
UNIVERSITÀ
degli STUDI
di CATANIA

Dipartimento di Ingegneria Elettrica Elettronica Informatica

**CORSO DI LAUREA IN INGEGNERIA INFORMATICA**

# Neo4j algorithms for clustering and hubs detection in Twitter

Academic Year 2019/2020

Candidate: Giovanni Castellano

Advisor: prof. Antonella Di Stefano

Co-advisor: eng. Andrea Di Maria

*To Graziano,*

*for all the smiles and hugs you gave me.*

*I miss you.*

# ABSTRACT

The Twitter APIs allows us to retrieve from Twitter the information needed to construct a graph representing how users follow each other. Using a graph database it is possible to quickly perform interesting analyses about the topology of the network. In this thesis, we give a detailed description of how to construct the graph, and we show how to detect clusters and hubs using Neo4j.

# CONTENTS

INTRODUCTION ..... 1

1. BACKGROUND ..... 2

1.1. SOCIAL NETWORKS ..... 2

Brief history ..... 2

Twitter ..... 3

1.2. GRAPH THEORY ..... 4

Graphs ..... 4

Complex networks ..... 6

Clusters ..... 6

Hubs ..... 8

2. NEO4J ..... 9

2.1. OVERVIEW ..... 9

2.2. CYPHER ..... 9

2.3. NEO4J DRIVERS ..... 11

2.4. GRAPH DATA SCIENCE LIBRARY ..... 11

Graph catalog and projections ..... 12

3. OTHER TOOLS ..... 13

3.1. TWITTER API ..... 13

Rate Limit ..... 13

3.2. PYTHON ..... 14

Tweepy ..... 14

Neo4j drivers ..... 15

3.3. GRAPHISTRY ..... 16

Neo4j and Graphistry ..... 16

4. DATA COLLECTION ..... 18
4.1. CHOICE OF DATA ..... 18
How to choose the main node? ..... 18
4.2. GRAPH CONSTRUCTION ..... 20
4.3. PREPROCESSING ..... 23
Tarjan's algorithm ..... 23
Strongly connected component on Neo4j ..... 25
4.4. RESULTS ..... 25
5. CLUSTERING ..... 27
5.1. LOUVAIN ALGORITHM ..... 27
Configuration model ..... 27
Modularity ..... 27
Description of the algorithm ..... 29
Execution ..... 29
5.2. LABEL PROPAGATION ALGORITHM ..... 30
Execution ..... 31
5.3. DATA EXTRACTION ..... 31
5.4. RESULTS ..... 33
6. HUBS DETECTION ..... 38
6.1. PAGERANK ..... 39
Description ..... 39
Execution ..... 41
6.2. DATA EXTRACTION ..... 41
6.3. RESULTS ..... 43
7. CONCLUSIONS ..... 45
ACKNOWLEDGMENTS ..... 46

LIST OF FIGURES .....47
LIST OF CHARTS .....48
LIST OF TABLES .....49
REFERENCES .....50

# INTRODUCTION

The large amount of public data that people generate using social networks, pave the way for many scientific studies that can be performed to understand the laws of sociology or for practical applications such as security and marketing. An interesting branch of social network analysis is the study of graphs that represent how users interact with each other. Questions that find their answers in the topology of this data structure are: which users share the same interests? Which users contribute most to the spread of information? In broad terms, these questions refer to the concepts of Clusters and Hubs. In this thesis, we show how to collect data from Twitter, and in particular, the information needed to represent how users follow each other. We use the graph database Neo4j to identify hubs and clusters in the network. A detailed description of the contents of this book is provided in the following.

In §1 we introduce social networks and graph theory, which are the main concepts on which this thesis is based. In the same chapter, we also give a brief introduction to the concepts of cluster and hub. The description of the graph database Neo4j is the subject of §2. All other tools that are used during this work are then presented in §3. In particular, we discuss the programming language Python, the Twitter APIs, and the graph visualizer Graphisty. §4 provides a detailed description of the work, showing how to retrieve and store the data using the Twitter APIs and Neo4j. An important step of this chapter is the preprocessing of data, which is necessary to proceed towards the detection of clusters. This phase is detailed in §5, which presents the Label Propagation algorithm and the Louvain Modularity algorithm. Then, §6 focuses on the detection of hubs, which is based on the identification of the most important nodes of each cluster. For this purpose, we present the PageRank algorithm. Finally, §7 concludes the thesis.

# 1. BACKGROUND

In this chapter, we give a brief introduction to social networks, with a particular focus on Twitter, which is the protagonist of our work. Considerations about social networks analysis lead us to the description of graphs, and to the introduction of the important concepts of cluster and hub, which are then detailed in §5 and §6.

## 1.1. SOCIAL NETWORKS

A social network is an online platform that people use to interact with each other sharing information. Kinds of contents that users usually share through socials include photos, videos, current position, relationship status, personal interests, and a myriad of other information.

It is interesting to think about the impact of social networks on our society. Today, almost every politician uses socials to spread information during election campaigns or to announce official decisions about any kind of topic. Not to mention journalism: nowadays most of the newspapers of the world have a page on more than one social network, allowing, once again, rapid diffusion of information or, sometimes, misinformation.

The description of this scenario inevitably leads us to a question: how did we come to this point?

### Brief history

The history of social networks as we know them today begins in 1989 when the WWW (World Wide Web) commonly known as “the Web” was born. Since then, the world of networks has started evolving rapidly. Thanks to the quick diffusion of network infrastructures across the world, and the spread of personal computers, a lot of users were able to access an always larger amount of information with just some clicks using a simple browser.

With the advent of dynamic web pages, any kind of service started to come out, among which, the firsts online communities such as theglobe.com (1995), GeoCities (1994) and tripod.com (1995).[1] Concepts like user-profiles and friends list became a central feature of these web sites.

In 1997 Microsoft launched the first instant messaging application, a kind of service that today is present on every social network. In 2003 a milestone was inaugurated: Myspace. This community aimed to encourage artists to promote themselves posting photos, text, adding comments, following other users, and managing their profiles.[2]

In 2004 Mark Zuckerberg created Facebook. The initial goal of the application was to connect students at Harvard University; however, in 2006, Facebook became a service open to the public. The always larger amount of personal information that Facebook allowed us to publish on our profiles led it to a rapid diffusion over the world and, therefore, to the launch of many new social networks such as Twitter (2006), Instagram (2010), and Snapchat (2011).

### Twitter

Twitter is an American social network created in 2006 and today it is one of the most used in the world.

Here we list some important concepts of Twitter that are useful to compare it with other social networks and to define the nomenclature that is used in the next pages:

- **Tweet:** it is the main content of the social, it was born as a simple text with a limited length that a user could share with other users. Today it is also possible to add photos or videos to the text.
- **Follower:** all users can follow another user, if they do it, they become his followers and they will be able to visualize his tweets on their homepages.
- **Friend:** we refer to users that are followed by another one as his “friends”.

[1].https://en.wikipedia.org/wiki/Social_networking_service
[2].https://www.antevenio.com/usa/a-brief-history-of-social-networks/

On Twitter, the relationship between two users can have two different directions. Therefore, the concept of friendship, as we can think it on Facebook, does not exist. The fact that a user is interested in another one, does not necessarily mean that the interest is returned. This information is of fundamental importance for our analysis and leads us to choose a directed graph as a data structure to carry out our work.

## 1.2. GRAPH THEORY

Graphs are probably the most important and complex data structures. Many practical problems can be represented by graphs; therefore, they find applications in a lot of fields like computer science, chemistry, physics, biology, and sociology.

### Graphs

Some important definitions:

- **Graph:** a graph (or undirected graph) is a pair $G = \{V, E\}$ where $V$ is a set of elements called vertices (or nodes) and $E$ is a set of unordered pairs called edges (or links) where each pair contains elements of $V$.

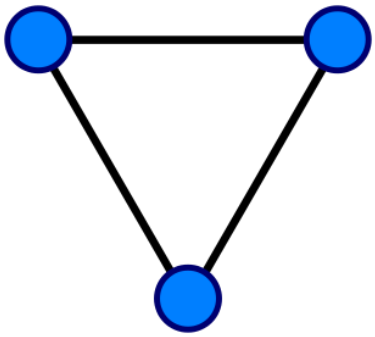

**Figure 1.1. An undirected graph with 3 vertices and 3 edges.**[3]

[3] https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

- **Directed graph:** a directed graph is a graph where elements of $E$ are ordered pairs rather than unordered.

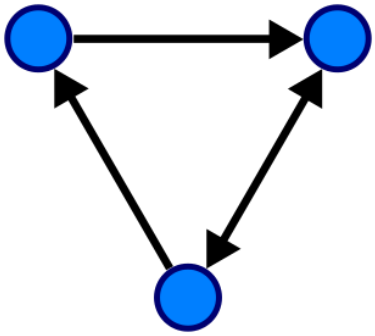

**Figure 1.2. A directed graph with 3 vertices and 4 edges.**[4]

- **Weighted graph:** a weighted graph is a graph in which a number is assigned to each edge.
- **Regular graph:** a regular graph is a graph in which each vertex has the same number of neighbors.
- **Complete graph:** a complete graph is a graph in which each pair of vertices are joined by an edge.
- **Connected graph:** a connected graph is an undirected graph in which every pair of vertices is connected.
- **Strongly connected pair:** in a directed graph, an ordered pair of vertices $(x, y)$ is said to be strongly connected if a directed path leads from $x$ to $y$. The pair is instead called **weakly connected** if an undirected path leads from $x$ to $y$.
- **Strongly connected graph:** a strongly connected graph is a graph in which each ordered pair of vertices is strongly connected.
- **Random graph:** A random graph is a graph obtained by starting with a set of isolated vertices and adding successive edges between them at random.

[4] https://en.wikipedia.org/wiki/Directed_graph

Each one of these concepts has particular importance in different fields of application. Weighted graphs, and connected graphs, for example, arise in contexts where it is necessary to find the shortest path from a node to another one. Random graphs instead are very useful when we want to compare graphs modeling real phenomena, with graphs generated randomly or following some probabilistic distribution.

### Complex networks

In network theory, a complex network is a graph that is neither regular nor random; in other words, it is a graph with non-trivial topological features. The reader who is interested in a more detailed definition of Complex Network can read Da Fontoura Costa, L., 2020.

Empirical studies (Albert and Barabási, 2002) show that networks describing real phenomena such as biological networks, brain networks and social networks can be classified as complex networks. The analysis of these kinds of graphs lead to define two properties that are common to most of them:

- **Scale-free network:** a scale-free network (Albert, R. and Barabási, A.L., 2002) is a network where the probability that a node has a certain number of links decreases as the number of links increases:

  $$P(k) \sim k^{-\gamma},$$

  where $P(k)$ is the fraction of nodes with $k$ links, and $2 < \gamma < 3$.
- **Small-world network:** A small-world network (Watts, D.J. and Strogatz, S.H., 1998) is a network where the distance $L$ between two randomly chosen nodes grows proportionally to the logarithm of the number of nodes $N$ in the network.

### Clusters

Imagine to be a user of some social media like Twitter, and to follow other users; how likely are these users to follow each other? Previous studies (Zachary, W.W., 1977) shows how graphs describing social networks tend to assume a community structure, that is, we can divide the graph into some subgraphs where each of them has a high density of links, and the density of links between two subgraphs is

relatively low (Gregory, S., 2007). These subgraphs are called clusters, and algorithms used to identify them are called clustering algorithms. The analysis of clusters in a social network is useful to identify which users are more likely to get in touch in the physical world, or which users share the same interests.

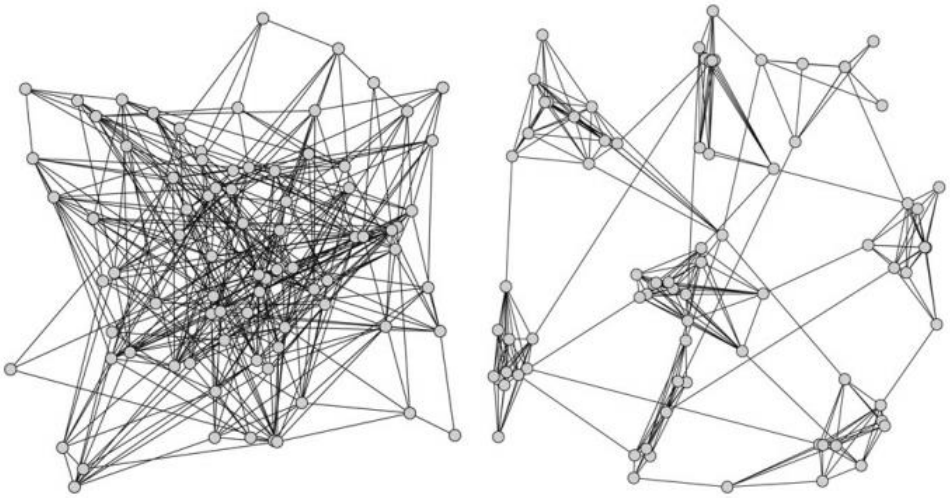

**Figure 1.3. A random graph (left) and a clustered graph (right). Schaeffer, S.E., 2007.**

The definition of cluster given above is very abstract, to better visualize the concept, we distinguish different types of clustering:

- **Exclusive vs non-exclusive:** in an exclusive clustering a node can belong just to one cluster.
- **Partial vs complete:** in a complete clustering, each node belongs at least to a cluster.
- **Heterogenous vs homogenous:** in a heterogeneous clustering, clusters can have very different sizes, shapes, and density.

A measure of how nodes tend to cluster together is given by the **clustering coefficient**. Two versions of the measure exist; for simplicity, we give the definitions for an undirected graph:

- **Local clustering coefficient:** the local clustering coefficient (Watts, D.J. and Strogatz, S.H., 1998) of a node $i$ in a graph $G = \{V, E\}$, quantifies how close its neighbors are to form a complete graph:

$$C_i = \frac{2\left|\{e_{jk}: v_j, v_k \in N_i, e_{jk} \in E\}\right|}{k_i(k_i - 1)},$$

where $N_i$ is the set of all neighbors of $i$ and $k_i$ is its cardinality.

- **Global clustering coefficient:** the global clustering coefficient (Barrat, A. and Weigt, M., 2000) is based on triplets of nodes. A triplet is three nodes that are connected by either two (open triplet) or three (closed triplet) links:

$$C = \frac{\text{number of closed triplets}}{\text{number of all triplets (open and closed)}}.$$

An alternative to the global clustering coefficient is the average of the local clustering coefficients of all nodes in the graph (Watts, D.J. and Strogatz, S.H., 1998):

$$\bar{C} = \frac{1}{n}\sum_{i=1}^{n} C_i.$$

### Hubs

During the analysis of a graph describing how users interact in a social network, it is interesting to understand which nodes are the most important ones. But when a node can be considered important? Users with a lot of followers, for example, have particular importance because they can easily spread information across the network. The number of incoming links of each node, also called degree, can therefore be a good measure. Another idea could be to use the PageRank algorithm (Page, L. et al., 1999) used by Google to identify the most important pages on the web. This algorithm takes into consideration not only the number of incoming links but also whether those links come from important nodes or not.

The algorithms described so far effectively detect only certain categories of important nodes. In clustered networks, however, particular importance is given to nodes that connect different clusters. We refer to these nodes as hubs. Note that many different definitions of a hub can be found in the literature.

# 2. NEO4J

To store our data, we decide to use Neo4j, which is the most popular graph database.[5] Neo4j was built to handle highly connected data, and its power lies in its flexibility in managing graphs with non-trivial topology. This makes it perfect to work with a complex network.

## 2.1. OVERVIEW

The most important concept to introduce in Neo4j is the concept of graph, and with it, nodes, and edges. A detailed description of this data structure is already given in §1.2; therefore, here we directly describe more specific elements of the database:

- **Label:** it is used to describe what a node represents; we can imagine that all nodes of the graph are divided into some sets and each of them is identified by a string called label.
- **Property:** it is a name-value pair that is used to store information inside a node or an edge. Each node or edge can have any number of properties. E.g., when considering a node that represents a person, we can imagine having properties such as name, surname, date of birth, and more.
- **Relationships type:** each edge of the graph has exactly one type which describes the relationship between the two nodes that it links.

## 2.2. CYPHER

Cypher is the query language of Neo4j. It can be used to describe a pattern representing a subgraph (or even the entire graph) of our database and to perform

[5] https://db-engines.com/en/ranking/graph+dbms

operations on it. Note that a subgraph could also be a simpler data structure like just a node, a tree, or an open path. We show some basics queries that are essential to understand the operations performed in the remainder of this thesis.

The following query returns all the nodes of the database with all their properties:

```
1. MATCH(n) RETURN n
```

We can also specify the label of nodes that we want to select and the list of needed properties:

```
1. MATCH(User:n)
2. RETURN n.username, n.name, n.surname
```

In this case, we get just the nodes of our graph that represents users, and for each of them, only those three properties. In big databases with a lot of nodes and properties, it is important to specify the properties that we need; otherwise, the execution of the query could take a lot of time.

Suppose that we want to get just nodes representing users whose name is “Giovanni”:

```
1. MATCH(User:n) WHERE n.username = “Giovanni”
2. RETURN n
```

This query can also be written as:

```
1. MATCH(User:n {username: “Giovanni”})
2. RETURN n
```

This last notation is used very often in this thesis. Imagine now that we want not to get just nodes but also relationships:

```
1. MATCH
2. (User:n {username: “Giovanni”})
3. -[r:FOLLOW]->
4. (User:m {username “Giuseppe”})
5. RETURN n,m,r
```

The query above returns a structure formed by two nodes and one relationship for each user “Giovanni” who follows any user named “Giuseppe”. Observe that, in

this case, “FOLLOW” is the type of the relationship and that the direction of the arrow represents the direction of the edge (we are working on a directed graph).

What if we want to add or edit a property of a node? We just need to replace the “RETURN” clause with the “SET” clause:

```
1. MATCH (User:n {username: “Giovanni”})
2. SET n.age = 22
```

Other interesting clauses can be used to perform a lot of different operations. In the next chapters, we deal with a series of more complex queries, which give us the possibility to understand more complex structures of this language.

## 2.3. NEO4J DRIVERS

The possibility to establish a connection between a database and a programming language is an important feature of each DBMS. In database terminology, we refer to the components needed to implement this feature as drivers. Neo4j provides drivers for the most popular programming languages. §3.2 provides a detailed description of Python drivers for Neo4j. For now, we just give an overview of how this communication takes place.

Neo4j supports either local or remote communications. The data is exchanged over a connection-oriented protocol called Bolt. As the driver is initialized, a pool of connections is established. Every time that a communication is needed, a component called session borrows a connection and uses it to execute one or more transactions on the databases. As the communication ends, the connection is freed and made available for another session.

## 2.4. GRAPH DATA SCIENCE LIBRARY

The combination of the Cypher language with the Neo4j drivers allows us to easily implement any kind of algorithm to execute on the graph. Neo4j, however, also

provides already implemented algorithms in its GDS library, e.g., community detection algorithms, centrality algorithms, similarity algorithms, and many others.

### Graph catalog and projections

To run algorithms as efficiently as possible, the GDS library creates a view of the graph in the main memory. In Neo4j terminology, this is also called a projection. There are two ways to describe which part of the graph we want to project:

- **Native projection:** the projected graph is specified in terms of labels, relationship types, and properties.
- **Cypher projection:** the projected graph is specified using the Cypher language.

The component of the database that allows us to manage more projections at the same time is called Graph Catalog. In the next chapters, we detail the execution of some GDS algorithms. Other information about Neo4j can be found in the official documentation.[6]

[6] https://db-engines.com/en/ranking/graph+dbms

# 3. OTHER TOOLS

This chapter provides a brief description of all the other tools used during this work. The following is not indeed as a general overview, as we focus only on the aspects of these tools that are related to the thesis.

## 3.1. TWITTER API

Twitter makes available some APIs (application programming interfaces) which can be used to easily collect public data from the social network. The service operates over HTTP. To exploit the APIs, it is enough to just sign up for the website, register a new application, and store all the keys needed for the authentication. The information about any possible request, the list of all the parameters needed for each of them, and the correct HTTP method to use, can be found on the official documentation.[7]

The HTTP requests needed to interact with the APIs require many parameters, often resulting in difficult request compositions. Luckily, there are many libraries for the most important programming languages that allow us to compose the desired requests very easily, without directly operate on the HTTP protocol. One of these libraries is later introduced in §3.2.

### Rate Limit

Unfortunately, the Twitter APIs are subjected to a limit on the number of requests that can be made. When this limit is reached, it is necessary to wait for a specific time interval to keep requesting data. This limit represents a big obstacle faced during this work and, despite the strategies adopted having in mind this limitation (§4.3), inevitably led to a loss of accuracy of the analysis.

[7] https://developer.twitter.com/en/docs

## 3.2. PYTHON

Python is a general-purpose programming language released in 1991 that aims to simplify software development as much as possible. The simplicity and the high-level features of Python make it one of the best languages to quickly write any kind of script. Given its flexibility and the large variety of available libraries, Python ended up to be the best choice for carrying on the work described in this thesis.

### Tweepy

Tweepy is an open-source Python library that gives us a very convenient way to access the Twitter API. It provides functions for each of the requests that can be made, and the parameters that we need to pass are much less than those needed to directly send the HTTP packet.

The library can be installed in different ways. We use the tool *pip*:

```
1. pip install tweepy
```

The following script sends a simple request to retrieve all tweets on our homepage. First, we need to authenticate (all necessary data is provided by Twitter):

```
1. import tweepy
2.
3. consumer_key = "consumer_key"
4. consumer_secret = "consumer_secret"
5. access_token = "access_token"
6. access_token_secret = "access_token_secret"
7. auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
8. auth.set_access_token(access_token, access_token_secret)
9. api = tweepy.API(auth)
```

we proceed sending the request and printing the results:

```
1. public_tweets = api.home_timeline()
2. for tweet in public_tweets:
3.     print(tweet.text)
```

### Neo4j drivers

In §2.3 the concept of a driver is introduced, however, no practical example of how to use them is shown. We show here, how it is possible to perform some queries on a Neo4j database using Python. The library that we need to install is neo4j:

```
pip install neo4j
```

This is our script:

```
from neo4j import GraphDatabase

class neo4jInterface:
    def __init__(self, uri):
        self.driver = GraphDatabase.driver(uri)

    def close(self):
        self.driver.close()

    def count_users(self, age):
        with self.driver.session() as session:
            n_users = session.write_transaction(self.run_count_us
ers, age)
            return n_users

    @staticmethod
    def run_count_users(tx,age):
        result = tx.run(
            "MATCH (a:User {age: $age}) RETURN  count(a)"
            , age=age)
        return result.single()[0]

db = neo4jInterface("bolt://localhost:7687")
n_users = db.count_users(21)
print(str(n_users))
community.close()
```

We are trying to perform a query that returns the number of users with a certain age. The class *neo4Interface* contains a constructor that starts the driver. Every time that a new object of this class is instanced, a pool of connections is established through the specified database. The method invoked to execute the query is count_users(). It takes as parameter the age that we want to specify, and after creating a session, it passes the parameter to run_count_users(). This last method deals with the execution of the transaction. The connections to the database are closed using the method close(). The class *neo4jInterface* is widely used in the next chapters.

## 3.3. GRAPHISTRY

An important help during the study of any network is the visualization of its graph. Neo4j already includes a tool to visualize the data returned by a query, however, the low flexibility of its interface makes it very hard to spot interesting information.

Graphistry is a platform thought to provide an interactive visualization of large graphs. To start using it, it is necessary to create an account on its official website[8]. From here, we can manage the catalog of our graphs, moreover, a personal key is provided. The procedure to upload a new graph to our catalog depends on how the graph is stored. We detail how to upload the data from Neo4j, but similar procedures can be used for any other graph database.

### Neo4j and Graphistry

To interface Graphistry with Neo4j we write a simple Python script called *graphistry.py*. The first step is installing a new library, also in this case we use pip:

```
1. pip install -user graphistry
```

we proceed by importing the library and defining two dictionaries. The first one contains the URL of our Neo4j database, the second one instead, contains information about the Graphistry server that we are using to process our data. Here, we need to write the key provided during the registration:

```
1. import graphistry
2.
3. NEO4J = {
4.     'uri': "neo4j://localhost:7687",
5. }
6.
7. GRAPHISTRY = {
8. 'server': 'hub.graphistry.com',
9.     'api': 3,
10.    'key': 'personal key'
11.}
```

now we have everything that is needed to authenticate to the server:

[8] https://db-engines.com/en/ranking/graph+dbms

```
graphistry.register(api=3, protocol="https",
server="hub.graphistry.com", username="myUsername",
password="myPassword")
graphistry.register(bolt=NEO4J, **GRAPHISTRY)
```

where the username and the password are the same used during the registration. The last step, is to write the query and plot the result that will be visualized using the default browser:

```
g = graphistry.cypher("""
MATCH (n)-[s:FOLLOW]->(m) RETURN s,m,n
""").bind(source='').bind(destination='')
g.plot()
```

These last lines also upload the graph to our catalog. In this way, we do not need to re-execute the script to visualize again the graph.

# 4. DATA COLLECTION

The work described in this book consists of the application of clustering and hubs detection techniques on a graph representing how Twitter users follow each other. We show now, how to collect the data.

## 4.1. CHOICE OF DATA

The huge number of people that today use Twitter makes it difficult to carry out a study on its whole user's network. To create our graph, we limit to a two-layers network: we start from a user and we take all its followers, and for each of them, we do the same. To limit the size of the final graph, first layer followers with more than 50.000 users are skipped. For clarity, some definitions are necessary:

- **Main node:** the node representing the user that is chosen to start the construction of the graph.
- **First layer followers:** all followers of the main node.
- **Second layer followers:** all followers of the first layer followers.

Proceeding in this way, we get a subset of all Twitter users, with a subset of information about how these users follow each other. What we lack is:

- Links between second layers followers.
- Links from second layers followers to the main node.
- Links from the main node to the second layers followers.
- Links from the first layer followers to the second layer followers.

The rate limit of the Twitter APIs does not allow us to retrieve all this information, therefore, we just add links described in the last two points.

### How to choose the main node?

The choice of the main node depends on which subset of the network we want to analyze. The goal of this work is just to have a general idea of how the network

behaves, we have no interest in collecting information about specific users. However, we must be aware that different kinds of users can lead to different topologies of the final graph. Another factor to consider is the number of followers of the main node, in fact, the larger this number the larger the graph.

An upper limit to the number of nodes of the final graph is given by the number of first layer users, plus the number of followers for each of them, plus one (the main node):

```
1. main_node = api.get_user('@userame')
2. n_followers = main_node.followers_count
3. upper_limit = n_followers + 1
4.
5. status = tweepy.Cursor(api.followers_ids, screen_name='@username'
   , tweet_mode="extended").items()
6. for i in range(0, n_followers):
7.     user = next(status)
8.     follower = api.get_user(user)
9.     fc = follower.followers_count
10.    if fc < 50000 and not follower.protected:
11.        upper_limit += fc
12. print('upper_limit: ', upper_limit)
```

In this code, the request “followers_ids” is used to retrieve the list of first layer users. For each of them, the property “followers_count” contains the number of followers. The method “tweepy.Cursor” helps us to iterate the returned data, it takes as parameters all information needed to make the request. Note that we skipped not just users with more than 50.000 followers but also, users with a private account, as we cannot get any information about them.

We do not explicitly write which user is chosen as the main node; however, we list some information about it:

- **Followers:** 404.
- **Friends:** 437.
- **Tweets about:** agriculture, deforestation, sustainable food, and animal feed.

For this user, the above-defined parameter has a value of about 800K.

## 4.2. GRAPH CONSTRUCTION

To import the graph on Neo4j, all data are previously stored in a CSV file where each record describes the relationship between two users. The only information stored about each user is his Twitter ID.

Now we show the essential parts of the script used to retrieve the list of first and the second layers followers. To begin with, the CSV file is created:

```
1. with open('followers.csv', 'w', newline='') as followers:
2.     followers_writer = csv.writer(followers)
3.     followers_writer.writerow(["followed", "follower"])
```

The request “followers_ids” is used to retrieve first layer followers that comply the conditions:

```
1. usrname = "@username"
2. for user in tweepy.Cursor(api.followers_ids, screen_name=usrname)
   .items():
3.     follower = api.get_user(user)
4.     fc = follower.followers_count
5.     if fc < 50000 and not follower.protected:
6.         first_level_followers_id.append(user)
7.         followers_writer.writerow([main_node.id, user])
```

Finally, all second layer followers are retrieved:

```
1. for user in first_level_followers_id:
2.     follower = api.get_user(user)
3.     for p in tweepy.Cursor(api.followers_ids, id=user).pages():
4.         for follower in p:
5.             followers_writer.writerow([user, follower])
```

The execution of the script leads to the following error:

tweepy.error.RateLimitError: [{'message': 'Rate limit exceeded', 'code': 88}]

What is happening? The Tweepy library launched an exception due to the high number of requests sent to Twitter. Reading the official documentation of the

Twitter API[9], we find out that the rate limit for the "followers_ids" request is 15 requests every 15 minutes, moreover, the maximum number of ids returned for each request is 5.000. This last number tells us that more requests are made for users with more than 5.000 followers. The only thing that is possible to do to avoid this exception is to slow down the execution of the script waiting between requests:

```
time.sleep(61)
```

In this way, a request is sent every 61 seconds, which is slow enough to respect the rate limit. When the execution ends, the CSV file is ready to be imported into the database.

Neo4j provides a specific command to import data from a CSV file. First, we load all nodes using these two queries:

```
USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///followers.csv" AS followers
MERGE (followed:User {twitter_id: followers.followed})

USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///followers.csv" AS followers
MERGE (follower:User {twitter_id: followers.follower})
```

The first query creates a node with the "twitter_id" property for each field in the column "followed" of the file. The second one instead, does the same with the column "follower". The "Merge" clause is used to create a node or a relationship only if it is not yet present in the database. We proceed by adding relationships:

```
USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///followers.csv" AS followers
Match (followed:User {twitter_id: followers.followed})
Match (follower:User {twitter_id: followers.follower})
MERGE(follower)-[:FOLLOW]->(followed)
```

The second step of the construction of the graph is adding relationships from first layer followers (and the main node) to second layers ones. To retrieve this information, we use the same script changing the request from "followers_ids" to

[9] https://developer.twitter.com/en/docs/twitter-api/v1/accounts-and-users/follow-search-get-users/api-reference/get-followers-ids

“friends_ids”. In this way, we get a new CSV file *friends.csv* containing all information that we need. However, this file also contains information about relationships with nodes that are not in our graph. We don’t want to add new nodes to the graph; therefore, we write a Python script to filter the information.

A list of the already present users is created:

```
with open('followers.csv') as csv_file:
    csv_reader = csv.DictReader(csv_file, delimiter=',')
    for row in csv_reader:
        if row['follower'] not in users:
            users.append(row['follower'])
    users.append("main-node-id")
```

all data into *friends.csv* are loaded:

```
csvdata = []
with open('friends.csv', 'r', newline='') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    csvdata = list(csv_reader)
```

now that all necessary data is in main memory a function filters the records and store them in a new list. The execution is balanced on eight threads:

```
def threads_main(n):
    start = (len(csvdata) // 8) * n
    end = start + (len(csvdata) // 8)
    if n == 7:
        end = end + (len(csvdata) % 8)
    if n == 0:
        start = 1
    for i in range(start, end):
        if csvdata[i][0] in users:
            relationships.append([csvdata[i][0], csvdata[i][1]])
threads = []
for i in range(8):
    thread = threading.Thread(target=threads_main, args=(i,))
    threads.append(thread)
    thread.start()
for thread in threads:
    thread.join()
```

Finally, the filtered data are written on a new file:

```
with open('filtered_friends.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(["followed", "follower"])
    for row in relationships:
        writer.writerow([row[0], row[1]])
```

After the execution of this script, we can import *filtered_freinds.csv* into our database using the previously shown queries. The construction of the graph is finished.

## 4.3. PREPROCESSING

The lack of links between second layer followers inevitably leads to a graph with a lot of useless nodes regarding the detection of clusters. We can divide them into two classes:

1. Second layer followers that follow just one user and are followed just by that user.
2. Second layer followers that follow just one user and are not followed by anyone else.

We decide to identify these nodes and not to consider them in some of the next phases of this work, we refer to the graph without these nodes as the filtered graph. To find nodes belonging to the first class, we can execute the following query:

```
1. MATCH(a)-[:FOLLOW]->(b)-[:FOLLOW]->(c)
2. WHERE a.twitterId = c.twitterId
3. AND size((a)-[:FOLLOW]->()) = 1
4. AND size(()-[:FOLLOW]->(a)) = 1
5. SET filteredGraph = false
```

A similar query could be used to find nodes belonging to the second class. However, we decide to identify them using a strongly connected components algorithm, as it is much quicker. We have already introduced the concept of strongly connected nodes (see §1.2). The nodes belonging to the second class above defined, are not strongly connected with anyone else in the graph. Each of them, therefore, is identified as a strongly connected graph.

### Tarjan's algorithm

There are different ways to identify the strongly connected subgraphs, we describe here, the Tarjan's algorithm (Tarjan, R., 1972). The algorithm associates to each node two properties: an *index* that uniquely identifies them, and the id of the

strongly connected subgraph to which they belong (usually called *LowLink*). Initially, these properties are not defined.

The basic idea is that strongly connected nodes are always connected by a loop (and vice versa), therefore once a loop is identified, the *Lowlink* property is set to the same value for each of its nodes. The algorithm uses the Depth-first search to visit every node of the graph. Initially, a random node is chosen, and recursively all its neighbors are visited. Every time that a node is processed its two parameters are set to the same value, and it is pushed into a stack that is used to remember which nodes have already been processed. If the next node to process is already in the stack it means that a loop was found and, if there are no other neighbors, the backpropagation of the recursion starts. During this phase, the *Index* of the first (and last) node of the loop is set as *Lowlink* of all other nodes of the loop.

```
1.  graph G = (V, E)
2.  index := 0
3.  S := empty stack
4.  for each v in V do
5.      if v.index is undefined then
6.          strongconnect(v)
7.      end if
8.  end for
9.  function strongconnect(v)
10.     v.index := index
11.     v.lowlink := index
12.     index := index + 1
13.     S.push(v)
14.     v.onStack := true
15.         for each (v, w) in E do
16.             if w.index is undefined then
17.                 strongconnect(w)
18.                 v.lowlink := min(v.lowlink, w.lowlink)
19.             else if w.onStack then
20.                 v.lowlink := min(v.lowlink, w.index)
21.             end if
22.         end for
```

When the backpropagation ends, the nodes concerning it are removed from the stack. If instead, the backpropagation starts without having found any loop, the last node processed is identified as a strongly connected subgraph with just one element. Even in this case, the node is removed from the stack.

```
    if v.lowlink = v.index then
        repeat
            w := S.pop()
            w.onStack := false
        while w ≠ v
    end if
end function
```

The pseudo-code above is taken from Wikipedia[10]:

### Strongly connected component on Neo4j

Using the GDS library, multiple modalities of execution of the algorithm are available. We choose the modality that stores for each node a property with the id of the subgraph it belongs to, that is the write mode.

A single query is needed:

```
CALL gds.alpha.scc.write({
  nodeProjection: 'User',
  relationshipProjection: 'FOLLOW',
  writeProperty: 'SCC'
})
```

Where the graph on which to run the algorithm is specified using a native projection. In particular, *nodeProjection* represents the label of the nodes and *relationshipProjection* represents the property of the relationship. *WriteProperty* instead, is the name of the new property that is stored in the database.

## 4.4. RESULTS

Main features of the graph:

- **Total graph nodes**: 610143.
- **Total graph links**: 1219604.
- **Filtered graph nodes**: 106502.
- **Filtered graph links**: 610650.

[10] https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

- **Nodes-links rate total graph**: 0.5.
- **Nodes-links rate filtered graph**: 0.17.

This data clearly shows the importance of the preprocessing phase, we passed from a graph with an average of two links for each node to a graph with more than five links for each node.

Another interesting data is the number of first layer nodes, which is 394. It might seem a strange number considering that we started from a user with 404 followers, however, during the retrieve of the first layer followers some of them were dropped:

- **Followers with more than 50.000 users**: 6.
- **Followers with a private profile**: 4.

Other interesting information:

- **Links between layer 1 node**s: 6772.
- **Links from layer 1 nodes to layer 2 nodes:** 403067.
- **Links from layer 2 nodes to layer 1 nodes:** 809030.
- **Links from the main node to layer 1 nodes:** 81.
- **Links from the main node to layer 2 nodes:** 260.

# 5. CLUSTERING

To detect clusters, two different algorithms are executed on the graph. We describe now how they work and how to execute them on Neo4j using the GDS library (see §2.4). The results are shown in the last section of this chapter.

## 5.1. LOUVAIN ALGORITHM

The Louvain algorithm (Blondel, V.D, et al. 2008.) is based on the optimization of a value called modularity (Newman, M.E. and Girvan, M., 2004). The modularity measures the relative density of edges inside communities with respect to edges connecting different communities. The parameter is based on the comparison of the graph with another one generated randomly. This last graph has the same number of nodes of the original one, moreover, the degree of each node is preserved.

### Configuration model

A method used to generate a random graph with these features is the **configuration model**. It can be divided into two steps:

1. Divide each link of the original graph into two halves, each half of the link is called *Stub*.
2. Take a pair of *Stubs* uniformly at random and connect them to create a link. Repeat the step for each remaining pair of *Stubs.*

Note that the number of *Stubs* $l$ is equal to twice the number of links $m$:

$$l = 2m.$$

### Modularity

Suppose now, to take two nodes $v$ and $w$ with degree respectively $k_v$ and $k_w$. We want to find the expected value of a random variable describing the final number $J_{vw}$ of links between them in the random graph:

$$J_{vw} = \sum_{i}^{k_v} I_i,$$

where $I_i$ is a random variable that is equal to 1 if the i-th *stub* of $v$ happens to connect with a *stub* of *w*, otherwise it is equal to 0. For the linearity of the expected value operator we can write:

$$E[J_{vw}] = E\left[\sum_{i} I_i\right] = \sum_{i}^{k_v} E[I_i],$$

considering that:

$$E[I_i] = p(I_i = 1) = \frac{k_w}{2m - 1},$$

we get:

$$E[J_{vw}] = \frac{k_v k_w}{2m - 1},$$

for graphs with many edges, it is possible to approximate this value to:

$$E[J_{vw}] = \frac{k_v k_w}{2m}.$$

We can finally define the modularity as:

$$Q = \frac{1}{(2m)} \sum_{vw} \left[A_{vw} - \frac{k_v k_w}{(2m)}\right] \delta(c_v, c_w),$$

where

- $\delta$ is the Kronecker delta function:

$$\delta(x, y) = 1 \; if \; x = y, 0 \; otherwise.$$

- $c_v$ is the cluster of the node *v.*
- *A* is the adjacency matrix of the original graph representing the number of links between an ordered pair of nodes (that can be 0 or 1).

In this formula, the initial division by $2m$ normalizes the modularity in the range $[-0.5, 1]$ where the value 1 describes a very clustered network.

### Description of the algorithm

The algorithm starts assigning each node to a different cluster. The change in modularity is calculated for removing a node from its initial cluster and assigning it to the cluster of each of its neighbors. In the end, the node is assigned to the cluster that increases the modularity more. This process is applied repeatedly for each node until no modularity increase can occur or the maximum number of iterations is reached.

The next step is to condense the graph to get a new graph where each node represents a cluster of the previous one. Edges between two nodes of the same cluster become self-loops on the new cluster node. Edges between two nodes belonging to different clusters instead become edges between the two new nodes representing those clusters.

The algorithm is repeated using the condensed graph, the number of iterations is a configurable parameter.

### Execution

To run the Louvain modularity on Neo4j we use the default configurations of the algorithm. Here the most important parameters:

- **MaxLevels:** the maximum number of levels in which the graph is clustered and then condensed. **Default value:** 10.
- **MaxIterations:** the maximum number of iterations that the modularity optimization runs for each level. **Default value:** 10.
- **Tolerance:** the minimum change in modularity between iterations. If the modularity changes less than the tolerance value, the result is considered stable and the algorithm returns. **Default value:** 0.0001.

The first step to execute the algorithm is the projection of the graph. We want to work only with the filtered graph; therefore, we use a Cypher projection:

```
CALL gds.graph.create.cypher(
     'filteredGraph',
     'MATCH (n:User {filteredGraph: true}) return id(n) as id',
     'MATCH (n)-[r:FOLLOW]-(m)
      RETURN id(n) AS source, id(m) AS target',
)
```

Where we specified the name of the projection, the set of nodes, and the set of relationships to project. The missing arrow in the "MATCH" clause indicates that we are not considering the direction of the edges. In other words, we consider an undirected graph as this works best with the Louvain Algorithm.

Finally, we can run the algorithm:

```
CALL gds.louvain.write(
     'filteredGraph', { writeProperty: 'louvainId' })
YIELD communityCount, modularity, modularities
```

Now, each node of the filtered graph has the property "louvainId" containing the id of the cluster to which it belongs according to the Louvain Algorithm.

## 5.2. LABEL PROPAGATION ALGORITHM

The label propagation algorithm (Raghavan, U.N., et al., 2007) is very simple to describe and does not require any particular prerequisite, therefore, we directly proceed explaining how it works.

The algorithm starts initialing each node with a unique label. Afterward, a random node updates its label to the one shared by most of its neighbors, if there are more possibilities a label is chosen randomly. This step is repeated for all nodes iteratively until each node has the most frequent label of its neighbors. Finally, all nodes sharing the same label are identified as a cluster. Intuitively, when a graph has a clustered topology, it is likely that a label propagates in a whole cluster, whereas it is unlikely that a label propagates from a cluster to another one. Note that, the order in which nodes are processed is chosen randomly, this means that different executions can lead to different results.

### Execution

We project the filtered graph using a cypher projection:

```
CALL gds.graph.create.cypher(
    'filteredGraph',
    'MATCH (n:User {filteredGraph: true}) return id(n) as id',
    'MATCH (n)-[r:FOLLOW]-(m)
     RETURN id(n) AS source, id(m) AS target',
)
```

When the graph is in main memory, we proceed to the execution:

```
CALL gds.labelPropagation.write('filteredGraph', { writeProperty: 'label
PropagationId' })
YIELD communityCount, ranIterations, didConverge
```

## 5.3. DATA EXTRACTION

The next step is to retrieve more information about each cluster. The information that we need can be divided into intra-cluster and inter-cluster.

**Intra-cluster information:**

- Number of nodes for each cluster.
- Number of links for each cluster.
- Number of first layer nodes for each cluster.

**Inter-cluster information:**

- Number of links from a cluster to another one.
- Number of nodes of a cluster that have a link with another cluster.

As a first step, we show how to retrieve the list of clusters ids. The query to get this information is:

```
MATCH(n) RETURN  DISTINCT n.louvainId;
```

We store the list of ids in a file executing the following command from the Linux shell:

```
$ CAT query.cql | cypher-shell > clusters_ids
```

where the file *query.cql* contains the above query.

In the following, we describe the *clustersInfo.py* script, which is used to retrieve both inter-cluster and intra-cluster information. This script relies on the *neo4jInterface* class already described in §3.2.

First, the script reads the *clusters_ids* file:

```
with open('clusters_ids','r') as communities_ids
    communities = communities_ids.readlines()
    communities = [int(community) for community in communities]
```

Later, intra-cluster information is retrieved:

```
for community in communities:
    links = db.count_links(community)
    users = db.count_users(community)
    lv1 = db.count_lv1_users(community)
```

where count_links() executes:

```
MATCH (a:User {louvainId: $community})
-[r:FOLLOW]->
(b:User {louvainId: $community})
RETURN count(*) as links
```

count_users() esecutes:

```
MATCH (a:User {louvainId: $community}) RETURN count(*) as users
```

while count_lv1_users() executes:

```
MATCH (a:User {louvainId: $community}) WHERE a.layer = 1
RETURN count(*) as lv1
```

The next step is to retrieve inter-cluster information:

```
for community in communities:
  for community2 in communities:
    if community != community2:
      inter_links=db.count_inter_links(community, community2)
      users_community2=db.count_inter_users(community,community2)
      users_community=db.count_inter_users(community,community2)
```

where count_inter_links() executes:

```
MATCH (a:User {louvainId: $community})
<-[r:FOLLOW]->
(b:User {louvainId: $community2})
RETURN count(r) as inter_links
```

while count_inter_users() executes:

```
MATCH (a:User {louvainId: $community})
<-[r:FOLLOW]->
(b:User {louvainId: $community})
RETURN  count(DISTINCT b.twitterId) as inter_users
```

The queries shown above are those used to extract information about clusters found with the Louvain algorithm. We use the same procedure also for clusters found with the label propagation algorithm.

The *graphistry.py* script (§3.3) can be used to visualize how clusters are interconnected. The query to use in the script is the following:

```
 match(n )<-[r:FOLLOW]->(m) WHERE
(n.louvainId = <id_cluster1> AND m.louvainId = <id_cluster2>) OR
(n.louvainId = <id_cluster1> AND m.louvainId = <id_cluster1>) OR
(n.louvainId = <id_cluster2> AND m.louvainId = <id_cluster2>)
return n,m,r
```

Some images are shown in the next section.

## 5.4. RESULTS

Using the label propagation algorithm, the filtered graph is identified as a single cluster. This result means that the algorithm does not perform well on our kind of graph. Using the Louvain algorithm instead, the filtered graph is divided into 10 clusters and the modularity is 0.4. This last value is very high considering that our graph does not contain all relationships of the network. In the following, we only consider the results of the Louvain algorithm.

**Table 5.1. Intra-cluster information.**

| Cluster-ID | Nodes | Links | First layer users |
|---|---|---|---|
| 215 | 7358 | 23299 | 30 |
| 198 | 14858 | 54004 | 59 |
| 177 | 5273 | 13322 | 28 |
| 276 | 16581 | 51566 | 64 |
| 204 | 11208 | 28740 | 56 |
| 99 | 15067 | 34421 | 46 |
| 52 | 16661 | 70071 | 61 |
| 318 | 798 | 1436 | 4 |
| 289 | 6389 | 17419 | 35 |
| 233 | 12319 | 38434 | 11 |

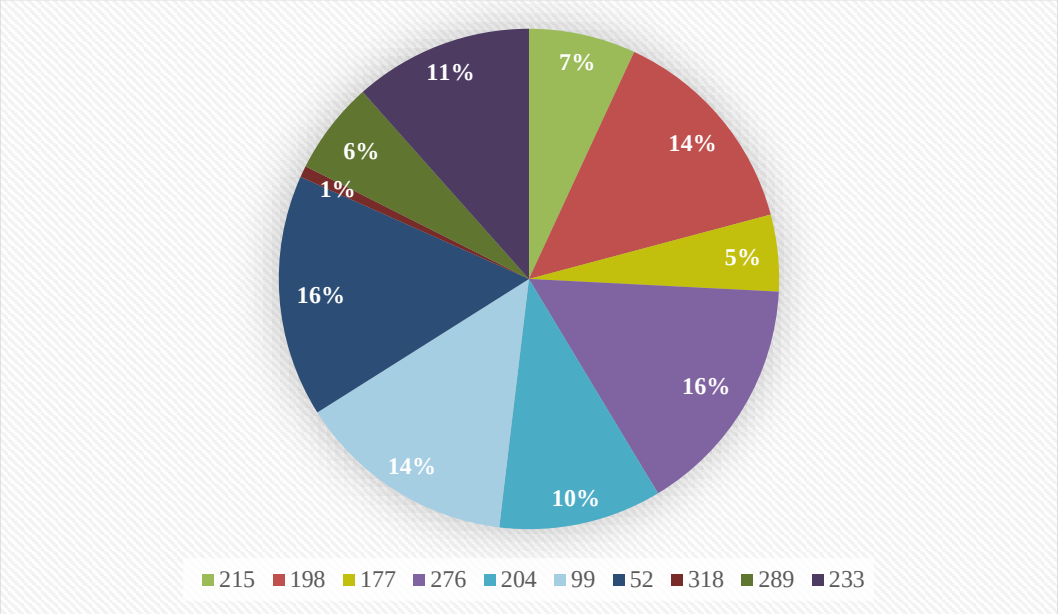


**Chart 5.1. Distribution of users over clusters.**

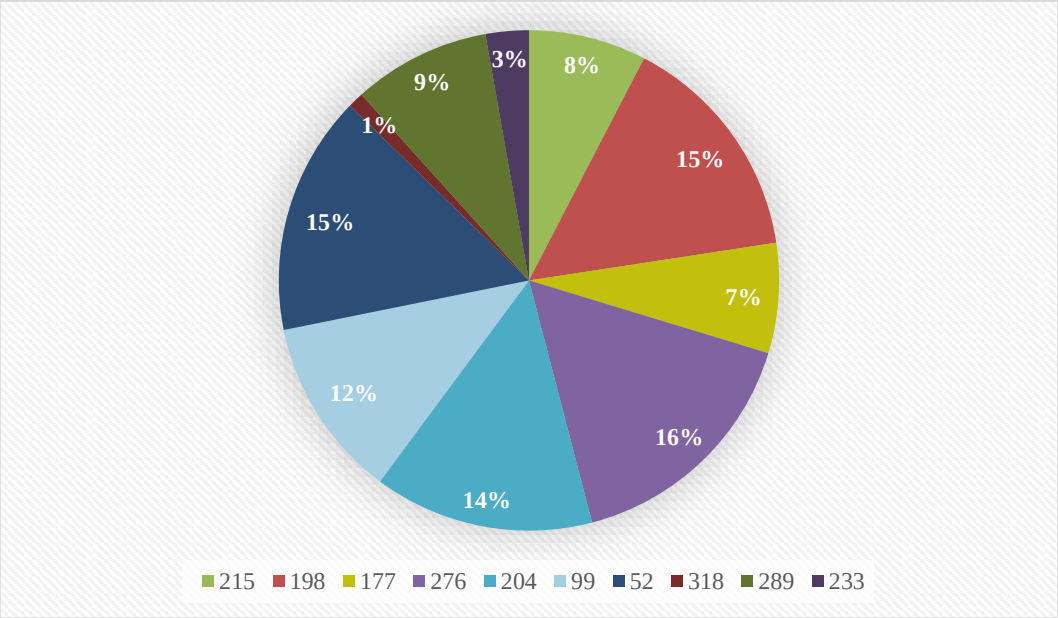


**Chart 5.2. Distribution of first layer users over clusters.**

**Table 5.2. Inter-cluster information.**

| Cluster 1 | Cluster 2 | Nodes cluster 1 | Nodes cluster 2 | Links |
|---|---|---|---|---|
| 215 | 198 | 942 | 833 | 3350 |
| 215 | 177 | 808 | 178 | 1302 |
| 215 | 276 | 627 | 576 | 1875 |
| 215 | 204 | 736 | 402 | 1845 |
| 215 | 99 | 616 | 359 | 1654 |
| 215 | 52 | 1703 | 1295 | 7596 |
| 215 | 318 | 54 | 25 | 103 |
| 215 | 289 | 423 | 267 | 1211 |
| 215 | 233 | 512 | 204 | 974 |
| 198 | 177 | 2941 | 1272 | 7468 |
| 198 | 276 | 3922 | 3948 | 18320 |
| 198 | 204 | 3045 | 3004 | 12483 |
| 198 | 99 | 4546 | 3271 | 17041 |
| 198 | 52 | 5556 | 4370 | 27547 |
| 198 | 318 | 452 | 229 | 1175 |
| 198 | 289 | 2435 | 1767 | 9787 |
| 198 | 233 | 2681 | 1630 | 7308 |

| | | | | |
|---|---|---|---|---|
| 177 | 276 | 1185 | 1611 | 4923 |
| 177 | 204 | 1269 | 1274 | 3829 |
| 177 | 99 | 609 | 1565 | 3265 |
| 177 | 52 | 872 | 2193 | 4724 |
| 177 | 318 | 60 | 78 | 171 |
| 177 | 289 | 965 | 881 | 3372 |
| 177 | 233 | 528 | 554 | 1452 |
| 276 | 204 | 2550 | 1669 | 8005 |
| 276 | 99 | 3500 | 2253 | 10826 |
| 276 | 52 | 3053 | 2285 | 11317 |
| 276 | 318 | 131 | 82 | 274 |
| 276 | 289 | 2669 | 1691 | 9381 |
| 276 | 233 | 2766 | 1433 | 6986 |
| 204 | 99 | 1781 | 1835 | 6477 |
| 204 | 52 | 2898 | 2972 | 13427 |
| 204 | 318 | 89 | 48 | 178 |
| 204 | 289 | 885 | 805 | 3135 |
| 204 | 233 | 1262 | 654 | 2715 |
| 99 | 52 | 3332 | 3615 | 14577 |
| 99 | 318 | 221 | 285 | 864 |
| 99 | 289 | 1336 | 915 | 3846 |
| 99 | 233 | 3706 | 2647 | 11063 |
| 52 | 318 | 530 | 395 | 1697 |
| 52 | 289 | 2042 | 1617 | 8604 |
| 52 | 233 | 4935 | 2878 | 17013 |
| 318 | 289 | 130 | 137 | 486 |
| 318 | 233 | 200 | 238 | 629 |
| 289 | 233 | 1269 | 1103 | 3663 |

**Figure 5.1. Clusters 215 (green) and 99 (light blue).**

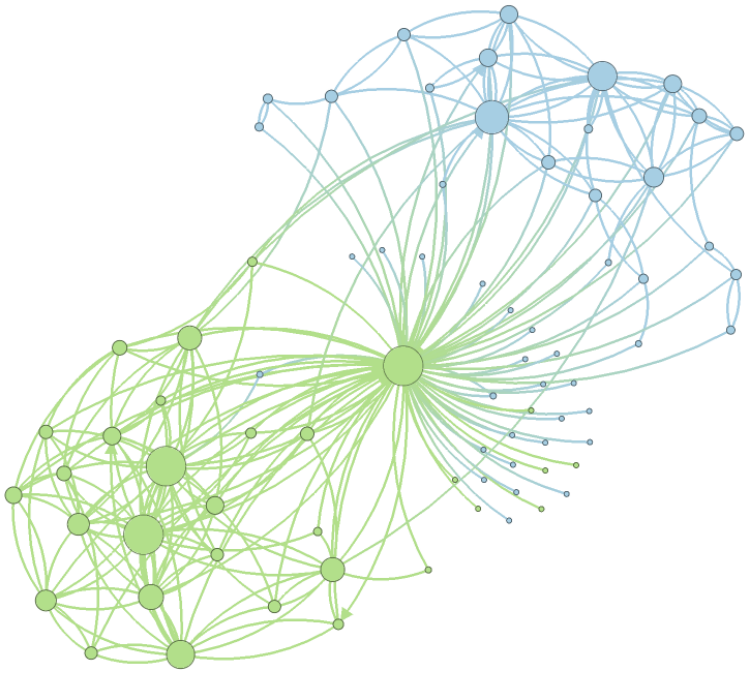

**Figure 5.2. First layer users (and the main node) of cluster 215 (green) and 99 (light blue).**

# 6. HUBS DETECTION

A hub is a node that quickly propagates information between clusters. We think about hubs as bridges between important nodes (*leaders*) of different clusters. To identify leaders, different algorithms can be used. In this work, we use the *PageRank* algorithm, which is presented in §6.1.

To clarify the concept of a hub, we directly show the query necessary to identify them:

```
1. MATCH(m {leader: true})-[:FOLLOW]->
2. (h)-[:FOLLOW]->(n {leader: true})
3. WHERE NOT m.louvainId = n.louvainId
4. SET h.hub = true
```

Note that the direction of the relationships is very important. Nodes identified using the following query have no role in the propagation of information:

```
1. MATCH(m {leader: true})<-[:FOLLOW]-
2. (h)-[:FOLLOW]->(n {leader: true})
3. RETURN h
```

We can define four interesting parameters regarding the capacity of a cluster to exchange information:

- $\text{P1} = \frac{\text{n users of the cluster}}{\text{n hubs that follow the leaders of the cluster}}$

  P1 describes the capacity of a cluster to disseminate the information generated internally to other clusters. The smaller it is, the greater this capacity will be.

- $\text{P2} = \frac{\text{n users of the cluster}}{\text{n hubs that are followed by the leaders of the cluster}}$

  P2 describes the capacity of a cluster to receive information from other clusters. The smaller it is, the greater this capacity will be.

- $\text{P3} = \frac{\text{n users of the cluster}}{\text{n hubs that are in any way linked with the leaders of the cluster}}$

  P3 describes the capacity of a cluster to exchange information with other clusters. The smaller it is, the greater this capacity will be.

- P4 $= \frac{\text{n hubs that are followed by the leaders of the cluster}}{\text{n hubs that follow the leaders of the cluster}}$

  P4 describes whether the cluster is more likely to receive (P4 > 1) or disseminate (P4 < 1) information.

To calculate these parameters, we consider the total graph rather than the filtered one. Nodes that are not part of any cluster are considered part of the cluster to which they are linked (each of them is linked to only one cluster).

## 6.1. PAGERANK

PageRank (Page, L. et al., 1999) is the algorithm used by Google to identify the most important webpages. The algorithm returns a probability distribution that describes the likelihood that a person randomly clicking on links will arrive at any particular page.[11]

We describe the algorithm considering a graph where each node is a page and each relationship is a link between pages, however, it can be applied to any kind of graph.

### Description

Each page is labeled with a number that describes how likely is to arrive on that page. We refer to this likelihood as $PR(u)$ where $u$ indicates a node. At the beginning, all pages are initialized with $PR(u)$ equal to $1/N$, where $N$ is the number of pages.

For each node, the value $PR(u)$ is updated using the following formula:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)},$$

where:

- $B_u$ is the set of neighbors of the node $u$.

[11] https://en.wikipedia.org/wiki/PageRank

- $L(v)$ is the number of outgoing links of the node v.

This last step is repeated until the value of $PR(u)$ converges for each node $u$. The convergence is ensured by a theorem about Markov chains. [12] Note that the value of $PR(v)$ in the formula refers to the previous iteration.

This description of the algorithm does not consider that a user surfing on the web can at any time decide to stop surfing. To consider this eventuality the formula used to update $PR(u)$ has to be changed:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v)}{L(v)},$$

where $d$ is the probability that at any step a person keeps surfing clicking on another link and is usually set to $0.85$. In particular, if $d = 1$, it means that the person keeps surfing forever and we get the previous formula.

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)},$$

whereas if $d = 0$, it means that the person randomly chooses the first page and stops surfing. The probability to arrive on a specific page becomes:

$$PR(u) = \frac{1}{N}.$$

Neo4j uses a slightly different formula in its implementation:

$$PR(u) = 1 - d + d \sum_{v \in B_u} \frac{PR(v)}{L(v)},$$

the result of this formula is not in the range $[0,1]$, therefore, it is not a probability.

[12] https://en.wikipedia.org/wiki/Markov_chain

### Execution

To run the PageRank algorithm on Neo4j we fisrst project the total graph in main memory:

```
CALL gds.graph.create(
   'totalGraph',
   'User',
   'FOLLOW',
)
```

Finally, we can proceed with the execution:

```
CALL gds.pageRank.write('totalGraph', {
   maxIterations: 20,
   dampingFactor: 0.85,
   writeProperty: 'pagerank'
})
YIELD nodePropertiesWritten, ranIterations
```

## 6.2. DATA EXTRACTION

To extract the data we use the class *neo4jInterface* (see §3.2), even in this case, we avoid to show the code of the class. First, ids of the clusters are read:

```
with open('clusters_id', 'r') as clusters_ids:
    clusters = clusters_ids.readlines()
    clusters = [int(cluster) for cluster in clusters]
```

Finally,the four parameters are calculated:

```
for cluster in clusters:
    n_users = db.count_user(cluster) + db.count_users_dropped(clu
ster)
    hubs = db.count_hubs(cluster)
    in_hubs = db.count_out_hubs(cluster)
    out_hubs = db.count_in_hubs(cluster)
    leaders = db.count_leaders(cluster)

    p1 = n_users / out_hubs
    p2 = n_users / in_hubs
    p3 = n_users / (in_hubs + out_hubs)
    p4 = in_hubs / out_hubs
```

where count_user() executes:

```
MATCH (a:User {louvainId: $cluster}) RETURN count(*) as users
```

count_users dropped() executes:

```
MATCH (a:User {filtered_graph: false})
-[FOLLOW]->
(b:User {louvainId: $cluster})
RETURN count(distinct a) as users
```

count_hubs() executes:

```
MATCH (a:User {hub: true})
<-[FOLLOW]->
(b:User {louvainId: $cluster})
WHERE b.leader= true
RETURN count(distinct a) as hubs
```

count_out_hubs() executes:

```
MATCH (a:User {hub: true})
-[FOLLOW]->
(b:User {louvainId: $cluster})
WHERE b.leader= true
RETURN count(distinct a) as hubs
```

count_in_hubs() executes:

```
MATCH (a:User {hub: true})
<-[FOLLOW]-
(b:User {louvainId: $cluster})
WHERE b.leader= true
RETURN count(distinct a) as hubs
```

And count_leaders() executes:

```
MATCH (a:User {louvainId: $cluster}) where a.leader= true
RETURN count(distinct a) as n_leaders
```

## 6.3. RESULTS

The following results consider as leaders the first 21 nodes returned by the PageRank algorithm. This number is the minimum necessary to have at least one leader for each cluster. The total number of hubs is 20357.

**Table 6.1. Hubs information for each cluster. The number of users for each cluster shown in this table also includes the nodes that are not in the filtered graph but are connected with that specific cluster.**

| Cluster | N users | N leaders | N hubs | N out-hubs | N in-hubs |
|---|---|---|---|---|---|
| 215 | 75489 | 3 | 909 | 795 | 440 |
| 198 | 50424 | 2 | 6027 | 4897 | 4383 |
| 177 | 59244 | 1 | 5855 | 912 | 5810 |
| 276 | 64574 | 2 | 4080 | 3292 | 3439 |
| 204 | 94957 | 2 | 3785 | 3340 | 2154 |
| 99 | 122461 | 4 | 8993 | 7361 | 6011 |
| 52 | 54018 | 2 | 6656 | 6249 | 4421 |
| 318 | 2625 | 1 | 1256 | 862 | 952 |
| 289 | 34906 | 1 | 3297 | 3155 | 1410 |
| 233 | 51445 | 3 | 11273 | 7732 | 9657 |

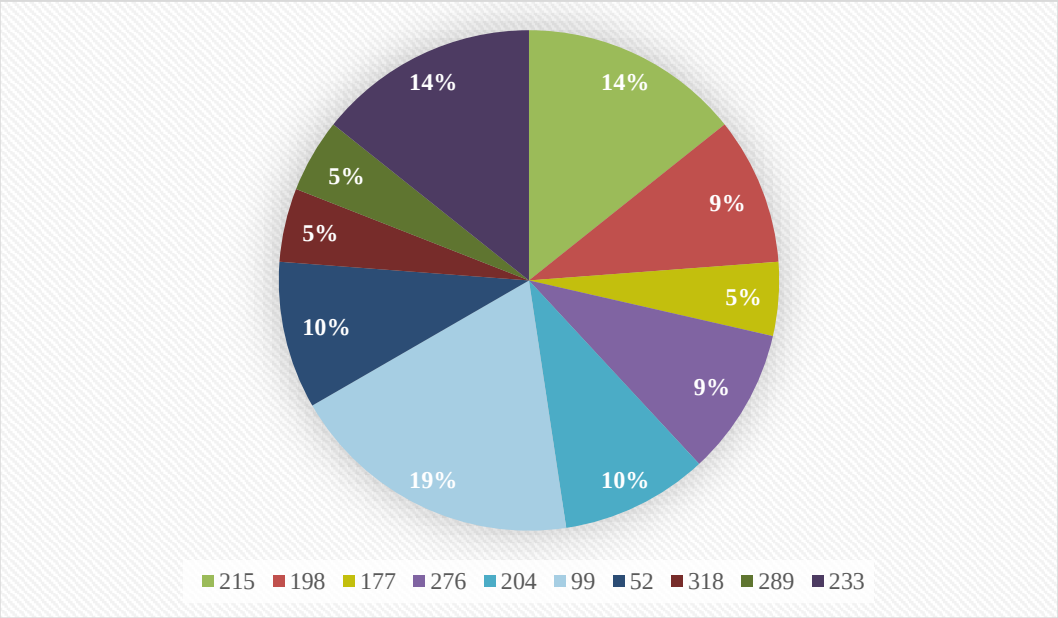


**Chart 6.1. Distribution of leaders over clusters.**

**Table 6.2. Parameters describing the capacity of a cluster to disseminate or receive information (see §6). The parameters are calculated considering the values of table 6.1.**

| Cluster | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| 215 | 94,95472 | 171,5659 | 61,1247 | 0,553459 |
| 198 | 10,29692 | 11,50445 | 5,433621 | 0,895038 |
| 177 | 64,96053 | 10,1969 | 8,813448 | 6,370614 |
| 276 | 19,61543 | 18,77697 | 9,593523 | 1,044654 |
| 204 | 28,43024 | 44,08403 | 17,28376 | 0,64491 |
| 99 | 16,63646 | 20,37282 | 9,158017 | 0,816601 |
| 52 | 8,644263 | 12,2185 | 5,062605 | 0,707473 |
| 318 | 3,045244 | 2,757353 | 1,447078 | 1,104408 |
| 289 | 11,06371 | 24,75603 | 7,64644 | 0,44691 |
| 233 | 6,653518 | 5,327224 | 2,958479 | 1,248965 |

# 7. CONCLUSIONS

The next step would be to analyze the results to get as much information as possible about each user of the network. The aim of this thesis is just to give a general idea about how to analyze a social network; therefore, with the detection of hubs, our work is concluded. A summary of the work is provided in the following.

Starting from a user with 404 followers we used the Twitter APIs to retrieve the list of first- and second-layer followers. The data was initially stored in a CSV file and later imported on Neo4j. Due to the rate limit of the APIs, it was not possible to add to the graph all relationships between users. This lack of information made it necessary a preprocessing phase thanks to which we passed from a graph with 610143 nodes and 1219604 links to a filtered graph with 106502 nodes and 610650 links.

To detect clusters, we executed the label propagation algorithm and the Louvain modularity algorithm on the filtered graph. Just the Louvain modularity algorithm returned interesting results: 10 different clusters have been detected.

During the phase of hubs detection, it was important to work on the total graph. All nodes that were not part of any cluster have been considered part of the cluster to which they were linked. We used the PageRank algorithm to find the leaders of each community. The nodes that act as bridges between leaders of different clusters have been identified as hubs. The total number of hubs is 20357. We concluded this phase by calculating some interesting parameters about the capacity of a cluster to exchange information with other clusters.

Note that the work of this thesis is based on a time-variant network. A future analysis starting from the same user and following the same steps could lead to different results.

# ACKNOWLEDGMENTS

I am extremely grateful to my supervisors prof. Antonella Di Stefano and eng. Andrea Di Maria for their excellent guidance and for having been patient and available throughout this thesis.

I would like to thank my colleagues Alessandro, Alex, Daniele, Francesco, Giuseppe, Pietro, Simona, and Sofia for keeping me company during the few classes I attended.

I thank Francesco, Isabella, Luca, Maria, and Rosario for all the nice moments we spent together, for making me feel at home during my stay in Catania, and for being the best friends I could ever meet.

I would like to extend my thanks to all people who made my Erasmus experience wonderful. Especially to Ardita, Fernando, Francisco, Lovro, and Martin for all the fun we had during our free time; to Ben, Iris, Jason, Kurt, Leo, and Skyler for everything they taught me and for the joyful moments we spent studying together.

Special thanks to my parents for their love and their sacrifices for educating and preparing me for my future, and to my brothers who always helped and encouraged me.

# LIST OF FIGURES

Figure 1.1. An undirected graph with 3 vertices and 3 edges .....4
Figure 1.2. A directed graph with 3 vertices and 4 edges .....5
Figure 1.3. A random graph(left) and a clustered graph(right) . .....7
Figure 5.1. Clusters 215 (green) and 99 (light blue) .....37
Figure 5.2. First layer users of cluster 215 (green) and 99 (light blue) .....37

# LIST OF CHARTS

Chart 5.1. Distribution of users over clusters .....34
Chart 5.2. Distribution of first layer users over clusters .....35
Chart 6.1. Distribution of leaders over clusters .....44

# LIST OF TABLES

Table 5.1. Intra-cluster information ..... 34
Table 5.2. Inter-cluster information ..... 35
Table 6.1. Hubs information for each cluster ..... 43
Table 6.2. Parameters describing how clusters exchange information ..... 44

# REFERENCES


Albert, R. and Barabási, A.L., 2002. *Statistical mechanics of complex networks.* Reviews of modern physics, 74(1).

Barrat, A. and Weigt, M., 2000. *On the properties of small-world network models*. The European Physical Journal B-Condensed Matter and Complex Systems, 13(3).

Blondel, V.D., Guillaume, J.L., Lambiotte, R. and Lefebvre, E., 2008. *Fast unfolding of communities in large networks*. Journal of statistical mechanics: theory and experiment, 2008(10).

Da Fontoura Costa, L., 2018. *What is a Complex Network*? (CDT-2).

Gregory, S., 2007, September. *An algorithm to find overlapping community structure in networks*. In European Conference on Principles of Data Mining and Knowledge Discovery. Springer, Berlin, Heidelberg.

Newman, M.E. and Girvan, M., 2004. *Finding and evaluating community structure in networks*. Physical review E, 69(2), p.026113.

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

Raghavan, U.N., Albert, R. and Kumara, S., 2007. *Near linear time algorithm to detect community structures in large-scale networks*. Physical review E, 76(3), p.036106.

Schaeffer, S.E., 2007. *Graph clustering*. Computer science review, 1(1).

Tarjan, R., 1972. *Depth-first search and linear graph algorithms*. SIAM journal on computing, 1(2), pp.146-160.

Watts, D.J. and Strogatz, S.H., 1998. *Collective dynamics of 'small-world'networks*. nature, 393(6684).

Zachary, W.W., 1977. *An information flow model for conflict and fission in small groups*. Journal of anthropological research, 33(4).