INTELCOMP PROJECT

A COMPETITIVE INTELLIGENCE CLOUD/HPC PLATFORM FOR AI-BASED STI POLICY MAKING

(GRANT AGREEMENT NUMBER 101004870)

D3.7. Graph Analysis Toolbox

Deliverable information	
Deliverable number and name	D3.7. Graph Analysis Toolbox
Due date	Dec 31, 2022
Delivery date	Mar 31, 2023
Work Package	WP3
Lead Partner for deliverable	Universidad Carlos III de Madrid
Author	Jesús Cid-Sueiro
Reviewers	Aurélien Leynet (HCERES) George Kakaletris (CITE)
Approved by	Jerónimo Arenas García (UC3M)
Dissemination level	Public
Version	1.0

Document revision history

Issue Date	Version	Comments
Feb 28, 2023	0.1	Submitted to internal review
March 08, 2023	0.2	Revised according to reviewers' comments.
March 31, 2023	1.0	Version ready for submission

DISCLAIMER

This document contains a description of the **IntelComp** project findings, work and products. Certain parts of it might be under partner Intellectual Property Right (IPR) rules so, prior to using its content please contact the consortium coordinator for approval.

In case you believe that this document harms in any way IPR held by you as a person or as a representative of an entity, please do notify us immediately.

The authors of this document have taken any available measure in order for its content to be accurate, consistent and lawful. However, neither the project consortium as a whole nor the individual partners that implicitly or explicitly participated in the creation and publication of this document hold any sort of responsibility that might occur as a result of using its content.

The content of this publication is the sole responsibility of **IntelComp** consortium and can in no way be taken to reflect the views of the European Union.

The European Union is established in accordance with the Treaty on European Union (Maastricht). There are currently 27 Member States of the Union. It is based on the European Communities and the member states cooperation in the fields of Common Foreign and Security Policy and Justice and Home Affairs. The five main institutions of the European Union are the European Parliament, the Council of Ministers, the European Commission, the Court of Justice and the Court of Auditors.

(http://europa.eu.int/)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101004870.

CONTENTS

FIGURES

TABLES

ACRONYMS

BC	Bhattacharyya Coefficient
CSV	Comma Separated Values
EWB	Evaluation WorkBench
GEXF	Graph Exchange XML Format
GPU	Graphic Processing Unit
IMT	Interactive Model Trainer
LDA	Latent Dirichlet Allocation
NLP	Natural Language Processing
STI	Science, Technology, and Innovation

EXECUTIVE SUMMARY

The Graph Analysis toolbox is a collection of software components for the computation, processing and analysis of graph collections. It is specifically oriented to graphs associated with the research production (publications, patents, project proposals, etc.) of a community or an organization, or their agents (authors, organizations, etc.), but it can also be used for other purposes.

The graph tool relies on powerful Python libraries for data analysis and graph processing (iGraph, NetworkX, Scikit-learn, fa2, etc.), but also includes some *ad hoc* implementations for the synthesis of similarity graphs that have turned out to be more efficient than other available options.

The key modules of the toolbox and their respective most relevant characteristics are as follows:

1. Data import modules, from files or SQL / Neo4J databases.
2. Graph processing module, which includes functionality for:
    a. Graph creation.
    b. Graph editing.
    c. Generation of similarity graphs.
    d. Application of community detection algorithms.
    e. Extraction of local parameters of the nodes of a graph based on centrality measures.
    f. Comparison of community structures.
    g. Calculation of community metrics.
    h. Computation and visualization of graph layout.
3. Module for managing and processing collections of graphs (that we name *supergraphs*), which includes functionality for:
    a. Editing the collection.
    b. Memory efficient management of the collection at runtime: activation and deactivation of supernodes and superlinks.
    c. Extraction of information (metadata) from supernodes and superlinks.
    d. Inference of new graphs (supernodes and superlinks) by:
        i. Subsampling.
        ii. Conversion of node attributes into new nodes.
        iii. Graph transduction (inference of a similarity graph based on a bipartite auxiliary graph that connects the nodes of the target graph with the nodes of another similarity graph), that can be used, for instance, to generate a graph of similarities between authors based on their research products (articles, patents).
        iv. Transitivity, which allows, for example, the generation of a bipartite graph: documents-organizations from two bipartite graphs documents-authors and authors-organizations.
    e. Comparison of graphs, using cosine distance type metrics.
4. Graph export modules to save graphs in different formats.

5. Module for direct visualization of bipartite graphs based on the Halo library.

Likewise, the toolbox includes several Python scripts that facilitate the use of all the software functionalities, by a non-expert user, through a terminal-based menu utility.

The graph toolbox can be applied to different types of data sources. In the context of IntelComp, the following are some examples:

- Graphs of relationships between research results:
  - Directed graphs of bibliographic citations between scientific documents, or undirected graphs of co-citations.
  - Undirected thematic graphs of research results: these are graphs connecting research papers, projects, patents, etc. They are built by metrics based on document embeddings or topic models.
- Collaboration graphs between agents:
  - From collaboration in projects or co-authorship of articles/patents.
  - From articles that include acknowledgments funding projects or institutions.
- Bipartite graphs of linkage relations:
  - Between the results of the research and its authors.
  - Between authors and their organizations.

The combination of all these graphs allows, in turn, to build new derived graphs that provide additional information for the analysis. For example, thematic relationships between the activity of researchers can be inferred from the thematic relationship between their publications and the authorship relationships between researchers and publications.

The toolbox facilitates the generation, management, and processing of all these related graphs efficiently, both in terms of the calculation and storage of the graphs themselves, and in terms of the implementation of algorithms:

- Inference of similarity graphs.
- Detection of communities.
- Inference of graphs by combining other graphs (transitive graphs, graph transduction).
- Extraction of statistical information from the graph.
- Characterization of graph nodes: centrality parameters, PageRank.
- Calculation of connected components.
- Calculation of layouts for visualization.

The results of the analysis can be stored in formats that facilitate their visualization through applications such as the IntelComp-native graph visualization component developed by WP4, and other external applications like Gephi or software modules like Halo.

1. INTRODUCTION

The Graph Analysis toolbox[1] is a software library for the computation, processing and analysis of graph collections. It is specifically oriented to graphs associated with the research production (publications, patents, project proposals, etc.) of a community or an organization, or their agents (authors, organizations, etc.), but it can be used for other general purposes.

This tool relies on powerful Python libraries for data analysis and graph processing (iGraph, NetworkX, Scikit-learn, Fa2, etc.), but also includes some *ad hoc* implementations for the synthesis of similarity graphs that have turned out to be more efficient than other available options.

The work carried out for this deliverable implies, on the one hand, developing the software components that provide the necessary functionality for the treatment of graphs and, on the other hand, applying the toolbox to the data sources of the project, also outlined.

1.1. General view of the Architecture

The general scheme of the architecture is shown in Figure 1, and its main components are the following:

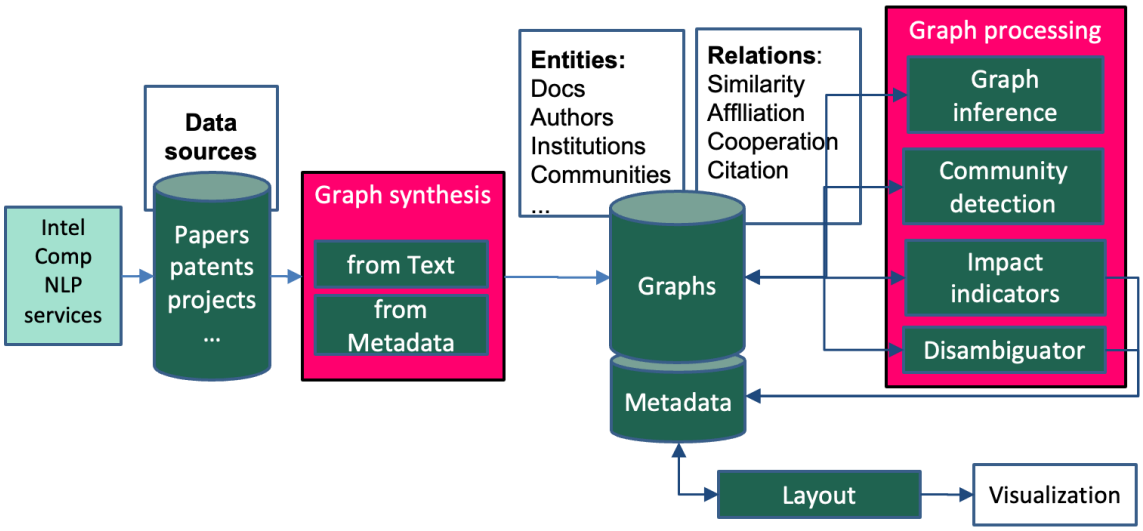

Figure 1: General Structure of the Graph Analysis Toolbox

1. **Data sources**: the graph tool is specifically oriented to graphs associated with the research production. In general, the data sources are expected to contain collections of documents (publications, patents, project proposals, etc.) with associated metadata (authors, organizations, citations, etc.).
2. **NLP services**. The natural language processing (NLP) services enrich the text corpora with the embedding of the documents in a vector space (bag of words, BM25, topic models, transformers, etc.). This layer is not part of the Graph Analysis toolbox: the

[1] https://github.com/IntelCompH2020/GraphAnalysisToolbox

document embeddings will be provided by the NLP tools implemented in WP3 tasks T3.1 and T3.5.

3. **Graph synthesis**: A collection of methods to compute different types of graphs based on document embeddings or metadata (citations, authors, organizations, etc.).
4. **Graph processing**: a collection of modules for the processing of existing graphs: three classes of modules have been implemented:
    a. Graph inference: used to generate derivative graphs based on the processing of another graph or the combination of several graphs.
    b. Community detection: clustering algorithms over graphs, used to unveil the internal structure of the graphs.
    c. Impact indicators: a collection of metrics to evaluate the relevance or the role of individual nodes in the graph.
    d. Disambiguator: a module used to discriminate between different entities (e.g. authors) with the same name, or associate an entity (e.g., authors) to all its representations (e.g., names).
5. **Graphs**: a database of graphs generated by the platform, including the nodes, edges and attributes of each graph, and metadata information about the relationships between all graphs.
6. **Layout**: algorithms for deploying graphs on a two-dimensional space through the visualization module
7. **Visualization**: methods for graph rendering included in the tool, though the visualization task over the integrated web UI of IntelComp is carried out by specific components of the STI Graph Visualizer, implemented in task T4.3 of the project.

1.2. General view of the functionality

We provide here a structured list, including a concise description, of all the functionalities that are included in the toolbox:

1. Import data from files or SQL / Neo4J databases, for the generation or the enrichment of graphs.
2. Generation and processing of graphs:
    a. **Creation** of graphs.
    b. Graph **editing** (single node/link edits, attribute management, node merging, link filtering, graph sampling, node ordering, etc.).
    c. Generation of **similarity graphs** from nodes and vector-space representations (embeddings, transformers, topic models, bag-of-word models, etc.), or from nodes and sets of attributes (keywords, citations, etc.). Several similarity measures, including measures based on L1, L2, Jensen-Shannon, Hellinger or cosine distances, with options for GPU execution, have been implemented.
    d. Application of **community detection algorithms**: Louvain, FastGreedy, Walktrap, InfoMap, label propagation, spectral clustering and Leiden.
    e. **Extraction of local parameters** of the nodes of a graph: degree, eigenvector centrality, betweenness, pageRank, clustering coefficient, closeness, Katz centrality.

   f. **Comparison of community structures**, using different metrics: information variation, normalized mutual information, Rand index, adjusted Rand index, split-join distance, asymmetric split-join distance.
   g. Calculation of **community metrics**: coverage, performance and modularity.
   h. Generation of **graph layouts**, based on the Force Atlas 2 and Fruchterman-Reingold algorithms.
3. Creation, management and processing structured collections of graphs (named *supergraphs*), which includes functionality for:
   a. **Editing of the collection**: adding and removing generic graphs and bipartite graphs.
   b. **Memory efficient management** of the collection at runtime: activation and deactivation of graphs and bipartite graphs.
   c. **Extraction of information** (metadata) from graphs and bipartite graphs.
   d. **Inference of new graphs** by:
      i. **Subsampling:** this includes methods for computing subgraphs by random sampling of nodes or filtering nodes by attribute values.
      ii. Generation of **bipartite graphs from attributes**: conversion of node attributes into new nodes (generating a new bipartite graph connecting the original nodes with their attributes).
      iii. **Graph transduction**, which consists of the generation of a similarity graph using a bipartite graph that connects the nodes with the nodes of another similarity graph. This is useful, for example, to generate a graph of similarities between authors based on the similarity between their research products (articles, patents).
      iv. **Transitivity**, which consists of the generation of a bipartite graph A-C, connecting supernodes A and C, by concatenation of two bipartite graphs A-B and B-C, connecting A and C with an intermediate supernode C. For example, the generation of a bipartite graph: documents-organizations from two bipartite graphs documents- authors and authors-organizations.
   e. **Comparison of graphs**, using cosine distance type metrics.
4. **Export of graphs**:
   a. Export to Neo4J databases.
   b. Export to files in CSV formats (importable from Gephi[2]).
   c. Export to GEXF (the standard Gephi format).
5. **Visualization** of graphs and bipartite graphs based on the NetworkX and Halo libraries.

Likewise, the toolbox includes a Python program that allows a non-expert user to access all the functionality through a menu system.

The software has a modular structure, which facilitates its future expansion with further functionality.

[2] https://gephi.org/

1.3. Basic definitions and notation

To read this document, no major background knowledge on graphs is required. However, to avoid an ambiguous use of terms and clarify the mathematical notation, we start with some basic definitions.

A *graph G* is a tuple (*S*, *E*, ***W***), where *S* is an ordered set of *nodes*, *E* is a set of *edges* (also named *links*) and **W** is the *weight matrix* (also named *adjacency matrix*). Each edge is an ordered pair of nodes from *S*. Weight w_{ij} from **W** is the weight of the edge connecting the *i*-th and the *j*-th nodes from *S*. If the weights of all edges are unity, the graph is said to be *unweighted*.

An *undirected* graph is a graph whose edges have no orientation: edge (*u*, *v*) in an undirected graph is equivalent to edge (*v*, *u*). Otherwise, the graph is said to be *directed*, and the nodes from each edge (*u*, *v*) are named *source* and *target* (or *destination*) nodes, respectively.

A *bipartite* graph *G* is a graph whose edges connect nodes from different sets. They can be represented as tuples (*S*, *T*, *E*, $\mathbf{W}_{ST}$) where *S* are the source nodes of the edge, *T* are the target nodes, *E* is the list of edges, and $\mathbf{W}_{ST}$ is the weight matrix.

2. DATA STRUCTURES

In this and the following sections, we provide a general description of the main data structures and python modules that provide the whole functionality.

2.1. Data sources

The input data sources must contain all the information required for the generation of the graphs. This data can be stored in different formats. A Data Manager module oversees the transport of the input corpora into the toolbox. It includes modules for the data import from csv files, SQL databases or Neo4J graph databases.

2.2. Output data

The output data of the toolbox is a collection of graphs and their associated metadata. The graph data contains information about the nodes, node attributes, edges, edge weights, and metadata with information about the graph generation (graph synthesis algorithm, number of nodes and edges, list of node and edge attributes and information about the processing algorithms applied over the graph).

Since some graphs can be generated from other graphs, the collection of graphs is also organized in another graph structure named **supergraph** in the toolbox.

In the context of this toolbox, a supergraph is a structured collection of graphs. It consists of supernodes and superedges, where each supernode is itself a graph, and each superedge is a bipartite graph connecting nodes from a supernode to nodes of another supernode. This is illustrated in Figure 2.

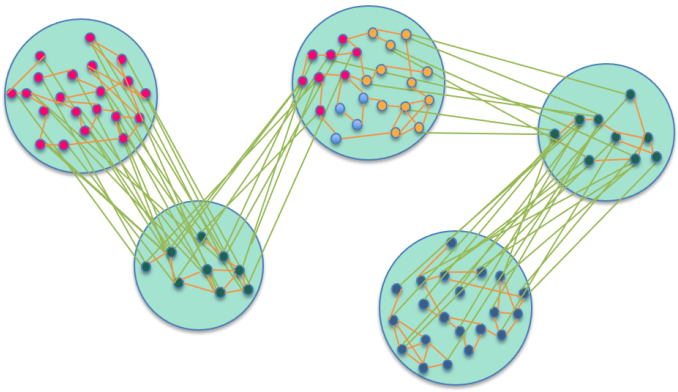

Figure 2: A supergraph is a structure of graphs. It consists of supernodes and superedges, where each supernode is itself a graph, and each superedge is a bipartite graph. The figure shows a supergraph with 5 supernodes and 4 superedges.

The relations defined by each graph in the structure may be of different nature. As an example, Figure 3 shows a supergraph structure linking (Pu)blications (supernode Pu), (Pa)tents and (Pr)ojects through similarity graphs. Publications are connected to (A)uthors through a bipartite directed graph and projects to (O)rganizations through an affiliation graph. The authors are connected through a cooperation graph, and patents and publications are also connected to themselves through a directed citation graph. The supergraph structure is a multigraph: some superedges my represent directed graphs, others undirected. Furthermore, any pair of nodes can be connected through several superedges, each one accounting for a different type of relation.

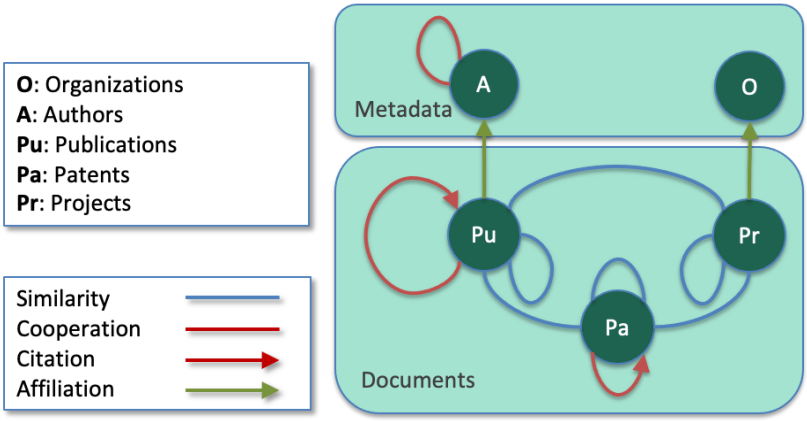

Figure 3: A supergraph structure connecting scientific documents and some of their agents (researchers and organizations). Note that each node represents a graph (supernode) and each edge represents a bipartite graph (superedge). Each supernode or superedge may

The graph toolbox facilitates the successive generation of the supergraphs components starting from the data sources, which are used to generate primary graphs, and generating secondary graphs as a result of the processing and analysis of the primary graphs. The difference between primary and secondary graphs is explained in the next subsection.

2.3. Entities

IntelComp's natural data sources for the generation of graphs are corpora of documents (patents, publications, etc.). Thus, the nodes of the primary graphs generated from the data sources will represent documents (typically). However, the graph toolbox facilitates the generation of secondary graphs whose nodes may represent other kinds of entities: authors, organization, funding institutions, keywords, etc. Therefore, we will use the generic term "entity" whenever the description can be applicable to arbitrary graphs, no matter what their nodes are representing.

3. FUNCTIONALITY

In this section we describe in more detail the whole functionality of the software. Part of the functionality is provided through external libraries for graph processing, and other has been developed natively in the project in search of efficiency or a better adaptability to our data structure. The main graph processing methods and the external libraries used in the graph will be summarized later, in Section 3.4 and Table 1.

3.1. Generation of Graphs

3.1.1. Generation of Similarity Graphs

A similarity graph is a graph whose edge weights represent some similarity measure between the corresponding nodes. We will use the term "similarity matrix" to refer to the adjacency matrix of a similarity graph.

Similarity graphs can be constructed from any collection of nodes (representing documents or any other entities), provided that a similarity measure between nodes can be computed. The generation of similarity graphs of documents based on their embedding representations is an important example. A symbolic representation of this process is shown in Figure 4

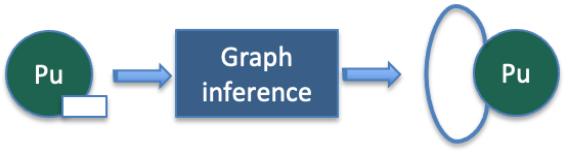

Figure 4: Generation of similarity graph from embeddings. Starting from a collection of nodes, where each node represents a document or any other entity, and each document or entity is represented by a vector in some embedding space (e.g., by means of a transfor

There are several challenges in the construction of a similarity graph:

1. Defining a similarity measure.
2. Setting the similarity threshold to keep just the most relevant edges.
3. Computing the graph for a large number of nodes.

3.1.1.1. Similarity measures

Similarity measures from node embeddings

Assume that the entities represented by the nodes *n* and *m* have a vector representation in some embedding space, **x** and **y**, respectively. The similarity measure is a function *s*(**x**, **y**) mapping pairs of vectors into a number in [0, 1].

We have implemented two similarity metrics:

- Cosine similarity: $s(x,y) = \frac{x^T y}{||x|| \cdot ||y||}$
- Bhattacharyya Coefficient (BC): $s(x,y) = \sqrt{x}^T \sqrt{y}$

The cosine similarity is defined for any vector representation. It has been used to compute document similarity graphs from embeddings based on transformers.

The BC measure requires probabilistic embeddings: vectors **x** and **y** must have components in [0, 1] summing up to one. The square roots of the vectors are computed component-wise. It has been used to compute document similarities from embeddings based on topic models.

Similarities based on distances.

An alternative to the direct computation of a similarity measure is to transform a distance between embeddings into a similarity metric. Three transformations have been included in the graph tool:

- Linear: $s(x,y) = 1 - \frac{d(x,y)}{D_{max}}$
- Polynomial: $s(x,y) = 1 - \left(\frac{d(x,y)}{D_{max}}\right)^a$
- Exponential: $s(x,y) = exp\left(-a \cdot d(x,y)\right)$

where D_{max} is the maximum distance between embeddings and $a > 0$ is a free parameter. The linear transformation is useful when the distance between embeddings can be bounded by an upper limit (this is usually the case, for instance, for probabilistic embeddings). The exponential transformation is useful because it translates the triangular inequality of the distance metric into a similarity bound: for any embedding, **z**,

$$s(x,y) \geq s(x,z) \cdot s(z,y)$$

Thus, we can lower-bound the similarity between two nodes in the graph that have no direct edge by multiplying the similarities of paths connecting them.

The graph tool implements the following distances:

- For arbitrary embeddings:
  - Euclidean: $d(x,y) = \left|\left|x-y\right|\right|^2$
  - L1 (absolute difference): $d(x,y) = \left|\left|x-y\right|\right|_1$
- For probabilistic embeddings:

- Jensen-Shannon: $d(x,y) = \frac{1}{2}\left(x^T \cdot log_2\left(\frac{2x}{x+y}\right) + y^T \cdot log_2\left(\frac{2y}{x+y}\right)\right)$
- Hellinger: $d(x,y) = \sqrt{1 - \sqrt{x}^T\sqrt{y}}$

In the expressions above, square roots, logs or fractions of vectors are computed component-wise.

Similarities based on categorical attributes.

A similarity measure has been defined for situations where the entities are characterized by a set of elements or categories. For instance, if nodes *n* and *m* are represented by sets X and Y, respectively, the following similarity measure can be used:

- Categorical cosine similarity: $s(X,Y) = \frac{|X \cap Y|}{\sqrt{|X||Y|}}$

where $|\cdot|$ represents the cardinality. This is named "categorical cosine similarity" because it is equivalent to the cosine similarity of the binary representation of the category sets. This similarity has been used because it allows a more efficient implementation than other measures, like the Jaccard similarity.

This similarity metric can be used for very different purposes, depending on the nature of the nodes and the type of sets. For instance:

- Citation graph of publications: X is the set of papers related to a document (cited papers, citing paper or co-citation, for example, depending on the type of citation graph).
- Collaboration graph of authors: X is the set of papers from an author.
- Collaboration graph of papers: X is the set of authors from a paper.
- Thematic graph of projects: X is the set of keywords associated with a project.

3.1.1.2. Similarity threshold

The main goal of the similarity graphs is to identify the most similar nodes in the graph and also to apply graph processing algorithms to unveil the internal structure of relations between the nodes. For this reason, we are mainly interested in the highest similarity values, and not on the small similarity relations. Thus, the weight matrix of the graph will be sparse, and only pairs of nodes with a similarity above a threshold will be connected through an edge.

The graph toolbox provides two ways to set the sparsity of the graph:

- Absolute threshold: the user sets a fixed threshold as a free parameter.
- Target number of edges per node: the user sets the target average number of edges per node and the threshold is computed in such a way that this target is satisfied.

In general, setting the absolute threshold is difficult, because it may depend on the similarity measure, the embedding and the corpus. Controlling the sparsity through the target number of edges is in general more practical.

3.1.1.3. Graph computation

In general, the computation of the similarity graph from a matrix of *m* embeddings with *d* dimensions requires $O(m^2 d)$ computations. This can state some scalability problems for large *m*. To alleviate this problem, the software includes two options:

- For similarities based on distance metrics, the similarity graph can be obtained without computing all pairs of distances between nodes through special structures like *balltrees* [1]. We have used the implementations of these methods from the sklearn.neighbors module from the python library *Scikit-learn*.
- For similarities that can be computed from cross products (BC, cosine distance), a high degree of parallelization is possible. Efficient methods have been implemented based on the use of GPU processing.

  For instance, given an input matrix of embeddings, X, the BC similarity matrix can be computed as

  $$S = \phi_t\left(\sqrt{X}^T \sqrt{X}\right)$$

  where square roots are computed component-wise, and ϕ_t is the threshold function with threshold t (that is $\phi_t(x) = x \cdot (x \geq t)$, which is applied component-wise. The matrix product is computed by blocks through a GPU unit, so that there is no need to save in memory the whole product before thresholding.

3.1.1.4. Equivalent graphs

When different nodes have identical embedding representation, they form clusters of fully connected nodes, and all nodes in a cluster are connected (by identical edges) to the same neighbors in the graph. Thus, we can get a more compact representation of the graph by collapsing all nodes of each cluster into a single node representing all of them. We call this collapsed graph the *equivalent graph*.

These clusters of nodes are frequent when nodes are represented by categorical attributes from a reduced set of categories. It may also arise from topic models based on the Latent Dirichlet Allocation (LDA) algorithm, when the hyperparameters are chosen to get highly sparse document-topic matrices.

Equivalent graphs are useful to reduce the cost of the graph computation. In general, for a graph with an average cluster size of *m* nodes, the cost of computing the similarity graph reduces by a factor of m^2.

3.1.1.5. Bipartite graphs

The module for generating similarity graphs can be applied to the generation of bipartite similarity graphs. Figure 5 illustrates the process with the generation of a similarity graph between publications and patents. It is required from both sources of nodes to have vector representations in the same embedding space.

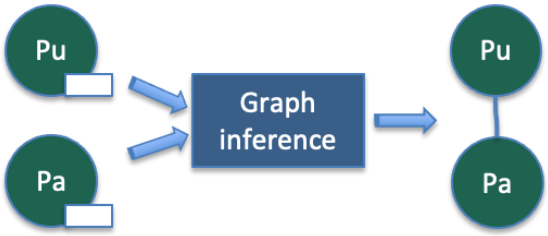

Figure 5: Generation of a similarity bipartite graph connecting patents and publications that are similar, according to some configurable similarity measure.

3.1.2. Secondary graphs

When the data sources for the graph generation contain tables of entities and attributes (categorical data or vector representations in an embedding space), the graph toolbox can construct primary graphs based on similarity. In addition, it includes methods for the construction of secondary graphs, based on the primary graphs and / or attributes from the source datasets. In this section, we describe the basic method to construct secondary graphs, and how they can be combined.

3.1.2.1. Bipartite graphs from attributes

Any categorical attribute from the nodes of a graph can be used to construct a bipartite graph connecting each node with its corresponding attribute values. As an example, if a table of authors contains an attribute with their respective organizations, we can construct a bipartite author-affiliation graph connecting each author to their organizations. This is illustrated in Figure 6.

Figure 6: Generation of a new bipartite graph from attributes. Initially, a graph of (A)uthors exists, where each node represents a researcher, and a node attribute contains the organization (research institution) associated with the researcher. The attribute

Some examples of the graphs that have been constructed by this method:

- Documents – authors.
- Authors – organizations.
- Documents – keywords.
- Documents – domain labels obtained from the domain classifier or the taxonomic classifier (see deliverables D3.3 and D3.4).
- Nodes – community labels. This is a graph connecting each node to its community, according to the assignment produced by a community detection algorithm.
- Patents – papers (based on citations).

The generation of these bipartite graphs is the primary step for the generation of transductive graphs.

3.1.2.2. Transductive graphs

A transductive graph is a graph computed for some entities based on their relations with other entities from another graph. A transductive graph is useful, for example, to construct a graph of entities when no embedding representation is available. It has been used to generate similarity graphs of authors based on similarity graphs of papers, as illustrated in Figure 7.

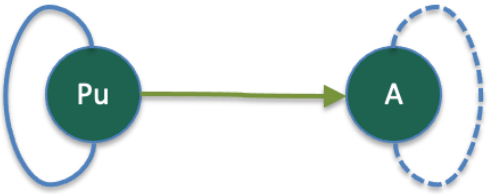

Figure 7: Example of an Author transductive graph. Starting from a similarity graph of papers and a bipartite graph connecting each paper to its authors, an author similarity graph is inferred.

Given two sets of nodes, S and T, a graph G_S with nodes in S and weight matrix W_S and an bipartite graph G_{ST} connecting the nodes in S to those in T, with weight matrix $\mathbf{W}_{ST}$, the weight matrix W_T of the transductive graph G_T is computed as:

$$W_T = W_{ST}^T \cdot W_S \cdot W_{ST}$$

According to this, the similarity between two nodes m and n from T is equal to a weighted sum of all paths going from m to n through 1 or 2 nodes from S. The weight of a path is the product of all its edge weights. This can be normalized to make sure that all self-edges in the output graph G_T have weight 1:

$$D = \sqrt{diag(W_T)}^{-1}$$

$$\underline{T} = D \cdot W_T \cdot D$$

3.1.2.3. Transitive graphs

Transitive graphs are useful to relate two sets of nodes based on a common attribute. This is illustrated in Figure 8, where a bipartite graph connecting publications to their authors is combined with another bipartite graph connecting authors to their current organizations to infer a graph connecting publications to the current organization of their authors[3].

Given three sets of nodes S, T and U and the bipartite graphs G_{ST} and G_{TU} connecting S to T and T to U, with weight matrices $\mathbf{W}_{ST}$ and $\mathbf{W}_{ST}$, respectively, the weight matrix of the transitive graph G_{SU} connecting S to U is computed as

$$W_{SU} = W_{ST} \cdot W_{TU}$$

[3] Note that the current organization of the author could differ from the affiliation of the author in the paper. We ignore this circumstance here for illustration purposes.

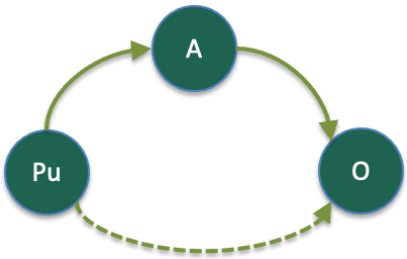

Figure 8: Transitive graph: the concatenation of a bipartite graph papers-authors and an affiliation graph authors-organizations, a bipartite graph associating papers to organizations is derived.

Thus, the similarity between node m from S and node n from T is computed as the sum of the weights of all paths going from m to n through any node in T.

3.2. Graph Analysis

3.2.1. Community Detection Algorithms

Community detection algorithms provide a partition of the set of nodes in the graph, providing useful information on the overall structure of the graph. The Community Detection module includes several open-source implementations of graph clustering and community detection algorithms, as well as some methods for further analysis of the graph partitions.

The clustering or community detection algorithms included in the Graph Analysis toolbox are the following:

1. Connected components[4]
2. Spectral clustering
3. Louvain
4. FastGreedy
5. WalkTrap
6. Info Map
7. Label Propagation
8. Leiden

Most algorithm implementations are based on the iGraph[5] python library, except for spectral clustering, which is based on Scikit-learn[6], and Louvain, which is based on the module "communities" of NetworkX[7].

[4] *Connected components are not usually interpreted as "communities", but they can be taken as a baseline. All the communities generated by the other algorithms are subsets of a connected component.*

[5] https://python.igraph.org

[6] https://Scikit-learn.org

[7] https://NetworkX.org/

In order to facilitate the analysis and evaluation of the graph partitions obtained by community detection algorithms (or by the same algorithm over two graphs with the same nodes) we have integrated two types of metrics available from iGraph:

- **Comparison of communities**. These are metrics that, given two graph partitions, return some metric of the divergence between them:
  - **Variation of Information** [2]. It is a distance metric between clusters commonly used in information theory.
  - **Split-Join distance** [3]. It is the number of transformations (of the cut-paste type) that must be done on one partition to obtain the other.
  - **NMI: Normalized Mutual Information** [4]. Similarity measure (takes maximum value 1 when both communities are equal).
  - **Rand index** [5], which is based on the number of pairs of nodes that coincide in the same or different community in the two partitions.
  - **Adjusted Rand index** [6], which is a measure of similarity between two data partitions related to the precision with which the members of each element of one partition are grouped into the same element of another.
- **Community metrics**. These are metrics that evaluate the quality of a community structure with respect to a given graph. Three measures have been implemented:
  - **Coverage**: ratio between the number of intra-community links with respect to the total number of links.
  - **Performance**: ratio between the number of intra-community links plus the number of non-intra-community links with the total number of potential links.
  - **Modularity.** It is the performance metric optimized by the Louvain and other community detection algorithms. It is the fraction of intra-community links minus the expected value that would be obtained if the links were generated randomly.

The community metrics can be used to evaluate the quality of a community detection algorithm, but also to evaluate the quality of a single graph with respect to a reference graph. For instance, they have been used to compare a similarity graph of papers with respect to a co-citation graph used as reference: both graphs are said to be similar if the community structure obtained for the similarity graph is also a good community structure (according to the community metrics) for the reference graph. These community-based metrics are used to compare two graphs avoiding a “edge-by-edge” comparison. Using these metrics, we can identify similarities between graphs even though they had no common edges.

3.2.2. Comparison of graphs

The metrics for the comparison of two community structures can be used as an indirect measure of the similarity between two graphs, which is grounded on the idea that two similar graphs should have a similar community structure. A more direct comparison can be done using edge-by-edge metrics. Thus, as an alternative to the comparison of communities, the cosine distance between the adjacency matrices of the graphs has been implemented. It is defined as

$$d(V,W) = \frac{trace(V^T W)}{\sqrt{trace(V^T V) \cdot trace(W^T W)}}$$

3.2.3. Impact indicators

A local analysis module has been implemented that allows calculating, from any directed graph, different measures of graph centrality:

- **Degree**, defined by the number of neighbors of a node, normalized with respect to the number of nodes. For directed graphs, it measures the number of outgoing neighbors, also normalized with respect to the total number of nodes.
- **Betweenness** (Betweenness Centrality) [7]: quantifies the number of times that a node is in the shortest path between any two nodes of the graph.
- **Clustering coefficient** [8], measures the proportion of neighbors of a node that are neighbors among themselves.
- **Closeness** (Closeness Centrality). Inverse of the mean distance from a node to all nodes in the network.
- **Eigenvector Centrality** (Eigenvector Centrality) [9], is a measure of the influence of a node in a network based on the calculation of the principal eigenvalue of the adjacency matrix of the graph.
- **PageRank** [10]: is a variant of eigenvector centrality, which measures the influence of a node on the graph, based on the statistical analysis of the probability that certain random walks on the graph pass through a node.
- **Katz** [11]: it is a measure of the influence of a node over all other nodes. The influence of a node is higher over neighbor nodes and, also, over non-neighbor nodes that are connected through multiple short paths.

The implementations of these measures available in the Python NetworkX library have been used.

3.2.4. Disambiguation

A disambiguation algorithm may be required in situations where several entities may be represented by the same name or identifiers. This is common for author graphs from large paper collections because different authors may have the same name.

Remind that the Graph Analysis toolbox can be used to generate graphs of entities from a document collection. For instance, if each document has a list of authors as an attribute, we can convert all attribute values (authors) into nodes and create a bipartite documents-authors graph by linking each document to their corresponding authors, and we can use these graphs to make secondary graphs (e.g., cooperation graphs of authors, similarity graphs of authors, coauthor graphs of documents, etc.). However, if two or more authors share the same name, they will be merged into a single node, and the ambiguity will be propagated over all secondary graphs.

To alleviate this problem, we have included a generic disambiguation tool which, given a node, runs a test to determine if the node may represent more than one entity. The tool is generic,

not necessarily linked to authors, and can be used to test if any node from any graph can be eventually split into several nodes.

We can illustrate the behavior of the disambiguation module with the simplified example in Figure 9: assume that the supergraph contains a similarity graph of papers, G, and a bipartite graph, B, connecting each paper with the names of their authors. Node n is connected to a subset of nodes in G that form two separate clusters that can be easily identified by a community detection algorithm. This may be an indicator that node n represents two authors, and can be split into two separate nodes, each one connected to the subcluster.

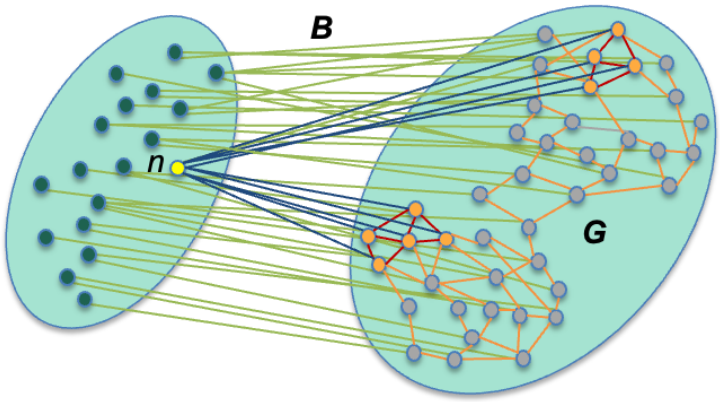

Figure 9: Example of a graph-based disambiguation: a graph B connects authors to their publications. Graph G is a similarity graph of publications. The subgraph of G given by all nodes connected to n through B form two separate clusters that can be detected thr

This clustering effect can be an indicator of an ambiguous node, but it is not conclusive, since it can appear from authors with multidisciplinary works. Therefore, this indicator must be combined with others to improve the identification of ambiguous nodes.

The disambiguation module works in the following steps:

1. For any given node n, identify all superedges, N-G (from the supergraph) that connect a supernode N that contains n to another supernode G.
2. For every superedge N-G:
    a. Select subgraph G_n from G that contains all nodes from G that are connected to N by the superedge.
    b. Apply the Louvain algorithm to find the best 2-community partition from G_n.
    c. Compute the modularity of the given partition.
3. If the product of all modularities obtained for all graphs is above a given threshold, disambiguate the node.

As an example, we can apply the disambiguation of authors from a collection of papers by creating a supergraph with the following components:

1. A similarity graph of papers.
2. A coauthor graph, connecting all papers that share some authors (through the categorical cosine similarity).

3. A graph connecting papers with close publication dates.
4. A graph connecting papers that share the same funding projects.
5. A bipartite graph (superedge) connecting the authors to their papers in each of the above graphs.

A high modularity in all subgraphs corresponding to this graph is a strong indicator of an ambiguous node.

3.2.5. Agent profiling

Since we can generate multiple graphs from a collection of documents, the information relative to any particular entity (e.g., an author) may be spread over different graphs in the supergraph. The graph tool includes a method to gather all information that is relevant to a particular node n, which may include:

- The list of graphs containing n.
- The structure of neighbors from n in each graph.
- The impact indicators for n in each graph.
- All attributes of n in each graph, taken from metadata or derived from the graph tool (like the communities associated to the node).

Since the agent profiling is part of the analysis of impact indicators, it will be described in more detail in Deliverable D3.8.

3.2.6. Example: combination of transformations

As an illustration of how a single data source can be used to generate a structure of graphs, Figure 10 shows a Gephi visualization of the Force Atlas 2 based display of the similarity graph of the 77,990 projects in the CORDIS[8] data set available at the IntelComp data space.

Each point of the graph represents a project, and the links between them their similarity value calculated from the BC of their embeddings. The effect of applying the Louvain algorithm for community detection (using the Python-louvain library) on the similarity graph is shown in the colored regions of the right graph, which shows the 20 largest communities.

To obtain a representation at a lower resolution level, it is possible to convert the community attribute associated with each project into a node and calculate similarities between communities from the similarities between the members of each community. The result is the cross-community similarity graph (Figure 11, left), on which community detection can be applied again, providing a view of the corpus at an even lower resolution level.

[8] The CORDIS corpus contains FP7 and H2020 projects funded by the European Commission. The similarity graph was computed based on a document embedding obtained through an LDA topic modeling algorithm with 50 topics, applied over the title and abstract of each project.

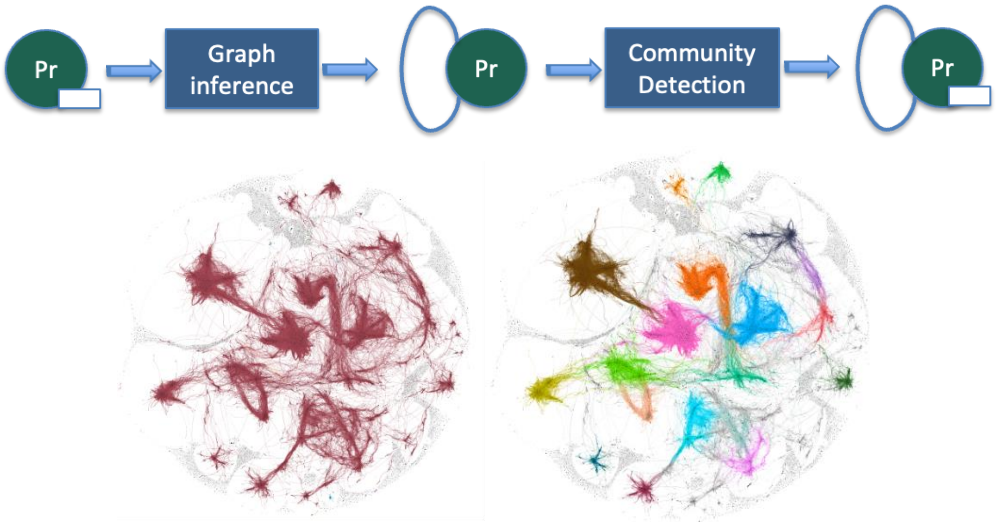

Figure 10: Similarity graph for a collection of projects from the CORDIS database (left). In color, the largest connected component of the graph. On the right, the 20 largest communities, according to the Louvain algorithm

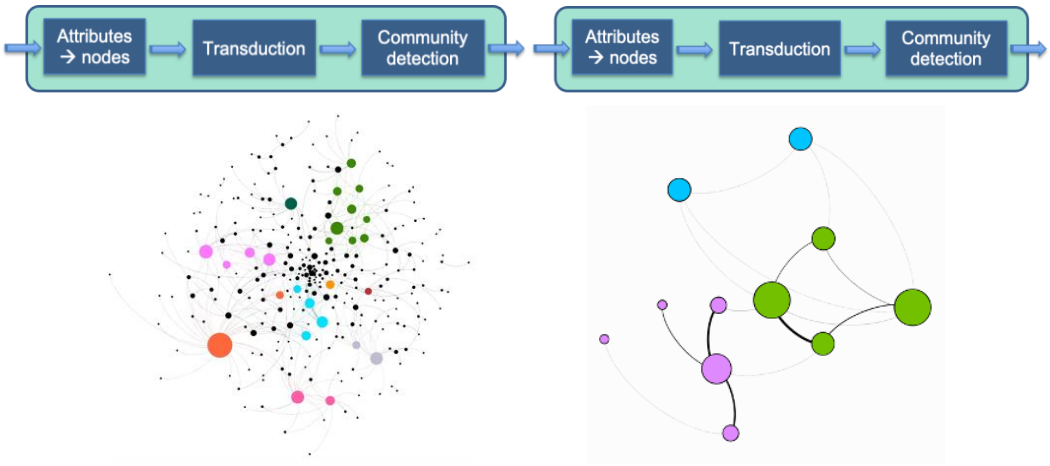

Figure 11: Result of applying the transduction of the similarity graph and detecting communities on the resulting graph, again with the Louvain algorithm. The new super communities add connected communities in the graph of Figure 10 (right).

This process can be repeated iteratively, providing a representation of communities on the original graph at different levels of resolution (Figure 11, right).

3.3. Layout algorithms and visualization

The visualization of graphs is not the focus of the toolbox described in this deliverable, and it is provided by other components of the IntelComp platform (namely the Graph Visualization component of WP4). However, two layout algorithms have been included, in charge of mapping the nodes of the graph into a two-dimensional plane:

- **FA2** (Force Atlas 2)
- **FR** (Fruchterman Reingold)

The coordinates of the nodes are saved as node attributes. The graphs are saved in csv files in a format that facilitates the data importation from the visualization tool and, also, from other open-source graph visualization software like Gephi. Also, graph data can be exported to GEXF, which is the standard format of Gephi.

In addition, the Graph Analysis toolbox includes some visualization methods that can be used to render small to medium size graphs, which are based on the NetworkX library. For bipartite graphs, a visualization module based on Halo is also included.

As an illustrative example, Figure 12 shows the community structure of a citation graph from a collection of papers, rendered with Gephi.

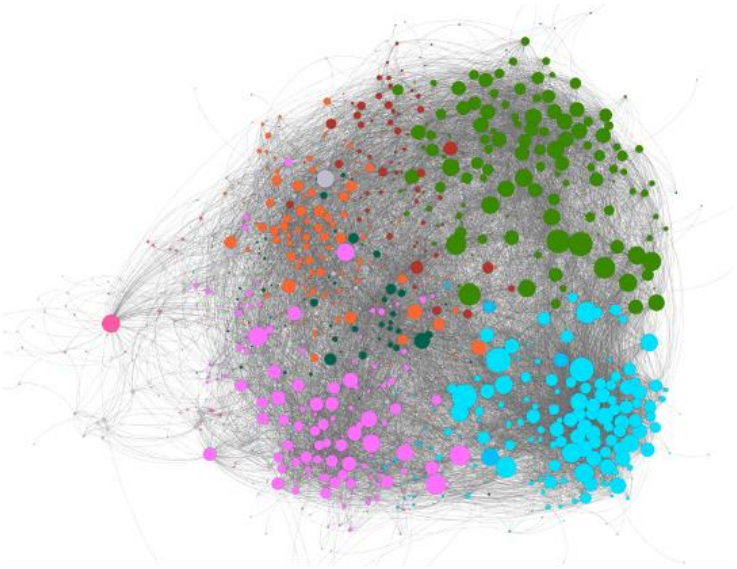

Figure 12: Communities of papers on a co-citation graph rendered with Gephi.

Figure 13 shows a visualization with Halo for a bipartite graph connecting research projects, represented as short marks around a circumference, and the hierarchical structure of communities obtained through the Louvain algorithm.

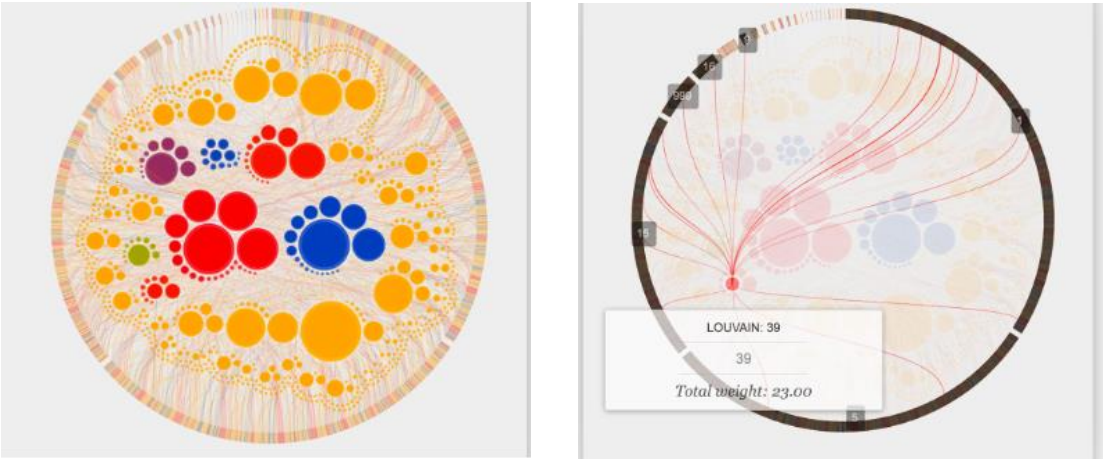

Figure 13: Visualization of a bipartite graph of research projects and thematic communities.

3.4. Summary of algorithms

Table 1 summarizes the main algorithms integrated into the tool, indicating the library and version used, in the cases in which implementations of existing libraries have been imported, and indicating the cases in which the GPU implementations have been tested.

Table 1: Main graph processing methods and the external libraries used in the graph tool

Type	Algorithm	External library	Version	GPU
Similarity graph generation	Battacharyya coefficient	-		Yes
	Jensen-Shannon	-		Yes
	L1 distance	Scikit-learn	1.1.3	NO
	L2 distance	Scikit-learn	1.1.3	NO
Community detection	Fastgreedy	Python-igraph	0.10.2	NO
	Infomap	Python-igraph	0.10.2	NO
	Label propagation	Python-igraph	0.10.2	NO
	Leiden	leidenalg - igraph	0.9.0	NO
	Louvain	Python-louvain	0.15	NO
	Walktrap	Python-igraph	0.10.2	NO
	Connected components	Python-igraph	0.10.2	NO
Graph transduction	Personalized PageRank (PPR)	NetworkX	2.8.4	NO
	Cosine similarity	-		NO
Other secondary graphs	Transitive graphs	-		NO
	Equivalent Graphs	-		NO
	Other	-		NO
Centrality and other local measures	Absolute (unnormalized) in-degree	NetworkX	2.8.4	NO
	Absolute (unnormalized) out_degree	NetworkX	2.8.4	NO
	Betweenness centrality	NetworkX	2.8.4	NO
	Closeness centrality	NetworkX	2.8.4	NO
	Clustering Coefficient	NetworkX	2.8.4	NO
	Degree Centrality	NetworkX	2.8.4	NO
	Eigenvector Centrality	NetworkX	2.8.4	NO
	Katz centrality	NetworkX	2.8.4	NO
	PageRank	NetworkX	2.8.4	NO
Community metrics	Coverage	-		NO
	Performance	-		NO
	Modularity	Python-louvain	0.13	NO
Community comparison	Variation of information	Python-igraph	0.10.2	NO
	NMI: Normalized mutual information	Python-igraph	0.10.2	NO
	Rand index	Python-igraph	0.10.2	NO
	Adjusted Rand index	Python-igraph	0.10.2	NO
	Split-join distance	Python-igraph	0.10.2	NO
	Split-join projection	Python-igraph	0.10.2	NO
Graph layout	Force atlas 2	Fa2	0.3.5	NO
	Fruchterman-Reingold	NetworkX	2.8.4	NO

4. SOFTWARE

4.1. Software structure

The software package can be downloaded from the IntelComp repository in GitHub[9]. It contains all software modules and their documentation. The documentation has been generated using Sphinx in ReadTheDocs format and it has been published in the GitHub page[10]. Figure 14 shows a snapshot of the main page of the documentation, and the page describing one of the main classes.

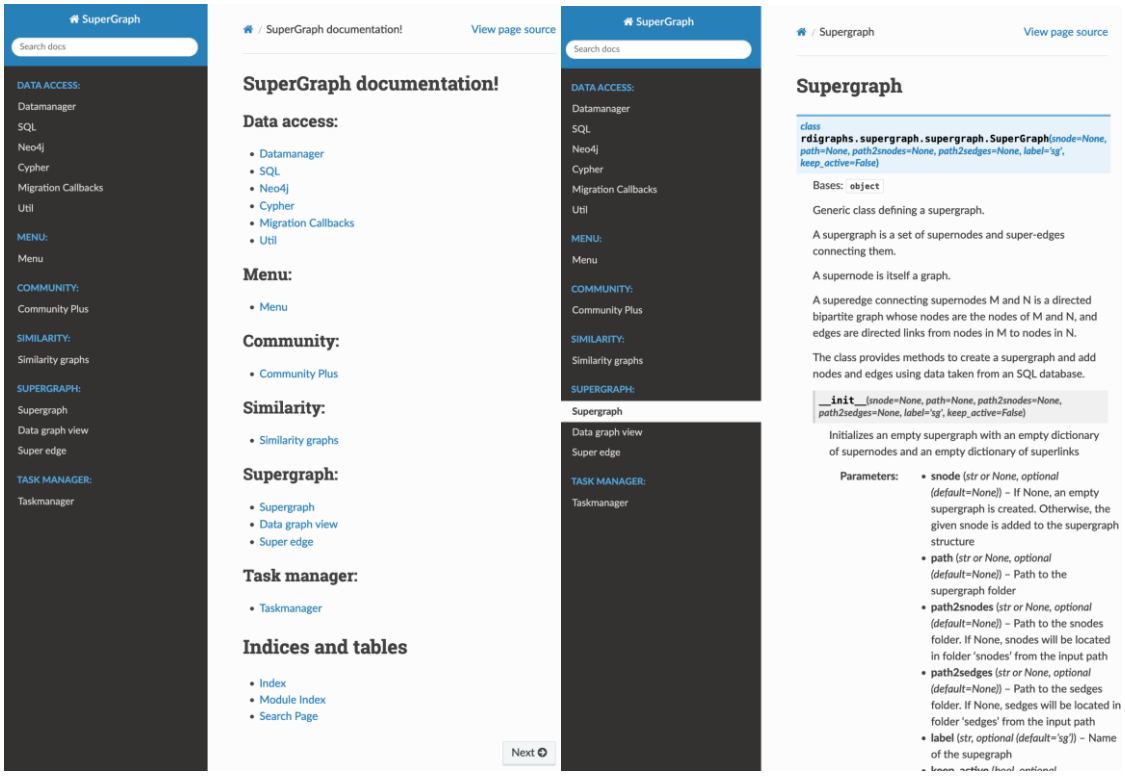

Figure 14: Snapshots of the software documentation. (Left): Main page. (Right) Sample page of the documentation of one of the main classes.

The main components of the folder structure in the software repository are:

- `docs/`: Documentation folder.
- `rdigraphs/`: The python software package. Contains all classes and methods.
- Main scripts: several executable python scripts that can be used to test all software modules through a terminal/command window.

The complete directory tree and some relevant files are shown below. Next to each folder, we provide a brief description of its purpose, indicating relevant files or subfolders when necessary:

[9] https://github.com/IntelCompH2020/GraphAnalysisToolbox
[10] https://intelcomph2020.github.io/GraphAnalysisToolbox/index.html

```
root
├──LICENSE: MIT license file
├──mainRDIgraphs.py: main file to start the basic console-based UI
├──mainRDIlab.py: main file to start an extended console-based UI
├──mainValidate.py: main file to start a console-based UI for graph validation
├──make_conda_environment.sh: shell script for library installation
├──README.md: markdown text file for the project presentation page in GitHub.
├──requirements.txt: file with required library information
├──setup_sphinx.sh: shell script to setup the documentation files
├──config: folder containing other configuration information
├──docs: HTML files used by Github-pages to generate the documentation pages
├──docs_all: other documentation files generated by Sphinx.
├──figures: figures used for project reports and other documents.
│    └──class_diagrams: several UML diagrams of classes from the software project.
├──rdigraphs: main code repository
│    ├──labtaskmanager.py: for tasks related to processing of multiple graphs
│    ├──sgtaskmanager.py: tasks related to supergraph generation, processing, analysis
│    ├──valtaskmanager.py: tasks related to validation of graphs
│    ├──community_plus: contains the class providing community detection algorithms
│    ├──datamanager: classes related to read/write data from/to files or databases
│    ├──menu_navigator: class to manage the user navigation through a multilevel
│    │                   options menu.
│    ├──sim_graph: classes for the generation of similarity graphs
│    │    ├──sim_bigraph.py: classes for the generation of similarity graphs
│    │    ├──sim_graph.py: classes for the generation of similarity graphs
│    │    ├──th_ops.py: classes for the generation of similarity graphs
│    │    └──test: data files related to software testing
│    ├──supergraph: classes for the generation, analysis and processing of graphs,
│    │    │           bipartite graphs and supergraphs
│    │    ├──sedge.py: class for generation and processing of bipartite graphs
│    │    ├──snode.py: class for generation and processing of graphs
│    │    └──supergraph.py: class for the generation and processing of supergraphs
│    └──test: data files related to software testing
├──sphinx-settings: configuration files for the generation of documentation in docs/
└──template:   ome template files used for Halo visualizations
```

The structure of classes of all modules in folder `rdigraphs/` is shown in the UML diagram in Figure 15. Arrows represent parent class relations and edges with diamond terminals represent classes used by other classes.

4.1.1. Graph processing classes

Figure 15 highlights, in yellow, the main classes in relation to the generation and processing of graphs, which are the most relevant software components to be integrated in the main IntelComp tools. Other classes may be needed too, but they should be adapted to the specific data structure required by the IntelComp tools, such as the Interactive Model Trainer (IMT), the STI viewer or the Evaluation Workbench Tools (EWT). A more detailed diagram including the main attributes and methods from each class is shown in Figure 16. These classes are described in some detail:

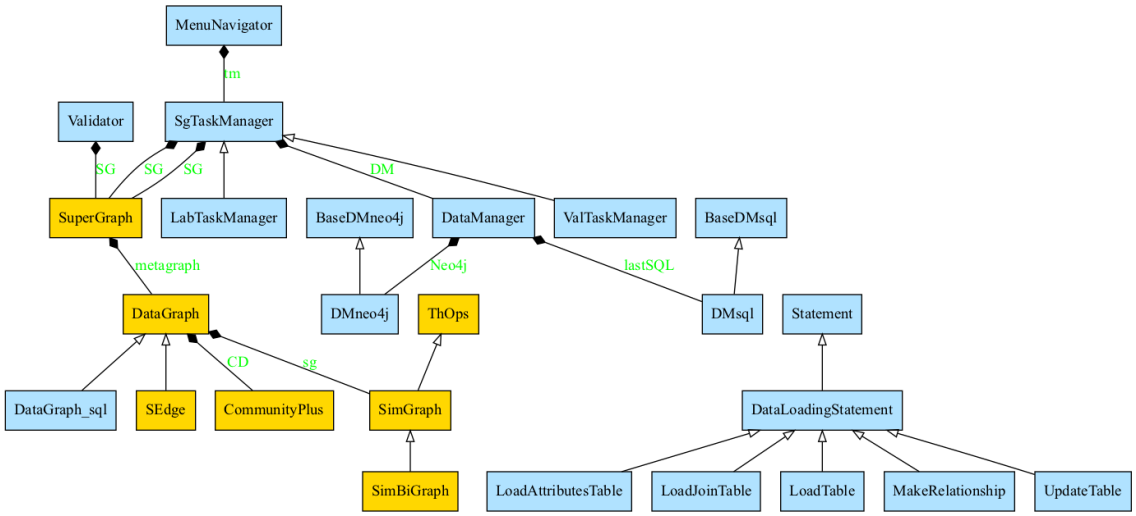

Figure 15: UML diagram of classes from the graph tool.

- ThOps (`rdigraphs/sim_graph/th_ops.py`): a generic class for the computation of thresholded matrix products, which is the basis for the generation of all types of similarity graphs in the tool: document similarity graphs, author similarity graphs, citation graphs, etc. Essentially, the class is in charge of the efficient and scalable computation of the thresholded products

  $$S = \phi_t(X^T Y)$$

  where $\mathbf{X}$ and $\mathbf{Y}$ are the input matrices of embeddings and ϕ_t is the threshold function with threshold t (that is $\phi_t(x) = x \cdot (x \geq t)$, which is applied component-wise). Since the direct computation of the above expression is infeasible for large graphs in the order of million nodes, the output matrix is computed blockwise taking advantage of GPU processing, if available.

- SimGraph (`rdigraphs/sim_graph/sim_graph.py`): this class inherits and extends ThOps to facilitate the creation of different types of similarity graphs, using different similarity measures or distance metrics. In addition, it takes advantage of efficient implementation of neighbor graphs available in the *Scikit-learn* library.
- SimBiGraph (`rdigraphs/sim_graph/sim_bigraph.py`): this class inherits and extends SimGraph to the generation of bipartite similarity graphs.
- CommunityPlus (`rdigraphs/community_plus/community_plus.py`): it provides community detection algorithms for graphs. This is a wrapper class that does not contain any proper implementation of the algorithms but provides access to different open-source implementations from NetworkX and other external packages.
- DataGraph (`rdigraphs/supergraph/snode.py`): the main class for the generation and management of supernodes (single graphs). It uses SimGraph for the generation of similarity graphs, and CommunityPlus for the application of community detection algorithms. Additionally includes functionality for graph edition, importation, analysis (graph sampling, graph filtering, identification of equivalent nodes, evaluation of community structures, etc.) and graph exportation methods.

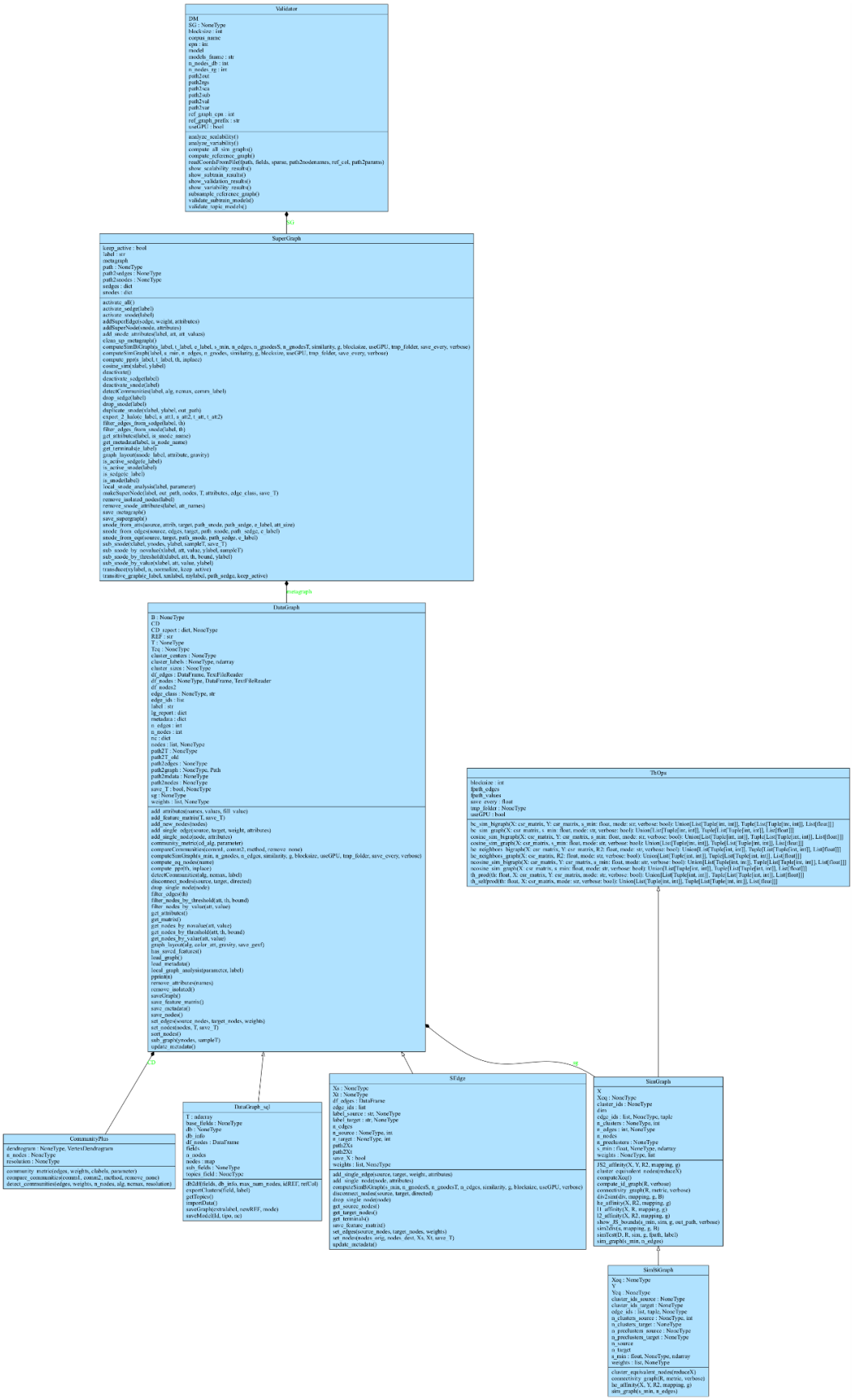

Figure 16: Diagram of classes related to the generation, analysis and processing of graphs.

- SEdge (`rdigraphs/supergraph/sedge.py`): extends DataGraph to bipartite graphs (superedges).
- SuperGraph (`rdigraphs/supergraph/supergraph.py`): this is the major class in the tool. It is in charge of generating a structured collection of graphs. A supergraph is defined as a collection of graphs (*supernodes*, which are objects from the DataGraph class). The relation between supernodes is defined by bipartite graphs (*superedges*, which are objects from the SEdge class). The overall supergraph structure is, thus, a multigraph, stored in the SuperGraph.metagraph attribute, which is also a DataGraph object.
- Validator (rdigraphs/Validator.py): this is a container class of several methods to evaluate the quality of a collection of graphs. It can be used to (1) validate the scalability of the graph construction with respect to the number of nodes and edges, (2) analyze the variability of graphs caused by variation in the embedding model (e.g., the hyperparameters of the LDA algorithm), (3) evaluate the quality of multiple graphs with respect to a reference graph that is taken as a gold standard (e.g., to evaluate similarity graphs of papers with respect to a co-citation reference graph). The validation methodology implemented in this class has been tested over scientific corpora in [12].

4.1.2. Task control classes

The graph tool includes some python scripts that facilitate the generation and processing of graphs by means of interactive menus. The tasks executed from the menus are defined by specific methods in several task control classes, whose UML diagram is shown in Figure 17.

- MenuNavigator (`rdigraphs/menu_navigator/menu_navigator.py`), a class that defines the logic for the navigation through the interactive menus. It reads the menu structure from a configuration file, prints the available options at each time of the menu navigation, requests the user to make successive choices during navigation and, finally, launches the task, from the task manager, selected by the user.
- SgTaskManager (`rdigraphs/sgtaskmanager.py`). This class is in charge of creating the main objects required to run a supergraph project and calling the methods required to carry out all of the processing tasks.
- LabTaskManager (`rdigraphs/labtaskmanager.py`): an extension of the SgTaskManager class that provides additional functionality for processing and visualizing results obtained from multiple graphs, and for graph validation tasks.
- ValTaskManager (`rdigraphs/valtaskmanager.py`): an extension of class SgTaskManager that provides additional functionality for validation tasks.

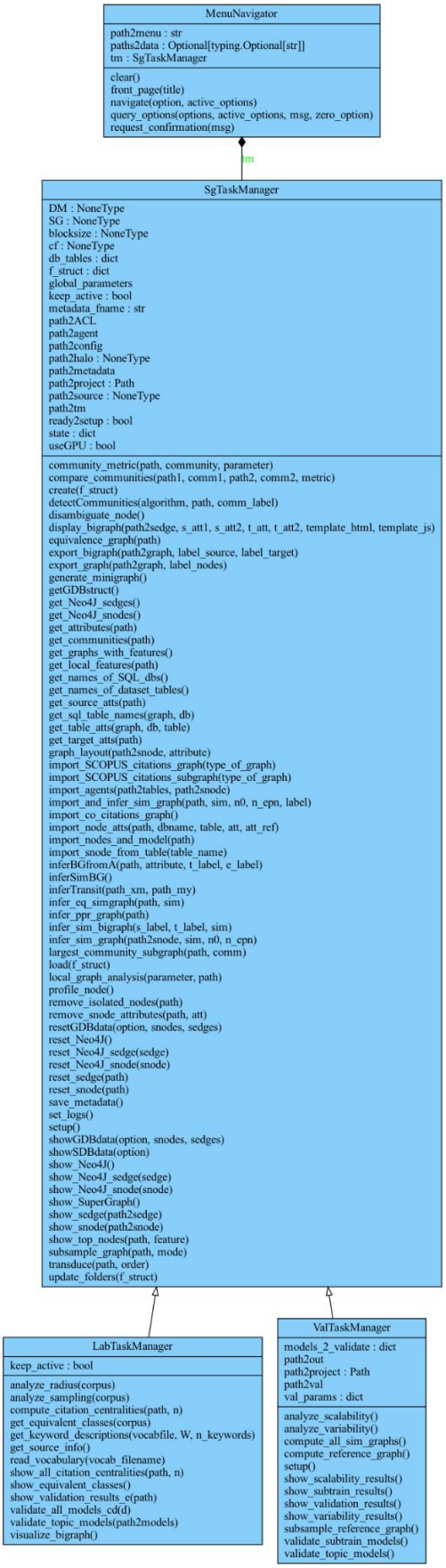

Figure 17: Diagram of classes related to task control.

4.1.3. Input / output classes

Finally, some additional classes have been generated in order to import and export data from specific databases, in particular, datasets with embeddings of projects, papers and patents, based on topic models or transformers, and citation graphs, taken from files, SQL databases or Neo4J graph databases. All these classes are controlled or used by the `DataManager` class (in `rdigraphs/datamanager/datamanager.py`). This is a provisional class that can be used to test the software using particular test datasets, but it should be replaced by the appropriate I/O software in charge of all read and write operations according to the IntelComp data structure in the integration phases of the project. For this reason, we omit a detailed explanation of these classes.

4.2. Software prerequisites

4.2.1. Python prerequisite packages

The software has been developed in Python, requiring some open-source libraries. These can be found in the *requirements.txt* file, and installed using the following command in a terminal:

```
$ pip install -r requirements.txt
```

The main libraries required to run the python application are listed below. The table also shows the library versions integrated in the latest code version, and its license.

Note that python 3.8 is required. The latest version of fa2, implementing the force-atlas layout algorithm, is not available for python 3.9+. Disabling fa2 (and, thus, the force-atlas option) the code has been successfully tested on python 3.10.

4.3. Running the application

4.3.1. Execution commands

The main goal of the graph toolbox is to provide classes and methods to be integrated in the IntelComp main tools. However, it can be used as a standalone application through several python scripts that can be used to explore the functionality of the software:

- `mainRDIgraphs.py`: to run all the basic operations with graphs, bipartite graphs and supergraphs
- `mainRDIlab.py`: to run tasks involving iterations over multiple graphs
- `mainValidate.py`: to run specific graph validation experiments

Table 2: Python prerequisite packages.

Package	Version	License
dask	2022.2.1	BSD
fa2	0.3.5	GPLv3
ipython	8.6.0	BSD
leidenalg	0.9.0	GPLv3+
matplotlib	3.5.3	Python Software Foundation
NetworkX	2.8.4	3-clause BSD
numba	0.56.3	2-clause BSD
numexpr	2.8.4	MIT
numpy	1.21.5	BSD
pandas	1.5.1	BSD
pyarrow	8.0.0	Apache Software
python	3.8.15	Python Software Foundation
python-igraph	0.10.2	GPL-2.0
python-louvain	0.15	3-clause BSD
pyyaml	6.0	MIT
Scikit-learn	1.1.3	3-clause BSD
scipy	1.9.3	3-clause BSD
seaborn	0.12.0	BSD
tqdm	4.64.1	MIT; MMPL 2.0

In order to start any version of the application, the following command needs to be executed:

```
$ python main_script.py
    --p project_folder
    --source datasets_folder
```

where:

- **main_script** is any of the executable scripts listed above.
- **project_folder** is the path to a new or an existing project in which the application output will be saved.
- **datasets_folder** is the path to the folder containing the source data files and folders

Since `mainRDIlab.py` and `mainValidate.py` are extensions of `mainRDIgraphs.py` that provide some extra functionality, we will focus the attention on `mainRDIgraphs.py`.

4.3.2. Startup

If the **project_folder** does not exist, the application creates it, along with the file and folder structure required to store the output files. The structure of this folder will be described later.

In addition, a copy of the default configuration file (located in `config/parameters.default.yaml`) is saved in the project folder with name `parameters.yaml`. The list of parameters, their function, and the default values used for application testing are shown in Annex A.

Since the user might be interested in modifying some of these parameters, the application needs a confirmation to activate the configuration file from the user. For this reason, the first time the application is run for a given file, only the activation of the configuration file is available, as shown in Figure 18.

```
**************************
*** RDI Graph Analyzer ***
**************************



***************
*** MAIN MENU.
Available options:
 1. Activate configuration file
 0. Exit the application

What would you like to do? [0-1]: 1
```

Figure 18: Startup menu. No options are available but the activation of the config file.

4.3.3. Menu navigation

After the activation, the whole set of options is available by navigating through a hierarchical structure of menus and sub-menus. The options in the root menu of `mainRDIgraphs.py` are shown in Figure 19.

```
***************
*** MAIN MENU.
Available options:
 1. Activate configuration file
 2. Import data
 3. Manage SQL or Neo4J databases
 4. Pre-visualize supergraph
 5. Reset graph (supernode)
 6. Reset bigraph (superedge)
 7. Graph tools
 8. Graph inference tools
 9. Local graph analysis
 10. Community detection tools
 11. Evaluate community partitions
 12. Compare two communities
 13. Graph visualization
 0. Exit the application

What would you like to do? [0-13]:
```

Figure 19: Main menu of the command-line application.

The complete tree of options from `mainRDIgraphs.py` is shown in Table 3. In the table, each option has a label and a short description. Labels in boldface are gates to further options. Shaded

labels correspond to specific tasks, which are executed by a method from the task manager (class `SgTaskManager` in `rdigraph/sgtaskmanager.py`) with the same name as the label. In some cases, some parameters are requested as additional options before the task is launched.

Table 3: Complete list of available options from the hierarchy of menus in mainRDIgraphs.py. Shaded labels correspond to the methods in the task manager that runs the corresponding task.

- [create]: Create new project [only if project_folder is not given]
- [load]: Load existing project [only if project_folder is not given]
- setup: Activate configuration file
- **import_data**: Import data
  - import_snode_from_table: Import nodes and features from table files
  - import_nodes_and_model: Import nodes from table files and features from npz files
  - import_co_citations_graph: Import co-citations graph from DB
  - import_node_atts: Load node attributes from SQL databases
    - Pu: Publications
    - Pr: Projects
    - Pa: Patents
  - import_agents: Import project-researchers bipartite graph from file
- **manage_databases**: Manage SQL or Neo4J databases
  - showSDBdata: Show SQL data sources
  - manage_Neo4J: Manage Neo4J database
    - **showGDBdata**: Show Neo4J Super Graph
      - show_Neo4J: Overview of the whole graph databases
      - show_Neo4J_snode: Show information about a snode
      - show_Neo4J_sedge: Show information about a sedge
    - **resetGDB**: Reset Neo4J Graphs
      - reset_Neo4J: Reset the whole Neo4J graph databases
      - reset_Neo4J_snode: Reset a specific Neo4J snode
      - reset_Neo4J_sedge: Reset a specific Neo4J sedge
    - export_graph: Export graphs to Neo4J
    - export_bigraph: Export bigraph to Neo4J
- **graph_previews**: Pre-visualize supergraph
  - show_SuperGraph: Show supergraph structure
  - show_snode: Quick preview of graph
  - show_sedge: Quick preview of bipartite graph
- reset_snode: Reset graph (supernode)
- reset_sedge: Reset bigraph (superedge)
- **graph_tools**: Graph tools
  - generate_minigraph: Generate a synthetic graph for simple testing
  - subsample_graph: Subsample snode
    - inplace: Replace the original snode
    - newgraph: Keep the original snode and create a new one
  - largest_community_subgraph: Make a subgraph with the largest community
  - remove_isolated_nodes: Remove isolated nodes
  - remove_snode_attributes: Remove attribute from graph nodes
  - disambiguate_snode: Evaluate if a node is ambiguous.
- **gInference**: Graph inference tools
  - equivalence_graph: Cluster equivalence classes: from A to eqA
  - infer_sim_graph: Similarity graph: from A_X to A-A
    - He: He: 1 minus squared Hellinger's distance (JS) (sklearn-based)
    - He2: He2: self implementation of He (faster)

        - BC: BC: Bhattacharyya coefficient
        - l1: l1: 1 minus l1 distance
        - JS: JS: Jensen-Shannon similarity (too slow)
        - Gauss: Gauss: An exponential function of the squared l2 distance
        - He->JS: He->JS: JS through He and a theoretical bound
        - He2->JS: He2->JS: Same as He->JS, but using implementation He2
        - l1->JS: l1->JS: JS through l1 and a theoretical bound
        - cosine: cosine: Cosine similarity
        - ncosine: ncosine: Normalized cosine similarity (rescaled to [0, 1])
    - import_and_infer_sim_graph: Import and infer Similarity graph: from A_X to A-A
        - [Same options than nfer_sim_graph]
    - infer_eq_simgraph: Equivalent Similarity graph: from A_X to eqA-eqA
        - [Same options than infer_sim_graph]
    - infer_sim_bigraph: Similarity bipartite graph: from A_X, B_X to A-B
        - He2: He2: 1 minus squared Hellinger's distance
        - He2->JS: Jensen-Shannon similarity
        - cosine: cosine: Cosine similarity
        - ncosine: ncosine: Normalized cosine similarity (rescaled to [0, 1])
    - infer_ppr_graph: PPR graph
    - inferBGfromA: Bipartite graph from attributes: from A_B to A->B
    - transduce: Transductive graph: from A-A->B to B-B
        - 1: First-order graph (for transduced similarity graphs)
        - 0: Zero-order graph (for cooperation graphs)
    - inferTransit: Transitive graph: from A->B->C to A->C
- local_graph_analysis: Local graph analysis
    - centrality: Eigenvector Centrality
    - degree: Degree Centrality
    - betweenness: Betweenness centrality
    - closeness: Closeness centrality
    - cluster_coef: Clustering Coefficient
    - pageRank: PageRank
    - katz: Katz centrality
    - abs_in_degree: Absolute (i.e. unnormalized) in-degree
    - abs_out_degree: Absolute (i.e. unnormalized) out_degree
- detectCommunities: Community detection tools
    - leiden: Leiden
    - louvain: Louvain
    - fastgreedy: Fastgreedy
    - walktrap: Walktrap
    - infomap: Infomap
    - labelprop: Label Propagation
    - kmeans: kmeans
    - aggKmeans: Agglomerative Kmeans
    - cc: Connected components
- community_metric: Evaluate community partitions
    - coverage: Coverage
    - performance: Performance
    - modularity: Modularity
- compare_communities: Compare two communities
    - vi: VI: Variation of information metric, Meila (2003)
    - nmi: NMI: Normalized mutual information, Danon et al (2005)
    - rand: RI: Rand index, Rand (1971)
    - adjusted_rand: ARI: Adjusted Rand index, Hubert and Arabie (1985)
    - split-join: SJD: Split-join distance of van Dongen (2000)
    - split-join-proj: SJP: Split-join projection of van Dongen (200)

- **display_graphs**: Graph visualization
  - show_top_nodes: Show top nodes ranked by attribute value
  - graph_layout: Graph layout
  - display_bigraph: Visualize bipartite graph
  - profile_node: Analyze single node over the whole supergraph

4.4. Project folder structure

If the **project_folder** does not exist, it is created by the application, along with the files and folders required to store the output files and save the project status: the default structure consists of the following:

1. `parameters.yaml`: the configuration file of the project.
2. `metadata.pkl`: a file with metadata that stores the status of the project (for internal use of the application).
3. `msgs.log`: log file of the latest code execution with this project (for debugging purposes).
4. `metagraph/`: it contains the files describing the supergraph structure.
5. `graphs/`: it contains all graphs (supernodes) from the supergraph.
6. `bigraphs/`: it contains all bipartite graphs (superedges) from the supergraph.
7. `output/`: to save some output files with some results of the graphs processing. All results from the local graph analysis options are saved here.
8. `import/`: to store some data imported from other sources that could be used to create new graphs or process existing graphs.

4.4.1. Graph folders

All data related to each graph of the supergraph is stored in a folder with the name of the graph. For instance, if graph G1 is created, the following structure of files is added to the `graphs` folder.

```
graphs
└──G1: Name of the graph.
     ├──G1_nodes.csv [required]: table of nodes and node attributes.
     ├──G1_edges.csv [required]: table of edges and edge attributes.
     ├──G1_mdata.yml [required]: metadata about the graph generation.
     ├──feature_matrix.npz [optional]:  matrix of node features.
     └──G1_mdata.gexf [optional]: graph with layout data.
```

Nodes and edges, including their respective attributes, are stored in csv files `G1_nodes.csv` and `G1_edges.csv`, respectively.File `G1_mdata.yml` contains metadata information about the generation of the graph and its attributes. A sample metadata file is shown in Figure 20(left), for a similarity graph of 42069 nodes. It contains general information about the graph, nodes, edges and the community detection algorithms applied over the graph.

The matrix of node features (`feature_matrix.npz`) is optionally saved when nodes have a vector-space representation, after the computation of similarity graphs.

File `G1_mdata.gexf` is optionally saved by the layout method, to save the spatial coordinates and the colors of the nodes in GEXF format, to facilitate a later visualization.

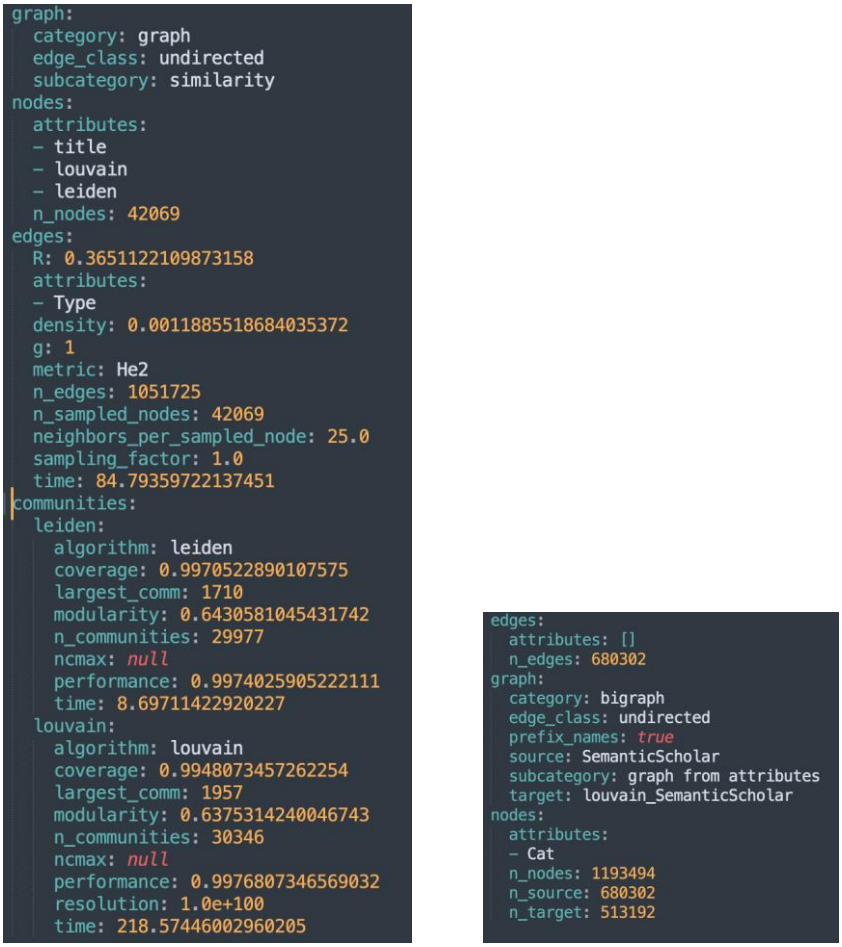

Figure 20: (Left) Sample metadata file of a similarity graph with 42069 nodes with 2 node attributes computed from community detection algorithms. (Right) Sample metadata of a bipartite graph. Source nodes are Semantic Scholar paper, target nodes are the Louvain

4.4.2. Bipartite Graph folders

Bipartite graph folders (*superedges* of the supergraph) have a similar structure to the graphs, but with an additional, optional, file holding information for halo visualization.

```
bigraphs
└──G2: Name of the bipartite graph.
     ├──G2_nodes.csv [required]: table of nodes and node attributes.
     ├──G2_edges.csv [required]: table of edges and edge attributes.
     ├──G2_mdata.yml [required]:  metadata about the graph generation.
     ├──feature_matrix.npz [optional]:  matrix of node features.
     ├──G2_mdata.gexf [optional]:  graph data for visualization in Gephi.
     └──halo_G2.csv [optional]:  metadata about the graph generation.
```

A sample metadata file of a bipartite graph is shown in Figure 20 (right): the graph connects 680302 papers from Semantic Scholar with the Louvain communities computed over a similarity graph.

4.4.3. Metagraph folder

The `metagraph` folder contains the files describing the whole supergraph structure. Therefore, it contains information about all the graphs and bipartite graphs in the project, and the relations between them. Since the supergraph is itself a graph, the folder structure is similar to that of any other graph

```
metagraph
 ├──metagraph_nodes.csv [required]: table of nodes and node attributes.
 ├──metagraph_edges.csv [required]: table of edges and edge attributes.
 └──metagraph_mdata.yml [required]:  metadata about the graph generation.
```

As an illustration, Figure 21 and Figure 22 show the file of nodes and edges, respectively, of a supergraph with 7 supernodes and 4 superedges. Figure 21 shows that 3 supernodes in the supergraph are similarity graphs computed from CORDIS projects and Semantic Scholar (SS) papers. Two transductive graphs of communities have been computed from CORDIS using the louvain and walktrap algorithms. Furthermore, two second-level transductive graphs of communities have been computed as the transduction over these community graphs. Figure 22 shows the superedges connecting the CORDIS graph with the 1st level transductive graphs, and the superedges connecting the first level transductive graphs with their corresponding second-level graphs.

Id	category
CORDIS_50	similarity
SS_25	similarity
SS_50	similarity
louvain_CORDIS_50	transductive
walktrap_CORDIS_50	transductive
louvain_louvain_CORDIS_50	transductive
walktrap_walktrap_CORDIS_50	transductive

Figure 21: Sample file of nodes (metagraph_nodes.csv) from a supergraph. Each node of the supergraph represents a graph.

Source	Target	Type	Weight	category	label
CORDIS_50	louvain_CORDIS_50	directed	1	snode_from_atts	CORDIS_50_2_louvain_CORDIS_50
CORDIS_50	walktrap_CORDIS_50	directed	1	snode_from_atts	CORDIS_50_2_walktrap_CORDIS_50
louvain_CORDIS_50	louvain_louvain_CORDIS_50	directed	1	snode_from_atts	louvain_CORDIS_50_2_louvain_louvain_CORDIS_50
walktrap_CORDIS_50	walktrap_walktrap_CORDIS_50	directed	1	snode_from_atts	walktrap_CORDIS_50_2_walktrap_walktrap_CORDIS_50

Figure 22: Sample file of edges (metagraph_edges.csv) from a supergraph. Each edge of the supergraph represents a bipartite graph. In the example, all bipartite graphs were computed by connecting each node to its community, according to a community detection algorithm.

5. CONCLUSIONS

This document describes the Graph Analysis toolbox that has been developed for Task T3.7 of the IntelComp project. The toolbox has been developed using python and provides methods for the generation of graphs from feature vectors, tools to use metadata as the basis of new graphs and methods to generate new graphs as a combination of other graphs.

Although the toolbox can be used generally for any type of data sources, it is specially oriented to the processing and analysis of large corpora of scientific documents. Starting from a given corpus of scientific documents (projects, papers, patents, etc.) from a scientific field, the tools can be used to generate graphs describing the semantic similarity relations between documents, infer graphs connecting authors according to different types of relations (research affinity, cooperation, co-citation, etc.), and also infer graphs associated to other metadata in the corpus (affiliation, funding organizations, keywords, etc.). Additional analysis tools for node profiling and centralization measures can be used to identify the main features of documents, authors or other metadata based on the structure of their connections in the graph.

The software has a modular structure and has been oriented to facilitate the integration of the key components into the main IntelComp tools. Moreover, a complete python application has been developed, and all the software functionalities can be exploited by using a terminal-based interface, allowing its integration in automated scripted processes and by user friendlier navigation through a menu hierarchy.

6. REFERENCES

[1] Liu, T., Moore, A. W., & Gray, A. (2006). New Algorithms for Efficient High-Dimensional Nonparametric Classification. *Journal of Machine Learning Research*, 7, 1135-1158.

[2] Meila, M., “Comparing clusterings by the variation of information”. In: B. Scholkopf, M.K. Warmuth (eds). *Learning Theory and Kernel Machines: 16th Annual Conf. on Computational Learning Theory and 7th Kernel Workshop, COLT/Kernel 2003, Washington, DC, USA. Lecture Notes in Computer Science*, vol. 2777, Springer, 2003.

[3] van Dongen, D., “Performance criteria for graph clustering and Markov cluster experiments”. *Technical Report INS-R0012*, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.

[4] Danon, L., Díaz-Guilera, A., Duch, J., Arenas, A., “Comparing community structure identification”. *J Stat Mech* P09008, 2005.

[5] Rand, W.M., “Objective criteria for the evaluation of clustering methods”, *J Am Stat Assoc* 66 (336): 846-850, 1971.

[6] Hubert, L., Arabie, P., “Comparing partitions”. *Journal of Classification*, 2:193-218, 1985.

[7] U. Brandes, (2001). A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, *25*(2), 163-177.

[8] M. Latapy, (2008). Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, *407*(1-3), 458-473.

[9] P. Bonacich (1986): Power and Centrality: A Family of Measures. *American Journal of Sociology* 92(5):1170–1182.

[10] L. Page, S. Brin, R. Motwani, T. Winograd (1999). *The PageRank citation ranking: Bringing order to the web*. Stanford InfoLab.

[11] L. Katz (1953): A New Status Index Derived from Sociometric Index. *Psychometrika* 18(1):39–43.

[12] Vázquez, M. A., Pereira-Delgado, J., Cid-Sueiro, J., & Arenas-García, J. (2022). Validation of scientific topic models using graph analysis and corpus metadata. *Scientometrics*, *127* (9), 5441-5458.

7. ANNEXES

A. Default Configuration File.

The first time the application is run, a copy of the configuration file is stored in the project folder with name `parameters.yaml`. Table 4 shows the list of parameters, their function, and the default values used for application testing.

Table 4: Default configuration file. Comments explain the meaning of each parameter.

```
# Path to Halo software
path2halo: '../myhalo'

# Parameters for algorithms
algorithms:
  # Size of blocks for the computation of similarity graphs. 25_000 is
  # ok for computation in a standard PC. Larger values may cause large
  # processing times caused by memory swapping.
  blocksize: 25_000
  # If True, cupy library will be used when possible
  useGPU: False
  # If True, affinities for similarity graphs are computed from
  # distances by rescaling values so that the minimum is zero and
  # maximum is 1.
  rescale: False

# Parameters for model validation
validate_all_models:
  spf: 1        # Sampling factor
  rescale: False
  n_edges_t: 100_000
  g: 1
  # Average number of edges per node in the graphs used for validation
  epn: 100     # Average
  # Prefix of the names of the reference graphs
  ref_graph_prefix: RG
  # Size of the initial set of nodes (only for some databases)
  # 4-5 times ref_graph_nodes_target is ok.
  ref_graph_nodes_init: 100000
  # Target number of nodes in the reference graph
  ref_graph_nodes_target: 20000
  # Target average number of edges per node in the reference graph
  ref_graph_epn: 100

# SQL and Graph DataBases
connections:
  SQL:
   # Select the databases to be used in the project.
   db_selection:
   # Each selection must have the form:
   #   label: db_name
   # where label is a mnemonic used to identify the database, and
   # db_name is the name of the db below. You can select different
   # dbs for projects, patents, publications & companies as
   #      Pr: db_name01
   #      Pa: db_name02
   #      Pu: db_name03
   #      Co: db_name04
   # where db_name01, db_name02, must be the
   databases:
```

```
   # Here, you can include a complete list of available databases.
   # Only those included in db_selection (above) will be connected.
   # The key of each DB is the name of the DB as specified
# when opening the connection. For instance:
   # db_name01:
   #  category: Pr                    # Type of database
   #  connector: &sql_con mysql       # Use & to allow dereferencing
   #  server: &sql_server hal01.tsc.uc3m.es  # Your server address
   #  user: &sql_user username        # Write username here
   #  password: &sql_password xxxxxxxx     # Write password here
  # Uncomment an set neo4j parameters if available
  # neo4j:
  #   server: xxxxxxx  # Write server here
  #   user: neo4j      # Write username here
  #   password: xxxxxx # Write password here

# Specify format for the log outputs
logformat:
  filename: msgs.log
  datefmt: '%m-%d %H:%M:%S'
  file_format: '%(asctime)s %(levelname)-8s %(message)s'
  file_level: INFO
  cons_level: DEBUG
  cons_format: '%(levelname)-8s %(message)s'
```