POLITECNICO DI TORINO

Master's Degree in Computer Engineering - Cybersecurity

Politecnico di Torino

Master's Degree Thesis

Design and implementation of a scalable multiprocessing crawler for Telegram.

Supervisors

Prof. Marco MELLIA

Nikhil JHA

Giordano PAOLETTI

Candidate

Simone GALOTA

December 2023

Abstract

Telegram is a well-known instant messaging service focused on providing a highly privacy-preserving social network environment to its users.

This work aims to build a platform for automatic analysis and collection of Telegram channel, mega-group, group, and bot data, enhancing the two scrapers previously written in Python by Edoardo Gabrielli [1] and Armando Chimirri [2]. To reach the goal, a multi-processing crawler has been designed and implemented using master and workers architecture. We developed a novel approach to automatically collect data from Telegram groups/channels, implementing a distributed and scalable platform in Python, leveraging the Telethon API library.

The core of the implementation of our crawler consists of the separation of operations between master and workers. With respect to the previous versions, we are able to collect, through Snowball Sampling, not only data (messages, links, ...) from groups or channels but also to interact with the bots present in the platform.

Each worker is a process connected to a Telegram account. Worker processes operate concurrently, each one is responsible for fetching and processing a task assigned by the master. Once the task is finished, the worker will wait, self-stopping its execution, for about 2 minutes to avoid the abuse of API functions calling and the consequent block imposed by the Telegram servers with a Flood Wait Error. After waiting, each worker returns the result of its work and puts its ID in the free processes queue, becoming ready for another task assignment.

The master process acts as the coordinator of the crawling procedure. Its responsibilities include management of the Telegram URL to be crawled, keeping track of the link states, and organizing results returned from the workers. Moreover, the master stores in the database the data collected from the workers and updates the statistics regarding the activity performed. The three kinds of tasks that the master can assign to a worker are JOIN (calling API function to try to join a channel/group), CHECK WAIT (checking if a request previously sent has been accepted or not), LEAVE (leaving the group/channel if are in for more than 24 hours).

The software, with all the design and platform limitations, is capable of evaluating links at the rate of 28/30 links for each worker per hour, resulting in an improvement compared to previous versions. Also, we obtained a directed and weighted graph, in which we analyzed the crawling progression starting from the provided seed.

Acknowledgements

I would like to thank the supervisor Prof. Marco Mellia along with Dr. Nikhil Jha and Dr. Giordano Paoletti for their kindness, helpfulness, and professionalism in supporting me during the development of this work.

Table of Contents

List of Figures

Chapter 1

Introduction

In the current digital landscape, the huge and interconnected world of online communication and content sharing has given rise to new ways of exploration and research of social network environments. Among the various platforms that facilitate these interactions, Telegram stands out as a prominent force, boasting an expansive user base and a diverse ecosystem of channels, groups, and bots. While Telegram has transformed the way we connect and exchange information, it has also raised important questions related to security and privacy. This master thesis aims to uncover the significance of crawling Telegram, with a specific focus on security and privacy implications.

The need for a scalable and efficient Telegram crawler becomes paramount as the platform continues to grow in influence. Telegram's unique structure and features create a challenging environment to navigate, which is increasingly vital for the detection and prevention of security threats. Illicit activities such as instances of sensitive data leaks, illicit content distribution, and even the sale of confidential information on Telegram are becoming more prevalent. This research seeks to address these challenges, shedding light on the scope of potential security issues and privacy breaches within the platform.

This introductory section will provide an in-depth exploration of the motivations, goals, and significance of this research endeavor. We will also outline the structure of the thesis and the methodology employed, setting the stage for a comprehensive examination of the design and implementation of a versatile, scalable, and, in the future, intelligent crawler for Telegram. This research aims to assist in the

automated detection of content that poses security risks, such as data leaks and the unauthorized distribution of sensitive data or other kinds of illegal behaviors.

Subsequent chapters will delve deeper into the complexity of this project, including a detailed analysis of the current state of the Telegram platform and crawling, the challenges presented by Telegram's security and privacy concerns, and the innovative multi-process master-workers architecture employed to make our crawler more efficient.

It is our goal that the results and insights generated by this research will not only contribute to academic discourse but also play a pivotal role in enhancing the security and privacy of users within the Telegram platform.

1.1 Background and state of the art

This section delves into the description of the Telegram application and its entities, as well as the current state of the art of Telegram crawling. As the privacy importance has become more and more discussed, Telegram has gained immense popularity in recent years. The goal of this section is to give an overview of both the environment we are working on and the activities already done by other research groups.

1.1.1 What is Telegram?

Telegram [3] is an instant messaging service that has as its main focus the privacy of the users interacting on the platform. It was created in 2013 by the Durov brothers. Its founders envisioned a platform that prioritizes individual confidentiality and data protection above all else.

It is a cloud-based service able to guarantee seamless synchronization across multiple devices. Its data centers are strategically located in five countries: North America, the United Kingdom, the Netherlands, Singapore, and the United Arab Emirates. This global presence not only optimizes data access and distribution but also strengthens the platform's resilience against potential disruptions, ensuring users can stay connected without interruption.

The application is available for all the major platforms such as Android, iOS,

iPadOS, Windows, Linux, and macOS. It presents also a web version accessible from the main browsers. In 2023, it reached 700 million active users and a market value of 30 billion dollars [4], underlying its prominent role in the social network industry.

The service is mainly free, but, since 2022, it does exist a premium version that lets the user overcome some limitations of the free version.

1.1.2 State of the art

The state of the art in Telegram crawling reflects a dynamic landscape where researchers and developers strive to unlock the vast potential of this popular messaging platform in collecting huge amounts of data useful for further analysis. Telegram, with its emphasis on user privacy and security, presents a unique challenge and opportunity for crawling initiatives. Indeed many research groups have focused their work on social network analysis and data mining. After a careful literature reading, we found interesting papers related to our work. However, while the majority of publications contain a consistent part of data analysis even using machine learning techniques, our work aims almost exclusively at the creation of an advanced and more sophisticated crawling tool.

Nobari *et al.* [5] developed a crawler to gather data and made a structural and topical analysis of messages on a dataset of more than 2000 groups and channels. Hashemi *et al.* [6] have performed for the first time a detailed analysis of Iranian users' behavior; they measured group qualities in more than 900.000 Persian channels and more than 300.000 Persian supergroups.

Khaund *et al.* [7] conducted both text and network analysis on the data collected in channels to gain insights into political discourse and public opinion; their results show that those channels are active in divulging information about political affairs.

In their work, La Morgia *et al.* [8] identified and analyzed fake channels on Telegram. They built a dataset made of over 120.000 channels and 247 million messages, proposing a machine-learning model for the detection phase, capable of achieving an accuracy of 85.49%.

1.1.3 Thesis outline

The topics are organized as follows:

- Chapter 2 deals with the description of the crawler.
- Chapter 3 deals with the crawler design and implementation.
- Chapter 4 shows some analysis and results.
- Chapter 5 contains conclusion and future work.

Chapter 2

The Crawler

The core of this work consisted of developing a crawler able to deal with the different entities present in the Telegram scenario. This work is to be considered as an improvement of the code already developed by Armando Chimirri [2] and previously by Edoardo Gabrielli [1].

A crawler, or scraper, is a software whose goal is to automatically browse web pages and collect information from them. Moreover, a crucial task of this kind of program or script is to follow links inside a web page. Hence, they are essential components of search engines, helping to index and update content for search results.

In a context like Telegram, a crawler takes on the same function with a little difference. Instead of browsing web pages, it tries to join groups/channels given a link and collects messages, metadata, and links inside them.

2.1 Entities in Telegram

Telethon API

Since we decided to continue and improve the work already done to make the crawling software more efficient, the choice of using Python as a language has been quite natural, as well as continuing to use the Telethon API [9].

Telethon is not an official Telegram API; rather, it is a third-party Python library that provides an easy-to-use interface for interacting with the Telegram API.

Telethon simplifies the process of working with the Telegram API by providing a high-level, asynchronous Python API. It allows developers to create, send, and receive messages, work with chats and channels, and perform various other actions within the Telegram ecosystem. It's essential to note that it's not officially developed or endorsed by Telegram. Telethon API is an open-source library aimed to provide functions to programmers for easily writing code able to interact with the Telegram service. To date, this API contains:

- 562 methods.
- 452 types.
- 1214 constructors.

Let's see now which entities we can interact with and which use cases we can find on the platform. In Telethon, an "entity" is a representation of a user, a chat, or a channel on the Telegram messaging platform

User

The entity User is associated with each account. The user inserts their phone number into the application to create an account. After receiving a six-digit OTP via SMS, the user logs into the application.

Chat between Users

This is the standard use case. Similar to numerous other instant messaging apps, when a user initiates contact with another user, they engage in one-on-one communication. Opting for the "secret chat" feature results in end-to-end encryption, ensuring that all exchanged information (text, pictures, videos, audio, GPS positions, and more) remains visible and understandable exclusively to the parties involved. Telegram offers the opportunity to communicate with other peers in complete privacy, depending on the configuration we choose. Unlike the main competitors, it also offers the opportunity to chat with anyone without sharing our phone number. It is needed to highlight that this use case is not useful for crawling data on the platform.

Group

Another kind of element present on Telegram is the entity *Group*. It is a many-to-many communication. A user can create a *Group* (becoming the admin) for sending messages to a large number of users. All of them can write answers in the group. A group can be:

- Private: an invite is needed through a link to access the group. There could be a request to be approved.
- Public: no link needed, it is sufficient to look for the name of the group in the search bar of the application.

Moreover, there are 3 different kinds of groups, based on the number of users participating:

- Basic: groups in which can participate 200 members at most. The users can access up to 100 messages previously written in the group.
- Super: groups in which between 200 and 200,000 users can participate. They can access either all messages previously written or none depending on the group settings the admin has chosen.
- Giga: groups in which are participating more than 200.000 users. In this case, only the admins can write in the group.

Channel

This entity implements the one-to-many communication. A user can create a channel (becoming the admin) to write messages in broadcast to a large number of subscribers. There is no limit of users number in a channel. However, the subscribers cannot answer the messages posted by the admins in the channel, but they can either react or comment on the message written. Also, there can be groups associated with the channel for discussion between users about channel content. A channel can be:

- Private: an invite is needed through a link to access the channel, and when we are in, we have access to all contents shared previously from the channel creation onward. Even in this case, there could be a request to be approved.

In addition, from the application or web interface, a user not participating in the channel with more than 200 subscriptions, has access to the channel contents for 5 minutes. After that time it is asked to join to keep access to vision.

- Public: no link needed, it is sufficient to look for the name of the group in the search bar of the application. We have access to the channel without joining it.

Bot

Since 2015 Telegram has been letting users create software for automating tasks and interacting with other users based on predefined rules (i.e. bot). They can execute filtering operations in a group or a channel, acting as a barrier. When a normal user tries to join those entities, it could be necessary to interact with a bot. Usually, the interaction starts with the user sending a message with the command "/start" (they do exist other commands like "/menu" or "/help", ...). It can ask to perform some operations before giving access or the invite link. These can include:

- Forwarding one or more links to a certain number of users.
- Joining other groups and/or channels sponsored by the entity we are trying to access.

A bot can ask a user to perform operations even after having joined a group, in order to keep the user inside the group or check that it is not a bot in its turn and gain the right to write in the group. Some of the controls may include:

- Solving an arithmetic operation.
- Answering a simple question.
- Solving a CAPTCHA by clicking on a link.

Usually, we have a few minutes to complete the task. If the checks fail, we are removed from the group.

2.2 Database

In this section, we will discuss the solution to permanently store the data, metadata, and statistics collected by our crawler.

2.2.1 MongoDB

MongoDB is a NoSQL, document-oriented database providing high performance, high availability, and easy scalability. It stores data in flexible, JSON-like BSON (Binary JSON) documents, allowing for dynamic and nested schemas. This nature makes it well-suited for scenarios with evolving data structures and dynamic data, like the one we have been facing. The importance of using a database is essentially given by two reasons:

1. The amount of data we are dealing with could be huge and growing exponentially, so keeping the data we are interested in memory would be impossible.
2. Besides, keeping data in memory is not a good practice because the run of the program could crash unexpectedly due to an unsolved bug or a simply voluntary interruption. Hence, all data collected would be lost.

2.2.2 Database Collections

Our database, called MyCrawler, is composed of 9 different collections, the contents of which are discussed in more detail in the section 2.2.

- Analytics: to compute analytics about crawling activity;
- Bot-collection: to store messages exchanged with bots;
- Done: to store links already processed and keep track of their states;
- Edges: to store the data for creating edges and vertexes of the graph;
- Gathered: to store metadata about the link crawled;
- groups: it contains the actual data collected inside the groups/channels;
- Leave: to keep track of the group to leave;

- TBP: it is the "to be processed" collection. Namely a list of links collected that have to be processed;
- Wait: to keep track of the groups you have requested to join;

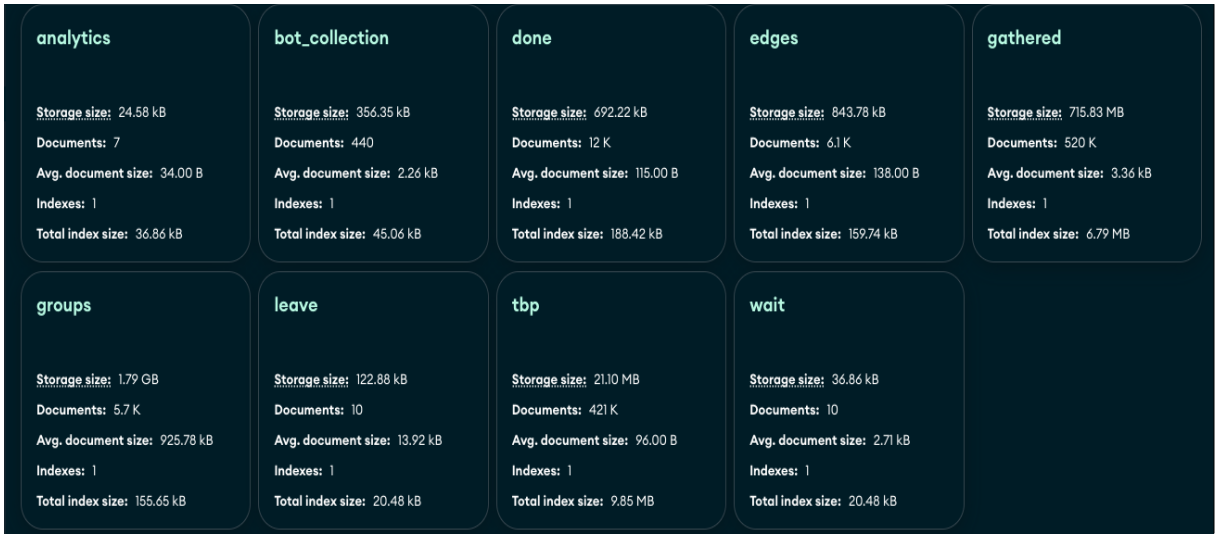

Figure 2.1: Collections view from MongoDB GUI

2.3 Requirements

The goal of this work is to overcome the limitations of previous versions of the crawler. Although they are well programmed, some of the challenges to make the crawling better have been left open. For this reason, we carried out this project: to reach a more stable, scalable, efficient, and performing version.

In this section, we discuss the requirements needed for reaching our target, for implementation details look at the next section 3.2.

2.3.1 Flood Wait Error

The main challenge to face when developing a crawler using Telethon API is the limitation of function calling, to avoid an abuse of the API and a misbehaviour inside the platform. So, Telegram puts limitations on the number of groups/channels you

can join in a specific time interval. To force respect for these limitations, an error is raised when API functions are called many times in a short time. In particular, it is called "Flood Wait Error". Hence, when we are "flooding" the platform with our requests, we will be stopped for an arbitrary amount of time expressed in seconds indicated by the exception raised. The waiting time range can be short (from 70 seconds) or long (even tens of thousands of seconds).

Unfortunately, Telegram does not explain the criteria behind this protection mechanism, so it is impossible to avoid or circumvent. The previous version dealt passively with the generation of these flooding errors, repeatedly calling API functions and stopping only when the platform dictated. However, this strategy is not that efficient because we can get several flood wait errors of thousands of seconds, making the performance of the program unpredictable, as well as keeping the program inactive (consuming resources in vain) for an arbitrary amount of time.

One of the tasks of our version is to avoid long waits imposed by the Flood Wait Error, by controlling the workflow stopping temporarily the code execution after an API functions call.

2.3.2 Bot interaction

In the section 2.1, among the descriptions of Telegram entities, we have seen what is a bot. Interacting with a bot from the Telegram application is quite simple. We can say the same for the automatic interaction using the Telethon API, due to the many different natures of existing bots. Notwithstanding, we decided to add this feature to our crawler, at least for the first and simplest level of interaction, because it is not rare to find links to bots. For the first level, we mean just sending the starting message and waiting for an answer.

2.3.3 Graph

Another modification with respect to the last version is the data used for creating the graph of visited groups. One way to proceed with tracking the graph is to keep track of the connection between links, i.e. visiting a link is a node and all the new links found in that visit are the edges outgoing from that node. Instead of using

links, we decided to use the unique ID of groups/channels/bots. This choice is justified by the fact that using links could lead to useless edges present in the graph since every entity could be pointed by one or more links. Using ID this problem is got over since the id is unique for each entity.

2.3.4 Data collection

Additionally, we determined to collect data of different natures and not only text messages. First, we keep track of the time of each insertion on the database, which can help when we approach data analysis. Then, we store the last 500 messages found in groups and channels in which we joined and messages exchanged with a bot. But, instead of saving just the text, we decided to save the metadata associated too. Examples of metadata are:

- ID
- Timestamp
- Number of views
- Replies
- Edit date
- Post author

In general, the data we gather for each group/channel we join are:

- The group id
- Timestamp of the collection
- Username
- Name
- Link
- Scam flag
- Members list if available
- Messages if any

2.3.5 Statistics

Last but not least, we collect statistics on the activities performed by the crawler. In the "analytics" collections, we keep track of the following parameters:

- The total number of links evaluated.
- The number of bots the crawler has interacted with.
- The number of groups/channels we managed to gather data.
- The amount of the requests sent.
- The amount of the requests accepted.

Chapter 3

Crawler Design and Implementation

3.1 Design

The software design of the crawler plays a determining role in its effectiveness, efficiency, and performance improvement. In this section, we delve into the architecture and design choices behind the development of our crawler. Like in the previous version, a key feature of our implementation is the utilization of a multi-process program, but differently, we decided to employ a master-workers architecture. These two design choices are motivated by the following reasons:

1. The need to enhance scalability and overall performance in terms of links evaluated and groups/channels joined in the time unit.

2. The need to keep the tasks of the master and workers separated and independent from each other, in order to be more resistant to a potential fault.

3.1.1 Multi-process Master-Workers Architecture

The master-workers architecture is a distributed computing paradigm where a single process, called the Master, manages and coordinates the activities of multiple processes, called Workers. This design is suitable for scenarios where tasks can

be divided into independent sub-tasks that can be executed concurrently. In the context of our crawler, the master-workers architecture enables the efficient exploration of the Telegram network by distributing the workload across multiple processes. It is possible to implement it, thanks to the creation of several accounts. Each of them has two credentials parameters for managing applications using Telegram API:

- API ID: it is a unique identifier assigned to your application when you register it on the Telegram API platform. It helps Telegram identify your application and associate it with the correct set of permissions.
- API hash: it is a secret hash key that is paired with your API ID. It serves as a form of authentication, ensuring that only authorized applications can access Telegram's API on behalf of users.

In the next section, we are going to see how to use them for connecting an account to the Telegram servers.

Workers

Worker processes operate concurrently, each one is responsible for fetching and processing a task assigned by the master, waiting once finished, and, then, putting itself in the free processes queue. By parallelizing the crawling tasks, we exploit the computational resources more efficiently, significantly reducing the time required to explore the vast Telegram network. Each worker process is designed to be independent and stateless, allowing for straightforward scalability by adding or removing worker processes dynamically.

Master

The master process acts as the coordinator of the crawling procedure. The main responsibilities include:

- The task assignment.
- The deployment of Telegram URLs to be crawled.
- The managing of results from workers.

- The managing of the links states.
- The interaction with MongoDB in both directions.
- The creation of data structures needed.
- The creation of the workers.

Specifically, there are three kinds of tasks to assign:

1. **JOIN**: the worker has to try to join the link the master has passed as a task parameter.
2. **LEAVE**: the worker should leave the previously-joined entities, and do it if it is the case
3. **WAIT**: the worker has to check if the requests previously sent have been accepted, if yes we are inside and we can start to collect the data and the new links.

The master process assigns each task to an available worker and keeps track of the free processes. It also monitors the progress of the crawling operation and ensures the overall coherence of the data retrieval from the MongoDB.

3.1.2 Design Considerations

The design incorporates fault-tolerance measures to handle potential failures during the crawling activity. If a worker process encounters an error or fails or the account associated is banned, the master process redistributes the failed task to an available worker, ensuring the continuity of the crawling process.

In the next section, we deepen the implementation details of our crawler, showing how the design principles outlined in this chapter are translated into a functional and effective Python program.

3.2 Implementation

3.2.1 Libraries

The crawler implementation relies on the following libraries providing essential services and components:

- **Multiprocessing library**: it is used for implementing the master-workers architecture. One of the most important aspects of our crawler is the improvement given by a well-structured parallelization. Moreover, it provides the fundamental queue data structure for communication between the master and workers.
- **PyMongo**: it is needed for the integration of MongoDB. PyMongo allows for interaction with our database, making the storage and retrieval of gathered data efficient.
- **Telethon library**: used for the interaction with Telegram API, namely for the communication with Telegram servers.

3.2.2 Authentication and Session File

In this context, a session file acts as a continuous identifier for a user's session and it contains the two previously mentioned parameters that must be kept confidential because they uniquely identify an account: API ID and API hash. Thanks to these an account authenticates itself with Telegram API.

Thanks to this file, a user keeps its connection with Telegram servers, hence even though the crawler is stopped, it can resume its activity without manual re-authentication.

Each account creates its session file in a single process run initializing a Telethon client with the provided API parameters:

client = TelegramClient('SessionFile_name', api_id, api_hash)

After the execution of this line of code, the program will request to digitize our phone number and then the OTP code we received in our account (so, we need

to be already logged in to another device). Then the session file will appear in the development environment.

Since it is impossible to create a session file in a multi-processing run, we need to do the above-described operation for every account we are going to use. Once we generate correctly all the session files, we can use them in concurrent execution.

3.2.3 Database installation and configuration

To do this work, we created a virtual machine running Ubuntu Linux 22.04.2 on the server of the University accessible via SSH or remote desktop where we could install the database and run our crawler.

Installation

After the download of the necessary package, we installed the MongoDB instance with the following command:

```
sudo apt install mongodb
```

Then, after the completion of the installation, we started the instance of the MongoDB with this command:

```
sudo systemctl start mongodb
```

Configuration

Once correctly installed and running, we modified the configuration file for:

1. Adjusting the **'bindIp'** to specify which IP addresses MongoDB will listen on. In our case, we made it reachable from anywhere, so we modified the value to **'0.0.0.0'** and **'port'** to run MongoDB on port **27015** and eventually restarted the instance.

2. Adding the security section on the file to enable authentication with username and password (default is not the best solution but for our work, it is enough).

Then, creating the user and downloading the MongoDB Compass (the graphical user interface) were the last steps to start using our database, which we connect with the following string:

```
mongodb://myUserAdmin:mypassword@myip:27015/?authMechanism=DEFAULT
```

3.2.4 Database organization

In the section 2.2 we illustrated MongoDB, its installation, and configuration. Now we are going to give details on the data collected and the schema of each collection, as well as some examples of PyMongo commands to insert, update, or delete an entry in our database collections.

- **Analytics collection, 3.1:** It includes some statistics computed during the execution of the crawler, each entry is made of two fields: id and counter. Both of them are `Int32`. See subsection 2.3.5 for details.

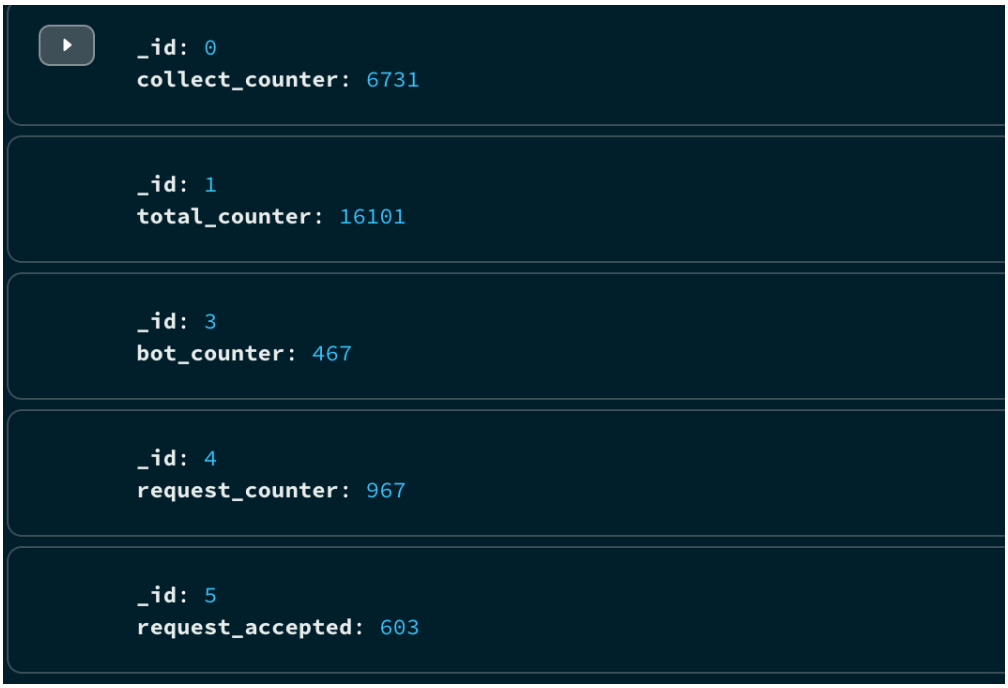

Figure 3.1: Analytics Collection

- **Bot collection. 3.2:** It includes the messages exchanged with the bots, each element is made of:
  - *"process_id"*: It is the id of the process or account that has interacted with the bot.
  - *"id"*: it is a *string* containing the entity unique identifier on Telegram.
  - *"link_hash"*: it is a *string* containing the link processed.

- *"messages"*: it is an *Array* of message object, containing the conversation the account had with the bot. Each message contains the text and all the metadata available mentioned at 2.3.4
- *"time"*: it is the time of the insertion in the database.

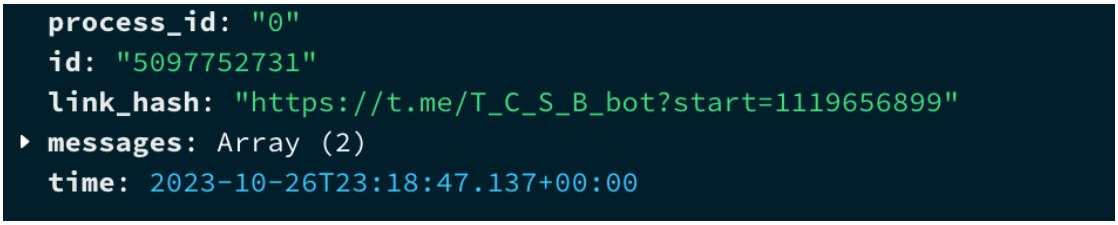

Figure 3.2: Bot Entry Example

- **Done collection, 3.3:** it keeps track of the links state, each element consists of:
  - *"process_id"*: it is the id of the process or account that has processed the link.
  - *"state"*: it is a *string* containing the processing state of the link. We have defined the following states:
    1. "to be processed": when a link has still to be processed and it is in the TBP list;
    2. "processing": when an account is processing the link and, maybe, collecting data from that group/channel;
    3. "inside": when an account managed to join the link and is inside waiting to leave;
    4. "join failed": when the attempt to join the link has gone wrong;
    5. "done": when an account completes correctly the processing of the link (with successful leave);
    6. "waiting": when an account has sent a request for accessing that link;
    7. "leave failed": when an account did not manage to leave the link.
  - *"link_hash"*: it is a *string* containing the link processed.
  - *"time"*: it is the time of the last state change.

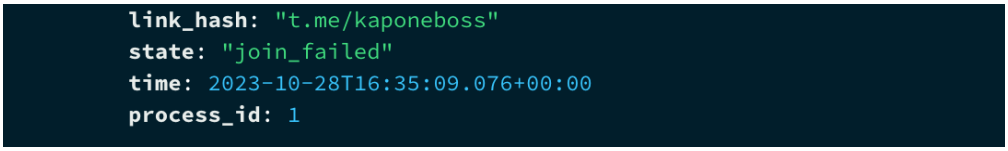

Figure 3.3: Done Entry Example

- **Edges collection, 3.4:** it keeps the information for drawing the crawling graph, each entry is made of:
  - *"dest"*: it is an *Int32* containing the ID of the entity joined.
  - *"sources"*: it is an array of *Int32* containing the IDs of all groups/channels where we found the link of the destination entity.

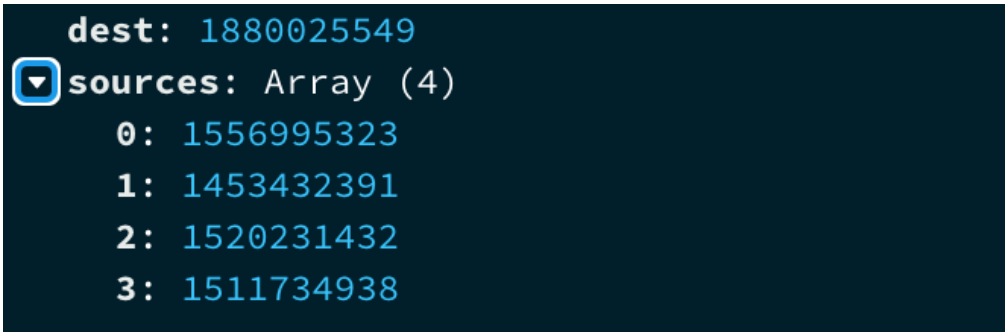

Figure 3.4: Edges Entry Example

- **Gathered collection, 3.5:** it keeps track of each message in which we found a link, each entry encompasses:
  - *"link_hash"*: it is a *String* indicating the link found and gathered.
  - *"message"*: it is the object message, its text field is where we found the link.
  - *"group_id"*: it is a *Int32* indicating the id of the group or channel where we found the link. This information helps us to build the crawling graph.
  - *"group_name"*: it is a *String* indicating the name of the group or channel where we found the link.
  - *"date"*: it is indicate when we found this link.

Figure 3.5: Gathered Entry Example

- **Groups collection, 3.6** the heaviest collection of our db. It contains the data actually collected once we are inside a group. The fields of this kind of entry have been specified in subsection 2.3.4

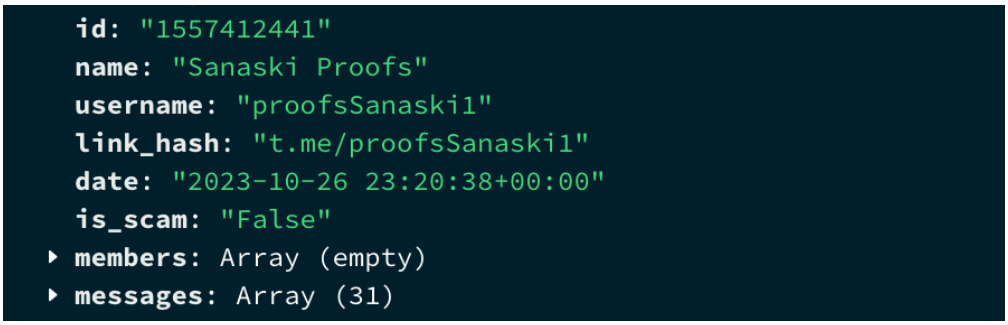

Figure 3.6: Groups Entry Example

- **Leave collection, 3.7:** It contains as many queues as the number of accounts configured to crawl. It keeps track of the groups or channels joined that we have to leave after a certain amount of time. We set a threshold of 24 hours. It means that we stay 24 hours inside a group or channel unless the code raises an error during a leave task. Each element of the queue contains:
  - *"link_hash"*: it is the link of the group to leave.
  - *"id"*: it is the id of the group to leave
  - *"time joined"*: it is the join time.

- **TBP collection, 3.8:** It is the list of links that have to be processed by the crawler. When a link is assigned to a worker, we update the time, the

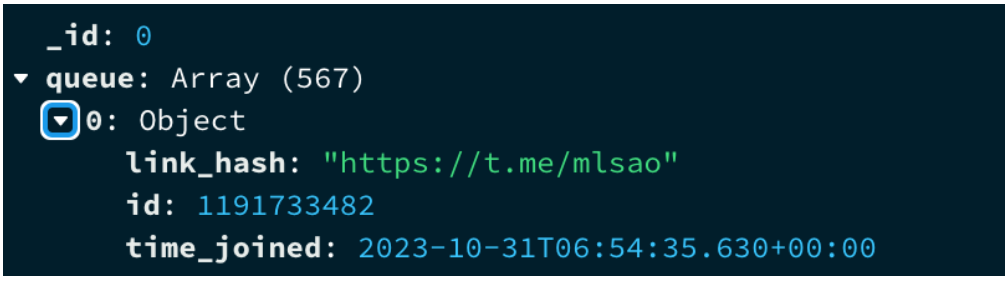

Figure 3.7: Leave Entry Example

process_id with the id of the worker, and we remove the entry from this collection and insert it in the done collection updating its state.

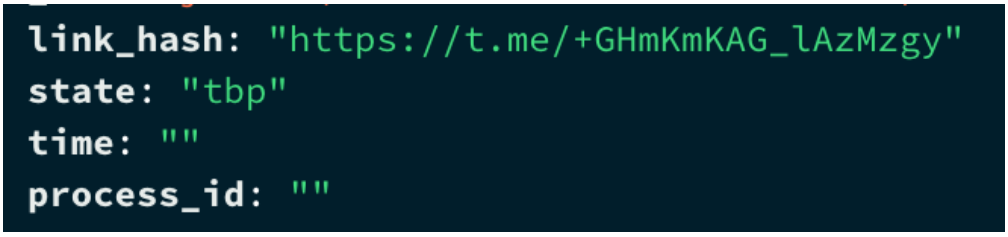

Figure 3.8: TBP Entry Example

- **Wait collection, 3.9:** It contains as many queues as the number of accounts configured to crawl. It keeps track of the groups or channels we sent requests to join. We set a threshold of 24 hours. It means that we wait 24 hours to check if the account involved has been accepted or not. Each element of the queue includes:
  - *"link_hash"*: it is the link of the group/channel we sent the request.
  - *"time request"*: it is the request time.

Examples of PyMongo command

groups_collection.insert_one(result['data'])
done_coll.update_one({"link_hash": result['link']}, {"$set": {"state": "inside"}})
tbp_collection.insert_many(result['new_links'])
analytics_collection.update_one({"_id": 0}, {"$inc": {"collect_counter": 1}})

Figure 3.9: Wait Entry Example

3.2.5 Workers implementation

A worker is a process acting in a passing way. Its only responsibility is waiting for an assignment from the master. It never interacts with the database, it only uses the Telethon API functions for executing tasks assigned and returning the results obtained. The core of our worker implementation is the function *crawler_worker*. After the creation of the workers, the master passes to each process all the data structures it created: Telegram client instance, *"task_queue"*, *"result_queue"*, *"process_queue"*, and the process identifier.

The *crawler_worker*, which is in charge of building the result data structure, consists of an infinite **while** loop: after the initial operation of getting a task from the specific *"task_queue"* (each worker has its own task queue passed from the master) the function logic can be divided into three branches *(LEAVE, CHECK WAIT, JOIN)*, as shown in the figure 3.10. The task element extracted is composed of two fields:

- *"name"*: it contains one of the three task to perform;
- *"data"*: it is an array containing the data (links or IDs) needed for the execution, whose content and length depend on the assignment to carry out.

JOIN

The data structure of this case contains just one link to evaluate. In this task, the core function is *evaluate_link(client, link, result)* which wraps many other functions needed for satisfying the requirements we defined in the section 2.3.
Let's deep dive into this function.
In turn, the *evaluate_link* is logically divided into three branches; the first two

aim to recognize if the link belongs to a bot, and the third is the one used for the actual evaluation of a generic link.
Therefore, there are two ways to understand is a bot or not:

1. **Looking for a parameter in the link (preceded by the symbol "?").**
   If there is no parameter in the link, move on to step 2. If yes, to be sure we are dealing with a bot, we can leverage the call to the API function ***client.get_entity(link)***, which lets us have some meta-information about the link before joining it. If the type of the entity returned by this function is *User* and its *bot flag* is *TRUE*, then we can set the result code to *"BOT_RESULT"*. If one of the last two conditions is not true, we can return a result with *"JOIN_FAILED"* code.

2. **Check if the link terminates with "bot", "BOT", "Bot", ...**
   If yes, set the result code to *"BOT_RESULT"*. Otherwise, jump to the last branch of the function.

If one of these two methods tells us the link under evaluation is a bot, we call the *bot_interaction* function. We implemented it to interact with a bot, just for the first level of conversation. We send a simple message "/start" plus a possible parameter and wait for the answer. When we receive it, we look for the presence of new links and save them and the entire conversation in the result structure.

The third branch of the function includes an attempt to join the link under evaluation without knowing anything about that. We try to join the link using the function *join_group_by_hash*. It calls the API function ***ImportChatInviteRequest(link)***, which is used to import a chat from an invite link. If we are not dealing with an invite link or it is no longer valid, the function raises an ***InviteHashExpiredError***, so we try with another function *join_group_public_by_link*, which in turn calls the already known API function ***JoinChannelRequest***. If neither of the two functions manages to access a group/channel or send a request (*"REQUEST_SENT result"*), an error is raised and the result code for this task is set to *"JOIN_FAILED"*. These functions can generate *FloodWaitError*, so we set again a waiting for about 2 minutes to prevent a huge stop by the API server.

Instead, if one of the two functions has successfully joined the entity it starts

the actual phase of data crawling and the result of this task is *"JOIN_SUCCESS"*. We delegated this assignment to the *process_link* function, which in turn calls the *gather_links* and the *collect_data*. The goal of the *gather_links* is to look for new links to crawl in all the messages present in that group/channel. It uses a regular expression to filter only the link pointing to Telegram. While the goal of the *collect_data* is to collect all the information we can about that entity: id, name, username, members, messages...

LEAVE

When this task is assigned, a worker will find in the data field one or more elements coming from its *leave_queue* containing the IDs to leave. We pass this value to the function *leave_group* which in turn wraps the call to the API function ***client.delete_dialog(id)***.

The operation could be successful or not, due to different motivations, e.g.:

- We could have been expelled from the group/channel, so we are no longer inside.
- The group or channel could have been already deleted.

So, the result code will be *"LEAVE_SUCCESS"*, otherwise *"LEAVE_FAILED"*. We inserted a short wait (5-10 s) between two consecutive leaves API calls, just because if the groups are managed by the same admins we do not raise suspects of being an automated application and not a human.

CHECK WAIT

Like the previous task, in the data field will be one or more elements coming from the specific *wait_queue* containing the link whose request status we need to check. The execution of this task harnesses the call to the API function for each link:
client(JoinChannelRequest("link_hash")). This one needs the link passed as a parameter and if it raises the error *UserAlreadyParticipantError*, it means that the group/channel admin accepted our request (result code: *REQUEST_ACCEPTED*). Hence, we are ready to collect links and data, but the process of this step will be

described in the next paragraph, being equivalent to a successful join. If another error is raised the result code is *STILL_WAITING*. We have to underline that after each call to that function, we set waiting for about 2 minutes, to avoid a huge stop imposed by the *FloodWaitError*.

Finally, at the end of each of the three tasks (including any waiting), each worker puts the result it built during the execution in the results queue and its identifier in the queue of free workers, becoming ready for receiving the next task.

Figure 3.10: Worker flow chart

3.2.6 Master implementation

The Master has the duty of coordinating and ensuring the correctness of the crawling activity. It has to prepare the environment for the crawler by creating the worker

processes and the data structures (task queues, processes queue, leave queues, wait queues, results queue) needed for communication between itself and the workers. As the figure 3.11 shows, the master consists of a ***while*** cycle that keeps going until there is an element in the "to be processed" collection. It comprehends two macro functions:

- Tasks assignments: the part of the chart with the three different colors.
- Results retrieval: it is the block of the code downstream of the three branches.

Tasks Assignments

As mentioned in the section 3.1.1, the master can assign 3 kinds of tasks, with the following priority: **LEAVE, CHECK WAIT, JOIN**. For each task, the master passes to the worker the name of the task and the necessary data for executing it.

First of all, the master checks if there are free workers available. If not, it is in charge of emptying the results queue out, saving the data obtained in the database. The free processes are stored in a queue: a shared data structure in which the master gets the last element added (an integer representing the process id) and the workers put its id when it has completed a task assigned.
Once the master has found an element in the free process queue, it proceeds with the task assignment:

- If the selected worker has groups to leave, then the master checks if the worker stayed inside those groups for a time that exceeds the threshold we set. If there is at least one entity for which the limit has been exceeded, then the master assigns the **LEAVE** task to that worker.
- Instead, if there are no groups ready to leave, then the master checks whether there are requests to monitor sent previously by that worker, following the same criteria described above. If it has elapsed enough time for one or more requests, the master assigns the **CHECK WAIT** task to the worker.
- Finally, if the master can not assign the two previous tasks, then it assigns the **JOIN** task to the worker. After the assignment the master takes care of keeping updated the "to_be_processed" list on the database, deleting all the

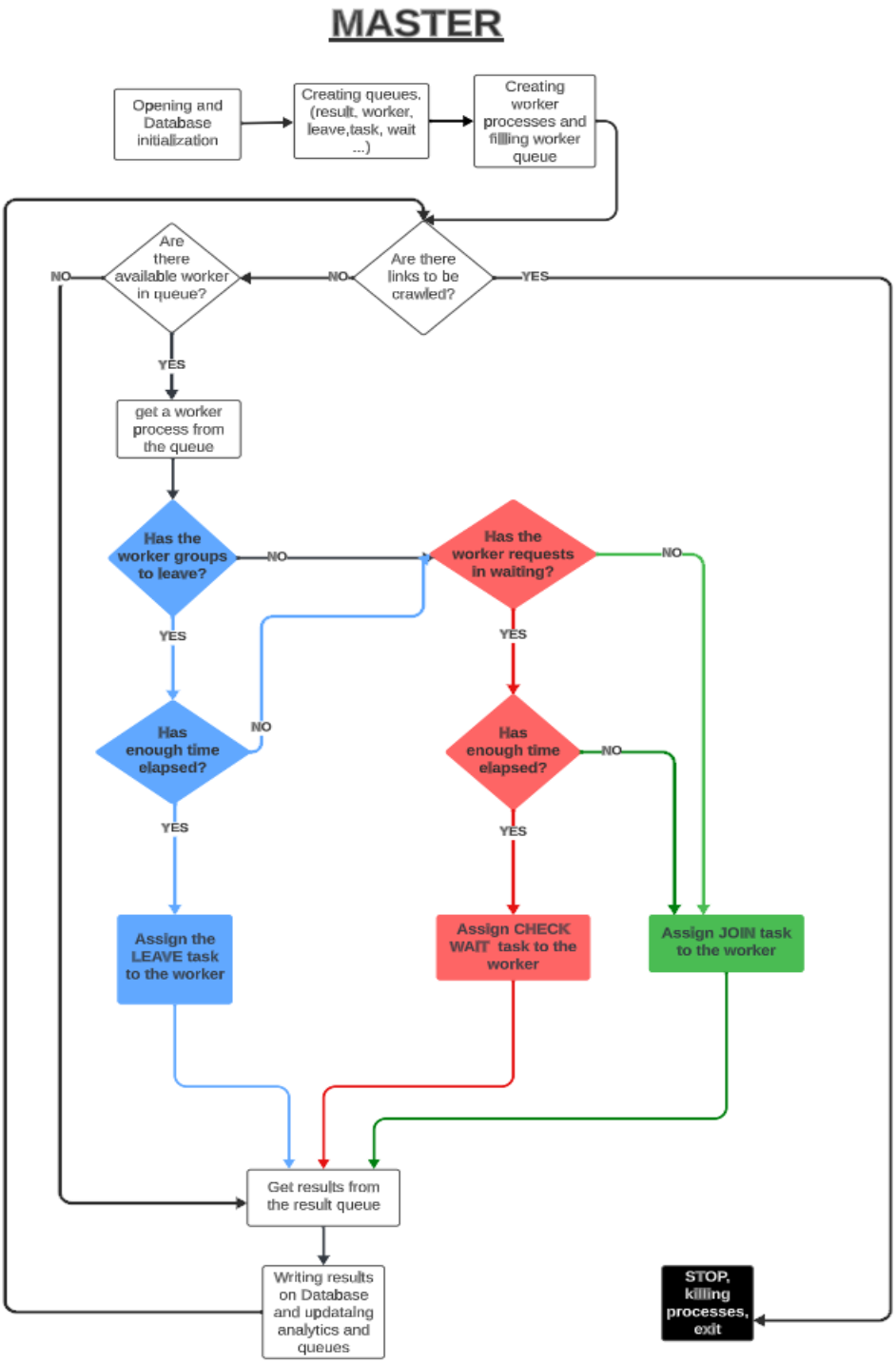

Figure 3.11: Master flow chart

possible occurrences of the link passed as a task data parameter to the worker. Once we assign this task, we update the *"total_counter"*.

Results retrieval

The master carries out this function when there are no free workers available and at the end of each task assigned. The *"result_queue"* contains all the results produced by the workers as they proceed with their tasks. We implemented inside the master a *"get_result"* function, whose purpose is:

- Cycling on the *"result_queue"* getting one result at a time until the queue is empty.
- Read the *code* of the extracted element and figure out what actions to perform.
- Update the appropriate database collections if needed.

We can read from the *code* field of the result data structure 8 different self-explanatory *String* returned from the workers:

1. *JOIN SUCCESS*: in this case, the link state goes from *processing* to *inside*. The result contains to insert in the database: the list of new links found and the actual data collected inside the entity joined. Moreover, we update the *"collect_counter"* and the *"leave_queue"* belonging to the specific worker.
2. *JOIN FAILED*: this is the case in which the tried join failed, so the only thing to do is to change the link state to *"join_failed"*.
3. *BOT RESULT*: this is the result coming from a bot conversation. We insert in the database the same elements of case 1 and the link state changing is the same too. In this case, the data collected are the bot conversation, so we update the *"bot_counter"* too.
4. *REQUEST SENT*: in this case, the link state changes to *waiting*. We update the *"request_counter"* and the wait queue belonging to the specific worker.
5. *LEAVE SUCCESS*: when we leave an entity previously joined, we have to change the link state to *done* and remove the element from the *"leave_queue"*.

6. *LEAVE FAILED*: in this case the state changes to *"leave_failed"*. We need to update the *"time_joined"* field in the queue element to let a new attempt leave after the threshold has expired.

7. *REQUEST ACCEPTED*: this case is totally equivalent to case 1 unless we need to update the *"request_accepted"* counter too.

8. *STILL WAITING*: this last case requires just to update the *"time_request"* field in the element of the *"wait_queue"* involved.

In addition to the above-listed database updates, in cases 1, 3, and 7 (namely, when we are managed to access an entity) we update also the *"edges_collection"*.

Chapter 4

Analysis and results

In this chapter, we are going to discuss the results obtained during the crawling activity. We are going to analyze the performance of our software under different circumstances. In particular, we tried our crawler with 5, 2, and 1 processes.

Before to start, it is due to make some clarification on terminology:

- Link evaluated: it is a group, channel, or bot link we tried to join.
- Link joined (or collected): it is a link, in which we joined successfully because the link is valid and we are able to collect the data inside it.

The crawling test, using the latest version of the software, started at the end of October 2023. We used the seed [10] just the first time. In the following days, we restarted the crawling from the point where it left, after an interruption due to several possible reasons (banning, fatal crash, database fault). As we said, our crawler can be manually interrupted at any time, and then re-start from where it is left.

Overall, the crawling lasted about 25 days. Here are some statistics about it:

- 24,083 links evaluated from all processes.
- 8,209 links from which data have been collected (including 566 bots).
- 1,291 requests sent, 902 accepted.
- 646,846 links gathered, started from a seed of about 380 links.

4.1 Performance analysis

4.1.1 5 Workers

Since we designed our crawler to be used with a high number of workers (Telegram clients), we created 4 brand new accounts for this purpose and used another one created years ago. At the full rate, the overall work made by the different processes can reach more than 140 links evaluated in an hour (about 28 links/worker per hour). We managed to collect to join and collect data in about 20 links/worker per hour.

With respect to the previous versions, it's a meaningful improvement, but unfortunately, this is the maximum rate achievable by our solution, because the crawling performance is slowed due to platform limitations such as unpredictable Flood Wait Error or ban. Furthermore, the rate is even slowed down by two design choices:

- Self-stop after API call to avoid too long Flood Wait Error
- Other two tasks to perform (LEAVE and CHECK WAIT)

In particular, during this test, we noticed that the new accounts suffered several Flood Wait Errors of order of magnitude until 40,000 seconds, despite the self-stop mechanism we implemented. Then, the performance dramatically dropped due to the progressive ban of the 4 new clients.

So, in order to do crawling that is not heavily hindered by the platform, it is necessary to use accounts that have not just been created but have already done some activity in the previous months or years within Telegram.

4.1.2 2 Workers

We decided to continue the crawling with 2 accounts existing for different years to avoid other blocks from the platform. The last crawling campaign lasted 18 days, without interruptions.

As shown in Figure 4.1, the total number of links evaluated per hour can reach a total of 60 units for the two workers. This metric, with all the mentioned

considerations related to performance limitations, is compatible with the previous scenario, with 28/30 links per worker per hour evaluated.

We can recognize a pattern in the distribution link evaluation. Every 24 hours more or less, there is a regular drop in the performance. Because we have set the temporal threshold for the other two tasks at 24 hours. It means that if there are groups to leave or requests to check for at least 24 hours, the worker will execute that task.

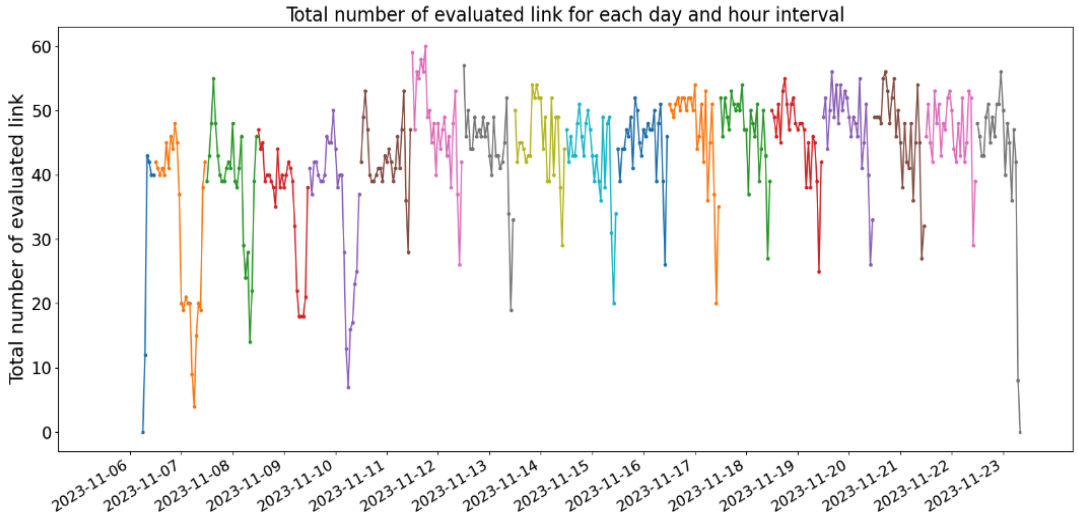

Figure 4.1: Overall representation of the total number of links evaluated during crawling with 2 workers. Each color represents a day. Each point in the chart represents a value in an hour interval. 24 points for each day, unless the first and the last one because the crawler started and ended with the day in progress.

Another interesting metric to show is the total number of entities joined in the hour and day intervals. We recognize the same kind of pattern as before. From Figure 4.2, we can see a peak at 40 entities globally joined per hour (we can reach 20 entities joined per hour for each worker). Of course, this metric strongly depends on the number of links valid among the ones evaluated, which is unpredictable.

Moreover, the graph can be divided into two parts. The first one is from the starting day to 2023-11-11, and the second one is from 2023-11-12 to the end of the activity.

After 6 days, one of the two workers disconnected for unknown reasons (probably because of a problem with the session file), but we decided to leave the crawler

running to see the behavior of our software and to make the right decision in bug fixing. Surely it was not banned, because we still had access to the account. The only problem is that a link under evaluation from a worker not connected can not be joined, because it is impossible to call the API function. Hence, the task to evaluate the link is assigned correctly (see Figure4.1), but at the moment of trying to access it, the bug comes up. The main issue to fix is that if a worker is disconnected for any reason, it should not receive a link to evaluate. So, we can see how after 2023-11-11 the overall performance of groups joined gets halved, since only one worker kept executing correctly.

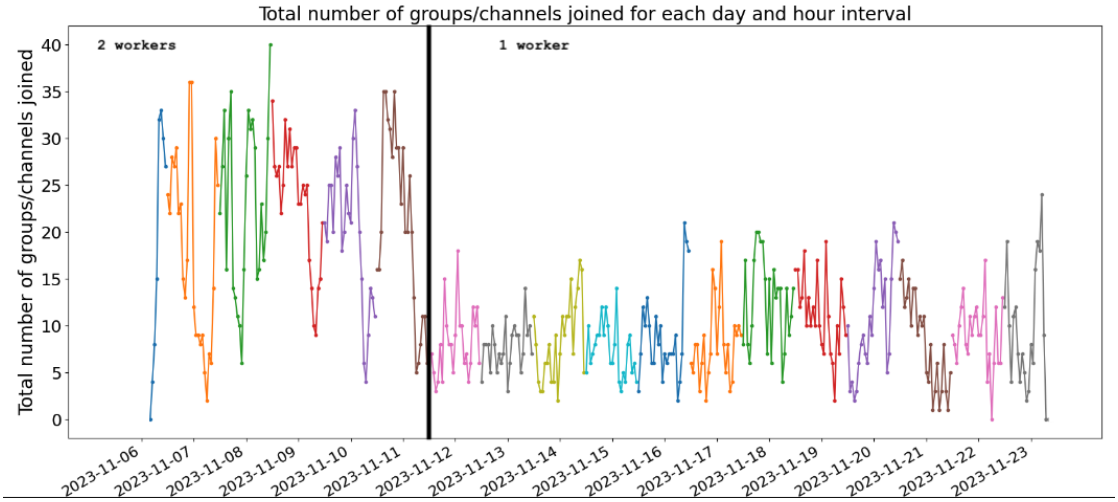

Figure 4.2: Overall representation of the total number of links joined during crawling with 2 workers. Each color represents a day. Each point in the chart represents a value in an hour interval. 24 points for each day, unless the first and the last one because the crawler started and ended with the day in progress. Since 12-11-2023, the crawling kept working with just 1 worker due to a connection problem with the other one. So performance gets halved.

4.2 Flood Wait Error (FWE) analysis

In this section, we are going to make some considerations about Flood Wait Error imposed by the platform due to the abuse of API functions calling.

To the best of our knowledge, a Flood Wait Error is something impossible to avoid and to predict, due to the restrictions Telegram puts in place to avoid abuse of API calling in its platform. Indeed, there is no public information on which criteria are behind the decision of the time to wait after a FWE. In our crawler, we dealt with this problem, by setting a waiting time of inactivity after an API call, usually the *JoinChannelRequest* or *ImportChatInviteRequest*, the two functions used in the JOIN and CHECK WAIT tasks.

For example, in Figure 4.3, we can see that after 6 days of crawling a worker received 165 FWEs, between 50 and 700 seconds. We can notice also that on the first day of crawling the number of FWE is sensibly smaller than the other days. In this way, it is possible to achieve some kind of predictability on the performance of our crawler, since we expect that we will surely receive Flood Wait Errors, but these are expected to be in the range represented.

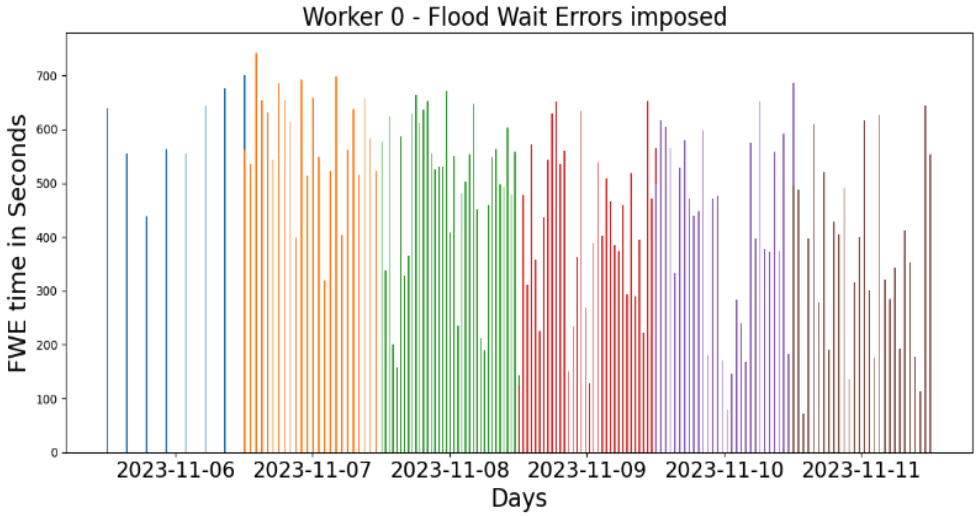

Figure 4.3: Representation of Flood Waits Error received each day by the worker 0. Each color represents a day. This error becomes more frequent as the crawling continues. The positive note is that in 6 days we have always received a waiting time between 50-700 seconds, which is acceptable. The darker the color becomes, the closer in time the errors reported are.

However, it is not guaranteed that self-stopping after an API call is enough to have an (almost) predictable waiting time range. Indeed, we plotted the Cumulative Distribution Function for the waiting values of the overall 1,262 FWEs received in 18 days of crawling. As shown in the figure 4.4, despite the self-stop mechanism enacted there are three FWEs two orders of magnitude larger than usual. It's not common, because the vast majority of FWEs are within the range we expect (50-700 seconds), but it happened 3 times to have 10,000, 20,000, and 30,000 seconds of wait. The positive side is that these FWEs have not been followed by a Telegram ban, like in the previous case (when we used 4 new accounts).

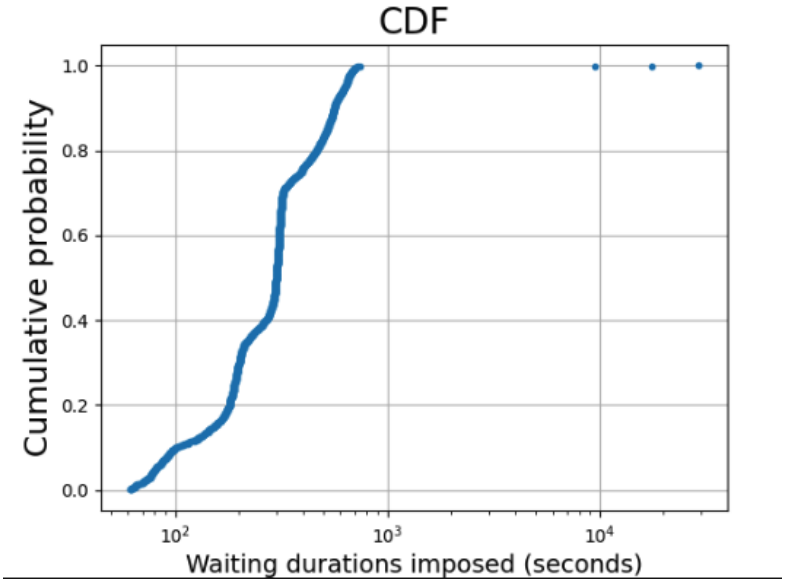

Figure 4.4: Representation of Cumulative Distribution Function of all the Flood Wait Errors received by all the workers. Log scale on the x-axis.

4.3 Graph Analysis

In this section, we are going to make some considerations about the graph created through the exploration of the Telegram environment. The graph created by the crawling activity illustrates the web of connections and relationships within a network of interconnected information, in our case Telegram. We decided to generate a directed and weighted graph in which the nodes are groups/channels/bots

and the weight of each directed edge from "i" to "j" represents how many different links collected in the entity "i" point to entity "j".

The graph provides a visual representation of the structure and complexity of the crawled data, it can reveal patterns, clusters, and central nodes, facilitating the understanding of the relationships within that digital ecosystem.

Our graph consists of 8,479 nodes and 66,935 edges. In Figure 4.5, looking at the coloring gradient, which indicates the temporal exploration ranking, we can assume that snowball sampling that started from the white-colored seeds was concentrated in the exploration of a very interconnected part of the network, of which many seeds were part. The dark coloring of some nodes in the peripheral areas suggests to us that, though slowly, exploration is finally coming out of this bubble. Future analysis of the graph, after a larger data collection, may then reveal other separate substructures.

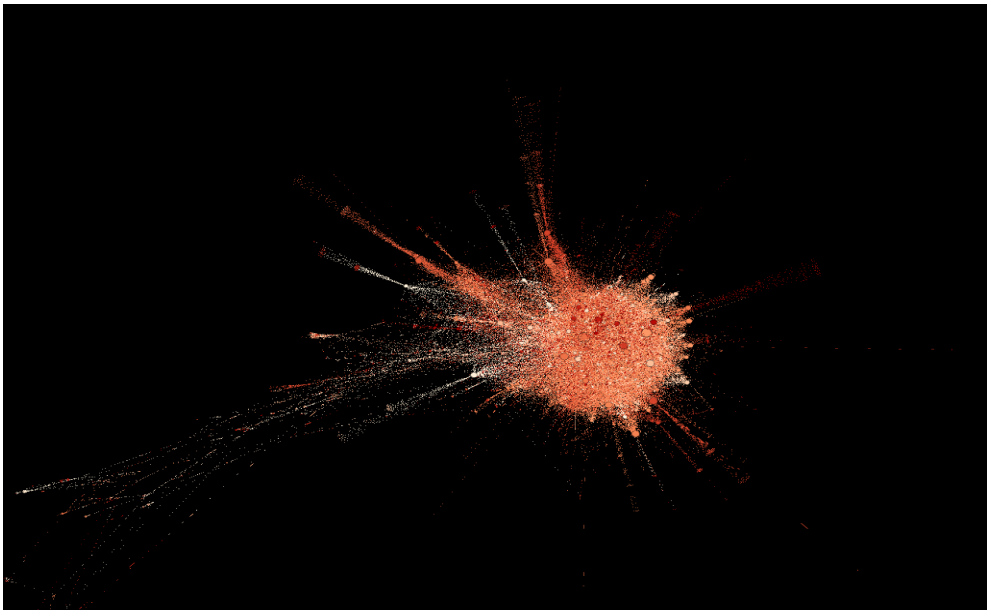

Figure 4.5: Crawling graph. A darker color means that the node has been visited later, while the size of the node indicates the in-degree.

In graph theory, the term "degree" refers to the number of edges incident to a node. In a directed graph, the in-degree represents the number of incoming edges. The out-degree represents the number of outgoing edges. The degree of a node is a fundamental concept in graph theory and is often used to analyze and characterize the structure of graphs. Vertices with higher degrees may be considered more central or important in certain contexts, and the distribution of degrees across all vertices provides insights into the connectivity and complexity of the graph. So, the graph we created with crawling has an average degree equal to 15.79. As in Figure 4.6, these degree distributions turn out to be very heterogeneous, a property that distinguishes most of the real graphs.

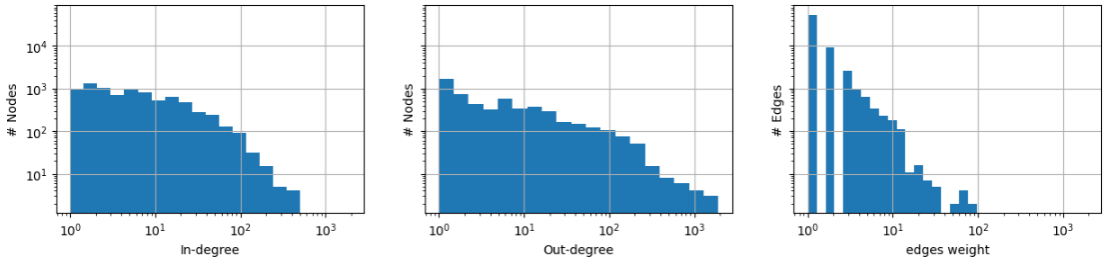

Figure 4.6: Graph degree metrics

Another example of an investigation done is the identification of the substructures present in the graph, called community. *Community division* is a term often used in the context of community detection in networks. A community in a graph represents a group of vertices that are densely connected to each other but less connected to vertices in other parts of the graph. Community detection algorithms aim to identify these groups or communities within a network. To search for community, we made the graph undirected and used the Leiden algorithm [11]. Figure 4.7 is an example of community division in our case.

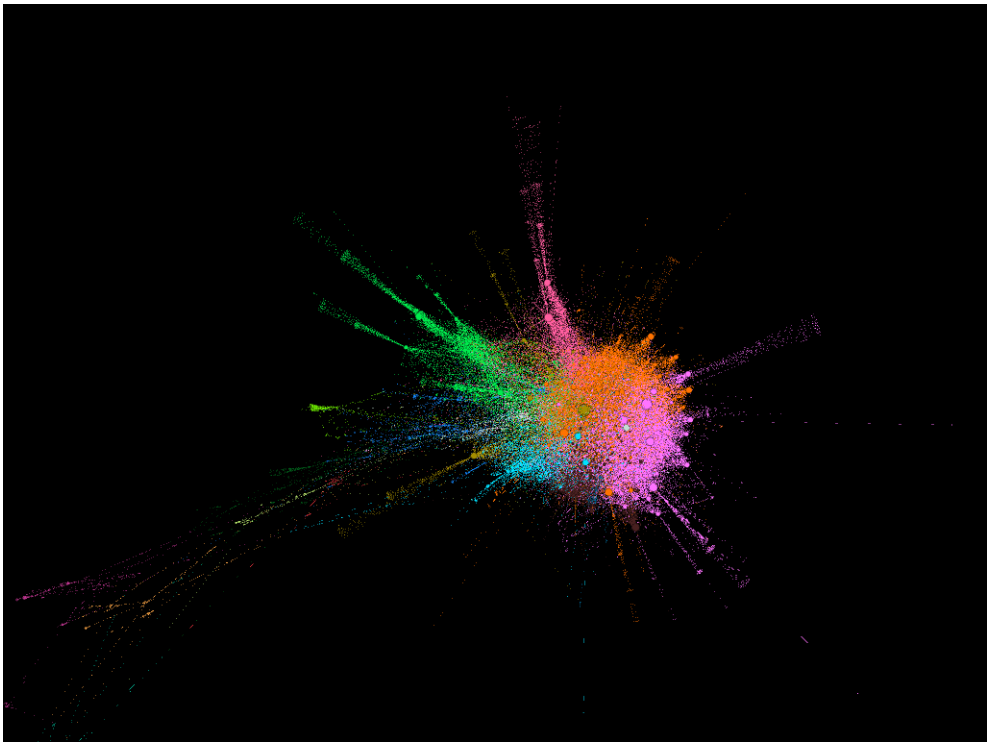

Figure 4.7: Graph community division. Each color represents a different community.

Chapter 5

Conclusion and future work

In the pursuit of understanding and harnessing the vast landscape of Telegram, this thesis focused on the development of a more advanced crawler than the previous works. The objective was clear: design a tool capable of navigating the Telegram network while upholding privacy, data collection practices, and principles of responsible crawling. The work began with a deep dive into the significance of Telegram as a leading instant messaging platform, setting the stage for the challenges and opportunities that lay ahead.

The core of this thesis lies in the design and implementation of a multi-process crawler, adopting a master-workers architecture. This architectural choice was pivotal, as it facilitated scalability, responsiveness, and overall efficiency in the retrieval of data from Telegram entities and interaction with MongoDB. Each worker, operating independently, contributed to the parallel execution of tasks, allowing the crawler to adapt dynamically to the ever-evolving Telegram ecosystem.

Implementation details, from task distribution mechanisms to fault tolerance considerations, have been thought to ensure a seamless and robust crawling experience. Indeed, one of the peculiarities of our crawler is the possibility of interrupting crawling and starting it again from the state of interruption.

As we conclude this phase of the Telegram crawling, several possibilities for future research and development emerge. These avenues aim to enhance the capabilities of our current crawler.

One promising area for future development lies in expanding the crawler's capabilities to have better bot interaction. Creating software to have a full conversation with a bot would require a huge work, especially due to the number of different kinds of bots that exist. Enabling the crawler to engage in more nuanced conversations and dynamically adapt to different bot functionalities would unlock new dimensions of data collection and analysis.

Another task addressed in our work, but not sufficiently in-depth, is the collection of external links found in Telegram messages. This augmentation would provide a holistic understanding of content sharing and user interactions, extending the reach of the crawler beyond the Telegram ecosystem.

Moreover, the integration of machine learning algorithms for data analysis stands as a significant next step in the evolution of our crawler. By infusing intelligence into the system, we can empower the crawler to adapt its strategies based on observed patterns, making it not just reactive but proactive in optimizing data extraction efficiency.

Last, but not least, the code can be enriched with improvements related to different aspects:

- Potential bug, logic error, or inaccuracies missed by the programmer.
- Adding a control using language detection if we want to limit the exploration of certain kinds of clusters.
- better error handling, especially when a client (worker) results disconnected for an unknown reason.
- Better code modularity.
- Improved log file outputs. Enhancing the output of log files emerges as a critical aspect of refining the crawler's transparency and usability. Detailed information about the crawling process, error handling, and task execution, when presented in a more user-friendly format, can greatly aid in monitoring and troubleshooting.

In conclusion, this work has laid the foundation for a powerful Telegram crawler, ready to meet the challenges of the dynamic platform landscape. The master-workers architectures position our crawler as a valuable tool in the realm of web

data mining. Looking to the future, the possibilities are vast. The potential for our crawler to evolve into a more intelligent and adaptive tool that comprehensively explores Telegram and beyond is challenging. This master thesis is not just an end point, but a springboard to continue to explore and innovate in the world of instant messaging data mining.

Bibliography

[1] Edoardo Gabrielli. *telegram-groups-crawler*. 2022. URL: https://github.com/edogab33/telegram-groups-crawler (cit. on pp. i, 5).

[2] Armando Chimirri. «Individuazione di data breach su Telegram via crawler e machine learning». M. Eng. thesis. Turin, Italy: Politecnico di Torino, Apr. 2023 (cit. on pp. i, 5).

[3] *Telegram – a new era of messaging — telegram.org*. https://telegram.org/. [Accessed 27-11-2023] (cit. on p. 2).

[4] Daniel Ruby. *90+ Telegram Statistics In 2023*. URL: https://www.demandsage.com/telegram-statistics/ (cit. on p. 3).

[5] Arash Dargahi Nobari, Negar Reshadatmand, and Mahmood Neshati. «Analysis of Telegram, an instant messaging service». In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2017, pp. 2035–2038 (cit. on p. 3).

[6] Ali Hashemi and Mohammad Ali Zare Chahooki. «Telegram group quality measurement by user behavior analysis». In: *Social Network Analysis and Mining* 9 (2019), pp. 1–12 (cit. on p. 3).

[7] Tuja Khaund, Muhammad Nihal Hussain, Mainuddin Shaik, and Nitin Agarwal. «Telegram: Data collection, opportunities and challenges». In: *Annual International Conference on Information Management and Big Data*. Springer. 2020, pp. 513–526 (cit. on p. 3).

[8] Massimo La Morgia, Alessandro Mei, Alberto Maria Mongardini, and Jie Wu. «It's a Trap! Detection and Analysis of Fake Channels on Telegram». In: *2023 IEEE International Conference on Web Services (ICWS)*. IEEE. 2023, pp. 97–104 (cit. on p. 3).

[9] Telethon. *Telethon API*. URL: `https://tl.telethon.dev/` (cit. on p. 5).

[10] fastfire. *deepdarkCTI*. URL: `https://github.com/fastfire/deepdarkCTI/blob/main/telegram.md` (cit. on p. 33).

[11] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. «From Louvain to Leiden: guaranteeing well-connected communities». In: *Scientific reports* 9.1 (2019), p. 5233 (cit. on p. 41).