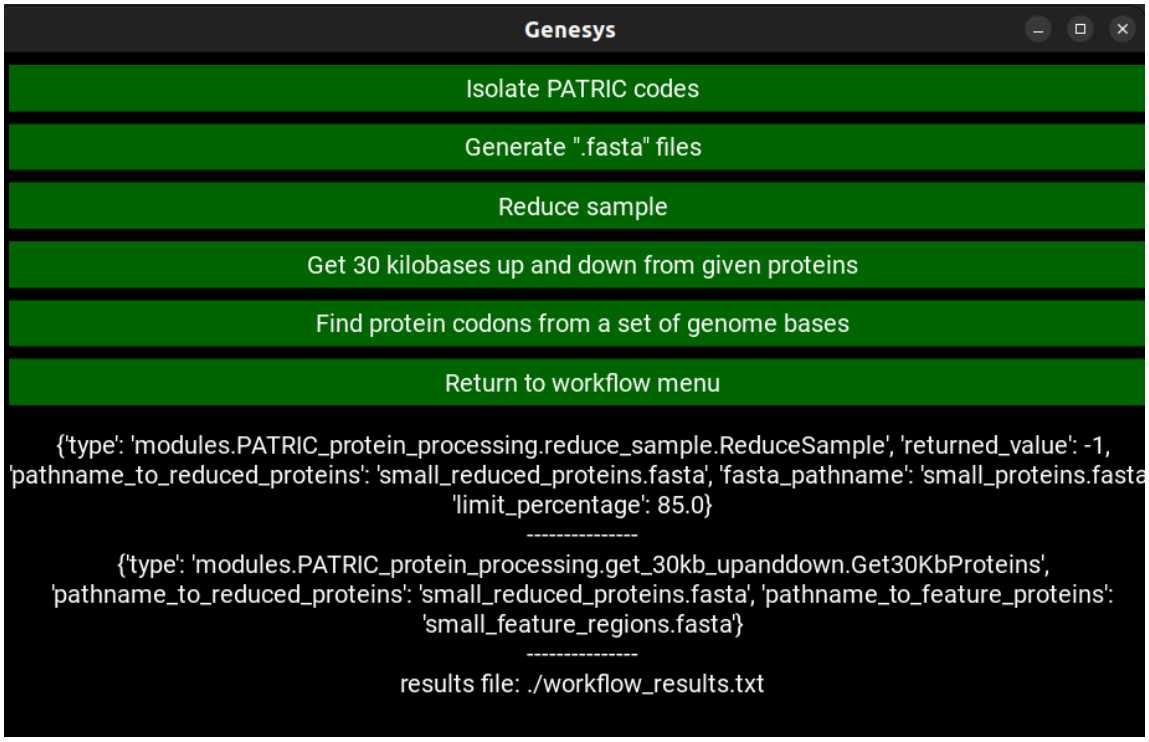


Image 7.8.2. Tasks menu with four tasks.



7.9. Find protein codons from a set of genomes bases task.

It will fill the first text input automatically if a previous get 30kb task is defined. Also asks for the excel file pathname where to save the final results of the preprocessing.

Image 7.9.1. Recognize codons task filled.

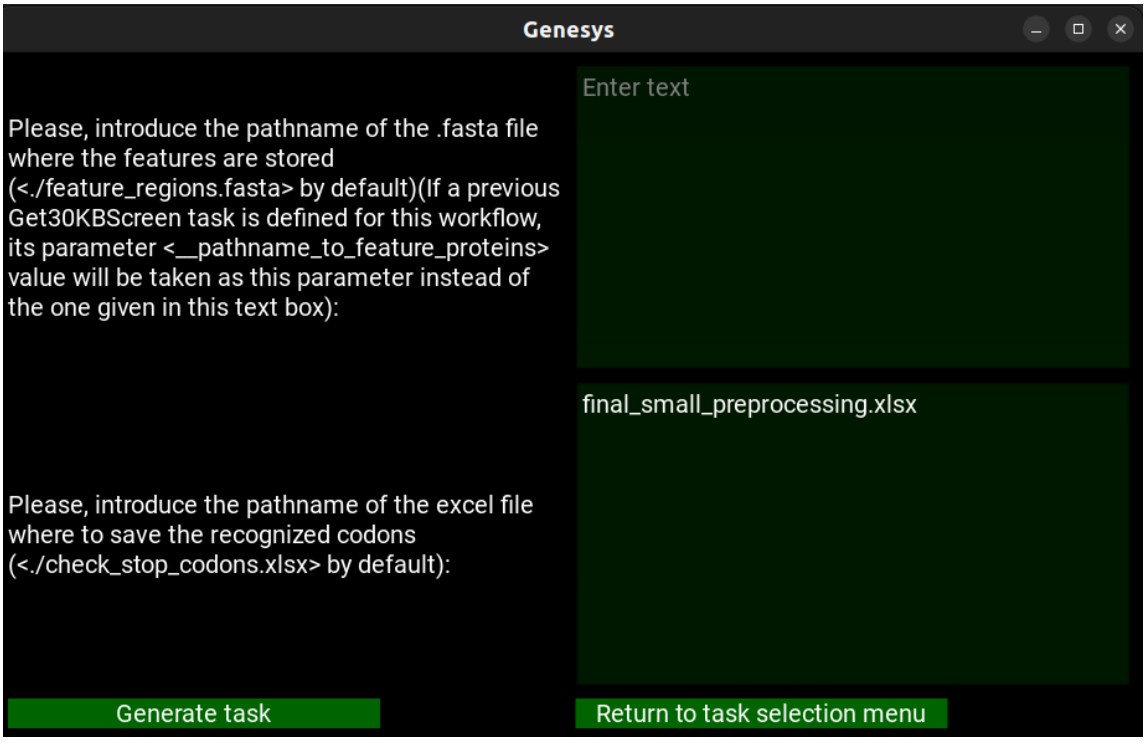
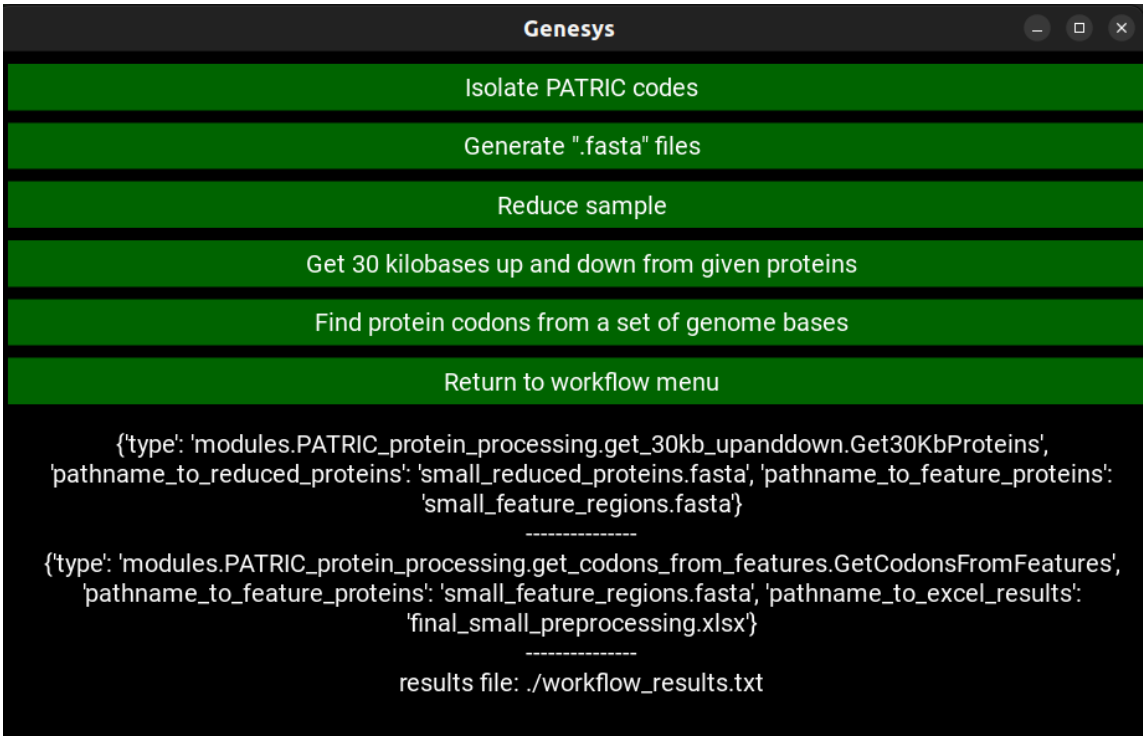


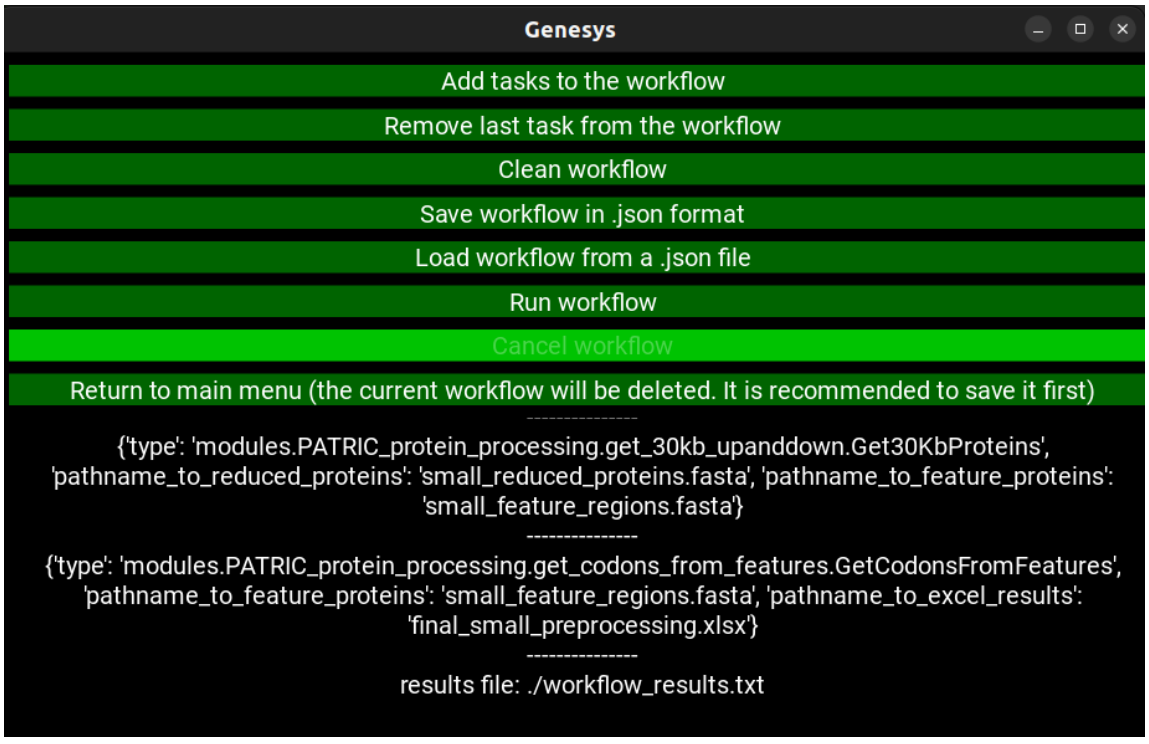
Image 7.9.2. Tasks menu with five tasks.



7.10. Moving back to the workflow manipulation menu.

Now that a full workflow has been created, the “Return to workflow menu” button can be clicked to return to the workflow manipulation menu, conserving the workflow that has been assembled.

Image 7.10.1. Workflow menu with five tasks.

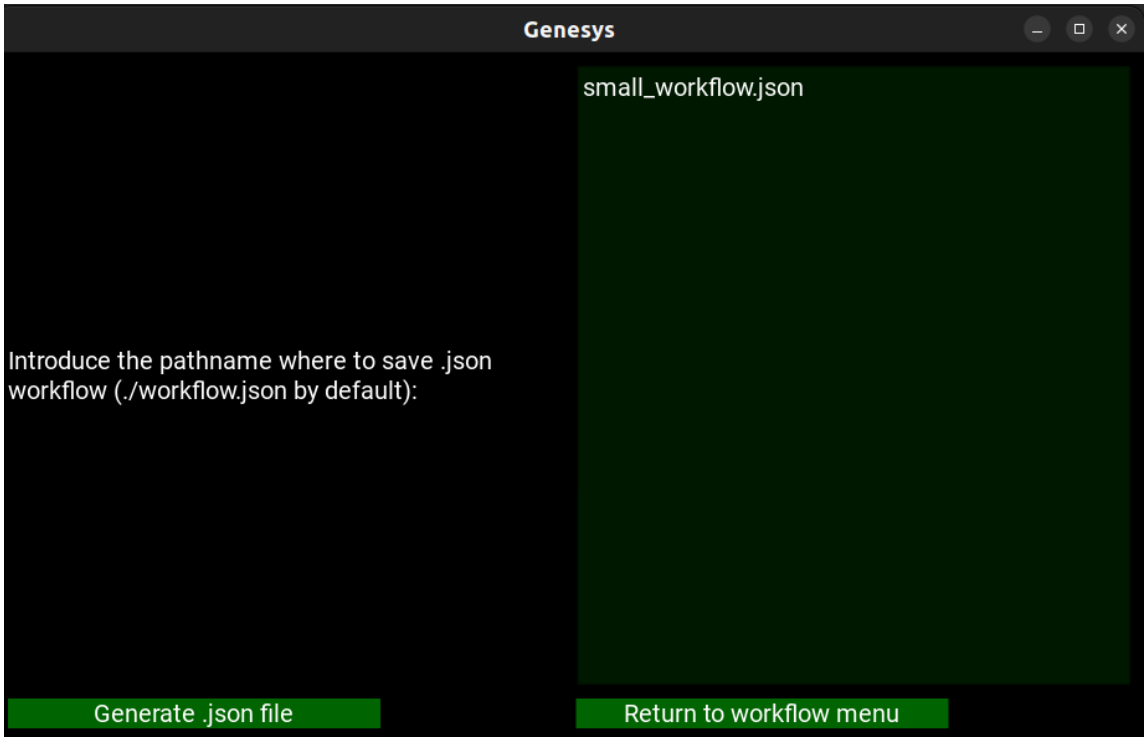


Now it is time to look at the functions that can be applied to the pipeline. Apart from adding new tasks to the workflow, GeneSys allows to remove the last task by clicking into the “Remove last task from the workflow”. If the “Clean workflow” button is clicked instead, it will eliminate all the tasks that compose the workflow. It is also possible to return to the main menu with its corresponding button, but doing so will eliminate the current workflow, too.

7.11. Save workflow in .json format.

This is the screen that appears if the button to save the workflow in json format is clicked. It asks the pathname where to save the json file.

Image 7.11.1. Save workflow in json format screen.



In the example, the name given to the json file that will store the workflow is “small_workflow.json”. If the button “Generate .json file” is clicked, then the current workflow will be saved in the specified path. If the button “Return to workflow menu” is selected instead, then the workflow will not be saved.

Image 7.11.2. json file workflow folder.

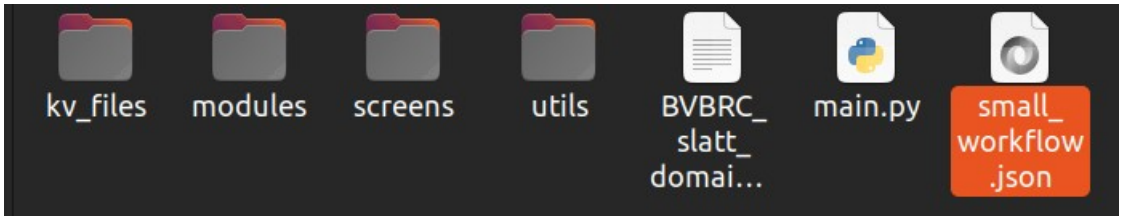
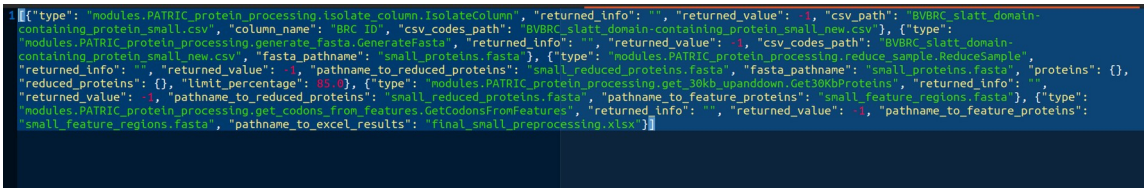


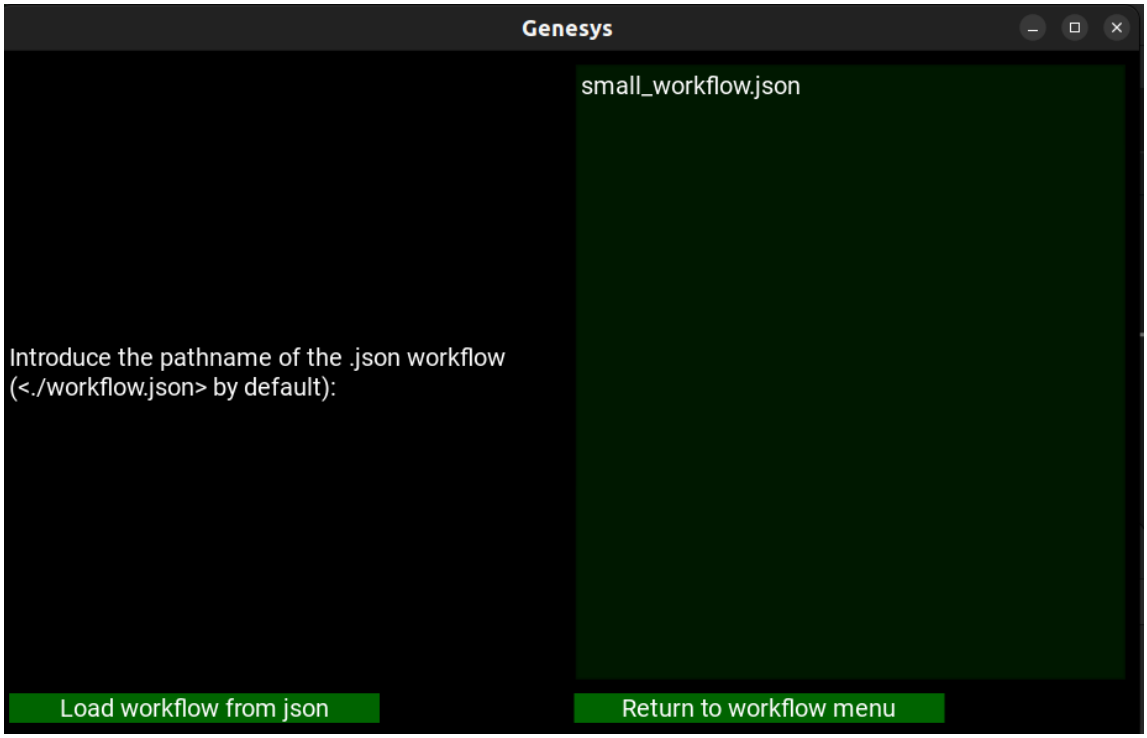
Image 7.11.3. json file workflow content.



7.12. Load workflow from a .json file.

A saved workflow can be loaded from a json file, too. That is the duty of the screen that appears when the “Save workflow in .json format” button is clicked in the workflow menu. It works the same way as the previous one, but instead of saving the workflow, checks if the given path exists and loads the content of the file in a new pipeline that overwrites any previous existing workflow.

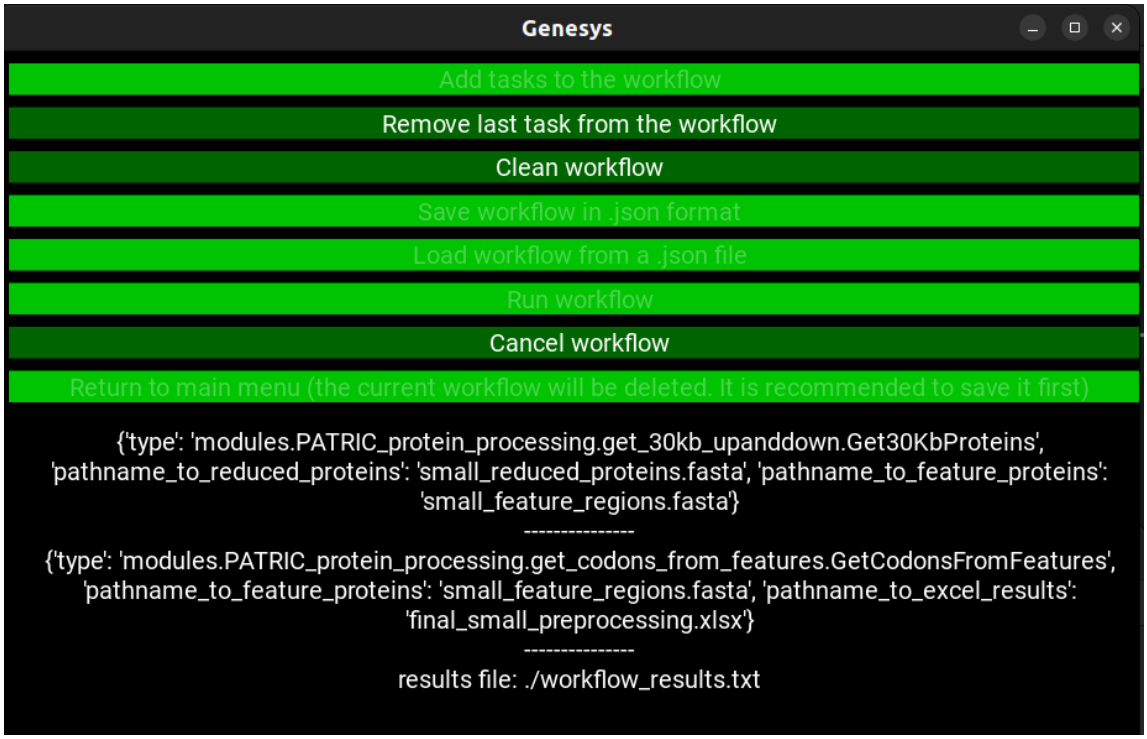
Image 7.12.1. Load workflow from json format screen.



7.13. Executing the workflow.

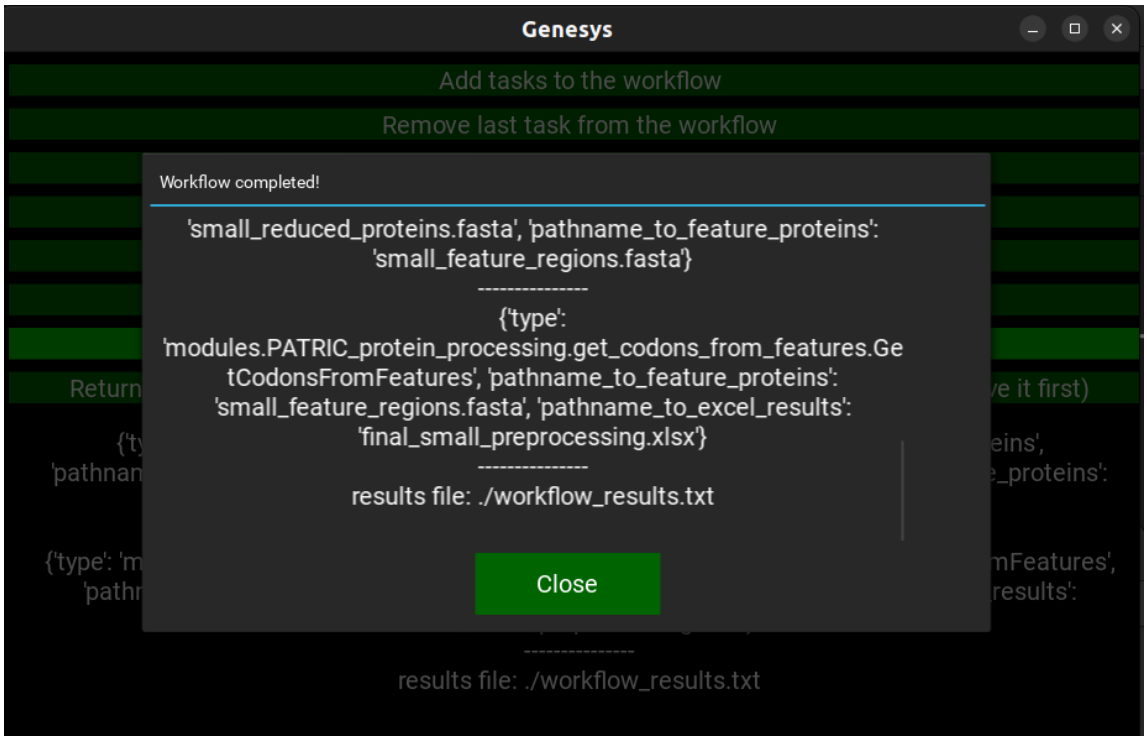
The workflow can be executed with its corresponding button. When the execution button is clicked, all the buttons that change the screen where users are working are automatically disabled, as well as the execution button itself. Only the buttons to remove the tasks contained in the workflow remain enabled. Also, the button that allows to cancel the execution becomes enabled.

Image 7.13.1. Executing the workflow screen.



When the execution ends, a pop up window emerges, showing context information about the results.

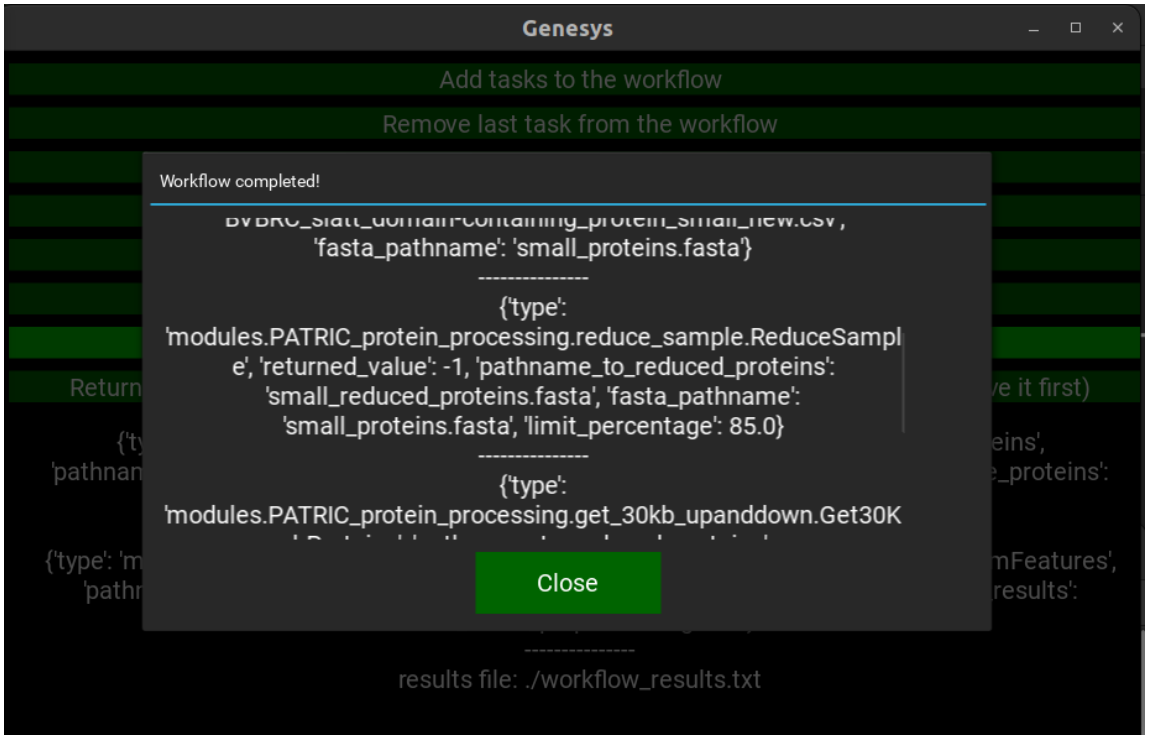
Image 7.13.2. Workflow completed pop up.



When the workflow is canceled, its execution is killed immediately. As a result, the same pop up window emerges. The difference is that in this case the context information about the

workflow's execution might have changed. Also, there are no guaranties that the results provided are reliable (notice that the returned_value of some tasks remains at -1, indicating that those tasks have not been executed).

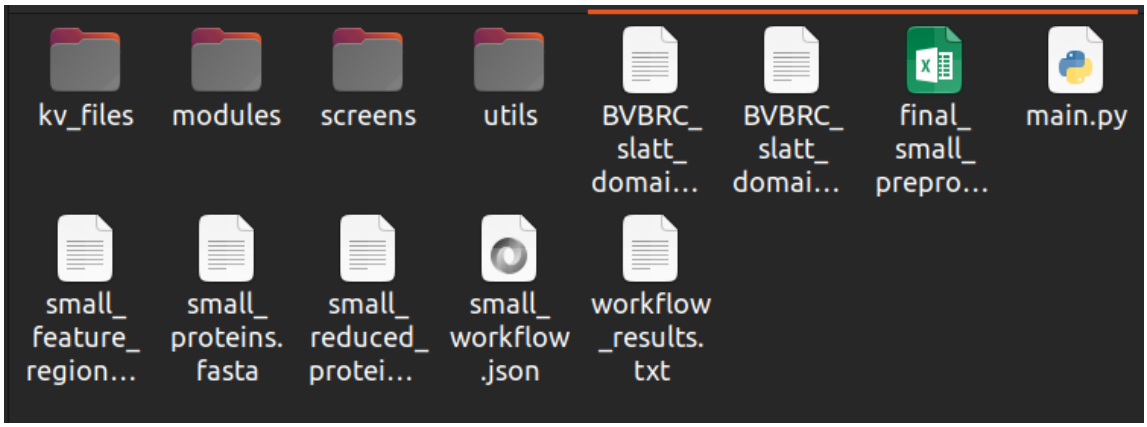
Image 7.13.3. Workflow canceled pop up.



7.14. Checking results.

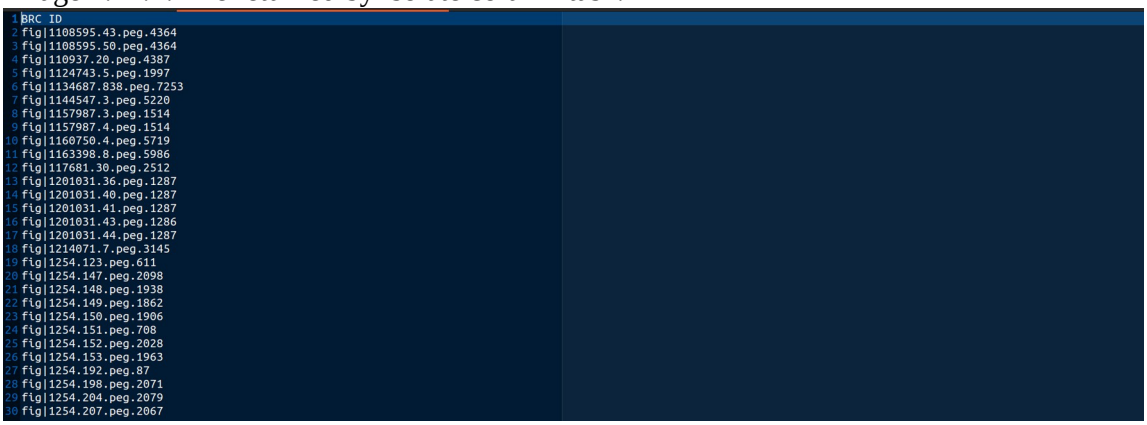
Once the execution ends, it can be seen that the root folder contains some new files. For each executed task, there is a returned file. Five tasks have been executed, so five files have been created, plus the workflow_results.txt file with an abstract of the execution process, equals to six new files.

Image 7.14.1. Execution resulting files.



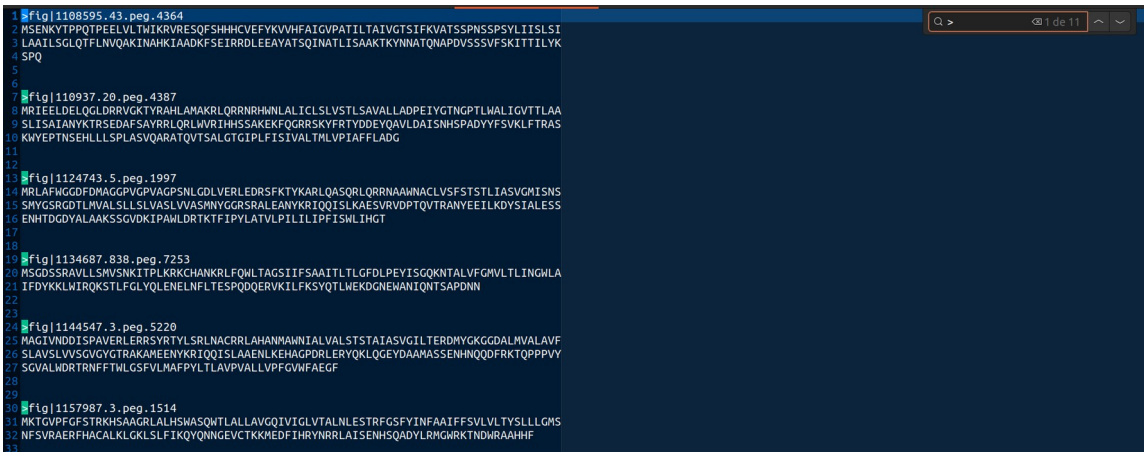
This is the content of the file returned by the isolate column task, corresponding to thirty slatt domain containing protein BRC IDs.

Image 7.14.2. File returned by isolate column task.



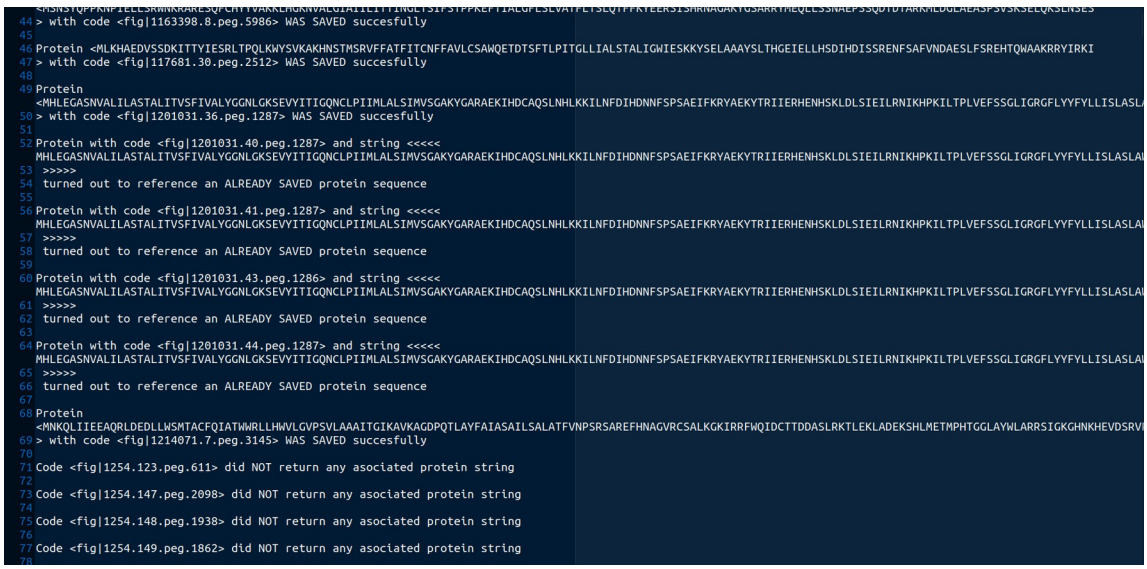
This is the content of the file returned by the generate fasta task, with only eleven protein sequences isolated.

Image 7.14.3. File returned by generate fasta task.



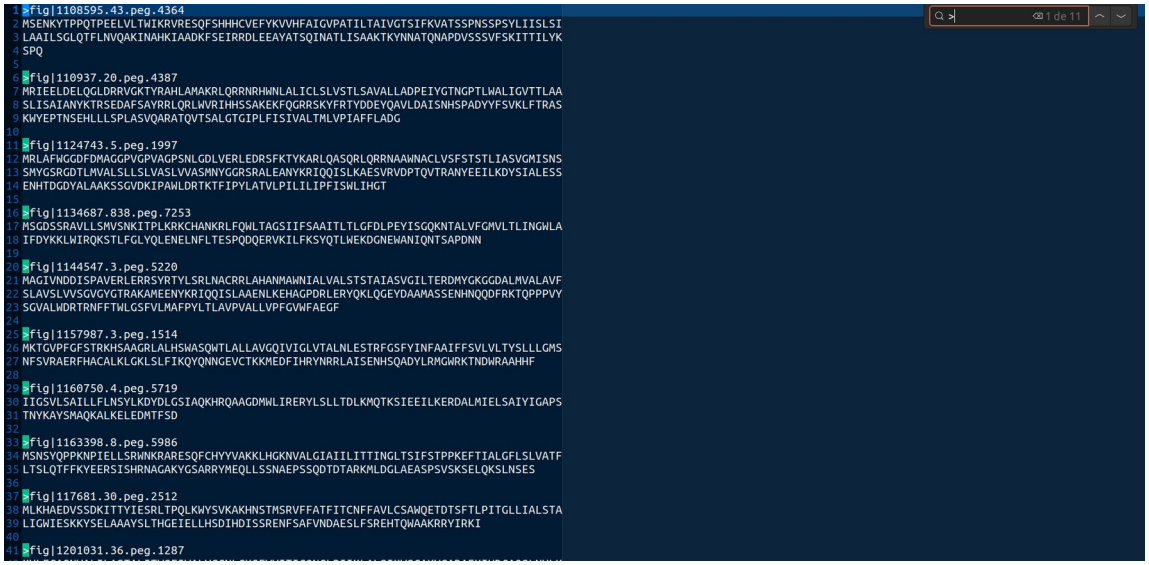
At first sight, the fact that there are only eleven saved sequences out of an initial set of thirty looks strange. In these cases, it is always a good idea to check the workflow_results.txt file content. By doing so, it becomes clear that the reason why there were only eleven out of thirty samples isolated is because some of them did not have any paired amino acid sequence in the PATRIC database, or such sequence corresponded to a repeated one which was already saved. Here are shown some of the returned lines in the workflow results file.

Image 7.14.4. Generate fasta task results in the workflow results file.



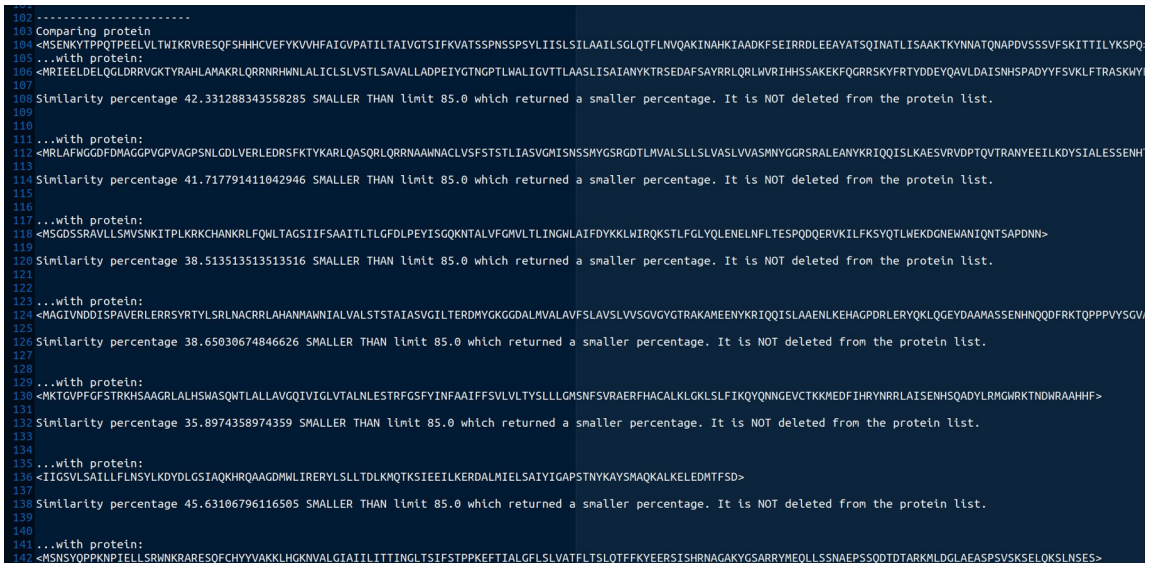
This is the content of the file returned by the reduce sample task.

Image 7.14.5. File returned by reduce sample task.



Given an 85 percentage similarity threshold, the sample has not been reduced. It keeps containing the previous eleven proteins. Why? If the workflow results file is checked again, it will be seen that no comparison between two proteins ever returned a similarity percentage above 85, which explains why any samples have been removed. Here is a screenshot of the results file that shows some of the obtained percentages.

Image 7.14.6. Reduce sample task results in the workflow results file.

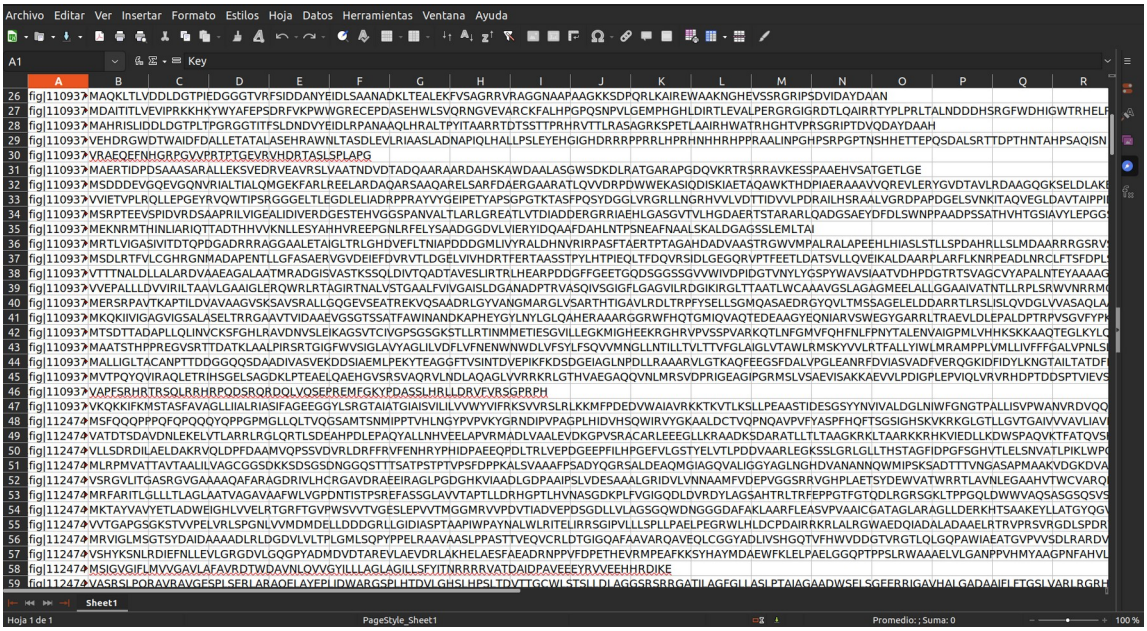


This is the content of the file returned by the get30kb task. Now, instead of containing amino acid strings, each of the eleven protein codes is employed as the bait to identify a very long nucleotide string where the subsequences that might correspond to valid neighbor proteins of the baits are marked in capital letters.

Image 7.14.7. File returned by get30kb task.

Finally, this is a screenshot of some of the content of the excel file returned by the recognize codons task. It has two columns corresponding to the ID of a bait and a protein that surrounds that bait respectively. All the rows that start by the same protein code indicate proteins that surround a same bait. This file can be employed by researchers to progress with their investigations.

Image 7.14.8. File returned by recognize codons task.



8. CONCLUSIONS AND FUTURE WORK

8.1. Conclusions.

Down below are exposed the objectives that the student wanted to accomplish with GeneSys, previously defined in section two, with their corresponding evaluations after the development has ended.

- **To develop a bug free logic that works exactly as the user needs it to work**. Until the moment of writing this memory, there are no uncorrected bugs. The proposed architecture and the modular organization of the code has helped to focalize errors and properly solve them. Also, the logic is well structured, making easy to prevent future bugs.
- **To provide an intuitive yet attractive friendly-user interface**. The user interface works uniquely with buttons and text boxes. Even though it requires users to domain some informatics basic concepts (such as pathnames), at no time GeneSys forces researchers to dive into more complex matters such as a command line tools. As a result, anyone with a minimum computer knowledge is able to learn how to use GeneSys.
- **To properly recognize which tasks must be automated by the application, so that the user receives a window to modify certain parts of the proteins preprocessing in order to adapt their experiments freely while not losing the automatization benefits that GeneSys provides**. This objective is more ambiguous, mainly because letting the user to choose freely how to use the preprocessing tasks tends to reduce the automatization that can be applied in the process, and vice versa. However, it can be stated that functionalities such as letting users to save and load their workflows with many tasks as they want (not forcing them to always assemble a workflow with the full preprocessing process), along with others such as automatically filling the entry data of a task if the previous one corresponds to the one that comes right before, overall, implements mechanisms to both giving freedom to researchers and helping them to fill repetitive tasks.
- **To provide a software framework that unifies all the preprocessing tasks in an unique execution context. In other words, we do not want users to open any other application than GeneSys to achieve their goals**. This objective has been achieved because GeneSys works entirely by its own without requiring users to open other applications in order to assemble their workflows.
- **To make an application that does not freeze or crash when it is executing a long task**. By launching the execution of workflows in separated threads, GeneSys does not freezes while running its workflows. Also, the disabling of buttons during the execution of a workflow prevents users from rerunning them again, serving as an effective way to prevent crashes.
- **To properly inform users about what is happening in the pipeline, so that they can study the results returned at all the steps of the process and draw their own conclusions for their research**. This has been achieved with the incorporation of a txt file with all the workflow's relevant execution information.

8.2. Suggested future improvements.

GeneSys is a modular application with a lot of potential. It is not only what it is now, but what it can be in the future if it continues with its development.

One of the most important matters in the current state of data science concerns the analysis of huge volume of data within AI models that are capable of extract useful information by observing data sets and applying machine learning algorithms to them. It is not necessary to go far to find an example of this: Dr. Martínez-Abarca himself needs to apply a clustering algorithm to the preprocessed samples returned by GeneSys. What if a new module oriented to assemble machine learning workflows is developed and added to the tool? It would be clearly useful for researchers. Such implementation probably would require to declare a general purpose AI model class that would inherit from the basic Task class, from which would later inherit all the AI models that would be required in future modules.

Long story short, it would be a great add-on to GeneSys to become more than a preprocessing tool and turn it into a full data science desktop application oriented to manipulate bioinformatic datasets.

And once that upgrade is implement, the rest would consist on implementing new modules whenever a new issue needs to be solved.

9. BIBLIOGRAPHY

1. Luisa B. Huber, Dr. Karin Betz, Dr. Andreas Marx. Reverse Transcriptases: From Discovery and Applications to Xenobiology (2022)

   https://chemistry-europe.onlinelibrary.wiley.com/doi/10.1002/cbic.202200521

2. Balbir B. Singh, Michael P. Ward, Navneet K. Dhand. Geodemography, environment and societal characteristics drive the global diversity of emerging, zoonotic and human pathogens (2021)

   https://onlinelibrary.wiley.com/doi/10.1111/tbed.14072

3. Marco Marani, Gabriel G. Katul, William K. Pan, Anthony J. Parolari.Intensity and frequency of extreme novel epidemics (2021)

   https://www.pnas.org/doi/10.1073/pnas.2105482118

4. Diego A. Forero, Vaibhav Chand. Methods in molecular biology and genetics: looking to the future (2023)

   https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9980850/

5. The Editors of Encyclopaedia Britannica. DNA (2024)

   https://www.britannica.com/science/DNA

6. Yao Wan, Kunal Chatterjee. RNA (2024)

   https://www.britannica.com/science/RNA

7. Melissa Petruzzello. What is the difference between DNA and RNA (2024)

   https://www.britannica.com/story/what-is-the-difference-between-dna-and-rna

8. Felix Haurowitz, Daniel E. Koshland. Protein (2024)

   https://www.britannica.com/science/protein

9. CK-12 Foundation. Introductory biology: Translation of RNA to protein.

   https://bio.libretexts.org/Bookshelves/Introductory_and_General_Biology/Introductory_Biology_(CK-12)/04%3A_Molecular_Biology/4.07%3A_Translation_of_RNA_to_Protein

10. Zhong Wang, Mark Gerstein, Michael Snyder. RNA-Seq: a revolutionary tool for transcriptomics (2009)

    https://pubmed.ncbi.nlm.nih.gov/19015660/

11. Fábio C. P. Navarro, Hussein Mohsen, Chengfei Yan, Shantao Li. Genomics and data science: An application within an umbrella (2019)

    https://www.researchgate.net/publication/333468103_Genomics_and_data_science_An_application_within_an_umbrella

12. Mario Rodríguez Mestre. Analysis of novel and unexplored groups of prokaryotic Reverse Transcriptases (2020). Available at University's of Granada repository.

13. Talent.com. Tax calculator in Andalusia (2024)

https://es.talent.com/tax-calculator?salary=1500&from=month&region=Andaluc%C3%ADa

14. Microsoft. What is the average lifespan of a laptop? (2024)

https://www.microsoft.com/en-us/surface/do-more-with-surface/what-is-the-average-lifespan-of-a-laptop

15. Tim Rogers. How Long Does a Computer Monitor Last: Lifespan Expectations Explained (2023)

https://thecomputerbasics.com/how-long-does-a-computer-monitor-last/

16. Roy Chng. How Long Can A Smartphone Last? (With 6 Real Examples) (2020)

https://valorvortech.com/how-long-can-a-smartphone-last/

17. precioelaluz.com. Precio medio de la electricidad por meses: 2024

https://preciodelaluz.com/precio-por-meses-2024

18. Plokiko. Qué fibra es mejor en España en relación calidad precio (2024)

https://www.xatakamovil.com/comparativa-de-tarifas/que-fibra-mejor-espana-relacion-calidad-precio

19. YordanoBOG. Genesys GitHub repository

https://github.com/YordanoBOG/GeneSys

20. Bacterial and Viral Bioinformatics Resource Center. Main page.

https://www.bv-brc.org/

21. Mario Rodríguez Mestre, Alejandro González-Delgado, Luis I Gutiérrez-Rus, Francisco Martínez-Abarca, Nicolás Toro. Systematic prediction of genes functionally associated with bacterial retrons and classification of the encoded tripartite systems (2020)

https://academic.oup.com/nar/article/48/22/12632/6020195?login=false

22. National Center for Biotechnology Information. Main page.

https://www.ncbi.nlm.nih.gov/

23. Protein Data Bank. Main page.

https://www.rcsb.org/docs/tools/pairwise-structure-alignment

24. CD Genomics. Protein databases.

https://bioinfo.cd-genomics.com/protein-databases.html

25. European Nucleotide Archive. Main page.

https://www.ebi.ac.uk/ena/browser/home

26. MGnify. Main page.

https://www.ebi.ac.uk/metagenomics

27. BV BRC Command Line Program Reference.
https://www.bv-brc.org/docs/cli_tutorial/command_list/index.html
28. Kivy documentation. Main page.
https://kivy.org/doc/stable/index.html