UDC 519.174.1

Overview of social networks research software

Migran N. Gevorkyan*, Anna V. Korolkova*, Dmitry S. Kulyabov*†

* *Department of Applied Probability and Informatics*
Peoples' Friendship University of Russia (RUDN University)
6 Miklukho-Maklaya St, Moscow, 117198, Russian Federation
† *Laboratory of Information Technologies*
Joint Institute for Nuclear Research
6 Joliot-Curie, Dubna, Moscow region, 141980, Russia

Email: gevorkyan-mn@rudn.ru, korolkova-av@rudn.ru, kulyabov-ds@rudn.ru

This paper discusses the software tools which are used to study of social networks (graphs). The study of social graphs received a significant boost with the development of the Internet, as there was open access to data on different communities, uniting millions of people. This allowed to conduct research on a large amount of real data, but the researcher is now required knowledge from different areas of computer technology and programming. The study of social graphs is a complex task. The researcher is required to obtain data, process them, visualize (build a graph) and calculate the metrics of interest to him. In this paper, we looked at the tools that allow you to implement each stage of the study: the NetworkX library, gephi and Graphviz utilities, and the language for graph representation .dot. In the report, we gave a concrete example of the application of all these software tools to the task of obtaining data about the authors and their co-authors from the Scopus abstract database using the open API provided by these services.

Key words and phrases: computer science, social networks, graphs, Python.

1. Introduction

The task of analyzing a particular social network can be divided into three stages.

- Getting data to build the studied social network. Make representation of network in a structured form, which makes possible to build a graph.
- Visualization of the obtained data for the purpose of preliminary analysis to identify the direction of further research. It is highly desirable to make interactive visualization which allows one to manipulate graph components in real time.
- Software representation of studied graph for mathematical analysis, in particular for calculation of various metrics. Usually for this purpose some programming language is used.

Each of these stages requires special knowledge from the researcher. To get data, one need to understand the concept of RESTFull interfaces, know the basics of HTTP Protocol, JSON and XML formats. To visualize the obtained data, it is necessary, on the one hand, to know the basics of mathematical graph theory, and on the other hand, to have an idea of the various software formats for graph storage: dot, GraphML, XGML, etc. Finally, for mathematical analysis, it is necessary not only to know the mathematical apparatus, but also to be able to use the programming language and the selected library.

Our goal is not to give any complete overview of the software. We will only describe the tools that will be used in our example with the Scopus abstract and citation database.

2. The software graphs representations

Nowadays `Python` programming language [1–4] become the standard language for data research because of its versatility, simple syntax and relatively easy to learn. `Python` serves as a binding language: some compiled language is used to implement resource-intensive computational algorithms (most often `C/C++/Fortran`), and `Python` itself is used to process and visualize data in combination with the interactive shell `Jupyter` [5].

From the whole variety of modules for `Python`, created to work with graphs, we can distinguish the following three most developed and functional.

- Library `NetworkX` [6, 7] is implemented in pure Python and provides tools to create, manipulate, visualize (using the `Matplotlib` [8]) and analyze graphs. It allows one to save graph data in various formats, which makes it possible to export the created graph to a file for storage and transfer to other programs for processing.
- The `graph-tool` [9] library is used to manipulate, visualize, and analyze graphs statistics. Basic algorithms and data structures are implemented using the `C++` language using templates and the `Boost` [10] library. Module supports multi-threaded computing through `OpenMP`. Its also implements its own binary graph data storage format, which is more efficient than standard text formats.
- Library `igraph` [11] is a set of tools for analyzing graphs. The main components are written in `C`, but there are interfaces for `Python`, `R`, `C++`, and `Mathematica`.
- The library `NetworKit` [12] is written in `C++` and `Python`. To link `C++` parts to `Python` parts, use `Cython`. Supports parallel computing (OpenMP-based) for some algorithms. It is possible to export graphs in a format supported by `NetworkX`.

In our opinion, if the task is to manipulate relatively small graphs, then the best choice is the library `NetworkX`, in the case of large graphs, the library `graph-tool` is better suited.

It is also worth noting that a large number of libraries and tools for working with graphs are also available for the JavaScript language. However, JavaScript is rarely used in academic research.

The following table provides summary information for the libraries in [13]

	graph-tool [9]	iGraph [11]	NetworkX [6]	NetworKit [12]
Lang	C++	C	Python	Python, Cython, C++
Bindings	Python	Python, R, C++, Mathematica	Python	C++, Python
Parallelization support	+	-	-	+

3. Network visualization

Each of the three libraries discussed above has the ability to visualize the generated graph. However, this is not their primary purpose, and visualization capabilities can often be insufficient. In addition, as noted above, it is useful to be able to manipulate graph elements interactively. From the whole set of visualization tools, we distinguish the following two programs.

- The set of programs and libraries **Graphviz** [14] is intended for automatic visualization of graphs. The main functionality is implemented as a command-line utility, but **Graphviz** can be used as a library. Although **Graphviz** itself is written in **C**, there are interfaces for **Python**, **Tcl**, **Ruby**, **Perl**, and **C#**. To describe the structure of the graph, **Graphviz** use its own language **DOT**, which allows one to describe the components of the graph in text form.
- The program **Gephi** [15] is also designed to render graphs. It has a graphical interface and allows one to manipulate graph elements interactively with the mouse. **Gephi** is written in **Java**, but it does not require knowledge of programming languages, as all manipulations are carried out through the graphical interface.

4. Data downloading and storage

4.1. Getting data from modern web services

Currently, the most common way to obtain a variety of data provided by different web-services is to use an API — application programming interface. This API is usually follows the principles of REST. As example, here is a link to the documentation of Facebook, VK and Scopus [16] APIs. In General, the work with the API of such web services can be roughly described by the following points.

- HTTP requests are made to different URLs specified in the documentation for developers. Each URL represents a particular resource from which programmer can get information (for example, user profile data, friends list, list of co-authors, etc.).
- HTTP requests are usually made with GET and POST methods. In response to a request, the server returns a document with data in XML (less often) or JSON (more often) format. The structure of the document is known from service documentation, so its analysis can be automated and the necessary information can be extracted for further analysis.
- There is often more flexibility to customize the query using the additional parameters for GET requests or passing them using POST requests.
- Often, access to services requires an access key, which should be passed with each request as a parameter value or inside the HTTP request header.

In the process of obtaining data, one often needs to perform a large number of requests to the web service. To automate this process, it is convenient to use libraries that simplify the formation of HTTP requests. Such libraries are available for almost all programming languages. For the **Python** language, the **Requests** [17] is the de facto

standard for generating HTTP requests. `Requests` has become so popular that libraries began to appear for other languages inspired by it. The standard utility for Unix systems `cURL` [18] and the program `Postman` [19] will also may be useful. `Postman` has a graphical interface and allows one to save and document the requests that are made, which is especially useful at the stage of studying the API of some service.

Note also that for common web services, there are often already official libraries that make it easier to write queries.

4.2. Graph presentation formats

At the moment there is a wide variety of formats that can be used to store graph data. They can be classified as follows.

- Binary formats, usually their own for each specific library or program.
- Text formats, which can also be divided into three subgroups.
  - Universal, such as CSV (Comma-Separated Values — simple tabular data format), JSON, YAML.
  - XML-based, such as GEXF (Graph Exchange XML Format) [20] and GraphML [21].
  - Specialized, such as DOT [14], GML [22] (Graph Modeling Language), GDF, etc.

In our opinion, it is optimal to use DOT language for small graphs, since it is supported by all the software tools we consider and it is convenient to edit files in this format manually. If one wants to store complex graph data where each node and edge can have additional attributes, the optimal solution is the GEXF format. To store extremely large graphs, it is more efficient to use a binary format, such as the one implemented in the [9] library.

5. Obtain information about co-authors from Scopus

5.1. Brief description of the Scopus API

The Scopus index service provides access to data not only through the classic web site interface, but also through the RESTfull API. This API can be used by developers to obtain and automate data processing. Interaction with the interface is carried out via HTTP in accordance with the concept of RESTfull. In response to requests, the server sends data in JSON or XML format. Both options are mostly interchangeable.

5.1.1. Getting API key

To interact with the Scopus API, requests must be made from a computer that has access to the Scopus itself and also an API key must be obtained. To obtain the key, one shell follow these steps:

- Create Elsevier user ID for Elsevier official site.
- Visit Elsevier Developers portal Registered API keys section and create new API key. To generate the key, one must fill all the fields of the web form. The first field is the key identifier (it can by any name by user chose). The second field indicates the address of the website from which the requests will be made. If the requests are performed from the local computer, then it is permissible to specify in this field any domain name, for example example.com. The third field will be filled automatically and in our use case it is not necessary to change.

The key is a 32-digit number in hexadecimal notation. Each time a request is made, this key must be passed in the HTTP header of the message as the value of the `X-ELS-APIKey` field. Also in the message header one should specify the format the server should send the response data. Two formats are supported: `application/json` and `text/xml`.

5.1.2. Basic Scopus API methods

The HTTP GET method is used for the requests. The API itself is divided into three groups

1. Interface for general search in the Scopus data base. Consists of three methods
   (a) An affiliation search `https://api.elsevier.com/content/search/affiliation`.
   (b) Authors search `https://api.elsevier.com/content/search/author`
   (c) General search in Scopus data base `https://api.elsevier.com/content/search/scopus`.
2. The interface to receive specific data. It also consists of three main methods and a number of more specific sub-methods.
   (a) Getting annotations `https://api.elsevier.com/content/abstract`.
   (b) Getting affiliations `https://api.elsevier.com/content/affiliation/affiliation_id`
   (c) Getting authors profile data `https://api.elsevier.com/content/author`
3. Interface for obtaining various scientometrics data.

5.1.3. General search

The most powerful and versatile is the search interface `https://api.elsevier.com/content/search/scopus`. It provides the same features as the advanced search page of the Scopus website.

To obtain data one should send the HTTP GET message at `https://api.elsevier.com/content/search/scopus`. The HTTP header of the message should include two fields

```
GET /content/search/scopus HTTP/1.1
Accept: application/json
X-ELS-APIKey: cab65b347d214c41bd54e54bb74ce085
```

Let's list some GET request parameters that we will use.

- The `query` parameter is used to send an extended search query. Its value must be a string containing field codes and boolean operators. The string must be url-encoded. For example, if we want to find all articles that have the word `Numeric` associated with RUDN in their title, we should make the following query

  ```
  title(Numerical) and affil(rudn)
  ```

  first line of `GET` request will by following `https://api.elsevier.com/content/search/scopus?query=title%28Numerical%29%20and%20affil%28rudn%29`. Parenthesis symbols «(» and «)» are in html-encoded form.
- The `view` parameter controls the amount of data that the server returns in response to a request. There are two options: `STANDARD` (default) and `COMPLETE`.
- The `start` and `count` parameters allow one to set the initial value for the result list, as well as the number of results returned. At a time, the server does not send all the search results, but only a certain amount, which depends on the user's access level to Scopus.

5.1.4. The author search

One can use a more specialized interface to search for data related to the authors of articles `https://api.elsevier.com/content/search/author`. Requests are also made by the GET method and one can pass the same parameters as for the general search. However, there are several specific parameters. Specifically, the `co-author` parameter allows to pass the author's Scopus ID and get a list of co-authors from the server in response. The `start` and `count` options also control the amount of data returned.

5.2. Using the requests module to get a list of all co-authors and information about them using the Scopus API

To get a list of co-authors of a particular author, one should make a `GET` request at `https://api.elsevier.com/content/search/author`. To make a `GET` request, we use the Python module `requests`, which we briefly described above.

Let us place the code, responsible for the request of co-authors in Python function.

```
import requests
def author_search(scopus_id, count_increment=20, start_offset=0):
    """Request for author's data"""
    headers = {
        'Accept':'application/json',
        'X-ELS-APIKey': MY_API_KEY
    }
    co_authors = "https://api.elsevier.com/content/search/author"
    params = {
        'co-author' : scopus_id,
        'count': count_increment,
        'start': start_offset,
        'field': ','.join(['dc:identifier',
                           'preferred-name',
                           'affiliation-current'])
    }
    result = requests.get(url=co_authors,params=params,headers=headers).json()
    return result
```

Let's analyze this function. The variable `MY_API_KEY` contains the API key that we received after registration. It should be passed every time in the `X-ELS-APIKey` field of the request header. Both header fields should be formatted as a dictionary (standard Python data type). We also pass four additional `GET` parameters, also formatting them as a dictionary.

- The value of `co-author` must be the Scopus ID of the author whose co-authors we want to get.
- Parameters `count` and `start` allow us to control how many items from the list of co-authors the server will send us in response to the request and from which item of this list it will be counted. The server sends a limited number of items at a time (usually 25), so if the author has a large number of co-authors, we have to repeat the requests several times to get all the co-authors.
- The `field` parameter takes a value as a string and allows us to list the data fields we want to retrieve. By specifying only the required fields, we reduce the size of forwarded messages and save time and traffic. In our example, we have specified three fields because we want to get the Scopus ID of the co-authors, their names and affiliations.

Finally, we use the `get` function to make the request, passing it all the necessary parameters. This function returns a special object that contains all information about the request and the response. This object has a method `json`, which can be used if the server sent a JSON document in the body of the HTTP message. After calling this method, the JSON data will be converted into a python dictionary and then we can work with it using standard python capabilities.

The document sent by the JSON server contains basic data about each co-author, but there is no data about the author's Hirsch index. To obtain the Hirsch index, we have to make a request to `https://api.elsevier.com/content/author/author_id/{scopus_id}` where `{scopus_id}` is the specific Scopus ID of the author, information about we want to get. The request itself is similar to the previous one. There are no specific `GET` parameters however, one can specify `field` value `h-index` to make the server sent only the h-index.

5.3. Analysis of the obtained data

After receiving all the necessary data, it should be processed. The JSON returned by the server has a standardized structure and does not change from request to request. Consider in detail the JSON document that returns the server in response to the request of the list of co-authors.

The results represent the search performed by the server. The response are contained in the key value `search-results`. Let's list some keys whose values we will need.

- Key `opensearch:totalResults` contains the total number of co-authors found.
- Key `opensearch:itemsPerPage` contains the number of items (i.e. co-authors) sent per request. This number is controlled by the count parameter, but does not exceed 25 for standard access level to the Scopus database.
- Key `opensearch:Query` contains information about the search query using the advanced search syntax in the Scopus database.
- Key `entry` contains a list of found objects. This is the basic data we need.

Each item in the `entry` list has the following structure.

```
{
  "@_fa": "true",
  "prism:url": "https://api.elsevier.com/content/
                author/author_id/1215131351",
  "dc:identifier": "AUTHOR_ID:1215131351",
  "preferred-name": {
      "surname": "Name",
      "given-name": "Name2",
      "initials": "M."
  },
  "affiliation-current": {
      "affiliation-url": "https://api.elsevier.com/content/
                         affiliation/affiliation_id/1546431",
      "affiliation-id": "1546431",
      "affiliation-name": "Some University",
      "affiliation-city": "SomeCity",
      "affiliation-country": "SomeCountry"
}
```

Since we point out that we want to get only Scopus ID (key `dc:identifier`), name (key `preferred-name`) and affiliation (key `affiliation-current`), then in response we get truncated document and many fields are absent.

Analysis of the above data structure is carried out by standard Python tools. It should be noted that there are records in which the authors do not have names and/or affiliations. These exceptional cases should be handled using the `try/except` construct.

All of the above steps produces the function `get_coauthors`. As a result of its work, it returns a dictionary, where the keys are Scopus IDs of co-authors, and the values of the dictionary, which contains information about the co-authors: name, affiliation and h-index.

The function `get_coauthors_tree` uses the function `get_coauthors` in order to obtain information about the co-authors of co-authors. Requests are made in a loop and have the opportunity to obtain data with arbitrarily large nesting. However, the time taken to obtain this data seems to be growing as $n!$, where n — the number of co-authors of each author (assuming for simplicity that it is the same).

The data obtained by the `get_coauthors_tree` function can be optimally organized using two dictionaries.

- the first dictionary is an adjacency list and contains the user IDs as keys and the `set` of their co-authors IDs as values. The use of the set is justified by the fact that the same co-author can not be included in the list more than once, and the order of the co-authors does not matter.

– the second dictionary contains data about all the authors we met as a result of queries. The dictionary is similar to the dictionary returned by the `get_coauthors` function.

The separation of the data for the two structures allows to avoid duplication of data, since one author may be coauthor of several people.

We also point out optimized method to get the author's Scopus ID. Since we only need the id of the co-authors, we can extract them from the @searchTerms. In this field server store the search request format:

`au-id(8783969400) OR au-id(35194130800) OR au-id(50462435500) etc..`

So we can extract all the id of the co-authors from the search query, thus we have to make only one request to get all co-authors IDs. Also we can reduce amount of transferring data by setting parameter `count` equal to 1 (0 does not work because it is returned 25 co-authors, i.e. default number).

The main work will be done by the following peace of code:

```
res = response.json()

query = res["search-results"]["opensearch:Query"]["@searchTerms"]
regex = r"([0-9]+)"
prog = re.compile(regex, flags=re.I | re.M)
return set(int(item.group(0)) for item in re.finditer(prog, query))
```

We use simple regular expression to extract only ID number and discard unnecessary characters, so we need to import the standard `re` module.

5.4. Building the graph

After receiving and organizing all required data, let us consider how to represent it in the form of a graph. To do this, we use the `networkx` module. Assume that the `coauthors_tree` dictionary contains a adjacency list of authors, and the `co_authors` dictionary contains information about each author. The following code creates a graph from the adjacency list.

```
import networkx as nx
G = nx.Graph(coauthors_tree)
for key, value in co_authors.items():
    print(key, value)
    G.add_node(key, **value)
```

The `add_node` method allows to add additional attributes to each node (vertex). Scopus ID is used as the node ID. By iterating through the dictionary `co_authors` we add to each node required attributes: name, affiliation and h-index.

After the graph is formed, it can be exported to an external file. The following method allows us to save the graph in the `gexf` format and supports the attributes of nodes and edges. A fragment of the resulting file with data of our graph is given below.

`nx.write_gexf(G, path='sevast_graph.gexf', encoding='utf-8')`

The format of the `gexf` is `xml` based and supports the attributes of the nodes.

```
[autogobble,breaklines=true,breakanywhere]{xml}
<?xml version='1.0' encoding='utf-8'?>
<gexf version="1.2">
  <graph defaultedgetype="undirected" mode="static" name="">
    <attributes class="node" mode="static">
      <attribute id="0" title="name" type="string" />
      <attribute id="1" title="affiliation" type="string" />
      <attribute id="2" title="h-factor" type="string" />
    </attributes>
    <meta>
      <creator>NetworkX 2.0</creator>
      <lastmodified>12/03/2019</lastmodified>
    </meta>
    <nodes>
```

```
      <node id="6603101968" label="6603101968">
        <attvalues>
          <attvalue for="0" value="Some Author" />
          <attvalue for="1" value="Some University" />
          <attvalue for="2" value="3.0" />
        </attvalues>
      </node>
      <!-- other nodes -->
    <edges>
      <edge id="0" source="6603101568" target="35227873700" />
      <!-- other edges -->
    </edges>
  </graph>
</gexf>
```

Inside the `attributes` tag, all attributes that nodes and edges can have are listed. Each of them is assigned an ID. Then, inside the `nodes` tag, all nodes (the `node` tag) and node attribute values (the `attvalues` tag) are listed. Inside the `edges` tag, all edges are listed, as a pair of node IDs that a particular edge links.

6. Conclusion

In this paper, we gave an overview of the software tools that were used in our work. In our opinion, this review will be useful for beginners who just transgress the study of this area and wanted to move as quickly as possible to the analysis of real data. The example of working with the Scopus API can be used as an introduction to the Scopus API documentation.

Acknowledgments

The publication has been prepared with the support of the "RUDN University Program 5-100" (Migran N. Gevorkyan, Anna V. Korolkova, calculation model development). The reported study was funded by Russian Foundation for Basic Research (RFBR), project number 19-01-00645 (Dmitry S. Kulyabov, research algorithm development).

References

1. G. Rossum, Python reference manual, Tech. rep., Amsterdam, The Netherlands, The Netherlands (1995).
2. Python home site (2019).
   URL `https://www.python.org/`
3. J. VanderPlas, Python Data Science Handbook, O'Reilly, Beijing Boston Fernham Sebastopol Tokyo, 2018.
4. M. Bonzanini, Mastering Social Media Mining with Python, Packt Publishing, Birmingham-Mumbai, 2016.
5. Project jupyter home (2017).
   URL `https://jupyter.org`
6. Networkx — software for complex networks (2019).
   URL `https://networkx.github.io/`
7. A. A. Hagberg, D. A. Schult, P. J. Swart, Exploring network structure, dynamics, and function using networkx, in: G. Varoquaux, T. Vaught, J. Millman (Eds.), Proceedings of the 7th Python in Science Conference, Pasadena, CA USA, 2008, pp. 11–15.
8. Matplotlib home site (2019).
   URL `https://matplotlib.org/`

9. Graph-tool — efficient network analysis (2019).
URL https://graph-tool.skewed.de/
10. Boost — c++ libraries (2019).
URL https://www.boost.org/
11. Igraph — the network analysis package (2019).
URL http://igraph.org/redirect.html
12. Networkit — large-scale network analysis (2019).
URL https://networkit.github.io/
13. D. Zinoviev, Complex Network Analysis in Python, The Pragmatic Bookshelf, Raleigh, North Carolina, 2018.
14. Graphviz — graph visualization software (2019).
URL https://www.graphviz.org/
15. Gephi — makes graphs handy (2019).
URL https://gephi.org/
16. Scopus search api (2019).
URL https://dev.elsevier.com/documentation/ScopusSearchAPI.wadl
17. Requests: HTTP for Humans (2019).
URL http://docs.python-requests.org/en/master/
18. Curl – command line tool and library (2019).
URL https://curl.haxx.se/
19. Postman – api development (2019).
URL https://www.getpostman.com/
20. Gexf file format (2019).
URL https://gephi.org/gexf/format/
21. The graphml file format (2019).
URL http://graphml.graphdrawing.org/
22. Gml: A portable Graph File Format (2019).
URL https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf