DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Hierarchical Graph Clustering

Julian Zimmermann

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Hierarchical Graph Clustering

Hierarchiches Gruppieren von Graphen

Author:	Julian Zimmermann
Supervisor:	Daniel Rückert
Advisor:	David Blumenthal
Submission Date:	15.09.2021

Acknowledgments

First I would like to thank my supervisor, Professor Daniel Rückert. It was for him that my shifting interest from games engineering over general computer science to bioinformatics could result in this rather untypical topic for a Bachelor's Thesis in Informatics: Games Engineering. As part of the informatics faculty at the TUM, he agreed to supervise a bioinformatics-related topic.

I also would like to thank my advisor David Blumenthal for his continuous support in these four months, always being available when problems arose. He was also the one to give me an introduction into the software this topic is closely related to, which is for a large part written by him too, as well as the idea for the topic itself.

Finally, I would like to thank my colleague Julian Geheeb for reading over the first draft, suggesting corrections and giving me the first impression of someone who's new to the topic. His help has made the polishing process a lot easier for me.

Abstract

Especially in bioinformatics, one is often confronted with large sets of data, creating a constant need for easily accessible and well performing data mining methods. One of the most common ways to analyze datasets is hierarchical clustering and arguably the most generic data structures in computer science are network graphs. While there are already various public libraries providing fast clustering algorithms, they mostly take precalculated distances as input. This paper introduces a Python interface that not only enables performing hierarchical clustering on NetworkX graphs, but also includes the precomputation of their distances. This allows a set of graphs to be the input rather than a distance matrix. Although the most runtime demanding parts are outsourced to C++, the limitations of Python as a programming language in terms of performance will still show at one point.

Contents

1. Introduction

Hierarchical Clustering describes the approach of categorizing a set of data points into clusters based on their distances to the rest of the set. It is a powerful method for discovering patterns and other useful information in large sets of data. The method consists of two parts:

- Determining the pairwise distances between the data points
- Generating the clusters based on those distances

There are already various public Python libraries providing fast clustering algorithms like SciKit-Learn or SciPy, but they mostly take a precalculated distance matrix as input, only covering the second part [26].
While the data can theoretically be anything to which a distance measure can be defined, we want to focus on the hierarchical clustering of network graphs. Them being arguably the most generic data structure in computer science, hierarchical graph clustering can be helpful in a lot of environments, let it be epidemic analysis [5], RNA sequencing [32] or desease treatment. Specifying the type of data also allows for a choice of the distance measure to be made, which is at that point the only step still needed to be able to include both parts in one software.

This paper introduces HGC, a Python interface that supports a full graph clustering by not only providing algorithms to generate it, but also including the computation of graph distances, allowing a set of graphs to be the input rather than a distance matrix. Additionally, it includes support for data in a certain CSV format [19] originating from bioinformatics. All about this in section 4.2

To cover the first part in a performant and widely applicable way, HGC works with the C++ library GEDLib [6] to determine the so-called graph edit distances (GED) [3]. GED is an attractive distance measure due to its flexibility, intuitiveness and expressivity. Exactly computing GED is NP-hard, but various heuristics have been proposed over the past years. [3]
In computer science, a graph consists of a set of vertices and a set of edges between them, which on our case are labeled. The graph edit distance between two labeled

Graphs G and H is defined as the minimum cost of an edit path between G and H. Such an edit path is a sequence of graphs, starting at G and ending at H', a graph isomorphic to H, such that each graph on the path can be obtained from its predecessor by applying one of the following operations:

- adding an isolated node
- deleting an isolated node
- relabeling a node
- adding an edge
- deleting an edge
- relabeling an edge

Each operation is mapped onto a non-negative edit cost, usually depending on the label of the node or edge to insert, delete or relabel. The sum of those edit costs defines the cost of the edit path. [3]

When it comes to clustering methods, there are two main approaches: bottom up clustering, also called agglomerative nesting (AGNES) and top-down clustering, also called divisive analysis (DIANA). Divisive analysis is a rather unattractive method, as it has long runtimes for larger datasets [14]. Therefore, HGC uses SciPy [27] to generate a bottom up clustering, as it supports the highest number of algorithms at the moment. Furthermore, HGC works with one of the most common network graph analysis packages, NetworkX [22], by including the conversion between NetworkX graphs and the graph format used by GEDLib, as well as making use of NetworkX's feature of saving and loading graphs from and to GML [9].
Since GEDLib is a C++ library, pybind11 [25] is used as an interface between Python and C++.
Regarding the design of HGC, there are always two questions in mind.
The first one is "How can we make it as performant as possible?". GEDLib provides a wide choice of heuristics for the GED, which all shift the trade-off between performance and accuracy in their own way. The library is also written in C++, a top tier programming language in terms of performance [24]. Computing the distance matrix for large datasets still takes a considerable amount of time though, highlighting the necessity for a C++ implementation of this part.
The second one is "How can we make it as usable as possible?". After all, the goal is to provide a library for programmers to embed into their program as well as a tool for users to analyze a given graph dataset. As much as one thinks of C++ as a top

tier programming language in terms of performance, one has to think of Python as a top tier language in terms of usability. It is one of the most popular and widely used languages, being heavily characterized by its simple and intuitive coding style [15].
To combine those two requirements, the tool is split into two parts, respectively taking care of the two parts of hierarchical clustering. Figure 1.1 illustrates the basic structure of the software.

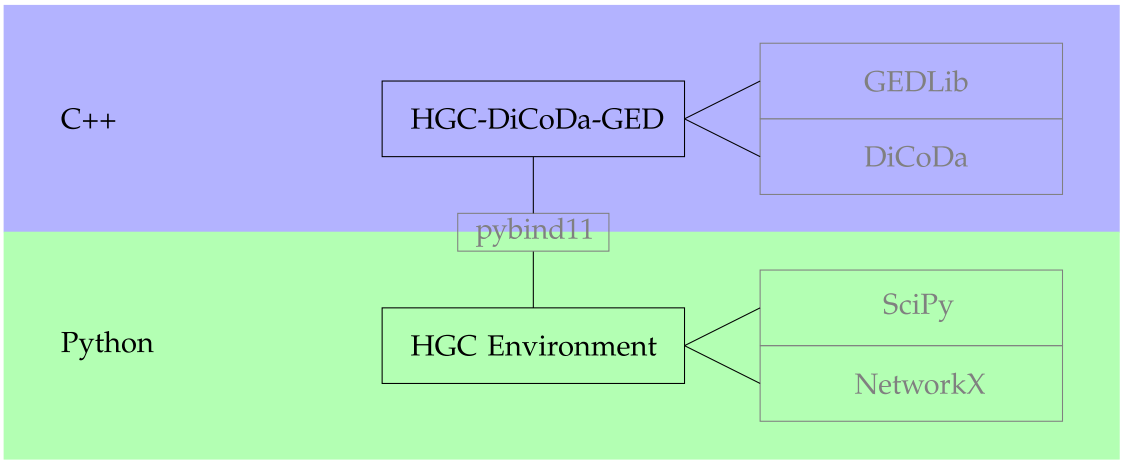


Figure 1.1.: Basic structure of HGC

The main Python program, here labeled HGC Environment, is responsible for the clustering and the NetworkX graph conversion.
The main C++ program, HGC-DiCoDa-GED, is responsible for computing the distance matrix of the graphs via GEDLib and uses another library, DiCoDa [1], concerned for working with the CSV data mentioned earlier. Both parts communicate with each other using pybind11.
This back and forth communication of Python and C++ presents some challenges which we will look at more precisely in subsection 4.5.2.

The remainder of this paper is organized as follows: In chapter 2, related work is discussed. In chapter 3, the theory and concepts behind hierachical clustering and GED are introduced. In chapter 4, the main part, the different components of the software are particularly explained, reasoned and analyzed with regard to established requirements. Also, remaining problems are identified. Finally, chapter 5 draws a conclusion and chapter 6 suggests improvements and further work.

2. Related Work

The central references for everything regarding GED are the library GEDLib [6] and its documentation and theory papers by David B. Blumenthal, Nicolas Boria, Johann Gamper, Sébastien Bougleux & Luc Brun [4], [3]. It provides a wide variety of different heuristics, various GXL [12] datasets, and multiple edit cost functions for the different dataset types. We want to specifically point out Sections 5.2.3, 5.2.4 and 7.2.5 of [3], describing the three heuristics supported by HGC, as well as Section 5.1 which lays the groundwork for understanding their general structure. They can be viewed as the default algorithm to yield results fast (BRANCH), the main algorithm to yield the tightest results (IPFP) and an option to speed up the process even more in some cases (BRANCH FAST). To learn more about GED in general, one might want to take a look at their Section 3: Preliminaries, especially Definitions 1 and 4. The whole theory regarding GED in chapter 3 of this paper is based on their work.
The library itself is used to compute the graph edit distances for HGC. As of the current state, we don't know of any other GED library written in C++ with a support of heuristics to this extent.
Another C++ library we use is DiCoDa by David Blumenthal [1]. It uses GEDLib itself to further provide methods to parse certain biological data into GEDLib graphs, including a convinient way to set up graph edit path cost.

Regarding the clustering, there are seven different algorithms to specify the distance between two clusters that are provided by SciPy and hence HGC. While the SciPy documentation gives a rather sparse description [26], "Modern hierarchical, agglomerative clustering algorithms" by Daniel Müllner [20] has a good overview, especially in Figure 2, while "Algorithms for hierarchical clustering: an overview" by Fionn Murtagh, and Pedro Contreras [21] goes more into detail with Table 1.

The original data presented in the attachments and in multiple other examples are taken from [19].

Other works to mention here are pybind11 [25], SciPy [27], NetworkX [22] and MatPlotLib [17].
For full references for UML class and sequence diagrams, refer to [30] and [31].

3. Preliminaries

Understanding the GED heuristics and clustering algorithms in detail is not necessary in order to understand how to use HGC. Default values are set, enabling to call the functions that execute them without most parameters. It enables a deeper understanding of the results though, so we don't want to miss out on quickly going over most of the theory in this chapter for those interested.

3.1. Graph Edit Distance

HGC supports three GED heuristics to compute the distance between two graphs G and H: "SUPER FAST", "FAST" and "TIGHT". In order to specify the method, a string with one of these values is passed in the constructor of the HGC environment class that holds the graphs. Based on the choice, one of the following GED heuristics will be applied.
All three heuristics follow the basic structure of mapping the two graphs of which the edit distance is desired onto an instance of the linear sum assignment problem with error correction (LSAPE) in a way that solutions for this instance, which can be found in polynomial time, correspond to upper bounds for the graph edit distance of the two graphs [3]. For more details on how the heuristics work, refer to chapter 2. For now, we will just take a look at the runtimes and some other important properties.

3.1.1. BRANCH

The algorithm BRANCH is the default algorithm used by HGC and is applied when passing "FAST" as the GED method. Its runtime of constructing the LSAPE instance is

$$O(|V_G| \cdot |V_H| \cdot {\Delta_{\min}^{G,H}}^2 \cdot \Delta_{\max}^{G,H}) \tag{3.1}$$

with $|V_X|$ denoting the number of vertices in graph X and
$\Delta_{\max}^{G,H} := \max\{\max\deg(G), \max\deg(H)\}$, $\Delta_{\min}^{G,H} := \min\{\max\deg(G), \max\deg(H)\}$
[3]

3.1.2. BRANCH FAST

BRANCH FAST, being applied when "SUPER FAST" is selected, is a sped up variant of BRANCH. Its main disadvantage is that it produces a looser lower bound [3], but since we only work with upper bounds in HGC, this restriction is not of a concern to us. It can reduce the LSAPE construction time complexity to

$$\begin{aligned} O(\, &\max\{|V_G|, |V_H|\} \cdot \Delta_{\max}^{G,H} \cdot \log(\Delta_{\max}^{G,H}) \\ &+ |V_G| \cdot |V_H| \cdot \Delta_{\min}^{G,H} \cdot \Delta_{\max}^{G,H}) \end{aligned} \tag{3.2}$$

[3].

3.1.3. IPFP

IFPF is the applied heuristic for the "TIGHT" option. As the name suggest, it focuses on producing an upper bound as tight as possible. It includes not only the construction of the LSAPE instance, but also its solving, and iteratively repeates this to improve the solution, until a certain threshold or a maximum number of iterations is reached. Constructing the LSAPE instance in each iteration already requires

$$O(k \cdot |V_G| \cdot |V_H| \cdot \max\{|V_G|, |V_H|\}) \tag{3.3}$$

time, with k being the current iteration. [3]
IPFP is a randomized algorithm and can therefore yield slightly different results each time it's applied. This will be picked up again later in section 4.6.

3.2. Agglomerative Clustering

Once a distance measure $d(a, b)$ between two Graphs a and b is defined, we can proceed to define algorithms to group the graphs into clusters. As mentioned in the introduction, there are two main approaches for this: DIANA (DIvisive ANAlysis), generating the clusters top-down, and AGNES (AGglomerative NESting), generating the clusters bottom-up. Both describe iteratve algorithms, being repeatedly applied on the dataset until the clustering is complete.
In case of DIANA, we start with the n graphs being all part of a single cluster. In each iteration, the largest subcluster based on a certain division method is split until every graph is inside its own cluster that contains no other graphs, resulting in n clusters. Because there are $2^{n-1} - 1$ possibilities to split each cluster, DIANA is quite unattractive for large datasets in terms of runtime. [14]
In case of AGNES, we start with n clusters, each containing a single graph. In each

iteration, the two closest clusters are merged, until all graphs are part of a single cluster.

In order to find the two closest clusters, we have to define a distance measure $d(A,B)$ between two clusters A and B first. Sections 3.2.1 to 3.2.7 describe the different methods of how to use a given graph distance measure $d(a,b)$, in our case GED, to define a cluster distance measure $d(A,B)$.
The basic procedure then consists of merging the two closest clusters I and J and using the Lance-Williams dissimilarity update formula [20] to determine the distances between the newly formed cluster $I \cup J$ and every other cluster K, according to the chosen distance measure. The general shape of the formula is given below.

$$d(I \cup J, K) := \alpha_I d(I,K) + \alpha_J d(J,K) + \beta d(I,J) + \gamma |d(I,K) - d(J,K)| \tag{3.4}$$

Different values for the coefficients α_I, α_J, β and γ define the different update formulas in the algorithms below. A complete table relating coefficience values to the algorithms can be found in [21], Table 1.
Let's take a closer look at those seven dissimilarity measures.

3.2.1. Nearest Point

This algorithm is also called "Single Linkage" [26]. The update formula is simply defined as

$$d(I \cup J, K) = \min(d(I,K), d(J,K)) \tag{3.5}$$

resulting in the general cluster distance measure

$$d(A,B) = \min_{a \in A, b \in B} d(a,b) \tag{3.6}$$

[20].

3.2.2. Farthest Point

This algorithm is also called "Complete Linkage" [26]. In contrast to Single Linkage, it uses the maximum instead of the minimum,

$$d(I \cup J, K) = \max(d(I,K), d(J,K)) \tag{3.7}$$

resulting in the general cluster distance measure

$$d(A,B) = \max_{a \in A, b \in B} d(a,b) \tag{3.8}$$

[20].

3.2.3. UPGMA

The standard algorithm used in HGC is UPGMA, also called "Average Linkage" [26]. It updates the distances by taking the average of both,

$$d(I \cup J, K) = \frac{|I|d(I,K) + |J|d(J,K)}{|I| + |J|} \tag{3.9}$$

resulting in the general cluster distance measure

$$d(A,B) = \frac{\sum_{a \in A} \sum_{b \in B} d(a,b)}{|A||B|} \tag{3.10}$$

[20].

3.2.4. WPGMA

WPGMA, also called "Weighted Linkage" [26], weights both distances equally, not taking the cluster size into account

$$d(I \cup J, K) = \frac{d(I,K) + d(J,K)}{2} \tag{3.11}$$

[20].

3.2.5. UPGMC

The remaining three methods are the more complex ones which assume the input points as vectors in euclidian space. UPGMC is also called "Centroid Linkage" [26] and uses the euclidian distance between the centroids of both clusters as a measure

$$d(I \cup J, K) = \sqrt{\frac{|I|d(I,K) + |J|d(J,K)}{|I| + |J|} - \frac{|I||J|d(I,J)}{(|I| + |J|)^2}} \tag{3.12}$$

[20]. With $\overrightarrow{c}_X$ denoting the centroid of cluster X, defined as the average over all data points of cluster X, the general distance measure is

$$d(A,B) = ||\overrightarrow{c}_A - \overrightarrow{c}_B||_2 \tag{3.13}$$

[26].

3.2.6. WPGMC

WPGMC, also called "Median Linkage" [26] is the WPGMA version of UPGMC,

$$d(I \cup J, K) = \sqrt{\frac{d(I,K) + d(J,K)}{2} - \frac{d(I,J)}{2}} \tag{3.14}$$

again not taking the cluster size into account. With $\overrightarrow{w}_X$ denoting the "midpoint" of cluster X, recursively defined as $\overrightarrow{w}_{I \cup J} = \frac{\overrightarrow{w}_I + \overrightarrow{w}_J}{2}$ (this is basically a centroid that is not recalculated in every step but iteratively updated as the median between the two old centroids), the general distance measure is defined anologously to UPGMC

$$d(A,B) = ||\overrightarrow{w}_A - \overrightarrow{w}_B||_2 \tag{3.15}$$

[20].

3.2.7. Incremental

The Incremental method, also known as "Ward Linkage" [26], uses the Ward variance minimization algorithm

$$d(I \cup J, K) = \sqrt{\frac{(|I| + |K|)d(I,K) + (|J| + |K|)d(J,K) - |K|d(I,J)}{|I| + |J| + |K|}} \tag{3.16}$$

[20]. Using Centroids again, the general distance measure is

$$d(A,B) = \sqrt{\frac{2|A||B|}{|A| + |B|}} ||\overrightarrow{c}_A - \overrightarrow{c}_B||_2 \tag{3.17}$$

[20]. It should be noted that SciPy instead uses squared distances in their version of Formula 3.16 [26].

4. Software Architecture

Now we can finally move on to the central chapter of this paper. It gives a detailed description of the software architecture, its components, how they work together and the overall features, pointing out strengths and weaknesses regarding the requirements, and adresses remaining problems.
Since HGC is supposed to be used as a library and as a tool, we define two types of users: The "direct" user as a programmer who directly calls the public functions of the HGC's central Python class, and the "indirect" user as someone who uses HGC's main executable through the terminal, specifying the parameters with command line arguments.
HGC's functional requirements are:

- HGC must be available as a library that direct users can embedd in their code, and also as a tool, providing its functionality to indirect users.
- HGC must allow the specification of graph edit cost functions, a GED heuristic and a clustering algorithm.
- HGC must work with sets of NetworkX graphs in the Python memory, and include options to load and save such graph sets onto and from the disk.
- HGC must support datasets saved as GML files
- HGC should support datasets saved in a certain CSV format used by the DiCoDa library, from now on called DiCoDa-CSV (more on that in section 4.2).

HGC's most important non-functional requirements are:

- Performance
- Usability
- Robustness

The UML class diagram 4.1 gives a more detailed overview.
To avoid confusion between the word "method", as a way to compute the graph edit distance and as the name for a function in code that's associated with an object, the

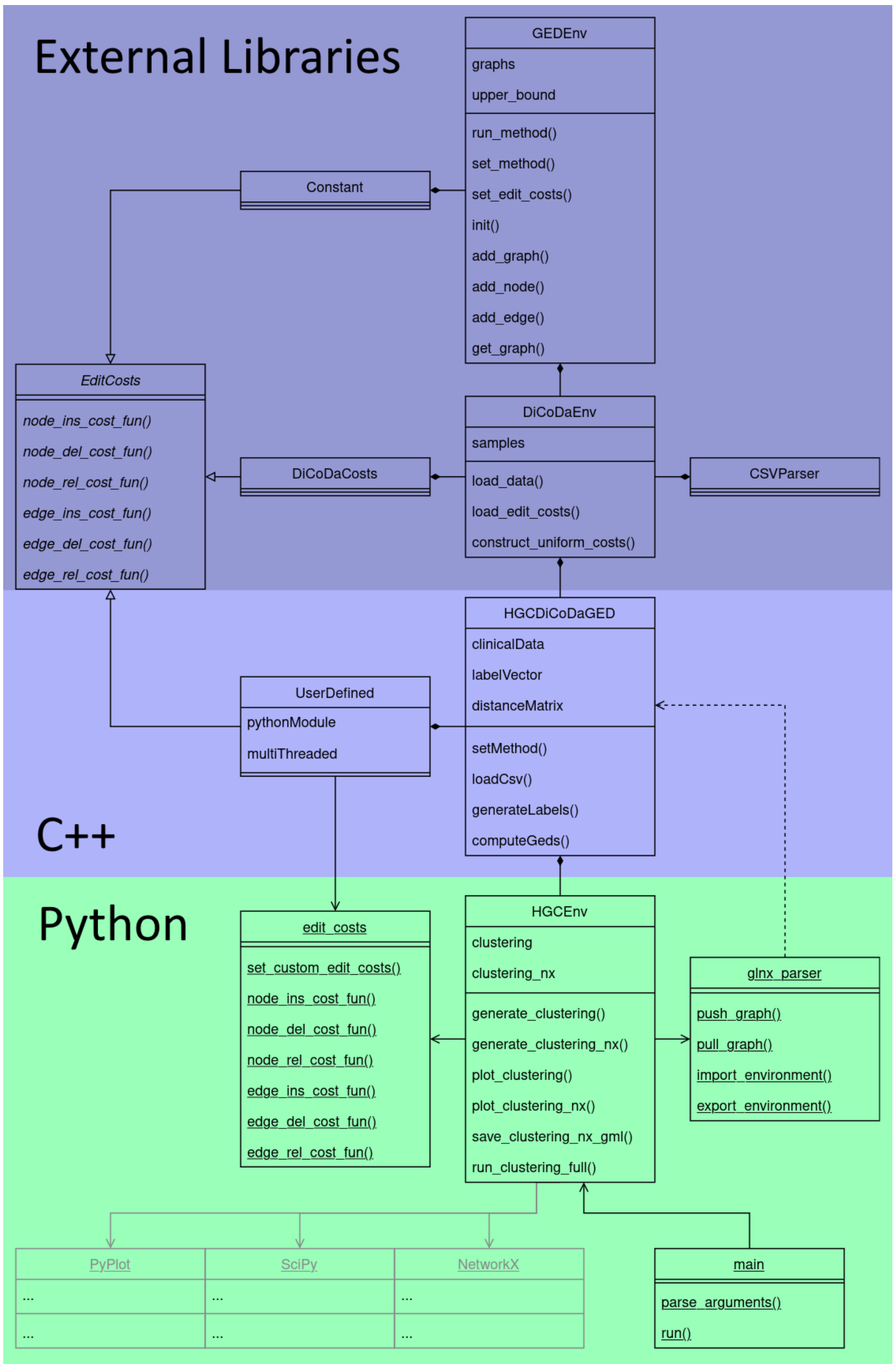


Figure 4.1.: Full overview of HGC

second meaning is from now on also just referred to as "function".
Some classes, especially `HGCEnv` ("HGC environment"), contain additional functions that do nothing more than call functions from other static classes, or other non static classes where they hold a reference to an object of them, with basically the same parameters, in other words delegate the call. Such "delegate" functions as well as constructors and private helper functions are only displayed in the following subdiagrams that focus on one specific class and also include the function signatures, while basic getter functions are completely omitted but mentioned next to the corresponding variable if they exist.
The structure consists of multiple levels: In the Python part, the "main"-module is the user interface for indirect users and the HGC environment class is the main interface to control the software as a direct user. It has a reference to an object of the HGC-DiCoDa-GED class, implemented in C++. This class manages everything C++ related. The DiCoDa environment and GED environment classes are part of the external header-only libraries DiCoDa and GEDLib. GEDLib is completely unchanged while some adjustments to DiCoDa were made, mainly revealing some functions and variables.
The GED environment is used to execute the GED heuristic computations, while DiCoDa, using GEDLib itself, is embedded in order to use its functionality of generating GEDLib graphs, edit costs and graph labels out of DiCoDa-CSV datasets.

4.1. Basic Functionality

We are first going to take a look at the basic funtionality, that is:

- Pushing NetworkX graphs lying in Python memory into the GEDLib C++ environment
- Pulling GEDLib graphs lying in C++ memory out of the environment into Python NetworkX
- Setting the GED method and clustering algorithm
- Calculating the GEDs with constant edit costs
- Generating the clustering and saving it
- Generating graphical figures that depict the clustering and saving them

Figure 4.2 shows the subdiagram containing all parts needed to provide these functionalities, while everything else is greyed out. For now this means especially everything edit costs related (covering it as a whole in its own section 4.3) and everything DiCoDa related is ignored.

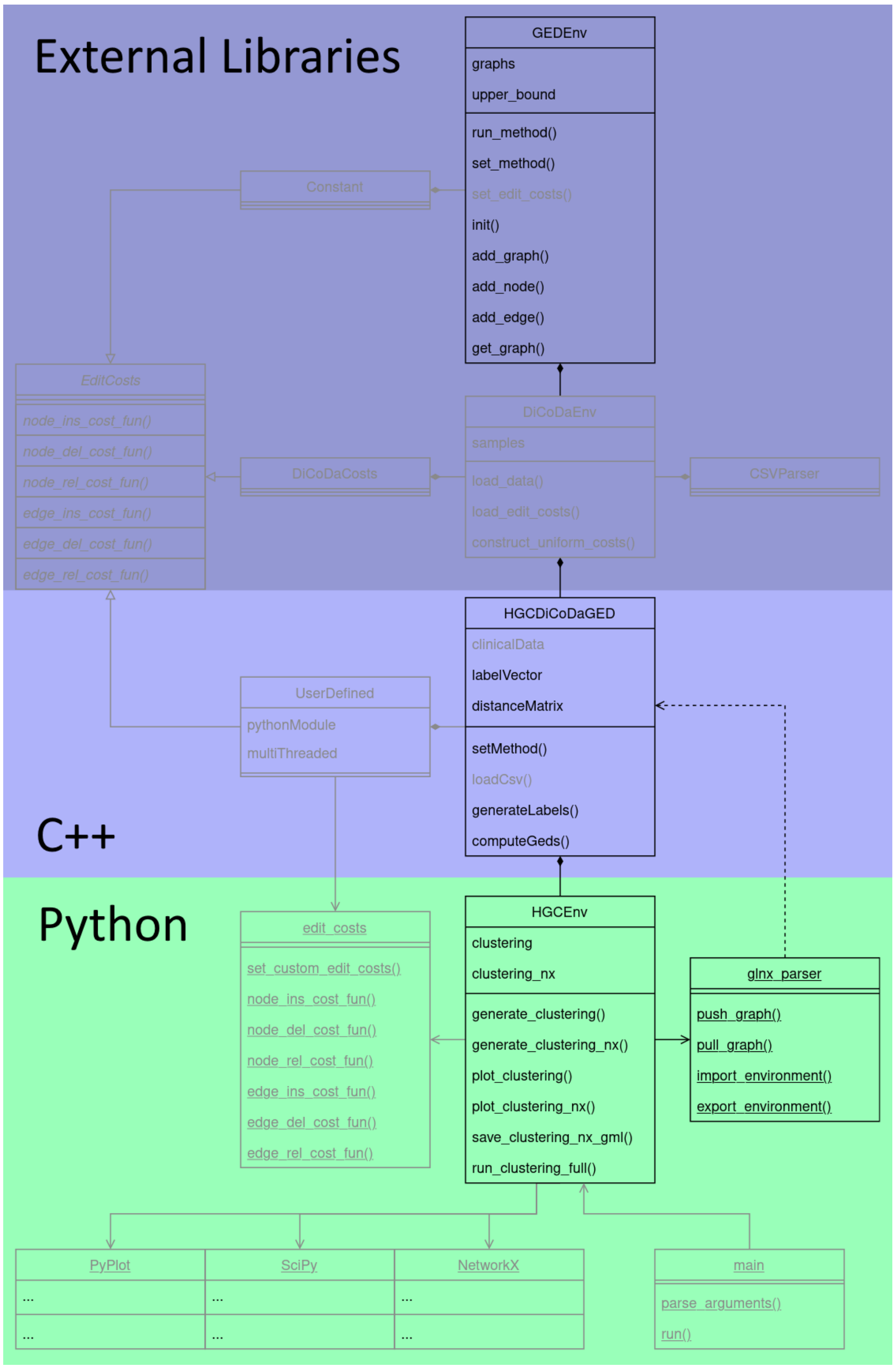


Figure 4.2.: Basic overview of HGC

4.1.1. GED Environment

First of all, it should be mentioned that while the lifetime of a `GEDEnv` object is bound to the lifetime of a `DiCoDaEnv` object and this lifetime again is bound to the lifetime of an `HGCDiCoDaGED` object, as shown by the composition arrows in the class diagram, ignoring the `DiCoDaEnv` object is no problem, since `HGCDiCoDaGED` simply queries the `GEDEnv` object that was indirectly created by the constructor of `DiCoDaEnv` and then works with a direct reference to it.

`GEDEnv` is the main class of GEDLib and contains references to multiple other classes of the library, but since every GEDLib feature can be controlled through this main class and the detailed structure of GEDLib is not part of this paper, we omitted those in the diagram. `GEDEnv`'s most important features are holding a set of graphs in its own format in memory and running the selected GED method on two of them respectively, saving the result in `upper_bound`.

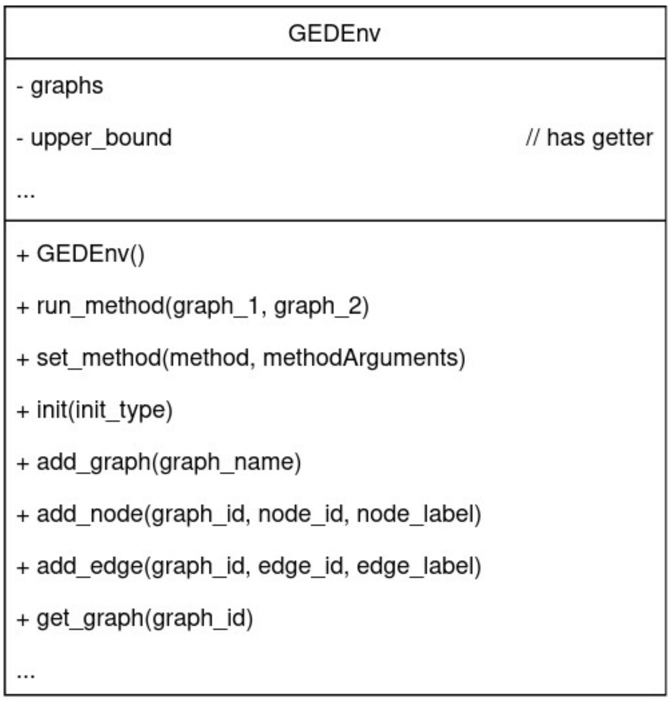

GEDEnv
- graphs
- upper_bound
// has getter
...
+ GEDEnv()
+ run_method(graph_1, graph_2)
+ set_method(method, methodArguments)
+ init(init_type)
+ add_graph(graph_name)
+ add_node(graph_id, node_id, node_label)
+ add_edge(graph_id, edge_id, edge_label)
+ get_graph(graph_id)
...

Figure 4.3.: Specific structure of GEDEnv

As the name suggests, the `HGCDiCoDaGED` class manages the communication between the Python class `HGCEnv` and the C++ library classes `DiCoDaEnv` and `GEDEnv`. Its constructor mainly constructs and initializes a `GEDEnv` and a `DiCoDaEnv` object. Initializing a `GEDEnv` object consists of setting three options: The GED method, the initialization type and the edit costs. String parameters for the first two are required in the constructor of `HGCDiCoDaGED`, parsed and then used there to initialize the `GEDEnv` object using `GEDEnv::set_method()` and `GEDEnv::init()`. The edit costs are handled differently, more on that later.

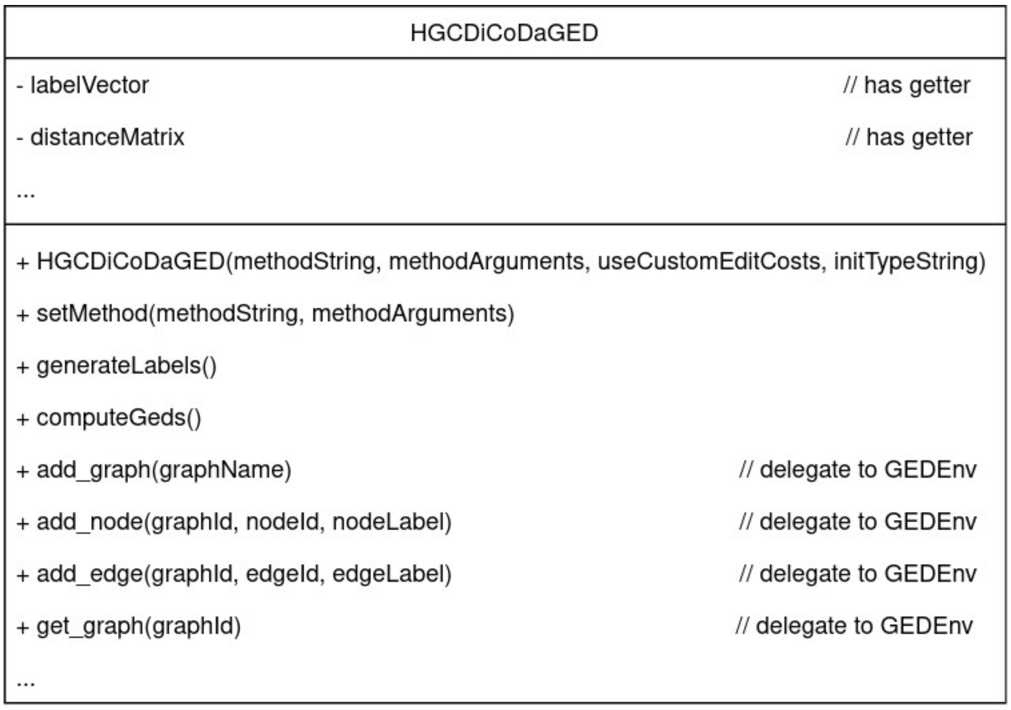


Figure 4.4.: Specific structure of HGC-DiCoDa-GED

The first parameter, `methodString`, specifies the heuristic used to compute the graph edit distance. This can be "FAST", "SUPER_FAST" or "TIGHT". The helper function `HGCDiCoDaGED::setMethod()` parses this parameter and respectively assigns BRANCH, BRANCH_FAST or IPFP as previously mentioned.
`initTypeString` defines the initialization type and is either "LAZY" or "EAGER". "EAGER" makes the environment precompute the whole relabel costs matrix of all label combinations for fast queries afterwards, while "LAZY" allows a combination to be computed not until that certain value is actually queried. For large datasets, "LAZY" is recommended. The private helper function parsing it is omitted in the figure.
An additional parameter is `methodArguments`, which is a string containing command-line style arguments that can be additionaly stated when calling `GEDEnv::set_method()`. Those arguments partially depend on the selected method, but for instance

"`--threads <thread-number>`" is valid for every method and determines the number of threads allocated to the computation, in order to increase the performance.

After a `GEDEnv` object is initialized, `HGCDiCoDaGED::computeGeds()` can be called. It simply loops through all two-elementary graph combinations of graphs saved in `GEDEnv` and executes `GEDEnv::run_method()` on them, then saving their `GEDEnv::upper_bound` in the associated slot in `HGCDiCoDaGED::distanceMatrix`.
Finally, every `GEDEnv` graph has a string attribute containing its name. `HGCDiCoDaGED::generateLabels()` loops through the graphs and saves their names in `HGCDiCoDaGED::labelVector`.

4.1.2. HGC Environment & GLNX Parser

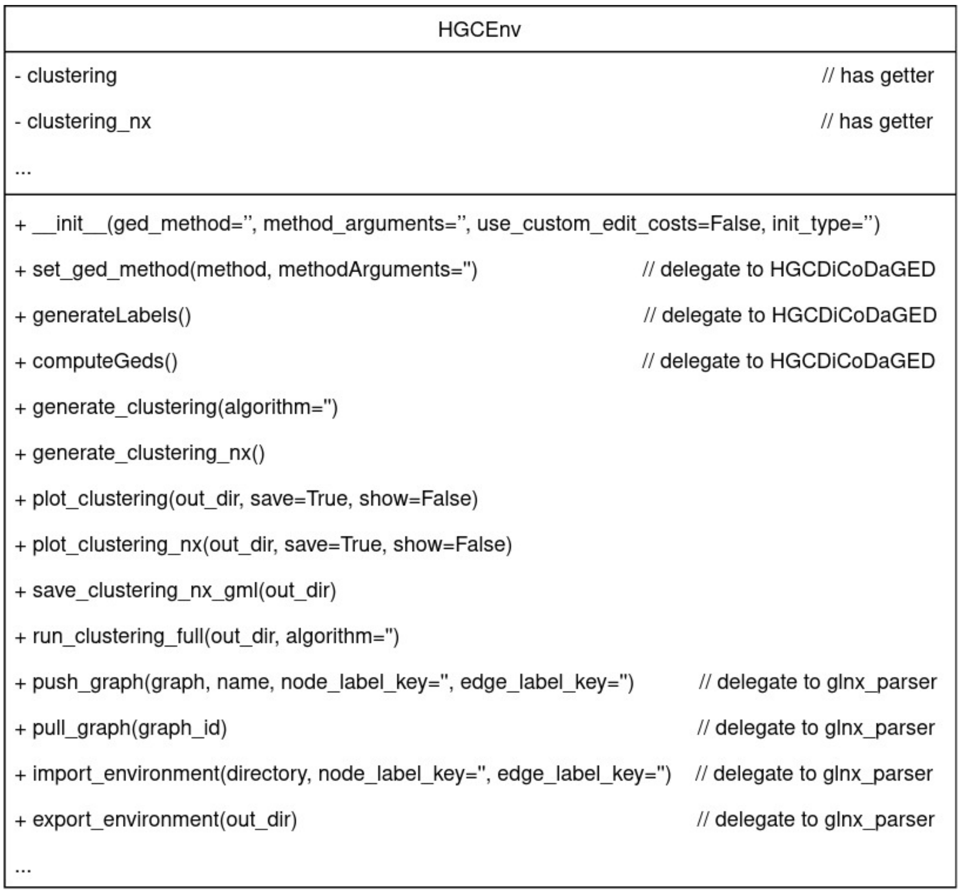

```
HGCEnv
- clustering                                                              // has getter
- clustering_nx                                                           // has getter
...
+ __init__(ged_method=”, method_arguments=”, use_custom_edit_costs=False, init_type=”)
+ set_ged_method(method, methodArguments='')                  // delegate to HGCDiCoDaGED
+ generateLabels()                                            // delegate to HGCDiCoDaGED
+ computeGeds()                                               // delegate to HGCDiCoDaGED
+ generate_clustering(algorithm='')
+ generate_clustering_nx()
+ plot_clustering(out_dir, save=True, show=False)
+ plot_clustering_nx(out_dir, save=True, show=False)
+ save_clustering_nx_gml(out_dir)
+ run_clustering_full(out_dir, algorithm='')
+ push_graph(graph, name, node_label_key='', edge_label_key='')         // delegate to glnx_parser
+ pull_graph(graph_id)                                                  // delegate to glnx_parser
+ import_environment(directory, node_label_key='', edge_label_key='')   // delegate to glnx_parser
+ export_environment(out_dir)                                           // delegate to glnx_parser
...
```

Figure 4.5.: Specific structure of HGCEnv

This Python class acts as the main API to HGC. Everything can be controlled through its public functions. It holds a private reference to an `HGCDiCoDaGED` object that is constructed in its `__init__()` function, which therefore asks for the same parameters as above, just passing them on.
`HGCEnv` also contains public delegates for all public functions of its `HGCDiCoDaGED` object (in that case using pybind) and of `glnx_parser`, fully centralizing the software around this main class.
The reason the helper function `HGCDiCoDaGED::setMethod()` is also public and `HGCEnv` therefore has the delegate `set_ged_method()` to it, is that this enables to set the GED method anew, even after the `GEDEnv` object is constructed. After all, we don't want to be required to construct a new environment and reload all graphs, only to run the clustering with a different graph edit distance heuristic.
After creating an `HGCEnv` object and therefore an initialized GED environment, it can be filled with graphs.

The `glnx_parser`, short for GEDLib-NetworkX Parser, is a Python module working like a static class with static public functions, that can convert NetworkX memory graphs into `GEDEnv` memory graphs and vice-versa. functions all require an instance of `HGCDiCoDaGED` as their first parameter to know which `GEDEnv` to push graphs to and pull graphs from. The delegate functions in `HGCEnv` set this parameter automatically.
`GEDEnv` contains library functions to add (`add_graph()`), fill (`add_node()` & `add_edge()`) and get graphs (`get_graph()`), and delegate functions to them in `HGCDiCoDaGED` are callable through pybind, which is what the parser eventually does.
NetworkX graphs can hold arbitrarily many attributes for every node and edge. This is implemented with dictionaries of the shape `<attribute key, attribute value>` [13]. In our case, `GEDEnv` graphs have node labels of type `std::size_t`, which is in most cases `unsigned long`, and edge labels of type `double`. When pulling a graph out of `GEDEnv` and converting it into the NetworkX format, the parser stores these labels with the keys `"hgc_node_label"` and `"hgc_edge_label"`.
When pushing a NetworkX graph into the environment, the user can pass the attribute keys which define the desired node and edge attributes to be used, as function parameters. The default values are the same ones used when pulling a graph.
The parser will always try to convert the label values to `int`s and `double`s and raises an exception if it is impossible.
Since NetworkX contains built-in functions to write a NetworkX graph into, and read a NetworkX from a GML file [11], `export_environment()` and `import_environment()` can be implemented pretty straight-forward.
`export_environment()` simply pulls every graph out of the environment and saves it

in a GML file named after the graph.
`import_environment()` does the reverse, going over every GML file in a given directory and reading it, then pushing its NetworkX graph into the environment.

After filling the environment with graphs, `HGCEnv`'s delegates to `HGCDiCoDaGED::computeGeds()` and `HGCDiCoDaGED::generateLabels()` can be called. Then, with `HGCDiCoDaGED::labelVector` and `HGCDiCoDaGED::distanceMatrix` populated, `HGCEnv` can run the SciPy clustering algorithms with `HGCDiCoDaGED::distanceMatrix` as the input.

`generate_clustering()` takes care of that, taking a string to the desired clustering algorithm as a parameter. Available are "nearest point", "farthest point", "upgma", "wpgma", "upgmc", "wpgmc" and "incremental". What they do was covered in chapter 3. The central part of the function is a call to SciPy's `linkage()` function that executes the algorithm and a call beforehand to SciPy's `pdist()` function that transforms the distance matrix into a condensed form, the format used by `linkage()` [28]. What that means and its consequences is explained in section 4.6.
`generate_clustering_nx()` converts the generated clustering into a NetworkX graph. This is done in order to be able to use the clustering results generically as well, since the format in which the `linkage()` function returns it is comparably specific [26].
Since the clustering is now present in SciPy's own format (`clustering`) and in a generic NetworkX format (`clustering_nx`), it can be saved and plotted easily using NetworkX and pyplot [18]:
`plot_clustering()`: plots `clustering` as a dendrogram and saves the plot.
`plot_clustering_nx()`: plots `clustering_nx` and saves the plot.
`save_clustering_nx_gml()`: saves `clustering_nx` in the GML format.

Wrapping it up, `run_clustering_full()` is a public helper function that makes use of HGC's full functionality by running all the just mentioned functions consecutively.

```python
def run_clustering_full(self, out_dir, algorithm=''):
    self.generate_clustering(algorithm)
    self.generate_clustering_nx()
    self.plot_clustering(out_dir, save=True, show=False)
    self.plot_clustering_nx(out_dir, save=True, show=False)
    self.save_clustering_nx_gml(out_dir)
```

Listing 4.1: Public helper function to run everything clustering-related

When plotting, `HGCDiCoDaGED::labelVector` is finally used to caption the nodes in the dendrogram that correspond to the original data points, in our case graphs, and also as labels for the leafs of `clustering_nx` since they also correspond to the original data points, while non-leaf nodes are labeled with the distance between their two corresponding subclusters. These labels are also saved in the GML file created by `save_clustering_nx_gml()`.

Finally, in order to fully release the results to the user, `HGCEnv` contains getter functions for `clustering` and `clustering_nx`.

Following this standard procedure while using default parameters, constant edit costs will automatically be assigned. They simply return a cost of 1 in each of the six edit cost functions.

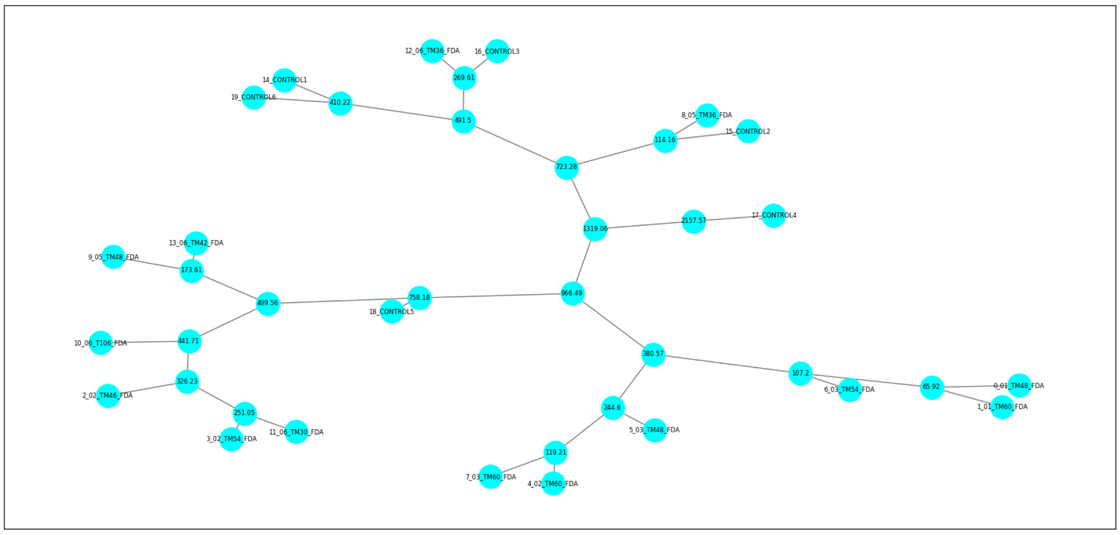


Figure 4.6.: Example figure generated by plot_clustering_nx(). Data from [19]

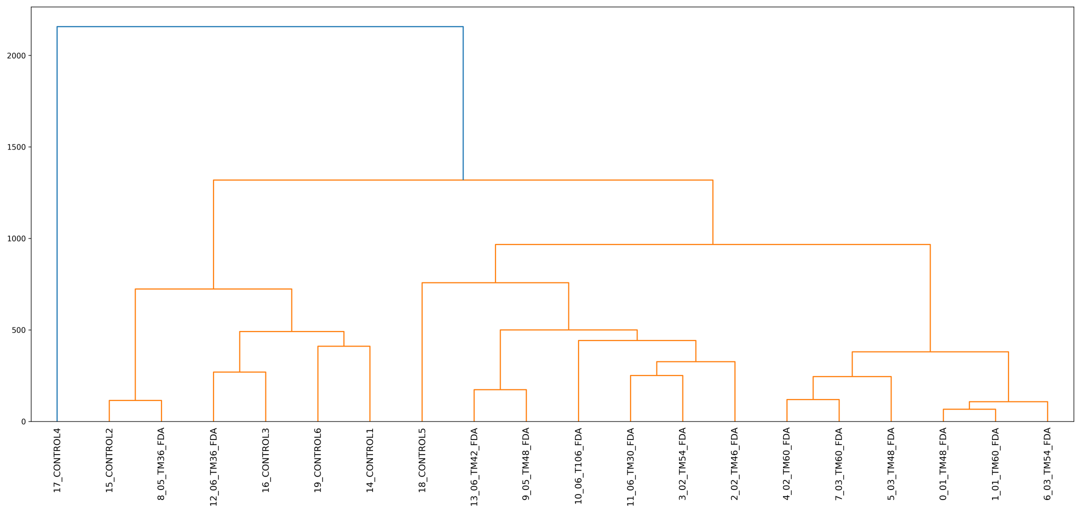


Figure 4.7.: Example dendrogram generated by plot_clustering(). Data from [19]

4.2. Additional Functionality

This section goes over the DiCoDa-CSV format and the functionality added to HGC by the DiCoDa environment class, but again skips edit costs, being fully covered after. That makes:

- Parsing DiCoDa-CSV omics & clinical datasets and using them to
  - generate graphs
  - label graphs with clinical attributes (one label per graph)

Figure 4.8 shows these additions.

4.2.1. DiCoDa-CSV format

Since HGC was originally conceptualized in the context of bioinformatics, it supports a special type of CSV format, natively used by the DiCoDa library. It consists of two mandatory and one optional CSV files. They define omics data, clinical data and distances data. Basic explanations for them are provided here, for more detail refer to [19].

Omics Data

The word "omics" is derived from the suffix of words like "genomics" and is used for the actual biological data to work with, whatever it may describe in the individual case.

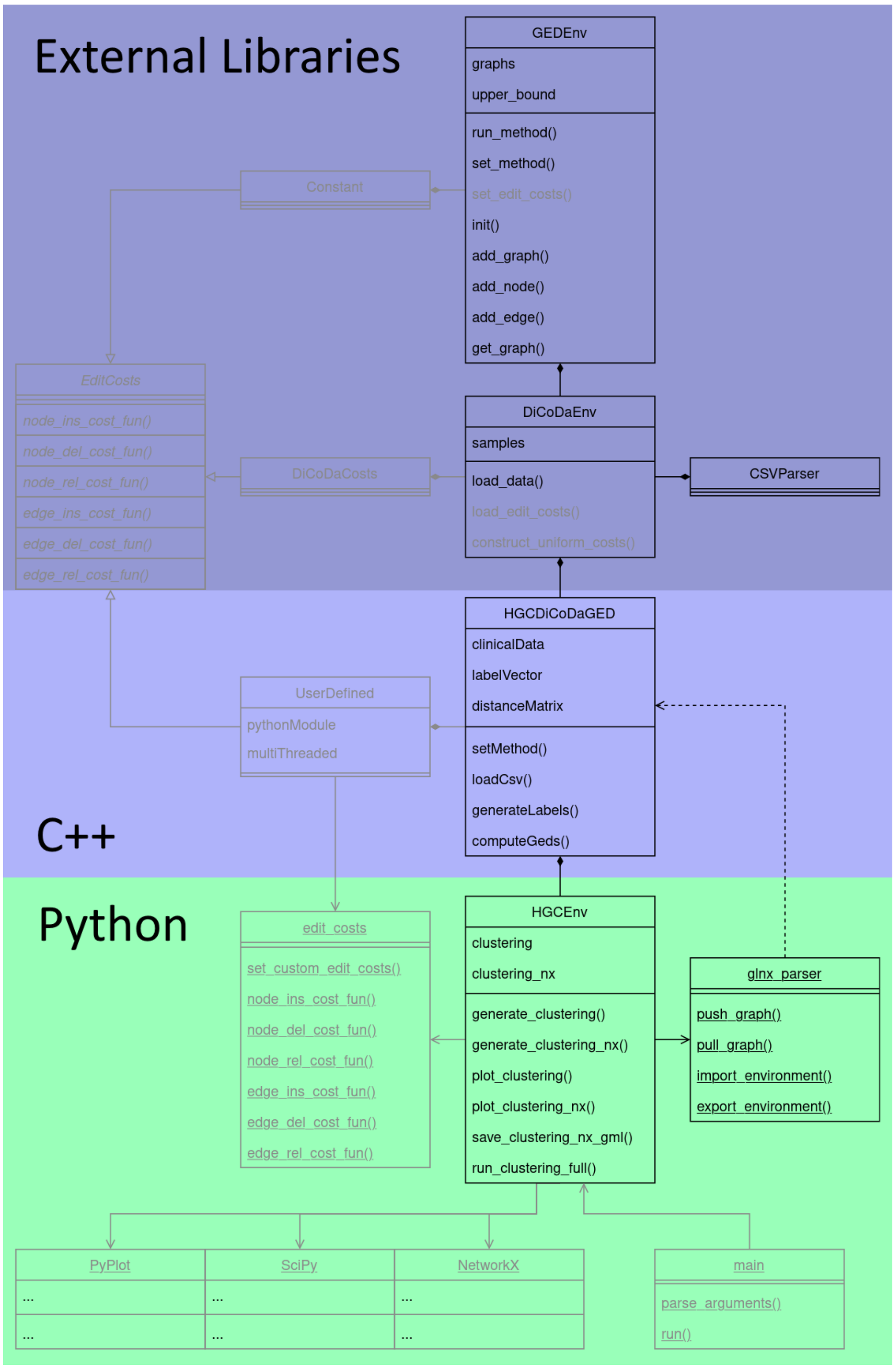


Figure 4.8.: Advanced overview of HGC

The omics CSV format originally describes a matrix A with samples as rows and OTU's as columns. In biology, an OTU, standing for "Operational Taxonomic Unit", describes a group of similar microorganisms. This similarity between two microorganisms is defined with respect to some given percentage x by their DNA sequence of a certain marker gene matching to $x\%$ [2].
The first column contains the sample IDs, the first row contains the OTU IDs. Starting from row and column 1, entry $a_{sampleID,\, OTUID}$ contains the abundance of that OTU in that sample. This abundance says how much percent of the microbiome's sample is made up of microorganisms from that OTU [19].
To take this to a more general level, it can be seen as some co-occurence measure between combinations of variables and environments. To keep a generalized vocabulary, we will use the words samples (column data) and features (row data) from now on.

Clinical Data

The clinical data is used to give the samples some attributes. It is required to contain the same samples as the omics data, also placed in the first column, while the first row contains names of clinical attributes, and $a_{sampleID,\, attributeID}$ is the value of that attribute for that sample.
In HGC, these attributes can be used to label the original graphs in the clustering instead of using their name. This enables analyzing the results with respect to any desired clinical attribute.

Distance Data

The distances dataset CSV is used for the edit costs in HGC. Again, this is covered in detail in section 4.3.

4.2.2. DiCoDa Environment

Just like `GEDEnv` for GEDLib, `DiCoDaEnv` is the main class of the external DiCoDa library. This library is included in HGC mainly in order to use its functionality of reading DiCoDa-CSV datasets using its `CSVParser`, and constructing GEDLib graphs out of them.
So `HGCDiCoDaGED` now also contains `loadCsv()`, which calls the helper function `HGCDiCoDaGED::loadClinicalData()` and `DiCoDaEnv::load_data_()`. The first saves the attributes for every sample in `HGCDiCoDaGED::clinicalData`. The second saves the sample IDs (used for attribute look-up) in `DiCoDaEnv::samples` and uses the omics data to generate a graph for every sample, representing its respective feature data. Since DiCoDa uses GEDLib itself, these graphs are directly generated into the `GEDEnv`

object to which `DiCoDaEnv` has a reference to.
One graph is created for every sample and every graph contains one node for every non-zero feature of the sample. The node is then labeled with a numerical ID for its feature. The edges and edge labels are generated out of the actual co-occurences data.

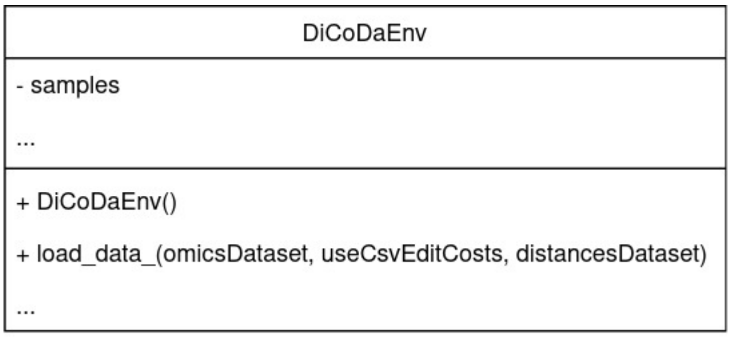

DiCoDaEnv
- samples
...
+ DiCoDaEnv()
+ load_data_(omicsDataset, useCsvEditCosts, distancesDataset)
...

Figure 4.9.: Specific structure of DiCoDaEnv

Figures 4.10 and 4.11 show the additions made to the other classes.

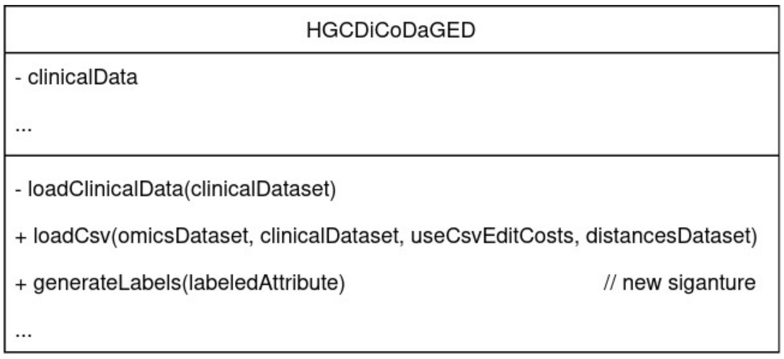

HGCDiCoDaGED
- clinicalData
...
- loadClinicalData(clinicalDataset)
+ loadCsv(omicsDataset, clinicalDataset, useCsvEditCosts, distancesDataset)
+ generateLabels(labeledAttribute)
// new siganture
...

Figure 4.10.: Additions to HGC-DiCoDa-GED

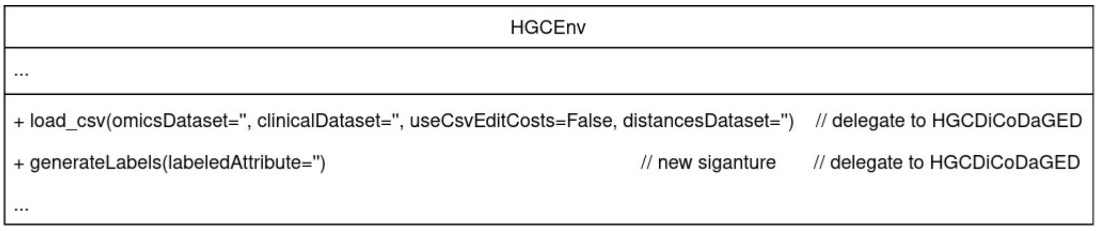

HGCEnv
...
+ load_csv(omicsDataset='', clinicalDataset='', useCsvEditCosts=False, distancesDataset='')
// delegate to HGCDiCoDaGED
+ generateLabels(labeledAttribute='')
// new siganture
// delegate to HGCDiCoDaGED
...

Figure 4.11.: Additions to HGCEnv

Since there are now graph attributes available, we can take a more extended look at the `HGCDiCoDaGED::generateLabels()` function, which now accepts a clinical attribute whose values are to be used as labels.
If an empty string is passed as `labeledAttribute`, the function goes back to using graph names. If not, it looks up the passed attribute and uses its value for the respective graph as its label.
With the same data as before [**3**], this feature is now applied with the clinical attribute "HSCT_responder" in Figure 4.12.

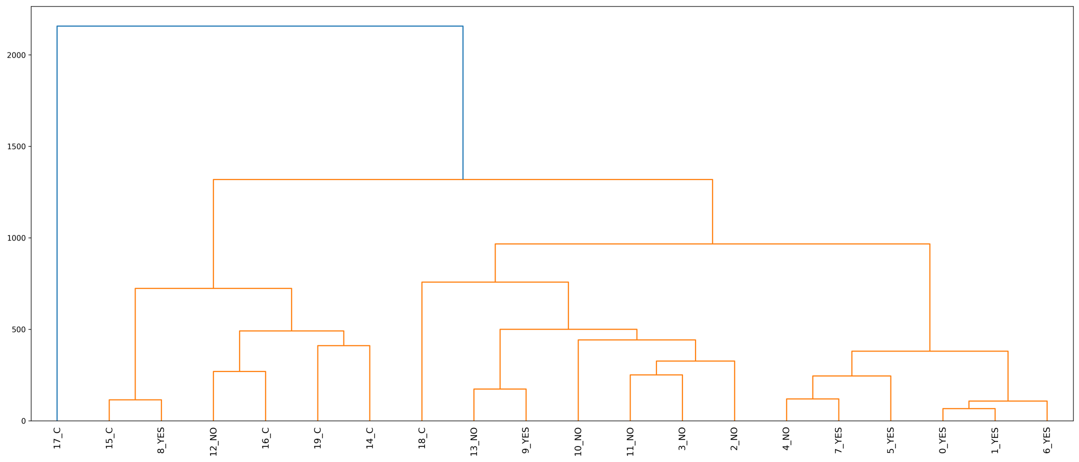


Figure 4.12.: Dendrogram from Figure 4.7 with attribute values as labels

4.3. Edit Costs Support

This chapter covers everything regarding the edit costs. The reason this is done so late resulting in numerous references to it beforehand is that in order to cover it as a whole, all other features of HGC had to be presented first. That also means we can now basically work with the full class diagram as presented in Figure 4.1. Only the `main` module is not important yet.

Fundamentally, edit costs are just another parameter used to initialize a `GEDEnv` object, just like the GED method and the initialization type. The GEDLib library contains the abstract class `EditCosts` with declarations for the six cost functions `node_ins_cost_fun`, `node_del_cost_fun`, `node_rel_cost_fun`, `edge_ins_cost_fun`, `edge_del_cost_fun` and `edge_rel_cost_fun` for that matter, as well as multiple implementations to choose from that overwrite them in their own way. The new function `GEDEnv::set_edit_costs()`

then takes an object of type `EditCosts` as a parameter, accepting all of its implementations. Unfortunately, GEDLib's own implementations depend strongly on the GXL datasets GEDLib natively provides and can in most cases not be used together with HGC's and DiCoDa's node and edge label types. But there are other ways to provide a range of edit costs to choose from.

4.3.1. Constant

GEDLib's most basic `EditCosts` implementation, `Constant`, is compatible with all label types. It simply returns a cost for 1 in every of the six cost functions. The case of relabeling makes a reasonable exception, as 0 is returned instead if the labels were already the same in the first place.

4.3.2. DiCoDa Costs

DiCoDa also has its own way of defining edit costs for its CSV data. So far, we didn't cover the distances dataset. It contains a $n \times n$ matrix A with n as the number of features, where the first row as well as the first column contain those feature IDs, respectively. The distance between the features in slots i and j is then written into the entry $a_{i,j}$. Within the biological context of the sample data for this paper [19], this distance describes the result of the similarity measure between the two OTUs in slots i and j.
Since the feature values are used as node labels, DiCoDa simply defines the node relabel costs between the node labels l_1 and l_2 as the distance a_{l_1,l_2}, scaled by a constant factor k, which is 0.25 by default.
k is also returned for each of the four cost functions regarding insertion and deletion, while the edge relabel costs are defined as the absolute difference between the two edge labels, also scaled by k, since DiCoDa knows the edge labels to always be `double`.
If no distances dataset is specified, uniform edit costs are created by populating A with 1s, again making the exception of using 0 instead in entries $a_{i,i}$ of same node labels.
Since DiCoDa itself natively uses GEDLib, its `DiCoDaCosts` implementation already inherits from GEDLib's abstract `EditCosts` class.

Here it is important to mention that as soon as DiCoDa costs are in use, importing a graph into the environment that contains a node with a label that does not correspond to a feature ID and is therefore not contained in A, the GED computation will run into a problem when trying to determine its edit distance to other graphs. DiCoDa costs are to be used with caution and when importing a NetworkX graph, it must be made sure that its node labels are compatible with the loaded DiCoDa costs.

4.3.3. User Defined

Lastly, the user should be given the possibility to manually define their own edit costs. For that purpose, HGCDiCoDaGED also contains a reference to a special implementation of GEDLib's EditCosts.
Here, the problems of providing usability together with performance shows the most. Since HGC is required to provide its features through Python, the C++ class UserDefined has to somehow get access to six edit cost functions defined by the user in Python. Since functions are basically associations between names and statement sequences [8], the challenge is to find a way of associating a C++ name with a Python statement sequence. Normally, when working with dynamic function definitions in C++, function pointers are used. They can be dynamically assigned and reassigned to starting point adresses of implemented functions. The problem is that C++ doesn't recognize adresses to Python functions as permitted adresses to point to. The following code example therefore results in a segmentation violation.

main.py

```python
import functionTest


def add(arg1, arg2):
    result = arg1 + arg2
    print(result)


functionTest.set_function(add)
functionTest.call_function(1, 1)    # SIGSEGV
```

functionTest.cpp

```cpp
#include <iostream>
#include <pybind11/pybind11.h>
#include <pybind11/functional.h>

std::function<void(int, int)>* functionPtr;

void setFunction(
        std::function<void(int, int)>& function
    ) {
    functionPtr = &function;
}

void callFunction(int arg1, int arg2) {
    (*functionPtr)(arg1, arg2);     // SIGSEGV
}

PYBIND11_MODULE(functionTest, module) {
    module.def("set_function", &setFunction);
    module.def("call_function", &callFunction);
}
```

Listing 4.2: SIGSEGV reproduction code

In C++, setFunction() sets the pointer functionPtr to the memory adress of the add() function defined in Python. Then, in callFunction(), it is invoked. At this line, the programm exits with the message:

```
Process finished with exit code 139 (interrupted by signal 11: SIGSEGV).
```

This means that the only way to embedd a Python defined function in C++ code, is to make Python invoke it, which means using pybind.
For that purpose, `edit_costs` is a Python module that simply acts as a container for user defined edit cost functions. Its static function `set_custom_edit_costs()` simply assigns the six global variables to the correspondig six parameters it requires. With this setup, the user can write edit costs functions themselves and pass them onto `set_custom_edit_costs()` to have them saved in `edit_costs`.
pybind supports importing a Python module and executing functions of it in C++ code, so `UserDefined` then imports that `edit_costs` module into its global `pythonModule` variable when constructed. Its implementations of the abstract edit cost functions then simply delegate the call to their corresponding Python implementation.
The file structure required for this to work is that the Python files, especially `edit_costs.py`, lie in the root folder (most likely called "src"), while the shared object files containing the compiled C++ code lie in a subfolder named "lib" inside the root folder.

There is a big problem with this though. The simplified UML sequence diagram 4.13 helps in pointing it out.

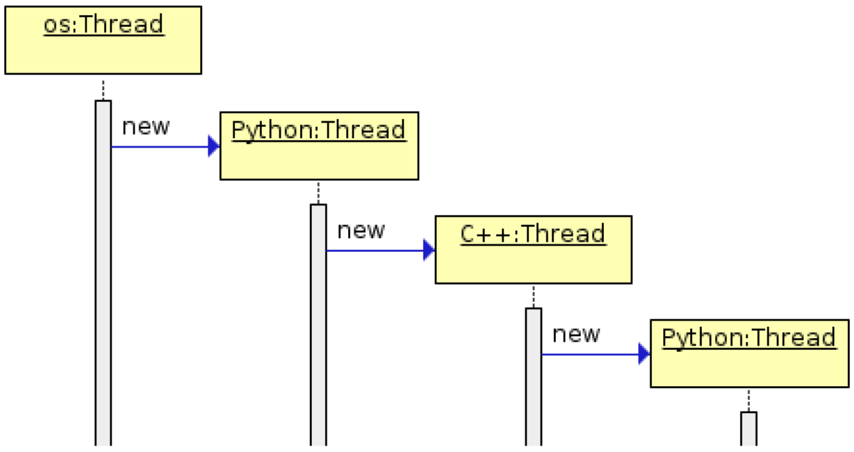


Figure 4.13.: UML sequence diagram for threads

In order to run the program, the operating system will at some point start a thread to interpret the Python source code of `HGCEnv`. At some point, its delegate to `HGCDiCoDaGED::computeGeds()` function will be called, starting a thread to run the compiled C++ code. If custom edit costs are in use, this will result in a cost function delegating its call to the imported `edit_costs` Python module through pybind, which will then start a second thread to interpret Python source code. At this point the program runs into a problem: The Python interpreter itself is treated as a resource by threads. While a thread has the interpreter in use, it is locked by the GIL, the "Global Interpreter Lock". That means while the first Python thread still exists, the second one is not allowed to access the interpreter [10]. This circumstance marks one of the most restricting constraints when using Python for performance-heavy tasks [29].

It becomes an even bigger problem if upon setting the GED method, the stated method arguments include a large number of threads to use. In this case, as depicted in Figure 4.14, the C++ thread will create multiple new C++ subthreads that will all proceed to create a new Python thread on their own, ending up with lots of simultaneously running python threads that all want to access the global intepreter.

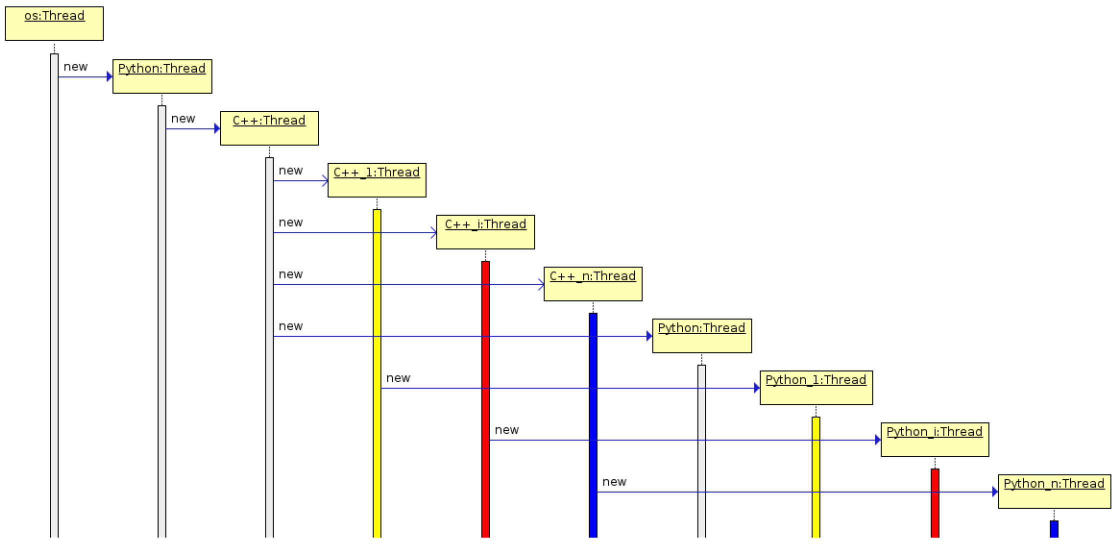


Figure 4.14.: UML sequence diagram for threads: multi-threading

Fortunately, pybind supports releasing the GIL and acquiring it during a certain program scope. `UserDefined`’s boolean `multiThreaded` is set after parsing the `methodArguments` and determines what to do with the GIL.
In case of single threading, it’s enough to acquire it once, just before the `HGCDiCoDaGED::computeGeds()` function, which contains the nested loop that runs the GED method between all graphs, is called. For that, the actual implementation is moved into the private helper function `HGCDiCoDaGED::computeGedsGilScope()`, while `computeGeds()` now only decides whether the GIL should be acquired or not and then invokes that helper function.

In case of multi-threading, it's not enough, since as showed earlier, the thread itself will create multiple C++ and therefore multiple Python threads. In this case, the GIL acquisition has to happen inside the `UserDefined` cost functions themselves, right before the call is delegated to the Python module.
A discussion of how this effects the fulfillment of the requirements follows in subsection 4.5.2.

4.3.4. Selection Management

There are four different edit costs to chose from established:

- Constant
- DiCoDa distances
- DiCoDa uniform
- UserDefined

The selection of which one to use happens in `HGCEnv`'s `__init__()` and `load_csv()` functions, whose definitions were already partially covered.
The important parameters now are `use_custom_edit_costs` of `__init__()`, and `use_csv_edit_costs` and `costs_dataset` (which refers to a distances dataset) of `load_csv()`.
Table 4.1 visualizes which option combinations select which edit costs.

	`use_custom_edit_costs == False`	`use_custom_edit_costs == True`
`use_csv_edit_costs == False`	Constant	UserDefined
`use_csv_edit_costs == True`	`costs_dataset == ''`: DiCoDa uniform `costs_dataset != ''`: DiCoDa distances	

Table 4.1.: Edit costs selection

When calling `load_csv()` with `use_csv_edit_costs` set to `False`, the previous choice, based on what was passed in `__init__()`, is kept.
When specifying a distances dataset in `load_csv()` together with overwriting that previous choice by setting `use_csv_edit_costs` to `True`, DiCoDa generates the edit costs based on the dataset with `load_edit_costs()`.
If the specified distances dataset path is empty but DiCoDa is still made to use its CSV edit costs, it will generate the uniform costs with `construct_uniform_costs()`.

And lastly, passing a valid distances dataset path but leaving `use_csv_edit_costs` on `False` will just ignore it and not change the previously selected edit costs in any way.

Finally, figures 4.15 to 4.18 show all the new additions again.

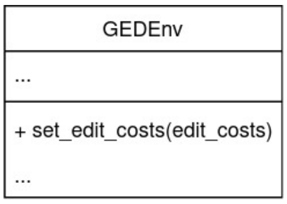


Figure 4.15.: Additions to GEDEnv

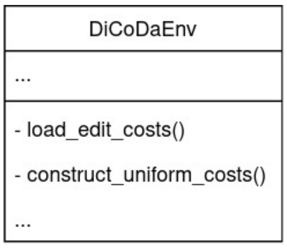


Figure 4.16.: Additions to DiCoDaEnv

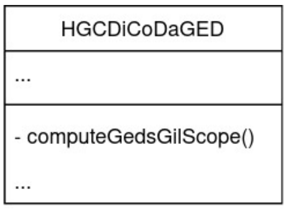


Figure 4.17.: Additions to HGC-DiCoDa-GED

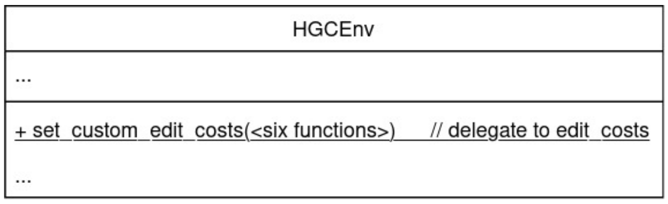

HGCEnv
...
+ set_custom_edit_costs(<six functions>) // delegate to edit_costs
...

Figure 4.18.: Additions to HGCEnv

4.4. User Interface

At this point, the class HGCEnv as the interface for direct users is fully established. The main module finally enables indirect users to execute HGC with the system terminal. For that purpose it contains a function to parse command line arguments. The possible arguments with their supported values are given below.

```
-out <path-to-out-directory>
-gml <path-to-directory>
[-gml_node_label <node-label-key>]
[-gml_edge_label <node-label-key>]
-csv_omics <path-to-csv-file>
[-csv_clinical <path-to-csv-file>]
[-csv_distances <path-to-csv-file>]
[-edit_costs constant|custom|auto]
[-label_attribute <clinical-attribute>]
[-cluster_algo nearest_point|farthest_point|upgma|wpgma|upgmc|wpgmc|incremental]
[-ged_method SUPER_FAST|FAST|TIGHT]
[--<method-option> <method-arg>] [...]
[-init_type LAZY|EAGER]
```

All optional arguments, denoted by square brackets, have default values. More on that in subsection 4.5.3. The user can specify a path to a directory containing GML files (-gml), and also directly use DiCoDa-CSV files. If the first is done, -csv_omics is also optional.
Setting edit_costs to auto will use DiCoDa costs if DiCoDa-CSV data was loaded and constant otherwise.

After parsing the arguments, the main program simply calls HGCEnv's public functions in the correct order.

```python
def run():
    #   parse
    parse_arguments(sys.argv)

    #   edit costs
    if edit_costs == "custom":
        hgc_env.set_custom_edit_costs(node_ins_cost,
                                      node_del_cost,
                                      node_rel_cost,
                                      edge_ins_cost,
                                      edge_del_cost,
                                                 edge_rel_cost)

    #   construct
    hgc = hgc_env.HGCEnv(ged_method,
                              method_arguments,
                              True if edit_costs == "custom" else False,
                              init_type)

    #   csv
    if csv_omics_path != '':
        hgc.load_data(csv_omics_path,
                          csv_clinical_path,
                          True if edit_costs == "auto" else False,
                          csv_distances_path)

    #   gml
    if gml_path != '':
        hgc.import_environment(gml_path,
                                   gml_node_label_key,
                                   gml_edge_label_key)

    #   label
    hgc.generate_labels(labeled_attribute)

    #   ged
    hgc.compute_geds()

    #   cluster
    hgc.run_clustering_full(out_path, cluster_algorithm)
```

Listing 4.3: Main run function

4.5. Requirement Analysis

Now that the full functionality of HGC is covered and all functional requirements are fulfilled, it's time to deal with the non-functional requirements.

4.5.1. Usability

When working with graphs, NetworkX is one of the most widely used Python packages. Direct users can use `push_graph()` to directly move NetworkX graphs from their program into the environment without the need to save them before and then use `import_environment()`. Since HGC is a class, multiple HGC environments can be created, each holding their own data. And due to the delegate functions, everything can be done by calling public functions of `HGCEnv`, with no need to study the sub-parts. The generated results can also be fully embedded into a main program that uses the library, by calling the getter functions that return the clustering raw or converted to a NetworkX graph itself.
With the `main` module, the API links are then built onto and used to create a tool that loads GML graphs or DiCoDa-CSV data and saves their clustering as an image and as a GML graph itself, which ultimately just moves the input & output shape from the more abstract shape of data structures within the memory, managed by a programming language, to generic files on the disk.
Additionaly, the possibility to choose the clustering algorithm and GED method, together with a number of threads to use, provides users with the ability to adjust the trade-off between response time and accuracy as well as response time and resource consumption in any way wanted.

4.5.2. Performance

The most crucial parts regarding runtime are the following:

- `import_environment()` & `export_environment()`
- `compute_geds()`
- `generate_clustering()`

and all communication spots between Python and C++ that require the use of pybind should be payed attention to.
In `generate_clustering()`, the clustering algorithm is called by `scipy.cluster.hierarchy.linkage()`. This function is also never called in a loop, giving it its own runtime that depends on its implementation in SciPy, which can be expected to be optimized as much as possible.
`import_environment()` and `export_environment()` respectively call `push_graph` and `pull_graph()` in loops, which then again include one-sided calls through pybind to C++ functions that add and get graphs, nodes and edges, suggesting a rather bad runtime. However, experience shows that they are still incomparably faster than `compute_geds()`,

which executes the GED method in C++ using `run_method()` n^2 times, with n as the number of graphs.
Reducing the runtime of `run_method()` is the main challenge of GEDLib's heuristics, that ensure us not more than a polynomial increase with respect to the size of the graphs [3]. In order to further decrease the runtime, we take advantage of the other reason we use GEDLib, that being that it's implemented in C++, enabeling its compilation with O3 optimization. Still with all this effort, computing the graph edit distances fast is the biggest challenge, since the theory it relies on itself contains the main problem, and not its actual implementation circumstances.

Lastly, there is the already broached problem with user defined edit costs.
`compute_geds()` invokes `run_method()` n^2 times, which then, depending on the heuristic, constructs an LSAPE instance, whose optimal solution is then obtained by solving a minimization problem over all feasable solutions - that being graph edit paths - which then again includes obtaining the cost of each edit path, which is doable in $O(\max\{|E_G|, |E_H|\})$ time. That means the cost functions are called at least that often within the process [3].
Having expensive calls in such a cost function therefore extremely impacts the performance. If custom edit costs are used, not only do pybind calls have to happen inside the cost function, but in case of multi-threading, even the expensive GIL acquisition needs to happen within them, which itself is also a pybind call again.
Table 4.2 shows the effect. It compares the performance of clustering the same data (a subset taken from [19] again) multiple times, using the BRANCH method with UPGMA, whereas the custom edit costs are implemented like constant costs.

configuration in use	constant edit costs with 1 thread	constant edit costs with 10 threads	DiCoDa edit costs with 1 thread	DiCoDa edit costs with 10 threads	custom edit costs with 1 thread	custom edit costs with 10 threads
absolute time	37.981 sec	10.243 sec	1 min, 21.140 sec	21.486 sec	17 min, 13.589 sec	1 h, 10 min, 9.809 sec
relative time w.r.t. 1st entry	1	0.27	2.14	0.57	27.21	110.84

Table 4.2.: Edit costs benchmarks

Using DiCoDa edit costs takes roughly double the time, but this can be counteracted by allowing more threads. But trying to counteract the performance loss when using custom edit costs the same way just causes the opposite.
The massive performance loss is the reason not using custom edit costs is highly recommended at this point in time. Possible ways to deal with this problem are explored in chapter 6.

4.5.3. Robustness

The idea of making HGC as robust as possible is to resort to exceptions as rarely as possible. For that purpose, HGC follows the basic principle to assume every parameter to be empty at first.
One might have noticed that in the code examples, the default values set in the function definitions almost always were empty strings. Also, if any command line argument is omitted when running HGC via the terminal, it becomes an empty string. In most cases, the `main` module then simply passes the argument into the `HGCEnv` function without taking a closer look at it. That means, the user leaving out a command line argument, as well as the programmer leaving out a function argument, end up in the same spot: The parameter will be an empty string. Like this, HGC centralizes multiple causes of problematic usage, enabling a generalized error handling.
It then tries to make the best out of the empty parameters, using default values defined in the function implementation as often as possible. Only for the most basic parameters that can't be guessed, like directories to datasets that are to be imported, exceptions are thrown. A user leaving out a parameter is interpreted by "The user doesn't care about that option.", rather than "The user forgot to specify that option." The interface accepts thirteen different command line arguments, but already works with only two (`out` and `gml`).
Also, these exceptions are thrown as late as possible. Apart from only two, all other command line arguments are parsed in `HGCEnv` or later, enabling most of the error handling to also work for programmers that use `HGCEnv` directly. One of these two is also the `edit_costs` argument, that is in that case not set through a string argument anyway, but by a combination of multiple parameters as shown in Table 4.1.

4.6. Problems

The most problematic issue is still the performance loss with custom edit costs. If the GIL aquisition was moved out of the cost functions, allocating more threads would be able to increase the performance, just like with the other edit costs. The main problem here is that while the global Python interpreter is treated as a resource and can therefore not be shared and accessed simultaneously [10], creating an own interpreter for each thread is also not easily possible. Without rewriting GEDLib, there is no interface to manage the thread creation manually. The only public function that is available to run the GED method is `run_method()`, which itself creates the threads. So even moving its call into Python code to execute it in a new thread with a sub-interpreter allocated as the thread's resource doesn't solve the problem. This means it would have to be implemented in C++. But creating Python sub-interpreters in C++ is not supported by pybind11 in its current version [7].

Lastly, there is an issue regarding the symmetry of distance measures. Normally, it holds for every metric space that it contains a distance measure $d(A, B)$ that is symmetric, meaning $d(A, B) = d(B, A)$. In the case of GED, that distance measure is defined by an edit path from graph G to graph H, that is mapped onto costs using the cost functions. But those cost functions do not necessarily have to be symmetric. The exemplary implementation of the custom node relabel cost function in Listing 4.4 yields `node_rel_cost(1, 2)`$= 4 \neq 5 =$`node_rel_cost(2, 1)`.

```python
def node_rel_cost(label1, label2):
    return label1 * 2 + label2;
```

Listing 4.4: Asymmetric node relabeling function example

This results in the possiblity of an edit path from G to H that uses this function to have a cost lower than the cost of the same edit path reversed: $d(G, H) < d(H, G)$. If these edit paths turn out to represent the smallest distances with respect to the chosen heuristic, their costs are saved in the distance matrix in slots $a_{G,H}$ and $a_{H,G}$. In the end, one ends up with a non-symmetric distance matrix.
SciPy does not work with the raw distance matrix to generate the clustering in their `linkage()` function. Instead, the function uses a condensed form. The function `scipy.spatial.distance.pdist()` converts raw distance matrices into condensed distance matrices [28].
With a raw distance matrix $A \in \mathbb{Z}^{m \times m}$, a condensed distance matrix is defined as a vector $d \in \mathbb{Z}^n$, with $n = \binom{m+1}{2}$. $\binom{m+1}{2}$ denotes the mth triangle number, namely $m + (m-1) + (m-2) + ... + 1$. The vector entries are then defined in the following way: For all $i < j < m$, the entry $m \cdot i + j - \lfloor \frac{1}{2} \cdot ((i+2) \cdot (i+1)) \rfloor$ stores $a_{i,j}$ [28].
Like this, the entries are consecutively filled with the distance values for all possible combinations of indices i and j, whereas i is always lower than j. That means, only entries of the upper triangular matrix of A are actually queried. In symmetric distance matrices, this upper triangle contains all the necessary information but in non symmetric matrices, relevant values are truncated.

It is disputable how much non symmetric distances even make sense in the context of distance based hierarchical clustering, so HGC deals with this problem by simply minimizing the upper triangle of the generated distance matrix, always using the smaller of both candidate values.
Applying this kind of post-processing to the GED results also has the neat side effect of optimizing randomized algorithms like IPFP that can yield slightly different results

for $d(G,H)$ and $d(H,G)$ even when using symmetric edit cost functions. Still, this is mostly a temporary solution.

5. Conclusion

With hierarchical clustering being such a basic and generally applicable method to find structures and patterns in large data, it's not a particularily far-fetched idea to create a library designed to make it easily accessible. Doing this is always a question of specification though, since the existing packages that in fact do provide it are designed to work with every data type and every distance measure and leaves that choice and computation to the user. SciPy's linkage method does not provide "hierarchical graph clustering", it provides "hierarchical clustering". So combining hierarchical with a support for creating the input data for SciPy requires a choice of data type and a choice of distance measure to be made. While HGC does make this choice and therefore specifies the use case, it tries to do it in the most general way possible within that context. Coming from the fact that HGC is originally designed to be used in the context of bioinformatics, to be more precise on the DiCoDa data from [19] that represents network graphs, as well as network graphs being one of the most basic structures in computer science, the choice of data type was clear from the beginning. Again, HGC tries to stay as universal as possible within its restrictions and uses GED as its distance measure, for reasons already explained.
However, the GED computation is the part that takes the most runtime, creating the need for optimization. Fortunately, with the release of GEDLib and its heuristics, the possibility was created to provide this functionality in a performance-wise satisfactory way. Still, the part of the library that covers the distance measure by providing ways to compute it, turned out to present the biggest challenges, still partly unresolved.
Those problems mostly arise from the all present trade-off between usability and performance, in this case Python and C++, which shows itself most at the use of custom edit costs. Even when using the otherwise very efficient way to increase runtimes, multithreading, especially Python quickly reaches its limits as a programming language [16]. When working further on HGC, that trade-off is one of the most urgent problems to adress.

Apart from that, with HGC, one of the first tools to support hierarchical clustering of graphs to this extent was made and its current functionality is likely sufficient in most use cases.

6. Further Work

There are multiple takes on HGC's problems.
Regarding custom edit costs, the most obvious is to provide a way to implement custom cost functions in C++. But to keep the usability high, this would then mean to include its automatic compilation and linking to the rest of the C++ code, that should not also have to be recompiled, meaning they're present as shared object files at that point. This would mean a lot of usability overhead.
Another way is to follow the style of DiCoDa costs and enable loading edit costs from a distances dataset, independantly from omics data. At the moment, a workaround to accomplish this would be to specify a distances dataset upon loading an omics dataset that then only contains some dummy data.
But the main problem is already shown by the fact that at the moment, the program will run into a problem if a graph contains a node with a label that corresponds to a feature ID that is not contained in the distances dataset: This method of specifying edit costs works with an explicit cost matrix rather than continuous cost functions. All distances would have to be precomputed in order to fill that matrix beforehand.
Lastly, the probably most appealing solution is to provide more native C++ edit cost implementations like the ones GEDLib already does. They would then either be required to work with most label types, or the user should have the choice between those templates, and can select costs based on the label types that are in use.

Once pybind adds support for sub-interpreters and makes multi-threading helpful in this context, the problem can be at least partially avoided.

Another interesting addition would be to move the feature of specifying attributes per graph out of DiCoDa to be usable by the whole environment, enabling changable labels and therefore easier interpretable results for all graphs, rather than just for the graphs that were generated from CSV files by DiCoDa.

Finally, the exact performance increase of using GEDLib is still to be thoroughly tested. NetworkX includes a method to compute the graph edit distance in Python [23], so a program executing the GED computation with GEDLib and with NetworkX, running benchmarks to compare the runtimes would certainly yield interesting results.

In the end, this paper presents the first actual use of HGC by running it on the full data [19] that parts from were used multiple times throughout the examples, taking advantage of its support of DiCoDa-CSV data for which is was originally intended. The dendrograms of four runs with some basic parameters and "HSCT_responder" as the labeled attribute are attached at the end. Interpreting them and deciding to which extent they are helpful would be the most obvious further work.

List of Figures

List of Tables

Listings

Bibliography

[1] *biomedbigdata/dicoda: DiCoDA: A C++ Library with Python Bindings for Differential Compositional Data Analysis*. URL: `https://github.com/biomedbigdata/dicoda` (visited on 09/08/2021).

[2] M. Blaxter, J. Mann, T. Chapman, F. Thomas, C. Whitton, R. Floyd, and E. Abebe. "Defining operational taxonomic units using DNA barcode data." In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 360(1462).1935-1943 (2005).

[3] D. B. Blumenthal, N. Boria, J. Gamper, S. Bougleux, and L. Brun. "Comparing heuristics for graph edit distance computation." In: *The VLDB journal* 29(1).419-458 (2020).

[4] D. B. Blumenthal, S. Bougleux, J. Gamper, and L. Brun. "GEDLIB: a C++ library for graph edit distance computation." In: *In International Workshop on Graph-Based Representations in Pattern Recognition* (2019), pp. 14–24.

[5] S. Clémençon, H. De Arazoza, F. Rossi, and V. C. Tran. "Hierarchical clustering for graph visualization." In: *arXiv preprint arXiv:1210.5693* (2012).

[6] *dbblumenthal/gedlib: An easily extensible C++ library for (suboptimally) computing the graph edit distance between attributed graphs.* URL: `https://github.com/dbblumenthal/gedlib` (visited on 09/08/2021).

[7] *Embedding the interpreter - pybind11 documentation*. URL: `https://pybind11.readthedocs.io/en/stable/advanced/embedding.html#sub-interpreter-support` (visited on 09/08/2021).

[8] *Functions - C++ reference*. URL: `https://en.cppreference.com/w/cpp/language/functions` (visited on 09/08/2021).

[9] *Geography Markup Language | OGC*. URL: `https://www.ogc.org/standards/gml` (visited on 09/08/2021).

[10] *GlobalInterpreterLock - Python Wiki*. URL: `https://wiki.python.org/moin/GlobalInterpreterLock` (visited on 09/08/2021).

[11] *GML - NetworkX 2.6.2 documentation*. URL: `https://networkx.org/documentation/stable/reference/readwrite/gml.html` (visited on 09/08/2021).

[12] *GXL - Graph eXchange Language*. URL: `https://userpages.uni-koblenz.de/~ist/GXL/index.php` (visited on 09/08/2021).

[13] *Introduction - NetworkX 2.6.2 documentation*. URL: `https://networkx.org/documentation/stable/reference/introduction.html#graphs` (visited on 09/08/2021).

[14] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Vol. 344. John Wiley & Sons, 2009.

[15] K. Kumar and S. Dahiya. "Programming languages: A survey." In: *International Journal on Recent and Innovation Trends in Computing and Communication* 5(5).307-313 (2017).

[16] A. Marowka. "On parallel software engineering education using python." In: *Education and Information Technologies* 23(1).357-372 (2018).

[17] *Matplotlib: Python plotting - Matplotlib 3.4.3 documentation*. URL: `https://matplotlib.org/` (visited on 09/08/2021).

[18] *matplotlib.pyplot - Matplotlib 3.4.3 documentation*. URL: `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html` (visited on 09/08/2021).

[19] A. Metwaly, A. Dunkel, N. Waldschmitt, A. C. D. Raj, I. Lagkouvardos, A. M. Corraliza, ..., and D. Haller. "Integrated microbiota and metabolite profiles link Crohn's disease to sulfur metabolism." In: *Nature communications* 11(1).1-15 (2020).

[20] D. Müllner. "Modern hierarchical, agglomerative clustering algorithms." In: *arXiv preprint arXiv:1109.2378* (2011).

[21] F. Murtagh and P. Contreras. "Algorithms for hierarchical clustering: an overview." In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2(1).86-97 (2012).

[22] *NetworkX - NetworkX documentation*. URL: `https://networkx.org/` (visited on 09/08/2021).

[23] *networkx.algorithms.similarity.graph_edit_distance - NetworkX 2.6.2 documentation*. URL: `https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.similarity.graph%5C_edit%5C_distance.html` (visited on 09/08/2021).

[24] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. "Energy efficiency across programming languages: how do energy, time, and memory relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. 2017, pp. 256–267.

[25] *pybind/pybind11: Seamless operability between C++11 and Python*. URL: `https://github.com/pybind/pybind11` (visited on 09/08/2021).

[26] *scipy.cluster.hierarchy.linkage - SciPy v1.7.1 Manual*. URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html` (visited on 09/08/2021).

[27] *SciPy.org - SciPy.org*. URL: `https://www.scipy.org/` (visited on 09/08/2021).

[28] *scipy.spatial.distance.pdist - SciPy v1.7.1 Manual*. URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html` (visited on 09/08/2021).

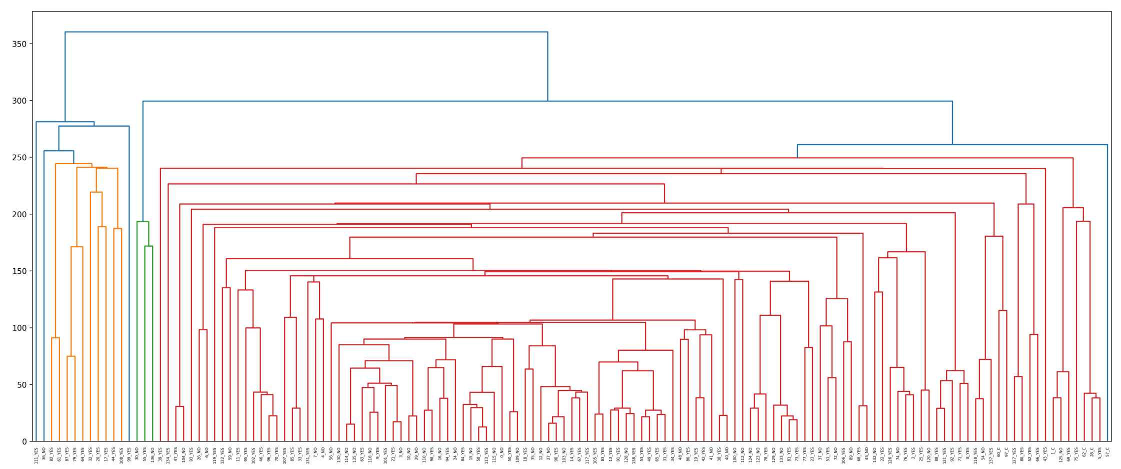
[29] R. Tayal. *Python's GIL — A Hurdle to Multithreaded Program*. `https://medium.com/python-features/pythons-gil-a-hurdle-to-multithreaded-program-d04ad9c1a63`. 2019.

[30] *UML Class Diagrams - Graphical Notation Reference*. URL: `https://www.uml-diagrams.org/class-reference.html` (visited on 09/08/2021).

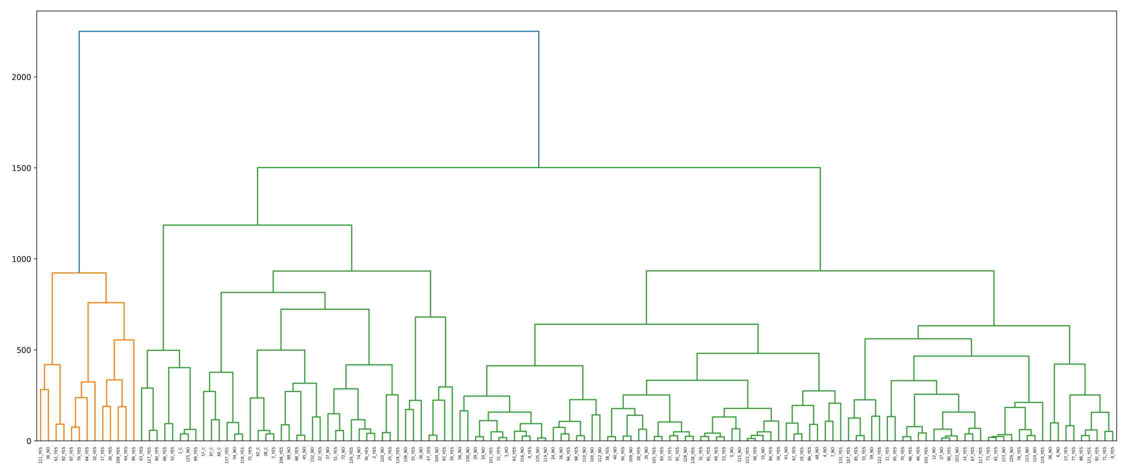
[31] *UML Sequence Diagrams - Graphical Notation Reference*. URL: `https://www.uml-diagrams.org/sequence-diagrams-reference.html` (visited on 09/08/2021).

[32] Z. Zou, K. Hua, and X. Zhang. "HGC: fast hierarchical clustering for large-scale single-cell data." In: *bioRxiv* (2021).

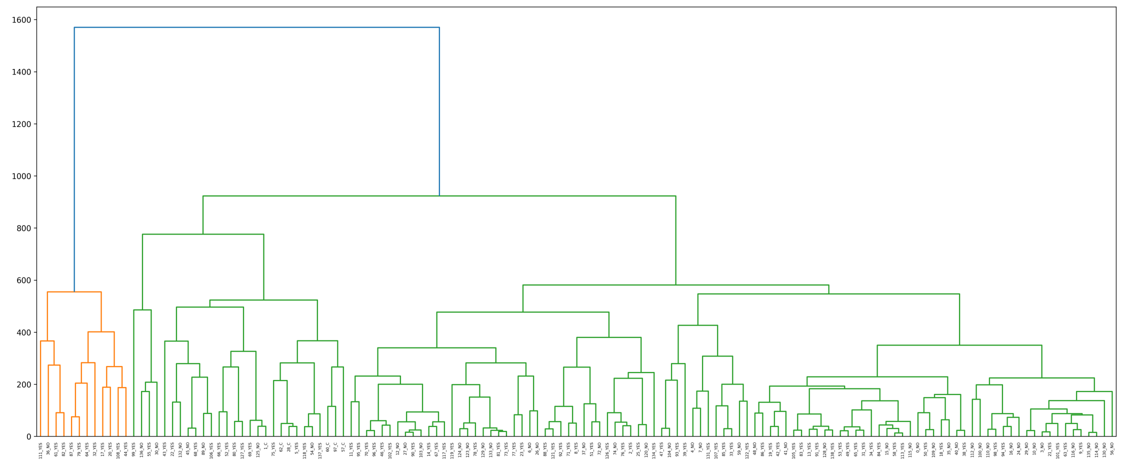
A. Results

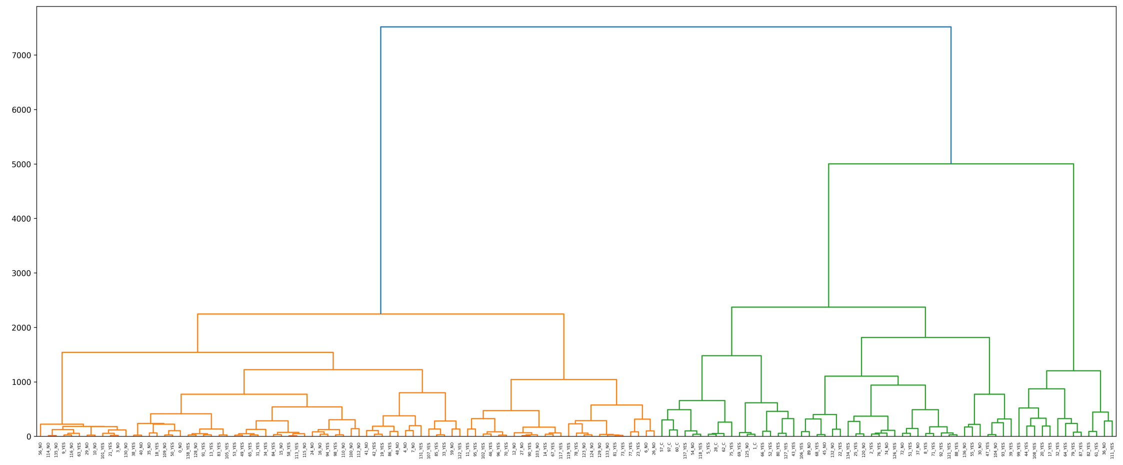


clustering algorithm: nearest point, GED method: branch, edit costs: DiCoDa distances



clustering algorithm: farthest point, GED method: branch, edit costs: DiCoDa distances


clustering algorithm: UPGMA, GED method: branch, edit costs: DiCoDa distances


clustering algorithm: incremental, GED method: branch, edit costs: DiCoDa distances